

axiomTM



The 30 Year Horizon

<i>Manuel Bronstein</i>	<i>William Burge</i>	<i>Timothy Daly</i>
<i>James Davenport</i>	<i>Michael Dewar</i>	<i>Martin Dunstan</i>
<i>Albrecht Fortenbacher</i>	<i>Patrizia Gianni</i>	<i>Johannes Grabmeier</i>
<i>Jocelyn Guidry</i>	<i>Richard Jenks</i>	<i>Larry Lambe</i>
<i>Michael Monagan</i>	<i>Scott Morrison</i>	<i>William Sit</i>
<i>Jonathan Steinbach</i>	<i>Robert Sutor</i>	<i>Barry Trager</i>
<i>Stephen Watt</i>	<i>Jim Wen</i>	<i>Clifton Williamson</i>

Volume 10: Axiom Algebra: Domains

Portions Copyright (c) 2005 Timothy Daly

The Blue Bayou image Copyright (c) 2004 Jocelyn Guidry

Portions Copyright (c) 2004 Martin Dunstan

Portions Copyright (c) 2007 Alfredo Portes

Portions Copyright (c) 2007 Arthur Ralfs

Portions Copyright (c) 2005 Timothy Daly

Portions Copyright (c) 1991-2002,
The Numerical ALgorithms Group Ltd.
All rights reserved.

This book and the Axiom software is licensed as follows:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of The Numerical ALgorithms Group Ltd. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Inclusion of names in the list of credits is based on historical information and is as accurate as possible. Inclusion of names does not in any way imply an endorsement but represents historical influence on Axiom development.

Michael Albaugh	Cyril Alberga	Roy Adler
Christian Aistleitner	Richard Anderson	George Andrews
S.J. Atkins	Henry Baker	Martin Baker
Stephen Balzac	Yurij Baransky	David R. Barton
Gerald Baumgartner	Gilbert Baumslag	Michael Becker
Nelson H. F. Beebe	Jay Belanger	David Bindel
Fred Blair	Vladimir Bondarenko	Mark Botch
Raoul Bourquin	Alexandre Bouyer	Karen Braman
Peter A. Broadbery	Martin Brock	Manuel Bronstein
Stephen Buchwald	Florian Bundschuh	Luanne Burns
William Burge	Ralph Byers	Quentin Carpent
Robert Caviness	Bruce Char	Ondrej Certik
Tzu-Yi Chen	Cheekai Chin	David V. Chudnovsky
Gregory V. Chudnovsky	Mark Clements	James Cloos
Jia Zhao Cong	Josh Cohen	Christophe Conil
Don Coppersmith	George Corliss	Robert Corless
Gary Cornell	Meino Cramer	Jeremy Du Croz
David Cyganski	Nathaniel Daly	Timothy Daly Sr.
Timothy Daly Jr.	James H. Davenport	David Day
James Demmel	Didier Deshombres	Michael Dewar
Jack Dongarra	Jean Della Dora	Gabriel Dos Reis
Claire DiCrescendo	Sam Dooley	Lionel Ducos
Iain Duff	Lee Duhem	Martin Dunstan
Brian Dupee	Dominique Duval	Robert Edwards
Heow Eide-Goodman	Lars Erickson	Richard Fateman
Bertfried Fauser	Stuart Feldman	John Fletcher
Brian Ford	Albrecht Fortenbacher	George Frances
Constantine Frangos	Timothy Freeman	Korrinn Fu
Marc Gaetano	Rudiger Gebauer	Van de Geijn
Kathy Gerber	Patricia Gianni	Samantha Goldrich
Holger Gollan	Teresa Gomez-Diaz	Laureano Gonzalez-Vega
Stephen Gortler	Johannes Grabmeier	Matt Grayson
Klaus Ebbe Grue	James Griesmer	Vladimir Grinberg
Oswald Gschnitzer	Ming Gu	Jocelyn Guidry
Gaetan Hache	Steve Hague	Satoshi Hamaguchi
Sven Hammarling	Mike Hansen	Richard Hanson
Richard Harke	Bill Hart	Vilya Harvey
Martin Hassner	Arthur S. Hathaway	Dan Hatton
Waldek Hebisch	Karl Hegbloom	Ralf Hemmecke

Henderson	Antoine Hersen	Roger House
Gernot Hueber	Pietro Iglio	Alejandro Jakubi
Richard Jenks	William Kahan	Kai Kaminski
Grant Keady	Wilfrid Kendall	Tony Kennedy
Ted Kosan	Paul Kosinski	Klaus Kusche
Bernhard Kutzler	Tim Lahey	Larry Lambe
Kaj Laurson	George L. Legendre	Franz Lehner
Frederic Lehouby	Michel Levaud	Howard Levy
Ren-Cang Li	Rudiger Loos	Michael Lucks
Richard Luczak	Camm Maguire	Francois Maltey
Alasdair McAndrew	Bob McElrath	Michael McGettrick
Edi Meier	Ian Meikle	David Mentre
Victor S. Miller	Gerard Milmeister	Mohammed Mobarak
H. Michael Moeller	Michael Monagan	Marc Moreno-Maza
Scott Morrison	Joel Moses	Mark Murray
William Naylor	Patrice Naudin	C. Andrew Neff
John Nelder	Godfrey Nolan	Arthur Norman
Jinzhong Niu	Michael O'Connor	Summat Oemrawsingh
Kostas Oikonomou	Humberto Ortiz-Zuazaga	Julian A. Padget
Bill Page	David Parnas	Susan Pelzel
Michel Petitot	Didier Pinchon	Ayal Pinkus
Frederick H. Pitts	Jose Alfredo Portes	Gregorio Quintana-Orti
Claude Quitte	Arthur C. Ralfs	Norman Ramsey
Anatoly Raportirenko	Albert D. Rich	Michael Richardson
Guilherme Reis	Huan Ren	Renaud Rioboo
Jean Rivlin	Nicolas Robidoux	Simon Robinson
Raymond Rogers	Michael Rothstein	Martin Rubey
Philip Santas	Alfred Scheerhorn	William Schelter
Gerhard Schneider	Martin Schoenert	Marshall Schor
Frithjof Schulze	Fritz Schwarz	Steven Segletes
V. Sima	Nick Simicich	William Sit
Elena Smirnova	Jonathan Steinbach	Fabio Stumbo
Christine Sundaresan	Robert Sutor	Moss E. Sweedler
Eugene Surowitz	Max Tegmark	T. Doug Telford
James Thatcher	Balbir Thomas	Mike Thomas
Dylan Thurston	Steve Toleque	Barry Trager
Themos T. Tsikas	Gregory Vanuxem	Bernhard Wall
Stephen Watt	Jaap Weel	Juergen Weiss
M. Weller	Mark Wegman	James Wen
Thorsten Werther	Michael Wester	R. Clint Whaley
John M. Wiley	Berhard Will	Clifton J. Williamson
Stephen Wilson	Shmuel Winograd	Robert Wisbauer
Sandra Wityak	Waldemar Wiwianka	Knut Wolf
Liu Xiaojun	Clifford Yapp	David Yun
Vadim Zhytnikov	Richard Zippel	Evelyn Zoernack
Bruno Zuercher	Dan Zwillinger	

Contents

1	Chapter Overview	1
2	Chapter A	3
	domain AFFPL AffinePlane	3
	AffinePlane (AFFPL)	4
	domain AFFPLPS AffinePlaneOverPseudoAlgebraicClosureOfFiniteField	5
	AffinePlaneOverPseudoAlgebraicClosureOfFiniteField (AFFPLPS)	7
	domain AFFSP AffineSpace	8
	AffineSpace (AFFSP)	9
	domain ALGSC AlgebraGivenByStructuralConstants	12
	AlgebraGivenByStructuralConstants (ALGSC)	14
	domain ALGFF AlgebraicFunctionField	23
	AlgebraicFunctionField (ALGFF)	27
	domain AN AlgebraicNumber	32
	AlgebraicNumber (AN)	35
	domain ANON AnonymousFunction	37
	AnonymousFunction (ANON)	38
	domain ANTISYM AntiSymm	39
	AntiSymm (ANTISYM)	40
	domain ANY Any	45
	Any (ANY)	50
	domain ASTACK ArrayStack	52
	ArrayStack (ASTACK)	65
	domain ASP1 Asp1	69
	Asp1 (ASP1)	71
	domain ASP10 Asp10	73
	Asp10 (ASP10)	75
	domain ASP12 Asp12	78
	Asp12 (ASP12)	80
	domain ASP19 Asp19	81
	Asp19 (ASP19)	85
	domain ASP20 Asp20	91
	Asp20 (ASP20)	92
	domain ASP24 Asp24	96

Asp24 (ASP24)	98
domain ASP27 Asp27	100
Asp27 (ASP27)	102
domain ASP28 Asp28	104
Asp28 (ASP28)	108
domain ASP29 Asp29	113
Asp29 (ASP29)	114
domain ASP30 Asp30	116
Asp30 (ASP30)	118
domain ASP31 Asp31	121
Asp31 (ASP31)	122
domain ASP33 Asp33	126
Asp33 (ASP33)	127
domain ASP34 Asp34	129
Asp34 (ASP34)	130
domain ASP35 Asp35	133
Asp35 (ASP35)	134
domain ASP4 Asp4	138
Asp4 (ASP4)	140
domain ASP41 Asp41	142
Asp41 (ASP41)	144
domain ASP42 Asp42	149
Asp42 (ASP42)	151
domain ASP49 Asp49	157
Asp49 (ASP49)	158
domain ASP50 Asp50	162
Asp50 (ASP50)	164
domain ASP55 Asp55	167
Asp55 (ASP55)	169
domain ASP6 Asp6	174
Asp6 (ASP6)	176
domain ASP7 Asp7	179
Asp7 (ASP7)	181
domain ASP73 Asp73	184
Asp73 (ASP73)	185
domain ASP74 Asp74	189
Asp74 (ASP74)	191
domain ASP77 Asp77	195
Asp77 (ASP77)	196
domain ASP78 Asp78	200
Asp78 (ASP78)	201
domain ASP8 Asp8	204
Asp8 (ASP8)	206
domain ASP80 Asp80	209
Asp80 (ASP80)	211
domain ASP9 Asp9	214

Asp9 (ASP9)	216
domain JORDAN AssociatedJordanAlgebra	219
AssociatedJordanAlgebra (JORDAN)	222
domain LIE AssociatedLieAlgebra	225
AssociatedLieAlgebra (LIE)	228
domain ALIST AssociationList	231
AssociationList (ALIST)	235
domain ATTRBUT AttributeButtons	237
AttributeButtons (ATTRBUT)	239
domain AUTOMOR Automorphism	244
Automorphism (AUTOMOR)	245
3 Chapter B	249
domain BBTREE BalancedBinaryTree	249
BalancedBinaryTree (BBTREE)	252
domain BPADIC BalancedPAdicInteger	256
BalancedPAdicInteger (BPADIC)	258
domain BPADICRT BalancedPAdicRational	259
BalancedPAdicRational (BPADICRT)	262
domain BFUNCT BasicFunctions	264
BasicFunctions (BFUNCT)	265
domain BOP BasicOperator	267
BasicOperator (BOP)	273
domain BSD BasicStochasticDifferential	278
BasicStochasticDifferential (BSD)	285
domain BINARY BinaryExpansion	288
BinaryExpansion (BINARY)	292
domain BINFILE BinaryFile	293
BinaryFile (BINFILE)	295
domain BSTREE BinarySearchTree	297
BinarySearchTree (BSTREE)	302
domain BTOURN BinaryTournament	304
BinaryTournament (BTOURN)	306
domain BTREE BinaryTree	308
BinaryTree (BTREE)	310
domain BITS Bits	312
Bits (BITS)	314
domain BLHN BlowUpWithHamburgerNoether	315
BlowUpWithHamburgerNoether (BLHN)	317
domain BLQT BlowUpWithQuadTrans	318
BlowUpWithQuadTrans (BLQT)	319
domain BOOLEAN Boolean	321
Boolean (BOOLEAN)	322

4 Chapter C	325
domain CARD CardinalNumber	325
CardinalNumber (CARD)	332
domain CARTEN CartesianTensor	336
CartesianTensor (CARTEN)	356
domain CHAR Character	368
Character (CHAR)	373
domain CCLASS CharacterClass	376
CharacterClass (CCLASS)	381
domain CLIF CliffordAlgebra[?, ?]	384
Vector (linear) spaces	384
Quadratic Forms[?]	386
Quadratic spaces, Clifford Maps[?, ?]	386
Universal Clifford algebras[?]	386
Real Clifford algebras $\mathbb{R}_{p,q}$ [?]	386
Notation for integer sets	387
Frames for Clifford algebras[?, ?, ?]	387
Real frame groups[?, ?]	387
Canonical products[?, ?, ?]	388
Clifford algebra of frame group[?, ?, ?, ?]	388
Neutral matrix representations[?, ?, ?]	388
CliffordAlgebra (CLIF)	401
domain COLOR Color	406
Color (COLOR)	407
domain COMM Commutator	409
Commutator (COMM)	411
domain COMPLEX Complex	412
Complex (COMPLEX)	418
domain CDFMAT ComplexDoubleFloatMatrix	421
ComplexDoubleFloatMatrix (CDFMAT)	426
domain CDFVEC ComplexDoubleFloatVector	428
ComplexDoubleFloatVector (CDFVEC)	432
domain CONTFRAC ContinuedFraction	434
ContinuedFraction (CONTFRAC)	445
5 Chapter D	453
domain DBASE Database	453
Database (DBASE)	454
domain DLIST DataList	456
DataList (DLIST)	459
domain DECIMAL DecimalExpansion	461
DecimalExpansion (DECIMAL)	465
Denavit-Hartenberg Matrices	466
Homogeneous Transformations	466
Notation	467
Vectors	468

Planes	469
Transformations	471
Translation Transformation	471
Rotation Transformations	473
Coordinate Frames	476
Relative Transformations	477
Objects	478
Inverse Transformations	478
General Rotation Transformation	479
Equivalent Angle and Axis of Rotation	482
Example 1.1	485
Stretching and Scaling	486
Perspective Transformations	487
Transform Equations	489
Summary	489
DenavitHartenbergMatrix (DHMATRIX)	491
domain DEQUEUE Dequeue	494
Dequeue (DEQUEUE)	512
domain DERHAM DeRhamComplex	518
DeRhamComplex (DERHAM)	530
domain DSTREE DesingTree	533
DesingTree (DSTREE)	534
domain DSMP DifferentialSparseMultivariatePolynomial	537
DifferentialSparseMultivariatePolynomial (DSMP)	541
domain DIRPROD DirectProduct	543
DirectProduct (DIRPROD)	547
domain DPMM DirectProductMatrixModule	549
DirectProductMatrixModule (DPMM)	552
domain DPMO DirectProductModule	554
DirectProductModule (DPMO)	557
domain DIRRING DirichletRing	558
DirichletRing (DIRRING)	563
domain DMP DistributedMultivariatePolynomial	567
DistributedMultivariatePolynomial (DMP)	571
domain DIV Divisor	573
Divisor (DIV)	575
domain DFLOAT DoubleFloat	578
DoubleFloat (DFLOAT)	586
domain DFMAT DoubleFloatMatrix	595
DoubleFloatMatrix (DFMAT)	599
domain DFVEC DoubleFloatVector	601
DoubleFloatVector (DFVEC)	605
domain DROPT DrawOption	606
DrawOption (DROPT)	608
domain D01AJFA d01ajfAnnaType	613
d01ajfAnnaType (D01AJFA)	614

domain D01AKFA d01akfAnnaType	615
d01akfAnnaType (D01AKFA)	617
domain D01ALFA d01alfAnnaType	618
d01alfAnnaType (D01ALFA)	620
domain D01AMFA d01amfAnnaType	622
d01amfAnnaType (D01AMFA)	623
domain D01ANFA d01anfAnnaType	625
d01anfAnnaType (D01ANFA)	626
domain D01APFA d01apfAnnaType	628
d01apfAnnaType (D01APFA)	629
domain D01AQFA d01aqfAnnaType	631
d01aqfAnnaType (D01AQFA)	633
domain D01ASFA d01asfAnnaType	635
d01asfAnnaType (D01ASFA)	636
domain D01FCFA d01fcfAnnaType	638
d01fcfAnnaType (D01FCFA)	639
domain D01GBFA d01gbfAnnaType	641
d01gbfAnnaType (D01GBFA)	643
domain D01TRNS d01TransformFunctionType	645
d01TransformFunctionType (D01TRNS)	646
domain D02BBFA d02bbfAnnaType	649
d02bbfAnnaType (D02BBFA)	651
domain D02BHFA d02bhfAnnaType	653
d02bhfAnnaType (D02BHFA)	654
domain D02CJFA d02cjfAnnaType	657
d02cjfAnnaType (D02CJFA)	658
domain D02EJFA d02ejfAnnaType	660
d02ejfAnnaType (D02EJFA)	661
domain D03EEFA d03eefAnnaType	664
d03eefAnnaType (D03EEFA)	665
domain D03FAFA d03fafAnnaType	667
d03fafAnnaType (D03FAFA)	668
6 Chapter E	671
domain EFULS ElementaryFunctionsUnivariateLaurentSeries	671
ElementaryFunctionsUnivariateLaurentSeries (EFULS)	672
domain EFUPXS ElementaryFunctionsUnivariatePuisseuxSeries	680
ElementaryFunctionsUnivariatePuisseuxSeries (EFUPXS)	681
domain EQ Equation	687
Equation (EQ)	691
domain EQTBL EqTable	696
EqTable (EQTBL)	698
domain EMR EuclideanModularRing	699
EuclideanModularRing (EMR)	701
domain EXIT Exit	704
Exit (EXIT)	707

domain EXPEXPAN ExponentialExpansion	708
ExponentialExpansion (EXPEXPAN)	711
domain EXPR Expression	715
Expression (EXPR)	723
domain EXPUPXS ExponentialOfUnivariatePuisseuxSeries	735
ExponentialOfUnivariatePuisseuxSeries (EXPUPXS)	739
domain EAB ExtAlgBasis	742
ExtAlgBasis (EAB)	743
domain E04DGFA e04dgfAnnaType	745
e04dgfAnnaType (E04DGFA)	747
domain E04FDFA e04fdfAnnaType	749
e04fdfAnnaType (E04FDFA)	750
domain E04GCFA e04gcfAnnaType	753
e04gcfAnnaType (E04GCFA)	754
domain E04JAFA e04jafAnnaType	757
e04jafAnnaType (E04JAFA)	758
domain E04MBFA e04mbfAnnaType	760
e04mbfAnnaType (E04MBFA)	762
domain E04NAFA e04nafAnnaType	764
e04nafAnnaType (E04NAFA)	765
domain E04UCFA e04ucfAnnaType	767
e04ucfAnnaType (E04UCFA)	769
7 Chapter F	773
domain FR Factored	773
Factored (FR)	786
domain FILE File	797
File (FILE)	802
domain FNAME FileName	804
FileName (FNAME)	810
domain FDIV FiniteDivisor	811
FiniteDivisor (FDIV)	813
domain FF FiniteField	816
FiniteField (FF)	819
domain FFCG FiniteFieldCyclicGroup	821
FiniteFieldCyclicGroup (FFCG)	824
domain FFCGX FiniteFieldCyclicGroupExtension	826
FiniteFieldCyclicGroupExtension (FFCGX)	829
domain FFCGP FiniteFieldCyclicGroupExtensionByPolynomial	831
FiniteFieldCyclicGroupExtensionByPolynomial (FFCGP)	834
domain FFX FiniteFieldExtension	842
FiniteFieldExtension (FFX)	845
domain FFP FiniteFieldExtensionByPolynomial	847
FiniteFieldExtensionByPolynomial (FFP)	850
domain FFNB FiniteFieldNormalBasis	856
FiniteFieldNormalBasis (FFNB)	859

domain FFNBX FiniteFieldNormalBasisExtension	861
FiniteFieldNormalBasisExtension (FFNBX)	865
domain FFNBP FiniteFieldNormalBasisExtensionByPolynomial	867
FiniteFieldNormalBasisExtensionByPolynomial (FFNBP)	871
domain FARRAY FlexibleArray	880
FlexibleArray (FARRAY)	885
domain FLOAT Float	887
Float (FLOAT)	908
domain FC FortranCode	929
FortranCode (FC)	931
domain FEXPR FortranExpression	943
FortranExpression (FEXPR)	946
domain FORTRAN FortranProgram	954
FortranProgram (FORTRAN)	955
domain FST FortranScalarType	960
FortranScalarType (FST)	962
domain FTEM FortranTemplate	965
FortranTemplate (FTEM)	967
domain FT FortranType	969
FortranType (FT)	971
domain FCOMP FourierComponent	973
FourierComponent (FCOMP)	975
domain FSERIES FourierSeries	976
FourierSeries (FSERIES)	978
domain FRAC Fraction	980
Fraction (FRAC)	985
domain FRIDEAL FractionalIdeal	993
FractionalIdeal (FRIDEAL)	994
domain FRMOD FramedModule	998
FramedModule (FRMOD)	1000
domain FAGROUP FreeAbelianGroup	1002
FreeAbelianGroup (FAGROUP)	1004
domain FAMONOID FreeAbelianMonoid	1006
FreeAbelianMonoid (FAMONOID)	1007
domain FGROUP FreeGroup	1008
FreeGroup (FGROUP)	1010
domain FM FreeModule	1012
FreeModule (FM)	1013
domain FM1 FreeModule1	1015
FreeModule1 (FM1)	1016
domain FMONOID FreeMonoid	1019
FreeMonoid (FMONOID)	1021
domain FNLA FreeNilpotentLie	1025
FreeNilpotentLie (FNLA)	1027
domain FPARFRAC FullPartialFractionExpansion	1030
FullPartialFractionExpansion (FPARFRAC)	1040

domain FUNCTION FunctionCalled	1044
FunctionCalled (FUNCTION)	1045
8 Chapter G	1047
domain GDMP GeneralDistributedMultivariatePolynomial	1047
GeneralDistributedMultivariatePolynomial (GDMP)	1052
domain GMODPOL GeneralModulePolynomial	1058
GeneralModulePolynomial (GMODPOL)	1059
domain GCNAALG GenericNonAssociativeAlgebra	1061
GenericNonAssociativeAlgebra (GCNAALG)	1064
domain GPOLSET GeneralPolynomialSet	1072
GeneralPolynomialSet (GPOLSET)	1074
domain GSTBL GeneralSparseTable	1076
GeneralSparseTable (GSTBL)	1078
domain GTSET GeneralTriangularSet	1080
GeneralTriangularSet (GTSET)	1082
domain GSERIES GeneralUnivariatePowerSeries	1086
GeneralUnivariatePowerSeries (GSERIES)	1089
domain GRIMAGE GraphImage	1093
GraphImage (GRIMAGE)	1094
domain GOPT GuessOption	1103
GuessOption (GOPT)	1104
domain GOPT0 GuessOptionFunctions0	1108
GuessOptionFunctions0 (GOPT0)	1110
9 Chapter H	1117
domain HASHTBL HashTable	1117
HashTable (HASHTBL)	1120
domain HEAP Heap	1121
Heap (HEAP)	1134
domain HEXADEC HexadecimalExpansion	1139
HexadecimalExpansion (HEXADEC)	1142
package HTMLFORM HTMLFormat	1144
Overview	1144
Why output to HTML?	1145
Using the formatter	1145
Form of the output	1146
Matrix Formatting	1146
Programmers Guide	1146
Future Developments	1147
HTMLFormat (HTMLFORM)	1152
domain HDP HomogeneousDirectProduct	1169
HomogeneousDirectProduct (HDP)	1172
domain HDMP HomogeneousDistributedMultivariatePolynomial	1174
HomogeneousDistributedMultivariatePolynomial (HDMP)	1179
domain HELLDIV HyperellipticFiniteDivisor	1181

HyperellipticFiniteDivisor (HELLFDIV)	1183
10 Chapter I	1189
domain ICP InfClsPt	1189
InfClsPt (ICP)	1190
domain ICARD IndexCard	1192
IndexCard (ICARD)	1193
domain IBITS IndexedBits	1195
IndexedBits (IBITS)	1199
domain IDPAG IndexedDirectProductAbelianGroup	1201
IndexedDirectProductAbelianGroup (IDPAG)	1202
domain IDPAM IndexedDirectProductAbelianMonoid	1204
IndexedDirectProductAbelianMonoid (IDPAM)	1205
domain IDPO IndexedDirectProductObject	1208
IndexedDirectProductObject (IDPO)	1209
domain IDPOAM IndexedDirectProductOrderedAbelianMonoid	1210
IndexedDirectProductOrderedAbelianMonoid (IDPOAM)	1212
domain IDPOAMS IndexedDirectProductOrderedAbelianMonoidSup	1213
IndexedDirectProductOrderedAbelianMonoidSup (IDPOAMS)	1214
domain INDE IndexedExponents	1216
IndexedExponents (INDE)	1217
domain IFARRAY IndexedFlexibleArray	1218
IndexedFlexibleArray (IFARRAY)	1222
domain ILIST IndexedList	1227
IndexedList (ILIST)	1231
domain IMATRIX IndexedMatrix	1235
IndexedMatrix (IMATRIX)	1238
domain IARRAY1 IndexedOneDimensionalArray	1240
IndexedOneDimensionalArray (IARRAY1)	1243
domain ISTRING IndexedString	1246
IndexedString (ISTRING)	1249
domain IARRAY2 IndexedTwoDimensionalArray	1254
IndexedTwoDimensionalArray (IARRAY2)	1256
domain IVECTOR IndexedVector	1257
IndexedVector (IVECTOR)	1260
domain ITUPLE InfiniteTuple	1261
InfiniteTuple (ITUPLE)	1262
domain INFCLSPT InfinitelyClosePoint	1263
InfinitelyClosePoint (INFCLSPT)	1265
domain INFCLSPS InfinitelyClosePointOverPseudoAlgebraicClosureOfFiniteField	1269
InfinitelyClosePointOverPseudoAlgebraicClosureOfFiniteField (INFCLSPS)	1271
domain IAN InnerAlgebraicNumber	1272
InnerAlgebraicNumber (IAN)	1276
domain IFF InnerFiniteField	1280
InnerFiniteField (IFF)	1283
domain IFAMON InnerFreeAbelianMonoid	1285

InnerFreeAbelianMonoid (IFAMON)	1286
domain IIARRAY2 InnerIndexedTwoDimensionalArray	1288
InnerIndexedTwoDimensionalArray (IIARRAY2)	1290
domain IPADIC InnerPAdicInteger	1292
InnerPAdicInteger (IPADIC)	1294
domain IPF InnerPrimeField	1300
InnerPrimeField (IPF)	1303
domain ISUPS InnerSparseUnivariatePowerSeries	1307
InnerSparseUnivariatePowerSeries (ISUPS)	1310
domain INTABL InnerTable	1333
InnerTable (INTABL)	1335
domain ITAYLOR InnerTaylorSeries	1336
InnerTaylorSeries (ITAYLOR)	1338
domain INFORM InputForm	1341
InputForm (INFORM)	1343
domain INT Integer	1347
Integer (INT)	1361
domain ZMOD IntegerMod	1366
IntegerMod (ZMOD)	1367
domain INTFTBL IntegrationFunctionsTable	1370
IntegrationFunctionsTable (INTFTBL)	1371
domain IR IntegrationResult	1373
IntegrationResult (IR)	1375
domain INTRVL Interval	1379
Interval (INTRVL)	1384
11 Chapter J	1395
12 Chapter K	1397
domain KERNEL Kernel	1397
Kernel (KERNEL)	1404
domain KAFfile KeyedAccessFile	1407
KeyedAccessFile (KAFfile)	1413
13 Chapter L	1419
domain LAUPOL LaurentPolynomial	1419
LaurentPolynomial (LAUPOL)	1421
domain LIB Library	1425
Library (LIB)	1428
domain LEXP LieExponentials	1430
LieExponentials (LEXP)	1435
domain LPOLY LiePolynomial	1438
LiePolynomial (LPOLY)	1446
domain LSQM LieSquareMatrix	1451
LieSquareMatrix (LSQM)	1455
domain LODO LinearOrdinaryDifferentialOperator	1458

LinearOrdinaryDifferentialOperator (LODO)	1469
domain LODO1 LinearOrdinaryDifferentialOperator1	1470
LinearOrdinaryDifferentialOperator1 (LODO1)	1479
domain LODO2 LinearOrdinaryDifferentialOperator2	1480
LinearOrdinaryDifferentialOperator2 (LODO2)	1491
domain LIST List	1492
List (LIST)	1504
domain LMOPS ListMonoidOps	1507
ListMonoidOps (LMOPS)	1509
domain LMDICT ListMultiDictionary	1513
ListMultiDictionary (LMDICT)	1515
domain LA LocalAlgebra	1518
LocalAlgebra (LA)	1520
domain LO Localize	1521
Localize (LO)	1522
domain LWORD LyndonWord	1524
LyndonWord (LWORD)	1531
14 Chapter M	1537
domain MCMPLX MachineComplex	1537
MachineComplex (MCMPLX)	1542
domain MFLOAT MachineFloat	1545
MachineFloat (MFLOAT)	1547
domain MINT MachineInteger	1554
MachineInteger (MINT)	1557
domain MAGMA Magma	1559
Magma (MAGMA)	1565
domain MKCHSET MakeCachableSet	1568
MakeCachableSet (MKCHSET)	1570
domain MMLFORM MathMLFormat	1571
Introduction to Mathematical Markup Language	1572
Displaying MathML	1572
Test Cases	1573
)set output mathml on	1574
File src/interp/setvars.boot.pamphlet	1574
File setvart.boot.pamphlet	1574
File src/algebra/Makefile.pamphlet	1575
File src/algebra/exposed.lsp.pamphlet	1575
File src/algebra/Lattice.pamphlet	1575
File src/doc/axiom.bib.pamphlet	1576
File interp/i-output.boot.pamphlet	1576
Public Declarations	1576
Private Constant Declarations	1578
Private Function Declarations	1579
Public Function Definitions	1581
Private Function Definitions	1582

Mathematical Markup Language Form	1599
MathMLForm (MMLFORM)	1603
domain MATRIX Matrix	1603
Matrix (MATRIX)	1622
domain MODMON ModMonic	1627
ModMonic (MODMON)	1631
domain MODFIELD ModularField	1636
ModularField (MODFIELD)	1638
domain MODRING ModularRing	1639
ModularRing (MODRING)	1641
domain MODMONOM ModuleMonomial	1643
ModuleMonomial (MODMONOM)	1644
domain MODOP ModuleOperator	1645
ModuleOperator (MODOP)	1647
domain MOEBIUS MoebiusTransform	1652
MoebiusTransform (MOEBIUS)	1654
domain MRING MonoidRing	1656
MonoidRing (MRING)	1658
domain MSET Multiset	1665
Multiset (MSET)	1670
domain MPOLY MultivariatePolynomial	1676
MultivariatePolynomial (MPOLY)	1681
domain MYEXPR MyExpression	1683
MyExpression (MYEXPR)	1687
domain MYUP MyUnivariatePolynomial	1689
MyUnivariatePolynomial (MYUP)	1694
15 Chapter N	1697
domain NSDPS NeitherSparseOrDensePowerSeries	1697
NeitherSparseOrDensePowerSeries (NSDPS)	1701
domain NSMP NewSparseMultivariatePolynomial	1709
NewSparseMultivariatePolynomial (NSMP)	1713
domain NSUP NewSparseUnivariatePolynomial	1723
NewSparseUnivariatePolynomial (NSUP)	1728
domain NONE None	1735
None (NONE)	1737
domain NNI NonNegativeInteger	1738
NonNegativeInteger (NNI)	1739
domain NOTTING NottinghamGroup	1740
NottinghamGroup (NOTTING)	1744
domain NIPROB NumericalIntegrationProblem	1745
NumericalIntegrationProblem (NIPROB)	1746
domain ODEPROB NumericalODEProblem	1748
NumericalODEProblem (ODEPROB)	1750
domain OPTPROB NumericalOptimizationProblem	1751
NumericalOptimizationProblem (OPTPROB)	1753

domain PDEPROB NumericalPDEProblem	1754
NumericalPDEProblem (PDEPROB)	1756
16 Chapter O	1759
domain OCT Octonion	1759
Octonion (OCT)	1765
domain ODEIFTBL ODEIntensityFunctionsTable	1767
ODEIntensityFunctionsTable (ODEIFTBL)	1768
domain ARRAY1 OneDimensionalArray	1770
OneDimensionalArray (ARRAY1)	1774
domain ONECOMP OnePointCompletion	1775
OnePointCompletion (ONECOMP)	1777
domain OMCONN OpenMathConnection	1780
OpenMathConnection (OMCONN)	1781
domain OMDEV OpenMathDevice	1782
OpenMathDevice (OMDEV)	1784
domain OMENC OpenMathEncoding	1788
OpenMathEncoding (OMENC)	1789
domain OMERR OpenMathError	1790
OpenMathError (OMERR)	1792
domain OMERRK OpenMathErrorKind	1793
OpenMathErrorKind (OMERRK)	1794
domain OP Operator	1796
Operator (OP)	1804
domain OMLO OppositeMonogenicLinearOperator	1805
OppositeMonogenicLinearOperator (OMLO)	1806
domain ORDCOMP OrderedCompletion	1807
OrderedCompletion (ORDCOMP)	1809
domain ODP OrderedDirectProduct	1813
OrderedDirectProduct (ODP)	1816
domain OFMONOID OrderedFreeMonoid	1817
OrderedFreeMonoid (OFMONOID)	1829
domain OVAR OrderedVariableList	1834
OrderedVariableList (OVAR)	1836
domain ODPOL OrderlyDifferentialPolynomial	1837
OrderlyDifferentialPolynomial (ODPOL)	1851
domain ODVAR OrderlyDifferentialVariable	1853
OrderlyDifferentialVariable (ODVAR)	1855
domain ODR OrdinaryDifferentialRing	1856
OrdinaryDifferentialRing (ODR)	1858
domain OWP OrdinaryWeightedPolynomials	1859
OrdinaryWeightedPolynomials (OWP)	1861
domain OSI OrdSetInts	1862
OrdSetInts (OSI)	1864
domain OUTFORM OutputForm	1865
OutputForm (OUTFORM)	1867

17 Chapter P	1879
domain PADIC PAdicInteger	1879
PAdicInteger (PADIC)	1881
domain PADICRAT PAdicRational	1882
PAdicRational (PADICRAT)	1885
domain PADICRC PAdicRationalConstructor	1887
PAdicRationalConstructor (PADICRC)	1890
domain PALETTE Palette	1895
Palette (PALETTE)	1896
domain PARPCURV ParametricPlaneCurve	1898
ParametricPlaneCurve (PARPCURV)	1899
domain PARSCURV ParametricSpaceCurve	1900
ParametricSpaceCurve (PARSCURV)	1901
domain PARSURF ParametricSurface	1903
ParametricSurface (PARSURF)	1904
domain PFR PartialFraction	1905
PartialFraction (PFR)	1913
domain PRITITION Partition	1921
Partition (PRITITION)	1923
domain PATTERN Pattern	1926
Pattern (PATTERN)	1928
domain PATLRES PatternMatchListResult	1936
PatternMatchListResult (PATLRES)	1937
domain PATRES PatternMatchResult	1939
PatternMatchResult (PATRES)	1940
domain PENDTREE PendantTree	1942
PendantTree (PENDTREE)	1944
domain PERM Permutation	1946
Permutation (PERM)	1949
domain PERMGRP PermutationGroup	1958
PermutationGroup (PERMGRP)	1959
domain HACKPI Pi	1975
Pi (HACKPI)	1977
domain ACPLOT PlaneAlgebraicCurvePlot	1979
PlaneAlgebraicCurvePlot (ACPLOT)	1992
domain PLACES Places	2017
Places (PLACES)	2019
domain PLACESPS PlacesOverPseudoAlgebraicClosureOffiniteField	2020
PlacesOverPseudoAlgebraicClosureOffiniteField (PLACESPS)	2021
domain PLCS Plcs	2022
Plcs (PLCS)	2023
domain PLOT Plot	2027
Plot (PLOT)	2029
domain PLOT3D Plot3D	2041
Plot3D (PLOT3D)	2042
domain PBWLB PoincareBirkhoffWittLyndonBasis	2052

PoincareBirkhoffWittLyndonBasis (PBWLB)	2054
domain POINT Point	2056
Point (POINT)	2059
domain POLY Polynomial	2061
Polynomial (POLY)	2077
domain IDEAL PolynomialIdeals	2079
PolynomialIdeals (IDEAL)	2081
domain PR PolynomialRing	2090
PolynomialRing (PR)	2092
domain PI PositiveInteger	2098
PositiveInteger (PI)	2100
domain PF PrimeField	2101
PrimeField (PF)	2104
domain PRIMARR PrimitiveArray	2106
PrimitiveArray (PRIMARR)	2109
domain PRODUCT Product	2110
Product (PRODUCT)	2112
domain PROJPL ProjectivePlane	2115
ProjectivePlane (PROJPL)	2117
domain PROJPLPS ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField	2118
ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField (PROJPLPS)	2119
domain PROJSP ProjectiveSpace	2120
ProjectiveSpace (PROJSP)	2121
domain PACEXT PseudoAlgebraicClosureOfAlgExtOfRationalNumber	2125
PseudoAlgebraicClosureOfAlgExtOfRationalNumber (PACEXT)	2126
domain PACOFF PseudoAlgebraicClosureOfFiniteField	2132
PseudoAlgebraicClosureOfFiniteField (PACOFF)	2135
domain PACRAT PseudoAlgebraicClosureOfRationalNumber	2143
PseudoAlgebraicClosureOfRationalNumber (PACRAT)	2146
18 Chapter Q	2153
domain QFORM QuadraticForm	2153
QuadraticForm (QFORM)	2154
domain QALGSET QuasiAlgebraicSet	2155
QuasiAlgebraicSet (QALGSET)	2157
domain QUAT Quaternion	2161
Quaternion (QUAT)	2167
domain QEQUAT QueryEquation	2168
QueryEquation (QEQUAT)	2169
domain QUEUE Queue	2170
Queue (QUEUE)	2184

19 Chapter R	2189
domain RADFF RadicalFunctionField	2189
RadicalFunctionField (RADFF)	2193
domain RADIX RadixExpansion	2199
RadixExpansion (RADIX)	2205
domain RECLOS RealClosure	2211
RealClosure (RECLOS)	2236
domain RMATRIX RectangularMatrix	2243
RectangularMatrix (RMATRIX)	2245
domain REF Reference	2248
Reference (REF)	2249
domain RGCHAIN RegularChain	2250
RegularChain (RGCHAIN)	2254
domain REGSET RegularTriangularSet	2257
RegularTriangularSet (REGSET)	2285
domain RESRING ResidueRing	2294
ResidueRing (RESRING)	2296
domain RESULT Result	2297
Result (RESULT)	2300
domain RULE RewriteRule	2302
RewriteRule (RULE)	2305
domain ROIRC RightOpenIntervalRootCharacterization	2308
RightOpenIntervalRootCharacterization (ROIRC)	2310
domain ROMAN RomanNumeral	2320
RomanNumeral (ROMAN)	2326
domain ROUTINE RoutinesTable	2327
RoutinesTable (ROUTINE)	2330
domain RULECOLD RuleCalled	2339
RuleCalled (RULECOLD)	2340
domain RULESET Ruleset	2341
Ruleset (RULESET)	2342
20 Chapter S	2345
domain FORMULA ScriptFormulaFormat	2345
ScriptFormulaFormat (FORMULA)	2346
domain SEG Segment	2355
Segment (SEG)	2359
domain SEGBIND SegmentBinding	2361
SegmentBinding (SEGBIND)	2364
domain SET Set	2365
Set (SET)	2372
domain SETMN SetOfMIntegersInOneToN	2376
SetOfMIntegersInOneToN (SETMN)	2377
domain SDPOL SequentialDifferentialPolynomial	2381
SequentialDifferentialPolynomial (SDPOL)	2385
domain SDVAR SequentialDifferentialVariable	2387

SequentialDifferentialVariable (SDVAR)	2389
domain SEX SExpression	2390
SExpression (SEX)	2391
domain SEXOF SExpressionOf	2392
SExpressionOf (SEXOF)	2394
domain SAE SimpleAlgebraicExtension	2396
SimpleAlgebraicExtension (SAE)	2399
domain SFORT SimpleFortranProgram	2404
SimpleFortranProgram (SFORT)	2405
domain SINT SingleInteger	2407
SingleInteger (SINT)	2411
domain SAOS SingletonAsOrderedSet	2416
SingletonAsOrderedSet (SAOS)	2418
domain SMP SparseMultivariatePolynomial	2419
SparseMultivariatePolynomial (SMP)	2422
domain SMTS SparseMultivariateTaylorSeries	2435
SparseMultivariateTaylorSeries (SMTS)	2440
domain STBL SparseTable	2447
SparseTable (STBL)	2450
domain SULS SparseUnivariateLaurentSeries	2451
SparseUnivariateLaurentSeries (SULS)	2456
domain SUP SparseUnivariatePolynomial	2462
SparseUnivariatePolynomial (SUP)	2466
domain SUEXPR SparseUnivariatePolynomialExpressions	2475
SparseUnivariatePolynomialExpressions (SUEXPR)	2480
domain SUPXS SparseUnivariatePuisseuxSeries	2483
SparseUnivariatePuisseuxSeries (SUPXS)	2487
domain ORESUP SparseUnivariateSkewPolynomial	2489
SparseUnivariateSkewPolynomial (ORESUP)	2492
domain SUTS SparseUnivariateTaylorSeries	2493
SparseUnivariateTaylorSeries (SUTS)	2496
domain SHDP SplitHomogeneousDirectProduct	2505
SplitHomogeneousDirectProduct (SHDP)	2508
domain SPLNODE SplittingNode	2510
SplittingNode (SPLNODE)	2512
domain SPLTREE SplittingTree	2515
SplittingTree (SPLTREE)	2518
domain SREGSET SquareFreeRegularTriangularSet	2525
SquareFreeRegularTriangularSet (SREGSET)	2534
domain SQMATRIX SquareMatrix	2544
SquareMatrix (SQMATRIX)	2547
domain STACK Stack	2551
Stack (STACK)	2563
domain SD StochasticDifferential	2567
StochasticDifferential (SD)	2572
domain STREAM Stream	2577

Stream (STREAM)	2581
domain STRING String	2596
String (STRING)	2606
domain STRTBL StringTable	2608
StringTable (STRTBL)	2610
domain SUBSPACE SubSpace	2611
SubSpace (SUBSPACE)	2614
domain COMPPROP SubSpaceComponentProperty	2623
SubSpaceComponentProperty (COMPPROP)	2624
domain SUCH SuchThat	2626
SuchThat (SUCH)	2627
domain SWITCH Switch	2628
Switch (SWITCH)	2629
domain SYMBOL Symbol	2632
Symbol (SYMBOL)	2640
domain SYMTAB SymbolTable	2646
SymbolTable (SYMTAB)	2648
domain SYMPOLY SymmetricPolynomial	2652
SymmetricPolynomial (SYMPOLY)	2654

21 Chapter T**2657**

domain TABLE Table	2657
Table (TABLE)	2663
domain TABLEAU Tableau	2665
Tableau (TABLEAU)	2666
domain TS TaylorSeries	2667
TaylorSeries (TS)	2670
domain TEX TexFormat	2672
product(product(i*j,i=a..b),j=c..d) fix	2672
TexFormat (TEX)	2676
domain TEXTFILE TextFile	2688
TextFile (TEXTFILE)	2692
domain SYMS TheSymbolTable	2694
TheSymbolTable (SYMS)	2696
domain M3D ThreeDimensionalMatrix	2700
ThreeDimensionalMatrix (M3D)	2702
domain VIEW3D ThreeDimensionalViewport	2708
ThreeDimensionalViewport (VIEW3D)	2710
domain SPACE3 ThreeSpace	2729
ThreeSpace (SPACE3)	2731
domain TREE Tree	2739
Tree (TREE)	2740
domain TUBE TubePlot	2748
TubePlot (TUBE)	2749
domain TUPLE Tuple	2751
Tuple (TUPLE)	2752

domain ARRAY2 TwoDimensionalArray	2753
TwoDimensionalArray (ARRAY2)	2763
domain VIEW2D TwoDimensionalViewport	2764
TwoDimensionalViewport (VIEW2D)	2769
22 Chapter U	2785
domain UFPS UnivariateFormalPowerSeries	2785
UnivariateFormalPowerSeries (UFPS)	2788
domain ULS UnivariateLaurentSeries	2790
UnivariateLaurentSeries (ULS)	2795
domain ULSCONS UnivariateLaurentSeriesConstructor	2798
UnivariateLaurentSeriesConstructor (ULSCONS)	2803
domain UP UnivariatePolynomial	2814
UnivariatePolynomial (UP)	2827
domain UPXS UnivariatePuisseuxSeries	2829
UnivariatePuisseuxSeries (UPXS)	2833
domain UPXSCONS UnivariatePuisseuxSeriesConstructor	2837
UnivariatePuisseuxSeriesConstructor (UPXSCONS)	2841
domain UPXSING UnivariatePuisseuxSeriesWithExponentialSingularity	2849
UnivariatePuisseuxSeriesWithExponentialSingularity (UPXSING)	2852
domain OREUP UnivariateSkewPolynomial	2858
UnivariateSkewPolynomial (OREUP)	2872
domain UTS UnivariateTaylorSeries	2873
UnivariateTaylorSeries (UTS)	2876
domain UTSZ UnivariateTaylorSeriesCZero	2883
UnivariateTaylorSeriesCZero (UTSZ)	2886
domain UNISEG UniversalSegment	2892
UniversalSegment (UNISEG)	2896
domain U8MAT U8Matrix	2899
U8Matrix (U8MAT)	2901
domain U16MAT U16Matrix	2903
U16Matrix (U16MAT)	2906
domain U32MAT U32Matrix	2908
U32Matrix (U32MAT)	2910
domain U8VEC U8Vector	2912
U8Vector (U8VEC)	2917
domain U16VEC U16Vector	2919
U16Vector (U16VEC)	2924
domain U32VEC U32Vector	2926
U32Vector (U32VEC)	2931
23 Chapter V	2935
domain VARIABLE Variable	2935
Variable (VARIABLE)	2936
domain VECTOR Vector	2937
Vector (VECTOR)	2941

domain VOID Void	2943
Void (VOID)	2945
24 Chapter W	2947
domain WP WeightedPolynomials	2947
WeightedPolynomials (WP)	2949
domain WUTSET WuWenTsunTriangularSet	2951
WuWenTsunTriangularSet (WUTSET)	2958
25 Chapter X	2965
domain XDPOLY XDistributedPolynomial	2965
XDistributedPolynomial (XPOLY)	2967
domain XPBWPOLY XPBWPolynomial	2970
XPBWPolynomial (XPBWPOLY)	2987
domain XPOLY XPolynomial	2992
XPolynomial (XPOLY)	2998
domain XPR XPolynomialRing	2999
XPolynomialRing (XPR)	3007
domain XRPOLY XRecursivePolynomial	3011
XRecursivePolynomial (XRPOLY)	3013
26 Chapter Y	3021
27 Chapter Z	3023
28 The bootstrap code	3025
BOOLEAN.lsp	3025
CHAR.lsp BOOTSTRAP	3030
DFLOAT.lsp BOOTSTRAP	3034
ILIST.lsp BOOTSTRAP	3050
INT.lsp BOOTSTRAP	3062
ISTRING.lsp BOOTSTRAP	3073
LIST.lsp BOOTSTRAP	3091
NNI.lsp BOOTSTRAP	3097
OUTFORM.lsp BOOTSTRAP	3100
PI.lsp BOOTSTRAP	3114
PRIMARR.lsp BOOTSTRAP	3116
REF.lsp BOOTSTRAP	3119
SINT.lsp BOOTSTRAP	3122
SYMBOL.lsp BOOTSTRAP	3135
VECTOR.lsp BOOTSTRAP	3151
29 Chunk collections	3155
30 Index	3165

New Foreword

On October 1, 2001 Axiom was withdrawn from the market and ended life as a commercial product. On September 3, 2002 Axiom was released under the Modified BSD license, including this document. On August 27, 2003 Axiom was released as free and open source software available for download from the Free Software Foundation's website, Savannah.

Work on Axiom has had the generous support of the Center for Algorithms and Interactive Scientific Computation (CAISS) at City College of New York. Special thanks go to Dr. Gilbert Baumslag for his support of the long term goal.

The online version of this documentation is roughly 1000 pages. In order to make printed versions we've broken it up into three volumes. The first volume is tutorial in nature. The second volume is for programmers. The third volume is reference material. We've also added a fourth volume for developers. All of these changes represent an experiment in print-on-demand delivery of documentation. Time will tell whether the experiment succeeded.

Axiom has been in existence for over thirty years. It is estimated to contain about three hundred man-years of research and has, as of September 3, 2003, 143 people listed in the credits. All of these people have contributed directly or indirectly to making Axiom available. Axiom is being passed to the next generation. I'm looking forward to future milestones.

With that in mind I've introduced the theme of the "30 year horizon". We must invent the tools that support the Computational Mathematician working 30 years from now. How will research be done when every bit of mathematical knowledge is online and instantly available? What happens when we scale Axiom by a factor of 100, giving us 1.1 million domains? How can we integrate theory with code? How will we integrate theorems and proofs of the mathematics with space-time complexity proofs and running code? What visualization tools are needed? How do we support the conceptual structures and semantics of mathematics in effective ways? How do we support results from the sciences? How do we teach the next generation to be effective Computational Mathematicians?

The "30 year horizon" is much nearer than it appears.

Tim Daly
CAISS, City College of New York
November 10, 2003 ((iHy))

Chapter 1

Chapter Overview

This book contains the domains in Axiom, in alphabetical order.

Each domain has an associated 'dotpic' chunk which only lists the domains, categories, and packages that are in the layer immediately below in the build order. For the full list see the algebra Makefile where this information is maintained.

Each domain is preceded by a picture. The picture indicates several things. The colors indicate whether the name refers to a category, domain, or package. An ellipse means that the name refers to something in the bootstrap set. Thus,



Chapter 2

Chapter A

domain AFFPL AffinePlane

— AffinePlane.input —

```
)set break resume
)sys rm -f AffinePlane.output
)spool AffinePlane.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AffinePlane
--R
--R AffinePlane(K: Field) is a domain constructor
--R Abbreviation for AffinePlane is AFFPL
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for AFFPL
--R
--R----- Operations -----
--R ?=? : (%,%) -> Boolean          affinePoint : List(K) -> %
--R coerce : List(K) -> %          coerce : % -> List(K)
--R coerce : % -> OutputForm       conjugate : % -> %
--R definingField : % -> K         degree : % -> PositiveInteger
--R ?.? : (%,Integer) -> K         hash : % -> SingleInteger
--R latex : % -> String            list : % -> List(K)
--R orbit : % -> List(%)           origin : () -> %
--R pointValue : % -> List(K)      rational? : % -> Boolean
--R setelt : (%,Integer,K) -> K    ?~=? : (%,%) -> Boolean
--R conjugate : (%,NonNegativeInteger) -> %
--R orbit : (%,NonNegativeInteger) -> List(%)
```

```

--R rational? : (% , NonNegativeInteger) -> Boolean
--R removeConjugate : List(%) -> List(%)
--R removeConjugate : (List(%) , NonNegativeInteger) -> List(%)
--R
--E 1

)spool
)lisp (bye)

```

— AffinePlane.help —

```

=====
AffinePlane examples
=====

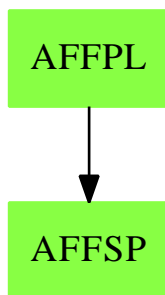
```

The related to projective space and part of the PAFF package

See Also:

- o)show AffinePlane

AffinePlane (AFFPL)



Exports:

— domain AFFPL AffinePlane —

```

)abbrev domain AFFPL AffinePlane

```

DOMAIN AFFPLPS AFFINEPLANEOVERPSEUDOALGEBRAICCLOSUREOFFINITEFIELD5

```
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is all the categories and domains related to projective
++ space and part of the PAFF package
AffinePlane(K):Exports == Implementation where
  K:Field

Exports ==> AffineSpaceCategory(K)

Implementation ==> AffineSpace(2,K)
```

— AFFPL.dotabb —

```
"AFFPL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=AFFPL"];
"AFFSP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=AFFSP"];
"AFFPL" -> "AFFSP"
```

domain AFFPLPS AffinePlaneOverPseudoAlgebraicClosureOfFiniteField

— AffinePlaneOverPseudoAlgebraicClosureOfFiniteField.input —

```
)set break resume
)sys rm -f AffinePlaneOverPseudoAlgebraicClosureOfFiniteField.output
)spool AffinePlaneOverPseudoAlgebraicClosureOfFiniteField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AffinePlaneOverPseudoAlgebraicClosureOfFiniteField
--R
--R AffinePlaneOverPseudoAlgebraicClosureOfFiniteField(K: FiniteFieldCategory) is a domain constructor
--R Abbreviation for AffinePlaneOverPseudoAlgebraicClosureOfFiniteField is AFFPLPS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for AFFPLPS
```



```

--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R conjugate : % -> %                degree : % -> PositiveInteger
--R hash : % -> SingleInteger         latex : % -> String
--R orbit : % -> List(%)              origin : () -> %
--R rational? : % -> Boolean          ?~=? : (%,% ) -> Boolean
--R affinePoint : List(PseudoAlgebraicClosureOfFiniteField(K)) -> %
--R coerce : List(PseudoAlgebraicClosureOfFiniteField(K)) -> %
--R coerce : % -> List(PseudoAlgebraicClosureOfFiniteField(K))
--R conjugate : (% ,NonNegativeInteger) -> %
--R definingField : % -> PseudoAlgebraicClosureOfFiniteField(K)
--R ?.? : (% ,Integer) -> PseudoAlgebraicClosureOfFiniteField(K)
--R list : % -> List(PseudoAlgebraicClosureOfFiniteField(K))
--R orbit : (% ,NonNegativeInteger) -> List(%)
--R pointValue : % -> List(PseudoAlgebraicClosureOfFiniteField(K))
--R rational? : (% ,NonNegativeInteger) -> Boolean
--R removeConjugate : List(%) -> List(%)
--R removeConjugate : (List(% ),NonNegativeInteger) -> List(%)
--R setelt : (% ,Integer,PseudoAlgebraicClosureOfFiniteField(K)) -> PseudoAlgebraicClosureOfFiniteField(K)
--R
--E 1

)spool
)lisp (bye)

```

— AffinePlaneOverPseudoAlgebraicClosureOfFiniteField.help —

```

=====
AffinePlaneOverPseudoAlgebraicClosureOfFiniteField examples
=====

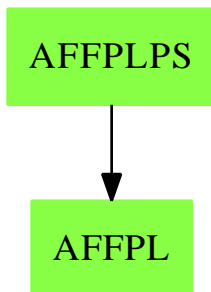
```

The related to projective space and part of the PAFF package

See Also:

o)show AffinePlaneOverPseudoAlgebraicClosureOfFiniteField

AffinePlaneOverPseudoAlgebraicClosureOfFiniteField (AFFPLPS)



Exports:

??	?=?	?~=?	affinePoint	coerce
conjugate	definingField	degree	hash	latex
list	orbit	origin	pointValue	rational?
removeConjugate	setelt			

— domain AFFPLPS AffinePlaneOverPseudoAlgebraicClosureOfFiniteField —

```

)abbrev domain AFFPLPS AffinePlaneOverPseudoAlgebraicClosureOfFiniteField
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is all the categories and domains related to projective
++ space and part of the PAFF package
AffinePlaneOverPseudoAlgebraicClosureOfFiniteField(K):Exports == Impl where
  K:FiniteFieldCategory

  KK ==> PseudoAlgebraicClosureOfFiniteField(K)

  Exports ==> AffineSpaceCategory(KK)

  Impl ==> AffinePlane(KK)
  
```

— AFFPLPS.dotabb —

```

"AFFPLPS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=AFFPLPS"];
"AFFPL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=AFFPL"];
"AFFPLPS" -> "AFFPL"
  
```

domain AFFSP AffineSpace

— AffineSpace.input —

```
)set break resume
)sys rm -f AffineSpace.output
)spool AffineSpace.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AffineSpace
--R
--R AffineSpace(dim: NonNegativeInteger,K: Field) is a domain constructor
--R Abbreviation for AffineSpace is AFFSP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for AFFSP
--R
--R----- Operations -----
--R ==? : (%,% ) -> Boolean          affinePoint : List(K) -> %
--R coerce : List(K) -> %           coerce : % -> List(K)
--R coerce : % -> OutputForm        conjugate : % -> %
--R definingField : % -> K          degree : % -> PositiveInteger
--R ?? : (%,Integer) -> K           hash : % -> SingleInteger
--R latex : % -> String             list : % -> List(K)
--R orbit : % -> List(%)            origin : () -> %
--R pointValue : % -> List(K)        rational? : % -> Boolean
--R setelt : (%,Integer,K) -> K      ?~=? : (%,% ) -> Boolean
--R conjugate : (%NonNegativeInteger) -> %
--R orbit : (%NonNegativeInteger) -> List(%)
--R rational? : (%NonNegativeInteger) -> Boolean
--R removeConjugate : List(%) -> List(%)
--R removeConjugate : (List(%),NonNegativeInteger) -> List(%)
--R
--E 1

)spool
)lisp (bye)
```

— AffineSpace.help —

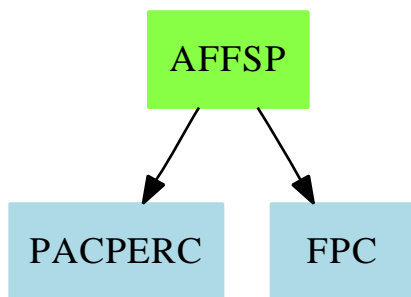
```
=====
AffineSpace examples
=====
```

This is rrelated to projective space and part of the PAFF package

See Also:

o)show AffineSpace

AffineSpace (AFFSP)



Exports:

??	?=?	?~=?	affinePoint	coerce
conjugate	definingField	degree	hash	latex
list	orbit	origin	pointValue	rational?
removeConjugate	setelt			

— domain AFFSP AffineSpace —

```

)abbrev domain AFFSP AffineSpace
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is all the categories and domains related to projective
++ space and part of the PAFF package
AffineSpace(dim,K):Exports == Implementation where
  dim:NonNegativeInteger
  K:Field

LIST ==> List
NNI ==> NonNegativeInteger

Exports ==> AffineSpaceCategory(K)

Implementation ==> List(K) add

```

```

Rep:= List(K)

origin== new(dim,0$K)$List(K)

coerce(pt:%):OutputForm ==
  dd:OutputForm:= ":" :: OutputForm
  llout:List(OutputForm):=[ hconcat(dd, a::OutputForm) for a in rest pt]
  lout:= cons( (first pt)::OutputForm , llout)
  out:= hconcat lout
  oo:=paren(out)
  ee:OutputForm:= degree(pt) :: OutputForm
  oo**ee

definingField(pt)==
  K has PseudoAlgebraicClosureOfPerfectFieldCategory => _
    maxTower(pt@Rep)
  1$K

degree(pt)==
  K has PseudoAlgebraicClosureOfPerfectFieldCategory => _
    extDegree definingField pt
  1

coerce(pt:%):List(K) == pt@Rep

affinePoint(pt:LIST(K))==
  pt :: %

list(ptt)==
  ptt@Rep

pointValue(ptt)==
  ptt@Rep

conjugate(p,e)==
  lp:Rep:=p
  pc:List(K):=[c**e for c in lp]
  affinePoint(pc)

rational?(p,n)== p=conjugate(p,n)

rational?(p)==rational?(p,characteristic()$K)

removeConjugate(l)==removeConjugate(l,characteristic()$K)

removeConjugate(l:LIST(%),n:NNI):LIST(%)==
  if K has FiniteFieldCategory then
    allconj:LIST(%):=empty()
    conjrem:LIST(%):=empty()

```

```

    for p in l repeat
      if ^member?(p,allconj) then
        conjrem:=cons(p,conjrem)
        allconj:=concat(allconj,orbit(p,n))
      conjrem
    else
      error "The field is not finite"

conjugate(p)==conjugate(p,characteristic())$K

orbit(p)==orbit(p,characteristic())$K

orbit(p,e)==
  if K has FiniteFieldCategory then
    l:LIST(%):=[p]
    np:%=conjugate(p,e)
    flag:=^(np=p)::Boolean
    while flag repeat
      l:=concat(np,l)
      np:=conjugate(np,e)
      flag:=not (np=p)::Boolean
    l
  else
    error "Cannot compute the conjugate"

aa:% = bb:% ==
  aa =$Rep bb

coerce(pt:LIST(K))==
  ^(dim=#pt) => error "Le point n'a pas la bonne dimension"
  ptt:%= pt
  ptt

```

— AFFSP.dotabb —

```

"AFFSP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=AFFSP"];
"PACPERC" [color=lightblue,href="bookvol10.2.pdf#nameddest=PACPERC"];
"FPC" [color=lightblue,href="bookvol10.2.pdf#nameddest=FPC"];
"AFFSP" -> "FPC"
"AFFSP" -> "PACPERC"

```

domain ALGSC AlgebraGivenByStructuralConstants

— AlgebraGivenByStructuralConstants.input —

```

)set break resume
)sys rm -f AlgebraGivenByStructuralConstants.output
)spool AlgebraGivenByStructuralConstants.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AlgebraGivenByStructuralConstants
--R
--R AlgebraGivenByStructuralConstants(R: Field,n: PositiveInteger,ls: List(Symbol),gamma: Vector
--R Abbreviation for AlgebraGivenByStructuralConstants is ALGSC
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ALGSC
--R
--R----- Operations -----
--R ???: (SquareMatrix(n,R),%) -> %      ??? : (R,%) -> %
--R ??? : (%,R) -> %                      ??? : (%,%) -> %
--R ??? : (Integer,%) -> %                ??? : (PositiveInteger,%) -> %
--R ??? : (%,PositiveInteger) -> %        ?+? : (%,%) -> %
--R ?-? : (%,%) -> %                      -? : % -> %
--R ?=? : (%,%) -> Boolean                 0 : () -> %
--R alternative? : () -> Boolean            antiAssociative? : () -> Boolean
--R antiCommutative? : () -> Boolean        antiCommutator : (%,%) -> %
--R apply : (Matrix(R),%) -> %              associative? : () -> Boolean
--R associator : (%,%,%) -> %               basis : () -> Vector(%)
--R coerce : Vector(R) -> %                 coerce : % -> OutputForm
--R commutative? : () -> Boolean             commutator : (%,%) -> %
--R convert : Vector(R) -> %                 convert : % -> Vector(R)
--R coordinates : % -> Vector(R)            ?.: (%,Integer) -> R
--R flexible? : () -> Boolean               hash : % -> SingleInteger
--R jacobiIdentity? : () -> Boolean          jordanAdmissible? : () -> Boolean
--R jordanAlgebra? : () -> Boolean          latex : % -> String
--R leftAlternative? : () -> Boolean         leftDiscriminant : () -> R
--R leftDiscriminant : Vector(%) -> R       leftNorm : % -> R
--R leftTrace : % -> R                     leftTraceMatrix : () -> Matrix(R)
--R lieAdmissible? : () -> Boolean          lieAlgebra? : () -> Boolean
--R powerAssociative? : () -> Boolean        rank : () -> PositiveInteger
--R represents : Vector(R) -> %             rightAlternative? : () -> Boolean
--R rightDiscriminant : () -> R             rightNorm : % -> R
--R rightTrace : % -> R                   rightTraceMatrix : () -> Matrix(R)
--R sample : () -> %                      someBasis : () -> Vector(%)
--R zero? : % -> Boolean                   ?~=? : (%,%) -> Boolean
--R ??? : (NonNegativeInteger,%) -> %

```

```

--R associatorDependence : () -> List(Vector(R)) if R has INTDOM
--R conditionsForIdempotents : () -> List(Polynomial(R))
--R conditionsForIdempotents : Vector(%) -> List(Polynomial(R))
--R coordinates : Vector(%) -> Matrix(R)
--R coordinates : (Vector(%),Vector(%)) -> Matrix(R)
--R coordinates : (% , Vector(%)) -> Vector(R)
--R leftCharacteristicPolynomial : % -> SparseUnivariatePolynomial(R)
--R leftMinimalPolynomial : % -> SparseUnivariatePolynomial(R) if R has INTDOM
--R leftPower : (% , PositiveInteger) -> %
--R leftRankPolynomial : () -> SparseUnivariatePolynomial(Polynomial(R)) if R has FIELD
--R leftRecip : % -> Union(% , "failed") if R has INTDOM
--R leftRegularRepresentation : % -> Matrix(R)
--R leftRegularRepresentation : (% , Vector(%)) -> Matrix(R)
--R leftTraceMatrix : Vector(%) -> Matrix(R)
--R leftUnit : () -> Union(% , "failed") if R has INTDOM
--R leftUnits : () -> Union(Record(particular: % , basis: List(%)) , "failed") if R has INTDOM
--R noncommutativeJordanAlgebra? : () -> Boolean
--R plenaryPower : (% , PositiveInteger) -> %
--R recip : % -> Union(% , "failed") if R has INTDOM
--R represents : (Vector(R) , Vector(%)) -> %
--R rightCharacteristicPolynomial : % -> SparseUnivariatePolynomial(R)
--R rightDiscriminant : Vector(%) -> R
--R rightMinimalPolynomial : % -> SparseUnivariatePolynomial(R) if R has INTDOM
--R rightPower : (% , PositiveInteger) -> %
--R rightRankPolynomial : () -> SparseUnivariatePolynomial(Polynomial(R)) if R has FIELD
--R rightRecip : % -> Union(% , "failed") if R has INTDOM
--R rightRegularRepresentation : % -> Matrix(R)
--R rightRegularRepresentation : (% , Vector(%)) -> Matrix(R)
--R rightTraceMatrix : Vector(%) -> Matrix(R)
--R rightUnit : () -> Union(% , "failed") if R has INTDOM
--R rightUnits : () -> Union(Record(particular: % , basis: List(%)) , "failed") if R has INTDOM
--R structuralConstants : () -> Vector(Matrix(R))
--R structuralConstants : Vector(%) -> Vector(Matrix(R))
--R subtractIfCan : (% , %) -> Union(% , "failed")
--R unit : () -> Union(% , "failed") if R has INTDOM
--R
--E 1

)spool
)lisp (bye)

```

— AlgebraGivenByStructuralConstants.help —

```

=====
AlgebraGivenByStructuralConstants examples
=====

```


`AlgebraGivenByStructuralConstants` implements finite rank algebras over a commutative ring, given by the structural constants `gamma` with respect to a fixed basis `[a1,...,an]`, where `gamma` is an `n`-vector of `n` by `n` matrices `[(gamma_ijk) for k in 1..rank()]` defined by

$$a_i * a_j = \text{gamma}_{ij1} * a_1 + \dots + \text{gamma}_{ijn} * a_n$$

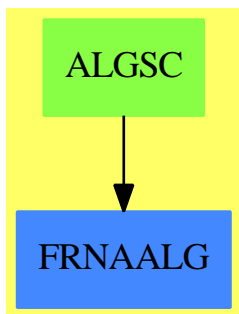
The symbols for the fixed basis have to be given as a list of symbols.

See Also:

o `)show AlgebraGivenByStructuralConstants`

—————→

AlgebraGivenByStructuralConstants (ALGSC)



Exports:

0	alternative?
antiAssociative?	antiCommutative?
antiCommutator	apply
associative?	associator
associatorDependence	basis
coerce	commutative?
commutator	conditionsForIdempotents
convert	coordinates
flexible?	hash
jacobiIdentity?	jordanAdmissible?
jordanAlgebra?	latex
leftAlternative?	leftCharacteristicPolynomial
leftDiscriminant	leftMinimalPolynomial
leftNorm	leftPower
leftRankPolynomial	leftRecip
leftRegularRepresentation	leftTrace
leftTraceMatrix	leftUnit
leftUnits	lieAdmissible?
lieAlgebra?	noncommutativeJordanAlgebra?
plenaryPower	powerAssociative?
rank	recip
represents	rightAlternative?
rightCharacteristicPolynomial	rightDiscriminant
rightMinimalPolynomial	rightNorm
rightPower	rightRankPolynomial
rightRecip	rightRegularRepresentation
rightTrace	rightTraceMatrix
rightUnit	rightUnits
sample	someBasis
structuralConstants	subtractIfCan
unit	zero?
?*?	?**?
?+?	?-?
-?	?=?
?..?	?~=?
?*?	

— domain ALGSC AlgebraGivenByStructuralConstants —

```

)abbrev domain ALGSC AlgebraGivenByStructuralConstants
++ Authors: J. Grabmeier, R. Wisbauer
++ Date Created: 01 March 1991
++ Date Last Updated: 22 January 1992
++ Reference:
++   R.D. Schafer: An Introduction to Nonassociative Algebras
++   Academic Press, New York, 1966
++ Description:

```

```

++ AlgebraGivenByStructuralConstants implements finite rank algebras
++ over a commutative ring, given by the structural constants \spad{gamma}
++ with respect to a fixed basis \spad{[a1,...,an]}, where
++ \spad{gamma} is an \spad{n}-vector of n by n matrices
++ \spad{[(gammaijk) for k in 1..rank()] } defined by
++ \spad{ai * aj = gammaij1 * a1 + ... + gammaijn * an}.
++ The symbols for the fixed basis
++ have to be given as a list of symbols.

```

```

AlgebraGivenByStructuralConstants(R:Field, n : PositiveInteger, _
  ls : List Symbol, gamma: Vector Matrix R ): public == private where

```

```

V ==> Vector
M ==> Matrix
I ==> Integer
NNI ==> NonNegativeInteger
REC ==> Record(particular: Union(V R,"failed"),basis: List V R)
LSMP ==> LinearSystemMatrixPackage(R,V R,V R, M R)

```

```

--public ==> FramedNonAssociativeAlgebra(R) with
public ==> Join(FramedNonAssociativeAlgebra(R), _
  LeftModule(SquareMatrix(n,R)) ) with

```

```

coerce : Vector R -> %
++ coerce(v) converts a vector to a member of the algebra
++ by forming a linear combination with the basis element.
++ Note: the vector is assumed to have length equal to the
++ dimension of the algebra.

```

```

private ==> DirectProduct(n,R) add

```

```

Rep := DirectProduct(n,R)

```

```

x,y : %
dp : DirectProduct(n,R)
v : V R

```

```

recip(x) == recip(x)$FiniteRankNonAssociativeAlgebra_&(% ,R)

```

```

(m:SquareMatrix(n,R))*(x:%) == apply((m :: Matrix R),x)
coerce v == directProduct(v) :: %

```

```

structuralConstants() == gamma

```

```

coordinates(x) == vector(entries(x :: Rep)$Rep)$Vector(R)

```

```

coordinates(x,b) ==
--not (maxIndex b = n) =>
-- error("coordinates: your 'basis' has not the right length")

```

```

m : NonNegativeInteger := (maxIndex b) :: NonNegativeInteger
transitionMatrix : Matrix R := new(n,m,0$R)$Matrix(R)
for i in 1..m repeat
  setColumn_!(transitionMatrix,i,coordinates(b.i))
res : REC := solve(transitionMatrix,coordinates(x))$LSMP
if (not every?(zero?$R,first res.basis)) then
  error("coordinates: warning your 'basis' is linearly dependent")
(res.particular case "failed") =>
  error("coordinates: first argument is not in linear span of second argument")
(res.particular) :: (Vector R)

basis() == [unitVector(i::PositiveInteger)::% for i in 1..n]

someBasis() == basis()$%

rank() == n

elt(x,i) == elt(x:Rep,i)$Rep

coerce(x:%):OutputForm ==
  zero?(x::Rep)$Rep => (0$R) :: OutputForm
  le : List OutputForm := nil
  for i in 1..n repeat
    coef : R := elt(x::Rep,i)
    not zero?(coef)$R =>
--      one?(coef)$R =>
        ((coef) = 1)$R =>
          -- sy : OutputForm := elt(ls,i)$(List Symbol) :: OutputForm
          le := cons(elt(ls,i)$(List Symbol) :: OutputForm, le)
          le := cons(coef :: OutputForm * elt(ls,i)$(List Symbol)_
            :: OutputForm, le)
  reduce("+",le)

x * y ==
  v : Vector R := new(n,0)
  for k in 1..n repeat
    h : R := 0
    for i in 1..n repeat
      for j in 1..n repeat
        h := h +$R elt(x,i) *$R elt(y,j) *$R elt(gamma.k,i,j )
    v.k := h
  directProduct v

alternative?() ==
  for i in 1..n repeat
    -- expression for check of left alternative is symmetric in i and j:
    -- expression for check of right alternative is symmetric in j and k:
    for j in 1..i-1 repeat

```

```

for k in j..n repeat
-- right check
for r in 1..n repeat
  res := 0$R
  for l in 1..n repeat
    res := res - _
      (elt(gamma.l,j,k)+elt(gamma.l,k,j))*elt(gamma.r,i,l)+_
      (elt(gamma.l,i,j)*elt(gamma.r,l,k) + elt(gamma.l,i,k)*_
      elt(gamma.r,l,j) )
  not zero? res =>
    messagePrint("algebra is not right alternative")$OutputForm
  return false
for j in i..n repeat
  for k in 1..j-1 repeat
    -- left check
    for r in 1..n repeat
      res := 0$R
      for l in 1..n repeat
        res := res + _
          (elt(gamma.l,i,j)+elt(gamma.l,j,i))*elt(gamma.r,l,k)-_
          (elt(gamma.l,j,k)*elt(gamma.r,i,l) + elt(gamma.l,i,k)*_
          elt(gamma.r,j,l) )
      not (zero? res) =>
        messagePrint("algebra is not left alternative")$OutputForm
      return false

for k in j..n repeat
-- left check
for r in 1..n repeat
  res := 0$R
  for l in 1..n repeat
    res := res + _
      (elt(gamma.l,i,j)+elt(gamma.l,j,i))*elt(gamma.r,l,k)-_
      (elt(gamma.l,j,k)*elt(gamma.r,i,l) + elt(gamma.l,i,k)*_
      elt(gamma.r,j,l) )
  not (zero? res) =>
    messagePrint("algebra is not left alternative")$OutputForm
  return false
-- right check
for r in 1..n repeat
  res := 0$R
  for l in 1..n repeat
    res := res - _
      (elt(gamma.l,j,k)+elt(gamma.l,k,j))*elt(gamma.r,i,l)+_
      (elt(gamma.l,i,j)*elt(gamma.r,l,k) + elt(gamma.l,i,k)*_
      elt(gamma.r,l,j) )
  not (zero? res) =>
    messagePrint("algebra is not right alternative")$OutputForm
  return false

```

```

messagePrint("algebra satisfies 2*associator(a,b,b) = 0 = 2*associator(a,a,b) = 0")$OutputForm
true

-- should be in the category, but is not exported
-- conditionsForIdempotents b ==
--   n := rank()
--   --gamma : Vector Matrix R := structuralConstants b
--   listOfNumbers : List String := [STRINGIMAGE(q)$Lisp for q in 1..n]
--   symbolsForCoef : Vector Symbol :=
--     [concat("%", concat("x", i))::Symbol for i in listOfNumbers]
--   conditions : List Polynomial R := []
--   for k in 1..n repeat
--     xk := symbolsForCoef.k
--     p : Polynomial R := monomial( - 1$Polynomial(R), [xk], [1] )
--     for i in 1..n repeat
--       for j in 1..n repeat
--         xi := symbolsForCoef.i
--         xj := symbolsForCoef.j
--         p := p + monomial(
--           elt((gamma.k),i,j) :: Polynomial(R), [xi,xj], [1,1])
--         conditions := cons(p,conditions)
--   conditions

associative?() ==
  for i in 1..n repeat
    for j in 1..n repeat
      for k in 1..n repeat
        for r in 1..n repeat
          res := 0$R
          for l in 1..n repeat
            res := res + elt(gamma.l,i,j)*elt(gamma.r,l,k)-_
              elt(gamma.l,j,k)*elt(gamma.r,i,l)
          not (zero? res) =>
            messagePrint("algebra is not associative")$OutputForm
            return false
  messagePrint("algebra is associative")$OutputForm
  true

antiAssociative?() ==
  for i in 1..n repeat
    for j in 1..n repeat
      for k in 1..n repeat
        for r in 1..n repeat
          res := 0$R
          for l in 1..n repeat
            res := res + elt(gamma.l,i,j)*elt(gamma.r,l,k)+_
              elt(gamma.l,j,k)*elt(gamma.r,i,l)
          not (zero? res) =>
            messagePrint("algebra is not anti-associative")$OutputForm

```

```

        return false
    messagePrint("algebra is anti-associative")$OutputForm
    true

commutative?() ==
    for i in 1..n repeat
        for j in (i+1)..n repeat
            for k in 1..n repeat
                not ( elt(gamma.k,i,j)=elt(gamma.k,j,i) ) =>
                    messagePrint("algebra is not commutative")$OutputForm
                    return false
    messagePrint("algebra is commutative")$OutputForm
    true

antiCommutative?() ==
    for i in 1..n repeat
        for j in i..n repeat
            for k in 1..n repeat
                not zero? (i=j => elt(gamma.k,i,i); elt(gamma.k,i,j)+elt(gamma.k,j,i) ) =>
                    messagePrint("algebra is not anti-commutative")$OutputForm
                    return false
    messagePrint("algebra is anti-commutative")$OutputForm
    true

leftAlternative?() ==
    for i in 1..n repeat
        -- expression is symmetric in i and j:
        for j in i..n repeat
            for k in 1..n repeat
                for r in 1..n repeat
                    res := 0$R
                    for l in 1..n repeat
                        res := res + (elt(gamma.l,i,j)+elt(gamma.l,j,i))*elt(gamma.r,l,k)-_
                            (elt(gamma.l,j,k)*elt(gamma.r,i,l) + elt(gamma.l,i,k)*elt(gamma.r,j,l) )
                    not (zero? res) =>
                        messagePrint("algebra is not left alternative")$OutputForm
                        return false
    messagePrint("algebra is left alternative")$OutputForm
    true

rightAlternative?() ==
    for i in 1..n repeat
        for j in 1..n repeat
            -- expression is symmetric in j and k:
            for k in j..n repeat
                for r in 1..n repeat
                    res := 0$R
                    for l in 1..n repeat
                        res := res - (elt(gamma.l,j,k)+elt(gamma.l,k,j))*elt(gamma.r,i,l)+_

```

```

        (elt(gamma.l,i,j)*elt(gamma.r,l,k) + elt(gamma.l,i,k)*elt(gamma.r,l,j) )
    not (zero? res) =>
        messagePrint("algebra is not right alternative")$OutputForm
        return false
    messagePrint("algebra is right alternative")$OutputForm
    true

flexible?() ==
    for i in 1..n repeat
        for j in 1..n repeat
            -- expression is symmetric in i and k:
            for k in 1..n repeat
                for r in 1..n repeat
                    res := 0$R
                    for l in 1..n repeat
                        res := res + elt(gamma.l,i,j)*elt(gamma.r,l,k)-_
                            elt(gamma.l,j,k)*elt(gamma.r,i,l)+_
                            elt(gamma.l,k,j)*elt(gamma.r,l,i)-_
                            elt(gamma.l,j,i)*elt(gamma.r,k,l)
                    not (zero? res) =>
                        messagePrint("algebra is not flexible")$OutputForm
                        return false
    messagePrint("algebra is flexible")$OutputForm
    true

lieAdmissible?() ==
    for i in 1..n repeat
        for j in 1..n repeat
            for k in 1..n repeat
                for r in 1..n repeat
                    res := 0$R
                    for l in 1..n repeat
                        res := res_
                            + (elt(gamma.l,i,j)-elt(gamma.l,j,i))*(elt(gamma.r,l,k)-elt(gamma.r,k,l)) _
                            + (elt(gamma.l,j,k)-elt(gamma.l,k,j))*(elt(gamma.r,l,i)-elt(gamma.r,i,l)) _
                            + (elt(gamma.l,k,i)-elt(gamma.l,i,k))*(elt(gamma.r,l,j)-elt(gamma.r,j,l))
                    not (zero? res) =>
                        messagePrint("algebra is not Lie admissible")$OutputForm
                        return false
    messagePrint("algebra is Lie admissible")$OutputForm
    true

jordanAdmissible?() ==
    recip(2 * 1$R) case "failed" =>
        messagePrint("this algebra is not Jordan admissible, as 2 is not invertible in the ground ring")$OutputForm
        false
    for i in 1..n repeat
        for j in 1..n repeat
            for k in 1..n repeat

```



```

for w in 1..n repeat
  for t in 1..n repeat
    res := 0$R
    for l in 1..n repeat
      for r in 1..n repeat
        res := res_
          + (elt(gamma.l,i,j)+elt(gamma.l,j,i))_
            * (elt(gamma.r,w,k)+elt(gamma.r,k,w))_
            * (elt(gamma.t,l,r)+elt(gamma.t,r,l))_
          - (elt(gamma.r,w,k)+elt(gamma.r,k,w))_
            * (elt(gamma.l,j,r)+elt(gamma.l,r,j))_
            * (elt(gamma.t,i,l)+elt(gamma.t,l,i))_
          + (elt(gamma.l,w,j)+elt(gamma.l,j,w))_
            * (elt(gamma.r,k,i)+elt(gamma.r,i,k))_
            * (elt(gamma.t,l,r)+elt(gamma.t,r,l))_
          - (elt(gamma.r,k,i)+elt(gamma.r,k,i))_
            * (elt(gamma.l,j,r)+elt(gamma.l,r,j))_
            * (elt(gamma.t,w,l)+elt(gamma.t,l,w))_
          + (elt(gamma.l,k,j)+elt(gamma.l,j,k))_
            * (elt(gamma.r,i,w)+elt(gamma.r,w,i))_
            * (elt(gamma.t,l,r)+elt(gamma.t,r,l))_
          - (elt(gamma.r,i,w)+elt(gamma.r,w,i))_
            * (elt(gamma.l,j,r)+elt(gamma.l,r,j))_
            * (elt(gamma.t,k,l)+elt(gamma.t,l,k))
        not (zero? res) =>
          messagePrint("algebra is not Jordan admissible")$OutputForm
          return false
      messagePrint("algebra is Jordan admissible")$OutputForm
    true

jordanAlgebra?() ==
  recip(2 * 1$R) case "failed" =>
    messagePrint("this is not a Jordan algebra, as 2 is not invertible in the ground ring")$
    false
  not commutative?() =>
    messagePrint("this is not a Jordan algebra")$OutputForm
    false
for i in 1..n repeat
  for j in 1..n repeat
    for k in 1..n repeat
      for l in 1..n repeat
        for t in 1..n repeat
          res := 0$R
          for r in 1..n repeat
            for s in 1..n repeat
              res := res + _
                elt(gamma.r,i,j)*elt(gamma.s,l,k)*elt(gamma.t,r,s) - _
                elt(gamma.r,l,k)*elt(gamma.s,j,r)*elt(gamma.t,i,s) + _
                elt(gamma.r,l,j)*elt(gamma.s,k,k)*elt(gamma.t,r,s) - _
                elt(gamma.r,k,i)*elt(gamma.s,j,r)*elt(gamma.t,l,s) + _

```

```

        elt(gamma.r,k,j)*elt(gamma.s,i,k)*elt(gamma.t,r,s) - _
        elt(gamma.r,i,l)*elt(gamma.s,j,r)*elt(gamma.t,k,s)
    not zero? res =>
        messagePrint("this is not a Jordan algebra")$OutputForm
        return false
    messagePrint("this is a Jordan algebra")$OutputForm
    true

jacobiIdentity?() ==
    for i in 1..n repeat
        for j in 1..n repeat
            for k in 1..n repeat
                for r in 1..n repeat
                    res := 0$R
                    for s in 1..n repeat
                        res := res + elt(gamma.r,i,j)*elt(gamma.s,j,k) +_
                        elt(gamma.r,j,k)*elt(gamma.s,k,i) +_
                        elt(gamma.r,k,i)*elt(gamma.s,i,j)
                    not zero? res =>
                        messagePrint("Jacobi identity does not hold")$OutputForm
                        return false
                messagePrint("Jacobi identity holds")$OutputForm
            true

```

— ALGSC.dotabb —

"ALGSC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALGSC"]
 "FRNAALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRNAALG"]
 "ALGSC" -> "FRNAALG"

domain ALGFF AlgebraicFunctionField

— AlgebraicFunctionField.input —

```

)set break resume
)sys rm -f AlgebraicFunctionField.output
)spool AlgebraicFunctionField.output
)set message test on
)set message auto off

```

```

)clear all

--S 1 of 1
)show AlgebraicFunctionField
--R
--R AlgebraicFunctionField(F: Field,UP: UnivariatePolynomialCategory(F),UPUP: UnivariatePolynomialCategory(F))
--R Abbreviation for AlgebraicFunctionField is ALGFF
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ALGFF
--R
--R----- Operations -----
--R ???: (Fraction(UP),%) -> %          ??: (% , Fraction(UP)) -> %
--R ??: (% , %) -> %                   ??: (Integer, %) -> %
--R ??: (PositiveInteger, %) -> %      ??: (% , PositiveInteger) -> %
--R ??: (% , %) -> %                   ??: (% , %) -> %
--R -?: % -> %                         ??: (% , %) -> Boolean
--R 1 : () -> %                        0 : () -> %
--R ??: (% , PositiveInteger) -> %     basis : () -> Vector(%)
--R branchPoint? : UP -> Boolean       branchPoint? : F -> Boolean
--R coerce : Fraction(UP) -> %         coerce : Integer -> %
--R coerce : % -> OutputForm          convert : UPUP -> %
--R convert : % -> UPUP               definingPolynomial : () -> UPUP
--R discriminant : () -> Fraction(UP)  elt : (% , F, F) -> F
--R generator : () -> %               genus : () -> NonNegativeInteger
--R hash : % -> SingleInteger          integral? : (% , UP) -> Boolean
--R integral? : (% , F) -> Boolean     integral? : % -> Boolean
--R integralBasis : () -> Vector(%)   latex : % -> String
--R lift : % -> UPUP                  norm : % -> Fraction(UP)
--R one? : % -> Boolean               primitivePart : % -> %
--R ramified? : UP -> Boolean         ramified? : F -> Boolean
--R rank : () -> PositiveInteger       rationalPoint? : (F, F) -> Boolean
--R recip : % -> Union(%, "failed")    reduce : UPUP -> %
--R represents : (Vector(UP), UP) -> % retract : % -> Fraction(UP)
--R sample : () -> %                  singular? : UP -> Boolean
--R singular? : F -> Boolean           trace : % -> Fraction(UP)
--R zero? : % -> Boolean              ?~?: (% , %) -> Boolean
--R ??: (% , Fraction(Integer)) -> % if Fraction(UP) has FIELD
--R ??: (Fraction(Integer), %) -> % if Fraction(UP) has FIELD
--R ??: (NonNegativeInteger, %) -> %
--R ??: (% , Integer) -> % if Fraction(UP) has FIELD
--R ??: (% , NonNegativeInteger) -> %
--R ?/? : (% , %) -> % if Fraction(UP) has FIELD
--R D : % -> % if Fraction(UP) has DIFRING and Fraction(UP) has FIELD or Fraction(UP) has FFIELD
--R D : (% , NonNegativeInteger) -> % if Fraction(UP) has DIFRING and Fraction(UP) has FIELD or FFIELD
--R D : (% , Symbol) -> % if Fraction(UP) has FIELD and Fraction(UP) has PDRING(SYMBOL)
--R D : (% , List(Symbol)) -> % if Fraction(UP) has FIELD and Fraction(UP) has PDRING(SYMBOL)
--R D : (% , Symbol, NonNegativeInteger) -> % if Fraction(UP) has FIELD and Fraction(UP) has PDRING(SYMBOL)
--R D : (% , List(Symbol), List(NonNegativeInteger)) -> % if Fraction(UP) has FIELD and Fraction(UP) has PDRING(SYMBOL)
--R D : (% , (Fraction(UP) -> Fraction(UP))) -> % if Fraction(UP) has FIELD
--R D : (% , (Fraction(UP) -> Fraction(UP)), NonNegativeInteger) -> % if Fraction(UP) has FIELD

```

```

--R ?? : (%,Integer) -> % if Fraction(UP) has FIELD
--R ?? : (%NonNegativeInteger) -> %
--R absolutelyIrreducible? : () -> Boolean
--R algSplitSimple : (%,(UP -> UP)) -> Record(num: %,den: UP,derivden: UP,gd: UP)
--R associates? : (%,% ) -> Boolean if Fraction(UP) has FIELD
--R branchPointAtInfinity? : () -> Boolean
--R characteristic : () -> NonNegativeInteger
--R characteristicPolynomial : % -> UPUP
--R charthRoot : % -> Union(%,"failed") if Fraction(UP) has CHARNZ
--R charthRoot : % -> % if Fraction(UP) has FFIELDC
--R coerce : % -> % if Fraction(UP) has FIELD
--R coerce : Fraction(Integer) -> % if Fraction(UP) has FIELD or Fraction(UP) has RETRACT(FRAC(INT))
--R complementaryBasis : Vector(%) -> Vector(%)
--R conditionP : Matrix(%) -> Union(Vector(%),"failed") if Fraction(UP) has FFIELDC
--R convert : Vector(Fraction(UP)) -> %
--R convert : % -> Vector(Fraction(UP))
--R coordinates : Vector(%) -> Matrix(Fraction(UP))
--R coordinates : % -> Vector(Fraction(UP))
--R coordinates : (Vector(%),Vector(%)) -> Matrix(Fraction(UP))
--R coordinates : (%Vector(%)) -> Vector(Fraction(UP))
--R createPrimitiveElement : () -> % if Fraction(UP) has FFIELDC
--R derivationCoordinates : (Vector(%),(Fraction(UP) -> Fraction(UP))) -> Matrix(Fraction(UP)) if Fraction(UP) has FIELD
--R differentiate : % -> % if Fraction(UP) has DIFRING and Fraction(UP) has FIELD or Fraction(UP) has FFIELDC
--R differentiate : (%NonNegativeInteger) -> % if Fraction(UP) has DIFRING and Fraction(UP) has FIELD or Fraction(UP) has FFIELDC
--R differentiate : (%Symbol) -> % if Fraction(UP) has FIELD and Fraction(UP) has PDRING(SYMBOL)
--R differentiate : (%List(Symbol)) -> % if Fraction(UP) has FIELD and Fraction(UP) has PDRING(SYMBOL)
--R differentiate : (%Symbol,NonNegativeInteger) -> % if Fraction(UP) has FIELD and Fraction(UP) has PDRING(SYMBOL)
--R differentiate : (%List(Symbol),List(NonNegativeInteger)) -> % if Fraction(UP) has FIELD and Fraction(UP) has PDRING(SYMBOL)
--R differentiate : (%(UP -> UP)) -> %
--R differentiate : (%(Fraction(UP) -> Fraction(UP))) -> % if Fraction(UP) has FIELD
--R differentiate : (%(Fraction(UP) -> Fraction(UP)),NonNegativeInteger) -> % if Fraction(UP) has FIELD
--R discreteLog : (%,% ) -> Union(NonNegativeInteger,"failed") if Fraction(UP) has FFIELDC
--R discreteLog : % -> NonNegativeInteger if Fraction(UP) has FFIELDC
--R discriminant : Vector(%) -> Fraction(UP)
--R divide : (%,% ) -> Record(quotient: %,remainder: %) if Fraction(UP) has FIELD
--R elliptic : () -> Union(UP,"failed")
--R enumerate : () -> List(%) if Fraction(UP) has FINITE
--R euclideanSize : % -> NonNegativeInteger if Fraction(UP) has FIELD
--R expressIdealMember : (List(%),%) -> Union(List(%),"failed") if Fraction(UP) has FIELD
--R exquo : (%,% ) -> Union(%,"failed") if Fraction(UP) has FIELD
--R extendedEuclidean : (%,% ) -> Record(coef1: %,coef2: %,generator: %) if Fraction(UP) has FIELD
--R extendedEuclidean : (%,%,% ) -> Union(Record(coef1: %,coef2: %),"failed") if Fraction(UP) has FIELD
--R factor : % -> Factored(%) if Fraction(UP) has FIELD
--R factorsOfCyclicGroupSize : () -> List(Record(factor: Integer,exponent: Integer)) if Fraction(UP) has FIELD
--R gcd : (%,% ) -> % if Fraction(UP) has FIELD
--R gcd : List(%) -> % if Fraction(UP) has FIELD
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R hyperelliptic : () -> Union(UP,"failed")
--R index : PositiveInteger -> % if Fraction(UP) has FINITE
--R init : () -> % if Fraction(UP) has FFIELDC

```

```

--R integralAtInfinity? : % -> Boolean
--R integralBasisAtInfinity : () -> Vector(%)
--R integralCoordinates : % -> Record(num: Vector(UP),den: UP)
--R integralDerivationMatrix : (UP -> UP) -> Record(num: Matrix(UP),den: UP)
--R integralMatrix : () -> Matrix(Fraction(UP))
--R integralMatrixAtInfinity : () -> Matrix(Fraction(UP))
--R integralRepresents : (Vector(UP),UP) -> %
--R inv : % -> % if Fraction(UP) has FIELD
--R inverseIntegralMatrix : () -> Matrix(Fraction(UP))
--R inverseIntegralMatrixAtInfinity : () -> Matrix(Fraction(UP))
--R knownInfBasis : NonNegativeInteger -> Void
--R lcm : (%,%) -> % if Fraction(UP) has FIELD
--R lcm : List(%) -> % if Fraction(UP) has FIELD
--R lookup : % -> PositiveInteger if Fraction(UP) has FINITE
--R minimalPolynomial : % -> UPUP if Fraction(UP) has FIELD
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed") if Fraction(UP) has FIELD
--R nextItem : % -> Union(%,"failed") if Fraction(UP) has FFIELDC
--R nonSingularModel : Symbol -> List(Polynomial(F)) if F has FIELD
--R normalizeAtInfinity : Vector(%) -> Vector(%)
--R numberOfComponents : () -> NonNegativeInteger
--R order : % -> OnePointCompletion(PositiveInteger) if Fraction(UP) has FFIELDC
--R order : % -> PositiveInteger if Fraction(UP) has FFIELDC
--R prime? : % -> Boolean if Fraction(UP) has FIELD
--R primeFrobenius : % -> % if Fraction(UP) has FFIELDC
--R primeFrobenius : (%,NonNegativeInteger) -> % if Fraction(UP) has FFIELDC
--R primitive? : % -> Boolean if Fraction(UP) has FFIELDC
--R primitiveElement : () -> % if Fraction(UP) has FFIELDC
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %) if Fraction(UP) has FIELD
--R ?quo? : (%,%) -> % if Fraction(UP) has FIELD
--R ramifiedAtInfinity? : () -> Boolean
--R random : () -> % if Fraction(UP) has FINITE
--R rationalPoints : () -> List(List(F)) if F has FINITE
--R reduce : Fraction(UPUP) -> Union(%,"failed") if Fraction(UP) has FIELD
--R reduceBasisAtInfinity : Vector(%) -> Vector(%)
--R reducedSystem : Matrix(%) -> Matrix(Fraction(UP))
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Fraction(UP)),vec: Vector(Fraction(UP)))
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) if Fraction(UP) has FIELD
--R reducedSystem : Matrix(%) -> Matrix(Integer) if Fraction(UP) has LINEXP(INT)
--R regularRepresentation : % -> Matrix(Fraction(UP))
--R regularRepresentation : (%,Vector(%)) -> Matrix(Fraction(UP))
--R ?rem? : (%,%) -> % if Fraction(UP) has FIELD
--R representationType : () -> Union("prime",polynomial,normal,cyclic) if Fraction(UP) has FFIELDC
--R represents : Vector(Fraction(UP)) -> %
--R represents : (Vector(Fraction(UP)),Vector(%)) -> %
--R retract : % -> Fraction(Integer) if Fraction(UP) has RETRACT(FRAC(INT))
--R retract : % -> Integer if Fraction(UP) has RETRACT(INT)
--R retractIfCan : % -> Union(Fraction(UP),"failed")
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if Fraction(UP) has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(Integer,"failed") if Fraction(UP) has RETRACT(INT)
--R singularAtInfinity? : () -> Boolean

```

```

--R size : () -> NonNegativeInteger if Fraction(UP) has FINITE
--R sizeLess? : (%,%) -> Boolean if Fraction(UP) has FIELD
--R squareFree : % -> Factored(%) if Fraction(UP) has FIELD
--R squareFreePart : % -> % if Fraction(UP) has FIELD
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R tableForDiscreteLogarithm : Integer -> Table(PositiveInteger, NonNegativeInteger) if Fraction(UP) has FF
--R traceMatrix : () -> Matrix(Fraction(UP))
--R traceMatrix : Vector(%) -> Matrix(Fraction(UP))
--R unit? : % -> Boolean if Fraction(UP) has FIELD
--R unitCanonical : % -> % if Fraction(UP) has FIELD
--R unitNormal : % -> Record(unit: %, canonical: %, associate: %) if Fraction(UP) has FIELD
--R yCoordinates : % -> Record(num: Vector(UP), den: UP)
--R
--E 1

```

```

)spool
)lisp (bye)

```

— AlgebraicFunctionField.help —

```

=====
AlgebraicFunctionField examples
=====

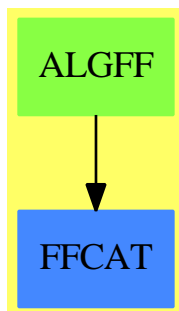
```

The Function field defined by $f(x, y) = 0$.

See Also:

o)show AlgebraicFunctionField

AlgebraicFunctionField (ALGFF)



See

⇒ “RadicalFunctionField” (RADFF) [19](#) on page [2193](#)

Exports:

1	0	absolutelyIrreducible?
algSplitSimple	associates?	basis
branchPoint?	branchPointAtInfinity?	characteristic
characteristicPolynomial	charthRoot	coerce
complementaryBasis	conditionP	convert
coordinates	createPrimitiveElement	D
definingPolynomial	derivationCoordinates	differentiate
discreteLog	discriminant	divide
elliptic	elt	euclideanSize
expressIdealMember	exquo	extendedEuclidean
factor	factorsOfCyclicGroupSize	gcd
gcdPolynomial	generator	genus
hash	hyperelliptic	index
init	integral?	integralAtInfinity?
integralBasis	integralBasisAtInfinity	integralCoordinates
integralDerivationMatrix	integralMatrix	integralMatrixAtInfinity
integralRepresents	inv	inverseIntegralMatrix
inverseIntegralMatrixAtInfinity	knownInfBasis	latex
lcm	lift	lookup
minimalPolynomial	multiEuclidean	nextItem
nonSingularModel	norm	normalizeAtInfinity
numberOfComponents	one?	order
prime?	primeFrobenius	primitive?
primitiveElement	primitivePart	principalIdeal
ramified?	ramifiedAtInfinity?	random
rank	rationalPoint?	rationalPoints
recip	reduce	reduceBasisAtInfinity
reducedSystem	regularRepresentation	representationType
represents	retract	retractIfCan
sample	singular?	singularAtInfinity?
size	sizeLess?	squareFree
squareFreePart	subtractIfCan	tableForDiscreteLogarithm
trace	traceMatrix	unit?
unitCanonical	unitNormal	yCoordinates
zero?	?*?	?**?
?+?	?-?	-?
?=?	?^?	?~=?
?/?	?quo?	?rem?

— domain ALGFF AlgebraicFunctionField —

```
)abbrev domain ALGFF AlgebraicFunctionField
++ Author: Manuel Bronstein
```

```

++ Date Created: 3 May 1988
++ Date Last Updated: 24 Jul 1990
++ Description:
++ Function field defined by  $f(x, y) = 0$ .

AlgebraicFunctionField(F, UP, UPUP, modulus): Exports == Impl where
  F      : Field
  UP      : UnivariatePolynomialCategory F
  UPUP    : UnivariatePolynomialCategory Fraction UP
  modulus: UPUP

  N ==> NonNegativeInteger
  Z ==> Integer
  RF ==> Fraction UP
  QF ==> Fraction UPUP
  UP2 ==> SparseUnivariatePolynomial UP
  SAE ==> SimpleAlgebraicExtension(RF, UPUP, modulus)
  INIT ==> if (deref brandNew?) then startUp false

Exports ==> FunctionFieldCategory(F, UP, UPUP) with
  knownInfBasis: N -> Void
  ++ knownInfBasis(n) is not documented

Impl ==> SAE add
  import ChangeOfVariable(F, UP, UPUP)
  import InnerCommonDenominator(UP, RF, Vector UP, Vector RF)
  import MatrixCommonDenominator(UP, RF)
  import UnivariatePolynomialCategoryFunctions2(RF, UPUP, UP, UP2)

  startUp      : Boolean -> Void
  vect         : Matrix RF -> Vector $
  getInfBasis: () -> Void

  brandNew?:Reference(Boolean) := ref true
  infBr?:Reference(Boolean) := ref true
  discPoly:Reference(RF) := ref 0
  n := degree modulus
  n1 := (n - 1)::N
  ibasis:Matrix(RF) := zero(n, n)
  invibasis:Matrix(RF) := copy ibasis
  infbasis:Matrix(RF) := copy ibasis
  invinfbasis:Matrix(RF) := copy ibasis

  branchPointAtInfinity?() == (INIT; infBr?())
  discriminant() == (INIT; discPoly())
  integralBasis() == (INIT; vect ibasis)
  integralBasisAtInfinity() == (INIT; vect infbasis)
  integralMatrix() == (INIT; ibasis)
  inverseIntegralMatrix() == (INIT; invibasis)
  integralMatrixAtInfinity() == (INIT; infbasis)

```



```

branchPoint?(a:F)          == zero?((retract(discriminant())@UP) a)
definingPolynomial()       == modulus
inverseIntegralMatrixAtInfinity() == (INIT; invinfbasis)

vect m ==
  [represents row(m, i) for i in minRowIndex m .. maxRowIndex m]

integralCoordinates f ==
  splitDenominator(coordinates(f) * inverseIntegralMatrix())

knownInfBasis d ==
  if deref brandNew? then
    alpha := [monomial(1, d * i)$UP :: RF for i in 0..n1]$Vector(RF)
    ib := diagonalMatrix
      [inv qelt(alpha, i) for i in minIndex alpha .. maxIndex alpha]
    invib := diagonalMatrix alpha
    for i in minRowIndex ib .. maxRowIndex ib repeat
      for j in minColIndex ib .. maxColIndex ib repeat
        infbasis(i, j) := qelt(ib, i, j)
        invinfbasis(i, j) := invib(i, j)
  void

getInfBasis() ==
  x := inv(monomial(1, 1)$UP :: RF)
  invmod := map(s +-> s(x), modulus)
  r := mkIntegral invmod
  degree(r.poly) ^= n => error "Should not happen"
  ninvmmod:UP2 := map(s +-> retract(s)@UP, r.poly)
  alpha := [(r.coef ** i) x for i in 0..n1]$Vector(RF)
  invalpha := [inv qelt(alpha, i)
    for i in minIndex alpha .. maxIndex alpha]$Vector(RF)
  invib := integralBasis()$FunctionFieldIntegralBasis(UP, UP2,
    SimpleAlgebraicExtension(UP, UP2, ninvmmod))
  for i in minRowIndex ibasis .. maxRowIndex ibasis repeat
    for j in minColIndex ibasis .. maxColIndex ibasis repeat
      infbasis(i, j) := ((invib.basis)(i,j) / invib.basisDen) x
      invinfbasis(i, j) := ((invib.basisInv) (i, j)) x
  ib2 := infbasis * diagonalMatrix alpha
  invib2 := diagonalMatrix(invalpha) * invinfbasis
  for i in minRowIndex ib2 .. maxRowIndex ib2 repeat
    for j in minColIndex ibasis .. maxColIndex ibasis repeat
      infbasis(i, j) := qelt(ib2, i, j)
      invinfbasis(i, j) := invib2(i, j)
  void

startUp b ==
  brandNew?() := b
  nmod:UP2 := map(retract, modulus)
  ib := integralBasis()$FunctionFieldIntegralBasis(UP, UP2,
    SimpleAlgebraicExtension(UP, UP2, nmod))

```

```

for i in minRowIndex ibasis .. maxRowIndex ibasis repeat
  for j in minColIndex ibasis .. maxColIndex ibasis repeat
    qsetelt_!(ibasis, i, j, (ib.basis)(i, j) / ib.basisDen)
    invibasis(i, j) := ((ib.basisInv) (i, j))::RF
  if zero?(infbasis(minRowIndex infbasis, minColIndex infbasis))
    then getInfBasis()
  ib2 := coordinates normalizeAtInfinity vect ibasis
  invib2 := inverse(ib2)::Matrix(RF)
  for i in minRowIndex ib2 .. maxRowIndex ib2 repeat
    for j in minColIndex ib2 .. maxColIndex ib2 repeat
      ibasis(i, j) := qelt(ib2, i, j)
      invibasis(i, j) := invib2(i, j)
  dsc := resultant(modulus, differentiate modulus)
  dsc0 := dsc * determinant(infbasis) ** 2
  degree(number dsc0) > degree(denom dsc0) =>error "Shouldn't happen"
  infBr?() := degree(number dsc0) < degree(denom dsc0)
  dsc := dsc * determinant(ibasis) ** 2
  discPoly() := primitivePart(number dsc) / denom(dsc)
  void

integralDerivationMatrix d ==
  w := integralBasis()
  splitDenominator(coordinates([differentiate(w.i, d)
    for i in minIndex w .. maxIndex w]$Vector($))
    * inverseIntegralMatrix())

integralRepresents(v, d) ==
  represents(coordinates(represents(v, d)) * integralMatrix())

branchPoint?(p:UP) ==
  INIT
  (r:=retractIfCan(p)@Union(F,"failed")) case F =>branchPoint?(r:F)
  not ground? gcd(retract(discriminant())@UP, p)

```

— ALGFF.dotabb —

```

"ALGFF" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALGFF"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"ALGFF" -> "FFCAT"

```

domain AN AlgebraicNumber

— AlgebraicNumber.input —

```
)set break resume
)sys rm -f AlgebraicNumber.output
)spool AlgebraicNumber.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AlgebraicNumber
--R
--R AlgebraicNumber is a domain constructor
--R Abbreviation for AlgebraicNumber is AN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for AN
--R
--R----- Operations -----
--R ???: (PositiveInteger,%) -> %      ???: (Integer,%) -> %
--R ???: (%,%) -> %                  ???: (%,Fraction(Integer)) -> %
--R ???: (Fraction(Integer),%) -> %    ????: (%,PositiveInteger) -> %
--R ????: (%,Integer) -> %            ?+?: (%,%) -> %
--R -?: % -> %                        ?-?: (%,%) -> %
--R ?/? : (%,%) -> %                  ?<?: (%,%) -> Boolean
--R ?<=: (%,%) -> Boolean              ?=: (%,%) -> Boolean
--R ?>?: (%,%) -> Boolean              ?>=: (%,%) -> Boolean
--R D : % -> %                        D : (%,NonNegativeInteger) -> %
--R 1 : () -> %                       0 : () -> %
--R ?^?: (%,PositiveInteger) -> %      ?^?: (%,Integer) -> %
--R associates?: (%,%) -> Boolean       belong?: BasicOperator -> Boolean
--R box : List(%) -> %                 box : % -> %
--R coerce : Integer -> %               coerce : % -> %
--R coerce : Fraction(Integer) -> %     coerce : Kernel(%) -> %
--R coerce : % -> OutputForm            convert : % -> Complex(Float)
--R convert : % -> DoubleFloat           convert : % -> Float
--R differentiate : % -> %              distribute : (%,%) -> %
--R distribute : % -> %                 elt : (BasicOperator,%,%) -> %
--R elt : (BasicOperator,%) -> %         eval : (%,%,%) -> %
--R eval : (%,Equation(%)) -> %          eval : (%,Kernel(%),%) -> %
--R factor : % -> Factored(%)           freeOf?: (%,Symbol) -> Boolean
--R freeOf?: (%,%) -> Boolean            gcd : (%,%) -> %
--R gcd : List(%) -> %                  hash : % -> SingleInteger
--R height : % -> NonNegativeInteger     inv : % -> %
--R is?: (%,Symbol) -> Boolean            kernel : (BasicOperator,%) -> %
--R kernels : % -> List(Kernel(%))       latex : % -> String
--R lcm : (%,%) -> %                    lcm : List(%) -> %
```

```

--R max : (% , %) -> %
--R norm : (% , List(Kernel(%))) -> %
--R nthRoot : (% , Integer) -> %
--R paren : List(%) -> %
--R prime? : % -> Boolean
--R recip : % -> Union(%, "failed")
--R ?rem? : (% , %) -> %
--R retract : % -> Integer
--R rootOf : Polynomial(%) -> %
--R sizeLess? : (% , %) -> Boolean
--R squareFree : % -> Factored(%)
--R subst : (% , Equation(%)) -> %
--R unit? : % -> Boolean
--R zero? : % -> Boolean
--R ?~=? : (% , %) -> Boolean
--R ?*? : (NonNegativeInteger, %) -> %
--R ??? : (% , NonNegativeInteger) -> %
--R ??? : (% , Fraction(Integer)) -> %
--R ?^? : (% , NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R coerce : SparseMultivariatePolynomial(Integer, Kernel(%)) -> %
--R definingPolynomial : % -> % if $ has RING
--R denom : % -> SparseMultivariatePolynomial(Integer, Kernel(%))
--R differentiate : (% , NonNegativeInteger) -> %
--R divide : (% , %) -> Record(quotient: %, remainder: %)
--R elt : (BasicOperator, List(%)) -> %
--R elt : (BasicOperator, %, %, %, %) -> %
--R elt : (BasicOperator, %, %, %) -> %
--R euclideanSize : % -> NonNegativeInteger
--R eval : (% , BasicOperator, (% -> %)) -> %
--R eval : (% , BasicOperator, (List(%) -> %)) -> %
--R eval : (% , List(BasicOperator), List((List(%) -> %))) -> %
--R eval : (% , List(BasicOperator), List((% -> %))) -> %
--R eval : (% , Symbol, (% -> %)) -> %
--R eval : (% , Symbol, (List(%) -> %)) -> %
--R eval : (% , List(Symbol), List((List(%) -> %))) -> %
--R eval : (% , List(Symbol), List((% -> %))) -> %
--R eval : (% , List(%), List(%)) -> %
--R eval : (% , List(Equation(%))) -> %
--R eval : (% , List(Kernel(%)), List(%)) -> %
--R even? : % -> Boolean if $ has RETRACT(INT)
--R expressIdealMember : (List(%), %) -> Union(List(%), "failed")
--R exquo : (% , %) -> Union(%, "failed")
--R extendedEuclidean : (% , %) -> Record(coef1: %, coef2: %, generator: %)
--R extendedEuclidean : (% , %, %) -> Union(Record(coef1: %, coef2: %), "failed")
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R is? : (% , BasicOperator) -> Boolean
--R kernel : (BasicOperator, List(%)) -> %
--R mainKernel : % -> Union(Kernel(%), "failed")
--R map : ((% -> %), Kernel(%)) -> %
--R min : (% , %) -> %
--R norm : (% , Kernel(%)) -> %
--R one? : % -> Boolean
--R paren : % -> %
--R ?quo? : (% , %) -> %
--R reduce : % -> %
--R retract : % -> Fraction(Integer)
--R retract : % -> Kernel(%)
--R sample : () -> %
--R sqrt : % -> %
--R squareFreePart : % -> %
--R tower : % -> List(Kernel(%))
--R unitCanonical : % -> %
--R zeroOf : Polynomial(%) -> %

```

```

--R minPoly : Kernel(%) -> SparseUnivariatePolynomial(%) if $ has RING
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed")
--R norm : (SparseUnivariatePolynomial(%),List(Kernel(%))) -> SparseUnivariatePolynomial(%)
--R norm : (SparseUnivariatePolynomial(%),Kernel(%)) -> SparseUnivariatePolynomial(%)
--R numer : % -> SparseMultivariatePolynomial(Integer,Kernel(%))
--R odd? : % -> Boolean if $ has RETRACT(INT)
--R operator : BasicOperator -> BasicOperator
--R operators : % -> List(BasicOperator)
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %)
--R reducedSystem : Matrix(%) -> Matrix(Fraction(Integer))
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Fraction(Integer)),vec: Vector(Fraction(Integer)))
--R reducedSystem : Matrix(%) -> Matrix(Integer)
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer))
--R retractIfCan : % -> Union(Fraction(Integer),"failed")
--R retractIfCan : % -> Union(Integer,"failed")
--R retractIfCan : % -> Union(Kernel(%),"failed")
--R rootOf : SparseUnivariatePolynomial(%) -> %
--R rootOf : (SparseUnivariatePolynomial(%),Symbol) -> %
--R rootsOf : Polynomial(%) -> List(%)
--R rootsOf : SparseUnivariatePolynomial(%) -> List(%)
--R rootsOf : (SparseUnivariatePolynomial(%),Symbol) -> List(%)
--R subst : (%,List(Kernel(%)),List(%)) -> %
--R subst : (%,List(Equation(%))) -> %
--R subtractIfCan : (%,%) -> Union(%,"failed")
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R zeroOf : SparseUnivariatePolynomial(%) -> %
--R zeroOf : (SparseUnivariatePolynomial(%),Symbol) -> %
--R zerosOf : Polynomial(%) -> List(%)
--R zerosOf : SparseUnivariatePolynomial(%) -> List(%)
--R zerosOf : (SparseUnivariatePolynomial(%),Symbol) -> List(%)
--R
--E 1

)spool
)lisp (bye)

```

— AlgebraicNumber.help —

```

=====
AlgebraicNumber examples
=====

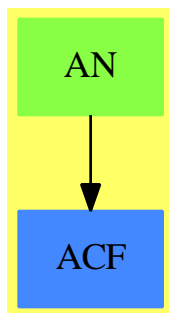
```

Algebraic closure of the rational numbers, with mathematical =

See Also:

o)show AlgebraicNumber

AlgebraicNumber (AN)



See

⇒ “InnerAlgebraicNumber” (IAN) [10](#) on page [1276](#)

Exports:

1	0	associates?	belong?
box	characteristic	coerce	convert
D	definingPolynomial	denom	differentiate
distribute	divide	elt	euclideanSize
eval	even?	expressIdealMember	exquo
extendedEuclidean	factor	freeOf?	gcd
gcdPolynomial	hash	height	inv
is?	kernel	kernels	latex
lcm	mainKernel	map	max
min	minPoly	multiEuclidean	norm
nthRoot	numer	odd?	one?
operator	operators	paren	prime?
principalIdeal	recip	reduce	reducedSystem
retract	retractIfCan	rootOf	rootsOf
sample	sizeLess?	sqrt	squareFree
squareFreePart	subst	subtractIfCan	tower
unit?	unitCanonical	unitNormal	zero?
zeroOf	zerosOf	?*?	?**?
?+?	-?	?-?	?/?
?<?	?<=?	?=?	?>?
?>=?	?^?	?quo?	?rem?
?~=?			

— domain AN AlgebraicNumber —

```

)abbrev domain AN AlgebraicNumber
++ Author: James Davenport

```

```

++ Date Created: 9 October 1995
++ Date Last Updated: 10 October 1995 (JHD)
++ Description:
++ Algebraic closure of the rational numbers, with mathematical =

AlgebraicNumber(): Exports == Implementation where
  Z ==> Integer
  P ==> SparseMultivariatePolynomial(Z, Kernel %)
  SUP ==> SparseUnivariatePolynomial

Exports ==> Join(ExpressionSpace, AlgebraicallyClosedField,
  RetractableTo Z, RetractableTo Fraction Z,
  LinearlyExplicitRingOver Z, RealConstant,
  LinearlyExplicitRingOver Fraction Z,
  CharacteristicZero,
  ConvertibleTo Complex Float, DifferentialRing) with
coerce : P -> %
  ++ coerce(p) returns p viewed as an algebraic number.
numer : % -> P
  ++ numer(f) returns the numerator of f viewed as a
  ++ polynomial in the kernels over Z.
denom : % -> P
  ++ denom(f) returns the denominator of f viewed as a
  ++ polynomial in the kernels over Z.
reduce : % -> %
  ++ reduce(f) simplifies all the unreduced algebraic numbers
  ++ present in f by applying their defining relations.
norm : (SUP(%),Kernel %) -> SUP(%)
  ++ norm(p,k) computes the norm of the polynomial p
  ++ with respect to the extension generated by kernel k
norm : (SUP(%),List Kernel %) -> SUP(%)
  ++ norm(p,l) computes the norm of the polynomial p
  ++ with respect to the extension generated by kernels l
norm : (% ,Kernel %) -> %
  ++ norm(f,k) computes the norm of the algebraic number f
  ++ with respect to the extension generated by kernel k
norm : (% ,List Kernel %) -> %
  ++ norm(f,l) computes the norm of the algebraic number f
  ++ with respect to the extension generated by kernels l
Implementation ==> InnerAlgebraicNumber add
Rep:=InnerAlgebraicNumber
a,b:%
zero? a == trueEqual(a::Rep,0::Rep)
one? a == trueEqual(a::Rep,1::Rep)
a=b == trueEqual((a-b)::Rep,0::Rep)

```

— AN.dotabb —

```
"AN" [color="#88FF44",href="bookvol10.3.pdf#nameddest=AN"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"AN" -> "ACF"
```

domain ANON AnonymousFunction

— AnonymousFunction.input —

```
)set break resume
)sys rm -f AnonymousFunction.output
)spool AnonymousFunction.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AnonymousFunction
--R AnonymousFunction is a domain constructor
--R Abbreviation for AnonymousFunction is ANON
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ANON
--R
--R----- Operations -----
--R ?=? : (% ,%) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger        latex : % -> String
--R ?~=? : (% ,%) -> Boolean
--R
--E 1

)spool
)lisp (bye)
```

— AnonymousFunction.help —

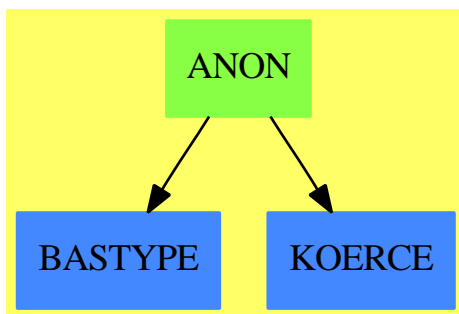
```
=====
AnonymousFunction examples
=====
```

This domain implements anonymous functions

See Also:

o `)show AnonymousFunction`

AnonymousFunction (ANON)



Exports:

`coerce` `hash` `latex` `?=?` `?~=?`

— domain ANON AnonymousFunction —

```

)abbrev domain ANON AnonymousFunction
++ Author: Mark Botch
++ Description:
++ This domain implements anonymous functions

AnonymousFunction():SetCategory == add
  coerce(x:%):OutputForm == x pretend OutputForm
  
```

— ANON.dotabb —

```

"ANON" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ANON"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"ANON" -> "BASTYPE"
"ANON" -> "KOERCE"
  
```

domain ANTISYM AntiSymm

— AntiSymm.input —

```

)set break resume
)sys rm -f AntiSymm.output
)spool AntiSymm.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AntiSymm
--R
--R AntiSymm(R: Ring,lVar: List(Symbol)) is a domain constructor
--R Abbreviation for AntiSymm is ANTISYM
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ANTISYM
--R
--R----- Operations -----
--R ??? : (R,%) -> %           ??? : (%,%) -> %
--R ??? : (Integer,%) -> %     ??? : (PositiveInteger,%) -> %
--R ??? : (%,PositiveInteger) -> %  ?+? : (%,%) -> %
--R ?-? : (%,%) -> %           -? : % -> %
--R ?=? : (%,%) -> Boolean      1 : () -> %
--R 0 : () -> %                ?? : (%,PositiveInteger) -> %
--R coefficient : (%,%) -> R    coerce : R -> %
--R coerce : Integer -> %       coerce : % -> OutputForm
--R degree : % -> NonNegativeInteger  exp : List(Integer) -> %
--R hash : % -> SingleInteger     homogeneous? : % -> Boolean
--R latex : % -> String           leadingBasisTerm : % -> %
--R leadingCoefficient : % -> R    map : ((R -> R),%) -> %
--R one? : % -> Boolean           recip : % -> Union(%, "failed")
--R reductum : % -> %            retract : % -> R
--R retractable? : % -> Boolean   sample : () -> %
--R zero? : % -> Boolean         ?~=? : (%,%) -> Boolean
--R ??? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R generator : NonNegativeInteger -> %
--R retractIfCan : % -> Union(R, "failed")
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)

```

— AntiSymm.help —

=====

AntiSymm examples

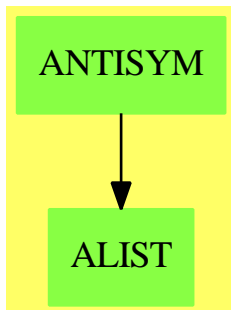
=====

The domain of antisymmetric polynomials.

See Also:

o)show AntiSymm

AntiSymm (ANTISYM)



See

⇒ “ExtAlgBasis” (EAB) [6](#) on page [743](#)

⇒ “DeRhamComplex” (DERHAM) [5](#) on page [530](#)

Exports:

1	0	coefficient	coerce	coerce
coerce	degree	exp	hash	homogeneous?
latex	leadingBasisTerm	leadingCoefficient	map	one?
recip	reductum	retract	retractable?	sample
zero?	characteristic	generator	retractIfCan	subtractIfCan
?*?	?**?	?+?	?-?	-?
?=?	?^?	?~=?		

— domain ANTISYM AntiSymm —

)abbrev domain ANTISYM AntiSymm

++ Author: Larry A. Lambe

++ Date : 01/26/91.

```

++ Revised   : 30 Nov 94
++ Description:
++ The domain of antisymmetric polynomials.

AntiSymm(R:Ring, lVar:List Symbol): Export == Implement where
  LALG ==> LeftAlgebra
  FMR  ==> FM(R,EAB)
  FM   ==> FreeModule
  I    ==> Integer
  L    ==> List
  EAB  ==> ExtAlgBasis      -- these are exponents of basis elements in order
  NNI  ==> NonNegativeInteger
  O    ==> OutputForm
  base ==> k
  coef ==> c
  Term ==> Record(k:EAB,c:R)

Export == Join(LALG(R), RetractableTo(R)) with
  leadingCoefficient : %          -> R
    ++ leadingCoefficient(p) returns the leading
    ++ coefficient of antisymmetric polynomial p.
--  leadingSupport    : %          -> EAB
  leadingBasisTerm   : %          -> %
    ++ leadingBasisTerm(p) returns the leading
    ++ basis term of antisymmetric polynomial p.
  reductum           : %          -> %
    ++ reductum(p), where p is an antisymmetric polynomial,
    ++ returns p minus the leading
    ++ term of p if p has at least two terms, and 0 otherwise.
  coefficient         : (%,% )    -> R
    ++ coefficient(p,u) returns the coefficient of
    ++ the term in p containing the basis term u if such
    ++ a term exists, and 0 otherwise.
    ++ Error: if the second argument u is not a basis element.
  generator          : NNI        -> %
    ++ generator(n) returns the nth multiplicative generator,
    ++ a basis term.
  exp                : L I        -> %
    ++ exp([i1,...in]) returns \spad{u_1}^{i_1} ... u_n^{i_n}
  homogeneous?       : %          -> Boolean
    ++ homogeneous?(p) tests if all of the terms of
    ++ p have the same degree.
  retractable?       : %          -> Boolean
    ++ retractable?(p) tests if p is a 0-form,
    ++ i.e., if degree(p) = 0.
  degree             : %          -> NNI
    ++ degree(p) returns the homogeneous degree of p.
  map                : (R -> R, %) -> %
    ++ map(f,p) changes each coefficient of p by the
    ++ application of f.

```

```

--      1 corresponds to the empty monomial Nul = [0,...,0]
--      from EAB. In terms of the exterior algebra on X,
--      it corresponds to the identity element which lives
--      in homogeneous degree 0.

Implement == FMR add
  Rep := L Term
  x,y : EAB
  a,b : %
  r    : R
  m    : I

  dim := #lVar

  1 == [[ Nul(dim)$EAB, 1$R ]]

  coefficient(a,u) ==
    not null u.rest => error "2nd argument must be a basis element"
    x := u.first.base
    for t in a repeat
      if t.base = x then return t.coef
      if t.base < x then return 0
    0

  retractable?(a) ==
    null a or (a.first.k = Nul(dim))

  retractIfCan(a):Union(R,"failed") ==
    null a => 0$R
    a.first.k = Nul(dim) => leadingCoefficient a
    "failed"

  retract(a):R ==
    null a => 0$R
    leadingCoefficient a

  homogeneous? a ==
    null a => true
    siz := _+/exponents(a.first.base)
    for ta in reductum a repeat
      _+/exponents(ta.base) ^= siz => return false
    true

  degree a ==
    null a => 0$NNI
    homogeneous? a => (_+/exponents(a.first.base)) :: NNI
    error "not a homogeneous element"

```

```

zo : (I,I) -> L I
zo(p,q) ==
  p = 0 => [1,q]
  q = 0 => [1,1]
  [0,0]

getsgn : (EAB,EAB) -> I
getsgn(x,y) ==
  sgn:I := 0
  xx:L I := exponents x
  yy:L I := exponents y
  for i in 1 .. (dim-1) repeat
    xx := rest xx
    sgn := sgn + (_+/xx)*yy.i
  sgn rem 2 = 0 => 1
  -1

Nalpha: (EAB,EAB) -> L I
Nalpha(x,y) ==
  i:I := 1
  dum2:L I := [0 for i in 1..dim]
  for j in 1..dim repeat
    dum:=zo((exponents x).j,(exponents y).j)
    (i:= i*dum.1) = 0 => leave
    dum2.j := dum.2
  i = 0 => cons(i, dum2)
  cons(getsgn(x,y), dum2)

a * b ==
  null a => 0
  null b => 0
  ((null a.rest) and (a.first.k = Nul(dim))) => a.first.c * b
  ((null b.rest) and (b.first.k = Nul(dim))) => b.first.c * a
  z:% := 0
  for tb in b repeat
    for ta in a repeat
      stuff:=Nalpha(ta.base,tb.base)
      r:=first(stuff)*ta.coef*tb.coef
      if r ^= 0 then z := z + [[rest(stuff)::EAB, r]]
    z

coerce(r):% ==
  r = 0 => 0
  [ [Nul(dim), r] ]

coerce(m):% ==
  m = 0 => 0
  [ [Nul(dim), m::R] ]

characteristic() == characteristic()$R

```

```

generator(j) ==
  -- j < 1 or j > dim => error "your subscript is out of range"
  -- error will be generated by dum.j if out of range
  dum:L I := [0 for i in 1..dim]
  dum.j:=1
  [[dum::EAB, 1::R]]

exp(li:(L I)) ==  [[li::EAB, 1]]

leadingBasisTerm a ==
  [[a.first.k, 1]]

displayList:EAB -> 0
displayList(x):0 ==
  le: L I := exponents(x)$EAB
  -- reduce(_*,[(lVar.i)::0 for i in 1..dim | le.i = 1])$L(0)
  -- reduce(_*,[(lVar.i)::0 for i in 1..dim | one?(le.i)])$L(0)
  reduce(_*,[(lVar.i)::0 for i in 1..dim | ((le.i) = 1)])$L(0)

makeTerm:(R,EAB) -> 0
makeTerm(r,x) ==
  -- we know that r ^= 0
  x = Nul(dim)$EAB => r::0
  -- one? r => displayList(x)
  (r = 1) => displayList(x)
  -- r = 1 => displayList(x)
  -- r = 0 => 0$I::0
  -- x = Nul(dim)$EAB => r::0
  r::0 * displayList(x)

coerce(a):0 ==
  zero? a      => 0$I::0
  null rest(a @ Rep) =>
    t := first(a @ Rep)
    makeTerm(t.coef,t.base)
  reduce(_+,[makeTerm(t.coef,t.base) for t in (a @ Rep)])$L(0)

```

— ANTISYM.dotabb —

```

"ANTISYM" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ANTISYM"]
"ALIST"   [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ANTISYM" -> "ALIST"

```

domain ANY Any

— Any.input —

```
)set break resume
)sys rm -f Any.output
)spool Any.output
)set message test on
)set message auto off
)clear all
```

```
--S 1 of 18
```

```
a:Any := [1,2]
```

```
--R
```

```
--R
```

```
--R (1) [1,2]
```

```
--R
```

Type: List(PositiveInteger)

```
--E 1
```

```
--S 2 of 18
```

```
b:Any := [1,2]
```

```
--R
```

```
--R
```

```
--R (2) [1,2]
```

```
--R
```

Type: List(PositiveInteger)

```
--E 2
```

```
--S 3 of 18
```

```
(a = b)@Boolean
```

```
--R
```

```
--R
```

```
--R (3) true
```

```
--R
```

Type: Boolean

```
--E 3
```

```
--S 4 of 18
```

```
c := [1,2]
```

```
--R
```

```
--R
```

```
--R (4) [1,2]
```

```
--R
```

Type: List(PositiveInteger)

```
--E 4
```

```
--S 5 of 18
```

```
typeOf a
```

```
--R
```

```
--R
```

```
--R (5) Any
```


[illegible]

[illegible]

```

--S 18 of 18
)show Any
--R
--R Any is a domain constructor
--R Abbreviation for Any is ANY
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ANY
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          any : (SExpression,None) -> %
--R coerce : % -> OutputForm        dom : % -> SExpression
--R domainOf : % -> OutputForm      hash : % -> SingleInteger
--R latex : % -> String            obj : % -> None
--R objectOf : % -> OutputForm      ?~=? : (%,% ) -> Boolean
--R showTypeInOutput : Boolean -> String
--R
--E 18

)spool
)lisp (bye)

```

— Any.help —

Any examples

Any implements a type that packages up objects and their types in objects of Any. Roughly speaking that means that if $s : S$ then when converted to Any, the new object will include both the original object and its type. This is a way of converting arbitrary objects into a single type without losing any of the original information. Any object can be converted to one of Any.

So we can convert a list to type Any

```

a:Any := [1,2]
      [1,2]

```

and another list to type Any

```

b:Any := [1,2]
      [1,2]

```

Equality works

```
(a = b)@Boolean
true
```

We can compare the Any type with other types:

```
c := [1,2]
```

```
typeof a
```

```
typeof c
```

```
(a = c)@Boolean
```

If the values are different than we see the difference:

```
b := [1,3]
[1,3]
```

```
(a = b)@Boolean
false
```

The Any type works with many types:

```
a := "A"
"A"
```

```
(a = b)@Boolean
false
```

```
b := "A"
"A"
```

```
(a = b)@Boolean
true
```

This is true for more complex types:

```
Sae := SAE(FRAC INT, UP(x, FRAC INT), x^2-3)
```

```
a := generator()$Sae
x
```

```
b := generator()$Sae
x
```

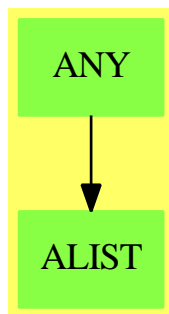
```
(a = b)@Boolean
true
```

See Also:

```
o )show Any
```

```
—————
```

Any (ANY)



See

⇒ “None” (NONE) [15](#) on page [1737](#)

Exports:

```
any    coerce    dom      domainOf      hash
latex  obj      objectOf  showTypeInOutput  ?=?
?~=?
```

— domain ANY Any —

```
)abbrev domain ANY Any
++ Author: Robert S. Sutor
++ Description:
++ \spadtype{Any} implements a type that packages up objects and their
++ types in objects of \spadtype{Any}. Roughly speaking that means
++ that if \spad{s : S} then when converted to \spadtype{Any}, the new
++ object will include both the original object and its type. This is
++ a way of converting arbitrary objects into a single type without
++ losing any of the original information. Any object can be converted
++ to one of \spadtype{Any}.
```

```
Any(): SetCategory with
  any          : (SExpression, None) -> %
  ++ any(type,object) is a technical function for creating
  ++ an object of \spadtype{Any}. Arugment \spad{type} is a
  ++ \spadgloss{LISP} form for the type of \spad{object}.
  domainOf     : % -> OutputForm
  ++ domainOf(a) returns a printable form of the type of the
  ++ original object that was converted to \spadtype{Any}.
```

```

objectOf      : % -> OutputForm
++ objectOf(a) returns a printable form of the
++ original object that was converted to \spadtype{Any}.
dom           : % -> SExpression
++ dom(a) returns a \spadgloss{LISP} form of the type of the
++ original object that was converted to \spadtype{Any}.
obj           : % -> None
++ obj(a) essentially returns the original object that was
++ converted to \spadtype{Any} except that the type is forced
++ to be \spadtype{None}.
showTypeInOutPut: Boolean -> String
++ showTypeInOutPut(bool) affects the way objects of
++ \spadtype{Any} are displayed. If \spad{bool} is true
++ then the type of the original object that was converted
++ to \spadtype{Any} will be printed. If \spad{bool} is
++ false, it will not be printed.

== add
Rep := Record(dm: SExpression, ob: None)

printTypeInOutPutP:Reference(Boolean) := ref false

obj x      == x.ob
dom x      == x.dm
domainOf x == x.dm pretend OutputForm
x = y      == (x.dm = y.dm) and EQ(x.ob, y.ob)$Lisp
x = y      ==
(x.dm = y.dm) and EQUAL(x.ob, y.ob)$Lisp

objectOf(x : %) : OutputForm ==
  spad2BootCoerce(x.ob, x.dm,
    list("OutputForm"::Symbol)$List(Symbol))$Lisp

showTypeInOutPut(b : Boolean) : String ==
printTypeInOutPutP := ref b
b=> "Type of object will be displayed in output of a member of Any"
"Type of object will not be displayed in output of a member of Any"

coerce(x):OutputForm ==
  obj1 : OutputForm := objectOf x
  not deref printTypeInOutPutP => obj1
  dom1 :=
    p:Symbol := prefix2String(devaluate(x.dm)$Lisp)$Lisp
    atom?(p pretend SExpression) => list(p)$List(Symbol)
    list(p)$Symbol
  hconcat cons(obj1,
    cons("::OutputForm, [a::OutputForm for a in dom1]))

any(domain, object) ==
  (isValidType(domain)$Lisp)@Boolean => [domain, object]

```

```

domain := devaluate(domain)$Lisp
(isValidType(domain)$Lisp)@Boolean => [domain, object]
error "function any must have a domain as first argument"

```

— ANY.dotabb —

```

"ANY" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ANY"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ANY" -> "ALIST"

```

domain ASTACK ArrayStack

— ArrayStack.input —

```

)set break resume
)sys rm -f ArrayStack.output
)spool ArrayStack.output
)set message test on
)set message auto off
)clear all

--S 1 of 44
a:ArrayStack INT:= arrayStack [1,2,3,4,5]
--R
--R
--R (1) [1,2,3,4,5]
--R
--R                                          Type: ArrayStack(Integer)
--E 1

--S 2 of 44
pop! a
--R
--R
--R (2) 1
--R
--R                                          Type: PositiveInteger
--E 2

--S 3 of 44
a
--R

```

```

--R
--R (3) [2,3,4,5]
--R                                         Type: ArrayStack(Integer)
--E 3

--S 4 of 44
extract! a
--R
--R
--R (4) 2
--R                                         Type: PositiveInteger
--E 4

--S 5 of 44
a
--R
--R
--R (5) [3,4,5]
--R                                         Type: ArrayStack(Integer)
--E 5

--S 6 of 44
push!(9,a)
--R
--R
--R (6) 9
--R                                         Type: PositiveInteger
--E 6

--S 7 of 44
a
--R
--R
--R (7) [9,3,4,5]
--R                                         Type: ArrayStack(Integer)
--E 7

--S 8 of 44
insert!(8,a)
--R
--R
--R (8) [8,9,3,4,5]
--R                                         Type: ArrayStack(Integer)
--E 8

--S 9 of 44
a
--R
--R
--R (9) [8,9,3,4,5]

```


[illegible]

```

--S 16 of 44
more?(a,9)
--R
--R
--R (16) false
--R
--R                                         Type: Boolean
--E 16

```

```

--S 17 of 44
size?(a,#a)
--R
--R
--R (17) true
--R
--R                                         Type: Boolean
--E 17

```

```

--S 18 of 44
size?(a,9)
--R
--R
--R (18) false
--R
--R                                         Type: Boolean
--E 18

```

```

--S 19 of 44
parts a
--R
--R
--R (19) [8,9,3,4,5]
--R
--R                                         Type: List(Integer)
--E 19

```

```

--S 20 of 44
bag([1,2,3,4,5])$ArrayStack(INT)
--R
--R
--R (20) [5,4,3,2,1]
--R
--R                                         Type: ArrayStack(Integer)
--E 20

```

```

--S 21 of 44
b:=empty()$(ArrayStack INT)
--R
--R
--R (21) []
--R
--R                                         Type: ArrayStack(Integer)
--E 21

```

```

--S 22 of 44

```

```

empty? b
--R
--R
--R (22) true
--R
--R                                          Type: Boolean
--E 22

--S 23 of 44
sample()$ArrayStack(INT)
--R
--R
--R (23) []
--R
--R                                          Type: ArrayStack(Integer)
--E 23

--S 24 of 44
c:=copy a
--R
--R
--R (24) [8,9,3,4,5]
--R
--R                                          Type: ArrayStack(Integer)
--E 24

--S 25 of 44
eq?(a,c)
--R
--R
--R (25) false
--R
--R                                          Type: Boolean
--E 25

--S 26 of 44
eq?(a,a)
--R
--R
--R (26) true
--R
--R                                          Type: Boolean
--E 26

--S 27 of 44
(a=c)@Boolean
--R
--R
--R (27) true
--R
--R                                          Type: Boolean
--E 27

--S 28 of 44
(a=a)@Boolean
--R

```

```

--R
--R (28) true
--R
--R                                         Type: Boolean
--E 28

```

```

--S 29 of 44
a:=c
--R
--R
--R (29) false
--R
--R                                         Type: Boolean
--E 29

```

```

--S 30 of 44
any?(x+-(x=4),a)
--R
--R
--R (30) true
--R
--R                                         Type: Boolean
--E 30

```

```

--S 31 of 44
any?(x+-(x=11),a)
--R
--R
--R (31) false
--R
--R                                         Type: Boolean
--E 31

```

```

--S 32 of 44
every?(x+-(x=11),a)
--R
--R
--R (32) false
--R
--R                                         Type: Boolean
--E 32

```

```

--S 33 of 44
count(4,a)
--R
--R
--R (33) 1
--R
--R                                         Type: PositiveInteger
--E 33

```

```

--S 34 of 44
count(x+-(x>2),a)
--R
--R
--R (34) 5

```

[illegible]

```

--S 41 of 44
coerce a
--R
--R
--R (41) [18,19,13,14,15]
--R
--R                                          Type: OutputForm
--E 41

--S 42 of 44
hash a
--R
--R
--R (42) 36310821
--R
--R                                          Type: SingleInteger
--E 42

--S 43 of 44
latex a
--R
--R
--R (43) "\mbox{\bf Unimplemented}"
--R
--R                                          Type: String
--E 43

--S 44 of 44
)show ArrayStack
--R
--R ArrayStack(S: SetCategory) is a domain constructor
--R Abbreviation for ArrayStack is ASTACK
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASTACK
--R
--R----- Operations -----
--R arrayStack : List(S) -> %          bag : List(S) -> %
--R copy : % -> %                      depth : % -> NonNegativeInteger
--R empty : () -> %                    empty? : % -> Boolean
--R eq? : (%,% ) -> Boolean            extract! : % -> S
--R insert! : (S,% ) -> %              inspect : % -> S
--R map : ((S -> S),%) -> %            pop! : % -> S
--R push! : (S,% ) -> S                sample : () -> %
--R top : % -> S
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?? : (%,% ) -> Boolean if S has SETCAT
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if S has SETCAT
--R count : (S,% ) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R eval : (% ,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (% ,S,S) -> % if S has EVALAB(S) and S has SETCAT

```

```

--R eval : (% ,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (% ,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R hash : % -> SingleInteger if S has SETCAT
--R latex : % -> String if S has SETCAT
--R less? : (% ,NonNegativeInteger) -> Boolean
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R member? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R members : % -> List(S) if $ has finiteAggregate
--R more? : (% ,NonNegativeInteger) -> Boolean
--R parts : % -> List(S) if $ has finiteAggregate
--R size? : (% ,NonNegativeInteger) -> Boolean
--R ?=? : (% ,%) -> Boolean if S has SETCAT
--R
--E 44

)spool
)lisp (bye)

```

— ArrayStack.help —

=====

ArrayStack examples

=====

An ArrayStack object is represented as a list ordered by last-in, first-out. It operates like a pile of books, where the "next" book is the one on the top of the pile.

Here we create an array stack of integers from a list. Notice that the order in the list is the order in the stack.

```

a:ArrayStack INT:= arrayStack [1,2,3,4,5]
[1,2,3,4,5]

```

We can remove the top of the stack using pop!:

```

pop! a
1

```

Notice that the use of pop! is destructive (destructive operations in Axiom usually end with ! to indicate that the underlying data structure is changed).

```

a
[2,3,4,5]

```

The `extract!` operation is another name for the `pop!` operation and has the same effect. This operation treats the stack as a `BagAggregate`:

```
extract! a
2
```

and you can see that it also has destructively modified the stack:

```
a
[3,4,5]
```

Next we push a new element on top of the stack:

```
push!(9,a)
9
```

Again, the `push!` operation is destructive so the stack is changed:

```
a
[9,3,4,5]
```

Another name for `push!` is `insert!`, which treats the stack as a `BagAggregate`:

```
insert!(8,a)
[8,9,3,4,5]
```

and it modifies the stack:

```
a
[8,9,3,4,5]
```

The `inspect` function returns the top of the stack without modification, viewed as a `BagAggregate`:

```
inspect a
8
```

The `empty?` operation returns true only if there are no element on the stack, otherwise it returns false:

```
empty? a
false
```

The `top` operation returns the top of stack without modification, viewed as a `Stack`:

```
top a
8
```


The depth operation returns the number of elements on the stack:

```
depth a
5
```

which is the same as the # (length) operation:

```
#a
5
```

The less? predicate will compare the stack length to an integer:

```
less?(a,9)
true
```

The more? predicate will compare the stack length to an integer:

```
more?(a,9)
false
```

The size? operation will compare the stack length to an integer:

```
size?(a,#a)
true
```

and since the last computation must always be true we try:

```
size?(a,9)
false
```

The parts function will return the stack as a list of its elements:

```
parts a
[8,9,3,4,5]
```

If we have a BagAggregate of elements we can use it to construct a stack. Notice that the elements are pushed in reverse order:

```
bag([1,2,3,4,5])$ArrayStack(INT)
[5,4,3,2,1]
```

The empty function will construct an empty stack of a given type:

```
b:=empty()$(ArrayStack INT)
[]
```

and the empty? predicate allows us to find out if a stack is empty:

```
empty? b
true
```

The sample function returns a sample, empty stack:

```
sample()$ArrayStack(INT)
[]
```

We can copy a stack and it does not share storage so subsequent modifications of the original stack will not affect the copy:

```
c:=copy a
[8,9,3,4,5]
```

The eq? function is only true if the lists are the same reference, so even though c is a copy of a, they are not the same:

```
eq?(a,c)
false
```

However, a clearly shares a reference with itself:

```
eq?(a,a)
true
```

But we can compare a and c for equality:

```
(a=c)@Boolean
true
```

and clearly a is equal to itself:

```
(a=a)@Boolean
true
```

and since a and c are equal, they are clearly NOT not-equal:

```
a~=c
false
```

We can use the any? function to see if a predicate is true for any element:

```
any?(x+>(x=4),a)
true
```

or false for every element:

```
any?(x+>(x=11),a)
false
```

We can use the every? function to check every element satisfies a predicate:

```
every?(x+>(x=11),a)
false
```

We can count the elements that are equal to an argument of this type:

```
count(4,a)
1
```

or we can count against a boolean function:

```
count(x+>(x>2),a)
5
```

You can also map a function over every element, returning a new stack:

```
map(x+>x+10,a)
[18,19,13,14,15]
```

Notice that the original stack is unchanged:

```
a
[8,9,3,4,5]
```

You can use `map!` to map a function over every element and change the original stack since `map!` is destructive:

```
map!(x+>x+10,a)
[18,19,13,14,15]
```

Notice that the original stack has been changed:

```
a
[18,19,13,14,15]
```

The `members` function can also get the element of the stack as a list:

```
members a
[18,19,13,14,15]
```

and using `member?` we can test if the stack holds a given element:

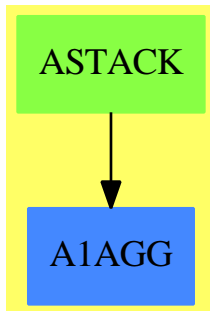
```
member?(14,a)
true
```

See Also:

- o `)show Stack`
- o `)show ArrayStack`
- o `)show Queue`
- o `)show Dequeue`
- o `)show Heap`

```
o )show BagAggregate
```

ArrayStack (ASTACK)



See

- ⇒ “Stack” (STACK) [20](#) on page [2563](#)
- ⇒ “Queue” (QUEUE) [18](#) on page [2184](#)
- ⇒ “Dequeue” (DEQUEUE) [5](#) on page [512](#)
- ⇒ “Heap” (HEAP) [9](#) on page [1134](#)

Exports:

any?	arrayStack	bag	coerce	copy
count	depth	empty	empty?	eq?
eval	every?	extract!	hash	insert!
inspect	latex	less?	map	map!
member?	members	more?	parts	pop!
push!	sample	size?	top	#?
?~=?	?=?			

— domain ASTACK ArrayStack —

```

)abbrev domain ASTACK ArrayStack
++ Author: Michael Monagan, Stephen Watt, Timothy Daly
++ Date Created: June 86 and July 87
++ Date Last Updated: Feb 92
++ Description:
++ A stack represented as a flexible array.
--% Dequeue and Heap data types

ArrayStack(S:SetCategory): StackAggregate(S) with
  arrayStack: List S -> %
  ++ arrayStack([x,y,...,z]) creates an array stack with first (top)

```

```

++ element x, second element y,...,and last element z.
++
++E c:ArrayStack INT:= arrayStack [1,2,3,4,5]

-- Inherited Signatures repeated for examples documentation

pop_! : % -> S
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X pop! a
++X a
extract_! : % -> S
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X extract! a
++X a
push_! : (S,%) -> S
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X push!(9,a)
++X a
insert_! : (S,%) -> %
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X insert!(8,a)
++X a
inspect : % -> S
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X inspect a
top : % -> S
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X top a
depth : % -> NonNegativeInteger
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X depth a
less? : (%,NonNegativeInteger) -> Boolean
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X less?(a,9)
more? : (%,NonNegativeInteger) -> Boolean
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X more?(a,9)
size? : (%,NonNegativeInteger) -> Boolean
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X size?(a,5)

```

```

bag : List S -> %
++
++X bag([1,2,3,4,5])$ArrayStack(INT)
empty? : % -> Boolean
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X empty? a
empty : () -> %
++
++X b:=empty()$(ArrayStack INT)
sample : () -> %
++
++X sample()$ArrayStack(INT)
copy : % -> %
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X copy a
eq? : (%,%) -> Boolean
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X b:=copy a
++X eq?(a,b)
map : ((S -> S),%) -> %
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X map(x+>x+10,a)
++X a
if $ has shallowlyMutable then
map! : ((S -> S),%) -> %
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X map!(x+>x+10,a)
++X a
if S has SetCategory then
latex : % -> String
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X latex a
hash : % -> SingleInteger
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X hash a
coerce : % -> OutputForm
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X coerce a
"=" : (%,%) -> Boolean
++
++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
++X b:ArrayStack INT:= arrayStack [1,2,3,4,5]

```

```

    ++X (a=b)@Boolean
    "~=" : (%,% ) -> Boolean
    ++
    ++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
    ++X b:=copy a
    ++X (a~=b)
if % has finiteAggregate then
    every? : ((S -> Boolean),%) -> Boolean
    ++
    ++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
    ++X every?(x+-(x=4),a)
    any? : ((S -> Boolean),%) -> Boolean
    ++
    ++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
    ++X any?(x+-(x=4),a)
    count : ((S -> Boolean),%) -> NonNegativeInteger
    ++
    ++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
    ++X count(x+-(x>2),a)
    _# : % -> NonNegativeInteger
    ++
    ++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
    ++X #a
    parts : % -> List S
    ++
    ++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
    ++X parts a
    members : % -> List S
    ++
    ++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
    ++X members a
if % has finiteAggregate and S has SetCategory then
    member? : (S,%) -> Boolean
    ++
    ++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
    ++X member?(3,a)
    count : (S,%) -> NonNegativeInteger
    ++
    ++X a:ArrayStack INT:= arrayStack [1,2,3,4,5]
    ++X count(4,a)

== add
Rep := IndexedFlexibleArray(S,0)

-- system operations
# s == _#(s)$Rep
s = t == s =$Rep t
copy s == copy(s)$Rep
coerce(d):OutputForm ==
    empty? d => empty()$(List S) ::OutputForm

```

```

      [(d.i::OutputForm) for i in 0..#d-1] ::OutputForm

-- stack operations
depth s == # s
empty? s == empty?(s)$Rep
extract_! s == pop_! s
insert_!(e,s) == (push_!(e,s);s)
push_!(e,s) == (concat(e,s); e)
pop_! s ==
  if empty? s then error "empty stack"
  r := s.0
  delete_!(s,0)
  r
top s == if empty? s then error "empty stack" else s.0
arrayStack l == construct(l)$Rep
empty() == new(0,0 pretend S)
parts s == [s.i for i in 0..#s-1]::List(S)
map(f,s) == construct [f(s.i) for i in 0..#s-1]
map!(f,s) == ( for i in 0..#s-1 repeat qsetelt!(s,i,f(s.i)) ; s )
inspect(s) ==
  if empty? s then error "empty stack"
  qelt(s,0)

```

— ASTACK.dotabb —

```

"ASTACK" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASTACK"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"ASTACK" -> "A1AGG"

```

domain ASP1 Asp1

— Asp1.input —

```

)set break resume
)sys rm -f Asp1.output
)spool Asp1.output
)set message test on
)set message auto off
)clear all

```



```

--S 1 of 1
)show Asp1
--R
--R Asp1(name: Symbol) is a domain constructor
--R Abbreviation for Asp1 is ASP1
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP1
--R
--R----- Operations -----
--R coerce : FortranCode -> %          coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm          outputAsFortran : % -> Void
--R retract : Polynomial(Float) -> %    retract : Expression(Float) -> %
--R coerce : FortranExpression([construct,QUOTEX],[construct],MachineFloat) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Fraction(Polynomial(Integer)) -> %
--R retract : Fraction(Polynomial(Float)) -> %
--R retract : Polynomial(Integer) -> %
--R retract : Expression(Integer) -> %
--R retractIfCan : Fraction(Polynomial(Integer)) -> Union(%, "failed")
--R retractIfCan : Fraction(Polynomial(Float)) -> Union(%, "failed")
--R retractIfCan : Polynomial(Integer) -> Union(%, "failed")
--R retractIfCan : Polynomial(Float) -> Union(%, "failed")
--R retractIfCan : Expression(Integer) -> Union(%, "failed")
--R retractIfCan : Expression(Float) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)

```

— Asp1.help —

=====

Asp1 examples

=====

Asp1 produces Fortran for Type 1 ASPs, needed for various NAG routines.
Type 1 ASPs take a univariate expression (in the symbol x) and turn it
into a Fortran Function like the following:

```

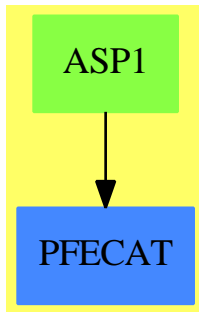
DOUBLE PRECISION FUNCTION F(X)
DOUBLE PRECISION X
F=DSIN(X)
RETURN
END

```

See Also:

```
o )show Asp1
```

Asp1 (ASP1)



Exports:

```
coerce outputAsFortran retract retractIfCan
```

— domain ASP1 Asp1 —

```

)abbrev domain ASP1 Asp1
++ Author: Mike Dewar, Grant Keady, Godfrey Nolan
++ Date Created: Mar 1993
++ Date Last Updated: 18 March 1994, 6 October 1994
++ Description:
++ \spadtype{Asp1} produces Fortran for Type 1 ASPs, needed for various
++ NAG routines. Type 1 ASPs take a univariate expression (in the symbol x)
++ and turn it into a Fortran Function like the following:
++
++ \tab{5}DOUBLE PRECISION FUNCTION F(X)\br
++ \tab{5}DOUBLE PRECISION X\br
++ \tab{5}F=DSIN(X)\br
++ \tab{5}RETURN\br
++ \tab{5}END
  
```

```

Asp1(name): Exports == Implementation where
  name : Symbol
  
```

```

FEXPR ==> FortranExpression
FST   ==> FortranScalarType
FT    ==> FortranType
SYMTAB ==> SymbolTable
RSFC  ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
  
```

```

FRAC    ==> Fraction
POLY    ==> Polynomial
EXPR    ==> Expression
INT     ==> Integer
FLOAT   ==> Float

Exports ==> FortranFunctionCategory with
  coerce : FEXPR(['X'],[],MachineFloat) -> $
    ++coerce(f) takes an object from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns it into an ASP.

Implementation ==> add

-- Build Symbol Table for Rep
syms : SYMTAB := empty()$SYMTAB
declare!(X,fortranReal()$FT,syms)$SYMTAB
real : FST := "real":FST

Rep := FortranProgram(name,[real]$Union(fst:FST,void:"void"),[X],syms)

retract(u:FRAC POLY INT):$ == (retract(u)@FEXPR(['X'],[],MachineFloat))::$
retractIfCan(u:FRAC POLY INT):Union($,"failed") ==
  foo : Union(FEXPR(['X'],[],MachineFloat),"failed")
  foo := retractIfCan(u)$FEXPR(['X'],[],MachineFloat)
  foo case "failed" => "failed"
  foo::FEXPR(['X'],[],MachineFloat)::$

retract(u:FRAC POLY FLOAT):$ == (retract(u)@FEXPR(['X'],[],MachineFloat))::$
retractIfCan(u:FRAC POLY FLOAT):Union($,"failed") ==
  foo : Union(FEXPR(['X'],[],MachineFloat),"failed")
  foo := retractIfCan(u)$FEXPR(['X'],[],MachineFloat)
  foo case "failed" => "failed"
  foo::FEXPR(['X'],[],MachineFloat)::$

retract(u:EXPR FLOAT):$ == (retract(u)@FEXPR(['X'],[],MachineFloat))::$
retractIfCan(u:EXPR FLOAT):Union($,"failed") ==
  foo : Union(FEXPR(['X'],[],MachineFloat),"failed")
  foo := retractIfCan(u)$FEXPR(['X'],[],MachineFloat)
  foo case "failed" => "failed"
  foo::FEXPR(['X'],[],MachineFloat)::$

retract(u:EXPR INT):$ == (retract(u)@FEXPR(['X'],[],MachineFloat))::$
retractIfCan(u:EXPR INT):Union($,"failed") ==
  foo : Union(FEXPR(['X'],[],MachineFloat),"failed")
  foo := retractIfCan(u)$FEXPR(['X'],[],MachineFloat)
  foo case "failed" => "failed"
  foo::FEXPR(['X'],[],MachineFloat)::$

retract(u:POLY FLOAT):$ == (retract(u)@FEXPR(['X'],[],MachineFloat))::$
retractIfCan(u:POLY FLOAT):Union($,"failed") ==

```

```

foo : Union(FEXPR(['X],[],MachineFloat),"failed")
foo := retractIfCan(u)$FEXPR(['X],[],MachineFloat)
foo case "failed" => "failed"
foo::FEXPR(['X],[],MachineFloat)::$

retract(u:POLY INT):$ == (retract(u)$FEXPR(['X],[],MachineFloat))::$
retractIfCan(u:POLY INT):Union($,"failed") ==
  foo : Union(FEXPR(['X],[],MachineFloat),"failed")
  foo := retractIfCan(u)$FEXPR(['X],[],MachineFloat)
  foo case "failed" => "failed"
  foo::FEXPR(['X],[],MachineFloat)::$

coerce(u:FEXPR(['X],[],MachineFloat)):$ ==
  coerce((u::Expression(MachineFloat))$FEXPR(['X],[],MachineFloat))$Rep

coerce(c:List FortranCode):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

coerce(c:FortranCode):$ == coerce(c)$Rep

coerce(u:$):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

```

— ASP1.dotabb —

```

"ASP1" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP1"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"ASP1" -> "PFECAT"

```

domain ASP10 Asp10

— Asp10.input —

```

)set break resume
)sys rm -f Asp10.output

```

```

)spool Asp10.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp10
--R
--R Asp10(name: Symbol) is a domain constructor
--R Abbreviation for Asp10 is ASP10
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP10
--R
--R----- Operations -----
--R coerce : FortranCode -> %               coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm               outputAsFortran : % -> Void
--R coerce : Vector(FortranExpression([construct,QUOTEJINT,QUOTEX,QUOTEELAM],[construct],Machine
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Vector(Fraction(Polynomial(Integer))) -> %
--R retract : Vector(Fraction(Polynomial(Float))) -> %
--R retract : Vector(Polynomial(Integer)) -> %
--R retract : Vector(Polynomial(Float)) -> %
--R retract : Vector(Expression(Integer)) -> %
--R retract : Vector(Expression(Float)) -> %
--R retractIfCan : Vector(Fraction(Polynomial(Integer))) -> Union(%, "failed")
--R retractIfCan : Vector(Fraction(Polynomial(Float))) -> Union(%, "failed")
--R retractIfCan : Vector(Polynomial(Integer)) -> Union(%, "failed")
--R retractIfCan : Vector(Polynomial(Float)) -> Union(%, "failed")
--R retractIfCan : Vector(Expression(Integer)) -> Union(%, "failed")
--R retractIfCan : Vector(Expression(Float)) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)

```

— Asp10.help —

=====

Asp10 examples

=====

ASP10 produces Fortran for Type 10 ASPs, needed for NAG routine d02kef.
This ASP computes the values of a set of functions, for example:

```

SUBROUTINE COEFFN(P,Q,DQDL,X,ELAM,JINT)
DOUBLE PRECISION ELAM,P,Q,X,DQDL

```

```

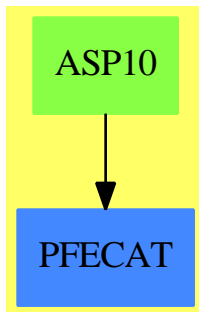
INTEGER JINT
P=1.0D0
Q=(-1.0D0*X**3)+ELAM*X*X-2.0D0)/(X*X)
DQDL=1.0D0
RETURN
END

```

See Also:

o)show Asp10

Asp10 (ASP10)



Exports:

coerce outputAsFortran retract retractIfCan

— domain ASP10 Asp10 —

```

)abbrev domain ASP10 Asp10
++ Author: Mike Dewar and Godfrey Nolan
++ Date Created: Mar 1993
++ Date Last Updated: 18 March 1994, 6 October 1994
++ Description:
++ \spadtype{ASP10} produces Fortran for Type 10 ASPs, needed for NAG routine
++ d02kef. This ASP computes the values of a set of functions, for example:
++
++ \tab{5}SUBROUTINE COEFFN(P,Q,DQDL,X,ELAM,JINT)\br
++ \tab{5}DOUBLE PRECISION ELAM,P,Q,X,DQDL\br
++ \tab{5}INTEGER JINT\br
++ \tab{5}P=1.0D0\br
++ \tab{5}Q=(-1.0D0*X**3)+ELAM*X*X-2.0D0)/(X*X)\br
++ \tab{5}DQDL=1.0D0\br
++ \tab{5}RETURN\br

```

```
++ \tab{5}END
```

```
Asp10(name): Exports == Implementation where
  name : Symbol
```

```

FST ==> FortranScalarType
FT   ==> FortranType
SYMTAB ==> SymbolTable
EXF ==> Expression Float
RSFC ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
FEXPR ==> FortranExpression(['JINT','X','ELAM'],[],MFLOAT)
MFLOAT ==> MachineFloat
FRAC  ==> Fraction
POLY  ==> Polynomial
EXPR  ==> Expression
INT   ==> Integer
FLOAT ==> Float
VEC   ==> Vector
VF2   ==> VectorFunctions2
```

```

Exports ==> FortranVectorFunctionCategory with
  coerce : Vector FEXPR -> %
    ++coerce(f) takes objects from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns them into an ASP.
```

```
Implementation ==> add
```

```

real : FST := "real"::FST
syms : SYMTAB := empty()$SYMTAB
declare!(P,fortranReal()$FT,syms)$SYMTAB
declare!(Q,fortranReal()$FT,syms)$SYMTAB
declare!(DQDL,fortranReal()$FT,syms)$SYMTAB
declare!(X,fortranReal()$FT,syms)$SYMTAB
declare!(ELAM,fortranReal()$FT,syms)$SYMTAB
declare!(JINT,fortranInteger()$FT,syms)$SYMTAB
Rep := FortranProgram(name,["void"]$Union(fst:FST,void:"void"),
                      [P,Q,DQDL,X,ELAM,JINT],syms)

retract(u:VEC FRAC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$(

retract(u:VEC FRAC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v::$
```

```

retractIfCan(u:VEC FRAC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC EXPR INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR INT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC EXPR FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

coerce(c:FortranCode):% == coerce(c)$Rep

coerce(r:RSFC):% == coerce(r)$Rep

coerce(c:List FortranCode):% == coerce(c)$Rep

-- To help the poor old compiler!
localAssign(s:Symbol,u:Expression MFLOAT):FortranCode ==

```



```

assign(s,u)$FortranCode

coerce(u:Vector FEXPR):% ==
  import Vector FEXPR
  not (#u = 3) => error "Incorrect Dimension For Vector"
  ([localAssign(P,elt(u,1)::Expression MFLOAT),_
    localAssign(Q,elt(u,2)::Expression MFLOAT),_
    localAssign(DQDL,elt(u,3)::Expression MFLOAT),_
    returns()$FortranCode ]$List(FortranCode))::Rep

coerce(u:%):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

_____

— ASP10.dotabb —

"ASP10" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP10"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"ASP10" -> "PFECAT"

_____

```

domain ASP12 Asp12

```

_____

— Asp12.input —

)set break resume
)sys rm -f Asp12.output
)spool Asp12.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp12
--R
--R Asp12(name: Symbol) is a domain constructor
--R Abbreviation for Asp12 is ASP12
--R This constructor is exposed in this frame.

```

```

--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP12
--R
--R----- Operations -----
--R coerce : % -> OutputForm          outputAsFortran : () -> Void
--R outputAsFortran : % -> Void
--R
--E 1

```

```

)spool
)lisp (bye)

```

— Asp12.help —

```

=====
Asp12 examples
=====

```

\spadtype{Asp12} produces Fortran for Type 12 ASPs, needed for NAG routine d02kef etc., for example:

```

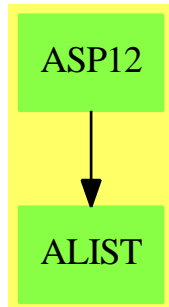
SUBROUTINE MONIT (MAXIT,IFLAG,ELAM,FINFO)
DOUBLE PRECISION ELAM,FINFO(15)
INTEGER MAXIT,IFLAG
IF(MAXIT.EQ.-1)THEN
PRINT*,"Output from Monit"
ENDIF
PRINT*,MAXIT,IFLAG,ELAM,(FINFO(I),I=1,4)
RETURN
END

```

See Also:

- o)show Asp12

Asp12 (ASP12)



Exports:

coerce outputAsFortran

— domain ASP12 Asp12 —

```

)abbrev domain ASP12 Asp12
++ Author: Mike Dewar and Godfrey Nolan
++ Date Created: Oct 1993
++ Date Last Updated: 21 June 1994 Changed print to printStatement
++ Description:
++ \spadtype{Asp12} produces Fortran for Type 12 ASPs, needed for NAG routine
++ d02kef etc., for example:
++
++ \tab{5}SUBROUTINE MONIT (MAXIT,IFLAG,ELAM,FINFO)\br
++ \tab{5}DOUBLE PRECISION ELAM,FINFO(15)\br
++ \tab{5}INTEGER MAXIT,IFLAG\br
++ \tab{5}IF(MAXIT.EQ.-1)THEN\br
++ \tab{7}PRINT*,"Output from Monit"\br
++ \tab{5}ENDIF\br
++ \tab{5}PRINT*,MAXIT,IFLAG,ELAM,(FINFO(I),I=1,4)\br
++ \tab{5}RETURN\br
++ \tab{5}END\

```

Asp12(name): Exports == Implementation where
 name : Symbol

```

O      ==> OutputForm
S      ==> Symbol
FST    ==> FortranScalarType
FT     ==> FortranType
FC     ==> FortranCode
SYMTAB ==> SymbolTable
EXI    ==> Expression Integer
RSFC   ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
U      ==> Union(I: Expression Integer,F: Expression Float,_)

```

```

CF: Expression Complex Float,switch:Switch)
UFST ==> Union(fst:FST,void:"void")

Exports ==> FortranProgramCategory with
  outputAsFortran:() -> Void
  ++outputAsFortran() generates the default code for \spadtype{ASP12}.

Implementation ==> add

import FC
import Switch

real : FST := "real"::FST
syms : SYMTAB := empty()$SYMTAB
declare!(MAXIT,fortranInteger()$FT,syms)$SYMTAB
declare!(IFLAG,fortranInteger()$FT,syms)$SYMTAB
declare!(ELAM,fortranReal()$FT,syms)$SYMTAB
fType : FT := construct([real]$UFST,["15"::Symbol],false)$FT
declare!(FINFO,fType,syms)$SYMTAB
Rep := FortranProgram(name,["void"]$UFST,[MAXIT,IFLAG,ELAM,FINFO],syms)

-- eqn : 0 := (I::0)=(1@Integer::EXI::0)
code:=(cond(EQ([MAXIT@S::EXI]$U,[-1::EXI]$U),
  printStatement(["_Output from Monit_"::0])),
  printStatement([MAXIT::0,IFLAG::0,ELAM::0,subscript("FINFO"::S,[I::0])::0,"I=1"::S::0,"4"::S::0,
  returns()$List(FortranCode))::Rep

coerce(u:%):OutputForm == coerce(u)$Rep

outputAsFortran(u:%):Void == outputAsFortran(u)$Rep
outputAsFortran():Void == outputAsFortran(code)$Rep

-----

— ASP12.dotabb —

"ASP12" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP12"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ASP12" -> "ALIST"

-----

```

domain ASP19 Asp19

— Asp19.input —

```

)set break resume
)sys rm -f Asp19.output
)spool Asp19.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp19
--R
--R Asp19(name: Symbol) is a domain constructor
--R Abbreviation for Asp19 is ASP19
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP19
--R
--R----- Operations -----
--R coerce : FortranCode -> %           coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm           outputAsFortran : % -> Void
--R coerce : Vector(FortranExpression([construct],[construct,QUOTEXC],MachineFloat)) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Vector(Fraction(Polynomial(Integer))) -> %
--R retract : Vector(Fraction(Polynomial(Float))) -> %
--R retract : Vector(Polynomial(Integer)) -> %
--R retract : Vector(Polynomial(Float)) -> %
--R retract : Vector(Expression(Integer)) -> %
--R retract : Vector(Expression(Float)) -> %
--R retractIfCan : Vector(Fraction(Polynomial(Integer))) -> Union(%, "failed")
--R retractIfCan : Vector(Fraction(Polynomial(Float))) -> Union(%, "failed")
--R retractIfCan : Vector(Polynomial(Integer)) -> Union(%, "failed")
--R retractIfCan : Vector(Polynomial(Float)) -> Union(%, "failed")
--R retractIfCan : Vector(Expression(Integer)) -> Union(%, "failed")
--R retractIfCan : Vector(Expression(Float)) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)

```

— Asp19.help —

=====

Asp19 examples

=====

\spadtype{Asp19} produces Fortran for Type 19 ASPs, evaluating a set of functions and their jacobian at a given point, for example:

```

SUBROUTINE LSFUN2(M,N,XC,FVECC,FJACC,LJC)
DOUBLE PRECISION FVECC(M),FJACC(LJC,N),XC(N)
INTEGER M,N,LJC
INTEGER I,J
DO 25003 I=1,LJC
DO 25004 J=1,N
FJACC(I,J)=0.0D0
004 CONTINUE
003 CONTINUE
FVECC(1)=((XC(1)-0.14D0)*XC(3)+(15.0D0*XC(1)-2.1D0)*XC(2)+1.0D0)/(
&XC(3)+15.0D0*XC(2))
FVECC(2)=((XC(1)-0.18D0)*XC(3)+(7.0D0*XC(1)-1.26D0)*XC(2)+1.0D0)/(
&XC(3)+7.0D0*XC(2))
FVECC(3)=((XC(1)-0.22D0)*XC(3)+(4.333333333333333D0*XC(1)-0.953333
&3333333333D0)*XC(2)+1.0D0)/(XC(3)+4.333333333333333D0*XC(2))
FVECC(4)=((XC(1)-0.25D0)*XC(3)+(3.0D0*XC(1)-0.75D0)*XC(2)+1.0D0)/(
&XC(3)+3.0D0*XC(2))
FVECC(5)=((XC(1)-0.29D0)*XC(3)+(2.2D0*XC(1)-0.637999999999999D0)*
&XC(2)+1.0D0)/(XC(3)+2.2D0*XC(2))
FVECC(6)=((XC(1)-0.32D0)*XC(3)+(1.666666666666667D0*XC(1)-0.533333
&3333333333D0)*XC(2)+1.0D0)/(XC(3)+1.666666666666667D0*XC(2))
FVECC(7)=((XC(1)-0.35D0)*XC(3)+(1.285714285714286D0*XC(1)-0.45D0)*
&XC(2)+1.0D0)/(XC(3)+1.285714285714286D0*XC(2))
FVECC(8)=((XC(1)-0.39D0)*XC(3)+(XC(1)-0.39D0)*XC(2)+1.0D0)/(XC(3)+
&XC(2))
FVECC(9)=((XC(1)-0.37D0)*XC(3)+(XC(1)-0.37D0)*XC(2)+1.285714285714
&286D0)/(XC(3)+XC(2))
FVECC(10)=((XC(1)-0.58D0)*XC(3)+(XC(1)-0.58D0)*XC(2)+1.66666666666
&6667D0)/(XC(3)+XC(2))
FVECC(11)=((XC(1)-0.73D0)*XC(3)+(XC(1)-0.73D0)*XC(2)+2.2D0)/(XC(3)
&+XC(2))
FVECC(12)=((XC(1)-0.96D0)*XC(3)+(XC(1)-0.96D0)*XC(2)+3.0D0)/(XC(3)
&+XC(2))
FVECC(13)=((XC(1)-1.34D0)*XC(3)+(XC(1)-1.34D0)*XC(2)+4.33333333333
&3333D0)/(XC(3)+XC(2))
FVECC(14)=((XC(1)-2.1D0)*XC(3)+(XC(1)-2.1D0)*XC(2)+7.0D0)/(XC(3)+X
&C(2))
FVECC(15)=((XC(1)-4.39D0)*XC(3)+(XC(1)-4.39D0)*XC(2)+15.0D0)/(XC(3)
&+XC(2))
FJACC(1,1)=1.0D0
FJACC(1,2)=-15.0D0/(XC(3)**2+30.0D0*XC(2)*XC(3)+225.0D0*XC(2)**2)
FJACC(1,3)=-1.0D0/(XC(3)**2+30.0D0*XC(2)*XC(3)+225.0D0*XC(2)**2)
FJACC(2,1)=1.0D0
FJACC(2,2)=-7.0D0/(XC(3)**2+14.0D0*XC(2)*XC(3)+49.0D0*XC(2)**2)
FJACC(2,3)=-1.0D0/(XC(3)**2+14.0D0*XC(2)*XC(3)+49.0D0*XC(2)**2)
FJACC(3,1)=1.0D0
FJACC(3,2)=((-0.1110223024625157D-15*XC(3))-4.333333333333333D0)/(
&XC(3)**2+8.666666666666666D0*XC(2)*XC(3)+18.7777777777778D0*XC(2)
&**2)
FJACC(3,3)=(0.1110223024625157D-15*XC(2)-1.0D0)/(XC(3)**2+8.666666

```

```

&666666666D0*XC(2)*XC(3)+18.7777777777778D0*XC(2)**2)
FJACC(4,1)=1.0D0
FJACC(4,2)=-3.0D0/(XC(3)**2+6.0D0*XC(2)*XC(3)+9.0D0*XC(2)**2)
FJACC(4,3)=-1.0D0/(XC(3)**2+6.0D0*XC(2)*XC(3)+9.0D0*XC(2)**2)
FJACC(5,1)=1.0D0
FJACC(5,2)=((-0.1110223024625157D-15*XC(3))-2.2D0)/(XC(3)**2+4.399
&999999999999D0*XC(2)*XC(3)+4.83999999999998D0*XC(2)**2)
FJACC(5,3)=(0.1110223024625157D-15*XC(2)-1.0D0)/(XC(3)**2+4.399999
&999999999999D0*XC(2)*XC(3)+4.83999999999998D0*XC(2)**2)
FJACC(6,1)=1.0D0
FJACC(6,2)=((-0.2220446049250313D-15*XC(3))-1.66666666666667D0)/(
&XC(3)**2+3.33333333333333D0*XC(2)*XC(3)+2.77777777777777D0*XC(2)
&**2)
FJACC(6,3)=(0.2220446049250313D-15*XC(2)-1.0D0)/(XC(3)**2+3.333333
&333333333333D0*XC(2)*XC(3)+2.77777777777777D0*XC(2)**2)
FJACC(7,1)=1.0D0
FJACC(7,2)=((-0.5551115123125783D-16*XC(3))-1.285714285714286D0)/(
&XC(3)**2+2.571428571428571D0*XC(2)*XC(3)+1.653061224489796D0*XC(2)
&**2)
FJACC(7,3)=(0.5551115123125783D-16*XC(2)-1.0D0)/(XC(3)**2+2.571428
&571428571D0*XC(2)*XC(3)+1.653061224489796D0*XC(2)**2)
FJACC(8,1)=1.0D0
FJACC(8,2)=-1.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)
FJACC(8,3)=-1.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)
FJACC(9,1)=1.0D0
FJACC(9,2)=-1.285714285714286D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)*
&**2)
FJACC(9,3)=-1.285714285714286D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)*
&**2)
FJACC(10,1)=1.0D0
FJACC(10,2)=-1.66666666666667D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)
&**2)
FJACC(10,3)=-1.66666666666667D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)
&**2)
FJACC(11,1)=1.0D0
FJACC(11,2)=-2.2D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)
FJACC(11,3)=-2.2D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)
FJACC(12,1)=1.0D0
FJACC(12,2)=-3.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)
FJACC(12,3)=-3.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)
FJACC(13,1)=1.0D0
FJACC(13,2)=-4.33333333333333D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)
&**2)
FJACC(13,3)=-4.33333333333333D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)
&**2)
FJACC(14,1)=1.0D0
FJACC(14,2)=-7.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)
FJACC(14,3)=-7.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)
FJACC(15,1)=1.0D0
FJACC(15,2)=-15.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)

```

```

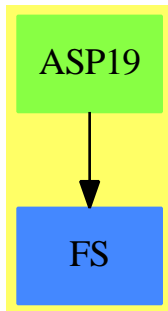
FJACC(15,3)=-15.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)
RETURN
END

```

See Also:

o)show Asp19

Asp19 (ASP19)



Exports:

coerce outputAsFortran retract retractIfCan

— domain ASP19 Asp19 —

```

)abbrev domain ASP19 Asp19
++ Author: Mike Dewar, Godfrey Nolan, Grant Keady
++ Date Created: Mar 1993
++ Date Last Updated: 6 October 1994
++ Description:
++\spadtype{Asp19} produces Fortran for Type 19 ASPs, evaluating a set of
++functions and their jacobian at a given point, for example:
++
++\tab{5}SUBROUTINE LSFUN2(M,N,XC,FVECC,FJACC,LJC)\br
++\tab{5}DOUBLE PRECISION FVECC(M),FJACC(LJC,N),XC(N)\br
++\tab{5}INTEGER M,N,LJC\br
++\tab{5}INTEGER I,J\br
++\tab{5}DO 25003 I=1,LJC\br
++\tab{7}DO 25004 J=1,N\br
++\tab{9}FJACC(I,J)=0.0D0\br
++25004 CONTINUE\br
++25003 CONTINUE\br
++\tab{5}FVECC(1)=(XC(1)-0.14D0)*XC(3)+(15.0D0*XC(1)-2.1D0)*XC(2)+1.0D0)/(\br

```



```

++\tab{4}&XC(3)+15.0D0*XC(2))\br
++\tab{5}FVECC(2)=((XC(1)-0.18D0)*XC(3)+(7.0D0*XC(1)-1.26D0)*XC(2)+1.0D0)/(\br
++\tab{4}&XC(3)+7.0D0*XC(2))\br
++\tab{5}FVECC(3)=((XC(1)-0.22D0)*XC(3)+(4.33333333333333D0*XC(1)-0.953333\br
++\tab{4}&3333333333D0)*XC(2)+1.0D0)/(XC(3)+4.33333333333333D0*XC(2))\br
++\tab{5}FVECC(4)=((XC(1)-0.25D0)*XC(3)+(3.0D0*XC(1)-0.75D0)*XC(2)+1.0D0)/(\br
++\tab{4}&XC(3)+3.0D0*XC(2))\br
++\tab{5}FVECC(5)=((XC(1)-0.29D0)*XC(3)+(2.2D0*XC(1)-0.637999999999999D0)*\br
++\tab{4}&XC(2)+1.0D0)/(XC(3)+2.2D0*XC(2))\br
++\tab{5}FVECC(6)=((XC(1)-0.32D0)*XC(3)+(1.666666666666667D0*XC(1)-0.533333\br
++\tab{4}&3333333333D0)*XC(2)+1.0D0)/(XC(3)+1.666666666666667D0*XC(2))\br
++\tab{5}FVECC(7)=((XC(1)-0.35D0)*XC(3)+(1.285714285714286D0*XC(1)-0.45D0)*\br
++\tab{4}&XC(2)+1.0D0)/(XC(3)+1.285714285714286D0*XC(2))\br
++\tab{5}FVECC(8)=((XC(1)-0.39D0)*XC(3)+(XC(1)-0.39D0)*XC(2)+1.0D0)/(XC(3)+\br
++\tab{4}&XC(2))\br
++\tab{5}FVECC(9)=((XC(1)-0.37D0)*XC(3)+(XC(1)-0.37D0)*XC(2)+1.285714285714\br
++\tab{4}&286D0)/(XC(3)+XC(2))\br
++\tab{5}FVECC(10)=((XC(1)-0.58D0)*XC(3)+(XC(1)-0.58D0)*XC(2)+1.66666666666\br
++\tab{4}&6667D0)/(XC(3)+XC(2))\br
++\tab{5}FVECC(11)=((XC(1)-0.73D0)*XC(3)+(XC(1)-0.73D0)*XC(2)+2.2D0)/(XC(3)\br
++\tab{4}&+XC(2))\br
++\tab{5}FVECC(12)=((XC(1)-0.96D0)*XC(3)+(XC(1)-0.96D0)*XC(2)+3.0D0)/(XC(3)\br
++\tab{4}&+XC(2))\br
++\tab{5}FVECC(13)=((XC(1)-1.34D0)*XC(3)+(XC(1)-1.34D0)*XC(2)+4.3333333333\br
++\tab{4}&3333D0)/(XC(3)+XC(2))\br
++\tab{5}FVECC(14)=((XC(1)-2.1D0)*XC(3)+(XC(1)-2.1D0)*XC(2)+7.0D0)/(XC(3)+X\br
++\tab{4}&C(2))\br
++\tab{5}FVECC(15)=((XC(1)-4.39D0)*XC(3)+(XC(1)-4.39D0)*XC(2)+15.0D0)/(XC(3)\br
++\tab{4}&)+XC(2))\br
++\tab{5}FJACC(1,1)=1.0D0\br
++\tab{5}FJACC(1,2)=-15.0D0/(XC(3)**2+30.0D0*XC(2)*XC(3)+225.0D0*XC(2)**2)\br
++\tab{5}FJACC(1,3)=-1.0D0/(XC(3)**2+30.0D0*XC(2)*XC(3)+225.0D0*XC(2)**2)\br
++\tab{5}FJACC(2,1)=1.0D0\br
++\tab{5}FJACC(2,2)=-7.0D0/(XC(3)**2+14.0D0*XC(2)*XC(3)+49.0D0*XC(2)**2)\br
++\tab{5}FJACC(2,3)=-1.0D0/(XC(3)**2+14.0D0*XC(2)*XC(3)+49.0D0*XC(2)**2)\br
++\tab{5}FJACC(3,1)=1.0D0\br
++\tab{5}FJACC(3,2)=((-0.1110223024625157D-15*XC(3))-4.33333333333333D0)/(\br
++\tab{4}&XC(3)**2+8.66666666666666D0*XC(2)*XC(3)+18.7777777777778D0*XC(2)\br
++\tab{4}&**2)\br
++\tab{5}FJACC(3,3)=(0.1110223024625157D-15*XC(2)-1.0D0)/(XC(3)**2+8.666666\br
++\tab{4}&66666666D0*XC(2)*XC(3)+18.7777777777778D0*XC(2)**2)\br
++\tab{5}FJACC(4,1)=1.0D0\br
++\tab{5}FJACC(4,2)=-3.0D0/(XC(3)**2+6.0D0*XC(2)*XC(3)+9.0D0*XC(2)**2)\br
++\tab{5}FJACC(4,3)=-1.0D0/(XC(3)**2+6.0D0*XC(2)*XC(3)+9.0D0*XC(2)**2)\br
++\tab{5}FJACC(5,1)=1.0D0\br
++\tab{5}FJACC(5,2)=((-0.1110223024625157D-15*XC(3))-2.2D0)/(XC(3)**2+4.399\br
++\tab{4}&9999999999D0*XC(2)*XC(3)+4.83999999999998D0*XC(2)**2)\br
++\tab{5}FJACC(5,3)=(0.1110223024625157D-15*XC(2)-1.0D0)/(XC(3)**2+4.399999\br
++\tab{4}&999999999D0*XC(2)*XC(3)+4.83999999999998D0*XC(2)**2)\br
++\tab{5}FJACC(6,1)=1.0D0\br

```

```

++\tab{5}FJACC(6,2)=((-0.2220446049250313D-15*XC(3))-1.666666666666667D0)/(\br
++\tab{4}&XC(3)**2+3.333333333333333D0*XC(2)*XC(3)+2.777777777777777D0*XC(2)\br
++\tab{4}&**2)\br
++\tab{5}FJACC(6,3)=(0.2220446049250313D-15*XC(2)-1.0D0)/(XC(3)**2+3.333333\br
++\tab{4}&333333333D0*XC(2)*XC(3)+2.777777777777777D0*XC(2)**2)\br
++\tab{5}FJACC(7,1)=1.0D0\br
++\tab{5}FJACC(7,2)=((-0.5551115123125783D-16*XC(3))-1.285714285714286D0)/(\br
++\tab{4}&XC(3)**2+2.571428571428571D0*XC(2)*XC(3)+1.653061224489796D0*XC(2)\br
++\tab{4}&**2)\br
++\tab{5}FJACC(7,3)=(0.5551115123125783D-16*XC(2)-1.0D0)/(XC(3)**2+2.571428\br
++\tab{4}&571428571D0*XC(2)*XC(3)+1.653061224489796D0*XC(2)**2)\br
++\tab{5}FJACC(8,1)=1.0D0\br
++\tab{5}FJACC(8,2)=-1.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)\br
++\tab{5}FJACC(8,3)=-1.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)\br
++\tab{5}FJACC(9,1)=1.0D0\br
++\tab{5}FJACC(9,2)=-1.285714285714286D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)*\br
++\tab{4}&**2)\br
++\tab{5}FJACC(9,3)=-1.285714285714286D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)*\br
++\tab{4}&**2)\br
++\tab{5}FJACC(10,1)=1.0D0\br
++\tab{5}FJACC(10,2)=-1.666666666666667D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)\br
++\tab{4}&**2)\br
++\tab{5}FJACC(10,3)=-1.666666666666667D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)\br
++\tab{4}&**2)\br
++\tab{5}FJACC(11,1)=1.0D0\br
++\tab{5}FJACC(11,2)=-2.2D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)\br
++\tab{5}FJACC(11,3)=-2.2D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)\br
++\tab{5}FJACC(12,1)=1.0D0\br
++\tab{5}FJACC(12,2)=-3.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)\br
++\tab{5}FJACC(12,3)=-3.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)\br
++\tab{5}FJACC(13,1)=1.0D0\br
++\tab{5}FJACC(13,2)=-4.333333333333333D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)\br
++\tab{4}&**2)\br
++\tab{5}FJACC(13,3)=-4.333333333333333D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)\br
++\tab{4}&**2)\br
++\tab{5}FJACC(14,1)=1.0D0\br
++\tab{5}FJACC(14,2)=-7.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)\br
++\tab{5}FJACC(14,3)=-7.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)\br
++\tab{5}FJACC(15,1)=1.0D0\br
++\tab{5}FJACC(15,2)=-15.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)\br
++\tab{5}FJACC(15,3)=-15.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)\br
++\tab{5}RETURN\br
++\tab{5}END

```

```

Asp19(name): Exports == Implementation where
name : Symbol

```

```

FST ==> FortranScalarType
FT   ==> FortranType
FC   ==> FortranCode

```

```

SYMTAB ==> SymbolTable
RSFC   ==> Record(localSymbols:SymbolTable,code:List(FC))
FSTU   ==> Union(fst:FST,void:"void")
FRAC   ==> Fraction
POLY   ==> Polynomial
EXPR   ==> Expression
INT    ==> Integer
FLOAT  ==> Float
MFLOAT ==> MachineFloat
VEC    ==> Vector
VF2    ==> VectorFunctions2
MF2    ==> MatrixCategoryFunctions2(FEXPR,VEC FEXPR,VEC FEXPR,Matrix FEXPR,EXPR MFLOAT,VEC EXP
FEXPR  ==> FortranExpression([],['XC'],MFLOAT)
S      ==> Symbol

Exports ==> FortranVectorFunctionCategory with
  coerce : VEC FEXPR -> $
    ++coerce(f) takes objects from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns them into an ASP.

Implementation ==> add

real : FSTU := ["real":FST]$FSTU
syms : SYMTAB := empty()$SYMTAB
declare!(M,fortranInteger()$FT,syms)$SYMTAB
declare!(N,fortranInteger()$FT,syms)$SYMTAB
declare!(LJC,fortranInteger()$FT,syms)$SYMTAB
xcType : FT := construct(real,[N],false)$FT
declare!(XC,xcType,syms)$SYMTAB
fveccType : FT := construct(real,[M],false)$FT
declare!(FVECC,fveccType,syms)$SYMTAB
fjaccType : FT := construct(real,[LJC,N],false)$FT
declare!(FJACC,fjaccType,syms)$SYMTAB
Rep := FortranProgram(name,["void"]$FSTU,[M,N,XC,FVECC,FJACC,LJC],syms)

coerce(c:List FC):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

coerce(c:FC):$ == coerce(c)$Rep

-- Take a symbol, pull of the script and turn it into an integer!!
o2int(u:S):Integer ==
  o : OutputForm := first elt(scripts(u)$S,sub)
  o pretend Integer

-- To help the poor old compiler!
fexpr2expr(u:FEXPR):EXPR MFLOAT == coerce(u)$FEXPR

localAssign1(s:S,j:Matrix FEXPR):FC ==

```

```

j' : Matrix EXPR MFLOAT := map(fexpr2expr,j)$MF2
assign(s,j')$FC

localAssign2(s:S,j:VEC FEXPR):FC ==
  j' : VEC EXPR MFLOAT := map(fexpr2expr,j)$VF2(FEXPR,EXPR MFLOAT)
  assign(s,j')$FC

coerce(u:VEC FEXPR):$ ==
  -- First zero the Jacobian matrix in case we miss some derivatives which
  -- are zero.
  import POLY INT
  seg1 : Segment (POLY INT) := segment(1::(POLY INT),LJC@S::(POLY INT))
  seg2 : Segment (POLY INT) := segment(1::(POLY INT),N@S::(POLY INT))
  s1 : SegmentBinding POLY INT := equation(I@S,seg1)
  s2 : SegmentBinding POLY INT := equation(J@S,seg2)
  as : FC := assign(FJACC,[I@S::(POLY INT),J@S::(POLY INT)],0.0::EXPR FLOAT)
  clear : FC := forLoop(s1,forLoop(s2,as))
  j:Integer
  x:S := XC::S
  pu:List(S) := []
  -- Work out which variables appear in the expressions
  for e in entries(u) repeat
    pu := setUnion(pu,variables(e)$FEXPR)
  scriptList : List Integer := map(o2int,pu)$ListFunctions2(S,Integer)
  -- This should be the maximum XC_n which occurs (there may be others
  -- which don't):
  n:Integer := reduce(max,scriptList)$List(Integer)
  p:List(S) := []
  for j in 1..n repeat p:= cons(subscript(x,[j::OutputForm])$S,p)
  p:= reverse(p)
  jac:Matrix(FEXPR) := _
  jacobian(u,p)$MultiVariableCalculusFunctions(S,FEXPR,VEC FEXPR,List(S))
  c1:FC := localAssign2(FVECC,u)
  c2:FC := localAssign1(FJACC,jac)
  [clear,c1,c2,returns()$List(FC)::]$

coerce(u:$):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

retract(u:VEC FRAC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY INT,FEXPR)

```

```

v case "failed" => "failed"
(v::VEC FEXPR):: $

retract(u:VEC FRAC POLY FLOAT):$ ==
v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY FLOAT,FEXPR)
v:: $

retractIfCan(u:VEC FRAC POLY FLOAT):Union($,"failed") ==
v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY FLOAT,FEXPR)
v case "failed" => "failed"
(v::VEC FEXPR):: $

retract(u:VEC EXPR INT):$ ==
v : VEC FEXPR := map(retract,u)$VF2(EXPR INT,FEXPR)
v:: $

retractIfCan(u:VEC EXPR INT):Union($,"failed") ==
v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR INT,FEXPR)
v case "failed" => "failed"
(v::VEC FEXPR):: $

retract(u:VEC EXPR FLOAT):$ ==
v : VEC FEXPR := map(retract,u)$VF2(EXPR FLOAT,FEXPR)
v:: $

retractIfCan(u:VEC EXPR FLOAT):Union($,"failed") ==
v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR FLOAT,FEXPR)
v case "failed" => "failed"
(v::VEC FEXPR):: $

retract(u:VEC POLY INT):$ ==
v : VEC FEXPR := map(retract,u)$VF2(POLY INT,FEXPR)
v:: $

retractIfCan(u:VEC POLY INT):Union($,"failed") ==
v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY INT,FEXPR)
v case "failed" => "failed"
(v::VEC FEXPR):: $

retract(u:VEC POLY FLOAT):$ ==
v : VEC FEXPR := map(retract,u)$VF2(POLY FLOAT,FEXPR)
v:: $

retractIfCan(u:VEC POLY FLOAT):Union($,"failed") ==
v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY FLOAT,FEXPR)
v case "failed" => "failed"
(v::VEC FEXPR):: $

```

— ASP19.dotabb —

```
"ASP19" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP19"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ASP19" -> "FS"
```

domain ASP20 Asp20

— Asp20.input —

```
)set break resume
)sys rm -f Asp20.output
)spool Asp20.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp20
--R
--R Asp20(name: Symbol) is a domain constructor
--R Abbreviation for Asp20 is ASP20
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP20
--R
--R----- Operations -----
--R coerce : FortranCode -> %               coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm               outputAsFortran : % -> Void
--R coerce : Matrix(FortranExpression([construct],[construct,QUOTEX,QUOTEHESS],MachineFloat)) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Matrix(Fraction(Polynomial(Integer))) -> %
--R retract : Matrix(Fraction(Polynomial(Float))) -> %
--R retract : Matrix(Polynomial(Integer)) -> %
--R retract : Matrix(Polynomial(Float)) -> %
--R retract : Matrix(Expression(Integer)) -> %
--R retract : Matrix(Expression(Float)) -> %
--R retractIfCan : Matrix(Fraction(Polynomial(Integer))) -> Union(%,"failed")
--R retractIfCan : Matrix(Fraction(Polynomial(Float))) -> Union(%,"failed")
--R retractIfCan : Matrix(Polynomial(Integer)) -> Union(%,"failed")
--R retractIfCan : Matrix(Polynomial(Float)) -> Union(%,"failed")
--R retractIfCan : Matrix(Expression(Integer)) -> Union(%,"failed")
--R retractIfCan : Matrix(Expression(Float)) -> Union(%,"failed")
--R
```

```
--E 1
```

```
)spool
)lisp (bye)
```

— Asp20.help —

```
=====
Asp20 examples
=====
```

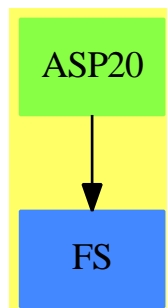
Asp20 produces Fortran for Type 20 ASPs, for example:

```
SUBROUTINE QPHESS(N,NROWH,NCOLH,JTHCOL,HESS,X,HX)
DOUBLE PRECISION HX(N),X(N),HESS(NROWH,NCOLH)
INTEGER JTHCOL,N,NROWH,NCOLH
HX(1)=2.0D0*X(1)
HX(2)=2.0D0*X(2)
HX(3)=2.0D0*X(4)+2.0D0*X(3)
HX(4)=2.0D0*X(4)+2.0D0*X(3)
HX(5)=2.0D0*X(5)
HX(6)=(-2.0D0*X(7))+(-2.0D0*X(6))
HX(7)=(-2.0D0*X(7))+(-2.0D0*X(6))
RETURN
END
```

See Also:

- o)show Asp20

Asp20 (ASP20)



Exports:

coerce outputAsFortran retract retractIfCan

— domain ASP20 Asp20 —

```
)abbrev domain ASP20 Asp20
++ Author: Mike Dewar and Godfrey Nolan and Grant Keady
++ Date Created: Dec 1993
++ Date Last Updated: 6 October 1994
++ Description:
++ \spadtype{Asp20} produces Fortran for Type 20 ASPs, for example:
++
++ \tab{5}SUBROUTINE QPHESS(N,NROWH,NCOLH,JTHCOL,HESS,X,HX)\br
++ \tab{5}DOUBLE PRECISION HX(N),X(N),HESS(NROWH,NCOLH)\br
++ \tab{5}INTEGER JTHCOL,N,NROWH,NCOLH\br
++ \tab{5}HX(1)=2.0D0*X(1)\br
++ \tab{5}HX(2)=2.0D0*X(2)\br
++ \tab{5}HX(3)=2.0D0*X(4)+2.0D0*X(3)\br
++ \tab{5}HX(4)=2.0D0*X(4)+2.0D0*X(3)\br
++ \tab{5}HX(5)=2.0D0*X(5)\br
++ \tab{5}HX(6)=(-2.0D0*X(7))+(-2.0D0*X(6))\br
++ \tab{5}HX(7)=(-2.0D0*X(7))+(-2.0D0*X(6))\br
++ \tab{5}RETURN\br
++ \tab{5}END
```

Asp20(name): Exports == Implementation where
name : Symbol

```
FST    ==> FortranScalarType
FT     ==> FortranType
SYMTAB ==> SymbolTable
PI     ==> PositiveInteger
UFST   ==> Union(fst:FST,void:"void")
RSFC   ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
FRAC   ==> Fraction
POLY   ==> Polynomial
EXPR   ==> Expression
INT    ==> Integer
FLOAT  ==> Float
VEC    ==> Vector
MAT    ==> Matrix
VF2    ==> VectorFunctions2
MFLOAT ==> MachineFloat
FEXPR  ==> FortranExpression([],['X','HESS'],MFLOAT)
0      ==> OutputForm
M2     ==> MatrixCategoryFunctions2
MF2a   ==> M2(FRAC POLY INT,VEC FRAC POLY INT,VEC FRAC POLY INT,
             MAT FRAC POLY INT,FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
MF2b   ==> M2(FRAC POLY FLOAT,VEC FRAC POLY FLOAT,VEC FRAC POLY FLOAT,
```



```

      MAT FRAC POLY FLOAT, FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
MF2c ==> M2(POLY INT,VEC POLY INT,VEC POLY INT,MAT POLY INT,
      FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
MF2d ==> M2(POLY FLOAT,VEC POLY FLOAT,VEC POLY FLOAT,
      MAT POLY FLOAT, FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
MF2e ==> M2(EXPR INT,VEC EXPR INT,VEC EXPR INT,MAT EXPR INT,
      FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
MF2f ==> M2(EXPR FLOAT,VEC EXPR FLOAT,VEC EXPR FLOAT,
      MAT EXPR FLOAT, FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)

Exports ==> FortranMatrixFunctionCategory with
  coerce: MAT FEXPR -> $
    ++coerce(f) takes objects from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns them into an ASP.

Implementation ==> add

  real : UFST := ["real":FST]$UFST
  syms : SYMTAB := empty()
  declare!(N,fortranInteger(),syms)$SYMTAB
  declare!(NROWH,fortranInteger(),syms)$SYMTAB
  declare!(NCOLH,fortranInteger(),syms)$SYMTAB
  declare!(JTHCOL,fortranInteger(),syms)$SYMTAB
  hessType : FT := construct(real,[NROWH,NCOLH],false)$FT
  declare!(HESS,hessType,syms)$SYMTAB
  xType : FT := construct(real,[N],false)$FT
  declare!(X,xType,syms)$SYMTAB
  declare!(HX,xType,syms)$SYMTAB
  Rep := FortranProgram(name,["void"]$UFST,
      [N,NROWH,NCOLH,JTHCOL,HESS,X,HX],syms)

  coerce(c:List FortranCode):$ == coerce(c)$Rep

  coerce(r:RSFC):$ == coerce(r)$Rep

  coerce(c:FortranCode):$ == coerce(c)$Rep

  -- To help the poor old compiler!
  fexpr2expr(u:FEXPR):EXPR MFLOAT == coerce(u)$FEXPR

  localAssign(s:Symbol,j:VEC FEXPR):FortranCode ==
    j' : VEC EXPR MFLOAT := map(fexpr2expr,j)$VF2(FEXPR,EXPR MFLOAT)
    assign(s,j')$FortranCode

  coerce(u:MAT FEXPR):$ ==
    j:Integer
    x:Symbol := X::Symbol
    n := nrows(u)::PI
    p:VEC FEXPR := [retract(subscript(x,[j::0])$Symbol)$FEXPR for j in 1..n]

```

```

prod:VEC FEXPR := u*p
([localAssign(HX,prod),returns()$FortranCode]$List(FortranCode))::$

retract(u:MAT FRAC POLY INT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2a
  v::$

retractIfCan(u:MAT FRAC POLY INT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2a
  v case "failed" => "failed"
  (v::MAT FEXPR):: $

retract(u:MAT FRAC POLY FLOAT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2b
  v::$

retractIfCan(u:MAT FRAC POLY FLOAT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2b
  v case "failed" => "failed"
  (v::MAT FEXPR):: $

retract(u:MAT EXPR INT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2e
  v::$

retractIfCan(u:MAT EXPR INT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2e
  v case "failed" => "failed"
  (v::MAT FEXPR):: $

retract(u:MAT EXPR FLOAT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2f
  v::$

retractIfCan(u:MAT EXPR FLOAT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2f
  v case "failed" => "failed"
  (v::MAT FEXPR):: $

retract(u:MAT POLY INT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2c
  v::$

retractIfCan(u:MAT POLY INT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2c
  v case "failed" => "failed"
  (v::MAT FEXPR):: $

retract(u:MAT POLY FLOAT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2d

```

```

v::$(

retractIfCan(u:MAT POLY FLOAT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2d
  v case "failed" => "failed"
  (v::MAT FEXPR)::$(

coerce(u:$):0 == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

```

— ASP20.dotabb —

```

"ASP20" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP20"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ASP20" -> "FS"

```

domain ASP24 Asp24

— Asp24.input —

```

)set break resume
)sys rm -f Asp24.output
)spool Asp24.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp24
--R
--R Asp24(name: Symbol) is a domain constructor
--R Abbreviation for Asp24 is ASP24
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP24
--R
--R----- Operations -----

```

```

--R coerce : FortranCode -> %          coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm          outputAsFortran : % -> Void
--R retract : Polynomial(Float) -> %    retract : Expression(Float) -> %
--R coerce : FortranExpression([construct],[construct,QUOTEXC],MachineFloat) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Fraction(Polynomial(Integer)) -> %
--R retract : Fraction(Polynomial(Float)) -> %
--R retract : Polynomial(Integer) -> %
--R retract : Expression(Integer) -> %
--R retractIfCan : Fraction(Polynomial(Integer)) -> Union(%, "failed")
--R retractIfCan : Fraction(Polynomial(Float)) -> Union(%, "failed")
--R retractIfCan : Polynomial(Integer) -> Union(%, "failed")
--R retractIfCan : Polynomial(Float) -> Union(%, "failed")
--R retractIfCan : Expression(Integer) -> Union(%, "failed")
--R retractIfCan : Expression(Float) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)

```

— Asp24.help —

=====

Asp24 examples

=====

Asp24 produces Fortran for Type 24 ASPs which evaluate a multivariate function at a point (needed for NAG routine e04jaf), for example:

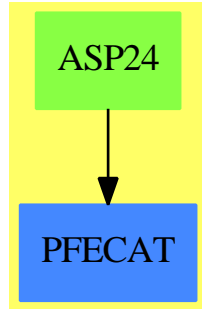
```

SUBROUTINE FUNCT1(N,XC,FC)
DOUBLE PRECISION FC,XC(N)
INTEGER N
FC=10.0D0*XC(4)**4+(-40.0D0*XC(1)*XC(4)**3)+(60.0D0*XC(1)**2+5
&.0D0)*XC(4)**2+((-10.0D0*XC(3))+(-40.0D0*XC(1)**3))*XC(4)+16.0D0*X
&C(3)**4+(-32.0D0*XC(2)*XC(3)**3)+(24.0D0*XC(2)**2+5.0D0)*XC(3)**2+
&(-8.0D0*XC(2)**3*XC(3))+XC(2)**4+100.0D0*XC(2)**2+20.0D0*XC(1)*XC(
&2)+10.0D0*XC(1)**4+XC(1)**2
RETURN
END

```

See Also:
o)show Asp24

Asp24 (ASP24)



Exports:

coerce outputAsFortran retract retractIfCan

— domain ASP24 Asp24 —

```

)abbrev domain ASP24 Asp24
++ Author: Mike Dewar, Grant Keady and Godfrey Nolan
++ Date Created: Mar 1993
++ Date Last Updated: 6 October 1994
++ Description:
++\spadtype{Asp24} produces Fortran for Type 24 ASPs which evaluate a
++multivariate function at a point (needed for NAG routine e04jaf),
++for example:
++
++\tab{5}SUBROUTINE FUNCT1(N,XC,FC)\br
++\tab{5}DOUBLE PRECISION FC,XC(N)\br
++\tab{5}INTEGER N\br
++\tab{5}FC=10.0D0*XC(4)**4+(-40.0D0*XC(1)*XC(4)**3)+(60.0D0*XC(1)**2+5\br
++\tab{4}&.0D0)*XC(4)**2+((-10.0D0*XC(3))+(-40.0D0*XC(1)**3))*XC(4)+16.0D0*X\br
++\tab{4}&C(3)**4+(-32.0D0*XC(2)*XC(3)**3)+(24.0D0*XC(2)**2+5.0D0)*XC(3)**2+\br
++\tab{4}&(-8.0D0*XC(2)**3*XC(3))+XC(2)**4+100.0D0*XC(2)**2+20.0D0*XC(1)*XC(\br
++\tab{4}&2)+10.0D0*XC(1)**4+XC(1)**2\br
++\tab{5}RETURN\br
++\tab{5}END\br
  
```

```

Asp24(name): Exports == Implementation where
  name : Symbol
  
```

```

FST ==> FortranScalarType
FT ==> FortranType
SYMTAB ==> SymbolTable
RSFC ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
FSTU ==> Union(fst:FST,void:"void")
FEXPR ==> FortranExpression([],['XC'],MachineFloat)
FRAC ==> Fraction
  
```

```

POLY    ==> Polynomial
EXPR    ==> Expression
INT      ==> Integer
FLOAT    ==> Float

Exports ==> FortranFunctionCategory with
  coerce : FEXPR -> $
  ++ coerce(f) takes an object from the appropriate instantiation of
    ++ \spadtype{FortranExpression} and turns it into an ASP.

Implementation ==> add

real : FSTU := ["real"::FST]$FSTU
syms : SYMTAB := empty()
declare!(N,fortranInteger(),syms)$SYMTAB
xcType : FT := construct(real,[N::Symbol],false)$FT
declare!(XC,xcType,syms)$SYMTAB
declare!(FC,fortranReal(),syms)$SYMTAB
Rep := FortranProgram(name,["void"]$FSTU,[N,XC,FC],syms)

coerce(c:List FortranCode):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

coerce(c:FortranCode):$ == coerce(c)$Rep

coerce(u:FEXPR):$ ==
  coerce(assign(FC,u::Expression(MachineFloat))$FortranCode)$Rep

retract(u:FRAC POLY INT):$ == (retract(u)@FEXPR)::$
retractIfCan(u:FRAC POLY INT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  (foo::FEXPR)::$

retract(u:FRAC POLY FLOAT):$ == (retract(u)@FEXPR)::$
retractIfCan(u:FRAC POLY FLOAT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  (foo::FEXPR)::$

retract(u:EXPR FLOAT):$ == (retract(u)@FEXPR)::$
retractIfCan(u:EXPR FLOAT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"

```

```

(foo::FEXPR)::$(

retract(u:EXPR INT):$ == (retract(u)@FEXPR)::$(
retractIfCan(u:EXPR INT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
(foo::FEXPR)::$(

retract(u:POLY FLOAT):$ == (retract(u)@FEXPR)::$(
retractIfCan(u:POLY FLOAT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
(foo::FEXPR)::$(

retract(u:POLY INT):$ == (retract(u)@FEXPR)::$(
retractIfCan(u:POLY INT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
(foo::FEXPR)::$(

coerce(u:$):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

```

— ASP24.dotabb —

```

"ASP24" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP24"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"ASP24" -> "PFECAT"

```

domain ASP27 Asp27

— Asp27.input —

```

)set break resume

```

```

)sys rm -f Asp27.output
)spool Asp27.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp27
--R
--R Asp27(name: Symbol) is a domain constructor
--R Abbreviation for Asp27 is ASP27
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP27
--R
--R----- Operations -----
--R coerce : FortranCode -> %               coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm               outputAsFortran : % -> Void
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R coerce : Matrix(MachineFloat) -> %
--R
--E 1

)spool
)lisp (bye)

```

— Asp27.help —

=====

Asp27 examples

=====

Asp27 produces Fortran for Type 27 ASPs, needed for NAG routine
f02fjf ,for example:

```

FUNCTION DOT(IFLAG,N,Z,W,RWORK,LRWORK,IWORK,LIWORK)
DOUBLE PRECISION W(N),Z(N),RWORK(LRWORK)
INTEGER N,LIWORK,IFLAG,LRWORK,IWORK(LIWORK)
DOT=(W(16)+(-0.5D0*W(15)))*Z(16)+((-0.5D0*W(16))+W(15)+(-0.5D0*W(1
&4)))*Z(15)+((-0.5D0*W(15))+W(14)+(-0.5D0*W(13)))*Z(14)+((-0.5D0*W(
&14))+W(13)+(-0.5D0*W(12)))*Z(13)+((-0.5D0*W(13))+W(12)+(-0.5D0*W(1
&11)))*Z(12)+((-0.5D0*W(12))+W(11)+(-0.5D0*W(10)))*Z(11)+((-0.5D0*W(
&11))+W(10)+(-0.5D0*W(9)))*Z(10)+((-0.5D0*W(10))+W(9)+(-0.5D0*W(8))
&)*Z(9)+((-0.5D0*W(9))+W(8)+(-0.5D0*W(7)))*Z(8)+((-0.5D0*W(8))+W(7)
&+(-0.5D0*W(6)))*Z(7)+((-0.5D0*W(7))+W(6)+(-0.5D0*W(5)))*Z(6)+((-0.
&5D0*W(6))+W(5)+(-0.5D0*W(4)))*Z(5)+((-0.5D0*W(5))+W(4)+(-0.5D0*W(3
&)))Z(4)+((-0.5D0*W(4))+W(3)+(-0.5D0*W(2)))*Z(3)+((-0.5D0*W(3))+W(
&2)+(-0.5D0*W(1)))*Z(2)+((-0.5D0*W(2))+W(1))*Z(1)

```



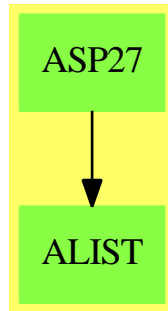
```

RETURN
END

```

See Also:
o)show Asp27

Asp27 (ASP27)



Exports:

coerce outputAsFortran

— domain ASP27 Asp27 —

```

)abbrev domain ASP27 Asp27
++ Author: Mike Dewar and Godfrey Nolan
++ Date Created: Nov 1993
++ Date Last Updated: 6 October 1994
++ Description:
++\spadtype{Asp27} produces Fortran for Type 27 ASPs, needed for NAG routine
++f02fjf ,for example:
++
++\tab{5}FUNCTION DOT(IFLAG,N,Z,W,RWORK,LRWORK,IWORK,LIWORK)\br
++\tab{5}DOUBLE PRECISION W(N),Z(N),RWORK(LRWORK)\br
++\tab{5}INTEGER N,LIWORK,IFLAG,LRWORK,IWORK(LIWORK)\br
++\tab{5}DOT=(W(16)+(-0.5D0*W(15)))*Z(16)+((-0.5D0*W(16))+W(15)+(-0.5D0*W(1)\br
++\tab{4}&4))*Z(15)+((-0.5D0*W(15))+W(14)+(-0.5D0*W(13)))*Z(14)+((-0.5D0*W(\br
++\tab{4}&14))+W(13)+(-0.5D0*W(12)))*Z(13)+((-0.5D0*W(13))+W(12)+(-0.5D0*W(1\br
++\tab{4}&1))*Z(12)+((-0.5D0*W(12))+W(11)+(-0.5D0*W(10)))*Z(11)+((-0.5D0*W(\br
++\tab{4}&11))+W(10)+(-0.5D0*W(9)))*Z(10)+((-0.5D0*W(10))+W(9)+(-0.5D0*W(8))\br
++\tab{4}&)*Z(9)+((-0.5D0*W(9))+W(8)+(-0.5D0*W(7)))*Z(8)+((-0.5D0*W(8))+W(7)\br
++\tab{4}&+(-0.5D0*W(6)))*Z(7)+((-0.5D0*W(7))+W(6)+(-0.5D0*W(5)))*Z(6)+((-0.\br
++\tab{4}&5D0*W(6))+W(5)+(-0.5D0*W(4)))*Z(5)+((-0.5D0*W(5))+W(4)+(-0.5D0*W(3\br

```

```

++\tab{4}&)))*Z(4)+((-0.5D0*W(4))+W(3)+(-0.5D0*W(2)))*Z(3)+((-0.5D0*W(3))+W(\br
++\tab{4}&2))+(-0.5D0*W(1))*Z(2)+((-0.5D0*W(2))+W(1))*Z(1)\br
++\tab{5}RETURN\br
++\tab{5}END

```

```

Asp27(name): Exports == Implementation where
  name : Symbol

```

```

0      ==> OutputForm
FST    ==> FortranScalarType
FT     ==> FortranType
SYMTAB ==> SymbolTable
UFST   ==> Union(fst:FST,void:"void")
FC     ==> FortranCode
PI     ==> PositiveInteger
RSFC   ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
EXPR   ==> Expression
MAT     ==> Matrix
MFLOAT ==> MachineFloat

```

```

Exports == FortranMatrixCategory

```

```

Implementation == add

```

```

real : UFST := ["real":FST]$UFST
integer : UFST := ["integer":FST]$UFST
syms : SYMTAB := empty()$SYMTAB
declare!(IFLAG,fortranInteger(),syms)$SYMTAB
declare!(N,fortranInteger(),syms)$SYMTAB
declare!(LRWORK,fortranInteger(),syms)$SYMTAB
declare!(LIWORK,fortranInteger(),syms)$SYMTAB
zType : FT := construct(real,[N],false)$FT
declare!(Z,zType,syms)$SYMTAB
declare!(W,zType,syms)$SYMTAB
rType : FT := construct(real,[LRWORK],false)$FT
declare!(RWORK,rType,syms)$SYMTAB
iType : FT := construct(integer,[LIWORK],false)$FT
declare!(IWORK,iType,syms)$SYMTAB
Rep := FortranProgram(name,real,
                      [IFLAG,N,Z,W,RWORK,LRWORK,IWORK,LIWORK],syms)

```

```

-- To help the poor old compiler!
localCoerce(u:Symbol):EXPR(MFLOAT) == coerce(u)$EXPR(MFLOAT)

```

```

coerce (u:MAT MFLOAT):$ ==
  Ws: Symbol := W
  Zs: Symbol := Z

```

```

code : List FC
l:EXPR MFLOAT := "+"/_
  [("+"/[localCoerce(elt(Ws,[j::0])$Symbol) * u(j,i)]_
                                     for j in 1..nrows(u)::PI)]_
  *localCoerce(elt(Zs,[i::0])$Symbol) for i in 1..ncols(u)::PI]
c := assign(name,l)$FC
code := [c,returns()]$List(FC)
code::$

coerce(c:List FortranCode):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

coerce(c:FortranCode):$ == coerce(c)$Rep

coerce(u:$):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

```

— ASP27.dotabb —

```

"ASP27" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP27"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ASP27" -> "ALIST"

```

domain ASP28 Asp28

— Asp28.input —

```

)set break resume
)sys rm -f Asp28.output
)spool Asp28.output
)set message test on
)set message auto off
)clear all

```

--S 1 of 1

```

)show Asp28
--R
--R Asp28(name: Symbol) is a domain constructor
--R Abbreviation for Asp28 is ASP28
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP28
--R
--R----- Operations -----
--R coerce : FortranCode -> %           coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm           outputAsFortran : % -> Void
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R coerce : Matrix(MachineFloat) -> %
--R
--E 1

)spool
)lisp (bye)

```

— Asp28.help —

=====

Asp28 examples

=====

Asp28 produces Fortran for Type 28 ASPs, used in NAG routine f02fjf,
for example:

```

SUBROUTINE IMAGE(IFLAG,N,Z,W,RWORK,LRWORK,IWORK,LIWORK)
DOUBLE PRECISION Z(N),W(N),IWORK(LRWORK),RWORK(LRWORK)
INTEGER N,LIWORK,IFLAG,LRWORK
W(1)=0.01707454969713436D0*Z(16)+0.001747395874954051D0*Z(15)+0.00
&2106973900813502D0*Z(14)+0.002957434991769087D0*Z(13)+(-0.00700554
&0882865317D0*Z(12))+(-0.01219194009813166D0*Z(11))+0.0037230647365
&3087D0*Z(10)+0.04932374658377151D0*Z(9)+(-0.03586220812223305D0*Z(
&8))+(-0.04723268012114625D0*Z(7))+(-0.02434652144032987D0*Z(6))+0.
&2264766947290192D0*Z(5)+(-0.1385343580686922D0*Z(4))+(-0.116530050
&8238904D0*Z(3))+(-0.2803531651057233D0*Z(2))+1.019463911841327D0*Z
&(1)
W(2)=0.0227345011107737D0*Z(16)+0.008812321197398072D0*Z(15)+0.010
&94012210519586D0*Z(14)+(-0.01764072463999744D0*Z(13))+(-0.01357136
&72105995D0*Z(12))+0.00157466157362272D0*Z(11)+0.05258889186338282D
&0*Z(10)+(-0.01981532388243379D0*Z(9))+(-0.06095390688679697D0*Z(8)
&)+(-0.04153119955569051D0*Z(7))+0.2176561076571465D0*Z(6)+(-0.0532
&5555586632358D0*Z(5))+(-0.1688977368984641D0*Z(4))+(-0.32440166056
&67343D0*Z(3))+0.9128222941872173D0*Z(2)+(-0.2419652703415429D0*Z(1
&))
W(3)=0.03371198197190302D0*Z(16)+0.02021603150122265D0*Z(15)+(-0.0

```

$\&06607305534689702D0*Z(14)) + (-0.03032392238968179D0*Z(13)) + 0.002033$
 $\&305231024948D0*Z(12)) + 0.05375944956767728D0*Z(11)) + (-0.0163213312502$
 $\&9967D0*Z(10)) + (-0.05483186562035512D0*Z(9)) + (-0.04901428822579872D$
 $\&0*Z(8)) + 0.2091097927887612D0*Z(7)) + (-0.05760560341383113D0*Z(6)) + (-$
 $\&0.1236679206156403D0*Z(5)) + (-0.3523683853026259D0*Z(4)) + 0.88929961$
 $\&32269974D0*Z(3)) + (-0.2995429545781457D0*Z(2)) + (-0.02986582812574917$
 $\&D0*Z(1))$
 $W(4) = 0.05141563713660119D0*Z(16) + 0.005239165960779299D0*Z(15) + (-0.$
 $\&01623427735779699D0*Z(14)) + (-0.01965809746040371D0*Z(13)) + 0.054688$
 $\&97337339577D0*Z(12)) + (-0.014224695935687D0*Z(11)) + (-0.0505181779315$
 $\&6355D0*Z(10)) + (-0.04353074206076491D0*Z(9)) + 0.2012230497530726D0*Z$
 $\&(8)) + (-0.06630874514535952D0*Z(7)) + (-0.1280829963720053D0*Z(6)) + (-0$
 $\&.305169742604165D0*Z(5)) + 0.8600427128450191D0*Z(4)) + (-0.32415033802$
 $\&68184D0*Z(3)) + (-0.09033531980693314D0*Z(2)) + 0.09089205517109111D0*$
 $\&Z(1)$
 $W(5) = 0.04556369767776375D0*Z(16) + (-0.001822737697581869D0*Z(15)) + (-$
 $\&-0.002512226501941856D0*Z(14)) + 0.02947046460707379D0*Z(13)) + (-0.014$
 $\&45079632086177D0*Z(12)) + (-0.05034242196614937D0*Z(11)) + (-0.0376966$
 $\&3291725935D0*Z(10)) + 0.2171103102175198D0*Z(9)) + (-0.0824949256021352$
 $\&4D0*Z(8)) + (-0.1473995209288945D0*Z(7)) + (-0.315042193418466D0*Z(6))$
 $\&+0.9591623347824002D0*Z(5)) + (-0.3852396953763045D0*Z(4)) + (-0.141718$
 $\&5427288274D0*Z(3)) + (-0.03423495461011043D0*Z(2)) + 0.319820917706851$
 $\&6D0*Z(1)$
 $W(6) = 0.04015147277405744D0*Z(16) + 0.01328585741341559D0*Z(15) + 0.048$
 $\&26082005465965D0*Z(14)) + (-0.04319641116207706D0*Z(13)) + (-0.04931323$
 $\&319055762D0*Z(12)) + (-0.03526886317505474D0*Z(11)) + 0.22295383396730$
 $\&01D0*Z(10)) + (-0.07375317649315155D0*Z(9)) + (-0.1589391311991561D0*Z($
 $\&8)) + (-0.328001910890377D0*Z(7)) + 0.952576555482747D0*Z(6)) + (-0.31583$
 $\&09975786731D0*Z(5)) + (-0.1846882042225383D0*Z(4)) + (-0.0703762046700$
 $\&4427D0*Z(3)) + 0.2311852964327382D0*Z(2)) + 0.04254083491825025D0*Z(1)$
 $W(7) = 0.06069778964023718D0*Z(16) + 0.06681263884671322D0*Z(15) + (-0.0$
 $\&2113506688615768D0*Z(14)) + (-0.083996867458326D0*Z(13)) + (-0.0329843$
 $\&8523869648D0*Z(12)) + 0.2276878326327734D0*Z(11)) + (-0.067356038933017$
 $\&95D0*Z(10)) + (-0.1559813965382218D0*Z(9)) + (-0.3363262957694705D0*Z($
 $\&8)) + 0.9442791158560948D0*Z(7)) + (-0.3199955249404657D0*Z(6)) + (-0.136$
 $\&2463839920727D0*Z(5)) + (-0.1006185171570586D0*Z(4)) + 0.2057504515015$
 $\&423D0*Z(3)) + (-0.02065879269286707D0*Z(2)) + 0.03160990266745513D0*Z(1$
 $\&)$
 $W(8) = 0.126386868896738D0*Z(16) + 0.002563370039476418D0*Z(15) + (-0.05$
 $\&581757739455641D0*Z(14)) + (-0.07777893205900685D0*Z(13)) + 0.23117338$
 $\&45834199D0*Z(12)) + (-0.06031581134427592D0*Z(11)) + (-0.14805474755869$
 $\&52D0*Z(10)) + (-0.3364014128402243D0*Z(9)) + 0.9364014128402244D0*Z(8)$
 $\&+(-0.3269452524413048D0*Z(7)) + (-0.1396841886557241D0*Z(6)) + (-0.056$
 $\&1733845834199D0*Z(5)) + 0.1777789320590069D0*Z(4)) + (-0.04418242260544$
 $\&359D0*Z(3)) + (-0.02756337003947642D0*Z(2)) + 0.07361313110326199D0*Z($
 $\&1)$
 $W(9) = 0.07361313110326199D0*Z(16) + (-0.02756337003947642D0*Z(15)) + (-$
 $\&0.04418242260544359D0*Z(14)) + 0.1777789320590069D0*Z(13)) + (-0.056173$
 $\&3845834199D0*Z(12)) + (-0.1396841886557241D0*Z(11)) + (-0.326945252441$
 $\&3048D0*Z(10)) + 0.9364014128402244D0*Z(9)) + (-0.3364014128402243D0*Z(8$

$\&)) + (-0.1480547475586952D0 * Z(7)) + (-0.06031581134427592D0 * Z(6)) + 0.23$
 $\&11733845834199D0 * Z(5)) + (-0.07777893205900685D0 * Z(4)) + (-0.0558175773$
 $\&9455641D0 * Z(3)) + 0.002563370039476418D0 * Z(2) + 0.126386868896738D0 * Z($
 $\&1)$
 $W(10) = 0.03160990266745513D0 * Z(16) + (-0.02065879269286707D0 * Z(15)) + 0$
 $\&.2057504515015423D0 * Z(14) + (-0.1006185171570586D0 * Z(13)) + (-0.136246$
 $\&3839920727D0 * Z(12)) + (-0.3199955249404657D0 * Z(11)) + 0.94427911585609$
 $\&48D0 * Z(10) + (-0.3363262957694705D0 * Z(9)) + (-0.1559813965382218D0 * Z(8$
 $\&)) + (-0.06735603893301795D0 * Z(7)) + 0.2276878326327734D0 * Z(6) + (-0.032$
 $\&98438523869648D0 * Z(5)) + (-0.083996867458326D0 * Z(4)) + (-0.02113506688$
 $\&615768D0 * Z(3)) + 0.06681263884671322D0 * Z(2) + 0.06069778964023718D0 * Z($
 $\&1)$
 $W(11) = 0.04254083491825025D0 * Z(16) + 0.2311852964327382D0 * Z(15) + (-0.0$
 $\&7037620467004427D0 * Z(14)) + (-0.1846882042225383D0 * Z(13)) + (-0.315830$
 $\&9975786731D0 * Z(12)) + 0.952576555482747D0 * Z(11) + (-0.328001910890377D$
 $\&0 * Z(10)) + (-0.1589391311991561D0 * Z(9)) + (-0.07375317649315155D0 * Z(8)$
 $\&) + 0.2229538339673001D0 * Z(7) + (-0.03526886317505474D0 * Z(6)) + (-0.0493$
 $\&1323319055762D0 * Z(5)) + (-0.04319641116207706D0 * Z(4)) + 0.048260820054$
 $\&65965D0 * Z(3) + 0.01328585741341559D0 * Z(2) + 0.04015147277405744D0 * Z(1)$
 $W(12) = 0.3198209177068516D0 * Z(16) + (-0.03423495461011043D0 * Z(15)) + (-$
 $\&0.1417185427288274D0 * Z(14)) + (-0.3852396953763045D0 * Z(13)) + 0.959162$
 $\&3347824002D0 * Z(12) + (-0.315042193418466D0 * Z(11)) + (-0.14739952092889$
 $\&45D0 * Z(10)) + (-0.08249492560213524D0 * Z(9)) + 0.2171103102175198D0 * Z(8$
 $\&) + (-0.03769663291725935D0 * Z(7)) + (-0.05034242196614937D0 * Z(6)) + (-0.$
 $\&01445079632086177D0 * Z(5)) + 0.02947046460707379D0 * Z(4) + (-0.002512226$
 $\&501941856D0 * Z(3)) + (-0.001822737697581869D0 * Z(2)) + 0.045563697677763$
 $\&75D0 * Z(1)$
 $W(13) = 0.09089205517109111D0 * Z(16) + (-0.09033531980693314D0 * Z(15)) + ($
 $\&-0.3241503380268184D0 * Z(14)) + 0.8600427128450191D0 * Z(13) + (-0.305169$
 $\&742604165D0 * Z(12)) + (-0.1280829963720053D0 * Z(11)) + (-0.0663087451453$
 $\&5952D0 * Z(10)) + 0.2012230497530726D0 * Z(9) + (-0.04353074206076491D0 * Z($
 $\&8)) + (-0.05051817793156355D0 * Z(7)) + (-0.014224695935687D0 * Z(6)) + 0.05$
 $\&468897337339577D0 * Z(5) + (-0.01965809746040371D0 * Z(4)) + (-0.016234277$
 $\&35779699D0 * Z(3)) + 0.005239165960779299D0 * Z(2) + 0.05141563713660119D0$
 $\&*Z(1)$
 $W(14) = (-0.02986582812574917D0 * Z(16)) + (-0.2995429545781457D0 * Z(15))$
 $\&+0.8892996132269974D0 * Z(14) + (-0.3523683853026259D0 * Z(13)) + (-0.1236$
 $\&679206156403D0 * Z(12)) + (-0.05760560341383113D0 * Z(11)) + 0.20910979278$
 $\&87612D0 * Z(10) + (-0.04901428822579872D0 * Z(9)) + (-0.05483186562035512D$
 $\&0 * Z(8)) + (-0.01632133125029967D0 * Z(7)) + 0.05375944956767728D0 * Z(6) + 0$
 $\&.002033305231024948D0 * Z(5) + (-0.03032392238968179D0 * Z(4)) + (-0.00660$
 $\&7305534689702D0 * Z(3)) + 0.02021603150122265D0 * Z(2) + 0.033711981971903$
 $\&02D0 * Z(1)$
 $W(15) = (-0.2419652703415429D0 * Z(16)) + 0.9128222941872173D0 * Z(15) + (-0$
 $\&.3244016605667343D0 * Z(14)) + (-0.1688977368984641D0 * Z(13)) + (-0.05325$
 $\&555586632358D0 * Z(12)) + 0.2176561076571465D0 * Z(11) + (-0.0415311995556$
 $\&9051D0 * Z(10)) + (-0.06095390688679697D0 * Z(9)) + (-0.01981532388243379D$
 $\&0 * Z(8)) + 0.05258889186338282D0 * Z(7) + 0.00157466157362272D0 * Z(6) + (-0.$
 $\&0135713672105995D0 * Z(5)) + (-0.01764072463999744D0 * Z(4)) + 0.010940122$
 $\&10519586D0 * Z(3) + 0.008812321197398072D0 * Z(2) + 0.0227345011107737D0 * Z$

```

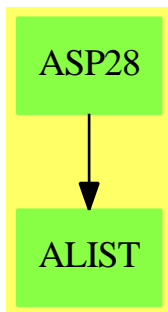
&(1)
W(16)=1.019463911841327D0*Z(16)+(-0.2803531651057233D0*Z(15))+(-0.
&1165300508238904D0*Z(14))+(-0.1385343580686922D0*Z(13))+0.22647669
&47290192D0*Z(12))+(-0.02434652144032987D0*Z(11))+(-0.04723268012114
&625D0*Z(10))+(-0.03586220812223305D0*Z(9))+0.04932374658377151D0*Z
&(8))+0.00372306473653087D0*Z(7))+(-0.01219194009813166D0*Z(6))+(-0.0
&07005540882865317D0*Z(5))+0.002957434991769087D0*Z(4))+0.0021069739
&00813502D0*Z(3))+0.001747395874954051D0*Z(2))+0.01707454969713436D0*
&Z(1)
RETURN
END

```

See Also:

o)show Asp28

Asp28 (ASP28)



Exports:

coerce outputAsFortran

— domain ASP28 Asp28 —

```

)abbrev domain ASP28 Asp28
++ Author: Mike Dewar
++ Date Created: 21 March 1994
++ Date Last Updated: 6 October 1994
++ Description:
++\spadtype{Asp28} produces Fortran for Type 28 ASPs, used in NAG routine
++f02fjf, for example:
++
++\tab{5}SUBROUTINE IMAGE(IFLAG,N,Z,W,RWORK,LRWORK,IWORK,LIWORK)\br
++\tab{5}DOUBLE PRECISION Z(N),W(N),IWORK(LRWORK),RWORK(LRWORK)\br

```

```

++\tab{5}INTEGER N,LIWORK,IFLAG,LRWORK\br
++\tab{5}W(1)=0.01707454969713436D0*Z(16)+0.001747395874954051D0*Z(15)+0.00\br
++\tab{4}&2106973900813502D0*Z(14)+0.002957434991769087D0*Z(13)+(-0.00700554\br
++\tab{4}&0882865317D0*Z(12))+(-0.01219194009813166D0*Z(11))+0.0037230647365\br
++\tab{4}&3087D0*Z(10)+0.04932374658377151D0*Z(9)+(-0.03586220812223305D0*Z(\br
++\tab{4}&8))+(-0.04723268012114625D0*Z(7))+(-0.02434652144032987D0*Z(6))+0.\br
++\tab{4}&2264766947290192D0*Z(5)+(-0.1385343580686922D0*Z(4))+(-0.116530050\br
++\tab{4}&8238904D0*Z(3))+(-0.2803531651057233D0*Z(2))+1.019463911841327D0*Z\br
++\tab{4}&(1)\br
++\tab{5}W(2)=0.0227345011107737D0*Z(16)+0.008812321197398072D0*Z(15)+0.010\br
++\tab{4}&94012210519586D0*Z(14)+(-0.01764072463999744D0*Z(13))+(-0.01357136\br
++\tab{4}&72105995D0*Z(12))+0.00157466157362272D0*Z(11)+0.05258889186338282D\br
++\tab{4}&0*Z(10)+(-0.01981532388243379D0*Z(9))+(-0.06095390688679697D0*Z(8)\br
++\tab{4}&)+(-0.04153119955569051D0*Z(7))+0.2176561076571465D0*Z(6)+(-0.0532\br
++\tab{4}&5555586632358D0*Z(5))+(-0.1688977368984641D0*Z(4))+(-0.32440166056\br
++\tab{4}&67343D0*Z(3))+0.9128222941872173D0*Z(2))+(-0.2419652703415429D0*Z(1\br
++\tab{4}&))\br
++\tab{5}W(3)=0.03371198197190302D0*Z(16)+0.02021603150122265D0*Z(15)+(-0.0\br
++\tab{4}&06607305534689702D0*Z(14))+(-0.03032392238968179D0*Z(13))+0.002033\br
++\tab{4}&305231024948D0*Z(12)+0.05375944956767728D0*Z(11)+(-0.0163213312502\br
++\tab{4}&9967D0*Z(10))+(-0.05483186562035512D0*Z(9))+(-0.04901428822579872D\br
++\tab{4}&0*Z(8))+0.2091097927887612D0*Z(7))+(-0.05760560341383113D0*Z(6))+(-\br
++\tab{4}&0.1236679206156403D0*Z(5))+(-0.3523683853026259D0*Z(4))+0.88929961\br
++\tab{4}&32269974D0*Z(3))+(-0.2995429545781457D0*Z(2))+(-0.02986582812574917\br
++\tab{4}&D0*Z(1))\br
++\tab{5}W(4)=0.05141563713660119D0*Z(16)+0.005239165960779299D0*Z(15)+(-0.\br
++\tab{4}&01623427735779699D0*Z(14))+(-0.01965809746040371D0*Z(13))+0.054688\br
++\tab{4}&97337339577D0*Z(12))+(-0.014224695935687D0*Z(11))+(-0.0505181779315\br
++\tab{4}&6355D0*Z(10))+(-0.04353074206076491D0*Z(9))+0.2012230497530726D0*Z\br
++\tab{4}&(8))+(-0.06630874514535952D0*Z(7))+(-0.1280829963720053D0*Z(6))+(-0\br
++\tab{4}&.305169742604165D0*Z(5))+0.8600427128450191D0*Z(4))+(-0.32415033802\br
++\tab{4}&68184D0*Z(3))+(-0.09033531980693314D0*Z(2))+0.09089205517109111D0*\br
++\tab{4}&Z(1)\br
++\tab{5}W(5)=0.04556369767776375D0*Z(16)+(-0.001822737697581869D0*Z(15))+(\br
++\tab{4}&-0.002512226501941856D0*Z(14))+0.02947046460707379D0*Z(13))+(-0.014\br
++\tab{4}&45079632086177D0*Z(12))+(-0.05034242196614937D0*Z(11))+(-0.0376966\br
++\tab{4}&3291725935D0*Z(10))+0.2171103102175198D0*Z(9))+(-0.0824949256021352\br
++\tab{4}&4D0*Z(8))+(-0.1473995209288945D0*Z(7))+(-0.315042193418466D0*Z(6))\br
++\tab{4}&+0.9591623347824002D0*Z(5))+(-0.3852396953763045D0*Z(4))+(-0.141718\br
++\tab{4}&5427288274D0*Z(3))+(-0.03423495461011043D0*Z(2))+0.319820917706851\br
++\tab{4}&6D0*Z(1)\br
++\tab{5}W(6)=0.04015147277405744D0*Z(16)+0.01328585741341559D0*Z(15)+0.048\br
++\tab{4}&26082005465965D0*Z(14))+(-0.04319641116207706D0*Z(13))+(-0.04931323\br
++\tab{4}&319055762D0*Z(12))+(-0.03526886317505474D0*Z(11))+0.22295383396730\br
++\tab{4}&01D0*Z(10))+(-0.07375317649315155D0*Z(9))+(-0.1589391311991561D0*Z(\br
++\tab{4}&8))+(-0.328001910890377D0*Z(7))+0.952576555482747D0*Z(6))+(-0.31583\br
++\tab{4}&09975786731D0*Z(5))+(-0.1846882042225383D0*Z(4))+(-0.0703762046700\br
++\tab{4}&4427D0*Z(3))+0.2311852964327382D0*Z(2))+0.04254083491825025D0*Z(1)\br
++\tab{5}W(7)=0.06069778964023718D0*Z(16)+0.06681263884671322D0*Z(15)+(-0.0\br
++\tab{4}&2113506688615768D0*Z(14))+(-0.083996867458326D0*Z(13))+(-0.0329843\br

```



```

++\tab{4}&8523869648D0*Z(12))+0.2276878326327734D0*Z(11))+(-0.067356038933017\br
++\tab{4}&95D0*Z(10))+(-0.1559813965382218D0*Z(9))+(-0.3363262957694705D0*Z(\br
++\tab{4}&8))+0.9442791158560948D0*Z(7))+(-0.3199955249404657D0*Z(6))+(-0.136\br
++\tab{4}&2463839920727D0*Z(5))+(-0.1006185171570586D0*Z(4))+0.2057504515015\br
++\tab{4}&423D0*Z(3))+(-0.02065879269286707D0*Z(2))+0.03160990266745513D0*Z(1\br
++\tab{4}&)\br
++\tab{5}W(8)=0.126386868896738D0*Z(16)+0.002563370039476418D0*Z(15))+(-0.05\br
++\tab{4}&581757739455641D0*Z(14))+(-0.07777893205900685D0*Z(13))+0.23117338\br
++\tab{4}&45834199D0*Z(12))+(-0.06031581134427592D0*Z(11))+(-0.14805474755869\br
++\tab{4}&52D0*Z(10))+(-0.3364014128402243D0*Z(9))+0.9364014128402244D0*Z(8)\br
++\tab{4}&+(-0.3269452524413048D0*Z(7))+(-0.1396841886557241D0*Z(6))+(-0.056\br
++\tab{4}&1733845834199D0*Z(5))+0.1777789320590069D0*Z(4))+(-0.04418242260544\br
++\tab{4}&359D0*Z(3))+(-0.02756337003947642D0*Z(2))+0.07361313110326199D0*Z(\br
++\tab{4}&1)\br
++\tab{5}W(9)=0.07361313110326199D0*Z(16))+(-0.02756337003947642D0*Z(15))+(-\br
++\tab{4}&0.04418242260544359D0*Z(14))+0.1777789320590069D0*Z(13))+(-0.056173\br
++\tab{4}&3845834199D0*Z(12))+(-0.1396841886557241D0*Z(11))+(-0.326945252441\br
++\tab{4}&3048D0*Z(10))+0.9364014128402244D0*Z(9))+(-0.3364014128402243D0*Z(8\br
++\tab{4}&))+(-0.1480547475586952D0*Z(7))+(-0.06031581134427592D0*Z(6))+0.23\br
++\tab{4}&11733845834199D0*Z(5))+(-0.07777893205900685D0*Z(4))+(-0.0558175773\br
++\tab{4}&9455641D0*Z(3))+0.002563370039476418D0*Z(2))+0.126386868896738D0*Z(\br
++\tab{4}&1)\br
++\tab{5}W(10)=0.03160990266745513D0*Z(16))+(-0.02065879269286707D0*Z(15))+0\br
++\tab{4}&.2057504515015423D0*Z(14))+(-0.1006185171570586D0*Z(13))+(-0.136246\br
++\tab{4}&3839920727D0*Z(12))+(-0.3199955249404657D0*Z(11))+0.94427911585609\br
++\tab{4}&48D0*Z(10))+(-0.3363262957694705D0*Z(9))+(-0.1559813965382218D0*Z(8\br
++\tab{4}&))+(-0.06735603893301795D0*Z(7))+0.2276878326327734D0*Z(6))+(-0.032\br
++\tab{4}&98438523869648D0*Z(5))+(-0.083996867458326D0*Z(4))+(-0.02113506688\br
++\tab{4}&615768D0*Z(3))+0.06681263884671322D0*Z(2))+0.06069778964023718D0*Z(\br
++\tab{4}&1)\br
++\tab{5}W(11)=0.04254083491825025D0*Z(16))+0.2311852964327382D0*Z(15))+(-0.0\br
++\tab{4}&7037620467004427D0*Z(14))+(-0.1846882042225383D0*Z(13))+(-0.315830\br
++\tab{4}&9975786731D0*Z(12))+0.952576555482747D0*Z(11))+(-0.328001910890377D\br
++\tab{4}&0*Z(10))+(-0.1589391311991561D0*Z(9))+(-0.07375317649315155D0*Z(8)\br
++\tab{4}&)+0.2229538339673001D0*Z(7))+(-0.03526886317505474D0*Z(6))+(-0.0493\br
++\tab{4}&1323319055762D0*Z(5))+(-0.04319641116207706D0*Z(4))+0.048260820054\br
++\tab{4}&65965D0*Z(3))+0.01328585741341559D0*Z(2))+0.04015147277405744D0*Z(1)\br
++\tab{5}W(12)=0.3198209177068516D0*Z(16))+(-0.03423495461011043D0*Z(15))+(-\br
++\tab{4}&0.1417185427288274D0*Z(14))+(-0.3852396953763045D0*Z(13))+0.959162\br
++\tab{4}&3347824002D0*Z(12))+(-0.315042193418466D0*Z(11))+(-0.14739952092889\br
++\tab{4}&45D0*Z(10))+(-0.08249492560213524D0*Z(9))+0.2171103102175198D0*Z(8\br
++\tab{4}&)+(-0.03769663291725935D0*Z(7))+(-0.05034242196614937D0*Z(6))+(-0.\br
++\tab{4}&01445079632086177D0*Z(5))+0.02947046460707379D0*Z(4))+(-0.002512226\br
++\tab{4}&501941856D0*Z(3))+(-0.001822737697581869D0*Z(2))+0.045563697677763\br
++\tab{4}&75D0*Z(1)\br
++\tab{5}W(13)=0.09089205517109111D0*Z(16))+(-0.09033531980693314D0*Z(15))+(\br
++\tab{4}&-0.3241503380268184D0*Z(14))+0.8600427128450191D0*Z(13))+(-0.305169\br
++\tab{4}&742604165D0*Z(12))+(-0.1280829963720053D0*Z(11))+(-0.0663087451453\br
++\tab{4}&5952D0*Z(10))+0.2012230497530726D0*Z(9))+(-0.04353074206076491D0*Z(\br
++\tab{4}&8))+(-0.05051817793156355D0*Z(7))+(-0.014224695935687D0*Z(6))+0.05\br

```

```

++\tab{4}&468897337339577D0*Z(5)+(-0.01965809746040371D0*Z(4))+(-0.016234277\br
++\tab{4}&35779699D0*Z(3))+0.005239165960779299D0*Z(2)+0.05141563713660119D0\br
++\tab{4}&*Z(1)\br
++\tab{5}W(14)=(-0.02986582812574917D0*Z(16))+(-0.2995429545781457D0*Z(15))\br
++\tab{4}&+0.8892996132269974D0*Z(14))+(-0.3523683853026259D0*Z(13))+(-0.1236\br
++\tab{4}&679206156403D0*Z(12))+(-0.05760560341383113D0*Z(11))+0.20910979278\br
++\tab{4}&87612D0*Z(10))+(-0.04901428822579872D0*Z(9))+(-0.05483186562035512D\br
++\tab{4}&0*Z(8))+(-0.01632133125029967D0*Z(7))+0.05375944956767728D0*Z(6)+0\br
++\tab{4}&.002033305231024948D0*Z(5))+(-0.03032392238968179D0*Z(4))+(-0.00660\br
++\tab{4}&7305534689702D0*Z(3))+0.02021603150122265D0*Z(2)+0.033711981971903\br
++\tab{4}&02D0*Z(1)\br
++\tab{5}W(15)=(-0.2419652703415429D0*Z(16))+0.9128222941872173D0*Z(15))+(-0\br
++\tab{4}&.3244016605667343D0*Z(14))+(-0.1688977368984641D0*Z(13))+(-0.05325\br
++\tab{4}&555586632358D0*Z(12))+0.2176561076571465D0*Z(11))+(-0.0415311995556\br
++\tab{4}&9051D0*Z(10))+(-0.06095390688679697D0*Z(9))+(-0.01981532388243379D\br
++\tab{4}&0*Z(8))+0.05258889186338282D0*Z(7)+0.00157466157362272D0*Z(6))+(-0.\br
++\tab{4}&0135713672105995D0*Z(5))+(-0.01764072463999744D0*Z(4))+0.010940122\br
++\tab{4}&10519586D0*Z(3))+0.008812321197398072D0*Z(2)+0.0227345011107737D0*Z\br
++\tab{4}&(1)\br
++\tab{5}W(16)=1.019463911841327D0*Z(16))+(-0.2803531651057233D0*Z(15))+(-0.\br
++\tab{4}&1165300508238904D0*Z(14))+(-0.1385343580686922D0*Z(13))+0.22647669\br
++\tab{4}&47290192D0*Z(12))+(-0.02434652144032987D0*Z(11))+(-0.04723268012114\br
++\tab{4}&625D0*Z(10))+(-0.03586220812223305D0*Z(9))+0.04932374658377151D0*Z\br
++\tab{4}&(8)+0.00372306473653087D0*Z(7))+(-0.01219194009813166D0*Z(6))+(-0.0\br
++\tab{4}&07005540882865317D0*Z(5))+0.002957434991769087D0*Z(4)+0.0021069739\br
++\tab{4}&00813502D0*Z(3))+0.001747395874954051D0*Z(2)+0.01707454969713436D0*\br
++\tab{4}&Z(1)\br
++\tab{5}RETURN\br
++\tab{5}END\br

```

Asp28(name): Exports == Implementation where

name : Symbol

```

FST    ==> FortranScalarType
FT     ==> FortranType
SYMTAB ==> SymbolTable
FC     ==> FortranCode
PI     ==> PositiveInteger
RSFC   ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
EXPR   ==> Expression
MFLOAT ==> MachineFloat
VEC    ==> Vector
UFST   ==> Union(fst:FST,void:"void")
MAT    ==> Matrix

```

Exports == FortranMatrixCategory

Implementation == add

```

real : UFST := ["real":FST]$UFST
syms : SYMTAB := empty()
declare!(IFLAG,fortranInteger(),syms)$SYMTAB
declare!(N,fortranInteger(),syms)$SYMTAB
declare!(LRWORK,fortranInteger(),syms)$SYMTAB
declare!(LIWORK,fortranInteger(),syms)$SYMTAB
xType : FT := construct(real,[N],false)$FT
declare!(Z,xType,syms)$SYMTAB
declare!(W,xType,syms)$SYMTAB
rType : FT := construct(real,[LRWORK],false)$FT
declare!(RWORK,rType,syms)$SYMTAB
iType : FT := construct(real,[LIWORK],false)$FT
declare!(IWORK,rType,syms)$SYMTAB
Rep := FortranProgram(name,["void"]$UFST,
                        [IFLAG,N,Z,W,RWORK,LRWORK,IWORK,LIWORK],syms)

-- To help the poor old compiler!
localCoerce(u:Symbol):EXPR(MFLOAT) == coerce(u)$EXPR(MFLOAT)

coerce (u:MAT MFLOAT):$ ==
  Zs: Symbol := Z
  code : List FC
  r: List EXPR MFLOAT
  r := ["+"/[u(j,i)*localCoerce(elt(Zs,[i::OutputForm]))$Symbol)_
        for i in 1..ncols(u)$MAT(MFLOAT)::PI]_
        for j in 1..nrows(u)$MAT(MFLOAT)::PI]
  code := [assign(W@Symbol,vector(r)$VEC(EXPR MFLOAT)),returns()$List(FC)
  code:::$

coerce(c:FortranCode):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

coerce(c:List FortranCode):$ == coerce(c)$Rep

coerce(u:$):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

```

— ASP28.dotabb —

"ASP28" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP28"]
 "ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]

"ASP28" -> "ALIST"

domain ASP29 Asp29

— Asp29.input —

```
)set break resume
)sys rm -f Asp29.output
)spool Asp29.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp29
--R
--R Asp29(name: Symbol) is a domain constructor
--R Abbreviation for Asp29 is ASP29
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP29
--R
--R----- Operations -----
--R coerce : % -> OutputForm          outputAsFortran : () -> Void
--R outputAsFortran : % -> Void
--R
--E 1

)spool
)lisp (bye)
```

— Asp29.help —

=====

Asp29 examples

=====

Asp29 produces Fortran for Type 29 ASPs, needed for NAG routine f02fjf,
for example:

```
SUBROUTINE MONIT(ISTATE,NEXTIT,NEVALS,NEVECS,K,F,D)
DOUBLE PRECISION D(K),F(K)
```

```

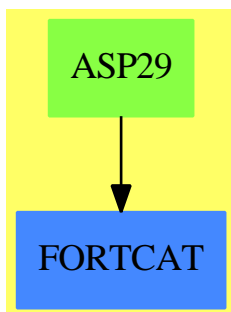
INTEGER K,NEXTIT,NEVALS,NVECS,ISTATE
CALL F02FJZ(ISTATE,NEXTIT,NEVALS,NEVECS,K,F,D)
RETURN
END

```

See Also:

o)show Asp29

Asp29 (ASP29)



Exports:

coerce outputAsFortran

— domain ASP29 Asp29 —

```

)abbrev domain ASP29 Asp29
++ Author: Mike Dewar and Godfrey Nolan
++ Date Created: Nov 1993
++ Date Last Updated: 18 March 1994
++ Description:
++\spadtype{Asp29} produces Fortran for Type 29 ASPs, needed for NAG routine
++f02fjf, for example:
++
++\tab{5}SUBROUTINE MONIT(ISTATE,NEXTIT,NEVALS,NEVECS,K,F,D)\br
++\tab{5}DOUBLE PRECISION D(K),F(K)\br
++\tab{5}INTEGER K,NEXTIT,NEVALS,NVECS,ISTATE\br
++\tab{5}CALL F02FJZ(ISTATE,NEXTIT,NEVALS,NEVECS,K,F,D)\br
++\tab{5}RETURN\br
++\tab{5}END\br

```

```

Asp29(name): Exports == Implementation where
  name : Symbol

```

```

FST ==> FortranScalarType
FT  ==> FortranType
FSTU ==> Union(fst:FST,void:"void")
SYMTAB ==> SymbolTable
FC  ==> FortranCode
PI  ==> PositiveInteger
EXF ==> Expression Float
EXI ==> Expression Integer
VEF ==> Vector Expression Float
VEI ==> Vector Expression Integer
MEI ==> Matrix Expression Integer
MEF ==> Matrix Expression Float
UEXPR ==> Union(I: Expression Integer,F: Expression Float,_
                CF: Expression Complex Float)
RSFC ==> Record(localSymbols:SymbolTable,code:List(FortranCode))

Exports == FortranProgramCategory with
  outputAsFortran:() -> Void
  ++outputAsFortran() generates the default code for \spadtype{ASP29}.

Implementation == add

import FST
import FT
import FC
import SYMTAB

real : FSTU := ["real":FST]$FSTU
integer : FSTU := ["integer":FST]$FSTU
syms : SYMTAB := empty()
declare!(ISTATE,fortranInteger(),syms)
declare!(NEXTIT,fortranInteger(),syms)
declare!(NEVALS,fortranInteger(),syms)
declare!(NEVECS,fortranInteger(),syms)
declare!(K,fortranInteger(),syms)
kType : FT := construct(real,[K],false)$FT
declare!(F,kType,syms)
declare!(D,kType,syms)
Rep := FortranProgram(name,["void"]$FSTU,
                      [ISTATE,NEXTIT,NEVALS,NEVECS,K,F,D],syms)

outputAsFortran():Void ==
  callOne := call("F02FJZ(ISTATE,NEXTIT,NEVALS,NEVECS,K,F,D)")
  code : List FC := [callOne,returns()$List(FC)
  outputAsFortran(coerce(code)$Rep)$Rep

```

— ASP29.dotabb —

```
"ASP29" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP29"]
"FORTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FORTCAT"]
"ASP29" -> "FORTCAT"
```

domain ASP30 Asp30

— Asp30.input —

```
)set break resume
)sys rm -f Asp30.output
)spool Asp30.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp30
--R
--R Asp30(name: Symbol) is a domain constructor
--R Abbreviation for Asp30 is ASP30
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP30
--R
--R----- Operations -----
--R coerce : FortranCode -> %           coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm           outputAsFortran : % -> Void
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R coerce : Matrix(MachineFloat) -> %
--R
--E 1

)spool
)lisp (bye)
```

— Asp30.help —

=====

Asp30 examples

=====

Asp30 produces Fortran for Type 30 ASPs, needed for NAG routine f04qaf,
for example:

```

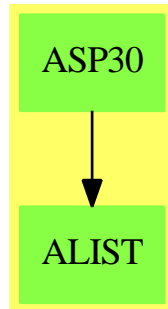
SUBROUTINE APROD(MODE,M,N,X,Y,RWORK,LRWORK,IWORK,LIWORK)
DOUBLE PRECISION X(N),Y(M),RWORK(LRWORK)
INTEGER M,N,LIWORK,IFAIL,LRWORK,IWORK(LIWORK),MODE
DOUBLE PRECISION A(5,5)
EXTERNAL F06PAF
A(1,1)=1.0D0
A(1,2)=0.0D0
A(1,3)=0.0D0
A(1,4)=-1.0D0
A(1,5)=0.0D0
A(2,1)=0.0D0
A(2,2)=1.0D0
A(2,3)=0.0D0
A(2,4)=0.0D0
A(2,5)=-1.0D0
A(3,1)=0.0D0
A(3,2)=0.0D0
A(3,3)=1.0D0
A(3,4)=-1.0D0
A(3,5)=0.0D0
A(4,1)=-1.0D0
A(4,2)=0.0D0
A(4,3)=-1.0D0
A(4,4)=4.0D0
A(4,5)=-1.0D0
A(5,1)=0.0D0
A(5,2)=-1.0D0
A(5,3)=0.0D0
A(5,4)=-1.0D0
A(5,5)=4.0D0
IF(MODE.EQ.1) THEN
CALL F06PAF('N',M,N,1.0D0,A,M,X,1,1.0D0,Y,1)
ELSEIF(MODE.EQ.2) THEN
CALL F06PAF('T',M,N,1.0D0,A,M,Y,1,1.0D0,X,1)
ENDIF
RETURN
END

```

See Also:

o)show Asp30

Asp30 (ASP30)



Exports:

coerce outputAsFortran

— domain ASP30 Asp30 —

```

)abbrev domain ASP30 Asp30
++ Author: Mike Dewar and Godfrey Nolan
++ Date Created: Nov 1993
++ Date Last Updated: 6 October 1994
++ Description:
++\spadtype{Asp30} produces Fortran for Type 30 ASPs, needed for NAG routine
++f04qaf, for example:
++
++\tab{5}SUBROUTINE APROD(MODE,M,N,X,Y,RWORK,LRWORK,IWORK,LIWORK)\br
++\tab{5}DOUBLE PRECISION X(N),Y(M),RWORK(LRWORK)\br
++\tab{5}INTEGER M,N,LIWORK,IFAIL,LRWORK,IWORK(LIWORK),MODE\br
++\tab{5}DOUBLE PRECISION A(5,5)\br
++\tab{5}EXTERNAL F06PAF\br
++\tab{5}A(1,1)=1.0D0\br
++\tab{5}A(1,2)=0.0D0\br
++\tab{5}A(1,3)=0.0D0\br
++\tab{5}A(1,4)=-1.0D0\br
++\tab{5}A(1,5)=0.0D0\br
++\tab{5}A(2,1)=0.0D0\br
++\tab{5}A(2,2)=1.0D0\br
++\tab{5}A(2,3)=0.0D0\br
++\tab{5}A(2,4)=0.0D0\br
++\tab{5}A(2,5)=-1.0D0\br
++\tab{5}A(3,1)=0.0D0\br
++\tab{5}A(3,2)=0.0D0\br
++\tab{5}A(3,3)=1.0D0\br
++\tab{5}A(3,4)=-1.0D0\br
++\tab{5}A(3,5)=0.0D0\br
++\tab{5}A(4,1)=-1.0D0\br
++\tab{5}A(4,2)=0.0D0\br

```

```

++\tab{5}A(4,3)=-1.0D0\br
++\tab{5}A(4,4)=4.0D0\br
++\tab{5}A(4,5)=-1.0D0\br
++\tab{5}A(5,1)=0.0D0\br
++\tab{5}A(5,2)=-1.0D0\br
++\tab{5}A(5,3)=0.0D0\br
++\tab{5}A(5,4)=-1.0D0\br
++\tab{5}A(5,5)=4.0D0\br
++\tab{5}IF(MODE.EQ.1)THEN\br
++\tab{7}CALL F06PAF('N',M,N,1.0D0,A,M,X,1,1.0D0,Y,1)\br
++\tab{5}ELSEIF(MODE.EQ.2)THEN\br
++\tab{7}CALL F06PAF('T',M,N,1.0D0,A,M,Y,1,1.0D0,X,1)\br
++\tab{5}ENDIF\br
++\tab{5}RETURN\br
++\tab{5}END

Asp30(name): Exports == Implementation where
  name : Symbol

  FST    ==> FortranScalarType
  FT     ==> FortranType
  SYMTAB ==> SymbolTable
  FC     ==> FortranCode
  PI     ==> PositiveInteger
  RSFC   ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
  UFST   ==> Union(fst:FST,void:"void")
  MAT    ==> Matrix
  MFLOAT ==> MachineFloat
  EXI    ==> Expression Integer
  UEXPR  ==> Union(I:Expression Integer,F:Expression Float,_
                  CF:Expression Complex Float,switch:Switch)
  S      ==> Symbol

Exports == FortranMatrixCategory

Implementation == add

import FC
import FT
import Switch

real : UFST := ["real"::FST]$UFST
integer : UFST := ["integer"::FST]$UFST
syms : SYMTAB := empty()$SYMTAB
declare!(MODE,fortranInteger()$FT,syms)$SYMTAB
declare!(M,fortranInteger()$FT,syms)$SYMTAB
declare!(N,fortranInteger()$FT,syms)$SYMTAB
declare!(LRWORK,fortranInteger()$FT,syms)$SYMTAB
declare!(LIWORK,fortranInteger()$FT,syms)$SYMTAB
xType : FT := construct(real,[N],false)$FT

```

```

declare!(X,xType,syms)$SYMTAB
yType : FT := construct(real,[M],false)$FT
declare!(Y,yType,syms)$SYMTAB
rType : FT := construct(real,[LRWORK],false)$FT
declare!(RWORK,rType,syms)$SYMTAB
iType : FT := construct(integer,[LIWORK],false)$FT
declare!(IWORK,iType,syms)$SYMTAB
declare!(IFAIL,fortranInteger())$FT,syms)$SYMTAB
Rep := FortranProgram(name,["void"]$UFST,
                        [MODE,M,N,X,Y,RWORK,LRWORK,IWORK,LIWORK],syms)

coerce(a:MAT MFLOAT):$ ==
  locals : SYMTAB := empty()
  numRows := nrows(a) :: Polynomial Integer
  numCols := ncols(a) :: Polynomial Integer
  declare!(A,[real,[numRows,numCols],false]$FT,locals)
  declare!(F06PAF@S,construct(["void"]$UFST,[],@List(S),true)$FT,locals)
  ptA:UEXPR := [("MODE":S)::EXI]
  ptB:UEXPR := [1::EXI]
  ptC:UEXPR := [2::EXI]
  sw1 : Switch := EQ(ptA,ptB)$Switch
  sw2 : Switch := EQ(ptA,ptC)$Switch
  callOne := call("F06PAF('N',M,N,1.ODO,A,M,X,1,1.ODO,Y,1)")
  callTwo := call("F06PAF('T',M,N,1.ODO,A,M,Y,1,1.ODO,X,1)")
  c : FC := cond(sw1,callOne,cond(sw2,callTwo))
  code : List FC := [assign(A,a),c,returns()]
  ([locals,code]$RSFC)::$

coerce(c:List FortranCode):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

coerce(c:FortranCode):$ == coerce(c)$Rep

coerce(u:$):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

```

— ASP30.dotabb —

```

"ASP30" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP30"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ASP30" -> "ALIST"

```

domain ASP31 Asp31

— Asp31.input —

```

)set break resume
)sys rm -f Asp31.output
)spool Asp31.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp31
--R
--R Asp31(name: Symbol) is a domain constructor
--R Abbreviation for Asp31 is ASP31
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP31
--R
--R----- Operations -----
--R coerce : FortranCode -> %               coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm               outputAsFortran : % -> Void
--R coerce : Vector(FortranExpression([construct,QUOTEX],[construct,QUOTEY],MachineFloat)) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Vector(Fraction(Polynomial(Integer))) -> %
--R retract : Vector(Fraction(Polynomial(Float))) -> %
--R retract : Vector(Polynomial(Integer)) -> %
--R retract : Vector(Polynomial(Float)) -> %
--R retract : Vector(Expression(Integer)) -> %
--R retract : Vector(Expression(Float)) -> %
--R retractIfCan : Vector(Fraction(Polynomial(Integer))) -> Union(%,"failed")
--R retractIfCan : Vector(Fraction(Polynomial(Float))) -> Union(%,"failed")
--R retractIfCan : Vector(Polynomial(Integer)) -> Union(%,"failed")
--R retractIfCan : Vector(Polynomial(Float)) -> Union(%,"failed")
--R retractIfCan : Vector(Expression(Integer)) -> Union(%,"failed")
--R retractIfCan : Vector(Expression(Float)) -> Union(%,"failed")
--R
--E 1

)spool
)lisp (bye)

```

— Asp31.help —

===== Asp31 examples =====

Asp31 produces Fortran for Type 31 ASPs, needed for NAG routine d02ejf, for example:

```

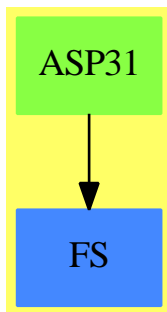
SUBROUTINE PEDERV(X,Y,PW)
DOUBLE PRECISION X,Y(*)
DOUBLE PRECISION PW(3,3)
PW(1,1)=-0.03999999999999999D0
PW(1,2)=10000.0D0*Y(3)
PW(1,3)=10000.0D0*Y(2)
PW(2,1)=0.03999999999999999D0
PW(2,2)=(-10000.0D0*Y(3))+(-60000000.0D0*Y(2))
PW(2,3)=-10000.0D0*Y(2)
PW(3,1)=0.0D0
PW(3,2)=60000000.0D0*Y(2)
PW(3,3)=0.0D0
RETURN
END

```

See Also:

o)show Asp31

Asp31 (ASP31)

**Exports:**

coerce outputAsFortran retract retractIfCan

— domain ASP31 Asp31 —

```

)abbrev domain ASP31 Asp31
++ Author: Mike Dewar, Grant Keady and Godfrey Nolan
++ Date Created: Mar 1993
++ Date Last Updated: 6 October 1994
++ Description:
++\spadtype{Asp31} produces Fortran for Type 31 ASPs, needed for NAG routine
++d02ejf, for example:
++
++\tab{5}SUBROUTINE PEDERV(X,Y,PW)\br
++\tab{5}DOUBLE PRECISION X,Y(*)\br
++\tab{5}DOUBLE PRECISION PW(3,3)\br
++\tab{5}PW(1,1)=-0.03999999999999999D0\br
++\tab{5}PW(1,2)=10000.0D0*Y(3)\br
++\tab{5}PW(1,3)=10000.0D0*Y(2)\br
++\tab{5}PW(2,1)=0.03999999999999999D0\br
++\tab{5}PW(2,2)=(-10000.0D0*Y(3))+(-60000000.0D0*Y(2))\br
++\tab{5}PW(2,3)=-10000.0D0*Y(2)\br
++\tab{5}PW(3,1)=0.0D0\br
++\tab{5}PW(3,2)=60000000.0D0*Y(2)\br
++\tab{5}PW(3,3)=0.0D0\br
++\tab{5}RETURN\br
++\tab{5}END

```

```

Asp31(name): Exports == Implementation where
  name : Symbol

```

```

O      ==> OutputForm
FST    ==> FortranScalarType
UFST   ==> Union(fst:FST,void:"void")
MFLOAT ==> MachineFloat
FEXPR  ==> FortranExpression(['X'],['Y'],MFLOAT)
FT     ==> FortranType
FC     ==> FortranCode
SYMTAB ==> SymbolTable
RSFC   ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
FRAC   ==> Fraction
POLY   ==> Polynomial
EXPR   ==> Expression
INT    ==> Integer
FLOAT  ==> Float
VEC    ==> Vector
MAT    ==> Matrix
VF2    ==> VectorFunctions2
MF2    ==> MatrixCategoryFunctions2(FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR,
    EXPR MFLOAT,VEC EXPR MFLOAT,VEC EXPR MFLOAT,MAT EXPR MFLOAT)

```

```

Exports ==> FortranVectorFunctionCategory with
  coerce : VEC FEXPR -> $
    ++coerce(f) takes objects from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns them into an ASP.

Implementation ==> add

real : UFST := ["real"::FST]$UFST
syms : SYMTAB := empty()
declare!(X,fortranReal(),syms)$SYMTAB
yType : FT := construct(real,["* "::Symbol],false)$FT
declare!(Y,yType,syms)$SYMTAB
Rep := FortranProgram(name,["void"$UFST,[X,Y,PW],syms)

-- To help the poor old compiler!
fexpr2expr(u:FEXPR):EXPR MFLOAT == coerce(u)$FEXPR

localAssign(s:Symbol,j:MAT FEXPR):FC ==
  j' : MAT EXPR MFLOAT := map(fexpr2expr,j)$MF2
  assign(s,j')$FC

makeXList(n:Integer):List(Symbol) ==
  j:Integer
  y:Symbol := Y::Symbol
  p:List(Symbol) := []
  for j in 1 .. n repeat p:= cons(subscript(y,[j::OutputForm])$Symbol,p)
  p:= reverse(p)

coerce(u:VEC FEXPR):$ ==
  dimension := #u::Polynomial Integer
  locals : SYMTAB := empty()
  declare!(PW,[real,[dimension,dimension],false]$FT,locals)$SYMTAB
  n:Integer := maxIndex(u)$VEC(FEXPR)
  p:List(Symbol) := makeXList(n)
  jac: MAT FEXPR := jacobian(u,p)$MultiVariableCalculusFunctions(_
    Symbol,FEXPR ,VEC FEXPR,List(Symbol))
  code : List FC := [localAssign(PW,jac),returns()$FC]$List(FC)
  ([locals,code]$RSFC)::$

retract(u:VEC FRAC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

```

```

retract(u:VEC FRAC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC EXPR INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR INT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC EXPR FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

coerce(c:List FC):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

coerce(c:FC):$ == coerce(c)$Rep

```



```

coerce(u:$):0 == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

```

— ASP31.dotabb —

```

"ASP31" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP31"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ASP31" -> "FS"

```

domain ASP33 Asp33

— Asp33.input —

```

)set break resume
)sys rm -f Asp33.output
)spool Asp33.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp33
--R
--R Asp33(name: Symbol) is a domain constructor
--R Abbreviation for Asp33 is ASP33
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP33
--R
--R----- Operations -----
--R coerce : % -> OutputForm          outputAsFortran : () -> Void
--R outputAsFortran : % -> Void
--R
--E 1

)spool
)lisp (bye)

```

— Asp33.help —

=====

Asp33 examples

=====

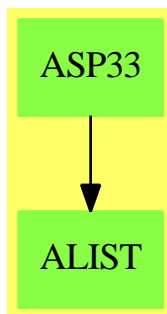
Asp33 produces Fortran for Type 33 ASPs, needed for NAG routine d02kef.
The code is a dummy ASP:

```
SUBROUTINE REPORT(X,V,JINT)
  DOUBLE PRECISION V(3),X
  INTEGER JINT
  RETURN
END
```

See Also:

o)show Asp33

Asp33 (ASP33)



Exports:

coerce outputAsFortran

— domain ASP33 Asp33 —

```
)abbrev domain ASP33 Asp33
++ Author: Mike Dewar and Godfrey Nolan
++ Date Created: Nov 1993
++ Date Last Updated: 30 March 1994
++ Description:
```

```

++\spadtype{Asp33} produces Fortran for Type 33 ASPs, needed for NAG routine
++d02kef. The code is a dummy ASP:
++
++\tab{5}SUBROUTINE REPORT(X,V,JINT)\br
++\tab{5}DOUBLE PRECISION V(3),X\br
++\tab{5}INTEGER JINT\br
++\tab{5}RETURN\br
++\tab{5}END

```

```

Asp33(name): Exports == Implementation where
  name : Symbol

```

```

FST    ==> FortranScalarType
UFST   ==> Union(fst:FST,void:"void")
FT     ==> FortranType
SYMTAB ==> SymbolTable
FC     ==> FortranCode
RSFC   ==> Record(localSymbols:SymbolTable,code:List(FortranCode))

```

```

Exports ==> FortranProgramCategory with
  outputAsFortran(): -> Void
  ++outputAsFortran() generates the default code for \spadtype{ASP33}.

```

```

Implementation ==> add

```

```

real : UFST := ["real":FST]$UFST
syms : SYMTAB := empty()
declare!(JINT,fortranInteger(),syms)$SYMTAB
declare!(X,fortranReal(),syms)$SYMTAB
vType : FT := construct(real,["3":Symbol],false)$FT
declare!(V,vType,syms)$SYMTAB
Rep := FortranProgram(name,["void"]$UFST,[X,V,JINT],syms)

```

```

outputAsFortran():Void ==
  outputAsFortran( (returns())$FortranCode)::Rep )$Rep

```

```

outputAsFortran(u):Void == outputAsFortran(u)$Rep

```

```

coerce(u:$):OutputForm == coerce(u)$Rep

```

— ASP33.dotabb —

```

"ASP33" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP33"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ASP33" -> "ALIST"

```

domain ASP34 Asp34

— Asp34.input —

```

)set break resume
)sys rm -f Asp34.output
)spool Asp34.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp34
--R
--R Asp34(name: Symbol) is a domain constructor
--R Abbreviation for Asp34 is ASP34
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP34
--R
--R----- Operations -----
--R coerce : FortranCode -> %               coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm               outputAsFortran : % -> Void
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R coerce : Matrix(MachineFloat) -> %
--R
--E 1

)spool
)lisp (bye)

```

— Asp34.help —

```

=====
Asp34 examples
=====

```

Asp34 produces Fortran for Type 34 ASPs, needed for NAG routine f04mbf,
for example:

```
SUBROUTINE MSOLVE(IFLAG,N,X,Y,RWORK,LRWORK,IWORK,LIWORK)
```

```

DOUBLE PRECISION RWORK(LRWORK),X(N),Y(N)
INTEGER I,J,N,LIWORK,IFLAG,LRWORK,IWORK(LIWORK)
DOUBLE PRECISION W1(3),W2(3),MS(3,3)
IFLAG=-1
MS(1,1)=2.0D0
MS(1,2)=1.0D0
MS(1,3)=0.0D0
MS(2,1)=1.0D0
MS(2,2)=2.0D0
MS(2,3)=1.0D0
MS(3,1)=0.0D0
MS(3,2)=1.0D0
MS(3,3)=2.0D0
CALL F04ASF(MS,N,X,N,Y,W1,W2,IFLAG)
IFLAG=-IFLAG
RETURN
END

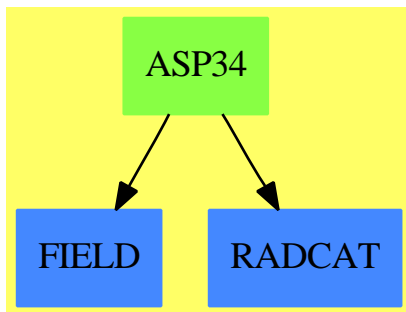
```

See Also:

o)show Asp34

—————→

Asp34 (ASP34)



Exports:

coerce outputAsFortran

— domain ASP34 Asp34 —

)abbrev domain ASP34 Asp34

++ Author: Mike Dewar and Godfrey Nolan and Themos Tsikas

++ Date Created: Nov 1993

++ Date Last Updated: 6 October 1994

```

++ Description:
++\spadtype{Asp34} produces Fortran for Type 34 ASPs, needed for NAG routine
++f04mbf, for example:
++
++\tab{5}SUBROUTINE MSOLVE(IFLAG,N,X,Y,RWORK,LRWORK,IWORK,LIWORK)\br
++\tab{5}DOUBLE PRECISION RWORK(LRWORK),X(N),Y(N)\br
++\tab{5}INTEGER I,J,N,LIWORK,IFLAG,LRWORK,IWORK(LIWORK)\br
++\tab{5}DOUBLE PRECISION W1(3),W2(3),MS(3,3)\br
++\tab{5}IFLAG=-1\br
++\tab{5}MS(1,1)=2.0D0\br
++\tab{5}MS(1,2)=1.0D0\br
++\tab{5}MS(1,3)=0.0D0\br
++\tab{5}MS(2,1)=1.0D0\br
++\tab{5}MS(2,2)=2.0D0\br
++\tab{5}MS(2,3)=1.0D0\br
++\tab{5}MS(3,1)=0.0D0\br
++\tab{5}MS(3,2)=1.0D0\br
++\tab{5}MS(3,3)=2.0D0\br
++\tab{5}CALL F04ASF(MS,N,X,N,Y,W1,W2,IFLAG)\br
++\tab{5}IFLAG=-IFLAG\br
++\tab{5}RETURN\br
++\tab{5}END

```

```

Asp34(name): Exports == Implementation where
name : Symbol

```

```

FST    ==> FortranScalarType
FT     ==> FortranType
UFST   ==> Union(fst:FST,void:"void")
SYMTAB ==> SymbolTable
FC     ==> FortranCode
PI     ==> PositiveInteger
EXI    ==> Expression Integer
RSFC   ==> Record(localSymbols:SymbolTable,code:List(FortranCode))

```

```
Exports == FortranMatrixCategory
```

```
Implementation == add
```

```

real : UFST := ["real"::FST]$UFST
integer : UFST := ["integer"::FST]$UFST
syms : SYMTAB := empty()$SYMTAB
declare!(IFLAG,fortranInteger(),syms)$SYMTAB
declare!(N,fortranInteger(),syms)$SYMTAB
xType : FT := construct(real,[N],false)$FT
declare!(X,xType,syms)$SYMTAB
declare!(Y,xType,syms)$SYMTAB
declare!(LRWORK,fortranInteger(),syms)$SYMTAB
declare!(LIWORK,fortranInteger(),syms)$SYMTAB
rType : FT := construct(real,[LRWORK],false)$FT

```

```

declare! (RWORK,rType,syms)$SYMTAB
iType : FT := construct(integer,[LIWORK],false)$FT
declare! (IWORK,iType,syms)$SYMTAB
Rep := FortranProgram(name,["void"]$UFST,
                        [IFLAG,N,X,Y,RWORK,LRWORK,IWORK,LIWORK],syms)

-- To help the poor old compiler
localAssign(s:Symbol,u:EXI):FC == assign(s,u)$FC

coerce(u:Matrix MachineFloat):$ ==
  dimension := nrows(u)::Polynomial Integer
  locals : SYMTAB := empty()$SYMTAB
  declare! (I,fortranInteger(),syms)$SYMTAB
  declare! (J,fortranInteger(),syms)$SYMTAB
  declare! (W1,[real,[dimension],false]$FT,locals)$SYMTAB
  declare! (W2,[real,[dimension],false]$FT,locals)$SYMTAB
  declare! (MS,[real,[dimension,dimension],false]$FT,locals)$SYMTAB
  assign1 : FC := localAssign(IFLAG@Symbol,(-1)@EXI)
  call : FC := call("F04ASF(MS,N,X,N,Y,W1,W2,IFLAG)")$FC
  assign2 : FC := localAssign(IFLAG::Symbol,-(IFLAG@Symbol::EXI))
  assign3 : FC := assign(MS,u)$FC
  code : List FC := [assign1,assign3,call,assign2,returns()$List(FC)
                    ([locals,code]$RSFC)::]$

coerce(c:List FortranCode):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

coerce(c:FortranCode):$ == coerce(c)$Rep

coerce(u:$):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

```

— ASP34.dotabb —

```

"ASP34" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP34"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"ASP34" -> "FIELD"
"ASP34" -> "RADCAT"

```

domain ASP35 Asp35**— Asp35.input —**

```

)set break resume
)sys rm -f Asp35.output
)spool Asp35.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp35
--R
--R Asp35(name: Symbol) is a domain constructor
--R Abbreviation for Asp35 is ASP35
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP35
--R
--R----- Operations -----
--R coerce : FortranCode -> %               coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm               outputAsFortran : % -> Void
--R coerce : Vector(FortranExpression([construct],[construct,QUOTEX],MachineFloat)) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Vector(Fraction(Polynomial(Integer))) -> %
--R retract : Vector(Fraction(Polynomial(Float))) -> %
--R retract : Vector(Polynomial(Integer)) -> %
--R retract : Vector(Polynomial(Float)) -> %
--R retract : Vector(Expression(Integer)) -> %
--R retract : Vector(Expression(Float)) -> %
--R retractIfCan : Vector(Fraction(Polynomial(Integer))) -> Union(%, "failed")
--R retractIfCan : Vector(Fraction(Polynomial(Float))) -> Union(%, "failed")
--R retractIfCan : Vector(Polynomial(Integer)) -> Union(%, "failed")
--R retractIfCan : Vector(Polynomial(Float)) -> Union(%, "failed")
--R retractIfCan : Vector(Expression(Integer)) -> Union(%, "failed")
--R retractIfCan : Vector(Expression(Float)) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)

```

— Asp35.help —

```

=====
Asp35 examples

```


=====

Asp35 produces Fortran for Type 35 ASPs, needed for NAG routines c05pbf, c05pcf, for example:

```

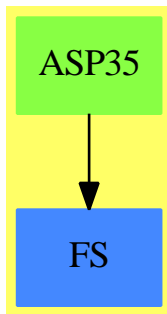
SUBROUTINE FCN(N,X,FVEC,FJAC,LDFJAC,IFLAG)
DOUBLE PRECISION X(N),FVEC(N),FJAC(LDFJAC,N)
INTEGER LDFJAC,N,IFLAG
IF(IFLAG.EQ.1)THEN
FVEC(1)=(-1.0D0*X(2))+X(1)
FVEC(2)=(-1.0D0*X(3))+2.0D0*X(2)
FVEC(3)=3.0D0*X(3)
ELSEIF(IFLAG.EQ.2)THEN
FJAC(1,1)=1.0D0
FJAC(1,2)=-1.0D0
FJAC(1,3)=0.0D0
FJAC(2,1)=0.0D0
FJAC(2,2)=2.0D0
FJAC(2,3)=-1.0D0
FJAC(3,1)=0.0D0
FJAC(3,2)=0.0D0
FJAC(3,3)=3.0D0
ENDIF
END

```

See Also:

o)show Asp35

Asp35 (ASP35)



Exports:

coerce outputAsFortran retract retractIfCan

— domain ASP35 Asp35 —

```

)abbrev domain ASP35 Asp35
++ Author: Mike Dewar, Godfrey Nolan, Grant Keady
++ Date Created: Mar 1993
++ Date Last Updated: 6 October 1994
++ Description:
++\spadtype{Asp35} produces Fortran for Type 35 ASPs, needed for NAG routines
++c05pbf, c05pcf, for example:
++
++\tab{5}SUBROUTINE FCN(N,X,FVEC,FJAC,LDFJAC,IFLAG)\br
++\tab{5}DOUBLE PRECISION X(N),FVEC(N),FJAC(LDFJAC,N)\br
++\tab{5}INTEGER LDFJAC,N,IFLAG\br
++\tab{5}IF(IFLAG.EQ.1)THEN\br
++\tab{7}FVEC(1)=(-1.0D0*X(2))+X(1)\br
++\tab{7}FVEC(2)=(-1.0D0*X(3))+2.0D0*X(2)\br
++\tab{7}FVEC(3)=3.0D0*X(3)\br
++\tab{5}ELSEIF(IFLAG.EQ.2)THEN\br
++\tab{7}FJAC(1,1)=1.0D0\br
++\tab{7}FJAC(1,2)=-1.0D0\br
++\tab{7}FJAC(1,3)=0.0D0\br
++\tab{7}FJAC(2,1)=0.0D0\br
++\tab{7}FJAC(2,2)=2.0D0\br
++\tab{7}FJAC(2,3)=-1.0D0\br
++\tab{7}FJAC(3,1)=0.0D0\br
++\tab{7}FJAC(3,2)=0.0D0\br
++\tab{7}FJAC(3,3)=3.0D0\br
++\tab{5}ENDIF\br
++\tab{5}END

```

```

Asp35(name): Exports == Implementation where
name : Symbol

```

```

FST    ==> FortranScalarType
FT      ==> FortranType
UFST    ==> Union(fst:FST,void:"void")
SYMTAB  ==> SymbolTable
FC       ==> FortranCode
PI       ==> PositiveInteger
RSFC    ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
FRAC    ==> Fraction
POLY    ==> Polynomial
EXPR    ==> Expression
INT     ==> Integer
FLOAT   ==> Float
VEC     ==> Vector
MAT     ==> Matrix
VF2     ==> VectorFunctions2
MFLOAT  ==> MachineFloat

```

```

FEXPR ==> FortranExpression([],['X'],MFLOAT)
MF2    ==> MatrixCategoryFunctions2(FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR,
    EXPR MFLOAT,VEC EXPR MFLOAT,VEC EXPR MFLOAT,MAT EXPR MFLOAT)
SWU    ==> Union(I:Expression Integer,F:Expression Float,
    CF:Expression Complex Float,switch:Switch)

Exports ==> FortranVectorFunctionCategory with
  coerce : VEC FEXPR -> $
    ++coerce(f) takes objects from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns them into an ASP.

Implementation ==> add

real : UFST := ["real":FST]$UFST
syms : SYMTAB := empty()$SYMTAB
declare!(N,fortranInteger(),syms)$SYMTAB
xType : FT := construct(real,[N],false)$FT
declare!(X,xType,syms)$SYMTAB
declare!(FVEC,xType,syms)$SYMTAB
declare!(LDFJAC,fortranInteger(),syms)$SYMTAB
jType : FT := construct(real,[LDFJAC,N],false)$FT
declare!(FJAC,jType,syms)$SYMTAB
declare!(IFLAG,fortranInteger(),syms)$SYMTAB
Rep := FortranProgram(name,["void"]$UFST,[N,X,FVEC,FJAC,LDFJAC,IFLAG],syms)

coerce(u:$):OutputForm == coerce(u)$Rep

makeXList(n:Integer):List(Symbol) ==
  x:Symbol := X::Symbol
  [subscript(x,[j::OutputForm])$Symbol for j in 1..n]

fexpr2expr(u:FEXPR):EXPR MFLOAT == coerce(u)$FEXPR

localAssign1(s:Symbol,j:MAT FEXPR):FC ==
  j' : MAT EXPR MFLOAT := map(fexpr2expr,j)$MF2
  assign(s,j')$FC

localAssign2(s:Symbol,j:VEC FEXPR):FC ==
  j' : VEC EXPR MFLOAT := map(fexpr2expr,j)$VF2(FEXPR,EXPR MFLOAT)
  assign(s,j')$FC

coerce(u:VEC FEXPR):$ ==
  n:Integer := maxIndex(u)
  p:List(Symbol) := makeXList(n)
  jac: MAT FEXPR := jacobian(u,p)$MultiVariableCalculusFunctions(
    Symbol,FEXPR,VEC FEXPR,List(Symbol))
  assf:FC := localAssign2(FVEC,u)
  assj:FC := localAssign1(FJAC,jac)
  iflag:SWU := [IFLAG@Symbol::EXPR(INT)]$SWU
  sw1:Switch := EQ(iflag,[1::EXPR(INT)]$SWU)

```

```

    sw2:Switch := EQ(iflag,[2::EXPR(INT)]$SWU)
    cond(sw1,assf,cond(sw2,assj)$FC)$FC::$

coerce(c:List FC):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

coerce(c:FC):$ == coerce(c)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

retract(u:VEC FRAC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

retract(u:VEC FRAC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

retract(u:VEC EXPR INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR INT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

retract(u:VEC EXPR FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

```

```

retract(u:VEC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

```

— ASP35.dotabb —

```

"ASP35" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP35"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ASP35" -> "FS"

```

domain ASP4 Asp4

— Asp4.input —

```

)set break resume
)sys rm -f Asp4.output
)spool Asp4.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp4
--R
--R Asp4(name: Symbol) is a domain constructor
--R Abbreviation for Asp4 is ASP4

```

```

--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP4
--R
--R----- Operations -----
--R coerce : FortranCode -> %           coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm           outputAsFortran : % -> Void
--R retract : Polynomial(Float) -> %    retract : Expression(Float) -> %
--R coerce : FortranExpression([construct],[construct,QUOTEX],MachineFloat) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Fraction(Polynomial(Integer)) -> %
--R retract : Fraction(Polynomial(Float)) -> %
--R retract : Polynomial(Integer) -> %
--R retract : Expression(Integer) -> %
--R retractIfCan : Fraction(Polynomial(Integer)) -> Union(%, "failed")
--R retractIfCan : Fraction(Polynomial(Float)) -> Union(%, "failed")
--R retractIfCan : Polynomial(Integer) -> Union(%, "failed")
--R retractIfCan : Polynomial(Float) -> Union(%, "failed")
--R retractIfCan : Expression(Integer) -> Union(%, "failed")
--R retractIfCan : Expression(Float) -> Union(%, "failed")
--R
--E 1

```

```

)spool
)lisp (bye)

```

— Asp4.help —

=====

Asp4 examples

=====

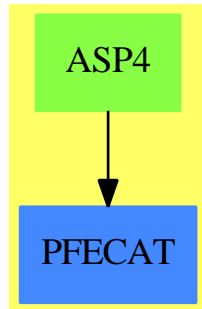
Asp4 produces Fortran for Type 4 ASPs, which take an expression
in X(1) .. X(NDIM) and produce a real function of the form:

```

DOUBLE PRECISION FUNCTION FUNCTN(NDIM,X)
DOUBLE PRECISION X(NDIM)
INTEGER NDIM
FUNCTN=(4.0D0*X(1)*X(3)**2*DEXP(2.0D0*X(1)*X(3)))/(X(4)**2+(2.0D0*
&X(2)+2.0D0)*X(4)+X(2)**2+2.0D0*X(2)+1.0D0)
RETURN
END

```

See Also:
o)show Asp4

Asp4 (ASP4)**Exports:**

coerce outputAsFortran retract retractIfCan

— domain ASP4 Asp4 —

```

)abbrev domain ASP4 Asp4
++ Author: Mike Dewar, Grant Keady and Godfrey Nolan
++ Date Created: Mar 1993
++ Date Last Updated: 6 October 1994
++ Description:
++\spadtype{Asp4} produces Fortran for Type 4 ASPs, which take an expression
++in X(1) .. X(NDIM) and produce a real function of the form:
++
++\tab{5}DOUBLE PRECISION FUNCTION FUNCTN(NDIM,X)\br
++\tab{5}DOUBLE PRECISION X(NDIM)\br
++\tab{5}INTEGER NDIM\br
++\tab{5}FUNCTN=(4.0D0*X(1)*X(3)**2*DEXP(2.0D0*X(1)*X(3)))/(X(4)**2+(2.0D0*\br
++\tab{4}&X(2)+2.0D0)*X(4)+X(2)**2+2.0D0*X(2)+1.0D0)\br
++\tab{5}RETURN\br
++\tab{5}END
  
```

```

Asp4(name): Exports == Implementation where
  name : Symbol
  
```

```

FEXPR ==> FortranExpression([],['X'],MachineFloat)
FST    ==> FortranScalarType
FT     ==> FortranType
SYMTAB ==> SymbolTable
RSFC   ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
FSTU   ==> Union(fst:FST,void:"void")
FRAC   ==> Fraction
POLY   ==> Polynomial
EXPR   ==> Expression
INT    ==> Integer
FLOAT  ==> Float
  
```

```

Exports ==> FortranFunctionCategory with
  coerce : FEXPR -> $
    ++coerce(f) takes an object from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns it into an ASP.

Implementation ==> add

real : FSTU := ["real"::FST]$FSTU
syms : SYMTAB := empty()$SYMTAB
declare!(NDIM,fortranInteger(),syms)$SYMTAB
xType : FT := construct(real,[NDIM],false)$FT
declare!(X,xType,syms)$SYMTAB
Rep := FortranProgram(name,real,[NDIM,X],syms)

retract(u:FRAC POLY INT):$ == (retract(u)$FEXPR):: $
retractIfCan(u:FRAC POLY INT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  foo::FEXPR::$

retract(u:FRAC POLY FLOAT):$ == (retract(u)$FEXPR):: $
retractIfCan(u:FRAC POLY FLOAT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  foo::FEXPR::$

retract(u:EXPR FLOAT):$ == (retract(u)$FEXPR):: $
retractIfCan(u:EXPR FLOAT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  foo::FEXPR::$

retract(u:EXPR INT):$ == (retract(u)$FEXPR):: $
retractIfCan(u:EXPR INT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  foo::FEXPR::$

retract(u:POLY FLOAT):$ == (retract(u)$FEXPR):: $
retractIfCan(u:POLY FLOAT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  foo::FEXPR::$

```



```

retract(u:POLY INT):$ == (retract(u)@FEXPR)::$
retractIfCan(u:POLY INT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  foo::FEXPR::$

coerce(u:FEXPR):$ ==
  coerce((u::Expression(MachineFloat))$FEXPR)$Rep

coerce(c:List FortranCode):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

coerce(c:FortranCode):$ == coerce(c)$Rep

coerce(u:$):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

```

— ASP4.dotabb —

```

"ASP4" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP4"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"ASP4" -> "PFECAT"

```

domain ASP41 Asp41

— Asp41.input —

```

)set break resume
)sys rm -f Asp41.output
)spool Asp41.output
)set message test on
)set message auto off
)clear all

```

```

--S 1 of 1
)show Asp41
--R
--R Asp41(nameOne: Symbol,nameTwo: Symbol,nameThree: Symbol) is a domain constructor
--R Abbreviation for Asp41 is ASP41
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP41
--R
--R----- Operations -----
--R coerce : FortranCode -> %               coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm               outputAsFortran : % -> Void
--R coerce : Vector(FortranExpression([construct,QUOTEX,QUOTEEPS],[construct,QUOTEY],MachineFloat)) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Vector(Fraction(Polynomial(Integer))) -> %
--R retract : Vector(Fraction(Polynomial(Float))) -> %
--R retract : Vector(Polynomial(Integer)) -> %
--R retract : Vector(Polynomial(Float)) -> %
--R retract : Vector(Expression(Integer)) -> %
--R retract : Vector(Expression(Float)) -> %
--R retractIfCan : Vector(Fraction(Polynomial(Integer))) -> Union(%,"failed")
--R retractIfCan : Vector(Fraction(Polynomial(Float))) -> Union(%,"failed")
--R retractIfCan : Vector(Polynomial(Integer)) -> Union(%,"failed")
--R retractIfCan : Vector(Polynomial(Float)) -> Union(%,"failed")
--R retractIfCan : Vector(Expression(Integer)) -> Union(%,"failed")
--R retractIfCan : Vector(Expression(Float)) -> Union(%,"failed")
--R
--E 1

)spool
)lisp (bye)

```

— Asp41.help —

=====

Asp41 examples

=====

Asp41 produces Fortran for Type 41 ASPs, needed for NAG routines d02raf and d02saf in particular. These ASPs are in fact three Fortran routines which return a vector of functions, and their derivatives wrt Y(i) and also a continuation parameter EPS, for example:

```

SUBROUTINE FCN(X,EPS,Y,F,N)
DOUBLE PRECISION EPS,F(N),X,Y(N)
INTEGER N
F(1)=Y(2)
F(2)=Y(3)

```

```

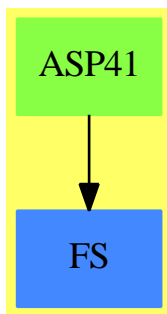
F(3)=(-1.0D0*Y(1)*Y(3))+2.0D0*EPS*Y(2)**2+(-2.0D0*EPS)
RETURN
END
SUBROUTINE JACOB(X, EPS, Y, F, N)
DOUBLE PRECISION EPS, F(N, N), X, Y(N)
INTEGER N
F(1,1)=0.0D0
F(1,2)=1.0D0
F(1,3)=0.0D0
F(2,1)=0.0D0
F(2,2)=0.0D0
F(2,3)=1.0D0
F(3,1)=-1.0D0*Y(3)
F(3,2)=4.0D0*EPS*Y(2)
F(3,3)=-1.0D0*Y(1)
RETURN
END
SUBROUTINE JACEPS(X, EPS, Y, F, N)
DOUBLE PRECISION EPS, F(N), X, Y(N)
INTEGER N
F(1)=0.0D0
F(2)=0.0D0
F(3)=2.0D0*Y(2)**2-2.0D0
RETURN
END

```

See Also:

o)show Asp41

Asp41 (ASP41)



Exports:

coerce outputAsFortran retract retractIfCan

— domain ASP41 Asp41 —

```

)abbrev domain ASP41 Asp41
++ Author: Mike Dewar, Godfrey Nolan
++ Date Created:
++ Date Last Updated: 6 October 1994
++ Description:
++\spadtype{Asp41} produces Fortran for Type 41 ASPs, needed for NAG
++routines d02raf and d02saf in particular. These ASPs are in fact
++three Fortran routines which return a vector of functions, and their
++derivatives wrt Y(i) and also a continuation parameter EPS, for example:
++
++\tab{5}SUBROUTINE FCN(X,EPS,Y,F,N)\br
++\tab{5}DOUBLE PRECISION EPS,F(N),X,Y(N)\br
++\tab{5}INTEGER N\br
++\tab{5}F(1)=Y(2)\br
++\tab{5}F(2)=Y(3)\br
++\tab{5}F(3)=(-1.0D0*Y(1)*Y(3))+2.0D0*EPS*Y(2)**2+(-2.0D0*EPS)\br
++\tab{5}RETURN\br
++\tab{5}END\br
++\tab{5}SUBROUTINE JACOFB(X,EPS,Y,F,N)\br
++\tab{5}DOUBLE PRECISION EPS,F(N,N),X,Y(N)\br
++\tab{5}INTEGER N\br
++\tab{5}F(1,1)=0.0D0\br
++\tab{5}F(1,2)=1.0D0\br
++\tab{5}F(1,3)=0.0D0\br
++\tab{5}F(2,1)=0.0D0\br
++\tab{5}F(2,2)=0.0D0\br
++\tab{5}F(2,3)=1.0D0\br
++\tab{5}F(3,1)=-1.0D0*Y(3)\br
++\tab{5}F(3,2)=4.0D0*EPS*Y(2)\br
++\tab{5}F(3,3)=-1.0D0*Y(1)\br
++\tab{5}RETURN\br
++\tab{5}END\br
++\tab{5}SUBROUTINE JACEPS(X,EPS,Y,F,N)\br
++\tab{5}DOUBLE PRECISION EPS,F(N),X,Y(N)\br
++\tab{5}INTEGER N\br
++\tab{5}F(1)=0.0D0\br
++\tab{5}F(2)=0.0D0\br
++\tab{5}F(3)=2.0D0*Y(2)**2-2.0D0\br
++\tab{5}RETURN\br
++\tab{5}END

```

```

Asp41(nameOne,nameTwo,nameThree): Exports == Implementation where
  nameOne : Symbol
  nameTwo : Symbol
  nameThree : Symbol

```

```

D      ==> differentiate

```

```

FST    ==> FortranScalarType
UFST   ==> Union(fst:FST,void:"void")
FT     ==> FortranType
FC     ==> FortranCode
SYMTAB ==> SymbolTable
RSFC   ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
FRAC   ==> Fraction
POLY   ==> Polynomial
EXPR   ==> Expression
INT    ==> Integer
FLOAT  ==> Float
VEC    ==> Vector
VF2    ==> VectorFunctions2
MFLOAT ==> MachineFloat
FEXPR  ==> FortranExpression(['X','EPS'],['Y'],MFLOAT)
S      ==> Symbol
MF2    ==> MatrixCategoryFunctions2(FEXPR,VEC FEXPR,VEC FEXPR,Matrix FEXPR,
                                   EXPR MFLOAT,VEC EXPR MFLOAT,VEC EXPR MFLOAT,Matrix EXPR MFLOAT)

Exports ==> FortranVectorFunctionCategory with
  coerce : VEC FEXPR -> $
    ++coerce(f) takes objects from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns them into an ASP.

Implementation ==> add
  real : UFST := ["real":FST]$UFST

  symOne : SYMTAB := empty()$SYMTAB
  declare!(N,fortranInteger(),symOne)$SYMTAB
  declare!(X,fortranReal(),symOne)$SYMTAB
  declare!(EPS,fortranReal(),symOne)$SYMTAB
  yType : FT := construct(real,[N],false)$FT
  declare!(Y,yType,symOne)$SYMTAB
  declare!(F,yType,symOne)$SYMTAB

  symTwo : SYMTAB := empty()$SYMTAB
  declare!(N,fortranInteger(),symTwo)$SYMTAB
  declare!(X,fortranReal(),symTwo)$SYMTAB
  declare!(EPS,fortranReal(),symTwo)$SYMTAB
  declare!(Y,yType,symTwo)$SYMTAB
  fType : FT := construct(real,[N,N],false)$FT
  declare!(F,fType,symTwo)$SYMTAB

  symThree : SYMTAB := empty()$SYMTAB
  declare!(N,fortranInteger(),symThree)$SYMTAB
  declare!(X,fortranReal(),symThree)$SYMTAB
  declare!(EPS,fortranReal(),symThree)$SYMTAB
  declare!(Y,yType,symThree)$SYMTAB
  declare!(F,yType,symThree)$SYMTAB

```

```

R1:=FortranProgram(nameOne,["void"]$UFST,[X,EPS,Y,F,N],symOne)
R2:=FortranProgram(nameTwo,["void"]$UFST,[X,EPS,Y,F,N],symTwo)
R3:=FortranProgram(nameThree,["void"]$UFST,[X,EPS,Y,F,N],symThree)
Rep := Record(f:R1,fJacob:R2,eJacob:R3)
Fsym:Symbol:=coerce "F"

fexpr2expr(u:FEXPR):EXPR MFLOAT == coerce(u)$FEXPR

localAssign1(s:S,j:Matrix FEXPR):FC ==
  j' : Matrix EXPR MFLOAT := map(fexpr2expr,j)$MF2
  assign(s,j')$FC

localAssign2(s:S,j:VEC FEXPR):FC ==
  j' : VEC EXPR MFLOAT := map(fexpr2expr,j)$VF2(FEXPR,EXPR MFLOAT)
  assign(s,j')$FC

makeCodeOne(u:VEC FEXPR):FortranCode ==
  -- simple assign
  localAssign2(Fsym,u)

makeCodeThree(u:VEC FEXPR):FortranCode ==
  -- compute jacobian wrt to eps
  jacEps:VEC FEXPR := [D(v,EPS) for v in entries(u)]$VEC(FEXPR)
  makeCodeOne(jacEps)

makeYList(n:Integer):List(Symbol) ==
  j:Integer
  y:Symbol := Y::Symbol
  p:List(Symbol) := []
  [subscript(y,[j::OutputForm])$Symbol for j in 1..n]

makeCodeTwo(u:VEC FEXPR):FortranCode ==
  -- compute jacobian wrt to f
  n:Integer := maxIndex(u)$VEC(FEXPR)
  p:List(Symbol) := makeYList(n)
  jac:Matrix(FEXPR) := _
  jacobian(u,p)$MultiVariableCalculusFunctions(S,FEXPR,VEC FEXPR,List(S))
  localAssign1(Fsym,jac)

coerce(u:VEC FEXPR):$ ==
  aF:FortranCode := makeCodeOne(u)
  bF:FortranCode := makeCodeTwo(u)
  cF:FortranCode := makeCodeThree(u)
  -- add returns() to complete subroutines
  aLF:List(FortranCode) := [aF,returns()$FortranCode]$List(FortranCode)
  bLF:List(FortranCode) := [bF,returns()$FortranCode]$List(FortranCode)
  cLF:List(FortranCode) := [cF,returns()$FortranCode]$List(FortranCode)
  [coerce(aLF)$R1,coerce(bLF)$R2,coerce(cLF)$R3]

coerce(u:$):OutputForm ==

```

```

bracket commaSeparate
  [nameOne::OutputForm,nameTwo::OutputForm,nameThree::OutputForm]

outputAsFortran(u:$):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran elt(u,f)$Rep
  outputAsFortran elt(u,fJacob)$Rep
  outputAsFortran elt(u,eJacob)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

retract(u:VEC FRAC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC FRAC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC EXPR INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR INT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC EXPR FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY INT,FEXPR)
  v::$

```

```

retractIfCan(u:VEC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

```

— ASP41.dotabb —

```

"ASP41" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP41"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ASP41" -> "FS"

```

domain ASP42 Asp42

— Asp42.input —

```

)set break resume
)sys rm -f Asp42.output
)spool Asp42.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp42
--R
--R Asp42(nameOne: Symbol,nameTwo: Symbol,nameThree: Symbol) is a domain constructor
--R Abbreviation for Asp42 is ASP42
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP42
--R
--R----- Operations -----

```



```

--R coerce : FortranCode -> %               coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm                 outputAsFortran : % -> Void
--R coerce : Vector(FortranExpression([construct,QUOTEEPS],[construct,QUOTEYA,QUOTEYB],MachineFl
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Vector(Fraction(Polynomial(Integer))) -> %
--R retract : Vector(Fraction(Polynomial(Float))) -> %
--R retract : Vector(Polynomial(Integer)) -> %
--R retract : Vector(Polynomial(Float)) -> %
--R retract : Vector(Expression(Integer)) -> %
--R retract : Vector(Expression(Float)) -> %
--R retractIfCan : Vector(Fraction(Polynomial(Integer))) -> Union(%,"failed")
--R retractIfCan : Vector(Fraction(Polynomial(Float))) -> Union(%,"failed")
--R retractIfCan : Vector(Polynomial(Integer)) -> Union(%,"failed")
--R retractIfCan : Vector(Polynomial(Float)) -> Union(%,"failed")
--R retractIfCan : Vector(Expression(Integer)) -> Union(%,"failed")
--R retractIfCan : Vector(Expression(Float)) -> Union(%,"failed")
--R
--E 1

)spool
)lisp (bye)

```

— Asp42.help —

=====

Asp42 examples

=====

Asp42 produces Fortran for Type 42 ASPs, needed for NAG routines d02raf and d02saf in particular. These ASPs are in fact three Fortran routines which return a vector of functions, and their derivatives wrt Y(i) and also a continuation parameter EPS, for example:

```

SUBROUTINE G(EPS,YA,YB,BC,N)
DOUBLE PRECISION EPS,YA(N),YB(N),BC(N)
INTEGER N
BC(1)=YA(1)
BC(2)=YA(2)
BC(3)=YB(2)-1.0D0
RETURN
END
SUBROUTINE JACOBG(EPS,YA,YB,AJ,BJ,N)
DOUBLE PRECISION EPS,YA(N),AJ(N,N),BJ(N,N),YB(N)
INTEGER N
AJ(1,1)=1.0D0
AJ(1,2)=0.0D0
AJ(1,3)=0.0D0

```

```

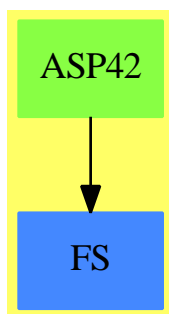
AJ(2,1)=0.0D0
AJ(2,2)=1.0D0
AJ(2,3)=0.0D0
AJ(3,1)=0.0D0
AJ(3,2)=0.0D0
AJ(3,3)=0.0D0
BJ(1,1)=0.0D0
BJ(1,2)=0.0D0
BJ(1,3)=0.0D0
BJ(2,1)=0.0D0
BJ(2,2)=0.0D0
BJ(2,3)=0.0D0
BJ(3,1)=0.0D0
BJ(3,2)=1.0D0
BJ(3,3)=0.0D0
RETURN
END
SUBROUTINE JACGEP(EPS,YA,YB,BCEP,N)
DOUBLE PRECISION EPS,YA(N),YB(N),BCEP(N)
INTEGER N
BCEP(1)=0.0D0
BCEP(2)=0.0D0
BCEP(3)=0.0D0
RETURN
END

```

See Also:

o)show Asp42

Asp42 (ASP42)



Exports:

coerce outputAsFortran retract retractIfCan

— domain ASP42 Asp42 —

```

)abbrev domain ASP42 Asp42
++ Author: Mike Dewar, Godfrey Nolan
++ Date Created:
++ Date Last Updated: 6 October 1994
++ Description:
++\spadtype{Asp42} produces Fortran for Type 42 ASPs, needed for NAG
++routines d02raf and d02saf
++in particular. These ASPs are in fact
++three Fortran routines which return a vector of functions, and their
++derivatives wrt Y(i) and also a continuation parameter EPS, for example:
++
++\tab{5}SUBROUTINE G(EPS,YA,YB,BC,N)\br
++\tab{5}DOUBLE PRECISION EPS,YA(N),YB(N),BC(N)\br
++\tab{5}INTEGER N\br
++\tab{5}BC(1)=YA(1)\br
++\tab{5}BC(2)=YA(2)\br
++\tab{5}BC(3)=YB(2)-1.0D0\br
++\tab{5}RETURN\br
++\tab{5}END\br
++\tab{5}SUBROUTINE JACOBG(EPS,YA,YB,AJ,BJ,N)\br
++\tab{5}DOUBLE PRECISION EPS,YA(N),AJ(N,N),BJ(N,N),YB(N)\br
++\tab{5}INTEGER N\br
++\tab{5}AJ(1,1)=1.0D0\br
++\tab{5}AJ(1,2)=0.0D0\br
++\tab{5}AJ(1,3)=0.0D0\br
++\tab{5}AJ(2,1)=0.0D0\br
++\tab{5}AJ(2,2)=1.0D0\br
++\tab{5}AJ(2,3)=0.0D0\br
++\tab{5}AJ(3,1)=0.0D0\br
++\tab{5}AJ(3,2)=0.0D0\br
++\tab{5}AJ(3,3)=0.0D0\br
++\tab{5}BJ(1,1)=0.0D0\br
++\tab{5}BJ(1,2)=0.0D0\br
++\tab{5}BJ(1,3)=0.0D0\br
++\tab{5}BJ(2,1)=0.0D0\br
++\tab{5}BJ(2,2)=0.0D0\br
++\tab{5}BJ(2,3)=0.0D0\br
++\tab{5}BJ(3,1)=0.0D0\br
++\tab{5}BJ(3,2)=1.0D0\br
++\tab{5}BJ(3,3)=0.0D0\br
++\tab{5}RETURN\br
++\tab{5}END\br
++\tab{5}SUBROUTINE JACGEP(EPS,YA,YB,BCEP,N)\br
++\tab{5}DOUBLE PRECISION EPS,YA(N),YB(N),BCEP(N)\br
++\tab{5}INTEGER N\br
++\tab{5}BCEP(1)=0.0D0\br
++\tab{5}BCEP(2)=0.0D0\br

```

```

++\tab{5}BCEP(3)=0.0D0\br
++\tab{5}RETURN\br
++\tab{5}END

```

```

Asp42(nameOne,nameTwo,nameThree): Exports == Implementation where
  nameOne : Symbol
  nameTwo : Symbol
  nameThree : Symbol

```

```

D      ==> differentiate
FST    ==> FortranScalarType
FT     ==> FortranType
FP     ==> FortranProgram
FC     ==> FortranCode
PI     ==> PositiveInteger
NNI    ==> NonNegativeInteger
SYMTAB ==> SymbolTable
RSFC   ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
UFST   ==> Union(fst:FST,void:"void")
FRAC   ==> Fraction
POLY   ==> Polynomial
EXPR   ==> Expression
INT    ==> Integer
FLOAT  ==> Float
VEC    ==> Vector
VF2    ==> VectorFunctions2
MFLOAT ==> MachineFloat
FEXPR  ==> FortranExpression(['EPS'],['YA','YB'],MFLOAT)
S      ==> Symbol
MF2    ==> MatrixCategoryFunctions2(FEXPR,VEC FEXPR,VEC FEXPR,Matrix FEXPR,
    EXPR MFLOAT,VEC EXPR MFLOAT,VEC EXPR MFLOAT,Matrix EXPR MFLOAT)

```

```

Exports ==> FortranVectorFunctionCategory with
  coerce : VEC FEXPR -> $
    ++coerce(f) takes objects from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns them into an ASP.

```

```

Implementation ==> add
  real : UFST := ["real"::FST]$UFST

  symOne : SYMTAB := empty()$SYMTAB
  declare!(EPS,fortranReal(),symOne)$SYMTAB
  declare!(N,fortranInteger(),symOne)$SYMTAB
  yType : FT := construct(real,[N],false)$FT
  declare!(YA,yType,symOne)$SYMTAB
  declare!(YB,yType,symOne)$SYMTAB
  declare!(BC,yType,symOne)$SYMTAB

  symTwo : SYMTAB := empty()$SYMTAB
  declare!(EPS,fortranReal(),symTwo)$SYMTAB

```

```

declare! (N,fortranInteger(),symTwo)$SYMTAB
declare! (YA,yType,symTwo)$SYMTAB
declare! (YB,yType,symTwo)$SYMTAB
ajType : FT := construct(real,[N,N],false)$FT
declare! (AJ,ajType,symTwo)$SYMTAB
declare! (BJ,ajType,symTwo)$SYMTAB

symThree : SYMTAB := empty()$SYMTAB
declare! (EPS,fortranReal(),symThree)$SYMTAB
declare! (N,fortranInteger(),symThree)$SYMTAB
declare! (YA,yType,symThree)$SYMTAB
declare! (YB,yType,symThree)$SYMTAB
declare! (BCEP,yType,symThree)$SYMTAB

rt := ["void"]$UFST
R1:=FortranProgram(nameOne,rt,[EPS,YA,YB,BC,N],symOne)
R2:=FortranProgram(nameTwo,rt,[EPS,YA,YB,AJ,BJ,N],symTwo)
R3:=FortranProgram(nameThree,rt,[EPS,YA,YB,BCEP,N],symThree)
Rep := Record(g:R1,gJacob:R2,geJacob:R3)
BCsym:Symbol:=coerce "BC"
AJsym:Symbol:=coerce "AJ"
BJsym:Symbol:=coerce "BJ"
BCEPsym:Symbol:=coerce "BCEP"

makeList(n:Integer,s:Symbol):List(Symbol) ==
  j:Integer
  p:List(Symbol) := []
  for j in 1 .. n repeat p:= cons(subscript(s,[j::OutputForm])$Symbol,p)
  reverse(p)

fexpr2expr(u:FEXPR):EXPR MFLOAT == coerce(u)$FEXPR

localAssign1(s:S,j:Matrix FEXPR):FC ==
  j' : Matrix EXPR MFLOAT := map(fexpr2expr,j)$MF2
  assign(s,j')$FC

localAssign2(s:S,j:VEC FEXPR):FC ==
  j' : VEC EXPR MFLOAT := map(fexpr2expr,j)$VF2(FEXPR,EXPR MFLOAT)
  assign(s,j')$FC

makeCodeOne(u:VEC FEXPR):FortranCode ==
  -- simple assign
  localAssign2(BCsym,u)

makeCodeTwo(u:VEC FEXPR):List(FortranCode) ==
  -- compute jacobian wrt to ya
  n:Integer := maxIndex(u)
  p:List(Symbol) := makeList(n,YA::Symbol)
  jacYA:Matrix(FEXPR) := _
    jacobian(u,p)$MultiVariableCalculusFunctions(S,FEXPR,VEC FEXPR,List(S))

```

```

-- compute jacobian wrt to yb
p:List(Symbol) := makeList(n,YB::Symbol)
jacYB: Matrix(FEXPR) := _
    jacobian(u,p)$MultiVariableCalculusFunctions(S,FEXPR,VEC FEXPR,List(S))
-- assign jacobians to AJ & BJ
[localAssign1(AJsym,jacYA),localAssign1(BJsym,jacYB),returns()$FC]$List(FC)

makeCodeThree(u:VEC FEXPR):FortranCode ==
-- compute jacobian wrt to eps
jacEps:VEC FEXPR := [D(v,EPS) for v in entries u]$VEC(FEXPR)
localAssign2(BCEPsym,jacEps)

coerce(u:VEC FEXPR):$ ==
aF:FortranCode := makeCodeOne(u)
bF:List(FortranCode) := makeCodeTwo(u)
cF:FortranCode := makeCodeThree(u)
-- add returns() to complete subroutines
aLF:List(FortranCode) := [aF,returns()$FC]$List(FortranCode)
cLF:List(FortranCode) := [cF,returns()$FC]$List(FortranCode)
[coerce(aLF)$R1,coerce(bF)$R2,coerce(cLF)$R3]

coerce(u:$) : OutputForm ==
    bracket commaSeparate
        [nameOne::OutputForm,nameTwo::OutputForm,nameThree::OutputForm]

outputAsFortran(u:$):Void ==
    p := checkPrecision()$NAGLinkSupportPackage
    outputAsFortran elt(u,g)$Rep
    outputAsFortran elt(u,gJacob)$Rep
    outputAsFortran elt(u,geJacob)$Rep
    p => restorePrecision()$NAGLinkSupportPackage

retract(u:VEC FRAC POLY INT):$ ==
    v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY INT,FEXPR)
    v::$

retractIfCan(u:VEC FRAC POLY INT):Union($,"failed") ==
    v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY INT,FEXPR)
    v case "failed" => "failed"
    (v::VEC FEXPR):: $

retract(u:VEC FRAC POLY FLOAT):$ ==
    v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY FLOAT,FEXPR)
    v::$

retractIfCan(u:VEC FRAC POLY FLOAT):Union($,"failed") ==
    v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY FLOAT,FEXPR)
    v case "failed" => "failed"
    (v::VEC FEXPR):: $

```

```

retract(u:VEC EXPR INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR INT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC EXPR FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

```

— ASP42.dotabb —

"ASP42" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP42"]

"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]

"ASP42" -> "FS"

domain ASP49 Asp49**— Asp49.input —**

```

)set break resume
)sys rm -f Asp49.output
)spool Asp49.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp49
--R
--R Asp49(name: Symbol) is a domain constructor
--R Abbreviation for Asp49 is ASP49
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP49
--R
--R----- Operations -----
--R coerce : FortranCode -> %          coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm          outputAsFortran : % -> Void
--R retract : Polynomial(Float) -> %    retract : Expression(Float) -> %
--R coerce : FortranExpression([construct],[construct,QUOTEX],MachineFloat) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Fraction(Polynomial(Integer)) -> %
--R retract : Fraction(Polynomial(Float)) -> %
--R retract : Polynomial(Integer) -> %
--R retract : Expression(Integer) -> %
--R retractIfCan : Fraction(Polynomial(Integer)) -> Union(%, "failed")
--R retractIfCan : Fraction(Polynomial(Float)) -> Union(%, "failed")
--R retractIfCan : Polynomial(Integer) -> Union(%, "failed")
--R retractIfCan : Polynomial(Float) -> Union(%, "failed")
--R retractIfCan : Expression(Integer) -> Union(%, "failed")
--R retractIfCan : Expression(Float) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)

```

— Asp49.help —

```

=====
Asp49 examples
=====

```


Asp49 produces Fortran for Type 49 ASPs, needed for NAG routines e04dgm, e04ucf, for example:

```

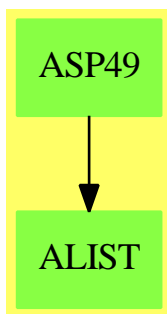
SUBROUTINE OBJFUN(MODE,N,X,OBJF,OBJGRD,NSTATE,IUSER,USER)
DOUBLE PRECISION X(N),OBJF,OBJGRD(N),USER(*)
INTEGER N,IUSER(*),MODE,NSTATE
OBJF=X(4)*X(9)+((-1.0D0*X(5))+X(3))*X(8)+((-1.0D0*X(3))+X(1))*X(7)
&+((-1.0D0*X(2))*X(6))
OBJGRD(1)=X(7)
OBJGRD(2)=-1.0D0*X(6)
OBJGRD(3)=X(8)+(-1.0D0*X(7))
OBJGRD(4)=X(9)
OBJGRD(5)=-1.0D0*X(8)
OBJGRD(6)=-1.0D0*X(2)
OBJGRD(7)=(-1.0D0*X(3))+X(1)
OBJGRD(8)=(-1.0D0*X(5))+X(3)
OBJGRD(9)=X(4)
RETURN
END

```

See Also:

o)show Asp49

Asp49 (ASP49)



Exports:

coerce outputAsFortran retract retractIfCan

— domain ASP49 Asp49 —

)abbrev domain ASP49 Asp49

```

++ Author: Mike Dewar, Grant Keady and Godfrey Nolan
++ Date Created: Mar 1993
++ Date Last Updated: 6 October 1994
++ Description:
++\spadtype{Asp49} produces Fortran for Type 49 ASPs, needed for NAG routines
++e04dgm, e04ucf, for example:
++
++\tab{5}SUBROUTINE OBJFUN(MODE,N,X,OBJF,OBJGRD,NSTATE,IUSER,USER)\br
++\tab{5}DOUBLE PRECISION X(N),OBJF,OBJGRD(N),USER(*)\br
++\tab{5}INTEGER N,IUSER(*),MODE,NSTATE\br
++\tab{5}OBJF=X(4)*X(9)+((-1.0D0*X(5))+X(3))*X(8)+((-1.0D0*X(3))+X(1))*X(7)\br
++\tab{4}&+((-1.0D0*X(2)*X(6))\br
++\tab{5}OBJGRD(1)=X(7)\br
++\tab{5}OBJGRD(2)=-1.0D0*X(6)\br
++\tab{5}OBJGRD(3)=X(8)+(-1.0D0*X(7))\br
++\tab{5}OBJGRD(4)=X(9)\br
++\tab{5}OBJGRD(5)=-1.0D0*X(8)\br
++\tab{5}OBJGRD(6)=-1.0D0*X(2)\br
++\tab{5}OBJGRD(7)=(-1.0D0*X(3))+X(1)\br
++\tab{5}OBJGRD(8)=(-1.0D0*X(5))+X(3)\br
++\tab{5}OBJGRD(9)=X(4)\br
++\tab{5}RETURN\br
++\tab{5}END

```

```

Asp49(name): Exports == Implementation where
  name : Symbol

```

```

FST    ==> FortranScalarType
UFST   ==> Union(fst:FST,void:"void")
FT     ==> FortranType
FC     ==> FortranCode
SYMTAB ==> SymbolTable
RSFC   ==> Record(localSymbols:SymbolTable,code:List(FC))
MFLOAT ==> MachineFloat
FEXPR  ==> FortranExpression([],['X'],MFLOAT)
FRAC   ==> Fraction
POLY   ==> Polynomial
EXPR   ==> Expression
INT     ==> Integer
FLOAT  ==> Float
VEC     ==> Vector
VF2     ==> VectorFunctions2
S       ==> Symbol

```

```

Exports ==> FortranFunctionCategory with
  coerce : FEXPR -> $
    ++coerce(f) takes an object from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns it into an ASP.

```

```

Implementation ==> add

```

```

real : UFST := ["real"::FST]$UFST
integer : UFST := ["integer"::FST]$UFST
syms : SYMTAB := empty()$SYMTAB
declare!(MODE,fortranInteger(),syms)$SYMTAB
declare!(N,fortranInteger(),syms)$SYMTAB
xType : FT := construct(real,[N::S],false)$FT
declare!(X,xType,syms)$SYMTAB
declare!(OBJF,fortranReal(),syms)$SYMTAB
declare!(OBJGRD,xType,syms)$SYMTAB
declare!(NSTATE,fortranInteger(),syms)$SYMTAB
iuType : FT := construct(integer,["*::S"],false)$FT
declare!(IUSER,iuType,syms)$SYMTAB
uType : FT := construct(real,["*::S"],false)$FT
declare!(USER,uType,syms)$SYMTAB
Rep := FortranProgram(name,["void"]$UFST,
                        [MODE,N,X,OBJF,OBJGRD,NSTATE,IUSER,USER],syms)

fexpr2expr(u:FEXPR):EXPR MFLOAT == coerce(u)$FEXPR

localAssign(s:S,j:VEC FEXPR):FC ==
  j' : VEC EXPR MFLOAT := map(fexpr2expr,j)$VF2(FEXPR,EXPR MFLOAT)
  assign(s,j')$FC

coerce(u:FEXPR):$ ==
  vars:List(S) := variables(u)
  grd:VEC FEXPR := gradient(u,vars)$MultiVariableCalculusFunctions(
    S,FEXPR,VEC FEXPR,List(S))
  code : List(FC) := [assign(OBJF@S,fexpr2expr u)$FC,
    localAssign(OBJGRD@S,grd),
    returns()$FC]
  code::$

coerce(u:$):OutputForm == coerce(u)$Rep

coerce(c:List FC):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

coerce(c:FC):$ == coerce(c)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

retract(u:FRAC POLY INT):$ == (retract(u)$FEXPR):$
retractIfCan(u:FRAC POLY INT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR

```

```

foo case "failed" => "failed"
(foo::FEXPR)::$(

retract(u:FRAC POLY FLOAT):$ == (retract(u)@FEXPR)::$(
retractIfCan(u:FRAC POLY FLOAT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  (foo::FEXPR)::$(

retract(u:EXPR FLOAT):$ == (retract(u)@FEXPR)::$(
retractIfCan(u:EXPR FLOAT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  (foo::FEXPR)::$(

retract(u:EXPR INT):$ == (retract(u)@FEXPR)::$(
retractIfCan(u:EXPR INT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  (foo::FEXPR)::$(

retract(u:POLY FLOAT):$ == (retract(u)@FEXPR)::$(
retractIfCan(u:POLY FLOAT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  (foo::FEXPR)::$(

retract(u:POLY INT):$ == (retract(u)@FEXPR)::$(
retractIfCan(u:POLY INT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  (foo::FEXPR)::$(

```

— ASP49.dotabb —

```

"ASP49" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP49"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ASP49" -> "ALIST"

```

domain ASP50 Asp50

— Asp50.input —

```
)set break resume
)sys rm -f Asp50.output
)spool Asp50.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp50
--R
--R Asp50(name: Symbol) is a domain constructor
--R Abbreviation for Asp50 is ASP50
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP50
--R
--R----- Operations -----
--R coerce : FortranCode -> %               coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm               outputAsFortran : % -> Void
--R coerce : Vector(FortranExpression([construct],[construct,QUOTEXC],MachineFloat)) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Vector(Fraction(Polynomial(Integer))) -> %
--R retract : Vector(Fraction(Polynomial(Float))) -> %
--R retract : Vector(Polynomial(Integer)) -> %
--R retract : Vector(Polynomial(Float)) -> %
--R retract : Vector(Expression(Integer)) -> %
--R retract : Vector(Expression(Float)) -> %
--R retractIfCan : Vector(Fraction(Polynomial(Integer))) -> Union(%,"failed")
--R retractIfCan : Vector(Fraction(Polynomial(Float))) -> Union(%,"failed")
--R retractIfCan : Vector(Polynomial(Integer)) -> Union(%,"failed")
--R retractIfCan : Vector(Polynomial(Float)) -> Union(%,"failed")
--R retractIfCan : Vector(Expression(Integer)) -> Union(%,"failed")
--R retractIfCan : Vector(Expression(Float)) -> Union(%,"failed")
--R
--E 1

)spool
)lisp (bye)
```

— Asp50.help —

```
=====
Asp50 examples
```

=====

Asp50 produces Fortran for Type 50 ASPs, needed for NAG routine
e04fdf, for example:

```

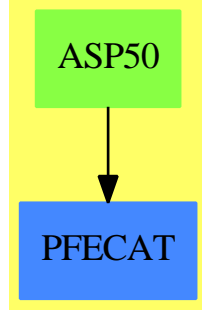
SUBROUTINE LSFUN1(M,N,XC,FVECC)
DOUBLE PRECISION FVECC(M),XC(N)
INTEGER I,M,N
FVECC(1)=((XC(1)-2.4D0)*XC(3)+(15.0D0*XC(1)-36.0D0)*XC(2)+1.0D0)/(
&XC(3)+15.0D0*XC(2))
FVECC(2)=((XC(1)-2.8D0)*XC(3)+(7.0D0*XC(1)-19.6D0)*XC(2)+1.0D0)/(X
&C(3)+7.0D0*XC(2))
FVECC(3)=((XC(1)-3.2D0)*XC(3)+(4.333333333333333D0*XC(1)-13.866666
&66666667D0)*XC(2)+1.0D0)/(XC(3)+4.333333333333333D0*XC(2))
FVECC(4)=((XC(1)-3.5D0)*XC(3)+(3.0D0*XC(1)-10.5D0)*XC(2)+1.0D0)/(X
&C(3)+3.0D0*XC(2))
FVECC(5)=((XC(1)-3.9D0)*XC(3)+(2.2D0*XC(1)-8.579999999999998D0)*XC
&(2)+1.0D0)/(XC(3)+2.2D0*XC(2))
FVECC(6)=((XC(1)-4.199999999999999D0)*XC(3)+(1.666666666666667D0*X
&C(1)-7.0D0)*XC(2)+1.0D0)/(XC(3)+1.666666666666667D0*XC(2))
FVECC(7)=((XC(1)-4.5D0)*XC(3)+(1.285714285714286D0*XC(1)-5.7857142
&85714286D0)*XC(2)+1.0D0)/(XC(3)+1.285714285714286D0*XC(2))
FVECC(8)=((XC(1)-4.899999999999999D0)*XC(3)+(XC(1)-4.8999999999999
&99D0)*XC(2)+1.0D0)/(XC(3)+XC(2))
FVECC(9)=((XC(1)-4.699999999999999D0)*XC(3)+(XC(1)-4.6999999999999
&99D0)*XC(2)+1.285714285714286D0)/(XC(3)+XC(2))
FVECC(10)=((XC(1)-6.8D0)*XC(3)+(XC(1)-6.8D0)*XC(2)+1.666666666666
&67D0)/(XC(3)+XC(2))
FVECC(11)=((XC(1)-8.299999999999999D0)*XC(3)+(XC(1)-8.2999999999999
&999D0)*XC(2)+2.2D0)/(XC(3)+XC(2))
FVECC(12)=((XC(1)-10.6D0)*XC(3)+(XC(1)-10.6D0)*XC(2)+3.0D0)/(XC(3)
&+XC(2))
FVECC(13)=((XC(1)-1.34D0)*XC(3)+(XC(1)-1.34D0)*XC(2)+4.33333333333
&3333D0)/(XC(3)+XC(2))
FVECC(14)=((XC(1)-2.1D0)*XC(3)+(XC(1)-2.1D0)*XC(2)+7.0D0)/(XC(3)+X
&C(2))
FVECC(15)=((XC(1)-4.39D0)*XC(3)+(XC(1)-4.39D0)*XC(2)+15.0D0)/(XC(3)
&+XC(2))
END

```

See Also:

o)show Asp50

Asp50 (ASP50)



Exports:

coerce outputAsFortran retract retractIfCan

— domain ASP50 Asp50 —

```

)abbrev domain ASP50 Asp50
++ Author: Mike Dewar, Grant Keady and Godfrey Nolan
++ Date Created: Mar 1993
++ Date Last Updated: 6 October 1994
++ Description:
++\spadtype{Asp50} produces Fortran for Type 50 ASPs, needed for NAG routine
++e04fdf, for example:
++
++\tab{5}SUBROUTINE LSFUN1(M,N,XC,FVECC)\br
++\tab{5}DOUBLE PRECISION FVECC(M),XC(N)\br
++\tab{5}INTEGER I,M,N\br
++\tab{5}FVECC(1)=((XC(1)-2.4D0)*XC(3)+(15.0D0*XC(1)-36.0D0)*XC(2)+1.0D0)/(\br
++\tab{4}&XC(3)+15.0D0*XC(2))\br
++\tab{5}FVECC(2)=((XC(1)-2.8D0)*XC(3)+(7.0D0*XC(1)-19.6D0)*XC(2)+1.0D0)/(X\br
++\tab{4}&C(3)+7.0D0*XC(2))\br
++\tab{5}FVECC(3)=((XC(1)-3.2D0)*XC(3)+(4.333333333333333D0*XC(1)-13.866666\br
++\tab{4}&66666667D0)*XC(2)+1.0D0)/(XC(3)+4.333333333333333D0*XC(2))\br
++\tab{5}FVECC(4)=((XC(1)-3.5D0)*XC(3)+(3.0D0*XC(1)-10.5D0)*XC(2)+1.0D0)/(X\br
++\tab{4}&C(3)+3.0D0*XC(2))\br
++\tab{5}FVECC(5)=((XC(1)-3.9D0)*XC(3)+(2.2D0*XC(1)-8.579999999999999D0)*XC\br
++\tab{4}&(2)+1.0D0)/(XC(3)+2.2D0*XC(2))\br
++\tab{5}FVECC(6)=((XC(1)-4.199999999999999D0)*XC(3)+(1.666666666666667D0*X\br
++\tab{4}&C(1)-7.0D0)*XC(2)+1.0D0)/(XC(3)+1.666666666666667D0*XC(2))\br
++\tab{5}FVECC(7)=((XC(1)-4.5D0)*XC(3)+(1.285714285714286D0*XC(1)-5.7857142\br
++\tab{4}&85714286D0)*XC(2)+1.0D0)/(XC(3)+1.285714285714286D0*XC(2))\br
++\tab{5}FVECC(8)=((XC(1)-4.899999999999999D0)*XC(3)+(XC(1)-4.899999999999\br
++\tab{4}&99D0)*XC(2)+1.0D0)/(XC(3)+XC(2))\br
++\tab{5}FVECC(9)=((XC(1)-4.699999999999999D0)*XC(3)+(XC(1)-4.699999999999\br
++\tab{4}&99D0)*XC(2)+1.285714285714286D0)/(XC(3)+XC(2))\br
++\tab{5}FVECC(10)=((XC(1)-6.8D0)*XC(3)+(XC(1)-6.8D0)*XC(2)+1.666666666666\br

```

```

++\tab{4}&67D0)/(XC(3)+XC(2))\br
++\tab{5}FVECC(11)=(XC(1)-8.299999999999999D0)*XC(3)+(XC(1)-8.299999999999999\br
++\tab{4}&999D0)*XC(2)+2.2D0)/(XC(3)+XC(2))\br
++\tab{5}FVECC(12)=(XC(1)-10.6D0)*XC(3)+(XC(1)-10.6D0)*XC(2)+3.0D0)/(XC(3)\br
++\tab{4}&+XC(2))\br
++\tab{5}FVECC(13)=(XC(1)-1.34D0)*XC(3)+(XC(1)-1.34D0)*XC(2)+4.33333333333\br
++\tab{4}&3333D0)/(XC(3)+XC(2))\br
++\tab{5}FVECC(14)=(XC(1)-2.1D0)*XC(3)+(XC(1)-2.1D0)*XC(2)+7.0D0)/(XC(3)+X\br
++\tab{4}&C(2))\br
++\tab{5}FVECC(15)=(XC(1)-4.39D0)*XC(3)+(XC(1)-4.39D0)*XC(2)+15.0D0)/(XC(3)\br
++\tab{4}&)+XC(2))\br
++\tab{5}END

```

```

Asp50(name): Exports == Implementation where
  name : Symbol

```

```

FST    ==> FortranScalarType
FT      ==> FortranType
UFST    ==> Union(fst:FST,void:"void")
SYMTAB  ==> SymbolTable
RSFC    ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
FRAC    ==> Fraction
POLY    ==> Polynomial
EXPR    ==> Expression
INT      ==> Integer
FLOAT   ==> Float
VEC      ==> Vector
VF2     ==> VectorFunctions2
FEXPR   ==> FortranExpression([],['XC'],MFLOAT)
MFLOAT  ==> MachineFloat

```

```

Exports ==> FortranVectorFunctionCategory with
  coerce : VEC FEXPR -> $
    ++coerce(f) takes objects from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns them into an ASP.

```

```

Implementation ==> add

```

```

real : UFST := ["real"::FST]$UFST
syms : SYMTAB := empty()$SYMTAB
declare!(M,fortranInteger(),syms)$SYMTAB
declare!(N,fortranInteger(),syms)$SYMTAB
xcType : FT := construct(real,[N],false)$FT
declare!(XC,xcType,syms)$SYMTAB
fveccType : FT := construct(real,[M],false)$FT
declare!(FVECC,fveccType,syms)$SYMTAB
declare!(I,fortranInteger(),syms)$SYMTAB
tType : FT := construct(real,[M,N],false)$FT
-- declare!(TC,tType,syms)$SYMTAB
-- declare!(Y,fveccType,syms)$SYMTAB

```



```

Rep := FortranProgram(name,["void"]$UFST, [M,N,XC,FVECC],syms)

fexpr2expr(u:FEXPR):EXPR MFLOAT == coerce(u)$FEXPR

coerce(u:VEC FEXPR):$ ==
  u' : VEC EXPR MFLOAT := map(fexpr2expr,u)$VF2(FEXPR,EXPR MFLOAT)
  assign(FVECC,u')$FortranCode::$

retract(u:VEC FRAC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

retract(u:VEC FRAC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

retract(u:VEC EXPR INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR INT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

retract(u:VEC EXPR FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

retract(u:VEC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY INT,FEXPR)

```

```

v case "failed" => "failed"
(v::VEC FEXPR):: $

retract(u:VEC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY FLOAT,FEXPR)
  v:: $

retractIfCan(u:VEC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

coerce(c:List FortranCode):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

coerce(c:FortranCode):$ == coerce(c)$Rep

coerce(u:$):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

```

— ASP50.dotabb —

```

"ASP50" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP50"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"ASP50" -> "PFECAT"

```

domain ASP55 Asp55

— Asp55.input —

```

)set break resume
)sys rm -f Asp55.output
)spool Asp55.output
)set message test on
)set message auto off

```

```

)clear all

--S 1 of 1
)show Asp55
--R
--R Asp55(name: Symbol) is a domain constructor
--R Abbreviation for Asp55 is ASP55
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP55
--R
--R----- Operations -----
--R coerce : FortranCode -> %               coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm               outputAsFortran : % -> Void
--R coerce : Vector(FortranExpression([construct],[construct,QUOTEX],MachineFloat)) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Vector(Fraction(Polynomial(Integer))) -> %
--R retract : Vector(Fraction(Polynomial(Float))) -> %
--R retract : Vector(Polynomial(Integer)) -> %
--R retract : Vector(Polynomial(Float)) -> %
--R retract : Vector(Expression(Integer)) -> %
--R retract : Vector(Expression(Float)) -> %
--R retractIfCan : Vector(Fraction(Polynomial(Integer))) -> Union(%,"failed")
--R retractIfCan : Vector(Fraction(Polynomial(Float))) -> Union(%,"failed")
--R retractIfCan : Vector(Polynomial(Integer)) -> Union(%,"failed")
--R retractIfCan : Vector(Polynomial(Float)) -> Union(%,"failed")
--R retractIfCan : Vector(Expression(Integer)) -> Union(%,"failed")
--R retractIfCan : Vector(Expression(Float)) -> Union(%,"failed")
--R
--E 1

)spool
)lisp (bye)

```

— Asp55.help —

=====

Asp55 examples

=====

Asp55 produces Fortran for Type 55 ASPs, needed for NAG routines
e04dgc and e04ucf, for example:

```

SUBROUTINE CONFUN(MODE,NCNLN,N,NROWJ,NEEDC,X,C,CJAC,NSTATE,IUSER
&,USER)
DOUBLE PRECISION C(NCNLN),X(N),CJAC(NROWJ,N),USER(*)
INTEGER N,IUSER(*),NEEDC(NCNLN),NROWJ,MODE,NCNLN,NSTATE
IF(NEEDC(1).GT.0)THEN

```

```

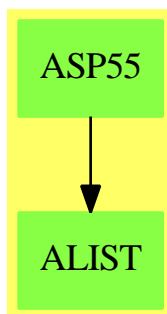
C(1)=X(6)**2+X(1)**2
CJAC(1,1)=2.0D0*X(1)
CJAC(1,2)=0.0D0
CJAC(1,3)=0.0D0
CJAC(1,4)=0.0D0
CJAC(1,5)=0.0D0
CJAC(1,6)=2.0D0*X(6)
ENDIF
IF(NEEDEC(2).GT.0)THEN
C(2)=X(2)**2+(-2.0D0*X(1)*X(2))+X(1)**2
CJAC(2,1)=(-2.0D0*X(2))+2.0D0*X(1)
CJAC(2,2)=2.0D0*X(2)+(-2.0D0*X(1))
CJAC(2,3)=0.0D0
CJAC(2,4)=0.0D0
CJAC(2,5)=0.0D0
CJAC(2,6)=0.0D0
ENDIF
IF(NEEDEC(3).GT.0)THEN
C(3)=X(3)**2+(-2.0D0*X(1)*X(3))+X(2)**2+X(1)**2
CJAC(3,1)=(-2.0D0*X(3))+2.0D0*X(1)
CJAC(3,2)=2.0D0*X(2)
CJAC(3,3)=2.0D0*X(3)+(-2.0D0*X(1))
CJAC(3,4)=0.0D0
CJAC(3,5)=0.0D0
CJAC(3,6)=0.0D0
ENDIF
RETURN
END

```

See Also:

o)show Asp55

Asp55 (ASP55)



Exports:

coerce outputAsFortran retract retractIfCan

— domain ASP55 Asp55 —

```

)abbrev domain ASP55 Asp55
++ Author: Mike Dewar, Grant Keady and Godfrey Nolan
++ Date Created: June 1993
++ Date Last Updated: 6 October 1994
++ Description:
++\spadtype{Asp55} produces Fortran for Type 55 ASPs, needed for NAG routines
++e04dgm and e04ucf, for example:
++
++\tab{5}SUBROUTINE CONFUN(MODE,NCNLN,N,NROWJ,NEEDC,X,C,CJAC,NSTATE,IUSER\br
++\tab{4}&,USER)\br
++\tab{5}DOUBLE PRECISION C(NCNLN),X(N),CJAC(NROWJ,N),USER(*)\br
++\tab{5}INTEGER N,IUSER(*),NEEDC(NCNLN),NROWJ,MODE,NCNLN,NSTATE\br
++\tab{5}IF(NEEDC(1).GT.0)THEN\br
++\tab{7}C(1)=X(6)**2+X(1)**2\br
++\tab{7}CJAC(1,1)=2.0D0*X(1)\br
++\tab{7}CJAC(1,2)=0.0D0\br
++\tab{7}CJAC(1,3)=0.0D0\br
++\tab{7}CJAC(1,4)=0.0D0\br
++\tab{7}CJAC(1,5)=0.0D0\br
++\tab{7}CJAC(1,6)=2.0D0*X(6)\br
++\tab{5}ENDIF\br
++\tab{5}IF(NEEDC(2).GT.0)THEN\br
++\tab{7}C(2)=X(2)**2+(-2.0D0*X(1)*X(2))+X(1)**2\br
++\tab{7}CJAC(2,1)=(-2.0D0*X(2))+2.0D0*X(1)\br
++\tab{7}CJAC(2,2)=2.0D0*X(2)+(-2.0D0*X(1))\br
++\tab{7}CJAC(2,3)=0.0D0\br
++\tab{7}CJAC(2,4)=0.0D0\br
++\tab{7}CJAC(2,5)=0.0D0\br
++\tab{7}CJAC(2,6)=0.0D0\br
++\tab{5}ENDIF\br
++\tab{5}IF(NEEDC(3).GT.0)THEN\br
++\tab{7}C(3)=X(3)**2+(-2.0D0*X(1)*X(3))+X(2)**2+X(1)**2\br
++\tab{7}CJAC(3,1)=(-2.0D0*X(3))+2.0D0*X(1)\br
++\tab{7}CJAC(3,2)=2.0D0*X(2)\br
++\tab{7}CJAC(3,3)=2.0D0*X(3)+(-2.0D0*X(1))\br
++\tab{7}CJAC(3,4)=0.0D0\br
++\tab{7}CJAC(3,5)=0.0D0\br
++\tab{7}CJAC(3,6)=0.0D0\br
++\tab{5}ENDIF\br
++\tab{5}RETURN\br
++\tab{5}END

```

Asp55(name): Exports == Implementation where

```

name : Symbol

FST  ==> FortranScalarType
FT   ==> FortranType
FSTU ==> Union(fst:FST,void:"void")
SYMTAB ==> SymbolTable
FC    ==> FortranCode
RSFC  ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
FRAC  ==> Fraction
POLY  ==> Polynomial
EXPR  ==> Expression
INT   ==> Integer
S     ==> Symbol
FLOAT ==> Float
VEC   ==> Vector
VF2   ==> VectorFunctions2
MAT   ==> Matrix
MFLOAT ==> MachineFloat
FEXPR ==> FortranExpression([],['X'],MFLOAT)
MF2   ==> MatrixCategoryFunctions2(FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR,
                                   EXPR MFLOAT,VEC EXPR MFLOAT,VEC EXPR MFLOAT,MAT EXPR MFLOAT)
SWU   ==> Union(I:Expression Integer,F:Expression Float,
               CF:Expression Complex Float,switch:Switch)

Exports ==> FortranVectorFunctionCategory with
  coerce : VEC FEXPR -> $
    ++coerce(f) takes objects from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns them into an ASP.

Implementation ==> add

real : FSTU := ["real":FST]$FSTU
integer : FSTU := ["integer":FST]$FSTU
syms : SYMTAB := empty()$SYMTAB
declare!(MODE,fortranInteger(),syms)$SYMTAB
declare!(NCNLN,fortranInteger(),syms)$SYMTAB
declare!(N,fortranInteger(),syms)$SYMTAB
declare!(NROWJ,fortranInteger(),syms)$SYMTAB
needcType : FT := construct(integer,[NCNLN::Symbol],false)$FT
declare!(NEEDC,needcType,syms)$SYMTAB
xType : FT := construct(real,[N::Symbol],false)$FT
declare!(X,xType,syms)$SYMTAB
cType : FT := construct(real,[NCNLN::Symbol],false)$FT
declare!(C,cType,syms)$SYMTAB
cjacType : FT := construct(real,[NROWJ::Symbol,N::Symbol],false)$FT
declare!(CJAC,cjacType,syms)$SYMTAB
declare!(NSTATE,fortranInteger(),syms)$SYMTAB
iuType : FT := construct(integer,["*":Symbol],false)$FT
declare!(IUSER,iuType,syms)$SYMTAB
uType : FT := construct(real,["*":Symbol],false)$FT

```

```

declare! (USER, uType, syms) $SYMTAB
Rep := FortranProgram(name, ["void"] $FSTU,
    [MODE, NCNLN, N, NROWJ, NEEDC, X, C, CJAC, NSTATE, IUSER, USER], syms)

-- Take a symbol, pull of the script and turn it into an integer!!
o2int(u:S):Integer ==
    o : OutputForm := first elt(scripts(u)$S, sub)
    o pretend Integer

localAssign(s:Symbol, dim:List POLY INT, u:FEXPR):FC ==
    assign(s, dim, (u::EXPR MFLOAT)$FEXPR)$FC

makeCond(index:INT, fun:FEXPR, jac:VEC FEXPR):FC ==
    needc : EXPR INT := (subscript(NEEDC, [index::OutputForm])$S)::EXPR(INT)
    sw : Switch := GT([needc]$SWU, [0::EXPR(INT)]$SWU)$Switch
    ass : List FC := [localAssign(CJAC, [index::POLY INT, i::POLY INT], jac.i)_
        for i in 1..maxIndex(jac)]
    cond(sw, block([localAssign(C, [index::POLY INT], fun), :ass])$FC)$FC

coerce(u:VEC FEXPR):$ ==
    ncnln:Integer := maxIndex(u)
    x:S := X::S
    pu:List(S) := []
    -- Work out which variables appear in the expressions
    for e in entries(u) repeat
        pu := setUnion(pu, variables(e)$FEXPR)
    scriptList : List Integer := map(o2int, pu)$ListFunctions2(S, Integer)
    -- This should be the maximum X_n which occurs (there may be others
    -- which don't):
    n:Integer := reduce(max, scriptList)$List(Integer)
    p:List(S) := []
    for j in 1..n repeat p:= cons(subscript(x, [j::OutputForm])$S, p)
    p:= reverse(p)
    jac:MAT FEXPR := _
        jacobian(u, p)$MultiVariableCalculusFunctions(S, FEXPR, VEC FEXPR, List(S))
    code : List FC := [makeCond(j, u.j, row(jac, j)) for j in 1..ncnln]
    [:code, returns()$FC]:::$

coerce(c:List FC):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

coerce(c:FC):$ == coerce(c)$Rep

coerce(u:$):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
    p := checkPrecision()$NAGLinkSupportPackage
    outputAsFortran(u)$Rep
    p => restorePrecision()$NAGLinkSupportPackage

```

```

retract(u:VEC FRAC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC FRAC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC EXPR INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR INT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC EXPR FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY FLOAT,FEXPR)
  v::$

```



```

retractIfCan(u:VEC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

```

— ASP55.dotabb —

```

"ASP55" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP55"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ASP55" -> "ALIST"

```

domain ASP6 Asp6

— Asp6.input —

```

)set break resume
)sys rm -f Asp6.output
)spool Asp6.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp6
--R
--R Asp6(name: Symbol) is a domain constructor
--R Abbreviation for Asp6 is ASP6
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP6
--R
--R----- Operations -----
--R coerce : FortranCode -> %               coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm               outputAsFortran : % -> Void
--R coerce : Vector(FortranExpression([construct],[construct,QUOTEX],MachineFloat)) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Vector(Fraction(Polynomial(Integer))) -> %
--R retract : Vector(Fraction(Polynomial(Float))) -> %
--R retract : Vector(Polynomial(Integer)) -> %
--R retract : Vector(Polynomial(Float)) -> %
--R retract : Vector(Expression(Integer)) -> %

```

```

--R retract : Vector(Expression(Float)) -> %
--R retractIfCan : Vector(Fraction(Polynomial(Integer))) -> Union(%, "failed")
--R retractIfCan : Vector(Fraction(Polynomial(Float))) -> Union(%, "failed")
--R retractIfCan : Vector(Polynomial(Integer)) -> Union(%, "failed")
--R retractIfCan : Vector(Polynomial(Float)) -> Union(%, "failed")
--R retractIfCan : Vector(Expression(Integer)) -> Union(%, "failed")
--R retractIfCan : Vector(Expression(Float)) -> Union(%, "failed")
--R
--E 1

```

```

)spool
)lisp (bye)

```

— Asp6.help —

=====

Asp6 examples

=====

Asp6 produces Fortran for Type 6 ASPs, needed for NAG routines
c05nbf, c05ncf. These represent vectors of functions of X(i) and look like:

```

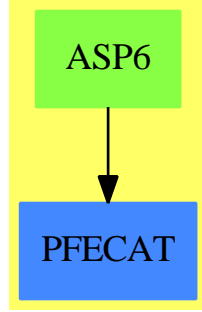
SUBROUTINE FCN(N,X,FVEC,IFLAG)
DOUBLE PRECISION X(N),FVEC(N)
INTEGER N,IFLAG
FVEC(1)=(-2.0D0*X(2))+(-2.0D0*X(1)**2)+3.0D0*X(1)+1.0D0
FVEC(2)=(-2.0D0*X(3))+(-2.0D0*X(2)**2)+3.0D0*X(2)+(-1.0D0*X(1))+1.
&0D0
FVEC(3)=(-2.0D0*X(4))+(-2.0D0*X(3)**2)+3.0D0*X(3)+(-1.0D0*X(2))+1.
&0D0
FVEC(4)=(-2.0D0*X(5))+(-2.0D0*X(4)**2)+3.0D0*X(4)+(-1.0D0*X(3))+1.
&0D0
FVEC(5)=(-2.0D0*X(6))+(-2.0D0*X(5)**2)+3.0D0*X(5)+(-1.0D0*X(4))+1.
&0D0
FVEC(6)=(-2.0D0*X(7))+(-2.0D0*X(6)**2)+3.0D0*X(6)+(-1.0D0*X(5))+1.
&0D0
FVEC(7)=(-2.0D0*X(8))+(-2.0D0*X(7)**2)+3.0D0*X(7)+(-1.0D0*X(6))+1.
&0D0
FVEC(8)=(-2.0D0*X(9))+(-2.0D0*X(8)**2)+3.0D0*X(8)+(-1.0D0*X(7))+1.
&0D0
FVEC(9)=(-2.0D0*X(9)**2)+3.0D0*X(9)+(-1.0D0*X(8))+1.0D0
RETURN
END

```

See Also:

o)show Asp6

Asp6 (ASP6)



Exports:

coerce outputAsFortran retract retractIfCan

— domain ASP6 Asp6 —

```

)abbrev domain ASP6 Asp6
++ Author: Mike Dewar and Godfrey Nolan and Grant Keady
++ Date Created: Mar 1993
++ Date Last Updated: 6 October 1994
++ Description:
++ \spadtype{Asp6} produces Fortran for Type 6 ASPs, needed for NAG routines
++ c05nbf, c05ncf. These represent vectors of functions of X(i) and look like:
++
++ \tab{5}SUBROUTINE FCN(N,X,FVEC,IFLAG)
++ \tab{5}DOUBLE PRECISION X(N),FVEC(N)
++ \tab{5}INTEGER N,IFLAG
++ \tab{5}FVEC(1)=(-2.0D0*X(2))+(-2.0D0*X(1)**2)+3.0D0*X(1)+1.0D0
++ \tab{5}FVEC(2)=(-2.0D0*X(3))+(-2.0D0*X(2)**2)+3.0D0*X(2)+(-1.0D0*X(1))+1.
++ \tab{4}&0D0
++ \tab{5}FVEC(3)=(-2.0D0*X(4))+(-2.0D0*X(3)**2)+3.0D0*X(3)+(-1.0D0*X(2))+1.
++ \tab{4}&0D0
++ \tab{5}FVEC(4)=(-2.0D0*X(5))+(-2.0D0*X(4)**2)+3.0D0*X(4)+(-1.0D0*X(3))+1.
++ \tab{4}&0D0
++ \tab{5}FVEC(5)=(-2.0D0*X(6))+(-2.0D0*X(5)**2)+3.0D0*X(5)+(-1.0D0*X(4))+1.
++ \tab{4}&0D0
++ \tab{5}FVEC(6)=(-2.0D0*X(7))+(-2.0D0*X(6)**2)+3.0D0*X(6)+(-1.0D0*X(5))+1.
++ \tab{4}&0D0
++ \tab{5}FVEC(7)=(-2.0D0*X(8))+(-2.0D0*X(7)**2)+3.0D0*X(7)+(-1.0D0*X(6))+1.
++ \tab{4}&0D0
++ \tab{5}FVEC(8)=(-2.0D0*X(9))+(-2.0D0*X(8)**2)+3.0D0*X(8)+(-1.0D0*X(7))+1.
++ \tab{4}&0D0
  
```

```

++ \tab{5}FVEC(9)=(-2.0D0*X(9)**2)+3.0D0*X(9)+(-1.0D0*X(8))+1.0D0
++ \tab{5}RETURN
++ \tab{5}END

```

```

Asp6(name): Exports == Implementation where
  name : Symbol

```

```

FEXPR ==> FortranExpression([],['X'],MFLOAT)
MFLOAT ==> MachineFloat
FST ==> FortranScalarType
FT ==> FortranType
SYMTAB ==> SymbolTable
RSFC ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
UFST ==> Union(fst:FST,void:"void")
FRAC ==> Fraction
POLY ==> Polynomial
EXPR ==> Expression
INT ==> Integer
FLOAT ==> Float
VEC ==> Vector
VF2 ==> VectorFunctions2

```

```

Exports == FortranVectorFunctionCategory with
  coerce: Vector FEXPR -> %
    ++coerce(f) takes objects from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns them into an ASP.

```

```

Implementation == add

```

```

real : UFST := ["real":FST]$UFST
syms : SYMTAB := empty()$SYMTAB
declare!(N,fortranInteger()$FT,syms)$SYMTAB
xType : FT := construct(real,[N],false)$FT
declare!(X,xType,syms)$SYMTAB
declare!(FVEC,xType,syms)$SYMTAB
declare!(IFLAG,fortranInteger()$FT,syms)$SYMTAB
Rep := FortranProgram(name,["void"]$Union(fst:FST,void:"void"),
                        [N,X,FVEC,IFLAG],syms)

retract(u:VEC FRAC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):$

retract(u:VEC FRAC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VectorFunctions2(FRAC POLY FLOAT,FEXPR)

```

```

v::$

retractIfCan(u:VEC FRAC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=_
    map(retractIfCan,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

retract(u:VEC EXPR INT):$ ==
  v : VEC FEXPR := map(retract,u)$VectorFunctions2(EXPR INT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

retract(u:VEC EXPR FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VectorFunctions2(EXPR FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

retract(u:VEC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VectorFunctions2(POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

retract(u:VEC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VectorFunctions2(POLY FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

fexpr2expr(u:FEXPR):EXPR MFLOAT ==
  (u::EXPR MFLOAT)$FEXPR

coerce(u:VEC FEXPR):% ==
  v : VEC EXPR MFLOAT
  v := map(fexpr2expr,u)$VF2(FEXPR,EXPR MFLOAT)

```

```

([assign(FVEC,v)$FortranCode,returns()$FortranCode]$List(FortranCode)):$
coerce(c:List FortranCode):% == coerce(c)$Rep
coerce(r:RSFC):% == coerce(r)$Rep
coerce(c:FortranCode):% == coerce(c)$Rep
coerce(u:%):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

```

— ASP6.dotabb —

```

"ASP6" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP6"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"ASP6" -> "PFECAT"

```

domain ASP7 Asp7

— Asp7.input —

```

)set break resume
)sys rm -f Asp7.output
)spool Asp7.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp7
--R
--R Asp7(name: Symbol) is a domain constructor
--R Abbreviation for Asp7 is ASP7
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP7
--R

```

```

--R----- Operations -----
--R coerce : FortranCode -> %               coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm               outputAsFortran : % -> Void
--R coerce : Vector(FortranExpression([construct,QUOTEX],[construct,QUOTEY],MachineFloat)) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Vector(Fraction(Polynomial(Integer))) -> %
--R retract : Vector(Fraction(Polynomial(Float))) -> %
--R retract : Vector(Polynomial(Integer)) -> %
--R retract : Vector(Polynomial(Float)) -> %
--R retract : Vector(Expression(Integer)) -> %
--R retract : Vector(Expression(Float)) -> %
--R retractIfCan : Vector(Fraction(Polynomial(Integer))) -> Union(%, "failed")
--R retractIfCan : Vector(Fraction(Polynomial(Float))) -> Union(%, "failed")
--R retractIfCan : Vector(Polynomial(Integer)) -> Union(%, "failed")
--R retractIfCan : Vector(Polynomial(Float)) -> Union(%, "failed")
--R retractIfCan : Vector(Expression(Integer)) -> Union(%, "failed")
--R retractIfCan : Vector(Expression(Float)) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)

```

— Asp7.help —

=====

Asp7 examples

=====

Asp7 produces Fortran for Type 7 ASPs, needed for NAG routines d02bbf, d02gaf. These represent a vector of functions of the scalar X and the array Z, and look like:

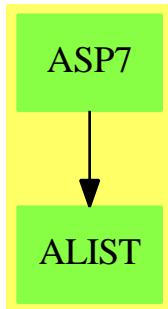
```

SUBROUTINE FCN(X,Z,F)
DOUBLE PRECISION F(*),X,Z(*)
F(1)=DTAN(Z(3))
F(2)=((-0.03199999999999999D0*DCOS(Z(3))*DTAN(Z(3)))+(-0.02D0*Z(2)
&**2))/(Z(2)*DCOS(Z(3)))
F(3)=-0.03199999999999999D0/(X*Z(2)**2)
RETURN
END

```

See Also:
o)show Asp7

Asp7 (ASP7)

**Exports:**

coerce outputAsFortran retract retractIfCan

— domain ASP7 Asp7 —

```

)abbrev domain ASP7 Asp7
++ Author: Mike Dewar and Godfrey Nolan and Grant Keady
++ Date Created: Mar 1993
++ Date Last Updated: 6 October 1994
++ Description:
++ \spadtype{Asp7} produces Fortran for Type 7 ASPs, needed for NAG routines
++ d02bbf, d02gaf. These represent a vector of functions of the scalar X and
++ the array Z, and look like:
++
++ \tab{5}SUBROUTINE FCN(X,Z,F)\br
++ \tab{5}DOUBLE PRECISION F(*),X,Z(*)\br
++ \tab{5}F(1)=DTAN(Z(3))\br
++ \tab{5}F(2)=((-0.0319999999999999D0*DCOS(Z(3))*DTAN(Z(3)))+(-0.02D0*Z(2)\br
++ \tab{4}&**2))/(Z(2)*DCOS(Z(3)))\br
++ \tab{5}F(3)=-0.0319999999999999D0/(X*Z(2)**2)\br
++ \tab{5}RETURN\br
++ \tab{5}END
  
```

```

Asp7(name): Exports == Implementation where
name : Symbol
  
```

```

FST    ==> FortranScalarType
FT     ==> FortranType
SYMTAB ==> SymbolTable
RSFC   ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
MFLOAT ==> MachineFloat
FEXPR  ==> FortranExpression(['X'],['Y'],MFLOAT)
UFST   ==> Union(fst:FST,void:"void")
FRAC   ==> Fraction
POLY   ==> Polynomial
  
```



```

EXPR ==> Expression
INT ==> Integer
FLOAT ==> Float
VEC ==> Vector
VF2 ==> VectorFunctions2

Exports ==> FortranVectorFunctionCategory with
  coerce : Vector FEXPR -> %
    ++coerce(f) takes objects from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns them into an ASP.

Implementation ==> add

real : UFST := ["real"::FST]$UFST
syms : SYMTAB := empty()$SYMTAB
declare!(X,fortranReal(),syms)$SYMTAB
yType : FT := construct(real,["* "::Symbol],false)$FT
declare!(Y,yType,syms)$SYMTAB
declare!(F,yType,syms)$SYMTAB
Rep := FortranProgram(name,["void"]$UFST,[X,Y,F],syms)

retract(u:VEC FRAC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC FRAC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC EXPR INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR INT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC EXPR FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR FLOAT,FEXPR)

```

```

v::$(

retractIfCan(u:VEC EXPR FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$(

retract(u:VEC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY INT,FEXPR)
  v::$(

retractIfCan(u:VEC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$(

retract(u:VEC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY FLOAT,FEXPR)
  v::$(

retractIfCan(u:VEC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$(

fexpr2expr(u:FEXPR):EXPR MFLOAT ==
  (u::EXPR MFLOAT)$FEXPR

coerce(u:Vector FEXPR ):% ==
  v : Vector EXPR MFLOAT
  v:=map(fexpr2expr,u)$VF2(FEXPR,EXPR MFLOAT)
  ([assign(F,v)$FortranCode,returns()$FortranCode]$List(FortranCode)):%

coerce(c:List FortranCode):% == coerce(c)$Rep

coerce(r:RSFC):% == coerce(r)$Rep

coerce(c:FortranCode):% == coerce(c)$Rep

coerce(u:%):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

```

— ASP7.dotabb —

```
"ASP7" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP7"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ASP7" -> "ALIST"
```

— —

domain ASP73 Asp73

— Asp73.input —

```
)set break resume
)sys rm -f Asp73.output
)spool Asp73.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp73
--R
--R Asp73(name: Symbol) is a domain constructor
--R Abbreviation for Asp73 is ASP73
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP73
--R
--R----- Operations -----
--R coerce : FortranCode -> %               coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm               outputAsFortran : % -> Void
--R coerce : Vector(FortranExpression([construct,QUOTEX,QUOTEY],[construct],MachineFloat)) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Vector(Fraction(Polynomial(Integer))) -> %
--R retract : Vector(Fraction(Polynomial(Float))) -> %
--R retract : Vector(Polynomial(Integer)) -> %
--R retract : Vector(Polynomial(Float)) -> %
--R retract : Vector(Expression(Integer)) -> %
--R retract : Vector(Expression(Float)) -> %
--R retractIfCan : Vector(Fraction(Polynomial(Integer))) -> Union(%,"failed")
--R retractIfCan : Vector(Fraction(Polynomial(Float))) -> Union(%,"failed")
--R retractIfCan : Vector(Polynomial(Integer)) -> Union(%,"failed")
--R retractIfCan : Vector(Polynomial(Float)) -> Union(%,"failed")
--R retractIfCan : Vector(Expression(Integer)) -> Union(%,"failed")
--R retractIfCan : Vector(Expression(Float)) -> Union(%,"failed")
--R
--E 1
```

```
)spool
)lisp (bye)
```

— Asp73.help —

=====

Asp73 examples

=====

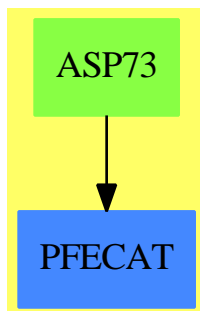
Asp73 produces Fortran for Type 73 ASPs, needed for NAG routine
d03eef, for example:

```
SUBROUTINE PDEF(X,Y,ALPHA,BETA,GAMMA,DELTA,EPSOLN,PHI,PSI)
DOUBLE PRECISION ALPHA, EPSOLN, PHI, X, Y, BETA, DELTA, GAMMA, PSI
ALPHA=DSIN(X)
BETA=Y
GAMMA=X*Y
DELTA=DCOS(X)*DSIN(Y)
EPSOLN=Y+X
PHI=X
PSI=Y
RETURN
END
```

See Also:

- o)show Asp73

Asp73 (ASP73)



Exports:

```
coerce outputAsFortran retract retractIfCan
```

— domain ASP73 Asp73 —

```
)abbrev domain ASP73 Asp73
++ Author: Mike Dewar, Grant Keady and Godfrey Nolan
++ Date Created: Mar 1993
++ Date Last Updated: 30 March 1994, 6 October 1994
++ Description:
++ \spadtype{Asp73} produces Fortran for Type 73 ASPs, needed for NAG routine
++ d03eef, for example:
++
++ \tab{5}SUBROUTINE PDEF(X,Y,ALPHA,BETA,GAMMA,DELTA,EPSOLN,PHI,PSI)\br
++ \tab{5}DOUBLE PRECISION ALPHA,EPSOLN,PHI,X,Y,BETA,DELTA,GAMMA,PSI\br
++ \tab{5}ALPHA=DSIN(X)\br
++ \tab{5}BETA=Y\br
++ \tab{5}GAMMA=X*Y\br
++ \tab{5}DELTA=DCOS(X)*DSIN(Y)\br
++ \tab{5}EPSOLN=Y+X\br
++ \tab{5}PHI=X\br
++ \tab{5}PSI=Y\br
++ \tab{5}RETURN\br
++ \tab{5}END
```

```
Asp73(name): Exports == Implementation where
  name : Symbol
```

```
FST    ==> FortranScalarType
FSTU   ==> Union(fst:FST,void:"void")
FEXPR  ==> FortranExpression(['X','Y'],[],MachineFloat)
FT     ==> FortranType
SYMTAB ==> SymbolTable
RSFC   ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
FRAC   ==> Fraction
POLY   ==> Polynomial
EXPR   ==> Expression
INT    ==> Integer
FLOAT  ==> Float
VEC    ==> Vector
VF2    ==> VectorFunctions2
```

```
Exports ==> FortranVectorFunctionCategory with
  coerce : VEC FEXPR -> $
    ++coerce(f) takes objects from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns them into an ASP.
```

```
Implementation ==> add
```

```

syms : SYMTAB := empty()$SYMTAB
declare!(X,fortranReal(),syms) $SYMTAB
declare!(Y,fortranReal(),syms) $SYMTAB
declare!(ALPHA,fortranReal(),syms)$SYMTAB
declare!(BETA,fortranReal(),syms) $SYMTAB
declare!(GAMMA,fortranReal(),syms) $SYMTAB
declare!(DELTA,fortranReal(),syms) $SYMTAB
declare!(EPSOLN,fortranReal(),syms) $SYMTAB
declare!(PHI,fortranReal(),syms) $SYMTAB
declare!(PSI,fortranReal(),syms) $SYMTAB
Rep := FortranProgram(name,["void"]$FSTU,
                        [X,Y,ALPHA,BETA,GAMMA,DELTA,EPSOLN,PHI,PSI],syms)

-- To help the poor compiler!
localAssign(u:Symbol,v:FEXPR):FortranCode ==
  assign(u,(v::EXPR MachineFloat)$FEXPR)$FortranCode

coerce(u:VEC FEXPR):$ ==
  maxIndex(u) ^= 7 => error "Vector is not of dimension 7"
  [localAssign(ALPHA@Symbol,elt(u,1)),_
   localAssign(BETA@Symbol,elt(u,2)),_
   localAssign(GAMMA@Symbol,elt(u,3)),_
   localAssign(DELTA@Symbol,elt(u,4)),_
   localAssign(EPSOLN@Symbol,elt(u,5)),_
   localAssign(PHI@Symbol,elt(u,6)),_
   localAssign(PSI@Symbol,elt(u,7)),_
   returns()$FortranCode]$List(FortranCode):: $

coerce(c:FortranCode):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

coerce(c:List FortranCode):$ == coerce(c)$Rep

coerce(u:$):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

retract(u:VEC FRAC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

```

```

retract(u:VEC FRAC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC EXPR INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR INT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC EXPR FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

```

```
"ASP73" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP73"]
"PFECA7" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECA7"]
"ASP73" -> "PFECA7"
```

domain ASP74 Asp74

— Asp74.input —

```
)set break resume
)sys rm -f Asp74.output
)spool Asp74.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp74
--R
--R Asp74(name: Symbol) is a domain constructor
--R Abbreviation for Asp74 is ASP74
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP74
--R
--R----- Operations -----
--R coerce : FortranCode -> %               coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm               outputAsFortran : % -> Void
--R coerce : Matrix(FortranExpression([construct,QUOTEX,QUOTEY],[construct],MachineFloat)) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Matrix(Fraction(Polynomial(Integer))) -> %
--R retract : Matrix(Fraction(Polynomial(Float))) -> %
--R retract : Matrix(Polynomial(Integer)) -> %
--R retract : Matrix(Polynomial(Float)) -> %
--R retract : Matrix(Expression(Integer)) -> %
--R retract : Matrix(Expression(Float)) -> %
--R retractIfCan : Matrix(Fraction(Polynomial(Integer))) -> Union(%,"failed")
--R retractIfCan : Matrix(Fraction(Polynomial(Float))) -> Union(%,"failed")
--R retractIfCan : Matrix(Polynomial(Integer)) -> Union(%,"failed")
--R retractIfCan : Matrix(Polynomial(Float)) -> Union(%,"failed")
--R retractIfCan : Matrix(Expression(Integer)) -> Union(%,"failed")
--R retractIfCan : Matrix(Expression(Float)) -> Union(%,"failed")
--R
--E 1

)spool
```


)lisp (bye)

— Asp74.help —

```
=====
Asp74 examples
=====
```

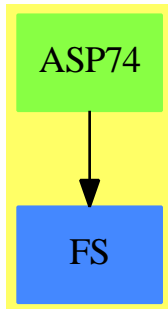
Asp74 produces Fortran for Type 74 ASPs, needed for NAG routine d03eef,
for example:

```
SUBROUTINE BNDY(X,Y,A,B,C,IBND)
DOUBLE PRECISION A,B,C,X,Y
INTEGER IBND
IF(IBND.EQ.0)THEN
A=0.0D0
B=1.0D0
C=-1.0D0*DSIN(X)
ELSEIF(IBND.EQ.1)THEN
A=1.0D0
B=0.0D0
C=DSIN(X)*DSIN(Y)
ELSEIF(IBND.EQ.2)THEN
A=1.0D0
B=0.0D0
C=DSIN(X)*DSIN(Y)
ELSEIF(IBND.EQ.3)THEN
A=0.0D0
B=1.0D0
C=-1.0D0*DSIN(Y)
ENDIF
END
```

See Also:

o)show Asp74

Asp74 (ASP74)

**Exports:**

coerce outputAsFortran retract retractIfCan

— domain ASP74 Asp74 —

```

)abbrev domain ASP74 Asp74
++ Author: Mike Dewar and Godfrey Nolan
++ Date Created: Oct 1993
++ Date Last Updated: 6 October 1994
++ Description:
++ \spadtype{Asp74} produces Fortran for Type 74 ASPs, needed for NAG routine
++ d03eef, for example:
++
++ \tab{5} SUBROUTINE BNDY(X,Y,A,B,C,IBND)\br
++ \tab{5} DOUBLE PRECISION A,B,C,X,Y\br
++ \tab{5} INTEGER IBND\br
++ \tab{5} IF(IBND.EQ.0)THEN\br
++ \tab{7} A=0.0D0\br
++ \tab{7} B=1.0D0\br
++ \tab{7} C=-1.0D0*DSIN(X)\br
++ \tab{5} ELSEIF (IBND.EQ.1)THEN\br
++ \tab{7} A=1.0D0\br
++ \tab{7} B=0.0D0\br
++ \tab{7} C=DSIN(X)*DSIN(Y)\br
++ \tab{5} ELSEIF (IBND.EQ.2)THEN\br
++ \tab{7} A=1.0D0\br
++ \tab{7} B=0.0D0\br
++ \tab{7} C=DSIN(X)*DSIN(Y)\br
++ \tab{5} ELSEIF (IBND.EQ.3)THEN\br
++ \tab{7} A=0.0D0\br
++ \tab{7} B=1.0D0\br
++ \tab{7} C=-1.0D0*DSIN(Y)\br
++ \tab{5} ENDIF\br
++ \tab{5} END
  
```

```

Asp74(name): Exports == Implementation where
  name : Symbol

  FST    ==> FortranScalarType
  FSTU   ==> Union(fst:FST,void:"void")
  FT     ==> FortranType
  SYMTAB ==> SymbolTable
  FC     ==> FortranCode
  PI     ==> PositiveInteger
  RSFC   ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
  FRAC   ==> Fraction
  POLY   ==> Polynomial
  EXPR   ==> Expression
  INT    ==> Integer
  FLOAT  ==> Float
  MFLOAT ==> MachineFloat
  FEXPR  ==> FortranExpression(['X','Y'],[],MFLOAT)
  U      ==> Union(I: Expression Integer,F: Expression Float,_
                  CF: Expression Complex Float,switch:Switch)
  VEC    ==> Vector
  MAT    ==> Matrix
  M2     ==> MatrixCategoryFunctions2
  MF2a   ==> M2(FRAC POLY INT,VEC FRAC POLY INT,VEC FRAC POLY INT,
               MAT FRAC POLY INT, FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
  MF2b   ==> M2(FRAC POLY FLOAT,VEC FRAC POLY FLOAT,VEC FRAC POLY FLOAT,
               MAT FRAC POLY FLOAT, FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
  MF2c   ==> M2(POLY INT,VEC POLY INT,VEC POLY INT,MAT POLY INT,
               FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
  MF2d   ==> M2(POLY FLOAT,VEC POLY FLOAT,VEC POLY FLOAT,
               MAT POLY FLOAT, FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
  MF2e   ==> M2(EXPR INT,VEC EXPR INT,VEC EXPR INT,MAT EXPR INT,
               FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
  MF2f   ==> M2(EXPR FLOAT,VEC EXPR FLOAT,VEC EXPR FLOAT,
               MAT EXPR FLOAT, FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)

Exports ==> FortranMatrixFunctionCategory with
  coerce : MAT FEXPR -> $
    ++coerce(f) takes objects from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns them into an ASP.

Implementation ==> add

syms : SYMTAB := empty()$SYMTAB
declare!(X,fortranReal(),syms)$SYMTAB
declare!(Y,fortranReal(),syms)$SYMTAB
declare!(A,fortranReal(),syms)$SYMTAB
declare!(B,fortranReal(),syms)$SYMTAB
declare!(C,fortranReal(),syms)$SYMTAB
declare!(IBND,fortranInteger(),syms)$SYMTAB
Rep := FortranProgram(name,["void"]$FSTU,[X,Y,A,B,C,IBND],syms)

```

```

-- To help the poor compiler!
localAssign(u:Symbol,v:FEXPR):FC == assign(u,(v::EXPR MFLOAT)$FEXPR)$FC

coerce(u:MAT FEXPR):$ ==
  (nrows(u) ^= 4 or ncols(u) ^= 3) => error "Not a 4X3 matrix"
  flag:U := [IBND@Symbol::EXPR INT]$U
  pt0:U := [0::EXPR INT]$U
  pt1:U := [1::EXPR INT]$U
  pt2:U := [2::EXPR INT]$U
  pt3:U := [3::EXPR INT]$U
  sw1: Switch := EQ(flag,pt0)$Switch
  sw2: Switch := EQ(flag,pt1)$Switch
  sw3: Switch := EQ(flag,pt2)$Switch
  sw4: Switch := EQ(flag,pt3)$Switch
  a11 : FC := localAssign(A,u(1,1))
  a12 : FC := localAssign(B,u(1,2))
  a13 : FC := localAssign(C,u(1,3))
  a21 : FC := localAssign(A,u(2,1))
  a22 : FC := localAssign(B,u(2,2))
  a23 : FC := localAssign(C,u(2,3))
  a31 : FC := localAssign(A,u(3,1))
  a32 : FC := localAssign(B,u(3,2))
  a33 : FC := localAssign(C,u(3,3))
  a41 : FC := localAssign(A,u(4,1))
  a42 : FC := localAssign(B,u(4,2))
  a43 : FC := localAssign(C,u(4,3))
  c : FC := cond(sw1,block([a11,a12,a13])$FC,
    cond(sw2,block([a21,a22,a23])$FC,
      cond(sw3,block([a31,a32,a33])$FC,
        cond(sw4,block([a41,a42,a43])$FC)$FC)$FC)$FC)$FC
  c::$

coerce(u:$):OutputForm == coerce(u)$Rep

coerce(c:FortranCode):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

coerce(c:List FortranCode):$ == coerce(c)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

retract(u:MAT FRAC POLY INT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2a
  v::$

```

```

retractIfCan(u:MAT FRAC POLY INT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2a
  v case "failed" => "failed"
  (v::MAT FEXPR)::$

retract(u:MAT FRAC POLY FLOAT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2b
  v::$

retractIfCan(u:MAT FRAC POLY FLOAT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2b
  v case "failed" => "failed"
  (v::MAT FEXPR)::$

retract(u:MAT EXPR INT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2e
  v::$

retractIfCan(u:MAT EXPR INT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2e
  v case "failed" => "failed"
  (v::MAT FEXPR)::$

retract(u:MAT EXPR FLOAT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2f
  v::$

retractIfCan(u:MAT EXPR FLOAT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2f
  v case "failed" => "failed"
  (v::MAT FEXPR)::$

retract(u:MAT POLY INT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2c
  v::$

retractIfCan(u:MAT POLY INT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2c
  v case "failed" => "failed"
  (v::MAT FEXPR)::$

retract(u:MAT POLY FLOAT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2d
  v::$

retractIfCan(u:MAT POLY FLOAT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2d
  v case "failed" => "failed"
  (v::MAT FEXPR)::$

```

— ASP74.dotabb —

```
"ASP74" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP74"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ASP74" -> "FS"
```

domain ASP77 Asp77

— Asp77.input —

```
)set break resume
)sys rm -f Asp77.output
)spool Asp77.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp77
--R
--R Asp77(name: Symbol) is a domain constructor
--R Abbreviation for Asp77 is ASP77
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP77
--R
--R----- Operations -----
--R coerce : FortranCode -> %               coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm               outputAsFortran : % -> Void
--R coerce : Matrix(FortranExpression([construct,QUOTEX],[construct],MachineFloat)) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Matrix(Fraction(Polynomial(Integer))) -> %
--R retract : Matrix(Fraction(Polynomial(Float))) -> %
--R retract : Matrix(Polynomial(Integer)) -> %
--R retract : Matrix(Polynomial(Float)) -> %
--R retract : Matrix(Expression(Integer)) -> %
--R retract : Matrix(Expression(Float)) -> %
--R retractIfCan : Matrix(Fraction(Polynomial(Integer))) -> Union(%, "failed")
--R retractIfCan : Matrix(Fraction(Polynomial(Float))) -> Union(%, "failed")
--R retractIfCan : Matrix(Polynomial(Integer)) -> Union(%, "failed")
--R retractIfCan : Matrix(Polynomial(Float)) -> Union(%, "failed")
--R retractIfCan : Matrix(Expression(Integer)) -> Union(%, "failed")
```

```

--R retractIfCan : Matrix(Expression(Float)) -> Union(%,"failed")
--R
--E 1

)spool
)lisp (bye)

```

— Asp77.help —

=====

Asp77 examples

=====

Asp77 produces Fortran for Type 77 ASPs, needed for NAG routine d02gbf, for example:

```

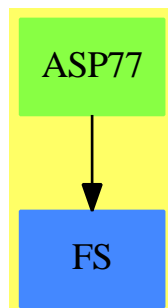
SUBROUTINE FCNF(X,F)
DOUBLE PRECISION X
DOUBLE PRECISION F(2,2)
F(1,1)=0.0D0
F(1,2)=1.0D0
F(2,1)=0.0D0
F(2,2)=-10.0D0
RETURN
END

```

See Also:

- o)show Asp77

Asp77 (ASP77)



Exports:

coerce outputAsFortran retract retractIfCan

— domain ASP77 Asp77 —

```
)abbrev domain ASP77 Asp77
++ Author: Mike Dewar, Grant Keady and Godfrey Nolan
++ Date Created: Mar 1993
++ Date Last Updated: 6 October 1994
++ Description:
++ \spadtype{Asp77} produces Fortran for Type 77 ASPs, needed for NAG routine
++ d02gbf, for example:
++
++ \tab{5}SUBROUTINE FCNF(X,F)\br
++ \tab{5}DOUBLE PRECISION X\br
++ \tab{5}DOUBLE PRECISION F(2,2)\br
++ \tab{5}F(1,1)=0.0D0\br
++ \tab{5}F(1,2)=1.0D0\br
++ \tab{5}F(2,1)=0.0D0\br
++ \tab{5}F(2,2)=-10.0D0\br
++ \tab{5}RETURN\br
++ \tab{5}END
```

Asp77(name): Exports == Implementation where
name : Symbol

```
FST    ==> FortranScalarType
FSTU   ==> Union(fst:FST,void:"void")
FT     ==> FortranType
FC     ==> FortranCode
SYMTAB ==> SymbolTable
RSFC   ==> Record(localSymbols:SymbolTable,code:List(FC))
FRAC   ==> Fraction
POLY   ==> Polynomial
EXPR   ==> Expression
INT    ==> Integer
FLOAT  ==> Float
MFLOAT ==> MachineFloat
FEXPR  ==> FortranExpression(['X'],[],MFLOAT)
VEC    ==> Vector
MAT    ==> Matrix
M2     ==> MatrixCategoryFunctions2
MF2    ==> M2(FEXPR,VEC FEXPR,VEC FEXPR,Matrix FEXPR,EXPR MFLOAT,
             VEC EXPR MFLOAT,VEC EXPR MFLOAT,Matrix EXPR MFLOAT)
MF2a   ==> M2(FRAC POLY INT,VEC FRAC POLY INT,VEC FRAC POLY INT,
             MAT FRAC POLY INT, FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
MF2b   ==> M2(FRAC POLY FLOAT,VEC FRAC POLY FLOAT,VEC FRAC POLY FLOAT,
             MAT FRAC POLY FLOAT, FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
MF2c   ==> M2(POLY INT,VEC POLY INT,VEC POLY INT,MAT POLY INT,
```



```

FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
MF2d ==> M2(POLY FLOAT,VEC POLY FLOAT,VEC POLY FLOAT,
            MAT POLY FLOAT, FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
MF2e ==> M2(EXPR INT,VEC EXPR INT,VEC EXPR INT,MAT EXPR INT,
            FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
MF2f ==> M2(EXPR FLOAT,VEC EXPR FLOAT,VEC EXPR FLOAT,
            MAT EXPR FLOAT, FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)

Exports ==> FortranMatrixFunctionCategory with
  coerce : MAT FEXPR -> $
    ++coerce(f) takes objects from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns them into an ASP.

Implementation ==> add

real : FSTU := ["real":FST]$FSTU
syms : SYMTAB := empty()$SYMTAB
declare!(X,fortranReal(),syms)$SYMTAB
Rep := FortranProgram(name,["void"]$FSTU,[X,F],syms)

fexpr2expr(u:FEXPR):EXPR MFLOAT == coerce(u)$FEXPR

localAssign(s:Symbol,j:MAT FEXPR):FortranCode ==
  j' : MAT EXPR MFLOAT := map(fexpr2expr,j)$MF2
  assign(s,j')$FortranCode

coerce(u:MAT FEXPR):$ ==
  dimension := nrows(u)::POLY(INT)
  locals : SYMTAB := empty()
  declare!(F,[real,[dimension,dimension]$List(POLY(INT)),false]$FT,locals)
  code : List FC := [localAssign(F,u),returns()$FC]
  ([locals,code]$RSFC)::$

coerce(c:List FC):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

coerce(c:FC):$ == coerce(c)$Rep

coerce(u:$):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

retract(u:MAT FRAC POLY INT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2a
  v::$

```

```

retractIfCan(u:MAT FRAC POLY INT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2a
  v case "failed" => "failed"
  (v::MAT FEXPR)::$

retract(u:MAT FRAC POLY FLOAT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2b
  v::$

retractIfCan(u:MAT FRAC POLY FLOAT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2b
  v case "failed" => "failed"
  (v::MAT FEXPR)::$

retract(u:MAT EXPR INT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2e
  v::$

retractIfCan(u:MAT EXPR INT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2e
  v case "failed" => "failed"
  (v::MAT FEXPR)::$

retract(u:MAT EXPR FLOAT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2f
  v::$

retractIfCan(u:MAT EXPR FLOAT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2f
  v case "failed" => "failed"
  (v::MAT FEXPR)::$

retract(u:MAT POLY INT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2c
  v::$

retractIfCan(u:MAT POLY INT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2c
  v case "failed" => "failed"
  (v::MAT FEXPR)::$

retract(u:MAT POLY FLOAT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2d
  v::$

retractIfCan(u:MAT POLY FLOAT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2d
  v case "failed" => "failed"
  (v::MAT FEXPR)::$

```

— ASP77.dotabb —

```
"ASP77" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP77"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ASP77" -> "FS"
```

domain ASP78 Asp78

— Asp78.input —

```
)set break resume
)sys rm -f Asp78.output
)spool Asp78.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp78
--R
--R Asp78(name: Symbol) is a domain constructor
--R Abbreviation for Asp78 is ASP78
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP78
--R
--R----- Operations -----
--R coerce : FortranCode -> %               coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm               outputAsFortran : % -> Void
--R coerce : Vector(FortranExpression([construct,QUOTEX],[construct],MachineFloat)) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Vector(Fraction(Polynomial(Integer))) -> %
--R retract : Vector(Fraction(Polynomial(Float))) -> %
--R retract : Vector(Polynomial(Integer)) -> %
--R retract : Vector(Polynomial(Float)) -> %
--R retract : Vector(Expression(Integer)) -> %
--R retract : Vector(Expression(Float)) -> %
--R retractIfCan : Vector(Fraction(Polynomial(Integer))) -> Union(%,"failed")
--R retractIfCan : Vector(Fraction(Polynomial(Float))) -> Union(%,"failed")
--R retractIfCan : Vector(Polynomial(Integer)) -> Union(%,"failed")
```

```

--R retractIfCan : Vector(Polynomial(Float)) -> Union(%,"failed")
--R retractIfCan : Vector(Expression(Integer)) -> Union(%,"failed")
--R retractIfCan : Vector(Expression(Float)) -> Union(%,"failed")
--R
--E 1

```

```

)spool
)lisp (bye)

```

— Asp78.help —

=====

Asp78 examples

=====

Asp78 produces Fortran for Type 78 ASPs, needed for NAG routine d02gbf,
for example:

```

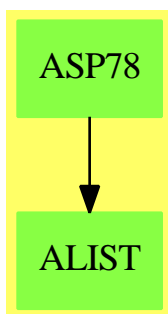
SUBROUTINE FCNG(X,G)
DOUBLE PRECISION G(*),X
G(1)=0.0D0
G(2)=0.0D0
END

```

See Also:

- o)show Asp78

Asp78 (ASP78)



Exports:

coerce outputAsFortran retract retractIfCan

— domain ASP78 Asp78 —

```

)abbrev domain ASP78 Asp78
++ Author: Mike Dewar, Grant Keady and Godfrey Nolan
++ Date Created: Mar 1993
++ Date Last Updated: 6 October 1994
++ Description:
++ \spadtype{Asp78} produces Fortran for Type 78 ASPs, needed for NAG routine
++ d02gbf, for example:
++
++ \tab{5}SUBROUTINE FCNG(X,G)\br
++ \tab{5}DOUBLE PRECISION G(*),X\br
++ \tab{5}G(1)=0.0D0\br
++ \tab{5}G(2)=0.0D0\br
++ \tab{5}END

Asp78(name): Exports == Implementation where
  name : Symbol

  FST    ==> FortranScalarType
  FSTU   ==> Union(fst:FST,void:"void")
  FT     ==> FortranType
  FC     ==> FortranCode
  SYMTAB ==> SymbolTable
  RSFC   ==> Record(localSymbols:SymbolTable,code:List(FC))
  FRAC   ==> Fraction
  POLY   ==> Polynomial
  EXPR   ==> Expression
  INT    ==> Integer
  FLOAT  ==> Float
  VEC    ==> Vector
  VF2    ==> VectorFunctions2
  MFLOAT ==> MachineFloat
  FEXPR  ==> FortranExpression(['X'],[],MFLOAT)

Exports ==> FortranVectorFunctionCategory with
  coerce : VEC FEXPR -> $
    ++coerce(f) takes objects from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns them into an ASP.

Implementation ==> add

  real : FSTU := ["real"::FST]$FSTU
  syms : SYMTAB := empty()$SYMTAB
  declare!(X,fortranReal(),syms)$SYMTAB
  gType : FT := construct(real,["* "::Symbol],false)$FT
  declare!(G,gType,syms)$SYMTAB
  Rep := FortranProgram(name,["void"]$FSTU,[X,G],syms)

```

```

fexpr2expr(u:FEXPR):EXPR MFLOAT == coerce(u)$FEXPR

coerce(u:VEC FEXPR):$ ==
  u' : VEC EXPR MFLOAT := map(fexpr2expr,u)$VF2(FEXPR,EXPR MFLOAT)
  (assign(G,u')$FC):: $

coerce(u:$):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

coerce(c:List FC):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

coerce(c:FC):$ == coerce(c)$Rep

retract(u:VEC FRAC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

retract(u:VEC FRAC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC FRAC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(FRAC POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

retract(u:VEC EXPR INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR INT,FEXPR)
  v::$

retractIfCan(u:VEC EXPR INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR):: $

retract(u:VEC EXPR FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(EXPR FLOAT,FEXPR)
  v::$

```

```

retractIfCan(u:VEC EXPR FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(EXPR FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC POLY INT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY INT,FEXPR)
  v::$

retractIfCan(u:VEC POLY INT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY INT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

retract(u:VEC POLY FLOAT):$ ==
  v : VEC FEXPR := map(retract,u)$VF2(POLY FLOAT,FEXPR)
  v::$

retractIfCan(u:VEC POLY FLOAT):Union($,"failed") ==
  v:Union(VEC FEXPR,"failed"):=map(retractIfCan,u)$VF2(POLY FLOAT,FEXPR)
  v case "failed" => "failed"
  (v::VEC FEXPR)::$

```

— ASP78.dotabb —

```

"ASP78" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP78"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ASP78" -> "ALIST"

```

domain ASP8 Asp8

— Asp8.input —

```

)set break resume
)sys rm -f Asp8.output
)spool Asp8.output
)set message test on
)set message auto off
)clear all

```

```

--S 1 of 1
)show Asp8
--R
--R Asp8(name: Symbol) is a domain constructor
--R Abbreviation for Asp8 is ASP8
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP8
--R
--R----- Operations -----
--R coerce : FortranCode -> %          coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm          outputAsFortran : % -> Void
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R coerce : Vector(MachineFloat) -> %
--R
--E 1

)spool
)lisp (bye)

```

— Asp8.help —

=====

Asp8 examples

=====

Asp8 produces Fortran for Type 8 ASPs, needed for NAG routine d02bbf.
 This ASP prints intermediate values of the computed solution of
 an ODE and might look like:

```

SUBROUTINE OUTPUT(XSOL,Y,COUNT,M,N,RESULT,FORWRD)
DOUBLE PRECISION Y(N),RESULT(M,N),XSOL
INTEGER M,N,COUNT
LOGICAL FORWRD
DOUBLE PRECISION X02ALF,POINTS(8)
EXTERNAL X02ALF
INTEGER I
POINTS(1)=1.0D0
POINTS(2)=2.0D0
POINTS(3)=3.0D0
POINTS(4)=4.0D0
POINTS(5)=5.0D0
POINTS(6)=6.0D0
POINTS(7)=7.0D0
POINTS(8)=8.0D0
COUNT=COUNT+1
DO 25001 I=1,N
RESULT(COUNT,I)=Y(I)

```



```

25001 CONTINUE
      IF (COUNT.EQ.M) THEN
      IF (FORWRD) THEN
      XSOL=X02ALF()
      ELSE
      XSOL=-X02ALF()
      ENDIF
      ELSE
      XSOL=POINTS(COUNT)
      ENDIF
      END

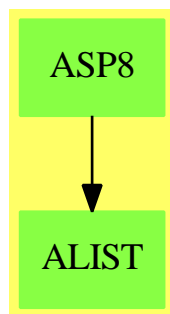
```

See Also:

o)show Asp8

—————→

Asp8 (ASP8)



Exports:

coerce outputAsFortran

— domain ASP8 Asp8 —

```

)abbrev domain ASP8 Asp8
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: 11 February 1994
++ Date Last Updated: 6 October 1994
++ Description:
++ \spadtype{Asp8} produces Fortran for Type 8 ASPs, needed for NAG routine
++ d02bbf. This ASP prints intermediate values of the computed solution of
++ an ODE and might look like:
++
++ \tab{5}SUBROUTINE OUTPUT(XSOL,Y,COUNT,M,N,RESULT,FORWRD)\br

```

```

++ \tab{5}DOUBLE PRECISION Y(N),RESULT(M,N),XSOL\br
++ \tab{5}INTEGER M,N,COUNT\br
++ \tab{5}LOGICAL FORWRD\br
++ \tab{5}DOUBLE PRECISION X02ALF,POINTS(8)\br
++ \tab{5}EXTERNAL X02ALF\br
++ \tab{5}INTEGER I\br
++ \tab{5}POINTS(1)=1.0D0\br
++ \tab{5}POINTS(2)=2.0D0\br
++ \tab{5}POINTS(3)=3.0D0\br
++ \tab{5}POINTS(4)=4.0D0\br
++ \tab{5}POINTS(5)=5.0D0\br
++ \tab{5}POINTS(6)=6.0D0\br
++ \tab{5}POINTS(7)=7.0D0\br
++ \tab{5}POINTS(8)=8.0D0\br
++ \tab{5}COUNT=COUNT+1\br
++ \tab{5}DO 25001 I=1,N\br
++ \tab{7} RESULT(COUNT,I)=Y(I)\br
++ 25001 CONTINUE\br
++ \tab{5}IF(COUNT.EQ.M)THEN\br
++ \tab{7}IF(FORWRD)THEN\br
++ \tab{9}XSOL=X02ALF()\br
++ \tab{7}ELSE\br
++ \tab{9}XSOL=-X02ALF()\br
++ \tab{7}ENDIF\br
++ \tab{5}ELSE\br
++ \tab{7} XSOL=POINTS(COUNT)\br
++ \tab{5}ENDIF\br
++ \tab{5}END

```

```

Asp8(name): Exports == Implementation where
name : Symbol

```

```

O      ==> OutputForm
S      ==> Symbol
FST    ==> FortranScalarType
UFST   ==> Union(fst:FST,void:"void")
FT     ==> FortranType
FC     ==> FortranCode
SYMTAB ==> SymbolTable
RSFC   ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
EX     ==> Expression Integer
MFLOAT ==> MachineFloat
EXPR   ==> Expression
PI     ==> Polynomial Integer
EXU    ==> Union(I: EXPR Integer,F: EXPR Float,CF: EXPR Complex Float,
                switch: Switch)

```

```

Exports ==> FortranVectorCategory

```

```

Implementation ==> add

```

```

real : UFST := ["real":FST]$UFST
syms : SYMTAB := empty()$SYMTAB
declare!([COUNT,M,N],fortranInteger(),syms)$SYMTAB
declare!(XSOL,fortranReal(),syms)$SYMTAB
yType : FT := construct(real,[N],false)$FT
declare!(Y,yType,syms)$SYMTAB
declare!(FORWRD,fortranLogical(),syms)$SYMTAB
declare!(RESULT,construct(real,[M,N],false)$FT,syms)$SYMTAB
Rep := _
  FortranProgram(name,["void"]$UFST,[XSOL,Y,COUNT,M,N,RESULT,FORWRD],syms)

coerce(c:List FC):% == coerce(c)$Rep

coerce(r:RSFC):% == coerce(r)$Rep

coerce(c:FC):% == coerce(c)$Rep

coerce(u:%):0 == coerce(u)$Rep

outputAsFortran(u:%):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

f2ex(u:MFLOAT):EXPR MFLOAT == (u::EXPR MFLOAT)$EXPR(MFLOAT)

coerce(points:Vector MFLOAT):% ==
  import PI
  import EXPR Integer
  -- Create some extra declarations
  locals : SYMTAB := empty()$SYMTAB
  nPol : PI := "N":S:PI
  iPol : PI := "I":S:PI
  countPol : PI := "COUNT":S:PI
  pointsDim : PI := max(#points,1):PI
  declare!(POINTS,[real,[pointsDim],false]$FT,locals)$SYMTAB
  declare!(X02ALF,[real,[],true]$FT,locals)$SYMTAB
  -- Now build up the code fragments
  index : SegmentBinding PI := equation(I@S,1::PI..nPol)$SegmentBinding(PI)
  ySym : EX := (subscript("Y":S,[I:0])$S)::EX
  loop := forLoop(index,assign(RESULT,[countPol,iPol],ySym)$FC)$FC
  v:Vector EXPR MFLOAT
  v := map(f2ex,points)$VectorFunctions2(MFLOAT,EXPR MFLOAT)
  assign1 : FC := assign(POINTS,v)$FC
  countExp: EX := COUNT@S:EX
  newValue: EX := 1 + countExp
  assign2 : FC := assign(COUNT,newValue)$FC
  newSymbol : S := subscript(POINTS,[COUNT]@List(0))$S

```

```

assign3 : FC := assign(XSOL, newSymbol::EX )$FC
fphuge : EX := kernel(operator X02ALF,empty())$List(EX))
assign4 : FC := assign(XSOL, fphuge)$FC
assign5 : FC := assign(XSOL, -fphuge)$FC
innerCond : FC := cond("FORWRD"::Symbol::Switch,assign4,assign5)
mExp : EX := M@S::EX
endCase : FC := cond(EQ([countExp]$EXU,[mExp]$EXU)$Switch,innerCond,assign3)
code := [assign1, assign2, loop, endCase]$List(FC)
([locals,code]$RSFC)::%

```

— ASP8.dotabb —

```

"ASP8" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP8"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ASP8" -> "ALIST"

```

domain ASP80 Asp80

— Asp80.input —

```

)set break resume
)sys rm -f Asp80.output
)spool Asp80.output
)set message test on
)set message auto off
)clear all

```

```

--S 1 of 1
)show Asp80

```

```

--R
--R Asp80(name: Symbol) is a domain constructor
--R Abbreviation for Asp80 is ASP80
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP80
--R

```

```

--R----- Operations -----

```

```

--R coerce : FortranCode -> %               coerce : List(FortranCode) -> %
--R coerce : % -> OutputForm               outputAsFortran : % -> Void
--R coerce : Matrix(FortranExpression([construct,QUOTEXL,QUOTEXR,QUOTEELAM],[construct],MachineFloat)) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %

```

```

--R retract : Matrix(Fraction(Polynomial(Integer))) -> %
--R retract : Matrix(Fraction(Polynomial(Float))) -> %
--R retract : Matrix(Polynomial(Integer)) -> %
--R retract : Matrix(Polynomial(Float)) -> %
--R retract : Matrix(Expression(Integer)) -> %
--R retract : Matrix(Expression(Float)) -> %
--R retractIfCan : Matrix(Fraction(Polynomial(Integer))) -> Union(%, "failed")
--R retractIfCan : Matrix(Fraction(Polynomial(Float))) -> Union(%, "failed")
--R retractIfCan : Matrix(Polynomial(Integer)) -> Union(%, "failed")
--R retractIfCan : Matrix(Polynomial(Float)) -> Union(%, "failed")
--R retractIfCan : Matrix(Expression(Integer)) -> Union(%, "failed")
--R retractIfCan : Matrix(Expression(Float)) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)

```

— Asp80.help —

=====

Asp80 examples

=====

Asp80 produces Fortran for Type 80 ASPs, needed for NAG routine d02kef,
for example:

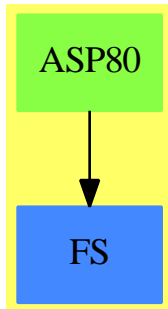
```

SUBROUTINE BDYVAL(XL,XR,ELAM,YL,YR)
DOUBLE PRECISION ELAM,XL,YL(3),XR,YR(3)
YL(1)=XL
YL(2)=2.0D0
YR(1)=1.0D0
YR(2)=-1.0D0*DSQRT(XR+(-1.0D0*ELAM))
RETURN
END

```

See Also:

o)show Asp80

Asp80 (ASP80)**Exports:**

coerce outputAsFortran retract retractIfCan

— domain ASP80 Asp80 —

```

)abbrev domain ASP80 Asp80
++ Author: Mike Dewar and Godfrey Nolan
++ Date Created: Oct 1993
++ Date Last Updated: 6 October 1994
++ Description:
++ \spadtype{Asp80} produces Fortran for Type 80 ASPs, needed for NAG routine
++ d02kef, for example:
++
++ \tab{5}SUBROUTINE BDYVAL(XL,XR,ELAM,YL,YR)\br
++ \tab{5}DOUBLE PRECISION ELAM,XL,YL(3),XR,YR(3)\br
++ \tab{5}YL(1)=XL\br
++ \tab{5}YL(2)=2.0D0\br
++ \tab{5}YR(1)=1.0D0\br
++ \tab{5}YR(2)=-1.0D0*DSQRT(XR+(-1.0D0*ELAM))\br
++ \tab{5}RETURN\br
++ \tab{5}END
  
```

```

Asp80(name): Exports == Implementation where
  name : Symbol
  
```

```

FST    ==> FortranScalarType
FSTU   ==> Union(fst:FST,void:"void")
FT     ==> FortranType
FC     ==> FortranCode
SYMTAB ==> SymbolTable
RSFC   ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
FRAC   ==> Fraction
POLY   ==> Polynomial
EXPR   ==> Expression
INT    ==> Integer
  
```

```

FLOAT ==> Float
MFLOAT ==> MachineFloat
FEXPR ==> FortranExpression(['XL','XR','ELAM'],[],MFLOAT)
VEC ==> Vector
MAT ==> Matrix
VF2 ==> VectorFunctions2
M2 ==> MatrixCategoryFunctions2
MF2a ==> M2(FRAC POLY INT,VEC FRAC POLY INT,VEC FRAC POLY INT,
            MAT FRAC POLY INT, FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
MF2b ==> M2(FRAC POLY FLOAT,VEC FRAC POLY FLOAT,VEC FRAC POLY FLOAT,
            MAT FRAC POLY FLOAT, FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
MF2c ==> M2(POLY INT,VEC POLY INT,VEC POLY INT,MAT POLY INT,
            FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
MF2d ==> M2(POLY FLOAT,VEC POLY FLOAT,VEC POLY FLOAT,
            MAT POLY FLOAT, FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
MF2e ==> M2(EXPR INT,VEC EXPR INT,VEC EXPR INT,MAT EXPR INT,
            FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)
MF2f ==> M2(EXPR FLOAT,VEC EXPR FLOAT,VEC EXPR FLOAT,
            MAT EXPR FLOAT, FEXPR,VEC FEXPR,VEC FEXPR,MAT FEXPR)

Exports ==> FortranMatrixFunctionCategory with
  coerce : MAT FEXPR -> $
    ++coerce(f) takes objects from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns them into an ASP.

Implementation ==> add

real : FSTU := ["real":FST]$FSTU
syms : SYMTAB := empty()$SYMTAB
declare!(XL,fortranReal(),syms)$SYMTAB
declare!(XR,fortranReal(),syms)$SYMTAB
declare!(ELAM,fortranReal(),syms)$SYMTAB
yType : FT := construct(real,["3":Symbol],false)$FT
declare!(YL,yType,syms)$SYMTAB
declare!(YR,yType,syms)$SYMTAB
Rep := FortranProgram(name,["void"]$FSTU, [XL,XR,ELAM,YL,YR],syms)

fexpr2expr(u:FEXPR):EXPR MFLOAT == coerce(u)$FEXPR

vecAssign(s:Symbol,u:VEC FEXPR):FC ==
  u' : VEC EXPR MFLOAT := map(fexpr2expr,u)$VF2(FEXPR,EXPR MFLOAT)
  assign(s,u')$FC

coerce(u:MAT FEXPR):$ ==
  [vecAssign(YL,row(u,1)),vecAssign(YR,row(u,2)),returns()$FC]$List(FC):: $

coerce(c:List FortranCode):$ == coerce(c)$Rep

coerce(r:RSFC):$ == coerce(r)$Rep

```

```

coerce(c:FortranCode):$ == coerce(c)$Rep

coerce(u:$):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

retract(u:MAT FRAC POLY INT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2a
  v::$

retractIfCan(u:MAT FRAC POLY INT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2a
  v case "failed" => "failed"
  (v::MAT FEXPR):: $

retract(u:MAT FRAC POLY FLOAT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2b
  v::$

retractIfCan(u:MAT FRAC POLY FLOAT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2b
  v case "failed" => "failed"
  (v::MAT FEXPR):: $

retract(u:MAT EXPR INT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2e
  v::$

retractIfCan(u:MAT EXPR INT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2e
  v case "failed" => "failed"
  (v::MAT FEXPR):: $

retract(u:MAT EXPR FLOAT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2f
  v::$

retractIfCan(u:MAT EXPR FLOAT):Union($,"failed") ==
  v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2f
  v case "failed" => "failed"
  (v::MAT FEXPR):: $

retract(u:MAT POLY INT):$ ==
  v : MAT FEXPR := map(retract,u)$MF2c
  v::$

retractIfCan(u:MAT POLY INT):Union($,"failed") ==

```



```

v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2c
v case "failed" => "failed"
(v::MAT FEXPR):: $

retract(u:MAT POLY FLOAT):$ ==
v : MAT FEXPR := map(retract,u)$MF2d
v:: $

retractIfCan(u:MAT POLY FLOAT):Union($,"failed") ==
v:Union(MAT FEXPR,"failed"):=map(retractIfCan,u)$MF2d
v case "failed" => "failed"
(v::MAT FEXPR):: $

```

— ASP80.dotabb —

```

"ASP80" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP80"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ASP80" -> "FS"

```

domain ASP9 Asp9

— Asp9.input —

```

)set break resume
)sys rm -f Asp9.output
)spool Asp9.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Asp9
--R
--R Asp9(name: Symbol) is a domain constructor
--R Abbreviation for Asp9 is ASP9
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ASP9
--R
--R----- Operations -----
--R coerce : FortranCode -> %          coerce : List(FortranCode) -> %

```

```

--R coerce : % -> OutputForm          outputAsFortran : % -> Void
--R retract : Polynomial(Float) -> %    retract : Expression(Float) -> %
--R coerce : FortranExpression([construct,QUOTEX],[construct,QUOTEY],MachineFloat) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R retract : Fraction(Polynomial(Integer)) -> %
--R retract : Fraction(Polynomial(Float)) -> %
--R retract : Polynomial(Integer) -> %
--R retract : Expression(Integer) -> %
--R retractIfCan : Fraction(Polynomial(Integer)) -> Union(%, "failed")
--R retractIfCan : Fraction(Polynomial(Float)) -> Union(%, "failed")
--R retractIfCan : Polynomial(Integer) -> Union(%, "failed")
--R retractIfCan : Polynomial(Float) -> Union(%, "failed")
--R retractIfCan : Expression(Integer) -> Union(%, "failed")
--R retractIfCan : Expression(Float) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)

```

— Asp9.help —

=====

Asp9 examples

=====

Asp9 produces Fortran for Type 9 ASPs, needed for NAG routines
d02bhf, d02cjf, d02ejf.
These ASPs represent a function of a scalar X and a vector Y, for example:

```

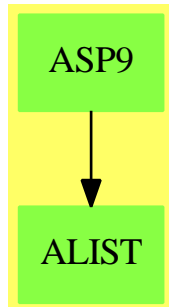
DOUBLE PRECISION FUNCTION G(X,Y)
DOUBLE PRECISION X,Y(*)
G=X+Y(1)
RETURN
END

```

If the user provides a constant value for G, then extra information is added
via COMMON blocks used by certain routines. This specifies that the value
returned by G in this case is to be ignored.

See Also:
o)show Asp9

Asp9 (ASP9)



Exports:

coerce outputAsFortran retract retractIfCan

— domain ASP9 Asp9 —

```

)abbrev domain ASP9 Asp9
++ Author: Mike Dewar, Grant Keady and Godfrey Nolan
++ Date Created: Mar 1993
++ Date Last Updated: 6 October 1994
++ Description:
++ \spadtype{Asp9} produces Fortran for Type 9 ASPs, needed for NAG routines
++ d02bhf, d02cjf, d02ejf.
++ These ASPs represent a function of a scalar X and a vector Y, for example:
++
++ \tab{5}DOUBLE PRECISION FUNCTION G(X,Y)\br
++ \tab{5}DOUBLE PRECISION X,Y(*)\br
++ \tab{5}G=X+Y(1)\br
++ \tab{5}RETURN\br
++ \tab{5}END
++
++ If the user provides a constant value for G, then extra information is added
++ via COMMON blocks used by certain routines. This specifies that the value
++ returned by G in this case is to be ignored.
  
```

```

Asp9(name): Exports == Implementation where
  name : Symbol
  
```

```

FEXPR ==> FortranExpression(['X'],['Y'],MFLOAT)
MFLOAT ==> MachineFloat
FC ==> FortranCode
FST ==> FortranScalarType
FT ==> FortranType
SYMTAB ==> SymbolTable
RSFC ==> Record(localSymbols:SymbolTable,code:List(FortranCode))
UFST ==> Union(fst:FST,void:"void")
  
```

```

FRAC    ==> Fraction
POLY    ==> Polynomial
EXPR    ==> Expression
INT     ==> Integer
FLOAT   ==> Float

Exports ==> FortranFunctionCategory with
  coerce : FEXPR -> %
    ++coerce(f) takes an object from the appropriate instantiation of
    ++\spadtype{FortranExpression} and turns it into an ASP.

Implementation ==> add

real : FST := "real":FST
syms : SYMTAB := empty()$SYMTAB
declare!(X,fortranReal()$FT,syms)$SYMTAB
yType : FT := construct([real]$UFST,["*":Symbol],false)$FT
declare!(Y,yType,syms)$SYMTAB
Rep := FortranProgram(name,[real]$UFST,[X,Y],syms)

retract(u:FRAC POLY INT):$ == (retract(u)$FEXPR)::$
retractIfCan(u:FRAC POLY INT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  (foo::FEXPR)::$

retract(u:FRAC POLY FLOAT):$ == (retract(u)$FEXPR)::$
retractIfCan(u:FRAC POLY FLOAT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  (foo::FEXPR)::$

retract(u:EXPR FLOAT):$ == (retract(u)$FEXPR)::$
retractIfCan(u:EXPR FLOAT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  (foo::FEXPR)::$

retract(u:EXPR INT):$ == (retract(u)$FEXPR)::$
retractIfCan(u:EXPR INT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  (foo::FEXPR)::$

retract(u:POLY FLOAT):$ == (retract(u)$FEXPR)::$
retractIfCan(u:POLY FLOAT):Union($,"failed") ==

```

```

foo : Union(FEXPR,"failed")
foo := retractIfCan(u)$FEXPR
foo case "failed" => "failed"
(foo::FEXPR)::$

retract(u:POLY INT):$ == (retract(u)@FEXPR)::$
retractIfCan(u:POLY INT):Union($,"failed") ==
  foo : Union(FEXPR,"failed")
  foo := retractIfCan(u)$FEXPR
  foo case "failed" => "failed"
  (foo::FEXPR)::$

coerce(u:FEXPR):% ==
  expr : Expression MachineFloat := (u::Expression(MachineFloat))$FEXPR
  (retractIfCan(u)@Union(MFLOAT,"failed"))$FEXPR case "failed" =>
    coerce(expr)$Rep
  locals : SYMTAB := empty()
  charType : FT := construct(["character":FST]$UFST,[6::POLY(INT)],false)$FT
  declare!([CHDUM1,CHDUM2,GOPT1,CHDUM,GOPT2],charType,locals)$SYMTAB
  common1 := common(CD02EJ,[CHDUM1,CHDUM2,GOPT1])$FC
  common2 := common(AD02CJ,[CHDUM,GOPT2])$FC
  assign1 := assign(GOPT1,"NOGOPT")$FC
  assign2 := assign(GOPT2,"NOGOPT")$FC
  result := assign(name,expr)$FC
  code : List FC := [common1,common2,assign1,assign2,result]
  ([locals,code]$RSFC)::Rep

coerce(c:List FortranCode):% == coerce(c)$Rep

coerce(r:RSFC):% == coerce(r)$Rep

coerce(c:FortranCode):% == coerce(c)$Rep

coerce(u:%):OutputForm == coerce(u)$Rep

outputAsFortran(u):Void ==
  p := checkPrecision()$NAGLinkSupportPackage
  outputAsFortran(u)$Rep
  p => restorePrecision()$NAGLinkSupportPackage

```

— ASP9.dotabb —

```

"ASP9" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ASP9"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ASP9" -> "ALIST"

```

domain JORDAN AssociatedJordanAlgebra

— AssociatedJordanAlgebra.input —

```
)set break resume
)sys rm -f AssociatedJordanAlgebra.output
)spool AssociatedJordanAlgebra.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AssociatedJordanAlgebra
--R
--R AssociatedJordanAlgebra(R: CommutativeRing,A: NonAssociativeAlgebra(R)) is a domain constructor
--R Abbreviation for AssociatedJordanAlgebra is JORDAN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for JORDAN
--R
--R----- Operations -----
--R ?? : (R,%) -> %           ?? : (%,R) -> %
--R ?? : (%,%) -> %           ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %           ?-? : (%,%) -> %
--R -? : % -> %               ?=? : (%,%) -> Boolean
--R 0 : () -> %               antiCommutator : (%,%) -> %
--R associator : (%,%,%) -> %   coerce : A -> %
--R coerce : % -> A             coerce : % -> OutputForm
--R commutator : (%,%) -> %     hash : % -> SingleInteger
--R latex : % -> String         sample : () -> %
--R zero? : % -> Boolean        ?~=? : (%,%) -> Boolean
--R ?? : (NonNegativeInteger,%) -> %
--R alternative? : () -> Boolean if A has FINAALG(R)
--R antiAssociative? : () -> Boolean if A has FINAALG(R)
--R antiCommutative? : () -> Boolean if A has FINAALG(R)
--R apply : (Matrix(R),%) -> % if A has FRNAALG(R)
--R associative? : () -> Boolean if A has FINAALG(R)
--R associatorDependence : () -> List(Vector(R)) if A has FINAALG(R) and R has INTDOM or A has FRNAALG(R)
--R basis : () -> Vector(%) if A has FRNAALG(R)
--R commutative? : () -> Boolean if A has FINAALG(R)
--R conditionsForIdempotents : Vector(%) -> List(Polynomial(R)) if A has FINAALG(R)
--R conditionsForIdempotents : () -> List(Polynomial(R)) if A has FRNAALG(R)
--R convert : % -> Vector(R) if A has FRNAALG(R)
--R convert : Vector(R) -> % if A has FRNAALG(R)
--R coordinates : (%,Vector(%)) -> Vector(R) if A has FINAALG(R)
```

```

--R coordinates : (Vector(%),Vector(%)) -> Matrix(R) if A has FINAALG(R)
--R coordinates : % -> Vector(R) if A has FRNAALG(R)
--R coordinates : Vector(%) -> Matrix(R) if A has FRNAALG(R)
--R ?.? : (%,Integer) -> R if A has FRNAALG(R)
--R flexible? : () -> Boolean if A has FINAALG(R)
--R jacobIdentity? : () -> Boolean if A has FINAALG(R)
--R jordanAdmissible? : () -> Boolean if A has FINAALG(R)
--R jordanAlgebra? : () -> Boolean if A has FINAALG(R)
--R leftAlternative? : () -> Boolean if A has FINAALG(R)
--R leftCharacteristicPolynomial : % -> SparseUnivariatePolynomial(R) if A has FINAALG(R)
--R leftDiscriminant : Vector(%) -> R if A has FINAALG(R)
--R leftDiscriminant : () -> R if A has FRNAALG(R)
--R leftMinimalPolynomial : % -> SparseUnivariatePolynomial(R) if A has FINAALG(R) and R has INT
--R leftNorm : % -> R if A has FINAALG(R)
--R leftPower : (%PositiveInteger) -> %
--R leftRankPolynomial : () -> SparseUnivariatePolynomial(Polynomial(R)) if A has FRNAALG(R) and
--R leftRecip : % -> Union(%,"failed") if A has FINAALG(R) and R has INTDOM or A has FRNAALG(R)
--R leftRegularRepresentation : (%Vector(%)) -> Matrix(R) if A has FINAALG(R)
--R leftRegularRepresentation : % -> Matrix(R) if A has FRNAALG(R)
--R leftTrace : % -> R if A has FINAALG(R)
--R leftTraceMatrix : Vector(%) -> Matrix(R) if A has FINAALG(R)
--R leftTraceMatrix : () -> Matrix(R) if A has FRNAALG(R)
--R leftUnit : () -> Union(%,"failed") if A has FINAALG(R) and R has INTDOM or A has FRNAALG(R)
--R leftUnits : () -> Union(Record(particular: %,basis: List(%)),"failed") if A has FINAALG(R) a
--R lieAdmissible? : () -> Boolean if A has FINAALG(R)
--R lieAlgebra? : () -> Boolean if A has FINAALG(R)
--R noncommutativeJordanAlgebra? : () -> Boolean if A has FINAALG(R)
--R plenaryPower : (%PositiveInteger) -> %
--R powerAssociative? : () -> Boolean if A has FINAALG(R)
--R rank : () -> PositiveInteger if A has FINAALG(R)
--R recip : % -> Union(%,"failed") if A has FINAALG(R) and R has INTDOM or A has FRNAALG(R) and
--R represents : (Vector(R),Vector(%)) -> % if A has FINAALG(R)
--R represents : Vector(R) -> % if A has FRNAALG(R)
--R rightAlternative? : () -> Boolean if A has FINAALG(R)
--R rightCharacteristicPolynomial : % -> SparseUnivariatePolynomial(R) if A has FINAALG(R)
--R rightDiscriminant : Vector(%) -> R if A has FINAALG(R)
--R rightDiscriminant : () -> R if A has FRNAALG(R)
--R rightMinimalPolynomial : % -> SparseUnivariatePolynomial(R) if A has FINAALG(R) and R has IN
--R rightNorm : % -> R if A has FINAALG(R)
--R rightPower : (%PositiveInteger) -> %
--R rightRankPolynomial : () -> SparseUnivariatePolynomial(Polynomial(R)) if A has FRNAALG(R) an
--R rightRecip : % -> Union(%,"failed") if A has FINAALG(R) and R has INTDOM or A has FRNAALG(R)
--R rightRegularRepresentation : (%Vector(%)) -> Matrix(R) if A has FINAALG(R)
--R rightRegularRepresentation : % -> Matrix(R) if A has FRNAALG(R)
--R rightTrace : % -> R if A has FINAALG(R)
--R rightTraceMatrix : Vector(%) -> Matrix(R) if A has FINAALG(R)
--R rightTraceMatrix : () -> Matrix(R) if A has FRNAALG(R)
--R rightUnit : () -> Union(%,"failed") if A has FINAALG(R) and R has INTDOM or A has FRNAALG(R)
--R rightUnits : () -> Union(Record(particular: %,basis: List(%)),"failed") if A has FINAALG(R)
--R someBasis : () -> Vector(%) if A has FINAALG(R)

```

```

--R structuralConstants : Vector(%) -> Vector(Matrix(R)) if A has FINAALG(R)
--R structuralConstants : () -> Vector(Matrix(R)) if A has FRNAALG(R)
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R unit : () -> Union(%, "failed") if A has FINAALG(R) and R has INTDOM or A has FRNAALG(R) and R has INTDOM
--R
--E 1

)spool
)lisp (bye)

```

— AssociatedJordanAlgebra.help —

```

=====
AssociatedJordanAlgebra examples
=====

```

AssociatedJordanAlgebra takes an algebra A and uses *\$A to define the new multiplications

$$a*b := (a *$A b + b *$A a)/2 \quad (\text{anticommutator})$$

The usual notation $\{a,b\}_+$ cannot be used due to restrictions in the current language.

This domain only gives a Jordan algebra if the Jordan-identity

$$(a*b)*c + (b*c)*a + (c*a)*b = 0$$

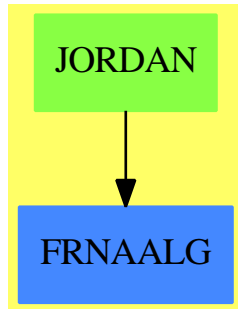
holds for all a, b, c in A.

This relation can be checked by `jordanAdmissible?()$A`.

If the underlying algebra is of type `FramedNonAssociativeAlgebra(R)` (i.e. a non-associative algebra over R which is a free R-module of finite rank, together with a fixed R-module basis), then the same is true for the associated Jordan algebra. Moreover, if the underlying algebra is of type `FiniteRankNonAssociativeAlgebra(R)` (i.e. a non-associative algebra over R which is a free R-module of finite rank), then the same true for the associated Jordan algebra.

See Also:

o `)show AssociatedJordanAlgebra`

AssociatedJordanAlgebra (JORDAN)**See**

⇒ “AssociatedLieAlgebra” (LIE) [2](#) on page [228](#)

⇒ “LieSquareMatrix” (LSQM) [13](#) on page [1455](#)

Exports:

0	alternative?
antiAssociative?	antiCommutative?
antiCommutator	apply
associative?	associator
associatorDependence	basis
coerce	commutative?
commutator	conditionsForIdempotents
convert	coordinates
flexible?	hash
jacobiIdentity?	jordanAdmissible?
jordanAlgebra?	latex
leftAlternative?	leftCharacteristicPolynomial
leftDiscriminant	leftMinimalPolynomial
leftNorm	leftPower
leftRankPolynomial	leftRecip
leftRegularRepresentation	leftTrace
leftTraceMatrix	leftUnit
leftUnits	lieAdmissible?
lieAlgebra?	noncommutativeJordanAlgebra?
plenaryPower	powerAssociative?
rank	recip
represents	rightAlternative?
rightCharacteristicPolynomial	rightDiscriminant
rightMinimalPolynomial	rightNorm
rightPower	rightRankPolynomial
rightRecip	rightRegularRepresentation
rightTrace	rightTraceMatrix
rightUnit	rightUnits
sample	someBasis
structuralConstants	subtractIfCan
unit	zero?
?*?	?**?
?+?	?-?
-?	?=?
?~=?	?..?

— domain JORDAN AssociatedJordanAlgebra —

```

)abbrev domain JORDAN AssociatedJordanAlgebra
++ Author: J. Grabmeier
++ Date Created: 14 June 1991
++ Date Last Updated: 14 June 1991
++ Description:
++ AssociatedJordanAlgebra takes an algebra \spad{A} and uses \spadfun{*$A}
++ to define the new multiplications \spad{a*b := (a *$A b + b *$A a)/2}
++ (anticommutator).
++ The usual notation \spad{{a,b}_+} cannot be used due to

```

```

++ restrictions in the current language.
++ This domain only gives a Jordan algebra if the
++ Jordan-identity  $\text{\spad{(a*b)*c + (b*c)*a + (c*a)*b = 0}}$  holds
++ for all  $\text{\spad{a}}, \text{\spad{b}}, \text{\spad{c}}$  in  $\text{\spad{A}}$ .
++ This relation can be checked by
++  $\text{\spadfun{jordanAdmissible?()}A}$ .
++
++ If the underlying algebra is of type
++  $\text{\spadtype{FramedNonAssociativeAlgebra(R)}}$  (i.e. a non
++ associative algebra over R which is a free R-module of finite
++ rank, together with a fixed R-module basis), then the same
++ is true for the associated Jordan algebra.
++ Moreover, if the underlying algebra is of type
++  $\text{\spadtype{FiniteRankNonAssociativeAlgebra(R)}}$  (i.e. a non
++ associative algebra over R which is a free R-module of finite
++ rank), then the same true for the associated Jordan algebra.

AssociatedJordanAlgebra(R:CommutativeRing,A:NonAssociativeAlgebra R):
  public == private where
  public ==> Join (NonAssociativeAlgebra R, CoercibleTo A) with
    coerce : A -> %
      ++ coerce(a) coerces the element  $\text{\spad{a}}$  of the algebra  $\text{\spad{A}}$ 
      ++ to an element of the Jordan algebra
      ++  $\text{\spadtype{AssociatedJordanAlgebra}(R,A)}$ .
  if A has FramedNonAssociativeAlgebra(R) then _
    FramedNonAssociativeAlgebra(R)
  if A has FiniteRankNonAssociativeAlgebra(R) then _
    FiniteRankNonAssociativeAlgebra(R)

private ==> A add
  Rep := A
  two : R := (1$R + 1$R)
  oneHalf : R := (recip two) :: R
  (a:%) * (b:%) ==
    zero? two => error
    "constructor must no be called with Ring of characteristic 2"
    ((a::Rep) * $Rep (b::Rep) + $Rep (b::Rep) * $Rep (a::Rep)) * oneHalf
  -- (a::Rep) * $Rep (b::Rep) + $Rep (b::Rep) * $Rep (a::Rep)
  coerce(a:%):A == a :: Rep
  coerce(a:A):% == a :: %
  (a:%) ** (n:PositiveInteger) == a

```

— JORDAN.dotabb —

"JORDAN" [color="#88FF44",href="bookvol10.3.pdf#nameddest=JORDAN"]
 "FRNAALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRNAALG"]

"JORDAN" -> "FRNAALG"

domain LIE AssociatedLieAlgebra

— AssociatedLieAlgebra.input —

```

)set break resume
)sys rm -f AssociatedLieAlgebra.output
)spool AssociatedLieAlgebra.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AssociatedLieAlgebra
--R
--R AssociatedLieAlgebra(R: CommutativeRing,A: NonAssociativeAlgebra(R)) is a domain constructor
--R Abbreviation for AssociatedLieAlgebra is LIE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for LIE
--R
--R----- Operations -----
--R ?? : (R,%) -> %           ?? : (%,R) -> %
--R ?? : (%,%) -> %           ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R ?? : (%,%) -> %           ?-? : (%,%) -> %
--R -? : % -> %               ?=? : (%,%) -> Boolean
--R 0 : () -> %               antiCommutator : (%,%) -> %
--R associator : (%,%,%) -> %   coerce : A -> %
--R coerce : % -> A            coerce : % -> OutputForm
--R commutator : (%,%) -> %     hash : % -> SingleInteger
--R latex : % -> String         sample : () -> %
--R zero? : % -> Boolean        ?~=? : (%,%) -> Boolean
--R ?? : (NonNegativeInteger,%) -> %
--R alternative? : () -> Boolean if A has FINAALG(R)
--R antiAssociative? : () -> Boolean if A has FINAALG(R)
--R antiCommutative? : () -> Boolean if A has FINAALG(R)
--R apply : (Matrix(R),%) -> % if A has FRNAALG(R)
--R associative? : () -> Boolean if A has FINAALG(R)
--R associatorDependence : () -> List(Vector(R)) if A has FINAALG(R) and R has INTDOM or A has FRNAALG(R)
--R basis : () -> Vector(%) if A has FRNAALG(R)
--R commutative? : () -> Boolean if A has FINAALG(R)
--R conditionsForIdempotents : Vector(%) -> List(Polynomial(R)) if A has FINAALG(R)
--R conditionsForIdempotents : () -> List(Polynomial(R)) if A has FRNAALG(R)

```

```

--R convert : % -> Vector(R) if A has FRNAALG(R)
--R convert : Vector(R) -> % if A has FRNAALG(R)
--R coordinates : (%,Vector(%)) -> Vector(R) if A has FINAALG(R)
--R coordinates : (Vector(%),Vector(%)) -> Matrix(R) if A has FINAALG(R)
--R coordinates : % -> Vector(R) if A has FRNAALG(R)
--R coordinates : Vector(%) -> Matrix(R) if A has FRNAALG(R)
--R ?.? : (%,Integer) -> R if A has FRNAALG(R)
--R flexible? : () -> Boolean if A has FINAALG(R)
--R jacobiiIdentity? : () -> Boolean if A has FINAALG(R)
--R jordanAdmissible? : () -> Boolean if A has FINAALG(R)
--R jordanAlgebra? : () -> Boolean if A has FINAALG(R)
--R leftAlternative? : () -> Boolean if A has FINAALG(R)
--R leftCharacteristicPolynomial : % -> SparseUnivariatePolynomial(R) if A has FINAALG(R)
--R leftDiscriminant : Vector(%) -> R if A has FINAALG(R)
--R leftDiscriminant : () -> R if A has FRNAALG(R)
--R leftMinimalPolynomial : % -> SparseUnivariatePolynomial(R) if A has FINAALG(R) and R has INTDOM
--R leftNorm : % -> R if A has FINAALG(R)
--R leftPower : (%,PositiveInteger) -> %
--R leftRankPolynomial : () -> SparseUnivariatePolynomial(Polynomial(R)) if A has FRNAALG(R) and R has INTDOM
--R leftRecip : % -> Union(%, "failed") if A has FINAALG(R) and R has INTDOM or A has FRNAALG(R)
--R leftRegularRepresentation : (%,Vector(%)) -> Matrix(R) if A has FINAALG(R)
--R leftRegularRepresentation : % -> Matrix(R) if A has FRNAALG(R)
--R leftTrace : % -> R if A has FINAALG(R)
--R leftTraceMatrix : Vector(%) -> Matrix(R) if A has FINAALG(R)
--R leftTraceMatrix : () -> Matrix(R) if A has FRNAALG(R)
--R leftUnit : () -> Union(%, "failed") if A has FINAALG(R) and R has INTDOM or A has FRNAALG(R)
--R leftUnits : () -> Union(Record(particular: %,basis: List(%)), "failed") if A has FINAALG(R) and R has INTDOM
--R lieAdmissible? : () -> Boolean if A has FINAALG(R)
--R lieAlgebra? : () -> Boolean if A has FINAALG(R)
--R noncommutativeJordanAlgebra? : () -> Boolean if A has FINAALG(R)
--R plenaryPower : (%,PositiveInteger) -> %
--R powerAssociative? : () -> Boolean if A has FINAALG(R)
--R rank : () -> PositiveInteger if A has FINAALG(R)
--R recip : % -> Union(%, "failed") if A has FINAALG(R) and R has INTDOM or A has FRNAALG(R) and R has INTDOM
--R represents : (Vector(R),Vector(%)) -> % if A has FINAALG(R)
--R represents : Vector(R) -> % if A has FRNAALG(R)
--R rightAlternative? : () -> Boolean if A has FINAALG(R)
--R rightCharacteristicPolynomial : % -> SparseUnivariatePolynomial(R) if A has FINAALG(R)
--R rightDiscriminant : Vector(%) -> R if A has FINAALG(R)
--R rightDiscriminant : () -> R if A has FRNAALG(R)
--R rightMinimalPolynomial : % -> SparseUnivariatePolynomial(R) if A has FINAALG(R) and R has INTDOM
--R rightNorm : % -> R if A has FINAALG(R)
--R rightPower : (%,PositiveInteger) -> %
--R rightRankPolynomial : () -> SparseUnivariatePolynomial(Polynomial(R)) if A has FRNAALG(R) and R has INTDOM
--R rightRecip : % -> Union(%, "failed") if A has FINAALG(R) and R has INTDOM or A has FRNAALG(R) and R has INTDOM
--R rightRegularRepresentation : (%,Vector(%)) -> Matrix(R) if A has FINAALG(R)
--R rightRegularRepresentation : % -> Matrix(R) if A has FRNAALG(R)
--R rightTrace : % -> R if A has FINAALG(R)
--R rightTraceMatrix : Vector(%) -> Matrix(R) if A has FINAALG(R)
--R rightTraceMatrix : () -> Matrix(R) if A has FRNAALG(R)

```

```

--R rightUnit : () -> Union(%,"failed") if A has FINAALG(R) and R has INTDOM or A has FRNAALG(R) and R has INTDOM
--R rightUnits : () -> Union(Record(particular: %,basis: List(%)),"failed") if A has FINAALG(R) and R has INTDOM
--R someBasis : () -> Vector(%) if A has FINAALG(R)
--R structuralConstants : Vector(%) -> Vector(Matrix(R)) if A has FINAALG(R)
--R structuralConstants : () -> Vector(Matrix(R)) if A has FRNAALG(R)
--R subtractIfCan : (%,%) -> Union(%,"failed")
--R unit : () -> Union(%,"failed") if A has FINAALG(R) and R has INTDOM or A has FRNAALG(R) and R has INTDOM
--R
--E 1

```

```

)spool
)lisp (bye)

```

— AssociatedLieAlgebra.help —

=====

AssociatedLieAlgebra examples

=====

AssociatedLieAlgebra takes an algebra A and uses *\$A to define the Lie bracket

$a*b := (a * \$A b - b * \$A a)$ (commutator).

Note that the notation $[a,b]$ cannot be used due to restrictions of the current compiler. This domain only gives a Lie algebra if the Jacobi-identity

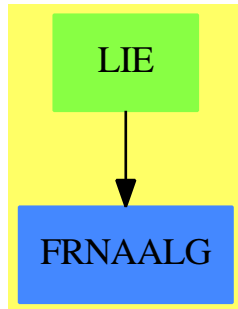
$$(a*b)*c + (b*c)*a + (c*a)*b = 0$$

holds for all a, b, c in A. This relation can be checked by `lieAdmissible?()$A`.

If the underlying algebra is of type `FramedNonAssociativeAlgebra(R)` (i.e. a non-associative algebra over R which is a free R-module of finite rank, together with a fixed R-module basis), then the same is true for the associated Lie algebra. Also, if the underlying algebra is of type `FiniteRankNonAssociativeAlgebra(R)` (i.e. a non-associative algebra over R which is a free R-module of finite rank), then the same is true for the associated Lie algebra.

See Also:

`o)show AssociatedLieAlgebra`

AssociatedLieAlgebra (LIE)**See**

⇒ “AssociatedJordanAlgebra” (JORDAN) [2](#) on page [222](#)

⇒ “LieSquareMatrix” (LSQM) [13](#) on page [1455](#)

Exports:

0	alternative?
antiAssociative?	antiCommutative?
antiCommutator	apply
associative?	associator
associatorDependence	basis
coerce	commutative?
commutator	conditionsForIdempotents
convert	coordinates
flexible?	hash
jacobiIdentity?	jordanAdmissible?
jordanAlgebra?	latex
leftAlternative?	leftCharacteristicPolynomial
leftDiscriminant	leftMinimalPolynomial
leftNorm	leftPower
leftRankPolynomial	leftRecip
leftRegularRepresentation	leftTrace
leftTraceMatrix	leftUnit
leftUnits	lieAdmissible?
lieAlgebra?	noncommutativeJordanAlgebra?
plenaryPower	powerAssociative?
rank	recip
represents	represents
rightAlternative?	rightCharacteristicPolynomial
rightDiscriminant	rightMinimalPolynomial
rightNorm	rightPower
rightRankPolynomial	rightRecip
rightRegularRepresentation	rightTrace
rightTraceMatrix	rightUnit
rightUnits	sample
someBasis	structuralConstants
structuralConstants	subtractIfCan
unit	zero?
?*?	?**?
?+?	?-?
-?	?=?
?~=?	?..?

— domain LIE AssociatedLieAlgebra —

```

)abbrev domain LIE AssociatedLieAlgebra
++ Author: J. Grabmeier
++ Date Created: 07 March 1991
++ Date Last Updated: 14 June 1991
++ Description:
++ AssociatedLieAlgebra takes an algebra \spad{A}
++ and uses \spadfun{*$A} to define the
++ Lie bracket \spad{a*b := (a *$A b - b *$A a)} (commutator). Note that

```



```

++ the notation \spad{[a,b]} cannot be used due to
++ restrictions of the current compiler.
++ This domain only gives a Lie algebra if the
++ Jacobi-identity \spad{(a*b)*c + (b*c)*a + (c*a)*b = 0} holds
++ for all \spad{a},\spad{b},\spad{c} in \spad{A}.
++ This relation can be checked by
++ \spad{lieAdmissible?()$A}.
++
++ If the underlying algebra is of type
++ \spadtype{FramedNonAssociativeAlgebra(R)} (i.e. a non
++ associative algebra over R which is a free \spad{R}-module of finite
++ rank, together with a fixed \spad{R}-module basis), then the same
++ is true for the associated Lie algebra.
++ Also, if the underlying algebra is of type
++ \spadtype{FiniteRankNonAssociativeAlgebra(R)} (i.e. a non
++ associative algebra over R which is a free R-module of finite
++ rank), then the same is true for the associated Lie algebra.

AssociatedLieAlgebra(R:CommutativeRing,A:NonAssociativeAlgebra R):
  public == private where
  public ==> Join (NonAssociativeAlgebra R, CoercibleTo A) with
    coerce : A -> %
      ++ coerce(a) coerces the element \spad{a} of the algebra \spad{A}
      ++ to an element of the Lie
      ++ algebra \spadtype{AssociatedLieAlgebra}(R,A).
  if A has FramedNonAssociativeAlgebra(R) then
    FramedNonAssociativeAlgebra(R)
  if A has FiniteRankNonAssociativeAlgebra(R) then
    FiniteRankNonAssociativeAlgebra(R)

private ==> A add
  Rep := A
  (a:%) * (b:%) == (a::Rep) * $Rep (b::Rep) - $Rep (b::Rep) * $Rep (a::Rep)
  coerce(a:%):A == a :: Rep
  coerce(a:A):% == a :: %
  (a:%) ** (n:PositiveInteger) ==
    n = 1 => a
    0

```

— LIE.dotabb —

```

"LIE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=LIE"]
"FRNAALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRNAALG"]
"LIE" -> "FRNAALG"

```

domain ALIST AssociationList

— AssociationList.input —

```

)set break resume
)sys rm -f AssociationList.output
)spool AssociationList.output
)set message test on
)set message auto off
)clear all
--S 1 of 10
Data := Record(monthsOld : Integer, gender : String)
--R
--R
--R (1) Record(monthsOld: Integer,gender: String)
--R
--R                                          Type: Domain
--E 1

--S 2 of 10
al : AssociationList(String,Data)
--R
--R
--R                                          Type: Void
--E 2

--S 3 of 10
al := table()
--R
--R
--R (3) table()
--R
--R      Type: AssociationList(String,Record(monthsOld: Integer,gender: String))
--E 3

--S 4 of 10
al."bob" := [407,"male"]$Data
--R
--R
--R (4) [monthsOld= 407,gender= "male"]
--R
--R      Type: Record(monthsOld: Integer,gender: String)
--E 4

--S 5 of 10
al."judith" := [366,"female"]$Data
--R
--R
--R (5) [monthsOld= 366,gender= "female"]
--R
--R      Type: Record(monthsOld: Integer,gender: String)
--E 5

```

```

--S 6 of 10
al."katie" := [24,"female"]$Data
--R
--R
--R (6) [monthsOld= 24,gender= "female"]
--R                                     Type: Record(monthsOld: Integer,gender: String)
--E 6

--S 7 of 10
al."smokie" := [200,"female"]$Data
--R
--R
--R (7) [monthsOld= 200,gender= "female"]
--R                                     Type: Record(monthsOld: Integer,gender: String)
--E 7

--S 8 of 10
al
--R
--R
--R (8)
--R table
--R      "smokie"= [monthsOld= 200,gender= "female"]
--R      ,
--R      "katie"= [monthsOld= 24,gender= "female"]
--R      ,
--R      "judith"= [monthsOld= 366,gender= "female"]
--R      ,
--R      "bob"= [monthsOld= 407,gender= "male"]
--R      Type: AssociationList(String,Record(monthsOld: Integer,gender: String))
--E 8

--S 9 of 10
al."katie" := [23,"female"]$Data
--R
--R
--R (9) [monthsOld= 23,gender= "female"]
--R                                     Type: Record(monthsOld: Integer,gender: String)
--E 9

--S 10 of 10
delete!(al,1)
--R
--R
--R (10)
--R table
--R      "katie"= [monthsOld= 23,gender= "female"]
--R      ,
--R      "judith"= [monthsOld= 366,gender= "female"]
--R      ,

```

```

--R      "bob"= [monthsOld= 407,gender= "male"]
--R      Type: AssociationList(String,Record(monthsOld: Integer,gender: String))
--E 10
)spool
)lisp (bye)

```

— AssociationList.help —

=====

AssociationList examples

=====

AssociationList implements association lists. These may be viewed as lists of pairs where the first part is a key and the second is the stored value. For example, the key might be a string with a persons employee identification number and the value might be a record with personnel data.

The AssociationList constructor provides a general structure for associative storage. This type provides association lists in which data objects can be saved according to keys of any type. For a given association list, specific types must be chosen for the keys and entries. You can think of the representation of an association list as a list of records with key and entry fields.

Association lists are a form of table and so most of the operations available for Table are also available for AssociationList. They can also be viewed as lists and can be manipulated accordingly.

This is a Record type with age and gender fields.

```

Data := Record(monthsOld : Integer, gender : String)
      Record(monthsOld: Integer,gender: String)
            Type: Domain

```

In this expression, al is declared to be an association list whose keys are strings and whose entries are the above records.

```

al : AssociationList(String,Data)
      Type: Void

```

The table operation is used to create an empty association list.

```

al := table()
table()
      Type: AssociationList(String,Record(monthsOld: Integer,gender: String))

```

You can use assignment syntax to add things to the association list.

```
al."bob" := [407,"male"]$Data
[monthsOld=407, gender= "male"]
      Type: Record(monthsOld: Integer,gender: String)

al."judith" := [366,"female"]$Data
[monthsOld=366, gender= "female"]
      Type: Record(monthsOld: Integer,gender: String)

al."katie" := [24,"female"]$Data
[monthsOld=24, gender= "female"]
      Type: Record(monthsOld: Integer,gender: String)
```

Perhaps we should have included a species field.

```
al."smokie" := [200,"female"]$Data
[monthsOld=200, gender= "female"]
      Type: Record(monthsOld: Integer,gender: String)
```

Now look at what is in the association list. Note that the last-added (key, entry) pair is at the beginning of the list.

```
al
table("smokie" = [monthsOld=200, gender= "female"],
      "katie" = [monthsOld=24, gender= "female"],
      "judith" = [monthsOld=366, gender= "female"],
      "bob" = [monthsOld=407, gender= "male"])
      Type: AssociationList(String,Record(monthsOld: Integer,gender: String))
```

You can reset the entry for an existing key.

```
al."katie" := [23,"female"]$Data
[monthsOld=23, gender= "female"]
      Type: Record(monthsOld: Integer,gender: String)
```

Use `delete!` to destructively remove an element of the association list. Use `delete` to return a copy of the association list with the element deleted. The second argument is the index of the element to delete.

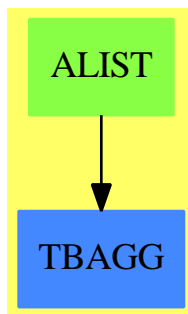
```
delete!(al,1)
table("katie" = [monthsOld=23, gender= "female"],
      "judith" = [monthsOld=366, gender= "female"],
      "bob" = [monthsOld=407, gender= "male"])
      Type: AssociationList(String,Record(monthsOld: Integer,gender: String))
```

See Also:

- o `)help Table`
- o `)help List`

```
o )show AssociationList
```

AssociationList (ALIST)



See

⇒ “IndexedList” (ILIST) [10](#) on page [1231](#)

⇒ “List” (LIST) [13](#) on page [1504](#)

Exports:

any?	assoc	bag	child?	children
coerce	concat	concat!	construct	convert
copy	copyInto!	count	cycleEntry	cycleLength
cycleSplit!	cycleTail	cyclic?	delete	delete!
dictionary	distance	elt	empty	empty?
entries	entry?	eq?	eval	every?
explicitlyFinite?	extract!	fill!	find	first
hash	index?	indices	insert	insert!
inspect	key?	keys	last	latex
leaf?	leaves	less?	list	map
map!	max	maxIndex	member?	members
merge	merge!	min	minIndex	more?
new	node?	nodes	parts	position
possiblyInfinite?	qelt	qsetelt!	reduce	remove
remove!	removeDuplicates	removeDuplicates!	rest	reverse
reverse!	sample	search	second	select
select!	setchildren!	setelt	setfirst!	setlast!
setrest!	setvalue!	size?	sort	sort!
sorted?	split!	swap!	table	tail
third	value	#?	?<?	?<=?
?=?	?>?	?>=?	?~=?	?..rest
?..value	?..first	?..last	?..?	

— domain ALIST AssociationList —

```

)abbrev domain ALIST AssociationList
++ Author: Mark Botch
++ Description:
++ \spadtype{AssociationList} implements association lists. These
++ may be viewed as lists of pairs where the first part is a key
++ and the second is the stored value. For example, the key might
++ be a string with a persons employee identification number and
++ the value might be a record with personnel data.

AssociationList(Key:SetCategory, Entry:SetCategory):
  AssociationListAggregate(Key, Entry) == add
    Pair ==> Record(key:Key, entry:Entry)
    Rep := Reference List Pair

    dictionary()      == ref empty()
    empty()           == dictionary()
    empty? t          == empty? deref t
    entries(t:%):List(Pair) == deref t
    parts(t:%):List(Pair) == deref t
    keys t            == [k.key for k in deref t]
    # t               == # deref t
    first(t:%):Pair   == first deref t
    rest t            == ref rest deref t
    concat(p:Pair, t:%) == ref concat(p, deref t)
    setrest_!(a:%, b:%) == ref setrest_!(deref a, deref b)
    setfirst_!(a:%, p:Pair) == setfirst_!(deref a,p)
    minIndex(a:%):Integer == minIndex(deref a)
    maxIndex(a:%):Integer == maxIndex(deref a)

    search(k, t) ==
      for r in deref t repeat
        k = r.key => return(r.entry)
      "failed"

    latex(a : %) : String ==
      l : List Pair := entries a
      s : String := "\left["
      while not empty?(l) repeat
        r : Pair := first l
        l := rest l
        s := concat(s, concat(latex r.key, concat(" = ", latex r.entry)$String)$String)$String
        if not empty?(l) then s := concat(s, ", ")$String
      concat(s, " \right]")$String

-- assoc(k, l) ==
--   (r := find(#1.key=k, l)) case "failed" => "failed"
--   r

```

```

assoc(k, t) ==
  for r in deref t repeat
    k = r.key => return r
  "failed"

setelt(t:%, k:Key, e:Entry) ==
  (r := assoc(k, t)) case Pair => (r::Pair).entry := e
  setref(t, concat([k, e], deref t))
  e

remove_!(k:Key, t:%) ==
  empty?(l := deref t) => "failed"
  k = first(l).key =>
    setref(t, rest l)
    first(l).entry
  prev := l
  curr := rest l
  while not empty? curr and first(curr).key ^= k repeat
    prev := curr
    curr := rest curr
  empty? curr => "failed"
  setrest_!(prev, rest curr)
  first(curr).entry

```

— ALIST.dotabb —

```

"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]
"ALIST" -> "TBAGG"

```

domain ATTRIBUT AttributeButtons

— AttributeButtons.input —

```

)set break resume
)sys rm -f AttributeButtons.output
)spool AttributeButtons.output
)set message test on
)set message auto off

```



```

)clear all

--S 1 of 1
)show AttributeButtons
--R AttributeButtons is a domain constructor
--R Abbreviation for AttributeButtons is ATTRBUT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ATTRBUT
--R
--R----- Operations -----
--R ?? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R decrease : String -> Float      hash : % -> SingleInteger
--R increase : String -> Float      latex : % -> String
--R resetAttributeButtons : () -> Void  ?~=? : (%,% ) -> Boolean
--R decrease : (String,String) -> Float
--R getButtonValue : (String,String) -> Float
--R increase : (String,String) -> Float
--R setAttributeButtonStep : Float -> Float
--R setButtonValue : (String,String,Float) -> Float
--R setButtonValue : (String,Float) -> Float
--R
--E 1

)spool
)lisp (bye)

```

— AttributeButtons.help —

=====

AttributeButtons examples

=====

AttributeButtons implements a database and associated adjustment mechanisms for a set of attributes.

For ODEs these attributes are "stiffness", "stability" (i.e. how much affect the cosine or sine component of the solution has on the stability of the result), "accuracy" and "expense" (i.e. how expensive is the evaluation of the ODE). All these have bearing on the cost of calculating the solution given that reducing the step-length to achieve greater accuracy requires considerable number of evaluations and calculations.

The effect of each of these attributes can be altered by increasing or decreasing the button value.

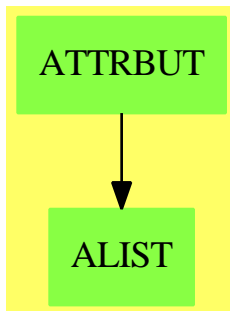
For Integration there is a button for increasing and decreasing the preset number of function evaluations for each method. This is automatically used

by ANNA when a method fails due to insufficient workspace or where the limit of function evaluations has been reached before the required accuracy is achieved.

See Also:

o)show AttributeButtons

AttributeButtons (ATTRBUT)



Exports:

coerce	decrease	getButtonValue	hash
increase	latex	resetAttributeButtons	setAttributeButtonStep
setButtonValue	?~=?	?=?	

— domain ATTRBUT AttributeButtons —

```

)abbrev domain ATTRBUT AttributeButtons
++ Author: Brian Dupee
++ Date Created: April 1996
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{AttributeButtons} implements a database and associated
++ adjustment mechanisms for a set of attributes.
++
++ For ODEs these attributes are "stiffness", "stability" (i.e. how much
++ affect the cosine or sine component of the solution has on the stability of
++ the result), "accuracy" and "expense" (i.e. how expensive is the evaluation
++ of the ODE). All these have bearing on the cost of calculating the
++ solution given that reducing the step-length to achieve greater accuracy
++ requires considerable number of evaluations and calculations.
++
++ The effect of each of these attributes can be altered by increasing or
++ decreasing the button value.
  
```

```

++
++ For Integration there is a button for increasing and decreasing the preset
++ number of function evaluations for each method. This is automatically used
++ by ANNA when a method fails due to insufficient workspace or where the
++ limit of function evaluations has been reached before the required
++ accuracy is achieved.

```

```

AttributeButtons(): E == I where
  F      ==> Float
  ST     ==> String
  LST    ==> List String
  Rec    ==> Record(key:Symbol,entry:Any)
  RList  ==> List(Record(key:Symbol,entry:Any))
  IFL    ==> List(Record(iffail:Integer,instruction:ST))
  Entry ==> Record(chapter:ST, type:ST, domainName: ST,
                  defaultMin:F, measure:F, failList:IFL, explList:LST)

```

```

E ==> SetCategory with

```

```

  increase:(ST,ST) -> F
    ++ \axiom{increase(routineName,attributeName)} increases the value
    ++ for the effect of the attribute \axiom{attributeName} with routine
    ++ \axiom{routineName}.
    ++
    ++ \axiom{attributeName} should be one of the values
    ++ "stiffness", "stability", "accuracy", "expense" or
    ++ "functionEvaluations".
  increase:(ST) -> F
    ++ \axiom{increase(attributeName)} increases the value for the
    ++ effect of the attribute \axiom{attributeName} with all routines.
    ++
    ++ \axiom{attributeName} should be one of the values
    ++ "stiffness", "stability", "accuracy", "expense" or
    ++ "functionEvaluations".
  decrease:(ST,ST) -> F
    ++ \axiom{decrease(routineName,attributeName)} decreases the value
    ++ for the effect of the attribute \axiom{attributeName} with routine
    ++ \axiom{routineName}.
    ++
    ++ \axiom{attributeName} should be one of the values
    ++ "stiffness", "stability", "accuracy", "expense" or
    ++ "functionEvaluations".
  decrease:(ST) -> F
    ++ \axiom{decrease(attributeName)} decreases the value for the
    ++ effect of the attribute \axiom{attributeName} with all routines.
    ++
    ++ \axiom{attributeName} should be one of the values
    ++ "stiffness", "stability", "accuracy", "expense" or
    ++ "functionEvaluations".

```

```

getButtonValue:(ST,ST) -> F
  ++ \axiom{getButtonValue(routineName,attributeName)} returns the
  ++ current value for the effect of the attribute \axiom{attributeName}
  ++ with routine \axiom{routineName}.
  ++
  ++ \axiom{attributeName} should be one of the values
  ++ "stiffness", "stability", "accuracy", "expense" or
  ++ "functionEvaluations".
resetAttributeButtons:() -> Void
  ++ \axiom{resetAttributeButtons()} resets the Attribute buttons to a
  ++ neutral level.
setAttributeButtonStep:(F) -> F
  ++ \axiom{setAttributeButtonStep(n)} sets the value of the steps for
  ++ increasing and decreasing the button values. \axiom{n} must be
  ++ greater than 0 and less than 1. The preset value is 0.5.
setButtonValue:(ST,F) -> F
  ++ \axiom{setButtonValue(attributeName,n)} sets the
  ++ value of all buttons of attribute \spad{attributeName}
  ++ to \spad{n}. \spad{n} must be in the range [0..1].
  ++
  ++ \axiom{attributeName} should be one of the values
  ++ "stiffness", "stability", "accuracy", "expense" or
  ++ "functionEvaluations".
setButtonValue:(ST,ST,F) -> F
  ++ \axiom{setButtonValue(attributeName,routineName,n)} sets the
  ++ value of the button of attribute \spad{attributeName} to routine
  ++ \spad{routineName} to \spad{n}. \spad{n} must be in the range [0..1].
  ++
  ++ \axiom{attributeName} should be one of the values
  ++ "stiffness", "stability", "accuracy", "expense" or
  ++ "functionEvaluations".
finiteAggregate

I ==> add

Rep := StringTable(F)
import Rep

buttons:() -> $

buttons():$ ==
  eList := empty()$List(Record(key:ST,entry:F))
  l1:List String := ["stability","stiffness","accuracy","expense"]
  l2:List String := ["functionEvaluations"]
  ro1 := selectODEIVPRoutines(r := routines())$RoutinesTable)$RoutinesTable
  ro2 := selectIntegrationRoutines(r)$RoutinesTable
  k1:List String := [string(i)$Symbol for i in keys(ro1)$RoutinesTable]
  k2:List String := [string(i)$Symbol for i in keys(ro2)$RoutinesTable]
  for i in k1 repeat
    for j in l1 repeat

```

```

        e:Record(key:ST,entry:F) := [i j,0.5]
        eList := cons(e,eList)$List(Record(key:ST,entry:F))
    for i in k2 repeat
        for j in l2 repeat
            e:Record(key:ST,entry:F) := [i j,0.5]
            eList := cons(e,eList)$List(Record(key:ST,entry:F))
    construct(eList)$Rep

attributeButtons:$ := buttons()

attributeStep:F := 0.5

setAttributeButtonStep(n:F):F ==
    positive?(n)$F and (n<1$F) => attributeStep:F := n
    error("setAttributeButtonStep","New value must be in (0..1)")$ErrorFunctions

resetAttributeButtons():Void ==
    attributeButtons := buttons()
    void()$Void

setButtonValue(routineName:ST,attributeName:ST,n:F):F ==
    f := search(routineName attributeName,attributeButtons)$Rep
    f case Float =>
        n>=0$F and n<=1$F =>
            setelt(attributeButtons,routineName attributeName,n)$Rep
            error("setAttributeButtonStep","New value must be in [0..1]")$ErrorFunctions
            error("setButtonValue","attribute name " attributeName
                " not found for routine " routineName)$ErrorFunctions

setButtonValue(attributeName:ST,n:F):F ==
    ro1 := selectODEIVPRoutines(r := routines())$RoutinesTable)$RoutinesTable
    ro2 := selectIntegrationRoutines(r)$RoutinesTable
    l1:List String := ["stability","stiffness","accuracy","expense"]
    l2:List String := ["functionEvaluations"]
    if attributeName="functionEvaluations" then
        for i in keys(ro2)$RoutinesTable repeat
            setButtonValue(string(i)$Symbol,attributeName,n)
    else
        for i in keys(ro1)$RoutinesTable repeat
            setButtonValue(string(i)$Symbol,attributeName,n)
    n

increase(routineName:ST,attributeName:ST):F ==
    f := search(routineName attributeName,attributeButtons)$Rep
    f case Float =>
        newValue:F := (1$F-attributeStep)*f+attributeStep
        setButtonValue(routineName,attributeName,newValue)
        error("increase","attribute name " attributeName
            " not found for routine " routineName)$ErrorFunctions

```

```

increase(attributeName:ST):F ==
  ro1 := selectODEIVPRoutines(r := routines())$RoutinesTable)$RoutinesTable
  ro2 := selectIntegrationRoutines(r)$RoutinesTable
  l1:List String := ["stability","stiffness","accuracy","expense"]
  l2:List String := ["functionEvaluations"]
  if attributeName="functionEvaluations" then
    for i in keys(ro2)$RoutinesTable repeat
      increase(string(i)$Symbol,attributeName)
  else
    for i in keys(ro1)$RoutinesTable repeat
      increase(string(i)$Symbol,attributeName)
  getButtonValue(string(i)$Symbol,attributeName)

decrease(routineName:ST,attributeName:ST):F ==
  f := search(routineName attributeName,attributeButtons)$Rep
  f case Float =>
    newValue:F := (1$F-attributeStep)*f
    setButtonValue(routineName,attributeName,newValue)
  error("increase","attribute name " attributeName
    " not found for routine " routineName)$ErrorFunctions

decrease(attributeName:ST):F ==
  ro1 := selectODEIVPRoutines(r := routines())$RoutinesTable)$RoutinesTable
  ro2 := selectIntegrationRoutines(r)$RoutinesTable
  l1:List String := ["stability","stiffness","accuracy","expense"]
  l2:List String := ["functionEvaluations"]
  if attributeName="functionEvaluations" then
    for i in keys(ro2)$RoutinesTable repeat
      decrease(string(i)$Symbol,attributeName)
  else
    for i in keys(ro1)$RoutinesTable repeat
      decrease(string(i)$Symbol,attributeName)
  getButtonValue(string(i)$Symbol,attributeName)

getButtonValue(routineName:ST,attributeName:ST):F ==
  f := search(routineName attributeName,attributeButtons)$Rep
  f case Float => f
  error("getButtonValue","attribute name " attributeName
    " not found for routine " routineName)$ErrorFunctions

```

— ATTRIBUT.dotabb —

```

"ATTRIBUT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ATTRIBUT"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ATTRIBUT" -> "ALIST"

```

domain AUTOMOR Automorphism

— Automorphism.input —

```
)set break resume
)sys rm -f Automorphism.output
)spool Automorphism.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Automorphism
--R
--R Automorphism(R: Ring) is a domain constructor
--R Abbreviation for Automorphism is AUTOMOR
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for AUTOMOR
--R
--R----- Operations -----
--R ???: (% ,%) -> %                ??? : (% ,Integer) -> %
--R ??? : (% ,PositiveInteger) -> %  ?/? : (% ,%) -> %
--R ?=? : (% ,%) -> Boolean          1 : () -> %
--R ?^? : (% ,Integer) -> %          ?^? : (% ,PositiveInteger) -> %
--R coerce : % -> OutputForm         commutator : (% ,%) -> %
--R conjugate : (% ,%) -> %           ?.? : (% ,R) -> R
--R hash : % -> SingleInteger         inv : % -> %
--R latex : % -> String               morphism : (R -> R) -> %
--R one? : % -> Boolean               recip : % -> Union(% , "failed")
--R sample : () -> %                 ?~=? : (% ,%) -> Boolean
--R ??? : (% ,NonNegativeInteger) -> %
--R ?^? : (% ,NonNegativeInteger) -> %
--R morphism : ((R ,Integer) -> R) -> %
--R morphism : ((R -> R) , (R -> R)) -> %
--R
--E 1

)spool
)lisp (bye)
```

— Automorphism.help —

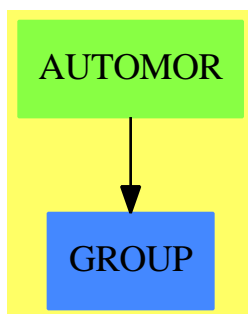
```
=====
Automorphism examples
=====
```

Automorphism R is the multiplicative group of automorphisms of R.
 In fact, non-invertible endomorphism are allowed as partial functions.
 This domain is noncanonical in that $f \cdot f^{-1}$ will be the identity
 function but won't be equal to 1.

See Also:

o)show Automorphism

Automorphism (AUTOMOR)



See

⇒ “SparseUnivariateSkewPolynomial” (ORESUP) [20](#) on page [2492](#)

⇒ “UnivariateSkewPolynomial” (OREUP) [22](#) on page [2872](#)

Exports:

1	coerce	commutator	conjugate	hash
inv	latex	morphism	one?	recip
sample	?~=?	?**?	?^?	?..?
?*?	?/?	?=?		

— domain AUTOMOR Automorphism —

```
)abbrev domain AUTOMOR Automorphism
```

```
++ Author: Manuel Bronstein
```

```
++ Date Created: 31 January 1994
```

```
++ Date Last Updated: 31 January 1994
```

```
++ Description:
```

```
++ Automorphism R is the multiplicative group of automorphisms of R.
```

```
-- In fact, non-invertible endomorphism are allowed as partial functions.
```



```

-- This domain is noncanonical in that  $f \cdot f^{-1}$  will be the identity
-- function but won't be equal to 1.

Automorphism(R:Ring): Join(Group, Eltable(R, R)) with
  morphism: (R -> R) -> %
    ++ morphism(f) returns the non-invertible morphism given by f.
  morphism: (R -> R, R -> R) -> %
    ++ morphism(f, g) returns the invertible morphism given by f, where
    ++ g is the inverse of f..
  morphism: ((R, Integer) -> R) -> %
    ++ morphism(f) returns the morphism given by  $\text{\spad}\{f^n(x) = f(x,n)\}$ .
== add
err:   R -> R
ident: (R, Integer) -> R
iter:  (R -> R, NonNegativeInteger, R) -> R
iterat: (R -> R, R -> R, Integer, R) -> R
apply: (% , R, Integer) -> R

Rep := ((R, Integer) -> R)

1 == ident
err r == error "Morphism is not invertible"
ident(r, n) == r
f = g == EQ(f, g)$Lisp
elt(f, r) == apply(f, r, 1)
inv f == (r1:R, i2:Integer):R +-> apply(f, r1, - i2)
f ** n == (r1:R, i2:Integer):R +-> apply(f, r1, n * i2)
coerce(f:%):OutputForm == message("R -> R")
morphism(f:(R, Integer) -> R):% == f
morphism(f:R -> R):% == morphism(f, err)
morphism(f, g) == (r1:R, i2:Integer):R +-> iterat(f, g, i2, r1)
apply(f, r, n) == (g := f pretend ((R, Integer) -> R); g(r, n))

iterat(f, g, n, r) ==
  n < 0 => iter(g, (-n)::NonNegativeInteger, r)
  iter(f, n::NonNegativeInteger, r)

iter(f, n, r) ==
  for i in 1..n repeat r := f r
  r

f * g ==
f = g => f**2
(r1:R, i2:Integer):R +->
  iterat((u1:R):R +-> f g u1,
    (v1:R):R +-> (inv g)(inv f) v1,
    i2, r1)

```

— AUTOMOR.dotabb —

```
"AUTOMOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=AUTOMOR"]  
"GROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=GROUP"]  
"AUTOMOR" -> "GROUP"
```

Chapter 3

Chapter B

domain BBTREE BalancedBinaryTree

— BalancedBinaryTree.input —

```
)set break resume
)sys rm -f BalancedBinaryTree.output
)spool BalancedBinaryTree.output
)set message test on
)set message auto off
)clear all
--S 1 of 7
lm := [3,5,7,11]
--E 1

--S 2 of 7
t := balancedBinaryTree(#lm, 0)
--E 2

--S 3 of 7
setleaves!(t,lm)
--E 3

--S 4 of 7
mapUp!(t,_)
--E 4

--S 5 of 7
t
--E 5

--S 6 of 7
```

```

mapDown!(t,12,_rem)
--E 6

--S 7 of 7
leaves %
--E 7

)spool
)lisp (bye)

```

— **BalancedBinaryTree.help** —

=====

BalancedBinaryTree examples

=====

BalancedBinaryTree(S) is the domain of balanced binary trees (bbtree). A balanced binary tree of 2^k leaves, for some $k > 0$, is symmetric, that is, the left and right subtree of each interior node have identical shape. In general, the left and right subtree of a given node can differ by at most leaf node.

BalancedBinaryTrees(S) is the domain of balanced binary trees with elements of type S at the nodes. A binary tree is either empty or else consists of a node having a value and two branches, each branch a binary tree. A balanced binary tree is one that is balanced with respect its leaves. One with 2^k leaves is perfectly "balanced": the tree has minimum depth, and the left and right branch of every interior node is identical in shape.

Balanced binary trees are useful in algebraic computation for so-called "divide-and-conquer" algorithms. Conceptually, the data for a problem is initially placed at the root of the tree. The original data is then split into two subproblems, one for each subtree. And so on. Eventually, the problem is solved at the leaves of the tree. A solution to the original problem is obtained by some mechanism that can reassemble the pieces. In fact, an implementation of the Chinese Remainder Algorithm using balanced binary trees was first proposed by David Y. Y. Yun at the IBM T. J. Watson Research Center in Yorktown Heights, New York, in 1978. It served as the prototype for polymorphic algorithms in Axiom.

In what follows, rather than perform a series of computations with a single expression, the expression is reduced modulo a number of integer primes, a computation is done with modular arithmetic for each prime, and the Chinese Remainder Algorithm is used to obtain the answer to the original problem. We illustrate this principle with the

computation of $12^2 = 144$.

A list of moduli:

```
lm := [3,5,7,11]
      [3,5,7,11]
                                     Type: PositiveInteger
```

The expression `modTree(n, lm)` creates a balanced binary tree with leaf values $n \bmod m$ for each modulus m in `lm`.

```
modTree(12,lm)
[0, 2, 5, 1]
                                     Type: List Integer
```

Operation `modTree` does this using operations on balanced binary trees. We trace its steps. Create a balanced binary tree `t` of zeros with four leaves.

```
t := balancedBinaryTree(#lm, 0)
    [[0, 0, 0], 0, [0, 0, 0]]
                                     Type: BalancedBinaryTree NonNegativeInteger
```

The leaves of the tree are set to the individual moduli.

```
setleaves!(t,lm)
[[3, 0, 5], 0, [7, 0, 11]]
                                     Type: BalancedBinaryTree NonNegativeInteger
```

`mapUp!` to do a bottom-up traversal of `t`, setting each interior node to the product of the values at the nodes of its children.

```
mapUp!(t,_)
1155
                                     Type: PositiveInteger
```

The value at the node of every subtree is the product of the moduli of the leaves of the subtree.

```
t
[[3, 15, 5], 1155, [7, 77, 11]]
                                     Type: BalancedBinaryTree NonNegativeInteger
```

Operation `mapDown!(t,a,fn)` replaces the value `v` at each node of `t` by `fn(a,v)`.

```
mapDown!(t,12,_rem)
[[0, 12, 2], 12, [5, 12, 1]]
                                     Type: BalancedBinaryTree NonNegativeInteger
```

The operation `leaves` returns the leaves of the resulting tree. In this case, it returns the list of $12 \bmod m$ for each modulus m .

```
leaves %
[0, 2, 5, 1]
Type: List NonNegativeInteger
```

Compute the square of the images of 12 modulo each m .

```
squares := [x**2 rem m for x in % for m in lm]
[0, 4, 4, 1]
Type: List NonNegativeInteger
```

Call the Chinese Remainder Algorithm to get the answer for 12^2 .

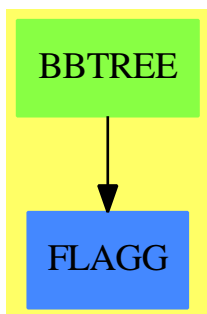
```
chineseRemainder(%,lm)
144
Type: PositiveInteger
```

See Also:

o `)show BalancedBinaryTree`

—————

BalancedBinaryTree (BBTREE)



See

- ⇒ “Tree” (TREE) [21](#) on page [2740](#)
- ⇒ “BinaryTree” (BTREE) [3](#) on page [310](#)
- ⇒ “BinarySearchTree” (BSTREE) [3](#) on page [302](#)
- ⇒ “BinaryTournament” (BTOURN) [3](#) on page [306](#)
- ⇒ “PendantTree” (PENDTREE) [17](#) on page [1944](#)

Exports:

any?	balancedBinaryTree	child?	children	coerce
copy	count	cyclic?	distance	empty
empty?	eq?	eval	every?	hash
latex	leaf?	leaves	left	less?
map	map!	mapDown!	mapUp!	member?
members	more?	node?	node	nodes
parts	right	sample	setchildren!	setelt
setleaves!	setleft!	setright!	setvalue!	size?
value	#?	?=?	?~=?	?..right
?..left	?..value			

— domain BBTREE BalancedBinaryTree —

```

)abbrev domain BBTREE BalancedBinaryTree
++ Description:
++ \spadtype{BalancedBinaryTree(S)} is the domain of balanced
++ binary trees (bbtree). A balanced binary tree of \spad{2**k} leaves,
++ for some \spad{k > 0}, is symmetric, that is, the left and right
++ subtree of each interior node have identical shape.
++ In general, the left and right subtree of a given node can differ
++ by at most leaf node.

BalancedBinaryTree(S: SetCategory): Exports == Implementation where
  Exports == BinaryTreeCategory(S) with
    finiteAggregate
    shallowlyMutable
-- BUG: applies wrong fnct for balancedBinaryTree(0,[1,2,3,4])
--   balancedBinaryTree: (S, List S) -> %
--     ++ balancedBinaryTree(s, ls) creates a balanced binary tree with
--     ++ s at the interior nodes and elements of ls at the
--     ++ leaves.
balancedBinaryTree: (NonNegativeInteger, S) -> %
  ++ balancedBinaryTree(n, s) creates a balanced binary tree with
  ++ n nodes each with value s.
  ++
  ++X balancedBinaryTree(4, 0)

setleaves_!: (% , List S) -> %
  ++ setleaves!(t, ls) sets the leaves of t in left-to-right order
  ++ to the elements of ls.
  ++
  ++X t1:=balancedBinaryTree(4, 0)
  ++X setleaves!(t1,[1,2,3,4])

mapUp_!: (% , (S,S) -> S) -> S
  ++ mapUp!(t,f) traverses balanced binary tree t in an "endorder"
  ++ (left then right then node) fashion returning t with the value
  ++ at each successive interior node of t replaced by
  ++ f(l,r) where l and r are the values at the immediate

```



```

++ left and right nodes.
++
++X T1:=BalancedBinaryTree Integer
++X t2:=balancedBinaryTree(4, 0)$T1
++X setleaves!(t2,[1,2,3,4]::List(Integer))
++X adder(a:Integer,b:Integer):Integer == a+b
++X mapUp!(t2,adder)
++X t2

mapUp_!: (% , % , (S,S,S,S) -> S) -> %
++ mapUp!(t,t1,f) traverses balanced binary tree t in an "endorder"
++ (left then right then node) fashion returning t with the value
++ at each successive interior node of t replaced by
++ f(l,r,l1,r1) where l and r are the values at the immediate
++ left and right nodes. Values l1 and r1 are values at the
++ corresponding nodes of a balanced binary tree t1, of identical
++ shape at t.
++
++X T1:=BalancedBinaryTree Integer
++X t2:=balancedBinaryTree(4, 0)$T1
++X setleaves!(t2,[1,2,3,4]::List(Integer))
++X adder4(i:INT,j:INT,k:INT,l:INT):INT == i+j+k+l
++X mapUp!(t2,t2,adder4)
++X t2

mapDown_!: (% ,S,(S,S,S) -> S) -> %
++ mapDown!(t,p,f) returns t after traversing t in "preorder"
++ (node then left then right) fashion replacing the successive
++ interior nodes as follows. The root value x is
++ replaced by q := f(p,x). The mapDown!(l,q,f) and
++ mapDown!(r,q,f) are evaluated for the left and right subtrees
++ l and r of t.
++
++X T1:=BalancedBinaryTree Integer
++X t2:=balancedBinaryTree(4, 0)$T1
++X setleaves!(t2,[1,2,3,4]::List(Integer))
++X adder(i:Integer,j:Integer):Integer == i+j
++X mapDown!(t2,4::INT,adder)
++X t2

mapDown_!: (% ,S, (S,S,S) -> List S) -> %
++ mapDown!(t,p,f) returns t after traversing t in "preorder"
++ (node then left then right) fashion replacing the successive
++ interior nodes as follows. Let l and r denote the left and
++ right subtrees of t. The root value x of t is replaced by p.
++ Then f(value l, value r, p), where l and r denote the left
++ and right subtrees of t, is evaluated producing two values
++ pl and pr. Then \spad{mapDown!(l,pl,f)} and \spad{mapDown!(l,pr,f)}
++ are evaluated.
++

```

```

++X T1:=BalancedBinaryTree Integer
++X t2:=balancedBinaryTree(4, 0)$T1
++X setleaves!(t2,[1,2,3,4]::List(Integer))
++X adder3(i:Integer,j:Integer,k:Integer):List Integer == [i+j,j+k]
++X mapDown!(t2,4::INT,adder3)
++X t2

Implementation == BinaryTree(S) add
Rep := BinaryTree(S)
leaf? x ==
  empty? x => false
  empty? left x and empty? right x
-- balancedBinaryTree(x: S, u: List S) ==
--   n := #u
--   n = 0 => empty()
--   setleaves_!(balancedBinaryTree(n, x), u)
setleaves_!(t, u) ==
  n := #u
  n = 0 =>
    empty? t => t
    error "the tree and list must have the same number of elements"
  n = 1 =>
    setvalue_!(t,first u)
    t
  m := n quo 2
  acc := empty()$(List S)
  for i in 1..m repeat
    acc := [first u,acc]
    u := rest u
  setleaves_!(left t, reverse_! acc)
  setleaves_!(right t, u)
  t
balancedBinaryTree(n: NonNegativeInteger, val: S) ==
  n = 0 => empty()
  n = 1 => node(empty(),val,empty())
  m := n quo 2
  node(balancedBinaryTree(m, val), val,
    balancedBinaryTree((n - m) pretend NonNegativeInteger, val))
mapUp_!(x,fn) ==
  empty? x => error "mapUp! called on a null tree"
  leaf? x => x.value
  x.value := fn(mapUp_!(x.left,fn),mapUp_!(x.right,fn))
mapUp_!(x,y,fn) ==
  empty? x => error "mapUp! is called on a null tree"
  leaf? x =>
    leaf? y => x
    error "balanced binary trees are incompatible"
  leaf? y => error "balanced binary trees are incompatible"
  mapUp_!(x.left,y.left,fn)
  mapUp_!(x.right,y.right,fn)

```

```

    x.value := fn(x.left.value,x.right.value,y.left.value,y.right.value)
  x
mapDown_!(x: %, p: S, fn: (S,S) -> S ) ==
  empty? x => x
  x.value := fn(p, x.value)
  mapDown_!(x.left, x.value, fn)
  mapDown_!(x.right, x.value, fn)
  x
mapDown_!(x: %, p: S, fn: (S,S,S) -> List S) ==
  empty? x => x
  x.value := p
  leaf? x => x
  u := fn(x.left.value, x.right.value, p)
  mapDown_!(x.left, u.1, fn)
  mapDown_!(x.right, u.2, fn)
  x

```

— BBTREE.dotabb —

```

"BBTREE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=BBTREE"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"BBTREE" -> "FLAGG"

```

domain BPADIC BalancedPAdicInteger

— BalancedPAdicInteger.input —

```

)set break resume
)sys rm -f BalancedPAdicInteger.output
)spool BalancedPAdicInteger.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show BalancedPAdicInteger
--R
--R BalancedPAdicInteger(p: Integer) is a domain constructor
--R Abbreviation for BalancedPAdicInteger is BPADIC
--R This constructor is not exposed in this frame.

```

```

--R Issue )edit bookvol10.3.pamphlet to see algebra source code for BPADIC
--R
--R----- Operations -----
--R ??? : (%,% ) -> %               ??? : (Integer,% ) -> %
--R ??? : (PositiveInteger,% ) -> %   ??? : (% ,PositiveInteger) -> %
--R ??+ : (%,% ) -> %               ?-? : (%,% ) -> %
--R -? : % -> %                     ?=? : (%,% ) -> Boolean
--R 1 : () -> %                     0 : () -> %
--R ?? : (% ,PositiveInteger) -> %   associates? : (%,% ) -> Boolean
--R coerce : % -> %                 coerce : Integer -> %
--R coerce : % -> OutputForm         complete : % -> %
--R digits : % -> Stream(Integer)    extend : (% ,Integer) -> %
--R gcd : List(% ) -> %              gcd : (%,% ) -> %
--R hash : % -> SingleInteger        latex : % -> String
--R lcm : List(% ) -> %              lcm : (%,% ) -> %
--R moduloP : % -> Integer           modulus : () -> Integer
--R one? : % -> Boolean              order : % -> NonNegativeInteger
--R ?quo? : (%,% ) -> %              quotientByP : % -> %
--R recip : % -> Union(% ,"failed")  ?rem? : (%,% ) -> %
--R sample : () -> %                 sizeLess? : (%,% ) -> Boolean
--R sqrt : (% ,Integer) -> %         unit? : % -> Boolean
--R unitCanonical : % -> %           zero? : % -> Boolean
--R ?~=? : (%,% ) -> Boolean
--R ??? : (NonNegativeInteger,% ) -> %
--R ??? : (% ,NonNegativeInteger) -> %
--R ??^ : (% ,NonNegativeInteger) -> %
--R approximate : (% ,Integer) -> Integer
--R characteristic : () -> NonNegativeInteger
--R divide : (%,% ) -> Record(quotient: %,remainder: %)
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(% ),%) -> Union(List(% ),"failed")
--R exquo : (%,% ) -> Union(% ,"failed")
--R extendedEuclidean : (%,% ,%) -> Union(Record(coef1: %,coef2: %),"failed")
--R extendedEuclidean : (%,% ) -> Record(coef1: %,coef2: %,generator: %)
--R gcdPolynomial : (SparseUnivariatePolynomial(% ),SparseUnivariatePolynomial(% )) -> SparseUnivariatePolynomial(% )
--R multiEuclidean : (List(% ),%) -> Union(List(% ),"failed")
--R principalIdeal : List(% ) -> Record(coef: List(% ),generator: %)
--R root : (SparseUnivariatePolynomial(Integer),Integer) -> %
--R subtractIfCan : (%,% ) -> Union(% ,"failed")
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R
--E 1

)spool
)lisp (bye)

```

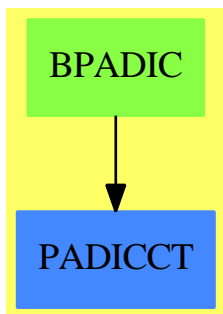
```
=====
BalancedPAdicInteger examples
=====
```

Stream-based implementation of \mathbb{Z}_p : p -adic numbers are represented as $\sum_{i=0}^{\infty} a[i] \cdot p^i$, where the $a[i]$ lie in $-(p-1)/2, \dots, (p-1)/2$.

See Also:

```
o )show BalancedPAdicInteger
```

BalancedPAdicInteger (BPADIC)



See

- ⇒ “InnerPAdicInteger” (IPADIC) [10](#) on page [1294](#)
- ⇒ “PAdicInteger” (PADIC) [17](#) on page [1881](#)
- ⇒ “PAdicRationalConstructor” (PADICRC) [17](#) on page [1890](#)
- ⇒ “PAdicRational” (PADICRAT) [17](#) on page [1885](#)
- ⇒ “BalancedPAdicRational” (BPADICRT) [3](#) on page [262](#)

Exports:

0	1	approximate	associates?
characteristic	coerce	complete	digits
divide	euclideanSize	expressIdealMember	exquo
extend	extendedEuclidean	gcd	gcdPolynomial
hash	latex	lcm	moduloP
modulus	multiEuclidean	one?	order
principalIdeal	quotientByP	recip	root
sample	sizeLess?	sqrt	subtractIfCan
unit?	unitCanonical	unitNormal	zero?
?^=?	?*?	?**?	?^?
?+?	?-?	-?	?=?
?quo?	?rem?		

— domain BPADIC BalancedPAdicInteger —

```
)abbrev domain BPADIC BalancedPAdicInteger
++ Author: Clifton J. Williamson
++ Date Created: 15 May 1990
++ Date Last Updated: 15 May 1990
++ Description:
++ Stream-based implementation of  $\mathbb{Z}_p$ : p-adic numbers are represented as
++  $\sum_{i=0}^{\infty} a[i] \cdot p^i$ , where the  $a[i]$  lie in  $-(p-1)/2, \dots, (p-1)/2$ .

BalancedPAdicInteger(p:Integer) == InnerPAdicInteger(p,false$Boolean)
```

—————

— BPADIC.dotabb —

```
"BPADIC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=BPADIC"]
"PADICCT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PADICCT"]
"BPADIC" -> "PADICCT"
```

—————

domain BPADICRT BalancedPAdicRational

— BalancedPAdicRational.input —

```
)set break resume
)sys rm -f BalancedPAdicRational.output
)spool BalancedPAdicRational.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show BalancedPAdicRational
--R
--R BalancedPAdicRational(p: Integer) is a domain constructor
--R Abbreviation for BalancedPAdicRational is BPADICRT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for BPADICRT
--R
--R----- Operations -----
--R ??? : (Fraction(Integer),%) -> %      ??? : (%,Fraction(Integer)) -> %
```

```

--R ?? : (% , %) -> %
--R ?? : (PositiveInteger , %) -> %
--R ??? : (% , PositiveInteger) -> %
--R ?-? : (% , %) -> %
--R ?/? : (% , %) -> %
--R 1 : () -> %
--R ?^? : (% , Integer) -> %
--R associates? : (% , %) -> Boolean
--R coerce : % -> %
--R coerce : % -> OutputForm
--R factor : % -> Factored(%)
--R gcd : (% , %) -> %
--R inv : % -> %
--R lcm : List(%) -> %
--R numerator : % -> %
--R prime? : % -> Boolean
--R recip : % -> Union(%, "failed")
--R removeZeroes : (Integer , %) -> %
--R sample : () -> %
--R squareFree : % -> Factored(%)
--R unit? : % -> Boolean
--R zero? : % -> Boolean
--R ?? : (% , BalancedPAdicInteger(p)) -> %
--R ?? : (BbalancedPAdicInteger(p) , %) -> %
--R ?? : (NonNegativeInteger , %) -> %
--R ??? : (% , NonNegativeInteger) -> %
--R ?/? : (BbalancedPAdicInteger(p) , BbalancedPAdicInteger(p)) -> %
--R ?<? : (% , %) -> Boolean if BbalancedPAdicInteger(p) has ORDSET
--R ?<=? : (% , %) -> Boolean if BbalancedPAdicInteger(p) has ORDSET
--R ?>? : (% , %) -> Boolean if BbalancedPAdicInteger(p) has ORDSET
--R ?>=? : (% , %) -> Boolean if BbalancedPAdicInteger(p) has ORDSET
--R D : (% , (BbalancedPAdicInteger(p) -> BbalancedPAdicInteger(p))) -> %
--R D : (% , (BbalancedPAdicInteger(p) -> BbalancedPAdicInteger(p)) , NonNegativeInteger) -> %
--R D : (% , List(Symbol) , List(NonNegativeInteger)) -> % if BbalancedPAdicInteger(p) has PDRING(SYM
--R D : (% , Symbol , NonNegativeInteger) -> % if BbalancedPAdicInteger(p) has PDRING(SYMBOL)
--R D : (% , List(Symbol)) -> % if BbalancedPAdicInteger(p) has PDRING(SYMBOL)
--R D : (% , Symbol) -> % if BbalancedPAdicInteger(p) has PDRING(SYMBOL)
--R D : (% , NonNegativeInteger) -> % if BbalancedPAdicInteger(p) has DIFRING
--R D : % -> % if BbalancedPAdicInteger(p) has DIFRING
--R ?^? : (% , NonNegativeInteger) -> %
--R abs : % -> % if BbalancedPAdicInteger(p) has OINTDOM
--R approximate : (% , Integer) -> Fraction(Integer)
--R ceiling : % -> BbalancedPAdicInteger(p) if BbalancedPAdicInteger(p) has INS
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if $ has CHARNZ and BbalancedPAdicInteger(p) has PFECAT o
--R coerce : Symbol -> % if BbalancedPAdicInteger(p) has RETRACT(SYMBOL)
--R coerce : BbalancedPAdicInteger(p) -> %
--R conditionP : Matrix(%) -> Union(Vector(%) , "failed") if $ has CHARNZ and BbalancedPAdicInteger
--R continuedFraction : % -> ContinuedFraction(Fraction(Integer))
--R convert : % -> DoubleFloat if BbalancedPAdicInteger(p) has REAL
?? : (Integer , %) -> %
??? : (% , Integer) -> %
?+? : (% , %) -> %
-? : % -> %
?=? : (% , %) -> Boolean
0 : () -> %
?^? : (% , PositiveInteger) -> %
coerce : Fraction(Integer) -> %
coerce : Integer -> %
denominator : % -> %
gcd : List(%) -> %
hash : % -> SingleInteger
latex : % -> String
lcm : (% , %) -> %
one? : % -> Boolean
?quo? : (% , %) -> %
?rem? : (% , %) -> %
removeZeroes : % -> %
sizeLess? : (% , %) -> Boolean
squareFreePart : % -> %
unitCanonical : % -> %
?~=? : (% , %) -> Boolean

```

```

--R convert : % -> Float if BalancedPAdicInteger(p) has REAL
--R convert : % -> InputForm if BalancedPAdicInteger(p) has KONVERT(INFORM)
--R convert : % -> Pattern(Float) if BalancedPAdicInteger(p) has KONVERT(PATTERN(FLOAT))
--R convert : % -> Pattern(Integer) if BalancedPAdicInteger(p) has KONVERT(PATTERN(INT))
--R denom : % -> BalancedPAdicInteger(p)
--R differentiate : (%,(BalancedPAdicInteger(p) -> BalancedPAdicInteger(p))) -> %
--R differentiate : (%,(BalancedPAdicInteger(p) -> BalancedPAdicInteger(p)),NonNegativeInteger) -> %
--R differentiate : (%,List(Symbol),List(NonNegativeInteger)) -> % if BalancedPAdicInteger(p) has PDRING(SYMBOL)
--R differentiate : (%,Symbol,NonNegativeInteger) -> % if BalancedPAdicInteger(p) has PDRING(SYMBOL)
--R differentiate : (%,List(Symbol)) -> % if BalancedPAdicInteger(p) has PDRING(SYMBOL)
--R differentiate : (%,Symbol) -> % if BalancedPAdicInteger(p) has PDRING(SYMBOL)
--R differentiate : (%,NonNegativeInteger) -> % if BalancedPAdicInteger(p) has DIFRING
--R differentiate : % -> % if BalancedPAdicInteger(p) has DIFRING
--R divide : (%,%) -> Record(quotient: %,remainder: %)
--R ?? : (%,BalancedPAdicInteger(p)) -> % if BalancedPAdicInteger(p) has ELTAB(BPADIC(p),BPADIC(p))
--R euclideanSize : % -> NonNegativeInteger
--R eval : (%,Symbol,BalancedPAdicInteger(p)) -> % if BalancedPAdicInteger(p) has IEVALAB(SYMBOL,BPADIC(p))
--R eval : (%,List(Symbol),List(BalancedPAdicInteger(p))) -> % if BalancedPAdicInteger(p) has IEVALAB(SYMBOL)
--R eval : (%,List(Equation(BalancedPAdicInteger(p)))) -> % if BalancedPAdicInteger(p) has EVALAB(BPADIC(p))
--R eval : (%,Equation(BalancedPAdicInteger(p))) -> % if BalancedPAdicInteger(p) has EVALAB(BPADIC(p))
--R eval : (%,BalancedPAdicInteger(p),BalancedPAdicInteger(p)) -> % if BalancedPAdicInteger(p) has EVALAB(BPADIC(p))
--R eval : (%,List(BalancedPAdicInteger(p)),List(BalancedPAdicInteger(p))) -> % if BalancedPAdicInteger(p) has EVALAB(BPADIC(p))
--R expressIdealMember : (List(%),%) -> Union(List(%),"failed")
--R exquo : (%,%) -> Union(%,"failed")
--R extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %),"failed")
--R extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %)
--R factorPolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if BalancedPAdicInteger(p) has INS
--R factorSquareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if BalancedPAdicInteger(p) has INS
--R floor : % -> BalancedPAdicInteger(p) if BalancedPAdicInteger(p) has INS
--R fractionPart : % -> % if BalancedPAdicInteger(p) has EUCDOM
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%) if BalancedPAdicInteger(p) has INS
--R init : () -> % if BalancedPAdicInteger(p) has STEP
--R map : ((BalancedPAdicInteger(p) -> BalancedPAdicInteger(p)),%) -> %
--R max : (%,%) -> % if BalancedPAdicInteger(p) has ORDSET
--R min : (%,%) -> % if BalancedPAdicInteger(p) has ORDSET
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed")
--R negative? : % -> Boolean if BalancedPAdicInteger(p) has OINTDOM
--R nextItem : % -> Union(%,"failed") if BalancedPAdicInteger(p) has STEP
--R numer : % -> BalancedPAdicInteger(p)
--R patternMatch : (%,Pattern(Float),PatternMatchResult(Float,%)) -> PatternMatchResult(Float,%) if BalancedPAdicInteger(p) has INS
--R patternMatch : (%,Pattern(Integer),PatternMatchResult(Integer,%)) -> PatternMatchResult(Integer,%) if BalancedPAdicInteger(p) has INS
--R positive? : % -> Boolean if BalancedPAdicInteger(p) has OINTDOM
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %)
--R random : () -> % if BalancedPAdicInteger(p) has INS
--R reducedSystem : Matrix(%) -> Matrix(BalancedPAdicInteger(p))
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(BalancedPAdicInteger(p)),vec: Vector(BalancedPAdicInteger(p))) if BalancedPAdicInteger(p) has INS
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) if BalancedPAdicInteger(p) has INS
--R reducedSystem : Matrix(%) -> Matrix(Integer) if BalancedPAdicInteger(p) has LINEXP(INT)
--R retract : % -> Integer if BalancedPAdicInteger(p) has RETRACT(INT)
--R retract : % -> Fraction(Integer) if BalancedPAdicInteger(p) has RETRACT(INT)

```



```

--R retract : % -> Symbol if BalancedPAdicInteger(p) has RETRACT(SYMBOL)
--R retract : % -> BalancedPAdicInteger(p)
--R retractIfCan : % -> Union(Integer,"failed") if BalancedPAdicInteger(p) has RETRACT(INT)
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if BalancedPAdicInteger(p) has RETRACT(FRAC)
--R retractIfCan : % -> Union(Symbol,"failed") if BalancedPAdicInteger(p) has RETRACT(SYMBOL)
--R retractIfCan : % -> Union(BalancedPAdicInteger(p),"failed")
--R sign : % -> Integer if BalancedPAdicInteger(p) has OINTDOM
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(%)),SparseUnivariatePolynomial(%)) -> Union(Integer,"failed")
--R squareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%))
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R wholePart : % -> BalancedPAdicInteger(p) if BalancedPAdicInteger(p) has EUCDOM
--R
--E 1

)spool
)lisp (bye)

```

— **BalancedPAdicRational.help** —

=====

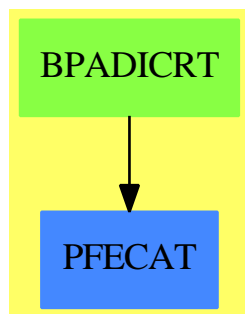
BalancedPAdicRational examples

=====

Stream-based implementation of \mathbb{Q}_p : numbers are represented as
 $\text{sum}(i = k.., a[i] * p^i)$, where the $a[i]$ lie in $-(p-1)/2, \dots, (p-1)/2$.

See Also:
o)show BalancedPAdicRational

BalancedPAdicRational (BPADICRT)



See

- ⇒ “InnerPAdicInteger” (IPADIC) [10](#) on page [1294](#)
- ⇒ “PAdicInteger” (PADIC) [17](#) on page [1881](#)
- ⇒ “BalancedPAdicInteger” (BPADIC) [3](#) on page [258](#)
- ⇒ “PAdicRationalConstructor” (PADICRC) [17](#) on page [1890](#)
- ⇒ “PAdicRational” (PADICRAT) [17](#) on page [1885](#)

Exports:

0	1	abs
approximate	associates?	ceiling
characteristic	charthRoot	coerce
conditionP	continuedFraction	convert
D	denom	denominator
differentiate	divide	??
euclideanSize	eval	expressIdealMember
exquo	extendedEuclidean	factor
factorPolynomial	factorSquareFreePolynomial	floor
fractionPart	gcd	gcdPolynomial
hash	init	inv
latex	lcm	map
max	min	multiEuclidean
negative?	nextItem	numer
numerator	one?	patternMatch
positive?	prime?	principalIdeal
random	recip	reducedSystem
removeZeroes	retract	retractIfCan
sample	sign	sizeLess?
solveLinearPolynomialEquation	squareFree	squareFreePart
squareFreePolynomial	subtractIfCan	unit?
unitCanonical	unitNormal	wholePart
zero?	?*?	?**?
?+?	?-?	-?
?/?	?=?	?^?
?~=?	?<?	?<=?
?>?	?>=?	?quo?
?rem?		

— domain BPADICRT BalancedPAdicRational —

```
)abbrev domain BPADICRT BalancedPAdicRational
++ Author: Clifton J. Williamson
++ Date Created: 15 May 1990
++ Date Last Updated: 15 May 1990
++ Description:
++ Stream-based implementation of Qp: numbers are represented as
++ sum(i = k.., a[i] * p^i), where the a[i] lie in -(p - 1)/2,...,(p - 1)/2.
```

```
BalancedPAdicRational(p:Integer) ==
  PAdicRationalConstructor(p,BalancedPAdicInteger p)
```

— BPADICRT.dotabb —

```
"BPADICRT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=BPADICRT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"BPADICRT" -> "PFECAT"
```

domain BFUNCT BasicFunctions

— BasicFunctions.input —

```
)set break resume
)sys rm -f BasicFunctions.output
)spool BasicFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show BasicFunctions
--R
--R BasicFunctions is a domain constructor
--R Abbreviation for BasicFunctions is BFUNCT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for BFUNCT
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          bfKeys : () -> List(Symbol)
--R coerce : % -> OutputForm        hash : % -> SingleInteger
--R latex : % -> String             ?~=? : (%,% ) -> Boolean
--R bfEntry : Symbol -> Record(zeros: Stream(DoubleFloat),ones: Stream(DoubleFloat),singularities: Stream(DoubleFloat))
--R
--E 1

)spool
)lisp (bye)
```

— BasicFunctions.help —

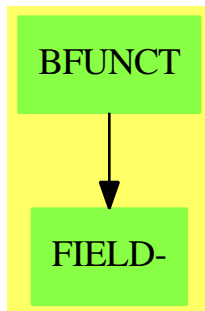
```
=====
BasicFunctions examples
=====
```

A Domain which implements a table containing details of points at which particular functions have evaluation problems.

See Also:

o)show BasicFunctions

BasicFunctions (BFUNCT)



Exports:

```
bfEntry  bfKeys  coerce  hash  latex
?~=?     ?=?
```

— domain BFUNCT BasicFunctions —

```
)abbrev domain BFUNCT BasicFunctions
++ Author: Brian Dupee
++ Date Created: August 1994
++ Date Last Updated: April 1996
++ Description:
++ A Domain which implements a table containing details of
++ points at which particular functions have evaluation problems.
```

```
BasicFunctions(): E == I where
  DF ==> DoubleFloat
```

```

SDF ==> Stream DoubleFloat
RS ==> Record(zeros: SDF, ones: SDF, singularities: SDF)

E ==> SetCategory with
  bfKeys:() -> List Symbol
    ++ bfKeys() returns the names of each function in the
    ++ \axiomType{BasicFunctions} table
  bfEntry:Symbol -> RS
    ++ bfEntry(k) returns the entry in the \axiomType{BasicFunctions} table
    ++ corresponding to \spad{k}
  finiteAggregate

I ==> add

Rep := Table(Symbol,RS)
import Rep, SDF

f(x:DF):DF ==
  positive?(x) => -x
  -x+1

bf():$ ==
  import RS
  dpi := pi()$DF
  ndpi:SDF := map(x1+-->x1*dpi,(z := generate(f,0))) -- [n pi for n in Z]
  n1dpi:SDF := map(x1+-->-(2*(x1)-1)*dpi/2,z) -- [(n+1) pi /2]
  n2dpi:SDF := map(x1+-->2*x1*dpi,z) -- [2 n pi for n in Z]
  n3dpi:SDF := map(x1+-->-(4*(x1)-1)*dpi/4,z)
  n4dpi:SDF := map(x1+-->-(4*(x1)-1)*dpi/2,z)
  sinEntry:RS := [ndpi, n4dpi, empty()$SDF]
  cosEntry:RS := [n1dpi, n2dpi, esdf := empty()$SDF]
  tanEntry:RS := [ndpi, n3dpi, n1dpi]
  asinEntry:RS := [construct([0$DF])$SDF,
    construct([float(8414709848078965,-16,10)$DF]), esdf]
  acosEntry:RS := [construct([1$DF])$SDF,
    construct([float(54030230586813977,-17,10)$DF]), esdf]
  atanEntry:RS := [construct([0$DF])$SDF,
    construct([float(15574077246549023,-16,10)$DF]), esdf]
  secEntry:RS := [esdf, n2dpi, n1dpi]
  cscEntry:RS := [esdf, n4dpi, ndpi]
  cotEntry:RS := [n1dpi, n3dpi, ndpi]
  logEntry:RS := [construct([1$DF])$SDF,esdf, construct([0$DF])$SDF]
  entryList:List(Record(key:Symbol,entry:RS)) :=
    [[sin@Symbol, sinEntry], [cos@Symbol, cosEntry],
     [tan@Symbol, tanEntry], [sec@Symbol, secEntry],
     [csc@Symbol, cscEntry], [cot@Symbol, cotEntry],
     [asin@Symbol, asinEntry], [acos@Symbol, acosEntry],
     [atan@Symbol, atanEntry], [log@Symbol, logEntry]]
  construct(entryList)$Rep

```

```

bfKeys():List Symbol == keys(bf())$Rep

bfEntry(k:Symbol):RS == qelt(bf(),k)$Rep

```

— **BFUNCT.dotabb** —

```

"BFUNCT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=BFUNCT"]
"FIELD-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FIELD"]
"BFUNCT" -> "FIELD-"

```

domain BOP BasicOperator

— **BasicOperator.input** —

```

)set break resume
)sys rm -f BasicOperator.output
)spool BasicOperator.output
)set message test on
)set message auto off
)clear all
--S 1 of 18
y := operator 'y
--R
--R
--R (1) y
--R
--R                                          Type: BasicOperator
--E 1

--S 2 of 18
deq := D(y x, x, 2) + D(y x, x) + y x = 0
--R
--R
--R      , ,
--R (2) y '(x) + y '(x) + y(x)= 0
--R
--R                                          Type: Equation(Expression(Integer))
--E 2

--S 3 of 18
solve(deq, y, x)

```

```

--R
--R
--R
--R      x      x
--R      +-+  - -  - -  +-+
--R      x\|3  2    2    x\|3
--R      (3) [particular= 0,basis= [cos(-----)%e ,%e sin(-----)]]
--R      2                                2
--RType: Union(Record(particular: Expression(Integer),basis: ...
--E 3

--S 4 of 18
nary? y
--R
--R
--R      (4) true
--R
--RType: Boolean
--E 4

--S 5 of 18
unary? y
--R
--R
--R      (5) false
--R
--RType: Boolean
--E 5

--S 6 of 18
op0ne := operator('op0ne, 1)
--R
--R
--R      (6) op0ne
--R
--RType: BasicOperator
--E 6

--S 7 of 18
nary? op0ne
--R
--R
--R      (7) false
--R
--RType: Boolean
--E 7

--S 8 of 18
unary? op0ne
--R
--R
--R      (8) true
--R
--RType: Boolean
--E 8

```

```

--S 9 of 18
arity opOne
--R
--R
--R (9) 1
--R
--R                                         Type: Union(NonNegativeInteger,...)
--E 9

--S 10 of 18
name opOne
--R
--R
--R (10) opOne
--R
--R                                         Type: Symbol
--E 10

--S 11 of 18
is?(opOne, 'z2)
--R
--R
--R (11) false
--R
--R                                         Type: Boolean
--E 11

--S 12 of 18
is?(opOne, "opOne")
--R
--R
--R (12) true
--R
--R                                         Type: Boolean
--E 12

--S 13 of 18
properties y
--R
--R
--R (13) table()
--R
--R                                         Type: AssociationList(String,None)
--E 13

--S 14 of 18
setProperty(y, "use", "unknown function" :: None )
--R
--R
--R (14) y
--R
--R                                         Type: BasicOperator
--E 14

--S 15 of 18
properties y

```



```

--R
--R
--R (15) table("use"= NONE)
--R
--R                                     Type: AssociationList(String,None)
--E 15

--S 16 of 18
property(y, "use") :: None pretend String
--R
--R
--R (16) "unknown function"
--R
--R                                     Type: String
--E 16

--S 17 of 18
deleteProperty!(y, "use")
--R
--R
--R (17) y
--R
--R                                     Type: BasicOperator
--E 17

--S 18 of 18
properties y
--R
--R
--R (18) table()
--R
--R                                     Type: AssociationList(String,None)
--E 18
)spool
)lisp (bye)

```

— BasicOperator.help —

=====

BasicOperator examples

=====

A basic operator is an object that can be symbolically applied to a list of arguments from a set, the result being a kernel over that set or an expression.

You create an object of type BasicOperator by using the operator operation. This first form of this operation has one argument and it must be a symbol. The symbol should be quoted in case the name has been used as an identifier to which a value has been assigned.

A frequent application of BasicOperator is the creation of an operator to represent the unknown function when solving a differential equation.

Let y be the unknown function in terms of x .

```
y := operator 'y
y
Type: BasicOperator
```

This is how you enter the equation $y'' + y' + y = 0$.

```
deq := D(y x, x, 2) + D(y x, x) + y x = 0
''
y (x) + y (x) + y(x) = 0
Type: Equation Expression Integer
```

To solve the above equation, enter this.

```
solve(deq, y, x)
      x      x
    +-+  - -  - -  +-+
    x\|3  2    2    x\|3
[particular= 0,basis= [cos(----)%e  ,%e  sin(----)]]
                    2          2
Type: Union(Record(particular: Expression Integer,
                    basis: List Expression Integer),...)
```

Use the single argument form of BasicOperator (as above) when you intend to use the operator to create functional expressions with an arbitrary number of arguments

Nary means an arbitrary number of arguments can be used in the functional expressions.

```
nary? y
true
Type: Boolean
```

```
unary? y
false
Type: Boolean
```

Use the two-argument form when you want to restrict the number of arguments in the functional expressions created with the operator.

This operator can only be used to create functional expressions with one argument.

```
opOne := operator('opOne, 1)
opOne
```

Type: BasicOperator

```
nary? opOne
false
```

Type: Boolean

```
unary? opOne
true
```

Type: Boolean

Use `arity` to learn the number of arguments that can be used. It returns "false" if the operator is nary.

```
arity opOne
1
```

Type: Union(NonNegativeInteger,...)

Use `name` to learn the name of an operator.

```
name opOne
opOne
```

Type: Symbol

Use `is?` to learn if an operator has a particular name.

```
is?(opOne, 'z2)
false
```

Type: Boolean

You can also use a string as the name to be tested against.

```
is?(opOne, "opOne")
true
```

Type: Boolean

You can attach named properties to an operator. These are rarely used at the top-level of the Axiom interactive environment but are used with Axiom library source code.

By default, an operator has no properties.

```
properties y
table()
```

Type: AssociationList(String,None)

The interface for setting and getting properties is somewhat awkward because the property values are stored as values of type `None`.

Attach a property by using `setProperty`.

```
setProperty(y, "use", "unknown function" :: None )
y
```

Type: BasicOperator

```
properties y
table("use"=NONE)
```

Type: AssociationList(String, None)

We know the property value has type String.

```
property(y, "use") :: None pretend String
"unknown function"
```

Type: String

Use deleteProperty! to destructively remove a property.

```
deleteProperty!(y, "use")
y
```

Type: BasicOperator

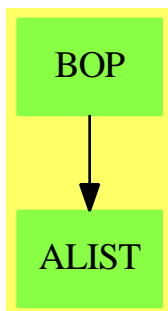
```
properties y
table()
```

Type: AssociationList(String, None)

See Also

- o)help Expression
- o)help Kernel
- o)show BasicOperator

BasicOperator (BOP)



Exports:

arity	assert	coerce	comparison	copy
deleteProperty!	display	display	display	equality
has?	hash	input	input	is?
latex	max	min	name	nary?
nullary?	operator	operator	properties	property
setProperty	setProperty	unary?	weight	weight
?^=?	?<?	?<=?	?=?	?>?
?>=?				

— domain BOP BasicOperator —

```

)abbrev domain BOP BasicOperator
++ Author: Manuel Bronstein
++ Date Created: 22 March 1988
++ Date Last Updated: 11 October 1993
++ Description:
++ Basic system operators.
++ A basic operator is an object that can be applied to a list of
++ arguments from a set, the result being a kernel over that set.

BasicOperator(): Exports == Implementation where
  O ==> OutputForm
  P ==> AssociationList(String, None)
  L ==> List Record(key:String, entry:None)
  SEX ==> InputForm
-- some internal properties
LESS? ==> "%less?"
EQUAL? ==> "%equal?"
WEIGHT ==> "%weight"
DISPLAY ==> "%display"
SEXPR ==> "%input"

Exports ==> OrderedSet with
  name : $ -> Symbol
    ++ name(op) returns the name of op.
  properties: $ -> P
    ++ properties(op) returns the list of all the properties
    ++ currently attached to op.
  copy : $ -> $
    ++ copy(op) returns a copy of op.
  operator : Symbol -> $
    ++ operator(f) makes f into an operator with arbitrary arity.
  operator : (Symbol, NonNegativeInteger) -> $
    ++ operator(f, n) makes f into an n-ary operator.
  arity : $ -> Union(NonNegativeInteger, "failed")
    ++ arity(op) returns n if op is n-ary, and
    ++ "failed" if op has arbitrary arity.
  nullary? : $ -> Boolean

```

```

    ++ nullary?(op) tests if op is nullary.
unary?    : $ -> Boolean
    ++ unary?(op) tests if op is unary.
nary?     : $ -> Boolean
    ++ nary?(op) tests if op has arbitrary arity.
weight    : $ -> NonNegativeInteger
    ++ weight(op) returns the weight attached to op.
weight    : ($, NonNegativeInteger) -> $
    ++ weight(op, n) attaches the weight n to op.
equality   : ($, ($, $) -> Boolean) -> $
    ++ equality(op, foo?) attaches foo? as the "%equal?" property
    ++ to op. If op1 and op2 have the same name, and one of them
    ++ has an "%equal?" property f, then \spad{f(op1, op2)} is called to
    ++ decide whether op1 and op2 should be considered equal.
comparison : ($, ($, $) -> Boolean) -> $
    ++ comparison(op, foo?) attaches foo? as the "%less?" property
    ++ to op. If op1 and op2 have the same name, and one of them
    ++ has a "%less?" property f, then \spad{f(op1, op2)} is called to
    ++ decide whether \spad{op1 < op2}.
display    : $ -> Union(List 0 -> 0, "failed")
    ++ display(op) returns the "%display" property of op if
    ++ it has one attached, and "failed" otherwise.
display    : ($, List 0 -> 0) -> $
    ++ display(op, foo) attaches foo as the "%display" property
    ++ of op. If op has a "%display" property f, then \spad{op(a1,...,an)}
    ++ gets converted to OutputForm as \spad{f(a1,...,an)}.
display    : ($, 0 -> 0) -> $
    ++ display(op, foo) attaches foo as the "%display" property
    ++ of op. If op has a "%display" property f, then \spad{op(a)}
    ++ gets converted to OutputForm as \spad{f(a)}.
    ++ Argument op must be unary.
input      : ($, List SEX -> SEX) -> $
    ++ input(op, foo) attaches foo as the "%input" property
    ++ of op. If op has a "%input" property f, then \spad{op(a1,...,an)}
    ++ gets converted to InputForm as \spad{f(a1,...,an)}.
input      : $ -> Union(List SEX -> SEX, "failed")
    ++ input(op) returns the "%input" property of op if
    ++ it has one attached, "failed" otherwise.
is?        : ($, Symbol) -> Boolean
    ++ is?(op, s) tests if the name of op is s.
has?       : ($, String) -> Boolean
    ++ has?(op, s) tests if property s is attached to op.
assert     : ($, String) -> $
    ++ assert(op, s) attaches property s to op.
    ++ Argument op is modified "in place", i.e. no copy is made.
deleteProperty_! : ($, String) -> $
    ++ deleteProperty!(op, s) unattaches property s from op.
    ++ Argument op is modified "in place", i.e. no copy is made.
property    : ($, String) -> Union(None, "failed")
    ++ property(op, s) returns the value of property s if

```

```

    ++ it is attached to op, and "failed" otherwise.
setProperty : ($, String, None) -> $
    ++ setProperty(op, s, v) attaches property s to op,
    ++ and sets its value to v.
    ++ Argument op is modified "in place", i.e. no copy is made.
setProperty : ($, P) -> $
    ++ setProperties(op, l) sets the property list of op to l.
    ++ Argument op is modified "in place", i.e. no copy is made.

Implementation ==> add
-- if nargs < 0 then the operator has variable arity.
Rep := Record(opname:Symbol, nargs:SingleInteger, props:P)

oper: (Symbol, SingleInteger, P) -> $

is?(op, s)          == name(op) = s
name op             == op.opname
properties op       == op.props
setProperty(op, l) == (op.props := l; op)
operator s          == oper(s, -1::SingleInteger, table())
operator(s, n)      == oper(s, n::Integer::SingleInteger, table())
property(op, name)  == search(name, op.props)
assert(op, s)       == setProperty(op, s, NIL$Lisp)
has?(op, name)      == key?(name, op.props)
oper(se, n, prop)   == [se, n, prop]
weight(op, n)       == setProperty(op, WEIGHT, n pretend None)
nullary? op         == zero?(op.nargs)
-- unary? op        == one?(op.nargs)
unary? op           == ((op.nargs) = 1)
nary? op            == negative?(op.nargs)
equality(op, func)  == setProperty(op, EQUAL?, func pretend None)
comparison(op, func) == setProperty(op, LESS?, func pretend None)
display(op:$, f:0 -> 0) == display(op, (x1:List(0)):0 +-> f first x1)
deleteProperty_!(op, name) == (remove_!(name, properties op); op)
setProperty(op, name, valu) == (op.props.name := valu; op)
coerce(op:$):OutputForm == name(op)::OutputForm
input(op:$, f:List SEX -> SEX) == setProperty(op, SEXPR, f pretend None)
display(op:$, f:List 0 -> 0) == setProperty(op, DISPLAY, f pretend None)

display op ==
    (u := property(op, DISPLAY)) case "failed" => "failed"
    (u::None) pretend (List 0 -> 0)

input op ==
    (u := property(op, SEXPR)) case "failed" => "failed"
    (u::None) pretend (List SEX -> SEX)

arity op ==
    negative?(n := op.nargs) => "failed"
    convert(n)@Integer :: NonNegativeInteger

```

```

copy op ==
  oper(name op, op.narg,
        table([[r.key, r.entry] for r in entries(properties op)@L]$L))

-- property EQUAL? contains a function f: (BOP, BOP) -> Boolean
-- such that f(o1, o2) is true iff o1 = o2
op1 = op2 ==
  (EQ$Lisp)(op1, op2) => true
  name(op1) ^= name(op2) => false
  op1.narg ^= op2.narg => false
  brace(keys properties op1)^=$Set(String) _
    brace(keys properties op2) => false
  (func := property(op1, EQUAL?)) case None =>
    ((func::None) pretend (($, $) -> Boolean)) (op1, op2)
  true

-- property WEIGHT allows one to change the ordering around
-- by default, every operator has weighth 1
weight op ==
  (w := property(op, WEIGHT)) case "failed" => 1
  (w::None) pretend NonNegativeInteger

-- property LESS? contains a function f: (BOP, BOP) -> Boolean
-- such that f(o1, o2) is true iff o1 < o2
op1 < op2 ==
  (w1 := weight op1) ^= (w2 := weight op2) => w1 < w2
  op1.narg ^= op2.narg => op1.narg < op2.narg
  name(op1) ^= name(op2) => name(op1) < name(op2)
  n1 := #(k1 := brace(keys(properties op1))$Set(String))
  n2 := #(k2 := brace(keys(properties op2))$Set(String))
  n1 ^= n2 => n1 < n2
  not zero?(n1 := #(d1 := difference(k1, k2))) =>
    n1 ^= (n2 := #(d2 := difference(k2, k1))) => n1 < n2
    inspect(d1) < inspect(d2)
  (func := property(op1, LESS?)) case None =>
    ((func::None) pretend (($, $) -> Boolean)) (op1, op2)
  (func := property(op1, EQUAL?)) case None =>
    not(((func::None) pretend (($, $) -> Boolean)) (op1, op2))
  false

```

— BOP.dotabb —

```

"BOP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=BOP"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"BOP" -> "ALIST"

```


domain BSD BasicStochasticDifferential

— BasicStochasticDifferential.input —

```
)set break resume
)sys rm -f BasicStochasticDifferential.output
)spool BasicStochasticDifferential.output
)set message test on
)set message auto off
)clear all

--S 1 of 30
print copyBSD()
--R
--R  []
--R
--R                                          Type: Void
--E 1

--S 2 of 30
print copyIto()
--R
--R  table()
--R
--R                                          Type: Void
--E 2

--S 3 of 30
dX:=introduce!(X,dX) -- dX is a new stochastic differential
--R
--R
--R  (3)  dX
--R
--R                                          Type: Union(BasicStochasticDifferential,...)
--E 3

--S 4 of 30
print copyBSD()
--R
--R  [dX]
--R
--R                                          Type: Void
--E 4

--S 5 of 30
print copyIto()
--R
```

```

--R   table(X= dX)
--R
--R                                          Type: Void
--E 5

--S 6 of 30
getSmgl(dX)
--R
--R
--R   (6)  X
--R
--R                                          Type: Union(Symbol,...)
--E 6

--S 7 of 30
print dX
--R
--R   dX
--R
--R                                          Type: Void
--E 7

--S 8 of 30
print domainOf(dX)
--R
--R   BasicStochasticDifferential()
--R
--R                                          Type: Void
--E 8

--S 9 of 30
print d X
--R
--R   dX
--R
--R                                          Type: Void
--E 9

--S 10 of 30
print ((d$BSD) X)
--R
--R   dX
--R
--R                                          Type: Void
--E 10

--S 11 of 30
print domainOf((d$BSD) X)
--R
--R   BasicStochasticDifferential()
--R
--R                                          Type: Void
--E 11

--S 12 of 30
introduce!(t,dt) -- dt is a new stochastic differential
--R

```

```

--R
--R (12) dt
--R                                         Type: Union(BasicStochasticDifferential,...)
--E 12

--S 13 of 30
print copyBSD()
--R
--R [dX,dt]
--R                                         Type: Void
--E 13

--S 14 of 30
print copyIto()
--R
--R table(t= dt,X= dX)
--R                                         Type: Void
--E 14

--S 15 of 30
introduce!(t,dt) -- fails (can't repeat an introduction!)
--R
--R
--R (15) "failed"
--R                                         Type: Union("failed",...)
--E 15

--S 16 of 30
print copyBSD()
--R
--R [dX,dt]
--R                                         Type: Void
--E 16

--S 17 of 30
print copyIto()
--R
--R table(t= dt,X= dX)
--R                                         Type: Void
--E 17

--S 18 of 30
'dZ::BSD -- can fail
--R
--R dZ is not a stochastic differential
--R
--RDaly Bug
--R >> Error detected within library code:
--R above causes failure in convert$BSD
--R

```

```
--R Continuing to read the file...
--R
--E 18

--S 19 of 30
dZ::BSD -- fails
--R
--R dZ is not a stochastic differential
--R
--RDaly Bug
--R >> Error detected within library code:
--R above causes failure in convert$BSD
--R
--R Continuing to read the file...
--R
--E 19

--S 20 of 30
string(dY)::BSD -- fails
--R
--R
--RDaly Bug
--R Cannot convert from type String to BasicStochasticDifferential for
--R value
--R "dY"
--R
--E 20

--S 21 of 30
print domainOf(dt)
--R
--R Variable(dt)
--R
--R Type: Void
--E 21

--S 22 of 30
dt::BSD -- succeeds!
--R
--R
--R (19) dt
--R
--R Type: BasicStochasticDifferential
--E 22

--S 23 of 30
getSmgl(dt::BSD)
--R
--R
--R (20) t
--R
--R Type: Union(Symbol,...)
--E 23
```

```

--S 24 of 30
dX::Symbol      -- succeeds
--R
--R
--R (21) dX
--R
--R                                          Type: Symbol
--E 24

--S 25 of 30
string(dX)      -- fails
--R
--R There are 3 exposed and 1 unexposed library operations named string
--R having 1 argument(s) but none was determined to be applicable.
--R Use HyperDoc Browse, or issue
--R                               )display op string
--R to learn more about the available operations. Perhaps
--R package-calling the operation or using coercions on the arguments
--R will allow you to apply the operation.
--R
--RDaly Bug
--R Cannot find a definition or applicable library operation named
--R string with argument type(s)
--R                               BasicStochasticDifferential
--R
--R Perhaps you should use "@" to indicate the required return type,
--R or "$" to specify which version of the function you need.
--E 25

--S 26 of 30
string(dX::Symbol) -- succeeds
--R
--R
--R (22) "dX"
--R
--R                                          Type: String
--E 26

--S 27 of 30
[introduce!(A[i],dA[i]) for i in 1..2]
--R
--R
--R (23) [dA ,dA ]
--R      1    2
--R
--R                                          Type: List(Union(BasicStochasticDifferential,"failed"))
--E 27

--S 28 of 30
print copyBSD()
--R
--R [dA ,dA ,dX,dt]

```

```

--R      1  2
--R
--R                                          Type: Void
--E 28

--S 29 of 30
print copyIto()
--R
--R      table(A = dA ,A = dA ,t= dt,X= dX)
--R      2      2  1      1
--R
--R                                          Type: Void
--E 29

--S 30 of 30
[d A[i] for i in 1..2]
--R
--R
--R      (26)  [dA ,dA ]
--R      1      2
--R
--R                                          Type: List(Union(BasicStochasticDifferential,Integer))
--E 30

)spool
)lisp (bye)

```

— BasicStochasticDifferential.help —

```

=====
BasicStochasticDifferential examples
=====

```

Based on Symbol: a domain of symbols representing basic stochastic differentials, used in StochasticDifferential(R) in the underlying sparse multivariate polynomial representation.

We create new BSD only by coercion from Symbol using a special function introduce! first of all to add to a private set SDset. We allow a separate function convertIfCan which will check whether the argument has previously been declared as a BSD.

The copyBSD() returns setBSD as a list of BSD.

Initially the BSD table of symbols is empty:

```

print copyBSD()
[]

```

as is the Ito table. The copyIto() returns the table relating semimartingales to basic stochastic differentials.

```
print copyIto()
      table()
```

We introduce a new stochastic differential:

```
dX:=introduce!(X,dX) -- dX is a new stochastic differential
      dX
```

and now it is in the BSD table of symbols:

```
print copyBSD()
      [dX]
```

and the Ito table:

```
print copyIto()
      table(X= dX)
```

The `getSmgl(bsd)` returns the semimartingale S related to the basic stochastic differential `bsd` by `introduce!`:

```
getSmgl(dX)
      X
```

Note that the `dX` symbol is of type `BasicStochasticDifferential`:

```
print domainOf(dX)
      BasicStochasticDifferential()
```

The `d` function, `d(X)` returns `dX` if `tableIto(X)=dX` otherwise 0:

```
print d X
      dX
```

and we can specify the domain:

```
print ((d$BSD) X)
      dX
```

which clearly is of domain `BasicStochasticDifferential`

```
print domainOf((d$BSD) X)
      BasicStochasticDifferential()
```

The `introduce!(X,dX)` returns `dX` as BSD if it isn't already in BSD

```
introduce!(t,dt) -- dt is a new stochastic differential
      dt
```

Now we can see that `dt` is in the BSD table:

```
print copyBSD()
      [dX,dt]
```

and the Ito table:

```
print copyIto()
      table(t= dt,X= dX)
```

You cannot repeat an `introduce!` for an existing entry:

```
introduce!(t,dt)
"failed"
```

Regular symbols can be added, especially with subscripts:

```
[introduce!(A[i],dA[i]) for i in 1..2]
      [dA1,dA2]
```

And we can see them in the BSD table:

```
print copyBSD()
      [dA1,dA2,dX,dt]
```

and the Ito table:

```
print copyIto()
      table(A = dA2, A = dA2, t= dt, X= dX1)
```

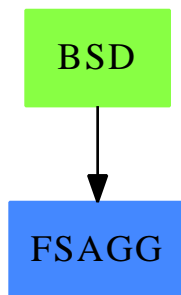
The d function can extract these symbols:

```
[d A[i] for i in 1..2]
      [dA1,dA2]
```

See Also:

```
o )show BasicStochasticDifferential
```

BasicStochasticDifferential (BSD)



See

⇒ “StochasticDifferential” (SD) [20](#) on page [2572](#)

Exports:

coerce	convert	convertIfCan	copyBSD
copyIto	d	getSmgl	hash
introduce!	latex	max	min
?~=?	?<?	?<=?	?=?
?>?	?>=?		

— domain BSD BasicStochasticDifferential —

```

)abbrev domain BSD BasicStochasticDifferential
++ Author: Wilfrid S. Kendall
++ Date Last Updated: July 26, 1999
++ Related Domains: StochasticDifferential(R)
++ References: Ito (1975), Kendall (1991a,b; 1993a,b; 1999a,b).
++ Description:
++ Based on Symbol: a domain of symbols representing basic stochastic
++ differentials, used in StochasticDifferential(R) in the underlying
++ sparse multivariate polynomial representation.
++
++ We create new BSD only by coercion from Symbol using a special
++ function introduce! first of all to add to a private set SDset.
++ We allow a separate function convertIfCan which will check whether the
++ argument has previously been declared as a BSD.

BasicStochasticDifferential(): category == implementation where
  INT ==> Integer
  OF ==> OutputForm
  category ==> OrderedSet with
    ConvertibleTo(Symbol)

  convertIfCan: Symbol -> Union(%, "failed")
    ++ convertIfCan(ds) transforms \axiom{dX} into a \axiom{BSD}
    ++ if possible (if \axiom{introduce(X,dX)} has
    ++ been invoked previously).

  convert: Symbol -> %
    ++ convert(dX) transforms \axiom{dX} into a \axiom{BSD}
    ++ if possible and otherwise produces an error.

  introduce!: (Symbol,Symbol) -> Union(%, "failed")
    ++ introduce!(X,dX) returns \axiom{dX} as \axiom{BSD} if it
    ++ isn't already in \axiom{BSD}
    ++
    ++X introduce!(t,dt) -- dt is a new stochastic differential
    ++X copyBSD()

  d: Symbol -> Union(%,INT)
    ++ d(X) returns \axiom{dX} if \axiom{tableIto(X)=dX}
    ++ and otherwise returns \axiom{0}

  copyBSD:() -> List %

```

```

++ copyBSD() returns \axiom{setBSD} as a list of \axiom{BSD}.
++
++X introduce!(t,dt) -- dt is a new stochastic differential
++X copyBSD()

copyIto:() -> Table(Symbol,%)
++ copyIto() returns the table relating semimartingales
++ to basic stochastic differentials.
++
++X introduce!(t,dt) -- dt is a new stochastic differential
++X copyIto()

getSmgl: % -> Union(Symbol,"failed")
++ getSmgl(bsd) returns the semimartingale \axiom{S} related
++ to the basic stochastic differential \axiom{bsd} by
++ \axiom{introduce!}
++
++X introduce!(t,dt) -- dt is a new stochastic differential
++X getSmgl(dt::BSD)

implementation ==> Symbol add

Rep := Symbol

setBSD := empty()$Set(Symbol)
tableIto:Table(Symbol,%) := table()
tableBSD:Table(%,Symbol) := table()

convertIfCan(ds:Symbol):Union(%, "failed") ==
not(member?(ds,setBSD)) => "failed"
ds::%

convert(ds:Symbol):% ==
(du:=convertIfCan(ds))
case "failed" =>
print(hconcat(ds::Symbol::OF,
message(" is not a stochastic differential")$OF))
error "above causes failure in convert$BSD"
du

introduce!(X,dX) ==
member?(dX,setBSD) => "failed"
insert!(dX,setBSD)
tableBSD(dX::%) := X
tableIto(X) := dX::%

d(X) ==
search(X,tableIto) case "failed" => 0::INT
tableIto(X)

```

```

copyBSD() == [ds::% for ds in members(setBSD)]
copyIto() == tableIto
getSmgl(ds:):Union(Symbol,"failed") == tableBSD(ds)

```

— BSD.dotabb —

```

"BSD" [color="#88FF44",href="bookvol10.3.pdf#nameddest=BSD"]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"BSD" -> "FSAGG"

```

domain BINARY BinaryExpansion

— BinaryExpansion.input —

```

)set break resume
)sys rm -f BinaryExpansion.output
)spool BinaryExpansion.output
)set message test on
)set message auto off
)clear all
--S 1 of 7
r := binary(22/7)
--R
--R
--R
--R      (1)  11.001
--R
--R                                          Type: BinaryExpansion
--E 1

--S 2 of 7
r + binary(6/7)
--R
--R
--R      (2)  100
--R
--R                                          Type: BinaryExpansion
--E 2

--S 3 of 7
[binary(1/i) for i in 102..106]
--R

```

```

--R
--R (3)
--R -----
--R [0.000000101, 0.00000010011111000100010110010111001110010010101001,
--R -----
--R 0.000000100111011, 0.000000100111,
--R -----
--R 0.00000010011010100100001110011111011001010110111100011]
--R
--R                                          Type: List(BinaryExpansion)
--E 3

--S 4 of 7
binary(1/1007)
--R
--R
--R (4)
--R 0.
--R OVERBAR
--R 0000000001000001000101001001011110000011111100001011111001011000111110
--R 1000100111001001100110001100100101011110110100110000000011000011001
--R 111011100011010001011110100100011110110000101011101110011101010111001
--R 100101001011100000001110001111001000000100100100110111001010100111010
--R 001101110110101110001001000001100101101100000010110010111110001010000
--R 01010101011010110000011011011101001010111111010111010100110010000101
--R 0011011000100110001000100001000011000111010011110001
--R
--R                                          Type: BinaryExpansion
--E 4

--S 5 of 7
p := binary(1/4)*x**2 + binary(2/3)*x + binary(4/9)
--R
--R
--R      2      --      -----
--R (5)  0.01x  + 0.10x + 0.011100
--R
--R                                          Type: Polynomial(BinaryExpansion)
--E 5

--S 6 of 7
q := D(p, x)
--R
--R
--R      --
--R (6)  0.1x + 0.10
--R
--R                                          Type: Polynomial(BinaryExpansion)
--E 6

--S 7 of 7
g := gcd(p, q)
--R
--R

```

```

--R
--R      (7)  x + 1.01
--R
--R                                          Type: Polynomial(BinaryExpansion)
--E 7
)spool
)lisp (bye)

```

— BinaryExpansion.help —

=====

BinaryExpansion examples

=====

All rational numbers have repeating binary expansions. Operations to access the individual bits of a binary expansion can be obtained by converting the value to RadixExpansion(2). More examples of expansions are available with

The expansion (of type BinaryExpansion) of a rational number is returned by the binary operation.

```

r := binary(22/7)
--
11.001
Type: BinaryExpansion

```

Arithmetic is exact.

```

r + binary(6/7)
100
Type: BinaryExpansion

```

The period of the expansion can be short or long.

```

[binary(1/i) for i in 102..106]
-----
[0.00000101,
-----
0.000000100111110001000101100101111001110010010101001,
-----
0.000000100111011, 0.000000100111,
-----
0.00000010011010100100001110011111011001010110111100011]
Type: List BinaryExpansion

```

or very long.

```

binary(1/1007)
-----
0.000000000100000100010100100101111000001111110000101111110010110001111101
-----
000100111001001100110001100100101010111101101001100000000110000110011110
-----
111000110100010111101001000111101100001010111011100111010101110011001010
-----
010111000000011100011110010000001001001001101110010101001110100011011101
-----
101011100010010000011001011011000000101100101111100010100000101010101101
-----
01100000110110111010010101111110101110101001100100001010011011000100110
-----
001000100001000011000111010011110001
Type: BinaryExpansion

```

These numbers are bona fide algebraic objects.

```

p := binary(1/4)*x**2 + binary(2/3)*x + binary(4/9)
-----
0.01 x^2 + 0.10 x + 0.011100
Type: Polynomial BinaryExpansion

q := D(p, x)
-----
0.1 x + 0.10
Type: Polynomial BinaryExpansion

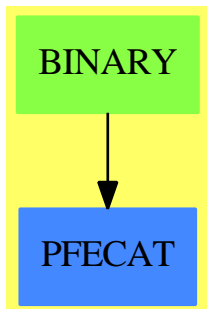
g := gcd(p, q)
-----
x+1.01
Type: Polynomial BinaryExpansion

```

See Also:

- o)help DecimalExpansion
- o)help HexadecimalExpansion
- o)show BinaryExpansion

BinaryExpansion (BINARY)



See

⇒ “RadixExpansion” (RADIX) [19](#) on page [2205](#)

⇒ “DecimalExpansion” (DECIMAL) [5](#) on page [465](#)

⇒ “HexadecimalExpansion” (HEXADEC) [9](#) on page [1142](#)

Exports:

0	1	abs
associates?	binary	ceiling
characteristic	charthRoot	coerce
conditionP	convert	D
denom	denominator	differentiate
divide	euclideanSize	eval
expressIdealMember	exquo	extendedEuclidean
factor	factorPolynomial	factorSquareFreePolynomial
floor	fractionPart	gcd
gcdPolynomial	hash	init
inv	latex	lcm
map	max	min
multiEuclidean	negative?	nextItem
numer	numerator	one?
patternMatch	positive?	prime?
principalIdeal	random	recip
reducedSystem	retract	retractIfCan
sample	sign	sizeLess?
solveLinearPolynomialEquation	squareFree	squareFreePart
squareFreePolynomial	subtractIfCan	unit?
unitCanonical	unitNormal	wholePart
zero?	?*?	?**?
?+?	?-?	-?
?/?	?=?	?^?
?~=?	?<?	?<=?
?>?	?>=?	?..?
?quo?	?rem?	

— domain BINARY BinaryExpansion —

```

)abbrev domain BINARY BinaryExpansion
++ Author: Clifton J. Williamson
++ Date Created: April 26, 1990
++ Date Last Updated: May 15, 1991
++ Description:
++ This domain allows rational numbers to be presented as repeating
++ binary expansions.

BinaryExpansion(): Exports == Implementation where
  Exports ==> QuotientFieldCategory(Integer) with
    coerce: % -> Fraction Integer
      ++ coerce(b) converts a binary expansion to a rational number.
    coerce: % -> RadixExpansion(2)
      ++ coerce(b) converts a binary expansion to a radix expansion with base 2.
    fractionPart: % -> Fraction Integer
      ++ fractionPart(b) returns the fractional part of a binary expansion.
    binary: Fraction Integer -> %
      ++ binary(r) converts a rational number to a binary expansion.
      ++
      ++X binary(22/7)

Implementation ==> RadixExpansion(2) add
  binary r == r :: %
  coerce(x:%): RadixExpansion(2) == x pretend RadixExpansion(2)

```

— BINARY.dotabb —

```

"BINARY" [color="#88FF44",href="bookvol10.3.pdf#nameddest=BINARY"]
"PFECA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECA"]
"BINARY" -> "PFECA"

```

domain BINFILE BinaryFile

— BinaryFile.input —

```

)set break resume
)sys rm -f BinaryFile.output
)spool BinaryFile.output

```



```

)set message test on
)set message auto off
)clear all

--S 1 of 1
)show BinaryFile
--R BinaryFile is a domain constructor
--R Abbreviation for BinaryFile is BINFILE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for BINFILE
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          close! : % -> %
--R coerce : % -> OutputForm        flush : % -> Void
--R hash : % -> SingleInteger        iomode : % -> String
--R latex : % -> String              name : % -> FileName
--R open : (FileName,String) -> %    open : FileName -> %
--R position : % -> SingleInteger    read! : % -> SingleInteger
--R reopen! : (% ,String) -> %      ?~=? : (%,% ) -> Boolean
--R position! : (% ,SingleInteger) -> SingleInteger
--R readIfCan! : % -> Union(SingleInteger,"failed")
--R write! : (% ,SingleInteger) -> SingleInteger
--R
--E 1

)spool
)lisp (bye)

```

— BinaryFile.help —

=====

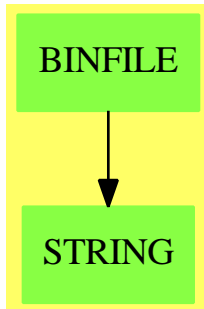
BinaryFile examples

=====

This domain provides an implementation of binary files. Data is accessed one byte at a time as a small integer.

See Also:

o)show BinaryFile

BinaryFile (BINFILE)**See**

- ⇒ “File” (FILE) [7](#) on page [802](#)
- ⇒ “TextFile” (TEXTFILE) [21](#) on page [2692](#)
- ⇒ “KeyedAccessFile” (KAFfile) [12](#) on page [1413](#)
- ⇒ “Library” (LIB) [13](#) on page [1428](#)

Exports:

close!	coerce	hash	iomode	latex
name	open	position	position!	read!
readIfCan!	reopen!	write!	?=?	?~=?

— domain **BINFILE** BinaryFile —

```

)abbrev domain BINFILE BinaryFile
++ Author: Barry M. Trager
++ Date Created: 1993
++ Description:
++ This domain provides an implementation of binary files. Data is
++ accessed one byte at a time as a small integer.

BinaryFile: Cat == Def where

  Cat == FileCategory(FileName, SingleInteger) with
    readIfCan_!: % -> Union(SingleInteger, "failed")
      ++ readIfCan!(f) returns a value from the file f, if possible.
      ++ If f is not open for reading, or if f is at the end of file
      ++ then \spad{"failed"} is the result.

  -- "#": % -> SingleInteger
  -- ++ #(f) returns the length of the file f in bytes.

  position: % -> SingleInteger
    ++ position(f) returns the current byte-position in the file f.

  position_!: (% , SingleInteger) -> SingleInteger
    ++ position!(f, i) sets the current byte-position to i.

```

```

Def == File(SingleInteger) add
    FileState ==> SExpression

    Rep := Record(fileName:  FileName,  _
                  fileState: FileState,  _
                  fileIOmode: String)

--    direc : Symbol := INTERN("DIRECTION","KEYWORD")$Lisp
--    input : Symbol := INTERN("INPUT","KEYWORD")$Lisp
--    output : Symbol := INTERN("OUTPUT","KEYWORD")$Lisp
--    eltype : Symbol := INTERN("ELEMENT-TYPE","KEYWORD")$Lisp
--    bytesize : SExpression := LIST(QUOTE(UNSIGNED$Lisp)$Lisp,8)$Lisp

defstream(fn: FileName, mode: String): FileState ==
    mode = "input" =>
        not readable? fn => error ["File is not readable", fn]
        BINARY__OPEN__INPUT(fn::String)$Lisp
--    OPEN(fn::String, direc, input, eltype, bytesize)$Lisp
    mode = "output" =>
        not writable? fn => error ["File is not writable", fn]
        BINARY__OPEN__OUTPUT(fn::String)$Lisp
--    OPEN(fn::String, direc, output, eltype, bytesize)$Lisp
        error ["IO mode must be input or output", mode]

open(fname, mode) ==
    fstream := defstream(fname, mode)
    [fname, fstream, mode]

reopen_!(f, mode) ==
    fname := f.fileName
    f.fileState := defstream(fname, mode)
    f.fileIOmode := mode
    f

close_! f ==
    f.fileIOmode = "output" =>
        BINARY__CLOSE__OUTPUT()$Lisp
        f
    f.fileIOmode = "input" =>
        BINARY__CLOSE__INPUT()$Lisp
        f
    error "file must be in read or write state"

read! f ==
    f.fileIOmode ^= "input" => error "File not in read state"
    BINARY__SELECT__INPUT(f.fileState)$Lisp
    BINARY__READBYTE()$Lisp
--    READ_-BYTE(f.fileState)$Lisp

```

```

readIfCan_! f ==
  f.fileIOmode ^= "input" => error "File not in read state"
  BINARY__SELECT__INPUT(f.fileState)$Lisp
  n:SingleInteger:=BINARY__READBYTE()$Lisp
  n = -1 => "failed"
  n::Union(SingleInteger,"failed")
--  READ_-BYTE(f.fileState,NIL$Lisp,
--    "failed"::Union(SingleInteger,"failed"))$Lisp
write_!(f, x) ==
  f.fileIOmode ^= "output" => error "File not in write state"
  x < 0 or x>255 => error "integer cannot be represented as a byte"
  BINARY__PRINBYTE(x)$Lisp
--  WRITE_-BYTE(x, f.fileState)$Lisp
  x

--  # f == FILE_-LENGTH(f.fileState)$Lisp
  position f ==
    f.fileIOmode ^= "input" => error "file must be in read state"
    FILE_-POSITION(f.fileState)$Lisp
  position_!(f,i) ==
    f.fileIOmode ^= "input" => error "file must be in read state"
    (FILE_-POSITION(f.fileState,i)$Lisp ; i)

```

— BINFILE.dotabb —

```

"BINFILE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=BINFILE"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"BINFILE" -> "STRING"

```

domain BSTREE BinarySearchTree

— BinarySearchTree.input —

```

)set break resume
)sys rm -f BinarySearchTree.output
)spool BinarySearchTree.output
)set message test on
)set message auto off
)clear all
--S 1 of 12

```

```

lv := [8,3,5,4,6,2,1,5,7]
--R
--R
--R (1) [8,3,5,4,6,2,1,5,7]
--R
--R                                          Type: List(PositiveInteger)
--E 1

--S 2 of 12
t := binarySearchTree lv
--R
--R
--R (2) [[1,2,.],3,[4,5,[5,6,7]]],8,.]
--R
--R                                          Type: BinarySearchTree(PositiveInteger)
--E 2

--S 3 of 12
emptybst := empty()$BSTREE(INT)
--R
--R
--R (3) []
--R
--R                                          Type: BinarySearchTree(Integer)
--E 3

--S 4 of 12
t1 := insert!(8,emptybst)
--R
--R
--R (4) 8
--R
--R                                          Type: BinarySearchTree(Integer)
--E 4

--S 5 of 12
insert!(3,t1)
--R
--R
--R (5) [3,8,.]
--R
--R                                          Type: BinarySearchTree(Integer)
--E 5

--S 6 of 12
leaves t
--R
--R
--R (6) [1,4,5,7]
--R
--R                                          Type: List(PositiveInteger)
--E 6

--S 7 of 12
split(3,t)
--R

```

```

--R
--R (7) [less= [1,2,.],greater= [[.],3,[4,5,[5,6,7]]],8,.]
--IType: Record(less: BinarySearchTree(PositiveInteger),greater: ...
--E 7

--S 8 of 12
insertRoot: (INT,BSTREE INT) -> BSTREE INT
--R
--R
--R                                          Type: Void
--E 8

--S 9 of 12
insertRoot(x, t) ==
  a := split(x, t)
  node(a.less, x, a.greater)
--R
--R
--R                                          Type: Void
--E 9

--S 10 of 12
buildFromRoot ls == reduce(insertRoot,ls,emptybst)
--R
--R
--R                                          Type: Void
--E 10

--S 11 of 12
rt := buildFromRoot reverse lv
--R
--R   Compiling function buildFromRoot with type List(PositiveInteger) ->
--R     BinarySearchTree(Integer)
--R   Compiling function insertRoot with type (Integer,BinarySearchTree(
--R     Integer)) -> BinarySearchTree(Integer)
--R
--R (11) [[[1,2,.],3,[4,5,[5,6,7]]],8,.]
--R                                          Type: BinarySearchTree(Integer)
--E 11

--S 12 of 12
(t = rt)@Boolean
--R
--R
--R (12) true
--R
--R                                          Type: Boolean
--E 12
)spool
)lisp (bye)

```

— BinarySearchTree.help —

```
=====
BinarySearchTree examples
=====
```

BinarySearchTree(R) is the domain of binary trees with elements of type R, ordered across the nodes of the tree. A non-empty binary search tree has a value of type R, and right and left binary search subtrees. If a subtree is empty, it is displayed as a period (".").

Define a list of values to be placed across the tree. The resulting tree has 8 at the root; all other elements are in the left subtree.

```
lv := [8,3,5,4,6,2,1,5,7]
      [8, 3, 5, 4, 6, 2, 1, 5, 7]
                                     Type: List PositiveInteger
```

A convenient way to create a binary search tree is to apply the operation binarySearchTree to a list of elements.

```
t := binarySearchTree lv
    [[[1, 2, .], 3, [4, 5, [5, 6, 7]]], 8, .]
                                     Type: BinarySearchTree PositiveInteger
```

Another approach is to first create an empty binary search tree of integers.

```
emptybst := empty()$BSTREE(INT)
[]
                                     Type: BinarySearchTree Integer
```

Insert the value 8. This establishes 8 as the root of the binary search tree. Values inserted later that are less than 8 get stored in the left subtree, others in the right subtree.

```
t1 := insert!(8,emptybst)
      8
                                     Type: BinarySearchTree Integer
```

Insert the value 3. This number becomes the root of the left subtree of t1. For optimal retrieval, it is thus important to insert the middle elements first.

```
insert!(3,t1)
      [3, 8, .]
                                     Type: BinarySearchTree Integer
```

We go back to the original tree t. The leaves of the binary search tree are those which have empty left and right subtrees.

```

leaves t
[1, 4, 5, 7]
Type: List PositiveInteger

```

The operation `split(k,t)` returns a record containing the two subtrees: one with all elements "less" than `k`, another with elements "greater" than `k`.

```

split(3,t)
[less=[1, 2, .], greater=[[., 3, [4, 5, [5, 6, 7]]], 8, .]]
Type: Record(less: BinarySearchTree PositiveInteger,greater:
BinarySearchTree PositiveInteger)

```

Define `insertRoot` to insert new elements by creating a new node.

```

insertRoot: (INT,BSTREE INT) -> BSTREE INT
Type: Void

```

The new node puts the inserted value between its "less" tree and "greater" tree.

```

insertRoot(x, t) ==
a := split(x, t)
node(a.less, x, a.greater)
Type: Void

```

Function `buildFromRoot` builds a binary search tree from a list of elements `ls` and the empty tree `emptybst`.

```

buildFromRoot ls == reduce(insertRoot,ls,emptybst)
Type: Void

```

Apply this to the reverse of the list `lv`.

```

rt := buildFromRoot reverse lv
[[[1, 2, . ], 3, [4, 5, [5, 6, 7]]], 8, .]
Type: BinarySearchTree Integer

```

Have Axiom check that these are equal.

```

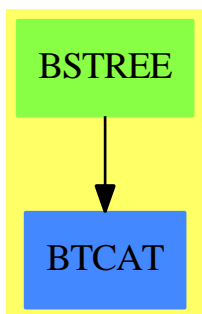
(t = rt)@Boolean
true
Type: Boolean

```

See Also:

```
o )show BinarySearchTree
```


BinarySearchTree (BSTREE)



See

- ⇒ “Tree” (TREE) [21](#) on page [2740](#)
- ⇒ “BinaryTree” (BTREE) [3](#) on page [310](#)
- ⇒ “BinaryTournament” (BTourn) [3](#) on page [306](#)
- ⇒ “BalancedBinaryTree” (BBTREE) [3](#) on page [252](#)
- ⇒ “PendantTree” (PENDTREE) [17](#) on page [1944](#)

Exports:

any?	binarySearchTree	child?	children	coerce
copy	count	count	cyclic?	distance
empty	empty?	eq?	eval	eval
eval	eval	every?	hash	insert!
insertRoot!	latex	leaf?	leaves	left
less?	map	map!	member?	members
more?	node	node?	nodes	parts
right	sample	setchildren!	setelt	setelt
setelt	setleft!	setright!	setvalue!	size?
split	value	#?	?=?	?~=?
?.right	?.left	?.value		

— domain BSTREE BinarySearchTree —

```

)abbrev domain BSTREE BinarySearchTree
++ Author: Mark Botch
++ Description:
++ BinarySearchTree(S) is the domain of
++ a binary trees where elements are ordered across the tree.
++ A binary search tree is either empty or has
++ a value which is an S, and a
++ right and left which are both BinaryTree(S)
++ Elements are ordered across the tree.

```

```

BinarySearchTree(S: OrderedSet): Exports == Implementation where
  Exports == BinaryTreeCategory(S) with
    shallowlyMutable
    finiteAggregate
    binarySearchTree: List S -> %
    ++ binarySearchTree(1) is not documented
    ++
    ++X binarySearchTree [1,2,3,4]

    insert_!: (S,%) -> %
    ++ insert!(x,b) inserts element x as leaves into binary search tree b.
    ++
    ++X t1:=binarySearchTree [1,2,3,4]
    ++X insert!(5,t1)

    insertRoot_!: (S,%) -> %
    ++ insertRoot!(x,b) inserts element x as a root of binary search tree b.
    ++
    ++X t1:=binarySearchTree [1,2,3,4]
    ++X insertRoot!(5,t1)

    split:      (S,%) -> Record(less: %, greater: %)
    ++ split(x,b) splits binary tree b into two trees, one with elements
    ++ greater than x, the other with elements less than x.
    ++
    ++X t1:=binarySearchTree [1,2,3,4]
    ++X split(3,t1)

Implementation == BinaryTree(S) add
  Rep := BinaryTree(S)
  binarySearchTree(u:List S) ==
    null u => empty()
    tree := binaryTree(first u)
    for x in rest u repeat insert_!(x,tree)
    tree
  insert_!(x,t) ==
    empty? t => binaryTree(x)
    x >= value t =>
      setright_!(t,insert_!(x,right t))
      t
    setleft_!(t,insert_!(x,left t))
    t
  split(x,t) ==
    empty? t => [empty(),empty()]
    x > value t =>
      a := split(x,right t)
      [node(left t, value t, a.less), a.greater]
    a := split(x,left t)
    [a.less, node(a.greater, value t, right t)]

```

```

insertRoot_!(x,t) ==
  a := split(x,t)
  node(a.less, x, a.greater)

```

— BSTREE.dotabb —

```

"BSTREE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=BSTREE"]
"BTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BTCAT"]
"BSTREE" -> "BTCAT"

```

domain BTOURN BinaryTournament

A BinaryTournament(S) is the domain of binary trees where elements are ordered down the tree. A binary search tree is either empty or is a node containing a value of type S, and a right and a left which are both BinaryTree(S)

— BinaryTournament.input —

```

)set break resume
)sys rm -f BinaryTournament.output
)spool BinaryTournament.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show BinaryTournament
--R
--R BinaryTournament(S: OrderedSet) is a domain constructor
--R Abbreviation for BinaryTournament is BTOURN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for BTOURN
--R
--R----- Operations -----
--R binaryTournament : List(S) -> %          children : % -> List(%)
--R copy : % -> %                             cyclic? : % -> Boolean
--R distance : (%,%) -> Integer                ?.right : (%,right) -> %
--R ?.left : (%,left) -> %                     ?.value : (%,value) -> S
--R empty : () -> %                           empty? : % -> Boolean
--R eq? : (%,%) -> Boolean                     insert! : (S,%) -> %

```

```

--R leaf? : % -> Boolean                leaves : % -> List(S)
--R left : % -> %                        map : ((S -> S),%) -> %
--R node : (% , S , %) -> %              nodes : % -> List(%)
--R right : % -> %                       sample : () -> %
--R value : % -> S
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ==? : (% , %) -> Boolean if S has SETCAT
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R child? : (% , %) -> Boolean if S has SETCAT
--R coerce : % -> OutputForm if S has SETCAT
--R count : (S , %) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R eval : (% , List(S) , List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (% , S , S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (% , Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (% , List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R hash : % -> SingleInteger if S has SETCAT
--R latex : % -> String if S has SETCAT
--R less? : (% , NonNegativeInteger) -> Boolean
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R member? : (S , %) -> Boolean if $ has finiteAggregate and S has SETCAT
--R members : % -> List(S) if $ has finiteAggregate
--R more? : (% , NonNegativeInteger) -> Boolean
--R node? : (% , %) -> Boolean if S has SETCAT
--R parts : % -> List(S) if $ has finiteAggregate
--R setchildren! : (% , List(%)) -> % if $ has shallowlyMutable
--R setelt : (% , right , %) -> % if $ has shallowlyMutable
--R setelt : (% , left , %) -> % if $ has shallowlyMutable
--R setelt : (% , value , S) -> S if $ has shallowlyMutable
--R setleft! : (% , %) -> % if $ has shallowlyMutable
--R setright! : (% , %) -> % if $ has shallowlyMutable
--R setvalue! : (% , S) -> S if $ has shallowlyMutable
--R size? : (% , NonNegativeInteger) -> Boolean
--R ~=? : (% , %) -> Boolean if S has SETCAT
--R
--E 1

)spool
)lisp (bye)

```

— BinaryTournament.help —

```

=====
BinaryTournament examples
=====

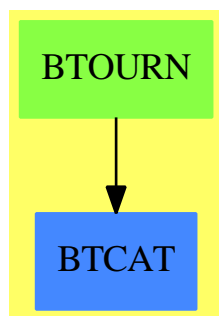
```

`BinaryTournament` creates a binary tournament with the elements of `ls` as values at the nodes.

See Also:

o `)show BinaryTournament`

BinaryTournament (BTOURN)



See

- ⇒ “Tree” (TREE) [21](#) on page [2740](#)
- ⇒ “BinaryTree” (BTREE) [3](#) on page [310](#)
- ⇒ “BinarySearchTree” (BSTREE) [3](#) on page [302](#)
- ⇒ “BalancedBinaryTree” (BBTREE) [3](#) on page [252](#)
- ⇒ “PendantTree” (PENDTREE) [17](#) on page [1944](#)

Exports:

<code>any?</code>	<code>binaryTournament</code>	<code>child?</code>	<code>children</code>	<code>coerce</code>
<code>copy</code>	<code>count</code>	<code>cyclic?</code>	<code>distance</code>	<code>empty</code>
<code>empty?</code>	<code>eq?</code>	<code>eval</code>	<code>every?</code>	<code>hash</code>
<code>insert!</code>	<code>latex</code>	<code>leaf?</code>	<code>leaves</code>	<code>left</code>
<code>less?</code>	<code>map</code>	<code>map!</code>	<code>member?</code>	<code>members</code>
<code>more?</code>	<code>node</code>	<code>node?</code>	<code>nodes</code>	<code>parts</code>
<code>right</code>	<code>sample</code>	<code>setchildren!</code>	<code>setelt</code>	<code>setleft!</code>
<code>setright!</code>	<code>setvalue!</code>	<code>size?</code>	<code>value</code>	<code>#?</code>
<code>?=?</code>	<code>?~=?</code>	<code>?..right</code>	<code>?..left</code>	<code>?..value</code>

— domain BTOURN BinaryTournament —

```

)abbrev domain BTOURN BinaryTournament
++ Author: Mark Botch
++ Description:
++ BinaryTournament creates a binary tournament with the
++ elements of ls as values at the nodes.
  
```

```

BinaryTournament(S: OrderedSet): Exports == Implementation where
  Exports == BinaryTreeCategory(S) with
    shallowlyMutable
  binaryTournament: List S -> %
    ++ binaryTournament(ls) creates a binary tournament with the
    ++ elements of ls as values at the nodes.
    ++
    ++X binaryTournament [1,2,3,4]

  insert_!: (S,%) -> %
    ++ insert!(x,b) inserts element x as leaves into binary tournament b.
    ++
    ++X t1:=binaryTournament [1,2,3,4]
    ++X insert!(5,t1)
    ++X t1

Implementation == BinaryTree(S) add
  Rep := BinaryTree(S)
  binaryTournament(u:List S) ==
    null u => empty()
    tree := binaryTree(first u)
    for x in rest u repeat insert_!(x,tree)
    tree
  insert_!(x,t) ==
    empty? t => binaryTree(x)
    x > value t =>
      setleft_!(t,copy t)
      setvalue_!(t,x)
      setright_!(t,empty())
    setright_!(t,insert_!(x,right t))
  t

```

— BTOURN.dotabb —

```

"BTourn" [color="#88FF44",href="bookvol10.3.pdf#nameddest=BTourn"]
"BTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BTCAT"]
"BTourn" -> "BTCAT"

```

domain BTREE BinaryTree

— BinaryTree.input —

```
)set break resume
)sys rm -f BinaryTree.output
)spool BinaryTree.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show BinaryTree
--R
--R BinaryTree(S: SetCategory) is a domain constructor
--R Abbreviation for BinaryTree is BTREE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for BTREE
--R
--R----- Operations -----
--R binaryTree : (%,S,%) -> %          binaryTree : S -> %
--R children : % -> List(%)          copy : % -> %
--R cyclic? : % -> Boolean          distance : (%,%) -> Integer
--R ?.right : (%,right) -> %        ?.left : (%,left) -> %
--R ?.value : (%,value) -> S        empty : () -> %
--R empty? : % -> Boolean          eq? : (%,%) -> Boolean
--R leaf? : % -> Boolean          leaves : % -> List(S)
--R left : % -> %                  map : ((S -> S),%) -> %
--R node : (%,S,%) -> %          nodes : % -> List(%)
--R right : % -> %                sample : () -> %
--R value : % -> S
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?=? : (%,%) -> Boolean if S has SETCAT
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R child? : (%,%) -> Boolean if S has SETCAT
--R coerce : % -> OutputForm if S has SETCAT
--R count : (S,%) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R eval : (%,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R hash : % -> SingleInteger if S has SETCAT
--R latex : % -> String if S has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R member? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
```

```

--R members : % -> List(S) if $ has finiteAggregate
--R more? : (% , NonNegativeInteger) -> Boolean
--R node? : (% , %) -> Boolean if S has SETCAT
--R parts : % -> List(S) if $ has finiteAggregate
--R setchildren! : (% , List(%)) -> % if $ has shallowlyMutable
--R setelt : (% , right, %) -> % if $ has shallowlyMutable
--R setelt : (% , left, %) -> % if $ has shallowlyMutable
--R setelt : (% , value, S) -> S if $ has shallowlyMutable
--R setleft! : (% , %) -> % if $ has shallowlyMutable
--R setright! : (% , %) -> % if $ has shallowlyMutable
--R setvalue! : (% , S) -> S if $ has shallowlyMutable
--R size? : (% , NonNegativeInteger) -> Boolean
--R ~=? : (% , %) -> Boolean if S has SETCAT
--R
--E 1

```

```

)spool
)lisp (bye)

```

— BinaryTree.help —

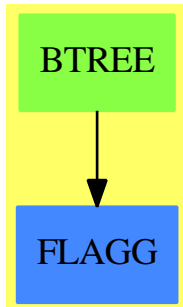
=====
 BinaryTree examples
 =====

BinaryTree(S) is the domain of all binary trees. A binary tree over S is either empty or has a value which is an S and a right and left which are both binary trees.

See Also:

o)show BinaryTree

BinaryTree (BTREE)



See

- ⇒ “Tree” (TREE) [21](#) on page [2740](#)
- ⇒ “BinarySearchTree” (BSTREE) [3](#) on page [302](#)
- ⇒ “BinaryTournament” (BTourn) [3](#) on page [306](#)
- ⇒ “BalancedBinaryTree” (BBTREE) [3](#) on page [252](#)
- ⇒ “PendantTree” (PENDTREE) [17](#) on page [1944](#)

Exports:

any?	binaryTree	child?	children	coerce
copy	count	cyclic?	distance	empty
empty?	eq?	eval	every?	hash
latex	leaf?	leaves	left	less?
map	map!	member?	members	more?
node	node?	nodes	parts	right
sample	setchildren!	setelt	setleft!	setright!
setvalue!	size?	value	#?	?=?
?~=?	?..right	?..left	?..value	

— domain BTREE BinaryTree —

```

)abbrev domain BTREE BinaryTree
++ Author: Mark Botch
++ Description:
++ \spadtype{BinaryTree(S)} is the domain of all
++ binary trees. A binary tree over \spad{S} is either empty or has
++ a \spadfun{value} which is an S and a \spadfun{right}
++ and \spadfun{left} which are both binary trees.

```

```

BinaryTree(S: SetCategory): Exports == Implementation where
  Exports == BinaryTreeCategory(S) with
    binaryTree: S -> %
    ++ binaryTree(v) is a non-empty binary tree
    ++ with value v, and left and right empty.
    ++
    ++X t1:=binaryTree([1,2,3])

```

```

binaryTree: (%S,%) -> %
++ binaryTree(l,v,r) creates a binary tree with
++ value v with left subtree l and right subtree r.
++
++X t1:=binaryTree([1,2,3])
++X t2:=binaryTree([4,5,6])
++X binaryTree(t1,[7,8,9],t2)

Implementation == add
Rep := List Tree S
t1 = t2 == (t1::Rep) =$Rep (t2::Rep)
empty() == [] pretend %
empty() == [] pretend %
node(l,v,r) == cons(tree(v,l:Rep),r:Rep)
binaryTree(l,v,r) == node(l,v,r)
binaryTree(v:S) == node(empty(),v,empty())
empty? t == empty?(t)$Rep
leaf? t == empty? t or empty? left t and empty? right t
right t ==
  empty? t => error "binaryTree:no right"
  rest t
left t ==
  empty? t => error "binaryTree:no left"
  children first t
value t ==
  empty? t => error "binaryTree:no value"
  value first t
setvalue_! (t,nd) ==
  empty? t => error "binaryTree:no value to set"
  setvalue_!(first(t:Rep),nd)
  nd
setleft_!(t1,t2) ==
  empty? t1 => error "binaryTree:no left to set"
  setchildren_!(first(t1:Rep),t2:Rep)
  t1
setright_!(t1,t2) ==
  empty? t1 => error "binaryTree:no right to set"
  setrest_!(t1:List Tree S,t2)

```

— BTREE.dotabb —

```

"BTREE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=BTREE"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"BTREE" -> "FLAGG"

```

domain BITS Bits

— Bits.input —

```

)set break resume
)sys rm -f Bits.output
)spool Bits.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Bits
--R
--R Bits is a domain constructor
--R Abbreviation for Bits is BITS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for BITS
--R
--R----- Operations -----
--R ?/\? : (%,%) -> %
--R ?<=? : (%,%) -> Boolean
--R ?>=? : (%,%) -> Boolean
--R ?\/? : (%,%) -> %
--R ?and? : (%,%) -> %
--R concat : (%,Boolean) -> %
--R concat : (%,%) -> %
--R construct : List(Boolean) -> %
--R delete : (%,Integer) -> %
--R empty : () -> %
--R entries : % -> List(Boolean)
--R hash : % -> SingleInteger
--R indices : % -> List(Integer)
--R latex : % -> String
--R min : (%,%) -> %
--R nor : (%,%) -> %
--R ?or? : (%,%) -> %
--R reverse : % -> %
--R xor : (%,%) -> %
--R ?~=? : (%,%) -> Boolean
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R any? : ((Boolean -> Boolean),%) -> Boolean if $ has finiteAggregate
--R bits : (NonNegativeInteger,Boolean) -> %
--R convert : % -> InputForm if Boolean has KONVERT(INFORM)
--R copyInto! : (%,%,Integer) -> % if $ has shallowlyMutable

--R ?<? : (%,%) -> Boolean
--R ?=? : (%,%) -> Boolean
--R ?>=? : (%,%) -> Boolean
--R ^? : % -> %
--R coerce : % -> OutputForm
--R concat : (Boolean,%) -> %
--R concat : List(%) -> %
--R copy : % -> %
--R ?.? : (%,Integer) -> Boolean
--R empty? : % -> Boolean
--R eq? : (%,%) -> Boolean
--R index? : (Integer,%) -> Boolean
--R insert : (%,%,Integer) -> %
--R max : (%,%) -> %
--R nand : (%,%) -> %
--R not? : % -> %
--R qelt : (%,Integer) -> Boolean
--R sample : () -> %
--R ~? : % -> %

```

```

--R count : ((Boolean -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R count : (Boolean,%) -> NonNegativeInteger if $ has finiteAggregate and Boolean has SETCAT
--R delete : (%,UniversalSegment(Integer)) -> %
--R elt : (%,Integer,Boolean) -> Boolean
--R ?.? : (%,UniversalSegment(Integer)) -> %
--R entry? : (Boolean,%) -> Boolean if $ has finiteAggregate and Boolean has SETCAT
--R eval : (%,List(Equation(Boolean))) -> % if Boolean has EVALAB(BOOLEAN) and Boolean has SETCAT
--R eval : (%,Equation(Boolean)) -> % if Boolean has EVALAB(BOOLEAN) and Boolean has SETCAT
--R eval : (%,Boolean,Boolean) -> % if Boolean has EVALAB(BOOLEAN) and Boolean has SETCAT
--R eval : (%,List(Boolean),List(Boolean)) -> % if Boolean has EVALAB(BOOLEAN) and Boolean has SETCAT
--R every? : ((Boolean -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (%,Boolean) -> % if $ has shallowlyMutable
--R find : ((Boolean -> Boolean),%) -> Union(Boolean,"failed")
--R first : % -> Boolean if Integer has ORDSET
--R insert : (Boolean,%,Integer) -> %
--R less? : (%,NonNegativeInteger) -> Boolean
--R map : ((Boolean -> Boolean),%) -> %
--R map : (((Boolean,Boolean) -> Boolean),%,%) -> %
--R map! : ((Boolean -> Boolean),%) -> % if $ has shallowlyMutable
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (Boolean,%) -> Boolean if $ has finiteAggregate and Boolean has SETCAT
--R members : % -> List(Boolean) if $ has finiteAggregate
--R merge : (((Boolean,Boolean) -> Boolean),%,%) -> %
--R merge : (%,%) -> % if Boolean has ORDSET
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (%,NonNegativeInteger) -> Boolean
--R new : (NonNegativeInteger,Boolean) -> %
--R parts : % -> List(Boolean) if $ has finiteAggregate
--R position : ((Boolean -> Boolean),%) -> Integer
--R position : (Boolean,%) -> Integer if Boolean has SETCAT
--R position : (Boolean,%,Integer) -> Integer if Boolean has SETCAT
--R qsetelt! : (%,Integer,Boolean) -> Boolean if $ has shallowlyMutable
--R reduce : (((Boolean,Boolean) -> Boolean),%,Boolean,Boolean) -> Boolean if $ has finiteAggregate and Boolean has SETCAT
--R reduce : (((Boolean,Boolean) -> Boolean),%,Boolean) -> Boolean if $ has finiteAggregate
--R reduce : (((Boolean,Boolean) -> Boolean),%) -> Boolean if $ has finiteAggregate
--R remove : (Boolean,%) -> % if $ has finiteAggregate and Boolean has SETCAT
--R remove : ((Boolean -> Boolean),%) -> % if $ has finiteAggregate
--R removeDuplicates : % -> % if $ has finiteAggregate and Boolean has SETCAT
--R reverse! : % -> % if $ has shallowlyMutable
--R select : ((Boolean -> Boolean),%) -> % if $ has finiteAggregate
--R setelt : (%,Integer,Boolean) -> Boolean if $ has shallowlyMutable
--R setelt : (%,UniversalSegment(Integer),Boolean) -> Boolean if $ has shallowlyMutable
--R size? : (%,NonNegativeInteger) -> Boolean
--R sort : (((Boolean,Boolean) -> Boolean),%) -> %
--R sort : % -> % if Boolean has ORDSET
--R sort! : (((Boolean,Boolean) -> Boolean),%) -> % if $ has shallowlyMutable
--R sort! : % -> % if $ has shallowlyMutable and Boolean has ORDSET
--R sorted? : (((Boolean,Boolean) -> Boolean),%) -> Boolean
--R sorted? : % -> Boolean if Boolean has ORDSET
--R swap! : (%,Integer,Integer) -> Void if $ has shallowlyMutable

```

```
--R
--E 1

)spool
)lisp (bye)
```

—————

— Bits.help —

```
=====
Bits examples
=====
```

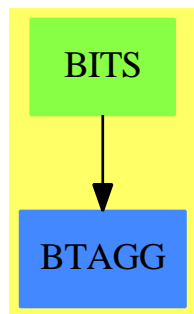
Bits provides logical functions for Indexed Bits.

See Also:

o)show Bits

—————

Bits (BITS)



See

- ⇒ “Reference” (REF) [19](#) on page [2249](#)
- ⇒ “Boolean” (BOOLEAN) [3](#) on page [322](#)
- ⇒ “IndexedBits” (IBITS) [10](#) on page [1199](#)

Exports:

any?	bits	coerce	concat	construct
convert	copy	copyInto!	count	delete
elt	empty	empty?	entries	entry?
eq?	eval	every?	fill!	find
first	hash	index?	indices	insert
latex	less?	map	map!	max
maxIndex	member?	members	merge	min
minIndex	more?	nand	new	nor
not?	parts	position	qelt	qsetelt!
reduce	remove	removeDuplicates	reverse	reverse!
sample	setelt	size?	sort	sort!
sorted?	swap!	xor	#?	??
?/\?	?<?	?<=?	?=?	?>?
?>=?	?\\/?	^?	?and?	?or?
??	~?	?~=?		

— domain BITS Bits —

```

)abbrev domain BITS Bits
++ Author: Stephen M. Watt
++ Description:
++ \spadtype{Bits} provides logical functions for Indexed Bits.

```

```

Bits(): Exports == Implementation where
  Exports == BitAggregate() with
    bits: (NonNegativeInteger, Boolean) -> %
      ++ bits(n,b) creates bits with n values of b
  Implementation == IndexedBits(1) add
    bits(n,b) == new(n,b)

```

— BITS.dotabb —

```

"BITS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=BITS"]
"BTAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BTAGG"]
"BITS" -> "BTAGG"

```

domain BLHN BlowUpWithHamburgerNoether

— BlowUpWithHamburgerNoether.input —

```

)set break resume
)sys rm -f BlowUpWithHamburgerNoether.output
)spool BlowUpWithHamburgerNoether.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show BlowUpWithHamburgerNoether
--R
--R BlowUpWithHamburgerNoether is a domain constructor
--R Abbreviation for BlowUpWithHamburgerNoether is BLHN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for BLHN
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          chartCoord : % -> Integer
--R coerce : List(Integer) -> %      coerce : % -> OutputForm
--R excepCoord : % -> Integer         hash : % -> SingleInteger
--R infClsPt? : % -> Boolean          latex : % -> String
--R quotValuation : % -> Integer      ramifMult : % -> Integer
--R transCoord : % -> Integer         ?~=? : (%,% ) -> Boolean
--R createHN : (Integer,Integer,Integer,Integer,Integer,Boolean,Union(left,center,right,vertical
--R type : % -> Union(left,center,right,vertical,horizontal)
--R
--E 1

)spool
)lisp (bye)

```

— BlowUpWithHamburgerNoether.help —

```

=====
BlowUpWithHamburgerNoether examples
=====

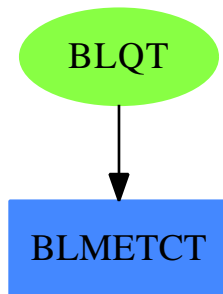
```

This domain is part of the PAFF package

See Also:

o)show BlowUpWithHamburgerNoether

BlowUpWithHamburgerNoether (BLHN)

**Exports:**

?=?	?~=?	chartCoord	coerce	createHN
excepCoord	hash	infClsPt?	latex	quotValuation
ramifMult	transCoord	type		

— domain BLHN BlowUpWithHamburgerNoether —

```

)abbrev domain BLHN BlowUpWithHamburgerNoether
++ Authors: Gaetan Hache
++ Date Created: june 1996
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ This domain is part of the PAFF package
BlowUpWithHamburgerNoether: Exports == Implementation where
  MetRec ==> Record(
    ex:Integer, tr:Integer, ch:Integer , quotVal:Integer, _
    ramif:Integer, infClsPt:Boolean, _
    type:Union("left","center","right","vertical","horizontal") )

Exports ==> BlowUpMethodCategory with HamburgerNoether

Implementation == add
  Rep := MetRec

  infClsPt_? a == a.infClsPt

  createHN( a,b,c,d,e,f,g)==[a,b,c,d,e,f,g]$Rep

  excepCoord a == a.ex

  chartCoord a == a.ch

  transCoord a == a.tr

  ramifMult a == a.ramif

```



```

quotValuation a == a.quotVal

type a == a.type

coerce(c: %): OutputForm == ( (c :: Rep) :: MetRec) :: OutputForm

-----

-- BLHN.dotabb --

"BLHN" [color="#88FF44",href="bookvol10.3.pdf#nameddest=BLHN",
       shape=ellipse]
"BLMETCT" [color="#4488FF",href="bookvol10.3.pdf#nameddest=BLMETCT"]
"BLHN" -> "BLMETCT"

-----

```

domain BLQT BlowUpWithQuadTrans

```

-- BlowUpWithQuadTrans.input --

)set break resume
)sys rm -f BlowUpWithQuadTrans.output
)spool BlowUpWithQuadTrans.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show BlowUpWithQuadTrans
--R
--R BlowUpWithQuadTrans is a domain constructor
--R Abbreviation for BlowUpWithQuadTrans is BLQT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for BLQT
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          chartCoord : % -> Integer
--R coerce : List(Integer) -> %      coerce : % -> OutputForm
--R excepCoord : % -> Integer        hash : % -> SingleInteger
--R infClsPt? : % -> Boolean         latex : % -> String
--R quotValuation : % -> Integer     ramifMult : % -> Integer
--R transCoord : % -> Integer        ?~=? : (%,% ) -> Boolean
--R createHN : (Integer,Integer,Integer,Integer,Integer,Boolean,Union(left,center,right,vertical

```

```
--R type : % -> Union(left,center,right,vertical,horizontal)
--R
--E 1

)spool
)lisp (bye)
```

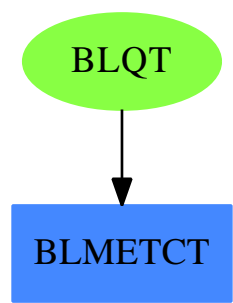
— BlowUpWithQuadTrans.help —

```
=====
BlowUpWithQuadTrans examples
=====

This domain is part of the PAFF package

See Also:
o )show BlowUpWithQuadTrans
```

BlowUpWithQuadTrans (BLQT)



Exports:

?=?	?~=?	chartCoord	coerce	createHN
excepCoord	hash	infClsPt?	latex	quotValuation
ramifMult	transCoord	type		

— domain BLQT BlowUpWithQuadTrans —

```
)abbrev domain BLQT BlowUpWithQuadTrans
```

```

++ Authors: Gaetan Hache
++ Date Created: june 1996
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ This domain is part of the PAFF package
BlowUpWithQuadTrans: Exports == Implementation where

MetRec ==> Record( ex:Integer, tr: Integer, ch: Integer , ramif: Integer )
outRec ==> Record( exCoord:Integer, affNeigh: Integer )
Exports ==> BlowUpMethodCategory with

    QuadraticTransform

Implementation == add
    Rep := MetRec

    coerce(la:List(Integer)):% == [la.1, la.2,la.3, 1 ]$Rep

    ramifMult a == One$Integer

    excepCoord a == a.ex

    chartCoord a == a.ch

    transCoord a == a.tr

    ramifMult a == a.ramif

    quotValuation a == One$Integer

    coerce(c:%):OutputForm==
        oo: outRec := [ excepCoord(c) , chartCoord(c) ]$outRec
        oo :: OutputForm

    -----

    — BLQT.dotabb —

    "BLQT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=BLQT",
            shape=ellipse]
    "BLMETCT" [color="#4488FF",href="bookvol10.3.pdf#nameddest=BLMETCT"]
    "BLQT" -> "BLMETCT"

    -----

```

domain BOOLEAN Boolean

— Boolean.input —

```

)set break resume
)sys rm -f Boolean.output
)spool Boolean.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Boolean
--R Boolean is a domain constructor
--R Abbreviation for Boolean is BOOLEAN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for BOOLEAN
--R
--R----- Operations -----
--R ?/\? : (%,%) -> %           ?<? : (%,%) -> Boolean
--R ?<=? : (%,%) -> Boolean      ?=? : (%,%) -> Boolean
--R ?>? : (%,%) -> Boolean      ?>=? : (%,%) -> Boolean
--R ?\/? : (%,%) -> %           ^? : % -> %
--R ?and? : (%,%) -> %          coerce : % -> OutputForm
--R convert : % -> InputForm    enumerate : () -> List(%)
--R false : () -> %             hash : % -> SingleInteger
--R implies : (%,%) -> %         index : PositiveInteger -> %
--R latex : % -> String          lookup : % -> PositiveInteger
--R max : (%,%) -> %             min : (%,%) -> %
--R nand : (%,%) -> %            nor : (%,%) -> %
--R not? : % -> %                ?or? : (%,%) -> %
--R random : () -> %             size : () -> NonNegativeInteger
--R test : % -> Boolean          true : () -> %
--R xor : (%,%) -> %             ^? : % -> %
--R ?~=? : (%,%) -> Boolean
--R
--E 1

)spool
)lisp (bye)

```

— Boolean.help —

```

=====
Boolean examples
=====

```

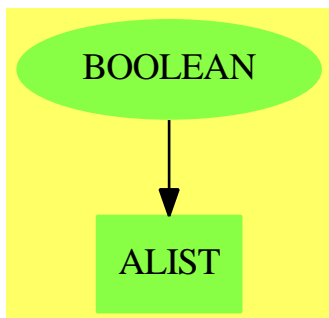
Boolean is the elementary logic with 2 values: true and false

See Also:

o)show Boolean

—————→

Boolean (BOOLEAN)



See

⇒ “Reference” (REF) [19](#) on page [2249](#)

⇒ “IndexedBits” (IBITS) [10](#) on page [1199](#)

⇒ “Bits” (BITS) [3](#) on page [314](#)

Exports:

coerce	convert	false	hash	implies
index	latex	lookup	max	min
nand	nor	not?	random	size
test	true	xor	~?	?~=?
?/\?	?<?	?<=?	?=?	?>?
?>=?	?\/?	^?	?and?	?or?

— domain **BOOLEAN** Boolean —

```
)abbrev domain BOOLEAN Boolean
```

```
++ Author: Stephen M. Watt
```

```
++ Description:
```

```
++ \spadtype{Boolean} is the elementary logic with 2 values:
```

```
++ true and false
```

```
Boolean(): Join(OrderedSet, Finite, Logic, ConvertibleTo InputForm) with
  true   : constant -> %
          ++ true is a logical constant.
  false  : constant -> %
```

```

    ++ false is a logical constant.
_ ^      : % -> %
    ++ ^ n returns the negation of n.
_not : % -> %
    ++ not n returns the negation of n.
_and : (% , %) -> %
    ++ a and b returns the logical and of Boolean \spad{a} and b.
_or : (% , %) -> %
    ++ a or b returns the logical inclusive or
    ++ of Boolean \spad{a} and b.
xor : (% , %) -> %
    ++ xor(a,b) returns the logical exclusive or
    ++ of Boolean \spad{a} and b.
nand : (% , %) -> %
    ++ nand(a,b) returns the logical negation of \spad{a} and b.
nor : (% , %) -> %
    ++ nor(a,b) returns the logical negation of \spad{a} or b.
implies: (% , %) -> %
    ++ implies(a,b) returns the logical implication
    ++ of Boolean \spad{a} and b.
test: % -> Boolean
    ++ test(b) returns b and is provided for compatibility with the
    ++ new compiler.
== add
nt: % -> %

test a      == a pretend Boolean

nt b        == (b pretend Boolean => false; true)
true        == EQ(2,2)$Lisp --well, 1 is rather special
false       == NIL$Lisp
sample()    == true
not b       == (test b => false; true)
_ ^ b       == (test b => false; true)
_ ~ b       == (test b => false; true)
_and(a, b)  == (test a => b; false)
_/_\ (a, b) == (test a => b; false)
_or(a, b)   == (test a => true; b)
_ \_ / (a, b) == (test a => true; b)
xor(a, b)   == (test a => nt b; b)
nor(a, b)   == (test a => false; nt b)
nand(a, b)  == (test a => nt b; true)
a = b       == BooleanEquality(a, b)$Lisp
implies(a, b) == (test a => b; true)
a < b       == (test b => not(test a);false)

size()      == 2
index i     ==
    even?(i::Integer) => false
true

```

```

lookup a      ==
  a pretend Boolean => 1
  2
random()      ==
  even?(random()$Integer) => false
  true

convert(x:):InputForm ==
  x pretend Boolean => convert("true"::Symbol)
  convert("false"::Symbol)

coerce(x:):OutputForm ==
  x pretend Boolean => message "true"
  message "false"

```

— **BOOLEAN.dotabb** —

```

"BOOLEAN" [color="#88FF44",href="bookvol10.3.pdf#nameddest=BOOLEAN",
           shape=ellipse]
"ALIST"   [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"BOOLEAN" -> "ALIST"

```

Chapter 4

Chapter C

domain CARD CardinalNumber

— CardinalNumber.input —

```
)set break resume
)sys rm -f CardinalNumber.output
)spool CardinalNumber.output
)set message test on
)set message auto off
)clear all
--S 1 of 20
c0 := 0 :: CardinalNumber
--R
--R
--R (1) 0
--R
--E 1
```

Type: CardinalNumber

```
--S 2 of 20
c1 := 1 :: CardinalNumber
--R
--R
--R (2) 1
--R
--E 2
```

Type: CardinalNumber

```
--S 3 of 20
c2 := 2 :: CardinalNumber
--R
--R
--R (3) 2
```


[illegible]

```

--S 10 of 20
countable? A0
--R
--R
--R (10) true
--R
--R                                         Type: Boolean
--E 10

```

```

--S 11 of 20
countable? A1
--R
--R
--R (11) false
--R
--R                                         Type: Boolean
--E 11

```

```

--S 12 of 20
[c2 + c2, c2 + A1]
--R
--R
--R (12) [4,Aleph(1)]
--R
--R                                         Type: List(CardinalNumber)
--E 12

```

```

--S 13 of 20
[c0*c2, c1*c2, c2*c2, c0*A1, c1*A1, c2*A1, A0*A1]
--R
--R
--R (13) [0,2,4,0,Aleph(1),Aleph(1),Aleph(1)]
--R
--R                                         Type: List(CardinalNumber)
--E 13

```

```

--S 14 of 20
[c2**c0, c2**c1, c2**c2, A1**c0, A1**c1, A1**c2]
--R
--R
--R (14) [1,2,4,1,Aleph(1),Aleph(1)]
--R
--R                                         Type: List(CardinalNumber)
--E 14

```

```

--S 15 of 20
[c2-c1, c2-c2, c2-c3, A1-c2, A1-A0, A1-A1]
--R
--R
--R (15) [1,0,"failed",Aleph(1),Aleph(1),"failed"]
--R                                         Type: List(Union(CardinalNumber,"failed"))
--E 15

```

```

--S 16 of 20

```

```

generalizedContinuumHypothesisAssumed true
--R
--R
--R (16) true
--R
--R                                          Type: Boolean
--E 16

--S 17 of 20
[c0**A0, c1**A0, c2**A0, A0**A0, A0**A1, A1**A0, A1**A1]
--R
--R
--R (17) [0,1,Aleph(1),Aleph(1),Aleph(2),Aleph(1),Aleph(2)]
--R                                          Type: List(CardinalNumber)
--E 17

--S 18 of 20
a := Aleph 0
--R
--R
--R (18) Aleph(0)
--R
--R                                          Type: CardinalNumber
--E 18

--S 19 of 20
c := 2**a
--R
--R
--R (19) Aleph(1)
--R
--R                                          Type: CardinalNumber
--E 19

--S 20 of 20
f := 2**c
--R
--R
--R (20) Aleph(2)
--R
--R                                          Type: CardinalNumber
--E 20
)spool
)lisp (bye)

```

— CardinalNumber.help —

```

=====
CardinalNumber examples
=====

```

The CardinalNumber domain can be used for values indicating the cardinality of sets, both finite and infinite. For example, the dimension operation in the category VectorSpace returns a cardinal number.

The non-negative integers have a natural construction as cardinals

$$0 = \#\{ \}, 1 = \{0\}, 2 = \{0, 1\}, \dots, n = \{i \mid 0 \leq i < n\}.$$

The fact that 0 acts as a zero for the multiplication of cardinals is equivalent to the axiom of choice.

Cardinal numbers can be created by conversion from non-negative integers.

```
c0 := 0 :: CardinalNumber
0
                                Type: CardinalNumber

c1 := 1 :: CardinalNumber
1
                                Type: CardinalNumber

c2 := 2 :: CardinalNumber
2
                                Type: CardinalNumber

c3 := 3 :: CardinalNumber
3
                                Type: CardinalNumber
```

They can also be obtained as the named cardinal Aleph(n).

```
A0 := Aleph 0
Aleph(0)
                                Type: CardinalNumber

A1 := Aleph 1
Aleph(1)
                                Type: CardinalNumber
```

The finite? operation tests whether a value is a finite cardinal, that is, a non-negative integer.

```
finite? c2
true
                                Type: Boolean

finite? A0
false
                                Type: Boolean
```

Similarly, the `countable?` operation determines whether a value is a countable cardinal, that is, finite or $\text{Aleph}(0)$.

```
countable? c2
true
Type: Boolean
```

```
countable? A0
true
Type: Boolean
```

```
countable? A1
false
Type: Boolean
```

Arithmetic operations are defined on cardinal numbers as follows:
If $x = \#X$ and $y = \#Y$ then

```
x+y = #(X+Y) cardinality of the disjoint union
x-y = #(X-Y) cardinality of the relative complement
x*y = #(X*Y) cardinality of the Cartesian product
x**y = #(X**Y) cardinality of the set of maps from Y to X
```

Here are some arithmetic examples.

```
[c2 + c2, c2 + A1]
[4, Aleph(1)]
Type: List CardinalNumber
```

```
[c0*c2, c1*c2, c2*c2, c0*A1, c1*A1, c2*A1, A0*A1]
[0, 2, 4, 0, Aleph(1), Aleph(1), Aleph(1)]
Type: List CardinalNumber
```

```
[c2**c0, c2**c1, c2**c2, A1**c0, A1**c1, A1**c2]
[1, 2, 4, 1, Aleph(1), Aleph(1)]
Type: List CardinalNumber
```

Subtraction is a partial operation: it is not defined when subtracting a larger cardinal from a smaller one, nor when subtracting two equal infinite cardinals.

```
[c2-c1, c2-c2, c2-c3, A1-c2, A1-A0, A1-A1]
[1, 0, "failed", Aleph(1), Aleph(1), "failed"]
Type: List Union(CardinalNumber,"failed")
```

The generalized continuum hypothesis asserts that

```
2**Aleph i = Aleph(i+1)
```

and is independent of the axioms of set theory.

(reference: Goedel, The consistency of the continuum hypothesis,
Ann. Math. Studies, Princeton Univ. Press, 1940.)

The CardinalNumber domain provides an operation to assert whether the hypothesis is to be assumed.

```
generalizedContinuumHypothesisAssumed true
true
```

Type: Boolean

When the generalized continuum hypothesis is assumed, exponentiation to a transfinite power is allowed.

```
[c0**A0, c1**A0, c2**A0, A0**A0, A0**A1, A1**A0, A1**A1]
[0, 1, Aleph(1), Aleph(1), Aleph(2), Aleph(1), Aleph(2)]
Type: List CardinalNumber
```

Three commonly encountered cardinal numbers are

```
a = #Z countable infinity
c = #R the continuum
f = #{g| g: [0,1] -> R}
```

In this domain, these values are obtained under the generalized continuum hypothesis in this way.

```
a := Aleph 0
Aleph(0)
```

Type: CardinalNumber

```
c := 2**a
Aleph(1)
```

Type: CardinalNumber

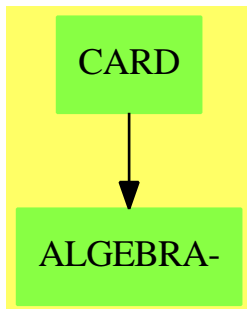
```
f := 2**c
Aleph(2)
```

Type: CardinalNumber

See Also:

```
o )show CardinalNumber
```

CardinalNumber (CARD)



Exports:

0	1
Aleph	coerce
countable?	finite?
generalizedContinuumHypothesisAssumed	hash
generalizedContinuumHypothesisAssumed?	latex
max	min
one?	recip
retract	retractIfCan
sample	zero?
?^?	?~=?
?*?	?**?
?-?	?+?
?<?	?<=?
?=?	?>?
?>=?	

— domain CARD CardinalNumber —

```

)abbrev domain CARD CardinalNumber
++ Author: S.M. Watt
++ Date Created: June 1986
++ Date Last Updated: May 1990
++ References:
++   Goedel, "The consistency of the continuum hypothesis",
++   Ann. Math. Studies, Princeton Univ. Press, 1940
++ Description:
++ Members of the domain CardinalNumber are values indicating the
++ cardinality of sets, both finite and infinite. Arithmetic operations
++ are defined on cardinal numbers as follows.
++
++ If \spad{x = #X} and \spad{y = #Y} then\br
++ \tab{5}\spad{x+y} = #(X+Y)} \tab{5}disjoint union\br
++ \tab{5}\spad{x-y} = #(X-Y)} \tab{5}relative complement\br
  
```

```

++ \tab{5}\spad{x*y = #(X*Y)} \tab{5}cartesian product\br
++ \tab{5}\spad{x**y = #(X**Y)} \tab{4}\spad{X**Y = g| g:Y->X}
++
++ The non-negative integers have a natural construction as cardinals\br
++ \spad{0 = #\{\}}, \spad{1 = \{0\}},
++ \spad{2 = \{0, 1\}}, ..., \spad{n = \{i| 0 <= i < n\}}.
++
++ That \spad{0} acts as a zero for the multiplication of cardinals is
++ equivalent to the axiom of choice.
++
++ The generalized continuum hypothesis asserts \br
++ \spad{2**Aleph i = Aleph(i+1)}
++ and is independent of the axioms of set theory [Goedel 1940].
++
++ Three commonly encountered cardinal numbers are\br
++ \tab{5}\spad{a = #Z} \tab{5}countable infinity\br
++ \tab{5}\spad{c = #R} \tab{5}the continuum\br
++ \tab{5}\spad{f = # g | g:[0,1]->R}\br
++
++ In this domain, these values are obtained using\br
++ \tab{5}\spad{a := Aleph 0}, \spad{c := 2**a}, \spad{f := 2**c}.

CardinalNumber: Join(OrderedSet, AbelianMonoid, Monoid,
    RetractableTo NonNegativeInteger) with
    commutative "*"
    ++ a domain D has \spad{commutative("*")} if it has an operation
    ++ \spad{"*": (D,D) -> D} which is commutative.

    "-": (%,% ) -> Union(%, "failed")
    ++ \spad{x - y} returns an element z such that
    ++ \spad{z+y=x} or "failed" if no such element exists.
    ++
    ++X c2:=2::CardinalNumber
    ++X c2-c2
    ++X A1:=Aleph 1
    ++X A1-c2

    "***": (%,% ) -> %
    ++ \spad{x**y} returns \spad{#(X**Y)} where \spad{X**Y} is defined
    ++ as \spad{\{g| g:Y->X\}}.
    ++
    ++X c2:=2::CardinalNumber
    ++X c2**c2
    ++X A1:=Aleph 1
    ++X A1**c2
    ++X generalizedContinuumHypothesisAssumed true
    ++X A1**A1

Aleph: NonNegativeInteger -> %
    ++ Aleph(n) provides the named (infinite) cardinal number.

```



```

++
++X A0:=Aleph 0

finite?: % -> Boolean
++ finite?(\spad{a}) determines whether
++ \spad{a} is a finite cardinal, i.e. an integer.
++
++X c2:=2::CardinalNumber
++X finite? c2
++X A0:=Aleph 0
++X finite? A0

countable?: % -> Boolean
++ countable?(\spad{a}) determines
++ whether \spad{a} is a countable cardinal,
++ i.e. an integer or \spad{Aleph 0}.
++
++X c2:=2::CardinalNumber
++X countable? c2
++X A0:=Aleph 0
++X countable? A0
++X A1:=Aleph 1
++X countable? A1

generalizedContinuumHypothesisAssumed?: () -> Boolean
++ generalizedContinuumHypothesisAssumed?()
++ tests if the hypothesis is currently assumed.
++
++X generalizedContinuumHypothesisAssumed?

generalizedContinuumHypothesisAssumed: Boolean -> Boolean
++ generalizedContinuumHypothesisAssumed(bool)
++ is used to dictate whether the hypothesis is to be assumed.
++
++X generalizedContinuumHypothesisAssumed true
++X a:=Aleph 0
++X c:=2**a
++X f:=2**c

== add
NNI ==> NonNegativeInteger
FINord ==> -1
DUMMYval ==> -1

Rep := Record(order: Integer, ival: Integer)

GCHypothesis: Reference(Boolean) := ref false

-- Creation
0 == [FINord, 0]
1 == [FINord, 1]

```

```

coerce(n:NonNegativeInteger):% == [FINord, n]
Aleph n      == [n, DUMMYval]

-- Output
ALEPHexpr := "Aleph":OutputForm

coerce(x: %): OutputForm ==
  x.order = FINord => (x.ival)::OutputForm
  prefix(ALEPHexpr, [(x.order)::OutputForm])

-- Manipulation
x = y ==
  x.order ^= y.order => false
  finite? x      => x.ival = y.ival
  true          -- equal transfinities
x < y ==
  x.order < y.order => true
  x.order > y.order => false
  finite? x      => x.ival < y.ival
  false         -- equal transfinities
x:% + y:% ==
  finite? x and finite? y => [FINord, x.ival+y.ival]
  max(x, y)
x - y ==
  x < y      => "failed"
  finite? x => [FINord, x.ival-y.ival]
  x > y      => x
  "failed" -- equal transfinities
x:% * y:% ==
  finite? x and finite? y => [FINord, x.ival*y.ival]
  x = 0 or y = 0          => 0
  max(x, y)
n:NonNegativeInteger * x:% ==
  finite? x => [FINord, n*x.ival]
  n = 0     => 0
  x
x**y ==
  y = 0 =>
    x ^= 0 => 1
    error "0**0 not defined for cardinal numbers."
  finite? y =>
    not finite? x => x
    [FINord,x.ival**(y.ival):NNI]
  x = 0 => 0
  x = 1 => 1
  GCHypothesis() => [max(x.order-1, y.order) + 1, DUMMYval]
  error "Transfinite exponentiation only implemented under GCH"

finite? x      == x.order = FINord
countable? x == x.order < 1

```

```

retract(x: %): NonNegativeInteger ==
  finite? x => (x.ival)::NNI
  error "Not finite"

retractIfCan(x: %): Union(NonNegativeInteger, "failed") ==
  finite? x => (x.ival)::NNI
  "failed"

-- State manipulation
generalizedContinuumHypothesisAssumed?() == GCHypothesis()
generalizedContinuumHypothesisAssumed b == (GCHypothesis() := b)

```

— CARD.dotabb —

```

"CARD" [color="#88FF44",href="bookvol10.3.pdf#nameddest=CARD"]
"ALGEBRA-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALGEBRA"]
"CARD" -> "ALGEBRA-"

```

domain CARTEN CartesianTensor

— CartesianTensor.input —

```

)set break resume
)sys rm -f CartesianTensor.output
)spool CartesianTensor.output
)set message test on
)set message auto off
)clear all
--S 1 of 48
CT := CARTEN(i0 := 1, 2, Integer)
--R
--R
--R (1) CartesianTensor(1,2,Integer)
--R
--E 1

--S 2 of 48
t0: CT := 8
--R

```

Type: Domain

```

--R
--R (2)  8
--R
--R                                         Type: CartesianTensor(1,2,Integer)
--E 2

--S 3 of 48
rank t0
--R
--R
--R (3)  0
--R
--R                                         Type: NonNegativeInteger
--E 3

--S 4 of 48
v: DirectProduct(2, Integer) := directProduct [3,4]
--R
--R
--R (4)  [3,4]
--R
--R                                         Type: DirectProduct(2,Integer)
--E 4

--S 5 of 48
Tv: CT := v
--R
--R
--R (5)  [3,4]
--R
--R                                         Type: CartesianTensor(1,2,Integer)
--E 5

--S 6 of 48
m: SquareMatrix(2, Integer) := matrix [ [1,2],[4,5] ]
--R
--R
--R
--R      +1  2+
--R (6)  |    |
--R      +4  5+
--R
--R                                         Type: SquareMatrix(2,Integer)
--E 6

--S 7 of 48
Tm: CT := m
--R
--R
--R
--R      +1  2+
--R (7)  |    |
--R      +4  5+
--R
--R                                         Type: CartesianTensor(1,2,Integer)
--E 7

--S 8 of 48

```

```

n: SquareMatrix(2, Integer) := matrix [ [2,3],[0,1] ]
--R
--R
--R      +2  3+
--R  (8)  |   |
--R      +0  1+
--R
--R                                          Type: SquareMatrix(2,Integer)
--E 8

--S 9 of 48
Tn: CT := n
--R
--R
--R      +2  3+
--R  (9)  |   |
--R      +0  1+
--R
--R                                          Type: CartesianTensor(1,2,Integer)
--E 9

--S 10 of 48
t1: CT := [2, 3]
--R
--R
--R  (10) [2,3]
--R
--R                                          Type: CartesianTensor(1,2,Integer)
--E 10

--S 11 of 48
rank t1
--R
--R
--R  (11) 1
--R
--R                                          Type: PositiveInteger
--E 11

--S 12 of 48
t2: CT := [t1, t1]
--R
--R
--R      +2  3+
--R  (12) |   |
--R      +2  3+
--R
--R                                          Type: CartesianTensor(1,2,Integer)
--E 12

--S 13 of 48
t3: CT := [t2, t2]
--R
--R
--R      +2  3+ +2  3+

```

```

--R (13) [| |,| |]
--R      +2 3+ +2 3+
--R
--R                                          Type: CartesianTensor(1,2,Integer)
--E 13

--S 14 of 48
tt: CT := [t3, t3]; tt := [tt, tt]
--R
--R
--R      ++2 3+ +2 3++ ++2 3+ +2 3++
--R      || | | || || | | ||
--R      |+2 3+ +2 3+| |+2 3+ +2 3+|
--R (14) [| |,| |]
--R      |+2 3+ +2 3+| |+2 3+ +2 3+|
--R      || | | || || | | ||
--R      ++2 3+ +2 3++ ++2 3+ +2 3++
--R
--R                                          Type: CartesianTensor(1,2,Integer)
--E 14

--S 15 of 48
rank tt
--R
--R
--R (15) 5
--R
--R                                          Type: PositiveInteger
--E 15

--S 16 of 48
Tmn := product(Tm, Tn)
--R
--R
--R      ++2 3+ +4 6+ +
--R      || | | | |
--R      |+0 1+ +0 2+ |
--R (16) | |
--R      |+8 12+ +10 15+|
--R      || | | ||
--R      ++0 4 + +0 5 ++
--R
--R                                          Type: CartesianTensor(1,2,Integer)
--E 16

--S 17 of 48
Tmv := contract(Tm,2,Tv,1)
--R
--R
--R (17) [11,32]
--R
--R                                          Type: CartesianTensor(1,2,Integer)
--E 17

--S 18 of 48

```

```

Tm*Tv
--R
--R
--R (18) [11,32]
--R
--R                                         Type: CartesianTensor(1,2,Integer)
--E 18

--S 19 of 48
Tmv = m * v
--R
--R
--R (19) [11,32]= [11,32]
--R
--R                                         Type: Equation(CartesianTensor(1,2,Integer))
--E 19

--S 20 of 48
t0()
--R
--R
--R (20) 8
--R
--R                                         Type: PositiveInteger
--E 20

--S 21 of 48
t1(1+1)
--R
--R
--R (21) 3
--R
--R                                         Type: PositiveInteger
--E 21

--S 22 of 48
t2(2,1)
--R
--R
--R (22) 2
--R
--R                                         Type: PositiveInteger
--E 22

--S 23 of 48
t3(2,1,2)
--R
--R
--R (23) 3
--R
--R                                         Type: PositiveInteger
--E 23

--S 24 of 48
Tmn(2,1,2,1)
--R

```

```

--R
--R (24)  0
--R
--R                                         Type: NonNegativeInteger
--E 24

--S 25 of 48
t0[]
--R
--R
--R (25)  8
--R
--R                                         Type: PositiveInteger
--E 25

--S 26 of 48
t1[2]
--R
--R
--R (26)  3
--R
--R                                         Type: PositiveInteger
--E 26

--S 27 of 48
t2[2,1]
--R
--R
--R (27)  2
--R
--R                                         Type: PositiveInteger
--E 27

--S 28 of 48
t3[2,1,2]
--R
--R
--R (28)  3
--R
--R                                         Type: PositiveInteger
--E 28

--S 29 of 48
Tmn[2,1,2,1]
--R
--R
--R (29)  0
--R
--R                                         Type: NonNegativeInteger
--E 29

--S 30 of 48
cTmn := contract(Tmn,1,2)
--R
--R
--R
--R          +12  18+

```



```

--R (30) | |
--R      +0 6 +
--R
--R                                          Type: CartesianTensor(1,2,Integer)
--E 30

--S 31 of 48
trace(m) * n
--R
--R
--R      +12 18+
--R (31) | |
--R      +0 6 +
--R
--R                                          Type: SquareMatrix(2,Integer)
--E 31

--S 32 of 48
contract(Tmn,1,2) = trace(m) * n
--R
--R
--R      +12 18+ +12 18+
--R (32) | | = | |
--R      +0 6 + +0 6 +
--R
--R                                          Type: Equation(CartesianTensor(1,2,Integer))
--E 32

--S 33 of 48
contract(Tmn,1,3) = transpose(m) * n
--R
--R
--R      +2 7 + +2 7 +
--R (33) | | = | |
--R      +4 11+ +4 11+
--R
--R                                          Type: Equation(CartesianTensor(1,2,Integer))
--E 33

--S 34 of 48
contract(Tmn,1,4) = transpose(m) * transpose(n)
--R
--R
--R      +14 4+ +14 4+
--R (34) | | = | |
--R      +19 5+ +19 5+
--R
--R                                          Type: Equation(CartesianTensor(1,2,Integer))
--E 34

--S 35 of 48
contract(Tmn,2,3) = m * n
--R
--R
--R      +2 5 + +2 5 +

```

```

--R (35) |      | = |      |
--R      +8 17+ +8 17+
--R
--R                                          Type: Equation(CartesianTensor(1,2,Integer))
--E 35

```

```

--S 36 of 48
contract(Tmn,2,4) = m * transpose(n)
--R
--R
--R      +8  2+  +8  2+
--R (36) |      | = |      |
--R      +23 5+  +23 5+
--R
--R                                          Type: Equation(CartesianTensor(1,2,Integer))
--E 36

```

```

--S 37 of 48
contract(Tmn,3,4) = trace(n) * m
--R
--R
--R      +3  6 +  +3  6 +
--R (37) |      | = |      |
--R      +12 15+ +12 15+
--R
--R                                          Type: Equation(CartesianTensor(1,2,Integer))
--E 37

```

```

--S 38 of 48
tTmn := transpose(Tmn,1,3)
--R
--R
--R      ++2  3 +  +4  6 ++
--R      ||      |  |      ||
--R      |+8 12+ +10 15+|
--R (38) |      |
--R      |+0 1+  +0 2+ |
--R      ||      |  |      ||
--R      ++0 4+  +0 5+ +
--R
--R                                          Type: CartesianTensor(1,2,Integer)
--E 38

```

```

--S 39 of 48
transpose Tmn
--R
--R
--R      ++2  8+  +4 10++
--R      ||      |  |      ||
--R      |+0 0+  +0 0 +|
--R (39) |      |
--R      |+3 12+ +6 15+|
--R      ||      |  |      ||
--R      ++1 4 +  +2 5 ++

```

```
--R                                         Type: CartesianTensor(1,2,Integer)
--E 39
```

```
--S 40 of 48
transpose Tm = transpose m
--R
--R
--R      +1  4+  +1  4+
--R  (40)  |    |=  |    |
--R      +2  5+  +2  5+
--R
--R                                         Type: Equation(CartesianTensor(1,2,Integer))
--E 40
```

```
--S 41 of 48
rTmn := reindex(Tmn, [1,4,2,3])
--R
--R
--R      ++2  0+  +3  1+  +
--R      ||    |  |    |  |
--R      |+4  0+  +6  2+  |
--R  (41)  |                                |
--R      |+8  0+  +12 4+ |
--R      ||    |  |    ||
--R      ++10 0+  +15 5++
--R
--R                                         Type: CartesianTensor(1,2,Integer)
--E 41
```

```
--S 42 of 48
tt := transpose(Tm)*Tn - Tn*transpose(Tm)
--R
--R
--R      +- 6  - 16+
--R  (42)  |    |
--R      + 2    6  +
--R
--R                                         Type: CartesianTensor(1,2,Integer)
--E 42
```

```
--S 43 of 48
Tv*(tt+Tn)
--R
--R
--R  (43)  [- 4, - 11]
--R
--R                                         Type: CartesianTensor(1,2,Integer)
--E 43
```

```
--S 44 of 48
reindex(product(Tn,Tn), [4,3,2,1])+3*Tn*product(Tm,Tm)
--R
--R
--R      ++46  84 +  +57  114++
--R
```

```

--R      ||      | |      ||
--R      |+174 212+ +228 285+|
--R  (44) |      |
--R      | +18 24+   +17 30+ |
--R      | |      | |      | |
--R      + +57 63+   +63 76+ +
--R
--R                                          Type: CartesianTensor(1,2,Integer)
--E 44

```

```

--S 45 of 48
delta: CT := kroneckerDelta()
--R
--R
--R      +1 0+
--R  (45) |  |
--R      +0 1+
--R
--R                                          Type: CartesianTensor(1,2,Integer)
--E 45

```

```

--S 46 of 48
contract(Tmn, 2, delta, 1) = reindex(Tmn, [1,3,4,2])
--R
--R
--R      + +2 4+   +0 0++ + +2 4+   +0 0++
--R      | |      | |      || | |      | |      ||
--R      | +3 6+   +1 2+| | +3 6+   +1 2+|
--R  (46) |      | = |      |
--R      |+8 10+ +0 0+| |+8 10+ +0 0+|
--R      ||      | |      || ||      | |      ||
--R      ++12 15+ +4 5++ ++12 15+ +4 5++
--R
--R                                          Type: Equation(CartesianTensor(1,2,Integer))
--E 46

```

```

--S 47 of 48
epsilon:CT := leviCivitaSymbol()
--R
--R
--R      + 0 1+
--R  (47) |  |
--R      +- 1 0+
--R
--R                                          Type: CartesianTensor(1,2,Integer)
--E 47

```

```

--S 48 of 48
contract(epsilon*Tm*epsilon, 1,2) = 2 * determinant m
--R
--R
--R  (48) - 6= - 6
--R
--R                                          Type: Equation(CartesianTensor(1,2,Integer))
--E 48

```

```
)spool
)lisp (bye)
```

— CartesianTensor.help —

=====
 CartesianTensor examples
 =====

CartesianTensor(i0,dim,R) provides Cartesian tensors with components belonging to a commutative ring R. Tensors can be described as a generalization of vectors and matrices. This gives a concise tensor algebra for multilinear objects supported by the CartesianTensor domain. You can form the inner or outer product of any two tensors and you can add or subtract tensors with the same number of components. Additionally, various forms of traces and transpositions are useful.

The CartesianTensor constructor allows you to specify the minimum index for subscripting. In what follows we discuss in detail how to manipulate tensors.

Here we construct the domain of Cartesian tensors of dimension 2 over the integers, with indices starting at 1.

```
CT := CARTEN(i0 := 1, 2, Integer)
      CartesianTensor(1,2,Integer)
                        Type: Domain
```

=====
 Forming tensors
 =====

Scalars can be converted to tensors of rank zero.

```
t0: CT := 8
      8
                        Type: CartesianTensor(1,2,Integer)

rank t0
      0
                        Type: NonNegativeInteger
```

Vectors (mathematical direct products, rather than one dimensional array structures) can be converted to tensors of rank one.

```
v: DirectProduct(2, Integer) := directProduct [3,4]
      [3, 4]
```

Type: DirectProduct(2,Integer)

Tv: CT := v
[3, 4]

Type: CartesianTensor(1,2,Integer)

Matrices can be converted to tensors of rank two.

m: SquareMatrix(2, Integer) := matrix [[1,2],[4,5]]
+1 2+
| |
+4 5+

Type: SquareMatrix(2,Integer)

Tm: CT := m
+1 2+
| |
+4 5+

Type: CartesianTensor(1,2,Integer)

n: SquareMatrix(2, Integer) := matrix [[2,3],[0,1]]
+2 3+
| |
+0 1+

Type: SquareMatrix(2,Integer)

Tn: CT := n
+2 3+
| |
+0 1+

Type: CartesianTensor(1,2,Integer)

In general, a tensor of rank k can be formed by making a list of rank k-1 tensors or, alternatively, a k-deep nested list of lists.

t1: CT := [2, 3]
[2, 3]

Type: CartesianTensor(1,2,Integer)

rank t1
1

Type: PositiveInteger

t2: CT := [t1, t1]
+2 3+
| |
+2 3+

Type: CartesianTensor(1,2,Integer)

t3: CT := [t2, t2]

```

+2  3+ +2  3+
[|   |,|   |]
+2  3+ +2  3+
Type: CartesianTensor(1,2,Integer)

```

```

tt: CT := [t3, t3]; tt := [tt, tt]
++2  3+ +2  3++ ++2  3+ +2  3++
||   | |   || ||   | |   ||
|+2  3+ +2  3+| |+2  3+ +2  3+|
[|   |,|   |]
|+2  3+ +2  3+| |+2  3+ +2  3+|
||   | |   || ||   | |   ||
++2  3+ +2  3++ ++2  3+ +2  3++
Type: CartesianTensor(1,2,Integer)

```

```

rank tt
5
Type: PositiveInteger

```

Multiplication

Given two tensors of rank k_1 and k_2 , the outer product forms a new tensor of rank k_1+k_2 . Here

```

Tmn(i,j,k,l) = Tm(i,j)Tn(k,l)

Tmn := product(Tm, Tn)
++2  3+   +4  6+ +
||   |   |   ||
|+0  1+   +0  2+ |
|   |   |   |
|+8  12+ +10  15+|
||   |   |   ||
++0  4 +   +0  5 ++
Type: CartesianTensor(1,2,Integer)

```

The inner product (contract) forms a tensor of rank k_1+k_2-2 . This product generalizes the vector dot product and matrix-vector product by summing component products along two indices.

Here we sum along the second index of T_m and the first index of T_v . Here

```

Tmv = sum {j=1..dim} Tm(i,j) Tv(j)

Tmv := contract(Tm,2,Tv,1)
[11,32]
Type: CartesianTensor(1,2,Integer)

```

The multiplication operator `*` is scalar multiplication or an inner product depending on the ranks of the arguments.

If either argument is rank zero it is treated as scalar multiplication. Otherwise, `a*b` is the inner product summing the last index of `a` with the first index of `b`.

```
Tm*Tv
[11,32]
Type: CartesianTensor(1,2,Integer)
```

This definition is consistent with the inner product on matrices and vectors.

```
Tmv = m * v
[11,32] = [11,32]
Type: Equation CartesianTensor(1,2,Integer)
```

```
=====
Selecting Components
=====
```

For tensors of low rank (that is, four or less), components can be selected by applying the tensor to its indices.

```
t0()
8
Type: PositiveInteger

t1(1+1)
3
Type: PositiveInteger

t2(2,1)
2
Type: PositiveInteger

t3(2,1,2)
3
Type: PositiveInteger

Tmn(2,1,2,1)
0
Type: NonNegativeInteger
```

A general indexing mechanism is provided for a list of indices.

```
t0[]
8
```


Type: PositiveInteger

```
t1[2]
  3
```

Type: PositiveInteger

```
t2[2,1]
  2
```

Type: PositiveInteger

The general mechanism works for tensors of arbitrary rank, but is somewhat less efficient since the intermediate index list must be created.

```
t3[2,1,2]
  3
```

Type: PositiveInteger

```
Tmn[2,1,2,1]
  0
```

Type: NonNegativeInteger

```
=====
Contraction
=====
```

A "contraction" between two tensors is an inner product, as we have seen above. You can also contract a pair of indices of a single tensor. This corresponds to a "trace" in linear algebra. The expression `contract(t,k1,k2)` forms a new tensor by summing the diagonal given by indices in position `k1` and `k2`.

This is the tensor given by

```
xTmn = sum{k=1..dim} Tmn(k,k,i,j)
```

```
cTmn := contract(Tmn,1,2)
+12 18+
|    |
+0  6 +
```

Type: CartesianTensor(1,2,Integer)

Since `Tmn` is the outer product of matrix `m` and matrix `n`, the above is equivalent to this.

```
trace(m) * n
+12 18+
|    |
+0  6 +
```

Type: SquareMatrix(2,Integer)

In this and the next few examples, we show all possible contractions

of T_{mn} and their matrix algebra equivalents.

```
contract(Tmn,1,2) = trace(m) * n
+12 18+ +12 18+
|      |= |      |
+0   6+ +0   6+
Type: Equation CartesianTensor(1,2,Integer)
```

```
contract(Tmn,1,3) = transpose(m) * n
+2  7+ +2  7+
|      |= |      |
+4 11+ +4 11+
Type: Equation CartesianTensor(1,2,Integer)
```

```
contract(Tmn,1,4) = transpose(m) * transpose(n)
+14 4+ +14 4+
|      |= |      |
+19 5+ +19 5+
Type: Equation CartesianTensor(1,2,Integer)
```

```
contract(Tmn,2,3) = m * n
+2  5+ +2  5+
|      |= |      |
+8 17+ +8 17+
Type: Equation CartesianTensor(1,2,Integer)
```

```
contract(Tmn,2,4) = m * transpose(n)
+8  2+ +8  2+
|      |= |      |
+23 5+ +23 5+
Type: Equation CartesianTensor(1,2,Integer)
```

```
contract(Tmn,3,4) = trace(n) * m
+3  6+ +3  6+
|      |= |      |
+12 15+ +12 15+
Type: Equation CartesianTensor(1,2,Integer)
```

Transpositions

You can exchange any desired pair of indices using the transpose operation.

Here the indices in positions one and three are exchanged, that is,
 $tTmn(i,j,k,l) = Tmn(k,j,i,l)$

```
tTmn := transpose(Tmn,1,3)
++2  3+ ++4  6++
```

```

||      |  |      ||
|+8  12+  +10  15+|
|      |      |
|+0  1+    +0  2+ |
||      |  |      ||
++0  4+    +0  5+ +

```

Type: CartesianTensor(1,2,Integer)

If no indices are specified, the first and last index are exchanged.

```

transpose Tmn
++2  8+  +4  10++
||      |  |      ||
|+0  0+  +0  0 +|
|      |      |
|+3  12+  +6  15+|
||      |  |      ||
++1  4 +  +2  5 ++

```

Type: CartesianTensor(1,2,Integer)

This is consistent with the matrix transpose.

```

transpose Tm = transpose m
+1  4+  +1  4+
|      | = |      |
+2  5+  +2  5+

```

Type: Equation CartesianTensor(1,2,Integer)

If a more complicated reordering of the indices is required, then the `reindex` operation can be used. This operation allows the indices to be arbitrarily permuted.

```

rTmn(i,j,k,l) = Tmn(i,l,j,k)

```

```

rTmn := reindex(Tmn, [1,4,2,3])

```

```

++2  0+  +3  1+ +
||      |  |      ||
|+4  0+  +6  2+ |
|      |      |
|+8  0+  +12  4+|
||      |  |      ||
++10  0+  +15  5++

```

Type: CartesianTensor(1,2,Integer)

Arithmetic

Tensors of equal rank can be added or subtracted so arithmetic

expressions can be used to produce new tensors.

```
tt := transpose(Tm)*Tn - Tn*transpose(Tm)
+- 6 - 16+
|      |
+ 2    6 +
Type: CartesianTensor(1,2,Integer)
```

```
Tv*(tt+Tn)
[- 4,- 11]
Type: CartesianTensor(1,2,Integer)
```

```
reindex(product(Tn,Tn),[4,3,2,1])+3*Tn*product(Tm,Tm)
++46 84 + +57 114++
||      | |      ||
|+174 212+ +228 285+|
|      |
| +18 24+ +17 30+ |
| |      | |      |
+ +57 63+ +63 76+ +
Type: CartesianTensor(1,2,Integer)
```

Specific Tensors

Two specific tensors have properties which depend only on the dimension.

The Kronecker delta satisfies

$$\text{delta}(i,j) = \begin{array}{cc} \begin{array}{c} +- \\ | \\ +- \end{array} & \begin{array}{c} 1 \text{ if } i = j \\ 0 \text{ if } i \neq j \end{array} & \begin{array}{c} -+ \\ | \\ -+ \end{array} \end{array}$$

```
delta: CT := kroneckerDelta()
+1 0+
|   |
+0 1+
Type: CartesianTensor(1,2,Integer)
```

This can be used to reindex via contraction.

```
contract(Tmn, 2, delta, 1) = reindex(Tmn, [1,3,4,2])
+ +2 4+ +0 0++ + +2 4+ +0 0++
| |      | |      | |      | |
| +3 6+ +1 2+| | +3 6+ +1 2+|
```

```

      |      |      |      | | | | | | | | |
      |+8    10+   +0   0+|  |+8    10+   +0   0+|
      ||      |   |   ||   ||      |   |   ||
      ++12   15+  +4   5++  ++12   15+  +4   5++
                                Type: Equation CartesianTensor(1,2,Integer)

```

The Levi Civita symbol determines the sign of a permutation of indices.

```

epsilon:CT := leviCivitaSymbol()
+ 0      1+
|         |
+- 1      0+
                                Type: CartesianTensor(1,2,Integer)

```

Here we have:

```

epsilon(i1,...,idim)
= +1 if i1,...,idim is an even permutation of i0,...,i0+dim-1
= -1 if i1,...,idim is an odd permutation of i0,...,i0+dim-1
= 0 if i1,...,idim is not a permutation of i0,...,i0+dim-1

```

This property can be used to form determinants.

```

contract(epsilon*Tm*epsilon, 1,2) = 2 * determinant m
- 6= - 6
                                Type: Equation CartesianTensor(1,2,Integer)

```

Properties of the CartesianTensor domain

`GradedModule(R,E)` denotes "E-graded R-module", that is, a collection of R-modules indexed by an abelian monoid E. An element g of $G[s]$ for some specific s in E is said to be an element of G with degree s . Sums are defined in each module $G[s]$ so two elements of G can be added if they have the same degree. Morphisms can be defined and composed by degree to give the mathematical category of graded modules.

`GradedAlgebra(R,E)` denotes "E-graded R-algebra". A graded algebra is a graded module together with a degree preserving R-bilinear map, called the product.

```

degree(product(a,b))    = degree(a) + degree(b)

product(r*a,b)          = product(a,r*b) = r*product(a,b)
product(a1+a2,b)        = product(a1,b) + product(a2,b)
product(a,b1+b2)         = product(a,b1) + product(a,b2)
product(a,product(b,c)) = product(product(a,b),c)

```

The domain `CartesianTensor(i0, dim, R)` belongs to the category `GradedAlgebra(R, NonNegativeInteger)`. The non-negative integer degree is the tensor rank and the graded algebra product is the tensor outer product. The graded module addition captures the notion that only tensors of equal rank can be added.

If V is a vector space of dimension dim over R , then the tensor module $T[k](V)$ is defined as

$$\begin{aligned} T[0](V) &= R \\ T[k](V) &= T[k-1](V) * V \end{aligned}$$

where $*$ denotes the R -module tensor product. `CartesianTensor(i0,dim,R)` is the graded algebra in which the degree k module is $T[k](V)$.

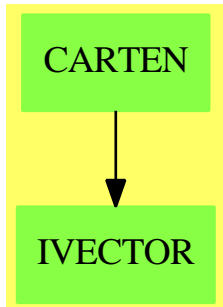
Tensor Calculus

It should be noted here that often tensors are used in the context of tensor-valued manifold maps. This leads to the notion of covariant and contravariant bases with tensor component functions transforming in specific ways under a change of coordinates on the manifold. This is no more directly supported by the `CartesianTensor` domain than it is by the `Vector` domain. However, it is possible to have the components implicitly represent component maps by choosing a polynomial or expression type for the components. In this case, it is up to the user to satisfy any constraints which arise on the basis of this interpretation.

See Also

`o)show CartesianTensor`

CartesianTensor (CARTEN)



Exports:

0	1	coerce	contract	degree
elt	hash	kroneckerDelta	latex	leviCivitaSymbol
product	rank	ravel	reindex	retract
retractIfCan	sample	transpose	unravel	?~=?
?.?	?*?	?+?	?-?	-?
?=?				

— domain CARTEN CartesianTensor —

```

)abbrev domain CARTEN CartesianTensor
++ Author: Stephen M. Watt
++ Date Created: December 1986
++ Date Last Updated: May 15, 1991
++ Description:
++ CartesianTensor(minix,dim,R) provides Cartesian tensors with
++ components belonging to a commutative ring R. These tensors
++ can have any number of indices. Each index takes values from
++ \spad{minix} to \spad{minix + dim - 1}.

CartesianTensor(minix, dim, R): Exports == Implementation where
  NNI ==> NonNegativeInteger
  I   ==> Integer
  DP  ==> DirectProduct
  SM  ==> SquareMatrix

  minix: Integer
  dim: NNI
  R: CommutativeRing

Exports ==> Join(GradedAlgebra(R, NNI), GradedModule(I, NNI)) with

  coerce: DP(dim, R) -> %
    ++ coerce(v) views a vector as a rank 1 tensor.
    ++

```

```

++X v:DirectProduct(2,Integer):=directProduct [3,4]
++X tv:CartesianTensor(1,2,Integer):=v

coerce: SM(dim, R) -> %
++ coerce(m) views a matrix as a rank 2 tensor.
++
++X v:SquareMatrix(2,Integer):=[[1,2],[3,4]]
++X tv:CartesianTensor(1,2,Integer):=v

coerce: List R -> %
++ coerce([r_1,...,r_dim]) allows tensors to be constructed
++ using lists.
++
++X v:=[2,3]
++X tv:CartesianTensor(1,2,Integer):=v

coerce: List % -> %
++ coerce([t_1,...,t_dim]) allows tensors to be constructed
++ using lists.
++
++X v:=[2,3]
++X tv:CartesianTensor(1,2,Integer):=v
++X tm:CartesianTensor(1,2,Integer):=[tv,tv]

rank: % -> NNI
++ rank(t) returns the tensorial rank of t (that is, the
++ number of indices). This is the same as the graded module
++ degree.
++
++X CT:=CARTEN(1,2,Integer)
++X t0:CT:=8
++X rank t0

elt: (%) -> R
++ elt(t) gives the component of a rank 0 tensor.
++
++X tv:CartesianTensor(1,2,Integer):=8
++X elt(tv)
++X tv[]

elt: (% , I) -> R
++ elt(t,i) gives a component of a rank 1 tensor.
++
++X v:=[2,3]
++X tv:CartesianTensor(1,2,Integer):=v
++X elt(tv,2)
++X tv[2]

elt: (% , I, I) -> R
++ elt(t,i,j) gives a component of a rank 2 tensor.

```



```

++
++X v:=[2,3]
++X tv:CartesianTensor(1,2,Integer):=v
++X tm:CartesianTensor(1,2,Integer):=[tv,tv]
++X elt(tm,2,2)
++X tm[2,2]

elt: (%, I, I, I) -> R
++ elt(t,i,j,k) gives a component of a rank 3 tensor.
++
++X v:=[2,3]
++X tv:CartesianTensor(1,2,Integer):=v
++X tm:CartesianTensor(1,2,Integer):=[tv,tv]
++X tn:CartesianTensor(1,2,Integer):=[tm,tm]
++X elt(tn,2,2,2)
++X tn[2,2,2]

elt: (%, I, I, I, I) -> R
++ elt(t,i,j,k,l) gives a component of a rank 4 tensor.
++
++X v:=[2,3]
++X tv:CartesianTensor(1,2,Integer):=v
++X tm:CartesianTensor(1,2,Integer):=[tv,tv]
++X tn:CartesianTensor(1,2,Integer):=[tm,tm]
++X tp:CartesianTensor(1,2,Integer):=[tn,tn]
++X elt(tp,2,2,2,2)
++X tp[2,2,2,2]

elt: (%, List I) -> R
++ elt(t,[i1,...,iN]) gives a component of a rank \spad{N} tensor.
++
++X v:=[2,3]
++X tv:CartesianTensor(1,2,Integer):=v
++X tm:CartesianTensor(1,2,Integer):=[tv,tv]
++X tn:CartesianTensor(1,2,Integer):=[tm,tm]
++X tp:CartesianTensor(1,2,Integer):=[tn,tn]
++X tq:CartesianTensor(1,2,Integer):=[tp,tp]
++X elt(tq,[2,2,2,2,2])

-- This specializes the documentation from GradedAlgebra.
product: (%,%) -> %
++ product(s,t) is the outer product of the tensors s and t.
++ For example, if \spad{r = product(s,t)} for rank 2 tensors
++ s and t, then \spad{r} is a rank 4 tensor given by
++ \spad{r(i,j,k,l) = s(i,j)*t(k,l)}.
++
++X m:SquareMatrix(2,Integer):=matrix [[1,2],[4,5]]
++X Tm:CartesianTensor(1,2,Integer):=m
++X n:SquareMatrix(2,Integer):=matrix [[2,3],[0,1]]
++X Tn:CartesianTensor(1,2,Integer):=n

```

```

++X Tmn:=product(Tm,Tn)

"*.": (% , %) -> %
++ s*t is the inner product of the tensors s and t which contracts
++ the last index of s with the first index of t, that is,
++ \spad{t*s = contract(t,rank t, s, 1)}
++ \spad{t*s = sum(k=1..N, t[i1,..,iN,k]*s[k,j1,..,jM])}
++ This is compatible with the use of \spad{M*v} to denote
++ the matrix-vector inner product.
++
++X m:SquareMatrix(2,Integer):=matrix [[1,2],[4,5]]
++X Tm:CartesianTensor(1,2,Integer):=m
++X v:DirectProduct(2,Integer):=directProduct [3,4]
++X Tv:CartesianTensor(1,2,Integer):=v
++X Tm*Tv

contract: (% , Integer, % , Integer) -> %
++ contract(t,i,s,j) is the inner product of tensors s and t
++ which sums along the \spad{k1}-th index of
++ t and the \spad{k2}-th index of s.
++ For example, if \spad{r = contract(s,2,t,1)} for rank 3 tensors
++ rank 3 tensors \spad{s} and \spad{t}, then \spad{r} is
++ the rank 4 \spad{(= 3 + 3 - 2)} tensor given by
++ \spad{r(i,j,k,l) = sum(h=1..dim,s(i,h,j)*t(h,k,l))}.
++
++X m:SquareMatrix(2,Integer):=matrix [[1,2],[4,5]]
++X Tm:CartesianTensor(1,2,Integer):=m
++X v:DirectProduct(2,Integer):=directProduct [3,4]
++X Tv:CartesianTensor(1,2,Integer):=v
++X Tmv:=contract(Tm,2,Tv,1)

contract: (% , Integer, Integer) -> %
++ contract(t,i,j) is the contraction of tensor t which
++ sums along the \spad{i}-th and \spad{j}-th indices.
++ For example, if
++ \spad{r = contract(t,1,3)} for a rank 4 tensor t, then
++ \spad{r} is the rank 2 \spad{(= 4 - 2)} tensor given by
++ \spad{r(i,j) = sum(h=1..dim,t(h,i,h,j))}.
++
++X m:SquareMatrix(2,Integer):=matrix [[1,2],[4,5]]
++X Tm:CartesianTensor(1,2,Integer):=m
++X v:DirectProduct(2,Integer):=directProduct [3,4]
++X Tv:CartesianTensor(1,2,Integer):=v
++X Tmv:=contract(Tm,2,1)

transpose: % -> %
++ transpose(t) exchanges the first and last indices of t.
++ For example, if \spad{r = transpose(t)} for a rank 4
++ tensor t, then \spad{r} is the rank 4 tensor given by
++ \spad{r(i,j,k,l) = t(l,j,k,i)}.

```

```

++
++X m:SquareMatrix(2,Integer):=matrix [[1,2],[4,5]]
++X Tm:CartesianTensor(1,2,Integer):=m
++X transpose(Tm)

transpose: (% , Integer, Integer) -> %
++ transpose(t,i,j) exchanges the \spad{i}-th and \spad{j}-th
++ indices of t. For example, if \spad{r = transpose(t,2,3)}
++ for a rank 4 tensor t, then \spad{r} is the rank 4 tensor
++ given by
++   \spad{r(i,j,k,l) = t(i,k,j,l)}.
++
++X m:SquareMatrix(2,Integer):=matrix [[1,2],[4,5]]
++X tm:CartesianTensor(1,2,Integer):=m
++X tn:CartesianTensor(1,2,Integer):=[tm,tm]
++X transpose(tn,1,2)

reindex: (% , List Integer) -> %
++ reindex(t,[i1,...,idim]) permutes the indices of t.
++ For example, if \spad{r = reindex(t, [4,1,2,3])}
++ for a rank 4 tensor t,
++ then \spad{r} is the rank for tensor given by
++   \spad{r(i,j,k,l) = t(1,i,j,k)}.
++
++X n:SquareMatrix(2,Integer):=matrix [[2,3],[0,1]]
++X tn:CartesianTensor(1,2,Integer):=n
++X p:=product(tn,tn)
++X reindex(p,[4,3,2,1])

kroneckerDelta: () -> %
++ kroneckerDelta() is the rank 2 tensor defined by
++   \spad{kroneckerDelta()(i,j)}
++   \spad{= 1 if i = j}
++   \spad{= 0 if i != j}
++
++X delta:CartesianTensor(1,2,Integer):=kroneckerDelta()

leviCivitaSymbol: () -> %
++ leviCivitaSymbol() is the rank \spad{dim} tensor defined by
++ \spad{leviCivitaSymbol()(i1,...,idim) = +1/0/-1}
++ if \spad{i1,...,idim} is an even/is nota /is an odd permutation
++ of \spad{minix,...,minix+dim-1}.
++
++X lcs:CartesianTensor(1,2,Integer):=leviCivitaSymbol()

ravel:      % -> List R
++ ravel(t) produces a list of components from a tensor such that
++   \spad{unravel(ravel(t)) = t}.
++
++X n:SquareMatrix(2,Integer):=matrix [[2,3],[0,1]]

```

```

++X tn:CartesianTensor(1,2,Integer):=n
++X ravel tn

unravel: List R -> %
++ unravel(t) produces a tensor from a list of
++ components such that
++ \spad{unravel(ravel(t)) = t}.

sample: () -> %
++ sample() returns an object of type %.

Implementation ==> add

PERM ==> Vector Integer -- 1-based entries from 1..n
INDEX ==> Vector Integer -- 1-based entries from minix..minix+dim-1

get ==> elt$Rep
set_! ==> setelt$Rep

-- Use row-major order:
-- x[h,i,j] <-> x[(h-minix)*dim**2+(i-minix)*dim+(j-minix)]

Rep := IndexedVector(R,0)

n: Integer
r,s: R
x,y,z: %

---- Local stuff
dim2: NNI := dim**2
dim3: NNI := dim**3
dim4: NNI := dim**4

sample()==kroneckerDelta()$%
int2index(n: Integer, indv: INDEX): INDEX ==
  n < 0 => error "Index error (too small)"
  rnk := #indv
  for i in 1..rnk repeat
    qr := divide(n, dim)
    n := qr.quotient
    indv.((rnk-i+1) pretend NNI) := qr.remainder + minix
  n ^= 0 => error "Index error (too big)"
  indv

index2int(indv: INDEX): Integer ==
  n: I := 0
  for i in 1..#indv repeat
    ix := indv.i - minix
    ix<0 or ix>dim-1 => error "Index error (out of range)"

```

```

    n := dim*n + ix
  n

lengthRankOrElse(v: Integer): NNI ==
  v = 1    => 0
  v = dim  => 1
  v = dim2 => 2
  v = dim3 => 3
  v = dim4 => 4
  rx := 0
  while v ^= 0 repeat
    qr := divide(v, dim)
    v := qr.quotient
    if v ^= 0 then
      qr.remainder ^= 0 => error "Rank is not a whole number"
      rx := rx + 1
  rx

-- l must be a list of the numbers 1..#l
mkPerm(n: NNI, l: List Integer): PERM ==
  #l ^= n =>
    error "The list is not a permutation."
  p: PERM := new(n, 0)
  seen: Vector Boolean := new(n, false)
  for i in 1..n for e in l repeat
    e < 1 or e > n => error "The list is not a permutation."
    p.i := e
    seen.e := true
  for e in 1..n repeat
    not seen.e => error "The list is not a permutation."
  p

-- permute s according to p into result t.
permute_!(t: INDEX, s: INDEX, p: PERM): INDEX ==
  for i in 1..#p repeat t.i := s.(p.i)
  t

-- permsign!(v) = 1, 0, or -1 according as
-- v is an even, is not, or is an odd permutation of minix..minix+#v-1.
permsign_!(v: INDEX): Integer ==
  -- sum minix..minix+#v-1.
  maxix := minix+#v-1
  psum := (((maxix+1)*maxix - minix*(minix-1)) exquo 2)::Integer
  -- +/v ^= psum => 0
  n := 0
  for i in 1..#v repeat n := n + v.i
  n ^= psum => 0
  -- Bubble sort! This is pretty grotesque.
  totTrans: Integer := 0
  nTrans: Integer := 1

```

```

while nTrans ^= 0 repeat
  nTrans := 0
  for i in 1..#v-1 for j in 2..#v repeat
    if v.i > v.j then
      nTrans := nTrans + 1
      e := v.i; v.i := v.j; v.j := e
  totTrans := totTrans + nTrans
for i in 1..dim repeat
  if v.i ^= minix+i-1 then return 0
odd? totTrans => -1
1

---- Exported functions
ravel x ==
  [get(x,i) for i in 0..#x-1]

unravel l ==
  -- lengthRankOrElse #l gives sytnax error
  nz: NNI := # l
  lengthRankOrElse nz
  z := new(nz, 0)
  for i in 0..nz-1 for r in l repeat set_!(z, i, r)
  z

kroneckerDelta() ==
  z := new(dim2, 0)
  for i in 1..dim for zi in 0.. by (dim+1) repeat set_!(z, zi, 1)
  z

leviCivitaSymbol() ==
  nz := dim**dim
  z := new(nz, 0)
  indv: INDEX := new(dim, 0)
  for i in 0..nz-1 repeat
    set_!(z, i, permsign_!(int2index(i, indv))::R)
  z

-- from GradedModule
degree x ==
  rank x

rank x ==
  n := #x
  lengthRankOrElse n

elt(x) ==
  #x ^= 1 => error "Index error (the rank is not 0)"
  get(x,0)
elt(x, i: I) ==
  #x ^= dim => error "Index error (the rank is not 1)"

```

```

    get(x,(i-minix))
elt(x, i: I, j: I) ==
    #x ^= dim2 => error "Index error (the rank is not 2)"
    get(x,(dim*(i-minix) + (j-minix)))
elt(x, i: I, j: I, k: I) ==
    #x ^= dim3 => error "Index error (the rank is not 3)"
    get(x,(dim2*(i-minix) + dim*(j-minix) + (k-minix)))
elt(x, i: I, j: I, k: I, l: I) ==
    #x ^= dim4 => error "Index error (the rank is not 4)"
    get(x,(dim3*(i-minix)+dim2*(j-minix)+dim*(k-minix)+(l-minix)))

elt(x, i: List I) ==
    #i ^= rank x => error "Index error (wrong rank)"
    n: I := 0
    for ii in i repeat
        ix := ii - minix
        ix<0 or ix>dim-1 => error "Index error (out of range)"
        n := dim*n + ix
    get(x,n)

coerce(lr: List R): % ==
    #lr ^= dim => error "Incorrect number of components"
    z := new(dim, 0)
    for r in lr for i in 0..dim-1 repeat set_!(z, i, r)
    z

coerce(lx: List %): % ==
    #lx ^= dim => error "Incorrect number of slices"
    rx := rank first lx
    for x in lx repeat
        rank x ^= rx => error "Inhomogeneous slice ranks"
    nx := # first lx
    z := new(dim * nx, 0)
    for x in lx for offz in 0.. by nx repeat
        for i in 0..nx-1 repeat set_!(z, offz + i, get(x,i))
    z

retractIfCan(x:%):Union(R,"failed") ==
    zero? rank(x) => x()
    "failed"
Outf ==> OutputForm

mkOutf(x:%, i0:I, rnk:NNI): Outf ==
    odd? rnk =>
        rnk1 := (rnk-1) pretend NNI
        nskip := dim**rnk1
        [mkOutf(x, i0+nskip*i, rnk1) for i in 0..dim-1]::Outf
    rnk = 0 =>
        get(x,i0)::Outf
    rnk1 := (rnk-2) pretend NNI
    nskip := dim**rnk1

```

```

matrix [[mkOutf(x, i0+nskip*(dim*i + j), rnk1)
        for j in 0..dim-1] for i in 0..dim-1]
coerce(x): Outf ==
  mkOutf(x, 0, rank x)

0 == 0$R::Rep
1 == 1$R::Rep

--coerce(n: I): % == new(1, n::R)
coerce(r: R): % == new(1,r)

coerce(v: DP(dim,R)): % ==
  z := new(dim, 0)
  for i in 0..dim-1 for j in minIndex v .. maxIndex v repeat
    set_!(z, i, v.j)
  z
coerce(m: SM(dim,R)): % ==
  z := new(dim**2, 0)
  offz := 0
  for i in 0..dim-1 repeat
    for j in 0..dim-1 repeat
      set_!(z, offz + j, m(i+1,j+1))
    offz := offz + dim
  z

x = y ==
  #x ^= #y => false
  for i in 0..#x-1 repeat
    if get(x,i) ^= get(y,i) then return false
  true
x + y ==
  #x ^= #y => error "Rank mismatch"
  -- z := [xi + yi for xi in x for yi in y]
  z := new(#x, 0)
  for i in 0..#x-1 repeat set_!(z, i, get(x,i) + get(y,i))
  z
x - y ==
  #x ^= #y => error "Rank mismatch"
  -- [xi - yi for xi in x for yi in y]
  z := new(#x, 0)
  for i in 0..#x-1 repeat set_!(z, i, get(x,i) - get(y,i))
  z
- x ==
  -- [-xi for xi in x]
  z := new(#x, 0)
  for i in 0..#x-1 repeat set_!(z, i, -get(x,i))
  z
n * x ==
  -- [n * xi for xi in x]
  z := new(#x, 0)

```



```

    for i in 0..#x-1 repeat set_!(z, i, n * get(x,i))
  z
x * n ==
  -- [n * xi for xi in x]
  z := new(#x, 0)
  for i in 0..#x-1 repeat set_!(z, i, n* get(x,i))  -- Commutative!!
  z
r * x ==
  -- [r * xi for xi in x]
  z := new(#x, 0)
  for i in 0..#x-1 repeat set_!(z, i, r * get(x,i))
  z
x * r ==
  -- [xi*r for xi in x]
  z := new(#x, 0)
  for i in 0..#x-1 repeat set_!(z, i, r* get(x,i))  -- Commutative!!
  z
product(x, y) ==
  nx := #x; ny := #y
  z := new(nx * ny, 0)
  for i in 0..nx-1 for ioff in 0.. by ny repeat
    for j in 0..ny-1 repeat
      set_!(z, ioff + j, get(x,i) * get(y,j))
  z
x * y ==
  rx := rank x
  ry := rank y
  rx = 0 => get(x,0) * y
  ry = 0 => x * get(y,0)
  contract(x, rx, y, 1)

contract(x, i, j) ==
  rx := rank x
  i < 1 or i > rx or j < 1 or j > rx or i = j =>
    error "Improper index for contraction"
  if i > j then (i,j) := (j,i)

  rl:= (rx- j) pretend NNI; nl:= dim**rl; zol:= 1;      xol:= zol
  rm:= (j-i-1) pretend NNI; nm:= dim**rm; zom:= nl;      xom:= zom*dim
  rh:= (i - 1) pretend NNI; nh:= dim**rh; zoh:= nl*nm
  xoh:= zoh*dim**2
  xok := nl*(1 + nm*dim)
  z := new(nl*nm*nh, 0)
  for h in 1..nh _
  for xh in 0.. by xoh for zh in 0.. by zoh repeat
    for m in 1..nm _
    for xm in xh.. by xom for zm in zh.. by zom repeat
      for l in 1..nl _
      for xl in xm.. by xol for zl in zm.. by zol repeat
        set_!(z, zl, 0)

```

```

        for k in 1..dim for xk in xl.. by xok repeat
            set_!(z, zl, get(z,zl) + get(x,xk))
z

contract(x, i, y, j) ==
    rx := rank x
    ry := rank y

    i < 1 or i > rx or j < 1 or j > ry =>
        error "Improper index for contraction"

    rly:= (ry-j) pretend NNI; nly:= dim**rly; oly:= 1;    zoly:= 1
    rhy:= (j -1) pretend NNI; nhx:= dim**rhy
    ohy:= nly*dim; zohy:= zoly*nly
    rlx:= (rx-i) pretend NNI; nlx:= dim**rlx
    olx:= 1;        zolx:= zohy*nhy
    rhx:= (i -1) pretend NNI; nhx:= dim**rhx
    ohx:= nlx*dim;  zohx:= zolx*nlx

    z := new(nlx*nhx*nly*nhy, 0)

    for dxh in 1..nhx _
    for xh in 0.. by ohx for zhx in 0.. by zohx repeat
        for dxl in 1..nlx _
        for xl in xh.. by olx for zlx in zhx.. by zolx repeat
            for dyh in 1..nhy _
            for yh in 0.. by ohy for zhy in zlx.. by zohy repeat
                for dyl in 1..nly _
                for yl in yh.. by oly for zly in zhy.. by zoly repeat
                    set_!(z, zly, 0)
                    for k in 1..dim _
                    for xk in xl.. by nlx for yk in yl.. by nly repeat
                        set_!(z, zly, get(z,zly)+get(x,xk)*get(y,yk))
z

transpose x ==
    transpose(x, 1, rank x)
transpose(x, i, j) ==
    rx := rank x
    i < 1 or i > rx or j < 1 or j > rx or i = j =>
        error "Improper indicies for transposition"
    if i > j then (i,j) := (j,i)

    rl:= (rx- j) pretend NNI; nl:= dim**rl; zol:= 1;        zoi := zol*nl
    rm:= (j-i-1) pretend NNI; nm:= dim**rm; zom:= nl*dim; zoj := zom*nm
    rh:= (i - 1) pretend NNI; nh:= dim**rh; zoh:= nl*nm*dim**2
    z := new(#x, 0)
    for h in 1..nh for zh in 0.. by zoh repeat _
    for m in 1..nm for zm in zh.. by zom repeat _
    for l in 1..nl for zl in zm.. by zol repeat _

```

```

        for p in 1..dim _
        for zp in zl.. by zoi for xp in zl.. by zoj repeat
            for q in 1..dim _
            for zq in zp.. by zoj for xq in xp.. by zoi repeat
                set_!(z, zq, get(x,xq))
z

reindex(x, l) ==
nx := #x
z: % := new(nx, 0)

rx := rank x
p := mkPerm(rx, l)
xiv: INDEX := new(rx, 0)
ziv: INDEX := new(rx, 0)

-- Use permutation
for i in 0..#x-1 repeat
    pi := index2int(permute_!(ziv, int2index(i,xiv),p))
    set_!(z, pi, get(x,i))
z

-----

— CARTEN.dotabb —

"CARTEN" [color="#88FF44",href="bookvol10.3.pdf#nameddest=CARTEN"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"CARTEN" -> "IVECTOR"

-----

```

domain CHAR Character

— Character.input —

```

)set break resume
)sys rm -f Character.output
)spool Character.output
)set message test on
)set message auto off
)clear all
--S 1 of 13
chars := [char "a", char "A", char "X", char "8", char "+"]

```

```
--R
--R
--R (1) [a,A,X,8,+]
--R
--R                                         Type: List(Character)
--E 1

--S 2 of 13
space()
--R
--R
--R (2)
--R
--R                                         Type: Character
--E 2

--S 3 of 13
quote()
--R
--R
--R (3) "
--R
--R                                         Type: Character
--E 3

--S 4 of 13
escape()
--R
--R
--R (4) _
--R
--R                                         Type: Character
--E 4

--S 5 of 13
[ord c for c in chars]
--R
--R
--R (5) [97,65,88,56,43]
--R
--R                                         Type: List(Integer)
--E 5

--S 6 of 13
[upperCase c for c in chars]
--R
--R
--R (6) [A,A,X,8,+]
--R
--R                                         Type: List(Character)
--E 6

--S 7 of 13
[lowerCase c for c in chars]
--R
--R
```

```

--R (7) [a,a,x,8,+]
--R
--E 7
Type: List(Character)

--S 8 of 13
[alphabetic? c for c in chars]
--R
--R
--R (8) [true,true,true,false,false]
--R
--E 8
Type: List(Boolean)

--S 9 of 13
[upperCase? c for c in chars]
--R
--R
--R (9) [false,true,true,false,false]
--R
--E 9
Type: List(Boolean)

--S 10 of 13
[lowerCase? c for c in chars]
--R
--R
--R (10) [true,false,false,false,false]
--R
--E 10
Type: List(Boolean)

--S 11 of 13
[digit? c for c in chars]
--R
--R
--R (11) [false,false,false,true,false]
--R
--E 11
Type: List(Boolean)

--S 12 of 13
[hexDigit? c for c in chars]
--R
--R
--R (12) [true,true,false,true,false]
--R
--E 12
Type: List(Boolean)

--S 13 of 13
[alphanumeric? c for c in chars]
--R
--R
--R (13) [true,true,true,true,false]
--R
Type: List(Boolean)

```

```
--E 13
)spool
)lisp (bye)
```

— Character.help —

```
=====
Character examples
=====
```

The members of the domain Character are values representing letters, numerals and other text elements.

Characters can be obtained using String notation.

```
chars := [char "a", char "A", char "X", char "8", char "+"]
[a,A,X,8,+]
```

Type: List Character

Certain characters are available by name. This is the blank character.

```
space()
```

Type: Character

This is the quote that is used in strings.

```
quote()
"
```

Type: Character

This is the escape character that allows quotes and other characters within strings.

```
escape()
```

```
-
```

Type: Character

Characters are represented as integers in a machine-dependent way. The integer value can be obtained using the ord operation. It is always true that char(ord c) = c and ord(char i) = i, provided that i is in the range 0..size()\$Character-1.

```
[ord c for c in chars]
[97,65,88,56,43]
```

Type: List Integer

The `lowerCase` operation converts an upper case letter to the corresponding lower case letter. If the argument is not an upper case letter, then it is returned unchanged.

```
[upperCase c for c in chars]
[A,A,X,8,+]
Type: List Character
```

The `upperCase` operation converts lower case letters to upper case.

```
[lowerCase c for c in chars]
[a,a,x,8,+]
Type: List Character
```

A number of tests are available to determine whether characters belong to certain families.

```
[alphabetic? c for c in chars]
[true,true,true,false,false]
Type: List Boolean
```

```
[upperCase? c for c in chars]
[false,true,true,false,false]
Type: List Boolean
```

```
[lowerCase? c for c in chars]
[true,false,false,false,false]
Type: List Boolean
```

```
[digit? c for c in chars]
[false,false,false,true,false]
Type: List Boolean
```

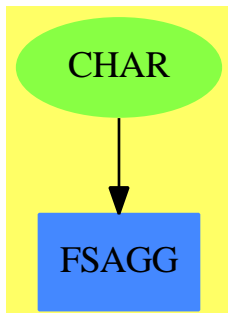
```
[hexDigit? c for c in chars]
[true,true,false,true,false]
Type: List Boolean
```

```
[alphanumeric? c for c in chars]
[true,true,true,true,false]
Type: List Boolean
```

See Also:

- o `)help CharacterClass`
- o `)help String`
- o `)show Character`

Character (CHAR)



See

- ⇒ “CharacterClass” (CCLASS) [4](#) on page [381](#)
- ⇒ “IndexedString” (ISTRING) [10](#) on page [1249](#)
- ⇒ “String” (STRING) [20](#) on page [2606](#)

Exports:

alphanumeric?	alphanumeric?	char	coerce	digit?
escape	hash	hexDigit?	index	latex
lookup	lowerCase	lowerCase?	max	min
ord	quote	random	size	space
upperCase	upperCase?	?~=?	?<?	?<=?
?=?	?>?	?>=?		

— domain CHAR Character —

```

)abbrev domain CHAR Character
++ Author: Stephen M. Watt
++ Date Created: July 1986
++ Date Last Updated: June 20, 1991
++ Description:
++ This domain provides the basic character data type.

```

```

Character: OrderedFinite() with
ord: % -> Integer
  ++ord(c) provides an integral code corresponding to the
  ++character c. It is always true that \spad{char ord c = c}.
  ++
  ++X chars := [char "a", char "A", char "X", char "8", char "+"]
  ++X [ord c for c in chars]
char: Integer -> %
  ++char(i) provides a character corresponding to the integer
  ++code i. It is always true that \spad{ord char i = i}.
  ++
  ++X [char c for c in [97,65,88,56,43]]
char: String -> %

```



```

++char(s) provides a character from a string s of length one.
++
++X [char c for c in ["a","A","X","8","+"]]
space: () -> %
++space() provides the blank character.
++
++X space()
quote: () -> %
++quote() provides the string quote character, \spad{".
++
++X quote()
escape: () -> %
++escape() provides the escape character, \spad{_, which
++is used to allow quotes and other characters within strings.
++
++X escape()
upperCase: % -> %
++upperCase(c) converts a lower case letter to the corresponding
++upper case letter. If c is not a lower case letter, then
++it is returned unchanged.
++
++X chars := [char "a", char "A", char "X", char "8", char "+"]
++X [upperCase c for c in chars]
lowerCase: % -> %
++lowerCase(c) converts an upper case letter to the corresponding
++lower case letter. If c is not an upper case letter, then
++it is returned unchanged.
++
++X chars := [char "a", char "A", char "X", char "8", char "+"]
++X [lowerCase c for c in chars]
digit?: % -> Boolean
++digit?(c) tests if c is a digit character, for example, one of 0..9.
++
++X chars := [char "a", char "A", char "X", char "8", char "+"]
++X [digit? c for c in chars]
hexDigit?: % -> Boolean
++hexDigit?(c) tests if c is a hexadecimal numeral,
++for example, one of 0..9, a..f or A..F.
++
++X chars := [char "a", char "A", char "X", char "8", char "+"]
++X [hexDigit? c for c in chars]
alphabetic?: % -> Boolean
++alphabetic?(c) tests if c is a letter,
++for example, one of a..z or A..Z.
++
++X chars := [char "a", char "A", char "X", char "8", char "+"]
++X [alphabetic? c for c in chars]
upperCase?: % -> Boolean
++upperCase?(c) tests if c is an upper case letter,
++for example, one of A..Z.

```

```

++
++X chars := [char "a", char "A", char "X", char "8", char "+"]
++X [upperCase? c for c in chars]
lowerCase?: % -> Boolean
++lowerCase?(c) tests if c is an lower case letter,
++for example, one of a..z.
++
++X chars := [char "a", char "A", char "X", char "8", char "+"]
++X [lowerCase? c for c in chars]
alphanumeric?: % -> Boolean
++alphanumeric?(c) tests if c is either a letter or number,
++for example, one of 0..9, a..z or A..Z.
++
++X chars := [char "a", char "A", char "X", char "8", char "+"]
++X [alphanumeric? c for c in chars]

== add

Rep := SingleInteger -- 0..255

CC ==> CharacterClass()
import CC

OutChars:PrimitiveArray(OutputForm) :=
  construct [CODE_-CHAR(i)$Lisp for i in 0..255]

minChar := minIndex OutChars

a = b          == a =$Rep b
a < b          == a <$Rep b
size()         == 256
index n        == char((n - 1)::Integer)
lookup c       == (1 + ord c)::PositiveInteger
char(n:Integer) == n::%
ord c          == convert(c)$Rep
random()       == char(random()$Integer rem size())
space          == QENUM(" ", 0$Lisp)$Lisp
quote          == QENUM("_" , 0$Lisp)$Lisp
escape         == QENUM("__" , 0$Lisp)$Lisp
coerce(c:%):OutputForm == OutChars(minChar + ord c)
digit? c       == member?(c pretend Character, digit())
hexDigit? c    == member?(c pretend Character, hexDigit())
upperCase? c   == member?(c pretend Character, upperCase())
lowerCase? c   == member?(c pretend Character, lowerCase())
alphabetic? c  == member?(c pretend Character, alphabetic())
alphanumeric? c == member?(c pretend Character, alphanumeric())

latex c ==
  concat("\mbox{", concat(new(1,c pretend Character)$String, "'')"_
    $String)$String

```

```

char(s:String) ==
  (#s) = 1 => s(minIndex s) pretend %
  error "String is not a single character"

upperCase c ==
  QENUM(PNAME(UPCASE(CODE_-CHAR(ord c)$Lisp)$Lisp)$Lisp,0$Lisp)$Lisp

lowerCase c ==
  QENUM(PNAME(DOWNCASE(CODE_-CHAR(ord c)$Lisp)$Lisp)$Lisp,0$Lisp)$Lisp

```

— CHAR.dotabb —

```

"CHAR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=CHAR",shape=ellipse]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"CHAR" -> "FSAGG"

```

domain CCLASS CharacterClass

— CharacterClass.input —

```

)set break resume
)sys rm -f CharacterClass.output
)spool CharacterClass.output
)set message test on
)set message auto off
)clear all
--S 1 of 16
cl1:=charClass[char "a",char "e",char "i",char "o",char "u",char "y"]
--R
--R
--R (1) "aeiouy"
--R
--R                                          Type: CharacterClass
--E 1

--S 2 of 16
cl2 := charClass "bcdfghjklmnpqrstvwxyz"
--R
--R
--R (2) "bcdfghjklmnpqrstvwxyz"

```

[illegible]

```
--S 9 of 16
member?(char "a", c11)
--R
--R
--R (9) true
--R
--R                                          Type: Boolean
--E 9

--S 10 of 16
member?(char "a", c12)
--R
--R
--R (10) false
--R
--R                                          Type: Boolean
--E 10

--S 11 of 16
intersect(c11, c12)
--R
--R
--R (11) "y"
--R
--R                                          Type: CharacterClass
--E 11

--S 12 of 16
union(c11,c12)
--R
--R
--R (12) "abcdefghijklmnopqrstuvwxyz"
--R
--R                                          Type: CharacterClass
--E 12

--S 13 of 16
difference(c11,c12)
--R
--R
--R (13) "aeiou"
--R
--R                                          Type: CharacterClass
--E 13

--S 14 of 16
intersect(complement(c11),c12)
--R
--R
--R (14) "bcdfghjklmnpqrstvwxyz"
--R
--R                                          Type: CharacterClass
--E 14

--S 15 of 16
```



```

upperCase()
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
      Type: CharacterClass

lowerCase()
"abcdefghijklmnopqrstuvwxyz"
      Type: CharacterClass

alphabetic()
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
      Type: CharacterClass

alphanumeric()
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
      Type: CharacterClass

```

You can quickly test whether a character belongs to a class.

```

member?(char "a", c11)
true
      Type: Boolean

member?(char "a", c12)
false
      Type: Boolean

```

Classes have the usual set operations because the `CharacterClass` domain belongs to the category `FiniteSetAggregate(Character)`.

```

intersect(c11, c12)
"y"
      Type: CharacterClass

union(c11, c12)
"abcdefghijklmnopqrstuvwxyz"
      Type: CharacterClass

difference(c11, c12)
"aeiou"
      Type: CharacterClass

intersect(complement(c11), c12)
"bcd fghjklmnpqrstvwxyz"
      Type: CharacterClass

```

You can modify character classes by adding or removing characters.

```

insert!(char "a", c12)
"abcd fghjklmnpqrstvwxyz"
      Type: CharacterClass

```

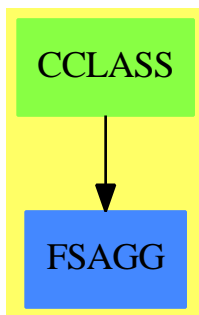
```
remove!(char "b", c12)
"acdfghjklmnpqrstvwxyz"
                                Type: CharacterClass
```

See Also:

- o)help Character
- o)help String
- o)show CharacterClass

—————→

CharacterClass (CCLASS)



See

- ⇒ “Character” (CHAR) [4](#) on page [373](#)
- ⇒ “IndexedString” (ISTRING) [10](#) on page [1249](#)
- ⇒ “String” (STRING) [20](#) on page [2606](#)

Exports:

any?	alphabetic	alphanumeric	bag	brace
brace	cardinality	charClass	coerce	complement
construct	convert	copy	count	count
dictionary	difference	digit	empty	empty?
eq?	eval	eval	eval	eval
every?	extract!	find	hash	hexDigit
index	insert!	inspect	intersect	latex
less?	lookup	lowerCase	map	map!
max	member?	members	min	more?
parts	random	reduce	reduce	reduce
remove	remove	remove!	remove!	removeDuplicates
sample	select	select!	set	size
size?	subset?	symmetricDifference	union	universe
upperCase	#?	?<?	?=?	?~=?

— domain CCLASS CharacterClass —

```

)abbrev domain CCLASS CharacterClass
++ Author: Stephen M. Watt
++ Date Created: July 1986
++ Date Last Updated: June 20, 1991
++ Description:
++ This domain allows classes of characters to be defined and manipulated
++ efficiently.

```

```

CharacterClass: Join(SetCategory, ConvertibleTo String,
  FiniteSetAggregate Character, ConvertibleTo List Character) with
  charClass: String -> %
    ++ charClass(s) creates a character class which contains
    ++ exactly the characters given in the string s.
  charClass: List Character -> %
    ++ charClass(l) creates a character class which contains
    ++ exactly the characters given in the list l.
  digit: constant -> %
    ++ digit() returns the class of all characters
    ++ for which digit? is true.
  hexDigit: constant -> %
    ++ hexDigit() returns the class of all characters for which
    ++ hexDigit? is true.
  upperCase: constant -> %
    ++ upperCase() returns the class of all characters for which
    ++ upperCase? is true.
  lowerCase: constant -> %
    ++ lowerCase() returns the class of all characters for which
    ++ lowerCase? is true.
  alphabetic : constant -> %
    ++ alphabetic() returns the class of all characters for which
    ++ alphabetic? is true.

```

```

alphanumeric: constant -> %
  ++ alphanumeric() returns the class of all characters for which
  ++ alphanumeric? is true.

== add
  Rep := IndexedBits(0)
  N    := size()$Character

  a, b: %

  digit()      == charClass "0123456789"
  hexDigit()   == charClass "0123456789abcdefABCDEF"
  upperCase()  == charClass "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
  lowerCase()  == charClass "abcdefghijklmnopqrstuvwxyz"
  alphabetic() == union(upperCase(), lowerCase())
  alphanumeric == union(alphabetic(), digit())

  a = b        == a =$Rep b

  member?(c, a) == a(ord c)
  union(a,b)    == Or(a, b)
  intersect (a,b) == And(a, b)
  difference(a,b) == And(a, Not b)
  complement a  == Not a

  convert(cl):String ==
    construct(convert(cl)@List(Character))
  convert(cl:%):List(Character) ==
    [char(i) for i in 0..N-1 | cl.i]

  charClass(s: String) ==
    cl := new(N, false)
    for i in minIndex(s)..maxIndex(s) repeat cl(ord s.i) := true
    cl

  charClass(l: List Character) ==
    cl := new(N, false)
    for c in l repeat cl(ord c) := true
    cl

  coerce(cl):OutputForm == (convert(cl)@String)::OutputForm

-- Stuff to make a legal SetAggregate view
# a == (n := 0; for i in 0..N-1 | a.i repeat n := n+1; n)
empty():% == charClass []
brace():% == charClass []

insert_!(c, a) == (a(ord c) := true; a)
remove_!(c, a) == (a(ord c) := false; a)

```

```

inspect(a) ==
  for i in 0..N-1 | a.i repeat
    return char i
  error "Cannot take a character from an empty class."
extract_!(a) ==
  for i in 0..N-1 | a.i repeat
    a.i := false
    return char i
  error "Cannot take a character from an empty class."

map(f, a) ==
  b := new(N, false)
  for i in 0..N-1 | a.i repeat b(ord f char i) := true
  b

temp: % := new(N, false)$Rep
map_!(f, a) ==
  fill_!(temp, false)
  for i in 0..N-1 | a.i repeat temp(ord f char i) := true
  copyInto_!(a, temp, 0)

parts a ==
  [char i for i in 0..N-1 | a.i]

```

— CCLASS.dotabb —

```

"CCLASS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=CCLASS"]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"CCLASS" -> "FSAGG"

```

domain CLIF CliffordAlgebra[?, ?]

Vector (linear) spaces

This information is originally from Paul Leopardi's presentation on the *Introduction to Clifford Algebras* and is included here as an outline with his permission. Further details are based on the book by Doran and Lasenby called *Geometric Algebra for Physicists*.

Consider the various kinds of products that can occur between vectors. There are scalar and vector products from 3D geometry. There are the complex and quaterion products. There is also the *outer* or *exterior* product.

Vector addition commutes:

$$a + b = b + a$$

Vector addition is associative:

$$a + (b + c) = (a + b) + c$$

The identity vector exists:

$$a + 0 = a$$

Every vector has an inverse:

$$a + (-a) = 0$$

If we consider vectors to be directed line segments, thus establishing a geometric meaning for a vector, then each of these properties has a geometric meaning.

A multiplication operator exists between scalars and vectors with the properties:

$$\lambda(a + b) = \lambda a + \lambda b$$

$$(\lambda + \mu)a = \lambda a + \mu a$$

$$(\lambda\mu)a = \lambda(\mu a)$$

If $1\lambda = \lambda$ for all scalars λ then $1a = a$ for all vectors a

These properties completely define a vector (linear) space. The $+$ operation for scalar arithmetic is not the same as the $+$ operation for vectors.

Definition: Isomorphic The vector space A is isomorphic to the vector space B if there exists a one-to-one correspondence between their elements which preserves sums and there is a one-to-one correspondence between the scalars which preserves sums and products.

Definition: Subspace Vector space B is a subspace of vector space A if all of the elements of B are contained in A and they share the same scalars.

Definition: Linear Combination Given vectors a_1, \dots, a_n the vector b is a linear combination of the vectors if we can find scalars λ_i such that

$$b = \lambda_1 a_1 + \dots + \lambda_n a_n = \sum_{k=1}^n \lambda_k a_k$$

Definition: Linearly Independent If there exists scalars λ_i such that

$$\lambda_1 a_1 + \dots + \lambda_n a_n = 0$$

and at least one of the λ_i is not zero then the vectors a_1, \dots, a_n are linearly dependent. If no such scalars exist then the vectors are linearly independent.

Definition: Span If every vector can be written as a linear combination of a fixed set of vectors a_1, \dots, a_n then this set of vectors is said to span the vector space.

Definition: Basis If a set of vectors a_1, \dots, a_n is linearly independent and spans a vector space A then the vectors form a basis for A .

Definition: Dimension The dimension of a vector space is the number of basis elements, which is unique since all bases of a vector space have the same number of elements.

Quadratic Forms[?]

For vector space \mathbb{V} over field \mathbb{F} , characteristic $\neq 2$:

Map $f : \mathbb{V} \rightarrow \mathbb{F}$, with

$$f(\lambda x) = \lambda^2 f(x), \forall \lambda \in \mathbb{F}, x \in \mathbb{V}$$

$f(x) = b(x, x)$, where

$$b : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{F}, \text{ given by}$$

$$b(x, y) := \frac{1}{2}(f(x+y) - f(x) - f(y))$$

is a symmetric bilinear form

Quadratic spaces, Clifford Maps[?, ?]

A quadratic space is the pair (\mathbb{V}, f) , where f is a quadratic form on \mathbb{V}

A Clifford map is a vector space homomorphism

$$\rho : \mathbb{V} \rightarrow \mathbb{A}$$

where \mathbb{A} is an associated algebra, and

$$(\rho v)^2 = f(v), \quad \forall v \in \mathbb{V}$$

Universal Clifford algebras[?]

The *universal Clifford algebra* $Cl(f)$ for the quadratic space (\mathbb{V}, f) is the algebra generated by the image of the Clifford map ρ_f such that $Cl(f)$ is the universal initial object such that \forall suitable algebra \mathbb{A} with Clifford map $\phi_{\mathbb{A}} \exists$ a homomorphism

$$P_{\mathbb{A}} : Cl(f) \rightarrow \mathbb{A}$$

$$\rho_{\mathbb{A}} = P_{\mathbb{A}} \circ \rho_f$$

Real Clifford algebras $\mathbb{R}_{p,q}$ [?]

The real quadratic space $\mathbb{R}^{p,q}$ is \mathbb{R}^{p+q} with

$$\phi(x) := - \sum_{k=-q}^{-1} x_k^2 + \sum_{k=1}^p x_k^2$$

For each $p, q \in \mathbb{N}$, the real universal Clifford algebra for $\mathbb{R}^{p,q}$ is called $\mathbb{R}_{p,q}$

$\mathbb{R}_{p,q}$ is isomorphic to some matrix algebra over one of: $\mathbb{R}, \mathbb{R} \oplus \mathbb{R}, \mathbb{C}, \mathbb{H}, \mathbb{H} \oplus \mathbb{H}$

For example, $\mathbb{R}_{1,1} \cong \mathbb{R}(2)$

Notation for integer sets

For $S \subseteq \mathbb{Z}$, define

$$\sum_{k \in S} f_k := \sum_{k=\min S, k \in S}^{\max S} f_k$$

$$\prod_{k \in S} f_k := \prod_{k=\min S, k \in S}^{\max S} f_k$$

$$\mathbb{P}(S) := \text{the power set of } S$$

For $m \leq n \in \mathbb{Z}$, define

$$\zeta(m, n) := \{m, m+1, \dots, n-1, n\} \setminus \{0\}$$

Frames for Clifford algebras[?, ?, ?]

A *frame* is an ordered basis $(\gamma_{-q}, \dots, \gamma_p)$ for $\mathbb{R}^{p,q}$ which puts a quadratic form into the canonical form ϕ

For $p, q \in \mathbb{N}$, embed the frame for $\mathbb{R}^{p,q}$ into $\mathbb{R}_{p,q}$ via the maps

$$\gamma : \zeta(-q, p) \rightarrow \mathbb{R}^{p,q}$$

$$\rho : \mathbb{R}^{p,q} \rightarrow \mathbb{R}_{p,q}$$

$$(\rho\gamma k)^2 = \phi\gamma k = \text{sgn } k$$

Real frame groups[?, ?]

For $p, q \in \mathbb{N}$, define the real *frame group* $\mathbb{G}_{p,q}$ via the map

$$g : \zeta(-q, p) \rightarrow \mathbb{G}_{p,q}$$

with generators and relations

$$\langle \mu, g_k | \mu g_k = g_k \mu, \quad \mu^2 = 1,$$

$$(g_k)^2 = \begin{cases} \mu, & \text{if } k < 0 \\ 1 & \text{if } k > 0 \end{cases}$$

$$g_k g_m = \mu g_m g_k \quad \forall k \neq m \rangle$$

Canonical products[?, ?, ?]

The real frame group $\mathbb{G}_{p,q}$ has order 2^{p+q+1}

Each member w can be expressed as the canonically ordered product

$$\begin{aligned} w &= \mu^a \prod_{k \in T} g_k \\ &= \mu^a \prod_{k=-q, k \neq 0}^p g_k^{b_k} \end{aligned}$$

where $T \subseteq \zeta(-q, p)$, $a, b_k \in \{0, 1\}$

Clifford algebra of frame group[?, ?, ?, ?]

For $p, q \in \mathbb{N}$ embed $\mathbb{G}_{p,q}$ into $\mathbb{R}_{p,q}$ via the map

$$\begin{aligned} \alpha \mathbb{G}_{p,q} &\rightarrow \mathbb{R}_{p,q} \\ \alpha 1 &:= 1, \quad \alpha \mu := -1 \\ \alpha g_k &:= \rho \gamma_k, \quad \alpha(gh) := (\alpha g)(\alpha h) \end{aligned}$$

Define *basis elements* via the map

$$e : \mathbb{P}\zeta(-q, p) \rightarrow \mathbb{R}_{p,q}, \quad e_T := \alpha \prod_{k \in T} g_k$$

Each $a \in \mathbb{R}_{p,q}$ can be expressed as

$$a = \sum_{T \subseteq \zeta(-q, p)} a_T e_T$$

Neutral matrix representations[?, ?, ?]

The *representation map* P_m and *representation matrix* R_m make the following diagram

$$\begin{array}{ccc} \mathbb{R}_{m,m} & \xrightarrow{\text{coord}} & \mathbb{R}^{4^m} \\ \downarrow P_m & & \downarrow R_m \\ \text{commute:} & & \\ \mathbb{V} & & \mathbb{V} \\ \downarrow & & \downarrow \\ \mathbb{R}(2^m) & \xrightarrow{\text{reshape}} & \mathbb{R}^{4^m} \\ \text{--- CliffordAlgebra.input ---} & & \end{array}$$

```

)set break resume
)sys rm -f CliffordAlgebra.output
)spool CliffordAlgebra.output
)set message test on
)set message auto off
)clear all
--S 1 of 36
K := Fraction Polynomial Integer
--R
--R
--R (1) Fraction(Polynomial(Integer))
--R
--R                                          Type: Domain
--E 1

--S 2 of 36
m := matrix [ [-1] ]
--R
--R
--R (2) [- 1]
--R
--R                                          Type: Matrix(Integer)
--E 2

--S 3 of 36
C := CliffordAlgebra(1, K, quadraticForm m)
--R
--R
--R (3) CliffordAlgebra(1,Fraction(Polynomial(Integer)))[[-1]]
--R
--R                                          Type: Domain
--E 3

--S 4 of 36
i: C := e(1)
--R
--R
--R (4) e
--R      1
--R
--R                                          Type: CliffordAlgebra(1,Fraction(Polynomial(Integer)))[[-1]]
--E 4

--S 5 of 36
x := a + b * i
--R
--R
--R (5) a + b e
--R      1
--R
--R                                          Type: CliffordAlgebra(1,Fraction(Polynomial(Integer)))[[-1]]
--E 5

--S 6 of 36
y := c + d * i

```



```

--R
--R
--R (6)  $c + d e$ 
--R 
$$1$$

--R Type: CliffordAlgebra(1,Fraction(Polynomial(Integer)))[[-1]]
--E 6

--S 7 of 36
x * y
--R
--R
--R (7)  $- b d + a c + (a d + b c) e$ 
--R 
$$1$$

--R Type: CliffordAlgebra(1,Fraction(Polynomial(Integer)))[[-1]]
--E 7
)clear all

--S 8 of 36
K := Fraction Polynomial Integer
--R
--R
--R (1) Fraction(Polynomial(Integer))
--R Type: Domain
--E 8

--S 9 of 36
m := matrix [ [-1,0],[0,-1] ]
--R
--R
--R 
$$\begin{array}{cc} +- & 1 & 0 & + \\ (2) & | & & | \\ & + & 0 & - & 1+ \end{array}$$

--R Type: Matrix(Integer)
--E 9

--S 10 of 36
H := CliffordAlgebra(2, K, quadraticForm m)
--R
--R
--R (3) CliffordAlgebra(2,Fraction(Polynomial(Integer)))[[-1,0],[0,-1]]
--R Type: Domain
--E 10

--S 11 of 36
i: H := e(1)
--R
--R
--R (4)  $e$ 
--R 
$$1$$

--R Type: CliffordAlgebra(2,Fraction(Polynomial(Integer)))[[-1,0],[0,-1]]

```

--E 11

--S 12 of 36

j: H := e(2)

--R

--R

--R (5) e

--R 2

--R Type: CliffordAlgebra(2,Fraction(Polynomial(Integer)))[[-1,0],[0,-1]]

--E 12

--S 13 of 36

k: H := i * j

--R

--R

--R (6) e e

--R 1 2

--R Type: CliffordAlgebra(2,Fraction(Polynomial(Integer)))[[-1,0],[0,-1]]

--E 13

--S 14 of 36

x := a + b * i + c * j + d * k

--R

--R

--R (7) a + b e + c e + d e e

--R 1 2 1 2

--R Type: CliffordAlgebra(2,Fraction(Polynomial(Integer)))[[-1,0],[0,-1]]

--E 14

--S 15 of 36

y := e + f * i + g * j + h * k

--R

--R

--R (8) e + f e + g e + h e e

--R 1 2 1 2

--R Type: CliffordAlgebra(2,Fraction(Polynomial(Integer)))[[-1,0],[0,-1]]

--E 15

--S 16 of 36

x + y

--R

--R

--R (9) e + a + (f + b)e + (g + c)e + (h + d)e e

--R 1 2 1 2

--R Type: CliffordAlgebra(2,Fraction(Polynomial(Integer)))[[-1,0],[0,-1]]

--E 16

--S 17 of 36

x * y

--R

```

--R
--R (10)
--R      - d h - c g - b f + a e + (c h - d g + a f + b e)e
--R                                         1
--R +
--R      (- b h + a g + d f + c e)e + (a h + b g - c f + d e)e e
--R          2                      1 2
--R      Type: CliffordAlgebra(2,Fraction(Polynomial(Integer))],[[-1,0],[0,-1]])
--E 17

--S 18 of 36
y * x
--R
--R
--R (11)
--R      - d h - c g - b f + a e + (- c h + d g + a f + b e)e
--R                                         1
--R +
--R      (b h + a g - d f + c e)e + (a h - b g + c f + d e)e e
--R          2                      1 2
--R      Type: CliffordAlgebra(2,Fraction(Polynomial(Integer))],[[-1,0],[0,-1]])
--E 18
)clear all

--S 19 of 36
K := Fraction Polynomial Integer
--R
--R
--R (1) Fraction(Polynomial(Integer))
--R
--R                                         Type: Domain
--E 19

--S 20 of 36
Ext := CliffordAlgebra(3, K, quadraticForm 0)
--R
--R
--R (2)
--R      CliffordAlgebra(3,Fraction(Polynomial(Integer))],[[0,0,0],[0,0,0],[0,0,0]])
--R
--R                                         Type: Domain
--E 20

--S 21 of 36
i: Ext := e(1)
--R
--R
--R (3) e
--R      1
--RType: CliffordAlgebra(3,Fraction(Polynomial(Integer))],[[0,0,0],[0,0,0],[0,0,0]])
--E 21

```

```

--S 22 of 36
j: Ext := e(2)
--R
--R
--R (4) e
--R      2
--RType: CliffordAlgebra(3,Fraction(Polynomial(Integer)),[[0,0,0],[0,0,0],[0,0,0]])
--E 22

```

```

--S 23 of 36
k: Ext := e(3)
--R
--R
--R (5) e
--R      3
--RType: CliffordAlgebra(3,Fraction(Polynomial(Integer)),[[0,0,0],[0,0,0],[0,0,0]])
--E 23

```

```

--S 24 of 36
x := x1*i + x2*j + x3*k
--R
--R
--R (6) x1 e + x2 e + x3 e
--R      1      2      3
--RType: CliffordAlgebra(3,Fraction(Polynomial(Integer)),[[0,0,0],[0,0,0],[0,0,0]])
--E 24

```

```

--S 25 of 36
y := y1*i + y2*j + y3*k
--R
--R
--R (7) y1 e + y2 e + y3 e
--R      1      2      3
--RType: CliffordAlgebra(3,Fraction(Polynomial(Integer)),[[0,0,0],[0,0,0],[0,0,0]])
--E 25

```

```

--S 26 of 36
x + y
--R
--R
--R (8) (y1 + x1)e + (y2 + x2)e + (y3 + x3)e
--R      1      2      3
--RType: CliffordAlgebra(3,Fraction(Polynomial(Integer)),[[0,0,0],[0,0,0],[0,0,0]])
--E 26

```

```

--S 27 of 36
x * y + y * x
--R
--R
--R (9) 0

```

```

--RType: CliffordAlgebra(3,Fraction(Polynomial(Integer))],[[0,0,0],[0,0,0],[0,0,0]])
--E 27

--S 28 of 36
dual2 a == coefficient(a,[2,3]) * i + coefficient(a,[3,1]) * j + coefficient(a,[1,2]) * k
--R
--R
--R                                          Type: Void
--E 28

--S 29 of 36
dual2(x*y)
--R
--R   Compiling function dual2 with type CliffordAlgebra(3,Fraction(
--R   Polynomial(Integer))],[[0,0,0],[0,0,0],[0,0,0]]) ->
--R   CliffordAlgebra(3,Fraction(Polynomial(Integer))],[[0,0,0],[0,0,0],
--R   [0,0,0]])
--R
--R   (11)  (x2 y3 - x3 y2)e1 + (- x1 y3 + x3 y1)e2 + (x1 y2 - x2 y1)e3
--R
--RType: CliffordAlgebra(3,Fraction(Polynomial(Integer))],[[0,0,0],[0,0,0],[0,0,0]])
--E 29
)clear all

--S 30 of 36
K := Fraction Integer
--R
--R
--R   (1)  Fraction(Integer)
--R
--R                                          Type: Domain
--E 30

--S 31 of 36
g := matrix [ [1,0,0,0], [0,-1,0,0], [0,0,-1,0], [0,0,0,-1] ]
--R
--R
--R
--R      +1   0   0   0 +
--R      |
--R      |0  - 1   0   0 |
--R   (2) |
--R      |0   0  - 1   0 |
--R      |
--R      +0   0   0  - 1+
--R
--R                                          Type: Matrix(Integer)
--E 31

--S 32 of 36
D := CliffordAlgebra(4,K, quadraticForm g)
--R
--R
--R   (3)

```

```
--R CliffordAlgebra(4,Fraction(Integer),[[1,0,0,0],[0,-1,0,0],[0,0,-1,0],[0,0,0,-
--R 1]])
--R
--R                                         Type: Domain
--E 32
```

```
--S 33 of 36
gam := [e(i)$D for i in 1..4]
--R
--R
--R      (4)  [e ,e ,e ,e ]
--R           1 2 3 4
--RType: List(CliffordAlgebra(4,Fraction(Integer),[[1,0,0,0],[0,-1,0,0],[0,0,-1,0],[0,0,0,-1]]))
--E 33
```

```
--S 34 of 36
m := 1; n:= 2; r := 3; s := 4;
--R
--R
--R
--R                                         Type: PositiveInteger
--E 34
```

```
--S 35 of 36
lhs := reduce(+, [reduce(+, [ g(l,t)*gam(l)*gam(m)*gam(n)*gam(r)*gam(s)*gam(t) for l in 1..4]) for t in 1..4]
--R
--R
--R      (6)  - 4e e e e
--R           1 2 3 4
--RType: CliffordAlgebra(4,Fraction(Integer),[[1,0,0,0],[0,-1,0,0],[0,0,-1,0],[0,0,0,-1]])
--E 35
```

```
--S 36 of 36
rhs := 2*(gam s * gam m*gam n*gam r + gam r*gam n*gam m*gam s)
--R
--R
--R      (7)  - 4e e e e
--R           1 2 3 4
--RType: CliffordAlgebra(4,Fraction(Integer),[[1,0,0,0],[0,-1,0,0],[0,0,-1,0],[0,0,0,-1]])
--E 36
)spool
)lisp (bye)
```

— CliffordAlgebra.help —

```
=====
CliffordAlgebra examples
=====
```

`CliffordAlgebra(n,K,Q)` defines a vector space of dimension 2^n over the field K with a given quadratic form Q . If $\{e_1..e_n\}$ is a basis for K^n then

```
{ 1,
  e(i)          1 <= i <= n,
  e(i1)*e(i2)   1 <= i1 < i2 <= n,
  ...,
  e(1)*e(2)*...*e(n) }
```

is a basis for the Clifford algebra. The algebra is defined by the relations

```
e(i)*e(i) = Q(e(i))
e(i)*e(j) = -e(j)*e(i), for i != j
```

Examples of Clifford Algebras are gaussians (complex numbers), quaternions, exterior algebras and spin algebras.

```
=====
The Complex Numbers as a Clifford Algebra
=====
```

This is the field over which we will work, rational functions with integer coefficients.

```
K := Fraction Polynomial Integer
Fraction Polynomial Integer
Type: Domain
```

We use this matrix for the quadratic form.

```
m := matrix [ [-1] ]
[- 1]
Type: Matrix Integer
```

We get complex arithmetic by using this domain.

```
C := CliffordAlgebra(1, K, quadraticForm m)
CliffordAlgebra(1,Fraction Polynomial Integer,MATRIX)
Type: Domain
```

Here is i , the usual square root of -1 .

```
i: C := e(1)
e
1
Type: CliffordAlgebra(1,Fraction Polynomial Integer,MATRIX)
```

Here are some examples of the arithmetic.

```

x := a + b * i
a + b e
1
Type: CliffordAlgebra(1,Fraction Polynomial Integer,MATRIX)

```

```

y := c + d * i
c + d e
1
Type: CliffordAlgebra(1,Fraction Polynomial Integer,MATRIX)

```

```

x * y
- b d + a c + (a d + b c)e
1
Type: CliffordAlgebra(1,Fraction Polynomial Integer,MATRIX)

```

```

=====
The Quaternion Numbers as a Clifford Algebra
=====

```

This is the field over which we will work, rational functions with integer coefficients.

```

K := Fraction Polynomial Integer
Fraction Polynomial Integer
Type: Domain

```

We use this matrix for the quadratic form.

```

m := matrix [ [-1,0],[0,-1] ]
+- 1  0 +
|      |
+ 0  - 1+
Type: Matrix Integer

```

The resulting domain is the quaternions.

```

H := CliffordAlgebra(2, K, quadraticForm m)
CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)
Type: Domain

```

We use Hamilton's notation for i, j, k.

```

i: H := e(1)
e
1
Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

j: H := e(2)
e

```



```

2
Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

k: H := i * j
e e
1 2
Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

x := a + b * i + c * j + d * k
a + b e + c e + d e e
1 2 1 2
Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

y := e + f * i + g * j + h * k
e + f e + g e + h e e
1 2 1 2
Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

x + y
e + a + (f + b)e + (g + c)e + (h + d)e e
1 2 1 2
Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

x * y
- d h - c g - b f + a e + (c h - d g + a f + b e)e
1
+
(- b h + a g + d f + c e)e + (a h + b g - c f + d e)e e
2 1 2
Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

y * x
- d h - c g - b f + a e + (- c h + d g + a f + b e)e
1
+
(b h + a g - d f + c e)e + (a h - b g + c f + d e)e e
2 1 2
Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

```

```

=====
The Exterior Algebra on a Three Space
=====

```

This is the field over which we will work, rational functions with integer coefficients.

```

K := Fraction Polynomial Integer
Fraction Polynomial Integer
Type: Domain

```

If we chose the three by three zero quadratic form, we obtain the exterior algebra on $e(1), e(2), e(3)$.

```
Ext := CliffordAlgebra(3, K, quadraticForm 0)
      CliffordAlgebra(3, Fraction Polynomial Integer, MATRIX)
      Type: Domain
```

This is a three dimensional vector algebra. We define i, j, k as the unit vectors.

```
i: Ext := e(1)
  e
  1
      Type: CliffordAlgebra(3, Fraction Polynomial Integer, MATRIX)

j: Ext := e(2)
  e
  2
      Type: CliffordAlgebra(3, Fraction Polynomial Integer, MATRIX)

k: Ext := e(3)
  e
  3
      Type: CliffordAlgebra(3, Fraction Polynomial Integer, MATRIX)
```

Now it is possible to do arithmetic.

```
x := x1*i + x2*j + x3*k
    x1 e  + x2 e  + x3 e
      1    2    3
      Type: CliffordAlgebra(3, Fraction Polynomial Integer, MATRIX)

y := y1*i + y2*j + y3*k
    y1 e  + y2 e  + y3 e
      1    2    3
      Type: CliffordAlgebra(3, Fraction Polynomial Integer, MATRIX)

x + y
    (y1 + x1)e  + (y2 + x2)e  + (y3 + x3)e
      1          2          3
      Type: CliffordAlgebra(3, Fraction Polynomial Integer, MATRIX)

x * y + y * x
  0
      Type: CliffordAlgebra(3, Fraction Polynomial Integer, MATRIX)
```

On an n space, a grade p form has a dual $n-p$ form. In particular, in three space the dual of a grade two element identifies

$e1*e2 \rightarrow e3, e2*e3 \rightarrow e1, e3*e1 \rightarrow e2$.

```
dual2 a == coefficient(a,[2,3]) * i + coefficient(a,[3,1]) * j + coefficient(a,[1,2]) * k
Type: Void
```

The vector cross product is then given by this.

```
dual2(x*y)
(x2 y3 - x3 y2)e1 + (- x1 y3 + x3 y1)e2 + (x1 y2 - x2 y1)e3
Type: CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)
```

```
=====
The Dirac Spin Algebra
=====
```

In this section we will work over the field of rational numbers.

```
K := Fraction Integer
Fraction Integer
Type: Domain
```

We define the quadratic form to be the Minkowski space-time metric.

```
g := matrix [ [1,0,0,0], [0,-1,0,0], [0,0,-1,0], [0,0,0,-1] ]
+1  0  0  0 +
|
|0  -1  0  0 |
|
|0  0  -1  0 |
|
+0  0  0  -1+
Type: Matrix Integer
```

We obtain the Dirac spin algebra used in Relativistic Quantum Field Theory.

```
D := CliffordAlgebra(4,K, quadraticForm g)
CliffordAlgebra(4,Fraction Integer,MATRIX)
Type: Domain
```

The usual notation for the basis is gamma with a superscript. For Axiom input we will use gam(i):

```
gam := [e(i)$D for i in 1..4]
[e1, e2, e3, e4]
Type: List CliffordAlgebra(4,Fraction Integer,MATRIX)
```

There are various contraction identities of the form

```
g(l,t)*gam(l)*gam(m)*gam(n)*gam(r)*gam(s)*gam(t) =
```

$$2*(\text{gam}(s)\text{gam}(m)\text{gam}(n)\text{gam}(r) + \text{gam}(r)\text{gam}(n)\text{gam}(m)\text{gam}(s))$$

where a sum over l and t is implied.

Verify this identity for particular values of m, n, r, s .

```
m := 1; n:= 2; r := 3; s := 4;
```

```
      Type: PositiveInteger
```

```
lhs := reduce(+, [reduce(+, [ g(l,t)*gam(l)*gam(m)*gam(n)*gam(r)*gam(s)*gam(t) for l in 1..4]) for t in 1
- 4e e e e
  1 2 3 4
```

```
      Type: CliffordAlgebra(4,Fraction Integer,MATRIX)
```

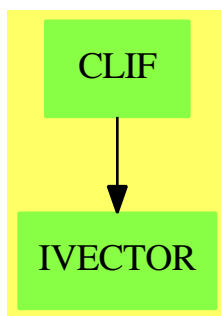
```
rhs := 2*(gam s * gam m*gam n*gam r + gam r*gam n*gam m*gam s)
- 4e e e e
  1 2 3 4
```

```
      Type: CliffordAlgebra(4,Fraction Integer,MATRIX)
```

See Also:

- o)help Complex
- o)help Quaternion
- o)show CliffordAlgebra
- o \$AXIOM/doc/src/algebra/clifford.spad

CliffordAlgebra (CLIF)



See

⇒ “QuadraticForm” (QFORM) [18](#) on page [2154](#)

Exports:

0	1	characteristic	coefficient	coerce
dimension	e	hash	latex	monomial
one?	recip	sample	subtractIfCan	zero?
?^=?	?*?	?**?	?^?	?+?
?-?	/?	?/?	?=?	

— domain CLIF CliffordAlgebra —

```

)abbrev domain CLIF CliffordAlgebra
++ Author: Stephen M. Watt
++ Date Created: August 1988
++ Date Last Updated: May 17, 1991
++ Description:
++ CliffordAlgebra(n, K, Q) defines a vector space of dimension \spad{2**n}
++ over K, given a quadratic form Q on \spad{K**n}.
++
++ If \spad{e[i]}, \spad{1<=i<=n} is a basis for \spad{K**n} then
++ 1, \spad{e[i]} (\spad{1<=i<=n}), \spad{e[i1]*e[i2]}
++ (\spad{1<=i1<i2<=n}), ..., \spad{e[1]*e[2]*...*e[n]}
++ is a basis for the Clifford Algebra.
++
++ The algebra is defined by the relations\br
++ \tab{5}\spad{e[i]*e[j] = -e[j]*e[i]} (\spad{i ~~= j}),\br
++ \tab{5}\spad{e[i]*e[i] = Q(e[i])}
++
++ Examples of Clifford Algebras are: gaussians, quaternions, exterior
++ algebras and spin algebras.

CliffordAlgebra(n, K, Q): T == Impl where
  n: PositiveInteger
  K: Field
  Q: QuadraticForm(n, K)

  PI ==> PositiveInteger
  NNI==> NonNegativeInteger

  T ==> Join(Ring, Algebra(K), VectorSpace(K)) with
    e: PI -> %
      ++ e(n) produces the appropriate unit element.
    monomial: (K, List PI) -> %
      ++ monomial(c, [i1,i2,...,iN]) produces the value given by
      ++ \spad{c*e(i1)*e(i2)*...*e(iN)}.
    coefficient: (% , List PI) -> K
      ++ coefficient(x, [i1,i2,...,iN]) extracts the coefficient of
      ++ \spad{e(i1)*e(i2)*...*e(iN)} in x.
    recip: % -> Union(% , "failed")
      ++ recip(x) computes the multiplicative inverse of x or "failed"
      ++ if x is not invertible.

```

```

Impl ==> add
  Qeelist := [Q unitVector(i::PositiveInteger) for i in 1..n]
  dim      := 2**n

  Rep      := PrimitiveArray K

  New      ==> new(dim, 0$K)$Rep

  x, y, z: %
  c: K
  m: Integer

  characteristic() == characteristic()$K
  dimension()      == dim::CardinalNumber

  x = y ==
    for i in 0..dim-1 repeat
      if x.i ^= y.i then return false
    true

  x + y == (z := New; for i in 0..dim-1 repeat z.i := x.i + y.i; z)
  x - y == (z := New; for i in 0..dim-1 repeat z.i := x.i - y.i; z)
  - x    == (z := New; for i in 0..dim-1 repeat z.i := - x.i; z)
  m * x == (z := New; for i in 0..dim-1 repeat z.i := m*x.i; z)
  c * x == (z := New; for i in 0..dim-1 repeat z.i := c*x.i; z)

  0      == New
  1      == (z := New; z.0 := 1; z)
  coerce(m): % == (z := New; z.0 := m::K; z)
  coerce(c): % == (z := New; z.0 := c; z)

  e b ==
    b::NNI > n => error "No such basis element"
    iz := 2**((b-1)::NNI)
    z := New; z.iz := 1; z

  -- The ei*ej products could instead be precomputed in
  -- a (2**n)**2 multiplication table.
  addMonomProd(c1: K, b1: NNI, c2: K, b2: NNI, z: %): % ==
    c := c1 * c2
    bz := b2
    for i in 0..n-1 | bit?(b1,i) repeat
      -- Apply rule ei*ej = -ej*ei for i^=j
      k := 0
      for j in i+1..n-1 | bit?(b1, j) repeat k := k+1
      for j in 0..i-1 | bit?(bz, j) repeat k := k+1
      if odd? k then c := -c
      -- Apply rule ei**2 = Q(ei)
      if bit?(bz,i) then
        c := c * Qeelist.(i+1)

```

```

        bz:= (bz - 2**i)::NNI
      else
        bz:= bz + 2**i
      z.bz := z.bz + c
    z

x * y ==
  z := New
  for ix in 0..dim-1 repeat
    if x.ix ^= 0 then for iy in 0..dim-1 repeat
      if y.iy ^= 0 then addMonomProd(x.ix,ix,y.iy,iy,z)
    z

canonMonom(c: K, lb: List PI): Record(coef: K, basel: NNI) ==
  -- 0. Check input
  for b in lb repeat b > n => error "No such basis element"

  -- 1. Apply identity  $e_i e_j = -e_j e_i$ ,  $i \wedge j$ .
  -- The Rep assumes n is small so bubble sort is ok.
  -- Using bubble sort keeps the exchange info obvious.
  wasordered := false
  exchanges := 0
  while not wasordered repeat
    wasordered := true
    for i in 1..#lb-1 repeat
      if lb.i > lb.(i+1) then
        t := lb.i; lb.i := lb.(i+1); lb.(i+1) := t
        exchanges := exchanges + 1
        wasordered := false
  if odd? exchanges then c := -c

  -- 2. Prepare the basis element
  -- Apply identity  $e_i e_i = Q(e_i)$ .
  bz := 0
  for b in lb repeat
    bn := (b-1)::NNI
    if bit?(bz, bn) then
      c := c * Qeelist bn
      bz:= ( bz - 2**bn )::NNI
    else
      bz:= bz + 2**bn
  [c, bz::NNI]

monomial(c, lb) ==
  r := canonMonom(c, lb)
  z := New
  z r.basel := r.coef
  z

coefficient(z, lb) ==
  r := canonMonom(1, lb)

```

```

    r.coef = 0 => error "Cannot take coef of 0"
    z r.basel/r.coef

Ex ==> OutputForm

coerceMonom(c: K, b: NNI): Ex ==
  b = 0 => c::Ex
  ml := [sub("e"::Ex, i::Ex) for i in 1..n | bit?(b,i-1)]
  be := reduce("*", ml)
  c = 1 => be
  c::Ex * be

coerce(x): Ex ==
  tl := [coerceMonom(x.i,i) for i in 0..dim-1 | x.i!=0]
  null tl => "0"::Ex
  reduce("+", tl)

localPowerSets(j:NNI): List(List(PI)) ==
  l: List List PI := list []
  j = 0 => l
  Sm := localPowerSets((j-1)::NNI)
  Sn: List List PI := []
  for x in Sm repeat Sn := cons(cons(j pretend PI, x),Sn)
  append(Sn, Sm)

powerSets(j:NNI):List List PI == map(reverse, localPowerSets j)

Pn:List List PI := powerSets(n)

recip(x: %): Union(%, "failed") ==
  one:% := 1
  -- tmp:c := x*yC - 1$C
  rhsEqs : List K := []
  lhsEqs: List List K := []
  lhsEqi: List K
  for pi in Pn repeat
    rhsEqs := cons(coefficient(one, pi), rhsEqs)

    lhsEqi := []
    for pj in Pn repeat
      lhsEqi := cons(coefficient(x*monomial(1,pj),pi),lhsEqi)
    lhsEqs := cons(reverse(lhsEqi),lhsEqs)
  ans := particularSolution(matrix(lhsEqs),vector(rhsEqs))_
    )$LinearSystemMatrixPackage(K, Vector K, Vector K, Matrix K)
  ans case "failed" => "failed"
  ansP := parts(ans)
  ansC:% := 0
  for pj in Pn repeat
    cj:= first ansP
    ansP := rest ansP

```



```

      ansC := ansC + cj*monomial(1,pj)
    ansC

```

— CLIF.dotabb —

```

"CLIF" [color="#88FF44",href="bookvol10.3.pdf#nameddest=CLIF"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"CLIF" -> "IVECTOR"

```

domain COLOR Color

— Color.input —

```

)set break resume
)sys rm -f Color.output
)spool Color.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Color
--R Color is a domain constructor
--R Abbreviation for Color is COLOR
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for COLOR
--R
--R----- Operations -----
--R ???: (DoubleFloat,%) -> %           ??? : (PositiveInteger,%) -> %
--R ?+?: (%,%) -> %                     ?=? : (%,%) -> Boolean
--R blue : () -> %                      coerce : % -> OutputForm
--R color : Integer -> %                 green : () -> %
--R hash : % -> SingleInteger            hue : % -> Integer
--R latex : % -> String                  red : () -> %
--R yellow : () -> %                     ?~=? : (%,%) -> Boolean
--R numberOfHues : () -> PositiveInteger
--R
--E 1

)spool
)lisp (bye)

```

— Color.help —

=====

Color examples

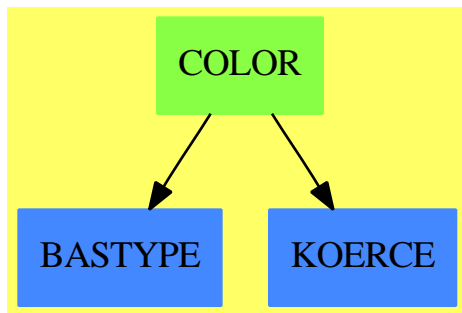
=====

Color() specifies a domain of 27 colors provided in the Axiom system
(the colors mix additively).

See Also:

o)show Color

Color (COLOR)



See

⇒ “Palette” (PALETTE) [17](#) on page [1896](#)

Exports:

blue	coerce	color	green	hash
hue	latex	numberOfHues	red	yellow
?~=?	?*?	?+?	?=?	

— domain COLOR Color —

```

)abbrev domain COLOR Color
++ Author: Jim Wen
++ Date Created: 10 May 1989
++ Date Last Updated: 19 Mar 1991 by Jon Steinbach
++ Description:
++ Color() specifies a domain of 27 colors provided in the
++ Axiom system (the colors mix additively).
  
```

```

Color(): Exports == Implementation where
  I      ==> Integer
  PI     ==> PositiveInteger
  SF     ==> DoubleFloat

Exports ==> AbelianSemiGroup with
  "*"    : (PI, %) -> %
    ++ s * c, returns the color c, whose weighted shade has been scaled by s.
  "*"    : (SF, %) -> %
    ++ s * c, returns the color c, whose weighted shade has been scaled by s.
  "+"    : (%, %) -> %
    ++ c1 + c2 additively mixes the two colors c1 and c2.
  red    : ()      -> %
    ++ red() returns the position of the red hue from total hues.
  yellow : ()      -> %
    ++ yellow() returns the position of the yellow hue from total hues.
  green  : ()      -> %
    ++ green() returns the position of the green hue from total hues.
  blue   : ()      -> %
    ++ blue() returns the position of the blue hue from total hues.
  hue    : %        -> I
    ++ hue(c) returns the hue index of the indicated color c.
  numberOfHues : ()  -> PI
    ++ numberOfHues() returns the number of total hues, set in totalHues.
  color    : Integer -> %
    ++ color(i) returns a color of the indicated hue i.

Implementation ==> add
  totalHues ==> 27 --see (header.h file) for the current number

Rep := Record(hue:I, weight:SF)

f:SF * c:% ==
  -- s * c returns the color c, whose weighted shade has been scaled by s
  zero? f => c
  -- 0 is the identitly function...or maybe an error is better?
  [c.hue, f * c.weight]

x + y ==
  x.hue = y.hue => [x.hue, x.weight + y.weight]
  if y.weight > x.weight then -- let x be color with bigger weight
    c := x
    x := y
    y := c
  diff := x.hue - y.hue
  if (xHueSmaller:=(diff < 0)) then diff := -diff
  if (moreThanHalf:=(diff > totalHues quo 2)) then diff := totalHues-diff
  offset : I := wholePart(round (diff::SF/(2::SF)**(x.weight/y.weight)) )

```

```

    if (xHueSmaller and ^moreThanHalf) or (^xHueSmaller and moreThanHalf) then
      ans := x.hue + offset
    else
      ans := x.hue - offset
    if (ans < 0) then ans := totalHues + ans
    else if (ans > totalHues) then ans := ans - totalHues
    [ans,1]

x = y      == (x.hue = y.hue) and (x.weight = y.weight)
red()      == [1,1]
yellow()   == [11::I,1]
green()    == [14::I,1]
blue()     == [22::I,1]
sample()   == red()
hue c      == c.hue
i:PositiveInteger * c:% == i::SF * c
numberOfHues() == totalHues

color i ==
  if (i<0) or (i>totalHues) then
    error concat("Color should be in the range 1..",totalHues::String)
  [i::I, 1]

coerce(c:%):OutputForm ==
  hconcat ["Hue: ">::OutputForm, (c.hue)::OutputForm,
    " Weight: ">::OutputForm, (c.weight)::OutputForm]

```

— COLOR.dotabb —

```

"COLOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=COLOR"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"COLOR" -> "BASTYPE"
"COLOR" -> "KOERCE"

```

domain COMM Commutator

— Commutator.input —

```

)set break resume

```

```

)sys rm -f Commutator.output
)spool Commutator.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Commutator
--R Commutator is a domain constructor
--R Abbreviation for Commutator is COMM
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for COMM
--R
--R----- Operations -----
--R ?? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger       latex : % -> String
--R mkcomm : (%,% ) -> %           mkcomm : Integer -> %
--R ?~? : (%,% ) -> Boolean
--R
--E 1

)spool
)lisp (bye)

```

— Commutator.help —

```

=====
Commutator examples
=====

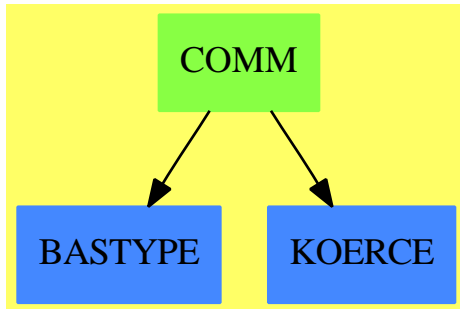
```

A type for basic commutators

See Also:

o)show Commutator

Commutator (COMM)



See

⇒ “OrdSetInts” (OSI) [16](#) on page [1864](#)

⇒ “FreeNilpotentLie” (FNLA) [7](#) on page [1027](#)

Exports:

coerce hash latex mkcomm ?=? ?~=?

— domain COMM Commutator —

```

)abbrev domain COMM Commutator
++ Author : Larry Lambe
++ Date created: 30 June 1988.
++ Updated      : 10 March 1991
++ Description:
++ A type for basic commutators

Commutator: Export == Implement where
  I  ==> Integer
  OSI ==> OrdSetInts
  O  ==> OutputForm

  Export == SetCategory with
    mkcomm : I -> %
      ++ mkcomm(i) is not documented
    mkcomm : (%,% ) -> %
      ++ mkcomm(i,j) is not documented

  Implement == add
    P := Record(left:%,right:%)
    Rep := Union(OSI,P)
    x,y: %
    i : I

    x = y ==
      (x case OSI) and (y case OSI) => x::OSI = y::OSI

```

```

(x case P) and (y case P) =>
  xx:P := x::P
  yy:P := y::P
  (xx.right = yy.right) and (xx.left = yy.left)
false

mkcomm(i) == i::OSI
mkcomm(x,y) == construct(x,y)$P

coerce(x: %): 0 ==
  x case OSI => x::OSI::0
  xx := x::P
  bracket([xx.left::0,xx.right::0])$0

```

— COMM.dotabb —

```

"COMM" [color="#88FF44",href="bookvol10.3.pdf#nameddest=COMM"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"COMM" -> "BASTYPE"
"COMM" -> "KOERCE"

```

domain COMPLEX Complex

— Complex.input —

```

)set break resume
)sys rm -f Complex.output
)spool Complex.output
)set message test on
)set message auto off
)clear all
--S 1 of 16
a := complex(4/3,5/2)
--R
--R
--R      4  5
--R  (1) - + - %i
--R      3  2
--R
Type: Complex(Fraction(Integer))

```

--E 1

--S 2 of 16

b := complex(4/3,-5/2)

--R

--R

--R (2)
$$-\frac{4}{3} - \frac{5}{2}i$$

--R

Type: Complex(Fraction(Integer))

--E 2

--S 3 of 16

a + b

--R

--R

--R (3)
$$-\frac{8}{3}$$

--R

Type: Complex(Fraction(Integer))

--E 3

--S 4 of 16

a - b

--R

--R

--R (4)
$$5i$$

--R

Type: Complex(Fraction(Integer))

--E 4

--S 5 of 16

a * b

--R

--R

--R (5)
$$-\frac{289}{36}$$

--R

Type: Complex(Fraction(Integer))

--E 5

--S 6 of 16

a / b

--R

--R

--R (6)
$$-\frac{161}{289} + \frac{240}{289}i$$

--R

Type: Complex(Fraction(Integer))

--E 6


```

--S 7 of 16
% :: Fraction Complex Integer
--R
--R
--R      - 15 + 8%i
--R (7)  -----
--R      15 + 8%i
--R
--R                                          Type: Fraction(Complex(Integer))
--E 7

--S 8 of 16
3.4 + 6.7 * %i
--R
--R
--R (8)  3.4 + 6.7 %i
--R
--R                                          Type: Complex(Float)
--E 8

--S 9 of 16
conjugate a
--R
--R
--R      4 5
--R (9)  - - - %i
--R      3 2
--R
--R                                          Type: Complex(Fraction(Integer))
--E 9

--S 10 of 16
norm a
--R
--R
--R      289
--R (10) ---
--R      36
--R
--R                                          Type: Fraction(Integer)
--E 10

--S 11 of 16
real a
--R
--R
--R      4
--R (11) -
--R      3
--R
--R                                          Type: Fraction(Integer)
--E 11

--S 12 of 16
imag a

```

```

--R
--R
--R      5
--R  (12)  -
--R      2
--R
--R                                          Type: Fraction(Integer)
--E 12

--S 13 of 16
gcd(13 - 13*%i, 31 + 27*%i)
--R
--R
--R  (13)  5 + %i
--R
--R                                          Type: Complex(Integer)
--E 13

--S 14 of 16
lcm(13 - 13*%i, 31 + 27*%i)
--R
--R
--R  (14)  143 - 39%i
--R
--R                                          Type: Complex(Integer)
--E 14

--S 15 of 16
factor(13 - 13*%i)
--R
--R
--R  (15)  - (1 + %i)(2 + 3%i)(3 + 2%i)
--R
--R                                          Type: Factored(Complex(Integer))
--E 15

--S 16 of 16
factor complex(2,0)
--R
--R
--R      2
--R  (16)  - %i (1 + %i)
--R
--R                                          Type: Factored(Complex(Integer))
--E 16
)spool
)lisp (bye)

```

— Complex.help —

=====
Complex examples

=====

The Complex constructor implements complex objects over a commutative ring R. Typically, the ring R is Integer, Fraction Integer, Float or DoubleFloat. R can also be a symbolic type, like Polynomial Integer.

Complex objects are created by the complex operation.

```
a := complex(4/3,5/2)
      4      5
      - + - %i
      3      2
                                     Type: Complex Fraction Integer
```

```
b := complex(4/3,-5/2)
      4      5
      - - - %i
      3      2
                                     Type: Complex Fraction Integer
```

The standard arithmetic operations are available.

```
a + b
      8
      -
      3
                                     Type: Complex Fraction Integer
```

```
a - b
      5%i
                                     Type: Complex Fraction Integer
```

```
a * b
      289
      ---
      36
                                     Type: Complex Fraction Integer
```

If R is a field, you can also divide the complex objects.

```
a / b
      161      240
      - --- + --- %i
      289      289
                                     Type: Complex Fraction Integer
```

We can view the last object as a fraction of complex integers.

```
% :: Fraction Complex Integer
```

$$\frac{-15 + 8i}{15 + 8i}$$

Type: Fraction Complex Integer

The predefined macro %i is defined to be complex(0,1).

$$\frac{3.4 + 6.7 * \%i}{3.4 + 6.7 \%i}$$

Type: Complex Float

You can also compute the conjugate and norm of a complex number.

$$\frac{\text{conjugate } a}{4 - 5i}$$

Type: Complex Fraction Integer

$$\frac{\text{norm } a}{289}$$

Type: Fraction Integer

The real and imag operations are provided to extract the real and imaginary parts, respectively.

$$\frac{\text{real } a}{4}$$

Type: Fraction Integer

$$\frac{\text{imag } a}{5}$$

Type: Fraction Integer

The domain Complex Integer is also called the Gaussian integers. If R is the integers (or, more generally, a EuclideanDomain), you can compute greatest common divisors.

$$\frac{\text{gcd}(13 - 13i, 31 + 27i)}{5 + i}$$

Type: Complex Integer

You can also compute least common multiples.

```
lcm(13 - 13*i, 31 + 27*i)
143 - 39*i
Type: Complex Integer
```

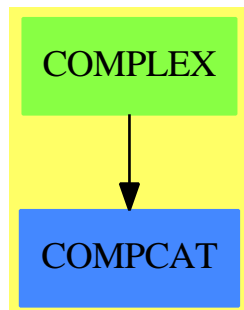
You can factor Gaussian integers.

```
factor(13 - 13*i)
- (1 + i)(2 + 3*i)(3 + 2*i)
Type: Factored Complex Integer
```

```
factor complex(2,0)
      2
- %i (1 + %i)
Type: Factored Complex Integer
```

See Also
o)show Complex

Complex (COMPLEX)



Exports:

0	1	abs
acos	acosh	acot
acoth	acsc	acsch
argument	asec	asech
asin	asinh	associates?
atan	atanh	basis
characteristic	characteristicPolynomial	charthRoot
coerce	complex	conditionP
conjugate	convert	coordinates
cos	cosh	cot
coth	createPrimitiveElement	csc
csch	D	definingPolynomial
derivationCoordinates	differentiate	discreteLog
discriminant	divide	euclideanSize
eval	exp	expressIdealMember
exquo	extendedEuclidean	factor
factorPolynomial	factorSquareFreePolynomial	factorsOfCyclicGroupSize
gcd	gcdPolynomial	generator
hash	imag	imaginary
index	init	inv
latex	lcm	lift
log	lookup	map
max	min	minimalPolynomial
multiEuclidean	nextItem	norm
nthRoot	OMwrite	one?
order	patternMatch	pi
polarCoordinates	prime?	primeFrobenius
primitive?	primitiveElement	principalIdeal
random	rank	rational
rational?	rationalIfCan	real
recip	reduce	reducedSystem
regularRepresentation	representationType	represents
retract	retractIfCan	sample
sec	sech	sin
sinh	size	sizeLess?
solveLinearPolynomialEquation	sqrt	squareFree
squareFreePart	squareFreePolynomial	subtractIfCan
tableForDiscreteLogarithm	tan	tanh
trace	traceMatrix	unit?
unitCanonical	unitNormal	zero?
?*?	?**?	?+?
?-?	?-?	?=?
?^?	?~=?	?/?
?<?	?<=?	?>?
?>=?	?..?	?quo?
?rem?		

— domain COMPLEX Complex —

```

)abbrev domain COMPLEX Complex
++ Author: Mark Botch
++ Description:
++ \spadtype{Complex(R)} creates the domain of elements of the form
++ \spad{a + b * i} where \spad{a} and b come from the ring R,
++ and i is a new element such that \spad{i**2 = -1}.

Complex(R:CommutativeRing): ComplexCategory(R) with
  if R has OpenMath then OpenMath
  == add
    Rep := Record(real:R, imag:R)

  if R has OpenMath then
    writeOMComplex(dev: OpenMathDevice, x: %): Void ==
      OMputApp(dev)
      OMputSymbol(dev, "complex1", "complex__cartesian")
      OMwrite(dev, real x)
      OMwrite(dev, imag x)
      OMputEndApp(dev)

    OMwrite(x: %): String ==
      s: String := ""
      sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
      dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML)
      OMputObject(dev)
      writeOMComplex(dev, x)
      OMputEndObject(dev)
      OMclose(dev)
      s := OM_-STRINGPTRTOSTRING(sp)$Lisp pretend String
      s

    OMwrite(x: %, wholeObj: Boolean): String ==
      s: String := ""
      sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
      dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML)
      if wholeObj then
        OMputObject(dev)
      writeOMComplex(dev, x)
      if wholeObj then
        OMputEndObject(dev)
      OMclose(dev)
      s := OM_-STRINGPTRTOSTRING(sp)$Lisp pretend String
      s

    OMwrite(dev: OpenMathDevice, x: %): Void ==
      OMputObject(dev)
      writeOMComplex(dev, x)

```

```

OMputEndObject(dev)

OMwrite(dev: OpenMathDevice, x: %, wholeObj: Boolean): Void ==
  if wholeObj then
    OMputObject(dev)
  writeOMComplex(dev, x)
  if wholeObj then
    OMputEndObject(dev)

0 == [0, 0]
1 == [1, 0]
zero? x == zero?(x.real) and zero?(x.imag)
-- one? x == one?(x.real) and zero?(x.imag)
one? x == ((x.real) = 1) and zero?(x.imag)
coerce(r:R):% == [r, 0]
complex(r, i) == [r, i]
real x == x.real
imag x == x.imag
x + y == [x.real + y.real, x.imag + y.imag]
-- by re-defining this here, we save 5 fn calls
x:% * y:% ==
  [x.real * y.real - x.imag * y.imag,
   x.imag * y.real + y.imag * x.real] -- here we save nine!

if R has IntegralDomain then
  _exquo(x:%, y:%) == -- to correct bad defaulting problem
    zero? y.imag => x exquo y.real
    x * conjugate(y) exquo norm(y)

-----

-- COMPLEX.dotabb --

"COMPLEX" [color="#88FF44",href="bookvol10.3.pdf#nameddest=COMPLEX"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"COMPLEX" -> "COMPCAT"

-----

```

domain CDFMAT ComplexDoubleFloatMatrix

— ComplexDoubleFloatMatrix.input —

```
)set break resume
```



```

)sys rm -f ComplexDoubleFloatMatrix.output
)spool ComplexDoubleFloatMatrix.output
)set message test on
)set message auto off
)clear all

--S 1 of 6
)show ComplexDoubleFloatMatrix
--R
--R ComplexDoubleFloatMatrix is a domain constructor
--R Abbreviation for ComplexDoubleFloatMatrix is CDFMAT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for CDFMAT
--R
--R----- Operations -----
--R ???: (Integer,%) -> %          ??? : (%,%) -> %
--R ?+?: (%,%) -> %              -? : % -> %
--R ?-?: (%,%) -> %              antisymmetric? : % -> Boolean
--R copy : % -> %                diagonal? : % -> Boolean
--R diagonalMatrix : List(%) -> % empty : () -> %
--R empty? : % -> Boolean         eq? : (%,%) -> Boolean
--R horizConcat : (%,%) -> %     maxColIndex : % -> Integer
--R maxRowIndex : % -> Integer    minColIndex : % -> Integer
--R minRowIndex : % -> Integer    ncols : % -> NonNegativeInteger
--R nrows : % -> NonNegativeInteger qnew : (Integer,Integer) -> %
--R sample : () -> %             square? : % -> Boolean
--R squareTop : % -> %           symmetric? : % -> Boolean
--R transpose : % -> %           vertConcat : (%,%) -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ??? : (ComplexDoubleFloatVector,%) -> ComplexDoubleFloatVector
--R ??? : (%,ComplexDoubleFloatVector) -> ComplexDoubleFloatVector
--R ??? : (%,Complex(DoubleFloat)) -> %
--R ??? : (Complex(DoubleFloat),%) -> %
--R ??? : (%,Integer) -> % if Complex(DoubleFloat) has FIELD
--R ??? : (%,NonNegativeInteger) -> %
--R ??/? : (%,Complex(DoubleFloat)) -> % if Complex(DoubleFloat) has FIELD
--R ??/? : (%,%) -> Boolean if Complex(DoubleFloat) has SETCAT
--R any? : ((Complex(DoubleFloat) -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : ComplexDoubleFloatVector -> %
--R coerce : % -> OutputForm if Complex(DoubleFloat) has SETCAT
--R column : (%,Integer) -> ComplexDoubleFloatVector
--R columnSpace : % -> List(ComplexDoubleFloatVector) if Complex(DoubleFloat) has EUCLDOM
--R count : (Complex(DoubleFloat),%) -> NonNegativeInteger if $ has finiteAggregate and Complex(
--R count : ((Complex(DoubleFloat) -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R determinant : % -> Complex(DoubleFloat) if Complex(DoubleFloat) has commutative(*)
--R diagonalMatrix : List(Complex(DoubleFloat)) -> %
--R elt : (%,List(Integer),List(Integer)) -> %
--R elt : (%,Integer,Integer,Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R elt : (%,Integer,Integer) -> Complex(DoubleFloat)
--R eval : (%,List(Complex(DoubleFloat)),List(Complex(DoubleFloat))) -> % if Complex(DoubleFloat)

```

```

--R eval : (% ,Complex(DoubleFloat),Complex(DoubleFloat)) -> % if Complex(DoubleFloat) has EVALAB(COMPLEX(DFLOAT))
--R eval : (% ,Equation(Complex(DoubleFloat))) -> % if Complex(DoubleFloat) has EVALAB(COMPLEX(DFLOAT)) and (
--R eval : (% ,List(Equation(Complex(DoubleFloat)))) -> % if Complex(DoubleFloat) has EVALAB(COMPLEX(DFLOAT))
--R every? : ((Complex(DoubleFloat) -> Boolean),%) -> Boolean if $ has finiteAggregate
--R exquo : (% ,Complex(DoubleFloat)) -> Union(%, "failed") if Complex(DoubleFloat) has INTDOM
--R fill! : (% ,Complex(DoubleFloat)) -> %
--R hash : % -> SingleInteger if Complex(DoubleFloat) has SETCAT
--R inverse : % -> Union(%, "failed") if Complex(DoubleFloat) has FIELD
--R latex : % -> String if Complex(DoubleFloat) has SETCAT
--R less? : (% ,NonNegativeInteger) -> Boolean
--R listOfLists : % -> List(List(Complex(DoubleFloat)))
--R map : (((Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)),%,%,Complex(DoubleFloat)) -> %
--R map : (((Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)),%,%) -> %
--R map : ((Complex(DoubleFloat) -> Complex(DoubleFloat)),%) -> %
--R map! : ((Complex(DoubleFloat) -> Complex(DoubleFloat)),%) -> %
--R matrix : (NonNegativeInteger,NonNegativeInteger,((Integer,Integer) -> Complex(DoubleFloat))) -> %
--R matrix : List(List(Complex(DoubleFloat))) -> %
--R member? : (Complex(DoubleFloat),%) -> Boolean if $ has finiteAggregate and Complex(DoubleFloat) has SETCAT
--R members : % -> List(Complex(DoubleFloat)) if $ has finiteAggregate
--R minordet : % -> Complex(DoubleFloat) if Complex(DoubleFloat) has commutative(*)
--R more? : (% ,NonNegativeInteger) -> Boolean
--R new : (NonNegativeInteger,NonNegativeInteger,Complex(DoubleFloat)) -> %
--R nullSpace : % -> List(ComplexDoubleFloatVector) if Complex(DoubleFloat) has INTDOM
--R nullity : % -> NonNegativeInteger if Complex(DoubleFloat) has INTDOM
--R parts : % -> List(Complex(DoubleFloat))
--R pfaffian : % -> Complex(DoubleFloat) if Complex(DoubleFloat) has COMRING
--R qelt : (% ,Integer,Integer) -> Complex(DoubleFloat)
--R qsetelt! : (% ,Integer,Integer,Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R rank : % -> NonNegativeInteger if Complex(DoubleFloat) has INTDOM
--R row : (% ,Integer) -> ComplexDoubleFloatVector
--R rowEchelon : % -> % if Complex(DoubleFloat) has EUCDOM
--R scalarMatrix : (NonNegativeInteger,Complex(DoubleFloat)) -> %
--R setColumn! : (% ,Integer,ComplexDoubleFloatVector) -> %
--R setRow! : (% ,Integer,ComplexDoubleFloatVector) -> %
--R setelt : (% ,List(Integer),List(Integer),%) -> %
--R setelt : (% ,Integer,Integer,Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R setsubMatrix! : (% ,Integer,Integer,%) -> %
--R size? : (% ,NonNegativeInteger) -> Boolean
--R subMatrix : (% ,Integer,Integer,Integer,Integer) -> %
--R swapColumns! : (% ,Integer,Integer) -> %
--R swapRows! : (% ,Integer,Integer) -> %
--R transpose : ComplexDoubleFloatVector -> %
--R zero : (NonNegativeInteger,NonNegativeInteger) -> %
--R ?~=? : (% ,%) -> Boolean if Complex(DoubleFloat) has SETCAT
--R
--E 1

--S 2 of 6
a:CDFMAT:=qnew(2,3)
--R

```

```

--R      +0.  0.  0.+
--R  (1)  |      |
--R      +0.  0.  0.+
--R
--R                                          Type: ComplexDoubleFloatMatrix
--E 2

--S 3 of 6
qsetelt!(a,1,1,1.0+2*%i)
--R
--R  (2)  1. + 2. %i
--R
--R                                          Type: Complex(DoubleFloat)
--E 3

--S 4 of 6
a
--R
--R      +0.      0.      0.+
--R  (3)  |      |
--R      +0.  1. + 2. %i  0.+
--R
--R                                          Type: ComplexDoubleFloatMatrix
--E 4

--S 5 of 6
qsetelt!(a,0,0,2.0+4*%i)
--R
--R  (4)  2. + 4. %i
--R
--R                                          Type: Complex(DoubleFloat)
--E 5

--S 6 of 6
a
--R
--R      +2. + 4. %i      0.      0.+
--R  (5)  |      |
--R      +      0.      1. + 2. %i  0.+
--R
--R                                          Type: ComplexDoubleFloatMatrix
--E 6

)spool
)lisp (bye)

```

— ComplexDoubleFloatMatrix.help —

```

=====
ComplexDoubleFloatMatrix examples
=====

```

This domain creates a lisp simple array of machine doublefloats. It provides one new function called qnew which takes an integer that gives the array length.

NOTE: Unlike normal Axiom arrays the ComplexDoubleFloatMatrix arrays are 0-based so the first element is 0. Axiom arrays normally start at 1.

```
a:CDFMAT:=qnew(2,3)
+0.  0.  0.+
|      |
+0.  0.  0.+
```

```
qsetelt!(a,1,1,1.0+2*%i)
1. + 2. %i
```

```
a
+0.      0.      0.+
|          |
+0.  1. + 2. %i  0.+
```

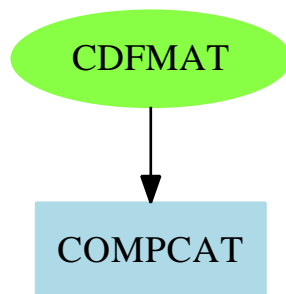
```
qsetelt!(a,0,0,2.0+4*%i)
2. + 4. %i
```

```
a
+2. + 4. %i      0.      0.+
|          |
+  0.      1. + 2. %i  0.+
```

See Also:

- o)help Float
- o)help DoubleFloat
- o)show ComplexDoubleFloatMatrix

ComplexDoubleFloatMatrix (CDFMAT)



Exports:

#?	-?	?**?	?*?
?+?	?-?	?/?	?=?
?~=?	antisymmetric?	any?	coerce
column	columnSpace	copy	count
determinant	diagonal?	diagonalMatrix	elt
empty	empty?	eq?	eval
every?	exquo	fill!	hash
horizConcat	inverse	latex	less?
listOfLists	map	map!	matrix
maxColIndex	maxRowIndex	member?	members
minColIndex	minRowIndex	minordet	more?
ncols	new	nrows	nullSpace
nullity	parts	pfaffian	qelt
qnew	qsetelt!	rank	row
rowEchelon	sample	scalarMatrix	setColumn!
setRow!	setelt	setsubMatrix!	size?
square?	squareTop	subMatrix	swapColumns!
swapRows!	symmetric?	transpose	vertConcat
zero			

— domain CDFMAT ComplexDoubleFloatMatrix —

```
)abbrev domain CDFMAT ComplexDoubleFloatMatrix
++ Author: Waldek Hebisch
++ Description:
++ This is a low-level domain which implements matrices
++ (two dimensional arrays) of complex double precision floating point
++ numbers. Indexing is 0 based, there is no bound checking (unless
++ provided by lower level).
```

```
ComplexDoubleFloatMatrix : MatrixCategory(Complex DoubleFloat,
ComplexDoubleFloatVector,
ComplexDoubleFloatVector) with
```

```

qnew : (Integer, Integer) -> %
  ++ qnew(n, m) creates a new uninitialized n by m matrix.
  ++
  ++X t1:CDFMAT:=qnew(3,4)

== add

NNI ==> Integer
Qelt2 ==> CDAREF2$Lisp
Qsetelt2 ==> CDSETAREF2$Lisp
Qnrows ==> CDANROWS$Lisp
Qncols ==> CDANCOLS$Lisp
Qnew ==> MAKE_-CDOUBLE_-MATRIX$Lisp

minRowIndex x == 0
minColIndex x == 0
nrows x == Qnrows(x)
ncols x == Qncols(x)
maxRowIndex x == Qnrows(x) - 1
maxColIndex x == Qncols(x) - 1

qelt(m, i, j) == Qelt2(m, i, j)
qsetelt_!(m, i, j, r) == Qsetelt2(m, i, j, r)

empty() == Qnew(0$Integer, 0$Integer)
qnew(rows, cols) == Qnew(rows, cols)
new(rows, cols, a) ==
  res := Qnew(rows, cols)
  for i in 0..(rows - 1) repeat
    for j in 0..(cols - 1) repeat
      Qsetelt2(res, i, j, a)
  res

```

— CDFMAT.dotabb —

```

"CDFMAT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=CDFMAT",
          shape=ellipse]
"COMPCAT" [color=lightblue,href="bookvol10.2.pdf#nameddest=COMPCAT"];
"CDFMAT" -> "COMPCAT"

```

domain CDFVEC ComplexDoubleFloatVector

— ComplexDoubleFloatVector.input —

```
)set break resume
)sys rm -f ComplexDoubleFloatVector.output
)spool ComplexDoubleFloatVector.output
)set message test on
)set message auto off
)clear all

--S 1 of 6
)show ComplexDoubleFloatVector
--R
--R ComplexDoubleFloatVector is a domain constructor
--R Abbreviation for ComplexDoubleFloatVector is CDFVEC
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for CDFVEC
--R
--R----- Operations -----
--R concat : List(%) -> %                concat : (%,%) -> %
--R copy : % -> %                        delete : (%,Integer) -> %
--R empty : () -> %                      empty? : % -> Boolean
--R eq? : (%,%) -> Boolean                index? : (Integer,%) -> Boolean
--R indices : % -> List(Integer)          insert : (%,%,Integer) -> %
--R qnew : Integer -> %                  reverse : % -> %
--R sample : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?? : (%,Complex(DoubleFloat)) -> % if Complex(DoubleFloat) has MONOID
--R ?? : (Complex(DoubleFloat),%) -> % if Complex(DoubleFloat) has MONOID
--R ?? : (Integer,%) -> % if Complex(DoubleFloat) has ABELGRP
--R ?? : (%,%) -> % if Complex(DoubleFloat) has ABELSG
--R ?? : (%,%) -> % if Complex(DoubleFloat) has ABELGRP
--R -? : % -> % if Complex(DoubleFloat) has ABELGRP
--R ?<? : (%,%) -> Boolean if Complex(DoubleFloat) has ORDSET
--R ?<=? : (%,%) -> Boolean if Complex(DoubleFloat) has ORDSET
--R ?=? : (%,%) -> Boolean if Complex(DoubleFloat) has SETCAT
--R ?>? : (%,%) -> Boolean if Complex(DoubleFloat) has ORDSET
--R ?>=? : (%,%) -> Boolean if Complex(DoubleFloat) has ORDSET
--R any? : ((Complex(DoubleFloat) -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if Complex(DoubleFloat) has SETCAT
--R concat : (Complex(DoubleFloat),%) -> %
--R concat : (%,Complex(DoubleFloat)) -> %
--R construct : List(Complex(DoubleFloat)) -> %
--R convert : % -> InputForm if Complex(DoubleFloat) has KONVERT(INFORM)
--R copyInto! : (%,%,Integer) -> % if $ has shallowlyMutable
--R count : (Complex(DoubleFloat),%) -> NonNegativeInteger if $ has finiteAggregate and Complex(
--R count : ((Complex(DoubleFloat) -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
```

```

--R cross : (%,% ) -> % if Complex(DoubleFloat) has RING
--R delete : (% ,UniversalSegment(Integer)) -> %
--R dot : (%,% ) -> Complex(DoubleFloat) if Complex(DoubleFloat) has RING
--R ?.? : (% ,UniversalSegment(Integer)) -> %
--R ?.? : (% ,Integer) -> Complex(DoubleFloat)
--R elt : (% ,Integer,Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R entries : % -> List(Complex(DoubleFloat))
--R entry? : (Complex(DoubleFloat),%) -> Boolean if $ has finiteAggregate and Complex(DoubleFloat) has SETCAT
--R eval : (% ,List(Complex(DoubleFloat)),List(Complex(DoubleFloat))) -> % if Complex(DoubleFloat) has EVALAB(COMPLEX(DFLOAT))
--R eval : (% ,Complex(DoubleFloat),Complex(DoubleFloat)) -> % if Complex(DoubleFloat) has EVALAB(COMPLEX(DFLOAT))
--R eval : (% ,Equation(Complex(DoubleFloat))) -> % if Complex(DoubleFloat) has EVALAB(COMPLEX(DFLOAT)) and $ has finiteAggregate
--R eval : (% ,List(Equation(Complex(DoubleFloat)))) -> % if Complex(DoubleFloat) has EVALAB(COMPLEX(DFLOAT))
--R every? : ((Complex(DoubleFloat) -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (% ,Complex(DoubleFloat)) -> % if $ has shallowlyMutable
--R find : ((Complex(DoubleFloat) -> Boolean),%) -> Union(Complex(DoubleFloat),"failed")
--R first : % -> Complex(DoubleFloat) if Integer has ORDSET
--R hash : % -> SingleInteger if Complex(DoubleFloat) has SETCAT
--R insert : (Complex(DoubleFloat),% ,Integer) -> %
--R latex : % -> String if Complex(DoubleFloat) has SETCAT
--R length : % -> Complex(DoubleFloat) if Complex(DoubleFloat) has RADCAT and Complex(DoubleFloat) has RING
--R less? : (% ,NonNegativeInteger) -> Boolean
--R magnitude : % -> Complex(DoubleFloat) if Complex(DoubleFloat) has RADCAT and Complex(DoubleFloat) has RING
--R map : (((Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)),%,%) -> %
--R map : ((Complex(DoubleFloat) -> Complex(DoubleFloat)),%) -> %
--R map! : (((Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)),%) -> % if $ has shallowlyMutable
--R max : (%,% ) -> % if Complex(DoubleFloat) has ORDSET
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (Complex(DoubleFloat),%) -> Boolean if $ has finiteAggregate and Complex(DoubleFloat) has SETCAT
--R members : % -> List(Complex(DoubleFloat)) if $ has finiteAggregate
--R merge : (%,% ) -> % if Complex(DoubleFloat) has ORDSET
--R merge : (((Complex(DoubleFloat),Complex(DoubleFloat)) -> Boolean),%,%) -> %
--R min : (%,% ) -> % if Complex(DoubleFloat) has ORDSET
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (% ,NonNegativeInteger) -> Boolean
--R new : (NonNegativeInteger,Complex(DoubleFloat)) -> %
--R outerProduct : (%,% ) -> Matrix(Complex(DoubleFloat)) if Complex(DoubleFloat) has RING
--R parts : % -> List(Complex(DoubleFloat)) if $ has finiteAggregate
--R position : (Complex(DoubleFloat),% ,Integer) -> Integer if Complex(DoubleFloat) has SETCAT
--R position : (Complex(DoubleFloat),%) -> Integer if Complex(DoubleFloat) has SETCAT
--R position : ((Complex(DoubleFloat) -> Boolean),%) -> Integer
--R qelt : (% ,Integer) -> Complex(DoubleFloat)
--R qsetelt! : (% ,Integer,Complex(DoubleFloat)) -> Complex(DoubleFloat) if $ has shallowlyMutable
--R reduce : (((Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)),%) -> Complex(DoubleFloat)
--R reduce : (((Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)),% ,Complex(DoubleFloat)) -> %
--R reduce : (((Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)),% ,Complex(DoubleFloat),Complex(DoubleFloat)) -> %
--R remove : ((Complex(DoubleFloat) -> Boolean),%) -> % if $ has finiteAggregate
--R remove : (Complex(DoubleFloat),%) -> % if $ has finiteAggregate and Complex(DoubleFloat) has SETCAT
--R removeDuplicates : % -> % if $ has finiteAggregate and Complex(DoubleFloat) has SETCAT
--R reverse! : % -> % if $ has shallowlyMutable
--R select : ((Complex(DoubleFloat) -> Boolean),%) -> % if $ has finiteAggregate

```



```

--R setelt : (% , UniversalSegment(Integer), Complex(DoubleFloat)) -> Complex(DoubleFloat) if $ has
--R setelt : (% , Integer, Complex(DoubleFloat)) -> Complex(DoubleFloat) if $ has shallowlyMutable
--R size? : (% , NonNegativeInteger) -> Boolean
--R sort : % -> % if Complex(DoubleFloat) has ORDSET
--R sort : (((Complex(DoubleFloat), Complex(DoubleFloat)) -> Boolean), %) -> %
--R sort! : % -> % if $ has shallowlyMutable and Complex(DoubleFloat) has ORDSET
--R sort! : (((Complex(DoubleFloat), Complex(DoubleFloat)) -> Boolean), %) -> % if $ has shallowly
--R sorted? : % -> Boolean if Complex(DoubleFloat) has ORDSET
--R sorted? : (((Complex(DoubleFloat), Complex(DoubleFloat)) -> Boolean), %) -> Boolean
--R swap! : (% , Integer, Integer) -> Void if $ has shallowlyMutable
--R vector : List(Complex(DoubleFloat)) -> %
--R zero : NonNegativeInteger -> % if Complex(DoubleFloat) has ABELMON
--R ?~=? : (% , %) -> Boolean if Complex(DoubleFloat) has SETCAT
--R
--E 1

--S 2 of 6
t1:CDFVEC:=qnew(5)
--R
--R (1) [0., 0., 0., 0., 0.]
--R
--R Type: ComplexDoubleFloatVector
--E 2

-- NOTE: CDFVEC arrays are 0-based, normal Axiom arrays are 1-based
--S 3 of 6
t1.1:=1.0+2*%i
--R
--R (2) 1. + 2. %i
--R
--R Type: Complex(DoubleFloat)
--E 3

--S 4 of 6
t1
--R
--R (3) [0., 1. + 2. %i, 0., 0., 0.]
--R
--R Type: ComplexDoubleFloatVector
--E 4

--S 5 of 6
t1.0:=3.0+4.0*%i
--R
--R (4) 3. + 4. %i
--R
--R Type: Complex(DoubleFloat)
--E 5

--S 6 of 6
t1
--R
--R (5) [3. + 4. %i, 1. + 2. %i, 0., 0., 0.]
--R
--R Type: ComplexDoubleFloatVector

```

--E 6

)spool
)lisp (bye)

— ComplexDoubleFloatVector.help —

=====
ComplexDoubleFloatVector examples
=====

This domain creates a lisp simple array of machine complex doublefloats.
It provides one new function called qnew which takes an integer
that gives the array length.

NOTE: Unlike normal Axiom arrays the ComplexDoubleFloatVector arrays
are 0-based so the first element is 0. Axiom arrays normally
start at 1.

```
t1:CDFVEC:=qnew(5)
      [0.,0.,0.,0.,0.]
```

```
t1.1:=1.0+2*%i
      1. + 2. %i
```

```
t1
      [0.,1. + 2. %i,0.,0.,0.]
```

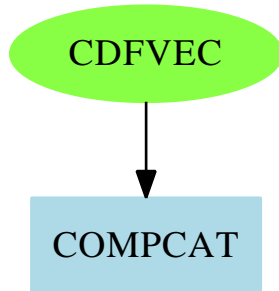
```
t1.0:=3.0+4.0*%i
      3. + 4. %i
```

```
t1
      [3. + 4. %i,1. + 2. %i,0.,0.,0.]
```

See Also:

- o)help Float
 - o)help DoubleFloat
 - o)show ComplexDoubleFloatVector
-

ComplexDoubleFloatVector (CDFVEC)

**Exports:**

#?	-?	?*?	?+?
?-?	?.?	?<=?	?<?
?=?	?>=?	?>?	?~=?
any?	coerce	concat	construct
convert	copy	copyInto!	count
cross	delete	dot	elt
empty	empty?	entries	entry?
eq?	eval	every?	fill!
find	first	hash	index?
indices	insert	latex	length
less?	magnitude	map	map!
max	maxIndex	member?	members
merge	min	minIndex	more?
new	outerProduct	parts	position
qelt	qnew	qsetelt!	reduce
remove	removeDuplicates	reverse	reverse!
sample	select	setelt	size?
sort	sort!	sorted?	swap!
vector	zero		

— domain CDFVEC ComplexDoubleFloatVector —

```

)abbrev domain CDFVEC ComplexDoubleFloatVector
++ Author: Waldek Hebisch
++ Description:
++ This is a low-level domain which implements vectors
++ (one dimensional arrays) of complex double precision floating point
++ numbers. Indexing is 0 based, there is no bound checking (unless
++ provided by lower level).
ComplexDoubleFloatVector : VectorCategory Complex DoubleFloat with
  qnew : Integer -> %
    ++ qnew(n) creates a new uninitialized vector of length n.
    ++
  
```

```

++X t1:CDFVEC:=qnew 7
vector: List Complex DoubleFloat -> %
++ vector(l) converts the list l to a vector.
++
++X t1:List(Complex(DoubleFloat)):= [1+2*i, 3+4*i, -5-6*i]
++X t2:CDFVEC:=vector(t1)
== add

```

```

Qelt1 ==> CDELT$Lisp
Qsetelt1 ==> CDSELT$Lisp

```

```

qelt(x, i) == Qelt1(x, i)
qsetelt_!(x, i, s) == Qsetelt1(x, i, s)
Qsize ==> CDLEN$Lisp
Qnew ==> MAKE_-CDOUBLE_-VECTOR$Lisp

```

```

#x == Qsize x
minIndex x == 0
empty() == Qnew(0$Lisp)
qnew(n) == Qnew(n)
new(n, x) ==
    res := Qnew(n)
    fill_!(res, x)
qelt(x, i) == Qelt1(x, i)
elt(x:%, i:Integer) == Qelt1(x, i)
qsetelt_!(x, i, s) == Qsetelt1(x, i, s)
setelt(x : %, i : Integer, s : Complex DoubleFloat) ==
    Qsetelt1(x, i, s)
fill_!(x, s) ==
    for i in 0..((Qsize(x)) - 1) repeat Qsetelt1(x, i, s)
x

```

— CDFVEC.dotabb —

```

"CDFVEC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=CDFVEC",
    shape=ellipse]
"COMPCAT" [color=lightblue,href="bookvol10.2.pdf#nameddest=COMPCAT"];
"CDFVEC" -> "COMPCAT"

```

domain CONTFRAC ContinuedFraction

— ContinuedFraction.input —

```

)set break resume
)sys rm -f ContinuedFraction.output
)spool ContinuedFraction.output
)set message test on
)set message auto off
)clear all
--S 1 of 22
c := continuedFraction(314159/100000)
--R
--R
--R      1 |      1 |      1 |      1 |      1 |      1 |      1 |
--R  (1)  3 + +---+ + +---+ + +---+ + +---+ + +---+ + +---+ + +---+
--R      | 7      | 15      | 1      | 25      | 1      | 7      | 4
--R
--R                                          Type: ContinuedFraction(Integer)
--E 1

--S 2 of 22
partialQuotients c
--R
--R
--R  (2)  [3,7,15,1,25,1,7,4]
--R
--R                                          Type: Stream(Integer)
--E 2

--S 3 of 22
convergents c
--R
--R
--R      22 333 355 9208 9563 76149 314159
--R  (3)  [3,--,--,--,--,--,--,-----]
--R      7 106 113 2931 3044 24239 100000
--R
--R                                          Type: Stream(Fraction(Integer))
--E 3

--S 4 of 22
approximants c
--R
--R
--R      22 333 355 9208 9563 76149 314159
--R  (4)  [3,--,--,--,--,--,--,-----]
--R      7 106 113 2931 3044 24239 100000
--R
--R                                          Type: Stream(Fraction(Integer))
--E 4

```

--S 5 of 22

pq := partialQuotients(1/c)

--R

--R

--R (5) [0,3,7,15,1,25,1,7,4]

--R

Type: Stream(Integer)

--E 5

--S 6 of 22

continuedFraction(first pq, repeating [1], rest pq)

--R

--R

--R 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

--R (6) +---+ + +---+ + +---+ + +---+ + +---+ + +---+ + +---+ + +---+

--R | 3 | 7 | 15 | 1 | 25 | 1 | 7 | 4

--R

Type: ContinuedFraction(Integer)

--E 6

--S 7 of 22

z:=continuedFraction(3, repeating [1], repeating [3,6])

--R

--R

--R (7)

--R 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

--R 3 + +---+ + +---+ + +---+ + +---+ + +---+ + +---+ + +---+ + +---+

--R | 3 | 6 | 3 | 6 | 3 | 6 | 3 | 6 | 3 | 3

--R +

--R 1 |

--R +---+ + ...

--R | 6

--R

Type: ContinuedFraction(Integer)

--E 7

--S 8 of 22

dens:Stream Integer := cons(1, generate((x+>x+4), 6))

--R

--R

--R (8) [1,6,10,14,18,22,26,30,34,38,...]

--R

Type: Stream(Integer)

--E 8

--S 9 of 22

cf := continuedFraction(0, repeating [1], dens)

--R

--R

--R (9)

--R 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

--R +---+ + +---+ + +---+ + +---+ + +---+ + +---+ + +---+

--R | 1 | 6 | 10 | 14 | 18 | 22 | 26 | 30

```

--R  +
--R      1 |      1 |
--R      +-----+ + +-----+ + ...
--R      | 34      | 38
--R
--R                                          Type: ContinuedFraction(Integer)
--E 9

--S 10 of 22
ccf := convergents cf
--R
--R
--R      6 61 860 15541 342762 8927353 268163352 9126481321
--R  (10) [0,1,--,--,--,--,--,--,--,--,--,--,...]
--R      7 71 1001 18089 398959 10391023 312129649 10622799089
--R                                          Type: Stream(Fraction(Integer))
--E 10

--S 11 of 22
eConvergents := [2*e + 1 for e in ccf]
--R
--R
--R      19 193 2721 49171 1084483 28245729 848456353 28875761731
--R  (11) [1,3,--,--,--,--,--,--,--,--,--,--,...]
--R      7 71 1001 18089 398959 10391023 312129649 10622799089
--R                                          Type: Stream(Fraction(Integer))
--E 11

--S 12 of 22
eConvergents :: Stream Float
--R
--R
--R  (12)
--R  [1.0, 3.0, 2.7142857142 857142857, 2.7183098591 549295775,
--R   2.7182817182 817182817, 2.7182818287 356957267, 2.7182818284 585634113,
--R   2.7182818284 590458514, 2.7182818284 590452348, 2.7182818284 590452354,
--R   ...]
--R
--R                                          Type: Stream(Float)
--E 12

--S 13 of 22
exp 1.0
--R
--R
--R  (13) 2.7182818284 590452354
--R
--R                                          Type: Float
--E 13

--S 14 of 22
cf := continuedFraction(1,[(2*i+1)**2 for i in 0..],repeating [2])
--R

```

```

--R      (14)
--R      1 | 9 | 25 | 49 | 81 | 121 | 169 | 225 |
--R      1 + +---+ + +---+ + +---+ + +---+ + +---+ + +---+ + +---+ + +---+
--R      | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
--R      +
--R      289 | 361 |
--R      +-----+ + +-----+ + ...
--R      | 2 | 2 |
--R
--R                                          Type: ContinuedFraction(Integer)
--E 14

--S 15 of 22
ccf := convergents cf
--R
--R
--R      3 15 105 315 3465 45045 45045 765765 14549535
--R      (15) [1,--,---,---,---,---,---,---,---,---,...]
--R      2 13 76 263 2578 36979 33976 622637 11064338
--R
--R                                          Type: Stream(Fraction(Integer))
--E 15

--S 16 of 22
piConvergents := [4/p for p in ccf]
--R
--R
--R      8 52 304 1052 10312 147916 135904 2490548 44257352
--R      (16) [4,--,---,---,---,---,---,---,---,---,...]
--R      3 15 105 315 3465 45045 45045 765765 14549535
--R
--R                                          Type: Stream(Fraction(Integer))
--E 16

--S 17 of 22
piConvergents :: Stream Float
--R
--R
--R      (17)
--R      [4.0, 2.6666666666 666666667, 3.4666666666 666666667,
--R      2.8952380952 380952381, 3.3396825396 825396825, 2.9760461760 461760462,
--R      3.2837384837 384837385, 3.0170718170 718170718, 3.2523659347 188758953,
--R      3.0418396189 294022111, ...]
--R
--R                                          Type: Stream(Float)
--E 17

--S 18 of 22
continuedFraction((- 122 + 597*i)/(4 - 4*i))
--R
--R
--R      1 | 1 |
--R      (18) - 90 + 597*i + +-----+ + +-----+

```



```

--R          | 1 - 2%i      | - 1 + 2%i
--R                                     Type: ContinuedFraction(Complex(Integer))
--E 18

--S 19 of 22
r : Fraction UnivariatePolynomial(x,Fraction Integer)
--R
--R
--R                                     Type: Void
--E 19

--S 20 of 22
r := ((x - 1) * (x - 2)) / ((x-3) * (x-4))
--R
--R
--R          2
--R      x  - 3x + 2
--R (20)  -----
--R          2
--R      x  - 7x + 12
--R                                     Type: Fraction(UnivariatePolynomial(x,Fraction(Integer)))
--E 20

--S 21 of 22
continuedFraction r
--R
--R
--R          1      |      1      |
--R (21)  1 + +-----+ + +-----+
--R          | 1      9      | 16      40
--R          | - x - -      | -- x - --
--R          | 4      8      | 3      3
--R                                     Type: ContinuedFraction(UnivariatePolynomial(x,Fraction(Integer)))
--E 21

--S 22 of 22
[i*i for i in convergents(z) :: Stream Float]
--R
--R
--R (22)
--R [9.0, 11.1111111111 11111111, 10.9944598337 9501385, 11.0002777777 77777778,
--R 10.9999860763 98799786, 11.0000006979 29731039, 10.9999999650 15834446,
--R 11.0000000017 53603304, 10.9999999999 12099531, 11.0000000000 04406066,
--R ...]
--R
--R                                     Type: Stream(Float)
--E 22
)spool
)lisp (bye)

```

— ContinuedFraction.help —

===== ContinuedFraction examples =====

Continued fractions have been a fascinating and useful tool in mathematics for well over three hundred years. Axiom implements continued fractions for fractions of any Euclidean domain. In practice, this usually means rational numbers. In this section we demonstrate some of the operations available for manipulating both finite and infinite continued fractions.

The ContinuedFraction domain is a field and therefore you can add, subtract, multiply and divide the fractions.

The continuedFraction operation converts its fractional argument to a continued fraction.

```
c := continuedFraction(314159/100000)
      1 |   1 |   1 |   1 |   1 |   1 |   1 |
3 + +---+ + +---+ + +---+ + +---+ + +---+ + +---+ + +---+
   | 7   | 15   | 1   | 25   | 1   | 7   | 4
                                Type: ContinuedFraction Integer
```

This display is a compact form of the bulkier

```
3 +
  -----
  7 +
    -----
    15 +
        -----
        1 +
            -----
            25 +
                -----
                1 +
                    -----
                    7 +
                        -----
                        1
                        4
```

You can write any rational number in a similar form. The fraction will be finite and you can always take the "numerators" to be 1. That is, any rational number can be written as a simple, finite continued fraction of the form

```
a(1) +
      1
```

$$\begin{array}{r}
 \text{-----} \\
 a(2) + \frac{1}{\text{-----}} \\
 \quad a(3) + \frac{1}{\text{-----}} \\
 \quad \quad \cdot \\
 \quad \quad \cdot \\
 \quad \quad \cdot \\
 \quad \quad \quad \frac{1}{\text{-----}} \\
 \quad \quad \quad a(n-1) + \frac{1}{\text{-----}} \\
 \quad \quad \quad \quad a(n)
 \end{array}$$

The $a(i)$ are called partial quotients and the operation `partialQuotients` creates a stream of them.

```
partialQuotients c
[3,7,15,1,25,1,7,4]
Type: Stream Integer
```

By considering more and more of the fraction, you get the convergents. For example, the first convergent is $a(1)$, the second is $a(1) + 1/a(2)$ and so on.

```
convergents c
22 333 355 9208 9563 76149 314159
[3,--,---,---,----,----,-----,-----]
7 106 113 2931 3044 24239 100000
Type: Stream Fraction Integer
```

Since this is a finite continued fraction, the last convergent is the original rational number, in reduced form. The result of `approximants` is always an infinite stream, though it may just repeat the "last" value.

```
approximants c
22 333 355 9208 9563 76149 314159
[3,--,---,---,----,----,-----,-----]
7 106 113 2931 3044 24239 100000
Type: Stream Fraction Integer
```

Inverting `c` only changes the partial quotients of its fraction by inserting a 0 at the beginning of the list.

```
pq := partialQuotients(1/c)
[0,3,7,15,1,25,1,7,4]
Type: Stream Integer
```

Do this to recover the original continued fraction from this list of

partial quotients. The three-argument form of the `continuedFraction` operation takes an element which is the whole part of the fraction, a stream of elements which are the numerators of the fraction, and a stream of elements which are the denominators of the fraction.

```
continuedFraction(first pq, repeating [1], rest pq)
  1 |    1 |    1 |    1 |    1 |    1 |    1 |    1 |
+---+ + +---+ + +---+ + +---+ + +---+ + +---+ + +---+ + +---+
| 3      | 7      | 15      | 1      | 25      | 1      | 7      | 4
Type: ContinuedFraction Integer
```

The streams need not be finite for `continuedFraction`. Can you guess which irrational number has the following continued fraction? See the end of this section for the answer.

```
z:=continuedFraction(3, repeating [1], repeating [3,6])
  1 |    1 |    1 |    1 |    1 |    1 |    1 |    1 |    1 |
3 + +---+ + +---+ + +---+ + +---+ + +---+ + +---+ + +---+ + +---+
  | 3      | 6      | 3      | 6      | 3      | 6      | 3      | 6      | 3
+
  1 |
+---+ + ...
  | 6
Type: ContinuedFraction Integer
```

In 1737 Euler discovered the infinite continued fraction expansion

$$e - 1 = \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{6 + \cfrac{1}{10 + \cfrac{1}{14 + \dots}}}}}$$

We use this expansion to compute rational and floating point approximations of e . For this and other interesting expansions, see C. D. Olds, *Continued Fractions*, New Mathematical Library, (New York: Random House, 1963), pp. 134--139.}

By looking at the above expansion, we see that the whole part is 0 and the numerators are all equal to 1. This constructs the stream of denominators.

```
dens:Stream Integer := cons(1, generate((x+>x+4),6))
[1,6,10,14,18,22,26,30,34,38,...]
Type: Stream Integer
```


$$\begin{array}{c}
 \text{-----} \\
 2 + \quad 25 \\
 \text{-----} \\
 2 + \quad 49 \\
 \text{-----} \\
 2 + \quad 81 \\
 \text{-----} \\
 2 + \dots
 \end{array}$$

Let's use this expansion to compute rational and floating point approximations for pi.

```

cf := continuedFraction(1,[(2*i+1)**2 for i in 0..],repeating [2])
      1 |   9 |   25 |   49 |   81 |  121 |  169 |  225 |
1 + +---+ + +---+ + +---+ + +---+ + +---+ + +---+ + +---+
   | 2   | 2   | 2   | 2   | 2   | 2   | 2   | 2   |
+
  289 |   361 |
+-----+ + +-----+ + ...
   | 2   | 2
Type: ContinuedFraction Integer

```

```

ccf := convergents cf
      3 15 105 315 3465 45045 45045 765765 14549535
[1,-,--,---,---,---,---,---,---,---,---,---,---,...]
      2 13 76 263 2578 36979 33976 622637 11064338
Type: Stream Fraction Integer

piConvergents := [4/p for p in ccf]
      8 52 304 1052 10312 147916 135904 2490548 44257352
[4,-,--,---,---,---,---,---,---,---,---,---,---,...]
      3 15 105 315 3465 45045 45045 765765 14549535
Type: Stream Fraction Integer

```

As you can see, the values are converging to
 $\pi = 3.14159265358979323846\dots$, but not very quickly.

```

piConvergents :: Stream Float
[4.0, 2.6666666666 666666667, 3.4666666666 666666667,
 2.8952380952 380952381, 3.3396825396 825396825, 2.9760461760 461760462,
 3.2837384837 384837385, 3.0170718170 718170718, 3.2523659347 188758953,
 3.0418396189 294022111, ...]
Type: Stream Float

```

You need not restrict yourself to continued fractions of integers.
 Here is an expansion for a quotient of Gaussian integers.

```

continuedFraction((- 122 + 597*i)/(4 - 4*i))
      1 |   1 |
- 90 + 59*i + +-----+ + +-----+

```

```

      | 1 - 2%i      | - 1 + 2%i
      Type: ContinuedFraction Complex Integer

```

This is an expansion for a quotient of polynomials in one variable with rational number coefficients.

```

r : Fraction UnivariatePolynomial(x,Fraction Integer)
  Type: Void

r := ((x - 1) * (x - 2)) / ((x-3) * (x-4))
      2
      x  - 3x + 2
      -----
      2
      x  - 7x + 12
      Type: Fraction UnivariatePolynomial(x,Fraction Integer)

continuedFraction r
      1      |      1      |
1 + +-----+ + +-----+
   | 1      9      | 16      40
   | - x - -      | -- x - --
   | 4      8      | 3      3
      Type: ContinuedFraction UnivariatePolynomial(x,Fraction Integer)

```

To conclude this section, we give you evidence that

```

z = 3 +
      1
      -----
      3 +
            1
            -----
            6 +
                  1
                  -----
                  3 +
                        1
                        -----
                        6 +
                              1
                              -----
                              3 + ...

```

is the expansion of `sqrt(11)`.

```

[i*i for i in convergents(z) :: Stream Float]
[9.0, 11.111111111 11111111, 10.9944598337 9501385, 11.0002777777 77777778,
 10.9999860763 98799786, 11.0000006979 29731039, 10.9999999650 15834446,
 11.0000000017 53603304, 10.9999999999 12099531, 11.0000000000 04406066,
 ...]
      Type: Stream Float

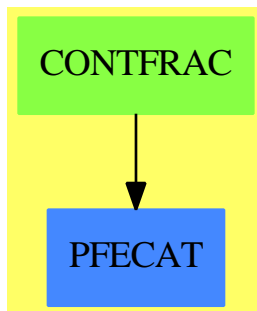
```

See Also:

o)help Stream

```
o )show ContinuedFraction
```

ContinuedFraction (CONTFRAC)



Exports:

0	1	approximants
associates?	characteristic	coerce
complete	continuedFraction	convergents
denominators	divide	extend
euclideanSize	expressIdealMember	exquo
extendedEuclidean	factor	gcd
gcdPolynomial	hash	inv
latex	lcm	multiEuclidean
numerators	one?	partialDenominators
partialNumerators	partialQuotients	prime?
principalIdeal	recip	reducedContinuedFraction
reducedForm	sample	sizeLess?
squareFree	squareFreePart	subtractIfCan
unit?	unitCanonical	unitNormal
wholePart	zero?	?*?
?**?	?+?	?-?
-?	?/?	?=?
?^?	?~=?	?quo?
?rem?		

— domain CONTFRAC ContinuedFraction —

```

)abbrev domain CONTFRAC ContinuedFraction
++ Author: Stephen M. Watt
++ Date Created: January 1987
  
```



```

++ Change History: 7 October 1991
++ Description:
++ \spadtype{ContinuedFraction} implements general
++ continued fractions. This version is not restricted to simple,
++ finite fractions and uses the \spadtype{Stream} as a
++ representation. The arithmetic functions assume that the
++ approximants alternate below/above the convergence point.
++ This is enforced by ensuring the partial numerators and partial
++ denominators are greater than 0 in the Euclidean domain view of \spad{R}
++ (i.e. \spad{sizeLess?(0, x)}).

```

```
ContinuedFraction(R): Exports == Implementation where
```

```

R :      EuclideanDomain
Q  ==> Fraction R
MT ==> MoebiusTransform Q
OUT ==> OutputForm

```

```
Exports ==> Join(Algebra R, Algebra Q, Field) with
```

```

continuedFraction:      Q -> %
++ continuedFraction(r) converts the fraction \spadvar{r} with
++ components of type \spad{R} to a continued fraction over
++ \spad{R}.

```

```

continuedFraction:      (R, Stream R, Stream R) -> %
++ continuedFraction(b0,a,b) constructs a continued fraction in
++ the following way: if \spad{a = [a1,a2,...]} and \spad{b =
++ [b1,b2,...]} then the result is the continued fraction
++ \spad{b0 + a1/(b1 + a2/(b2 + ...))}.

```

```

reducedContinuedFraction: (R, Stream R) -> %
++ reducedContinuedFraction(b0,b) constructs a continued
++ fraction in the following way: if \spad{b = [b1,b2,...]}
++ then the result is the continued fraction \spad{b0 + 1/(b1 +
++ 1/(b2 + ...))}. That is, the result is the same as
++ \spad{continuedFraction(b0,[1,1,1,...],[b1,b2,b3,...])}.

```

```

partialNumerators:      % -> Stream R
++ partialNumerators(x) extracts the numerators in \spadvar{x}.
++ That is, if \spad{x = continuedFraction(b0, [a1,a2,a3,...],
++ [b1,b2,b3,...])}, then \spad{partialNumerators(x) =
++ [a1,a2,a3,...]}.

```

```

partialDenominators: % -> Stream R
++ partialDenominators(x) extracts the denominators in
++ \spadvar{x}. That is, if \spad{x = continuedFraction(b0,
++ [a1,a2,a3,...], [b1,b2,b3,...])}, then
++ \spad{partialDenominators(x) = [b1,b2,b3,...]}.

```

```

partialQuotients:      % -> Stream R
++ partialQuotients(x) extracts the partial quotients in

```

```

++ \spadvar{x}. That is, if \spad{x = continuedFraction(b0,
++ [a1,a2,a3,...], [b1,b2,b3,...])}, then
++ \spad{partialQuotients(x) = [b0,b1,b2,b3,...]}.

wholePart:          % -> R
++ wholePart(x) extracts the whole part of \spadvar{x}. That
++ is, if \spad{x = continuedFraction(b0, [a1,a2,a3,...],
++ [b1,b2,b3,...])}, then \spad{wholePart(x) = b0}.

reducedForm:        % -> %
++ reducedForm(x) puts the continued fraction \spadvar{x} in
++ reduced form, i.e. the function returns an equivalent
++ continued fraction of the form
++ \spad{continuedFraction(b0,[1,1,1,...],[b1,b2,b3,...])}.

approximants:       % -> Stream Q
++ approximants(x) returns the stream of approximants of the
++ continued fraction \spadvar{x}. If the continued fraction is
++ finite, then the stream will be infinite and periodic with
++ period 1.

convergents:        % -> Stream Q
++ convergents(x) returns the stream of the convergents of the
++ continued fraction \spadvar{x}. If the continued fraction is
++ finite, then the stream will be finite.

numerators:         % -> Stream R
++ numerators(x) returns the stream of numerators of the
++ approximants of the continued fraction \spadvar{x}. If the
++ continued fraction is finite, then the stream will be finite.

denominators:       % -> Stream R
++ denominators(x) returns the stream of denominators of the
++ approximants of the continued fraction \spadvar{x}. If the
++ continued fraction is finite, then the stream will be finite.

extend:             (%,Integer) -> %
++ extend(x,n) causes the first \spadvar{n} entries in the
++ continued fraction \spadvar{x} to be computed. Normally
++ entries are only computed as needed.

complete:           % -> %
++ complete(x) causes all entries in \spadvar{x} to be computed.
++ Normally entries are only computed as needed. If \spadvar{x}
++ is an infinite continued fraction, a user-initiated interrupt is
++ necessary to stop the computation.

Implementation ==> add

-- isOrdered ==> R is Integer

```

```

isOrdered ==> R has OrderedRing and R has multiplicativeValuation
canReduce? ==> isOrdered or R has additiveValuation

Rec ==> Record(num: R, den: R)
Str ==> Stream Rec
Rep := Record(value: Record(whole: R, fract: Str), reduced?: Boolean)

import Str

genFromSequence:      Stream Q -> %
genReducedForm:        (Q, Stream Q, MT) -> Stream Rec
genFractionA:          (Stream R, Stream R) -> Stream Rec
genFractionB:          (Stream R, Stream R) -> Stream Rec
genNumDen:             (R, R, Stream Rec) -> Stream R

genApproximants:      (R, R, R, R, Stream Rec) -> Stream Q
genConvergents:        (R, R, R, R, Stream Rec) -> Stream Q
iGenApproximants:      (R, R, R, R, Stream Rec) -> Stream Q
iGenConvergents:       (R, R, R, R, Stream Rec) -> Stream Q

reducedForm c ==
  c.reduced? => c
  explicitlyFinite? c.value.fract =>
    continuedFraction last complete convergents c
  canReduce? => genFromSequence approximants c
  error "Reduced form not defined for this continued fraction."

eucWhole(a: Q): R == numer a quo denom a

eucWhole0(a: Q): R ==
  isOrdered =>
    n := numer a
    d := denom a
    q := n quo d
    r := n - q*d
    if r < 0 then q := q - 1
    q
  eucWhole a

x = y ==
  x := reducedForm x
  y := reducedForm y

x.value.whole ^= y.value.whole => false

x1 := x.value.fract; y1 := y.value.fract

while not empty? x1 and not empty? y1 repeat
  frst.x1.den ^= frst.y1.den => return false
  x1 := rst x1; y1 := rst y1

```

```

empty? x1 and empty? y1

continuedFraction q == q :: %

if isOrdered then
  continuedFraction(wh,nums,dens) == [[wh,genFractionA(nums,dens)],false]

  genFractionA(nums,dens) ==
    empty? nums or empty? dens => empty()
    n := first nums
    d := first dens
    n < 0 => error "Numerators must be greater than 0."
    d < 0 => error "Denominators must be greater than 0."
    concat([n,d]$Rec, delay genFractionA(rst nums,rst dens))
else
  continuedFraction(wh,nums,dens) == [[wh,genFractionB(nums,dens)],false]

  genFractionB(nums,dens) ==
    empty? nums or empty? dens => empty()
    n := first nums
    d := first dens
    concat([n,d]$Rec, delay genFractionB(rst nums,rst dens))

reducedContinuedFraction(wh,dens) ==
  continuedFraction(wh, repeating [1], dens)

coerce(n:Integer):% == [[n::R,empty()], true]
coerce(r:R):% == [[r, empty()], true]

coerce(a: Q): % ==
  wh := eucWhole0 a
  fr := a - wh::Q
  zero? fr => [[wh, empty()], true]

l : List Rec := empty()
n := numer fr
d := denom fr
while not zero? d repeat
  qr := divide(n,d)
  l := concat([1,qr.quotient],l)
  n := d
  d := qr.remainder
[[wh, construct rest reverse! l], true]

characteristic() == characteristic()$Q

genFromSequence apps ==
  lo := first apps; apps := rst apps
  hi := first apps; apps := rst apps

```

```

while eucWhole0 lo ^= eucWhole0 hi repeat
  lo := first apps; apps := rst apps
  hi := first apps; apps := rst apps
wh := eucWhole0 lo
[[wh, genReducedForm(wh::Q, apps, moebius(1,0,0,1))], canReduce?]

genReducedForm(wh0, apps, mt) ==
  lo: Q := first apps - wh0; apps := rst apps
  hi: Q := first apps - wh0; apps := rst apps
  lo = hi and zero? eval(mt, lo) => empty()
  mt := recip mt
  wlo := eucWhole eval(mt, lo)
  whi := eucWhole eval(mt, hi)
  while wlo ^= whi repeat
    wlo := eucWhole eval(mt, first apps - wh0); apps := rst apps
    whi := eucWhole eval(mt, first apps - wh0); apps := rst apps
  concat([1,wlo], delay genReducedForm(wh0, apps, shift(mt, -wlo::Q)))

wholePart c ==
  c.value.whole
partialNumerators c ==
  map(x1+>x1.num, c.value.fract)$StreamFunctions2(Rec,R)
partialDenominators c ==
  map(x1+>x1.den, c.value.fract)$StreamFunctions2(Rec,R)
partialQuotients c ==
  concat(c.value.whole, partialDenominators c)

approximants c ==
  empty? c.value.fract => repeating [c.value.whole::Q]
  genApproximants(1,0,c.value.whole,1,c.value.fract)
convergents c ==
  empty? c.value.fract => concat(c.value.whole::Q, empty())
  genConvergents (1,0,c.value.whole,1,c.value.fract)
numerators c ==
  empty? c.value.fract => concat(c.value.whole, empty())
  genNumDen(1,c.value.whole,c.value.fract)
denominators c ==
  genNumDen(0,1,c.value.fract)

extend(x,n) == (extend(x.value.fract,n); x)
complete(x) == (complete(x.value.fract); x)

iGenApproximants(pm2,qm2,pm1,qm1,fr) == delay
  nd := frst fr
  pm := nd.num*pm2 + nd.den*pm1
  qm := nd.num*qm2 + nd.den*qm1
  genApproximants(pm1,qm1,pm,qm,rst fr)

genApproximants(pm2,qm2,pm1,qm1,fr) ==
  empty? fr => repeating [pm1/qm1]

```

```

concat(pm1/qm1,iGenApproximants(pm2,qm2,pm1,qm1,fr))

iGenConvergents(pm2,qm2,pm1,qm1,fr) == delay
  nd := frst fr
  pm := nd.num*pm2 + nd.den*pm1
  qm := nd.num*qm2 + nd.den*qm1
  genConvergents(pm1,qm1,pm,qm,rst fr)

genConvergents(pm2,qm2,pm1,qm1,fr) ==
  empty? fr => concat(pm1/qm1, empty())
  concat(pm1/qm1,iGenConvergents(pm2,qm2,pm1,qm1,fr))

genNumDen(m2,m1,fr) ==
  empty? fr => concat(m1,empty())
  concat(m1,delay genNumDen(m1,m2*frst(fr).num + m1*frst(fr).den,rst fr))

gen ==> genFromSequence
apx ==> approximants

c, d: %
a: R
q: Q
n: Integer

0 == (0$R) :: %
1 == (1$R) :: %

c + d == genFromSequence map((x,y) +-> x + y, apx c, apx d)
c - d == genFromSequence map((x,y) +-> x - y, apx c, rest apx d)
- c == genFromSequence map(x +-> - x, rest apx c)
c * d == genFromSequence map((x,y) +-> x * y, apx c, apx d)
a * d == genFromSequence map(x +-> a * x, apx d)
q * d == genFromSequence map(x +-> q * x, apx d)
n * d == genFromSequence map(x +-> n * x, apx d)
c / d == genFromSequence map((x,y) +-> x / y, apx c, rest apx d)
recip c == (c = 0 => "failed";
  genFromSequence map(x +-> 1/x, rest apx c))

showAll?: () -> Boolean
showAll?() ==
  NULL(_$streamsShowAll$Lisp)$Lisp => false
  true

zagRec(t:Rec):OUT == zag(t.num :: OUT,t.den :: OUT)

coerce(c:%): OUT ==
  wh := c.value.whole
  fr := c.value.fract
  empty? fr => wh :: OUT
  count : NonNegativeInteger := _$streamCount$Lisp

```

```

l : List OUT := empty()
for n in 1..count while not empty? fr repeat
  l := concat(zagRec frst fr,l)
  fr := rst fr
if showAll?() then
  for n in (count + 1).. while explicitEntries? fr repeat
    l := concat(zagRec frst fr,l)
    fr := rst fr
if not explicitlyEmpty? fr then l := concat("..." :: OUT,l)
l := reverse! l
e := reduce("+",l)
zero? wh => e
(wh :: OUT) + e

```

— CONTFRAC.dotabb —

```

"CONTRAC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=CONTRAC"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"CONTRAC" -> "PFECAT"

```

Chapter 5

Chapter D

domain DBASE Database

— Database.input —

```
)set break resume
)sys rm -f Database.output
)spool Database.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Database
--R
--R Database(S) where
--R   S: OrderedSet with
--R     ?? : (%,Symbol) -> String
--R     display : % -> Void
--R     fullDisplay : % -> Void  is a domain constructor
--R Abbreviation for Database is DBASE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for DBASE
--R
--R----- Operations -----
--R ?? : (%,%) -> %           ?-? : (%,%) -> %
--R ?? : (%,%) -> Boolean    coerce : List(S) -> %
--R coerce : % -> OutputForm  display : % -> Void
--R ?? : (%,QueryEquation) -> %  fullDisplay : % -> Void
--R hash : % -> SingleInteger    latex : % -> String
--R ?~=? : (%,%) -> Boolean
--R ?? : (%,Symbol) -> DataList(String)
```



```

--R fullDisplay : (%,PositiveInteger,PositiveInteger) -> Void
--R
--E 1

)spool
)lisp (bye)

```

— Database.help —

=====

Database examples

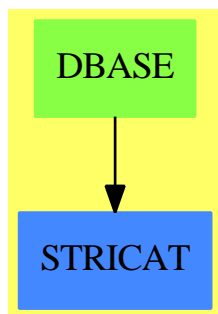
=====

This domain implements a simple view of a database whose fields are indexed by symbols

See Also:

- o)show Database

Database (DBASE)



See

⇒ “DataList” (DLIST) [5](#) on page [459](#)

⇒ “IndexCard” (ICARD) [10](#) on page [1193](#)

⇒ “QueryEquation” (QEQUAT) [18](#) on page [2169](#)

Exports:

coerce	display	fullDisplay	hash	latex
?+?	?-?	?=?	?~=?	?..?

— domain DBASE Database —

```

)abbrev domain DBASE Database
++ Author: Mark Botch
++ Description:
++ This domain implements a simple view of a database whose fields are
++ indexed by symbols

Database(S): Exports == Implementation where
  S: OrderedSet with
    elt: (%,Symbol) -> String
      ++ elt(x,s) returns an element of x indexed by s
    display: % -> Void
      ++ display(x) displays x in some form
    fullDisplay: % -> Void
      ++ fullDisplay(x) displays x in detail
  Exports == SetCategory with
    elt: (%,QueryEquation) -> %
      ++ elt(db,q) returns all elements of \axiom{db} which satisfy \axiom{q}.
    elt: (%,Symbol) -> DataList String
      ++ elt(db,s) returns the \axiom{s} field of each element of \axiom{db}.
    _+: (%,%) -> %
      ++ db1+db2 returns the merge of databases db1 and db2
    _-: (%,%) -> %
      ++ db1-db2 returns the difference of databases db1 and db2 i.e. consisting
      ++ of elements in db1 but not in db2
    coerce: List S -> %
      ++ coerce(l) makes a database out of a list
    display: % -> Void
      ++ display(db) prints a summary line for each entry in \axiom{db}.
    fullDisplay: % -> Void
      ++ fullDisplay(db) prints full details of each entry in \axiom{db}.
    fullDisplay: (%,PositiveInteger,PositiveInteger) -> Void
      ++ fullDisplay(db,start,end ) prints full details of entries in the range
      ++ \axiom{start..end} in \axiom{db}.
  Implementation == List S add
    s: Symbol
    Rep := List S
    coerce(u: List S):% == u@%
    elt(data: %,s: Symbol) == [x.s for x in data] :: DataList(String)
    elt(data: %,eq: QueryEquation) ==
      field := variable eq
      val := value eq
      [x for x in data | stringMatches?(val,x.field)$Lisp]
    x+y==removeDuplicates_! merge(x,y)
    x-y==mergeDifference(copy(x::Rep),y::Rep)$MergeThing(S)
    coerce(data): OutputForm == (#data):: OutputForm
    display(data) == for x in data repeat display x
    fullDisplay(data) == for x in data repeat fullDisplay x
    fullDisplay(data,n,m) == for x in data for i in 1..m repeat
      if i >= n then fullDisplay x

```

— DBASE.dotabb —

```
"DBASE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DBASE"]
"STRICAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=STRICAT"]
"DBASE" -> "STRICAT"
```

domain DLIST DataList

— DataList.input —

```
)set break resume
)sys rm -f DataList.output
)spool DataList.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DataList
--R
--R DataList(S: OrderedSet) is a domain constructor
--R Abbreviation for DataList is DLIST
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for DLIST
--R
--R----- Operations -----
--R children : % -> List(%)
--R coerce : List(S) -> %
--R concat : List(%) -> %
--R concat : (%,% ) -> %
--R concat! : (%,% ) -> %
--R copy : % -> %
--R cycleTail : % -> %
--R datalist : List(S) -> %
--R delete! : (%,Integer) -> %
--R ?.sort : (% ,sort) -> %
--R elt : (% ,Integer,S) -> S
--R ?.last : (% ,last) -> S
--R ?.first : (% ,first) -> S
--R empty : () -> %
--R entries : % -> List(S)
--R coerce : % -> List(S)
--R concat : (% ,S) -> %
--R concat : (S,% ) -> %
--R concat! : (% ,S) -> %
--R construct : List(S) -> %
--R cycleEntry : % -> %
--R cyclic? : % -> Boolean
--R delete : (% ,Integer) -> %
--R distance : (% ,%) -> Integer
--R ?.unique : (% ,unique) -> %
--R ?.? : (% ,Integer) -> S
--R ?.rest : (% ,rest) -> %
--R ?.value : (% ,value) -> S
--R empty? : % -> Boolean
--R eq? : (% ,%) -> Boolean
```

```

--R explicitlyFinite? : % -> Boolean
--R index? : (Integer,%) -> Boolean
--R insert : (S,%,Integer) -> %
--R insert! : (S,%,Integer) -> %
--R last : % -> S
--R leaves : % -> List(S)
--R map : (((S,S) -> S),%,%) -> %
--R new : (NonNegativeInteger,S) -> %
--R possiblyInfinite? : % -> Boolean
--R rest : % -> %
--R sample : () -> %
--R tail : % -> %
--R value : % -> S
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?<? : (%,%) -> Boolean if S has ORDSET
--R ?<=? : (%,%) -> Boolean if S has ORDSET
--R ?=? : (%,%) -> Boolean if S has SETCAT
--R ?>? : (%,%) -> Boolean if S has ORDSET
--R ?>=? : (%,%) -> Boolean if S has ORDSET
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R child? : (%,%) -> Boolean if S has SETCAT
--R coerce : % -> OutputForm if S has SETCAT
--R convert : % -> InputForm if S has KONVERT(INFORM)
--R copyInto! : (%,%,Integer) -> % if $ has shallowlyMutable
--R count : (S,%) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R cycleLength : % -> NonNegativeInteger
--R cycleSplit! : % -> % if $ has shallowlyMutable
--R delete : (%,UniversalSegment(Integer)) -> %
--R delete! : (%,UniversalSegment(Integer)) -> %
--R ?.count : (%,count) -> NonNegativeInteger
--R ?? : (%,UniversalSegment(Integer)) -> %
--R entry? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R eval : (%,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (%,S) -> % if $ has shallowlyMutable
--R find : ((S -> Boolean),%) -> Union(S,"failed")
--R first : (%,NonNegativeInteger) -> %
--R hash : % -> SingleInteger if S has SETCAT
--R last : (%,NonNegativeInteger) -> %
--R latex : % -> String if S has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R max : (%,%) -> % if S has ORDSET
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R members : % -> List(S) if $ has finiteAggregate
--R first : % -> S
--R indices : % -> List(Integer)
--R insert : (%,%,Integer) -> %
--R insert! : (%,%,Integer) -> %
--R leaf? : % -> Boolean
--R list : S -> %
--R map : ((S -> S),%) -> %
--R nodes : % -> List(%)
--R qelt : (%,Integer) -> S
--R reverse : % -> %
--R second : % -> S
--R third : % -> S

```

```

--R merge : ((S,S) -> Boolean),%,%) -> %
--R merge : (%,%) -> % if S has ORDSET
--R merge! : ((S,S) -> Boolean),%,%) -> %
--R merge! : (%,%) -> % if S has ORDSET
--R min : (%,%) -> % if S has ORDSET
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (%,NonNegativeInteger) -> Boolean
--R node? : (%,%) -> Boolean if S has SETCAT
--R parts : % -> List(S) if $ has finiteAggregate
--R position : ((S -> Boolean),%) -> Integer
--R position : (S,%) -> Integer if S has SETCAT
--R position : (S,%,Integer) -> Integer if S has SETCAT
--R qsetelt! : (%,Integer,S) -> S if $ has shallowlyMutable
--R reduce : ((S,S) -> S),%,S,S) -> S if $ has finiteAggregate and S has SETCAT
--R reduce : ((S,S) -> S),%,S) -> S if $ has finiteAggregate
--R reduce : ((S,S) -> S),%) -> S if $ has finiteAggregate
--R remove : (S,%) -> % if $ has finiteAggregate and S has SETCAT
--R remove : ((S -> Boolean),%) -> % if $ has finiteAggregate
--R remove! : ((S -> Boolean),%) -> %
--R remove! : (S,%) -> % if S has SETCAT
--R removeDuplicates : % -> % if $ has finiteAggregate and S has SETCAT
--R removeDuplicates! : % -> % if S has SETCAT
--R rest : (%,NonNegativeInteger) -> %
--R reverse! : % -> % if $ has shallowlyMutable
--R select : ((S -> Boolean),%) -> % if $ has finiteAggregate
--R select! : ((S -> Boolean),%) -> %
--R setchildren! : (%,List(%)) -> % if $ has shallowlyMutable
--R setelt : (%,Integer,S) -> S if $ has shallowlyMutable
--R setelt : (%,UniversalSegment(Integer),S) -> S if $ has shallowlyMutable
--R setelt : (%,last,S) -> S if $ has shallowlyMutable
--R setelt : (%,rest,%) -> % if $ has shallowlyMutable
--R setelt : (%,first,S) -> S if $ has shallowlyMutable
--R setelt : (%,value,S) -> S if $ has shallowlyMutable
--R setfirst! : (%,S) -> S if $ has shallowlyMutable
--R setlast! : (%,S) -> S if $ has shallowlyMutable
--R setrest! : (%,%) -> % if $ has shallowlyMutable
--R setvalue! : (%,S) -> S if $ has shallowlyMutable
--R size? : (%,NonNegativeInteger) -> Boolean
--R sort : ((S,S) -> Boolean),%) -> %
--R sort : % -> % if S has ORDSET
--R sort! : ((S,S) -> Boolean),%) -> % if $ has shallowlyMutable
--R sort! : % -> % if $ has shallowlyMutable and S has ORDSET
--R sorted? : ((S,S) -> Boolean),%) -> Boolean
--R sorted? : % -> Boolean if S has ORDSET
--R split! : (%,Integer) -> % if $ has shallowlyMutable
--R swap! : (%,Integer,Integer) -> Void if $ has shallowlyMutable
--R ?~=?: (%,%) -> Boolean if S has SETCAT
--R
--E 1

```

```
)spool  
)lisp (bye)
```

— **DataList.help** —

=====

DataList examples

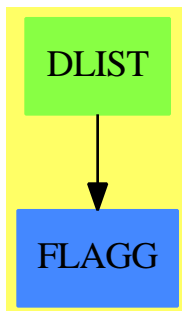
=====

This domain provides some nice functions on lists

See Also:

o)show DataList

DataList (DLIST)



See

- ⇒ “IndexCard” (ICARD) [10](#) on page [1193](#)
- ⇒ “Database” (DBASE) [5](#) on page [454](#)
- ⇒ “QueryEquation” (QEQUAT) [18](#) on page [2169](#)

Exports:

any?	child?	children	coerce
concat	concat!	construct	convert
copy	copyInto!	count	cycleEntry
cycleLength	cycleSplit!	cycleTail	cyclic?
datalist	delete	delete!	distance
elt	empty	empty?	entries
entry?	eq?	eval	every?
explicitlyFinite?	fill!	find	first
hash	index?	indices	insert
insert!	last	latex	leaf?
leaves	less?	list	map
map!	max	maxIndex	member?
members	merge	merge!	min
minIndex	more?	new	node?
nodes	parts	position	possiblyInfinite?
qelt	qsetelt!	reduce	remove
remove!	removeDuplicates	removeDuplicates!	rest
reverse	reverse!	sample	second
select	select!	setchildren!	setelt
setfirst!	setlast!	setrest!	setValue!
size?	sort	sort!	sorted?
split!	swap!	tail	third
value	#?	?<?	?<=?
?=?	?>?	?>=?	?.
?~=?	?..count	?..sort	?..unique
?..last	?..rest	?..first	?..value

— domain DLIST DataList —

```

)abbrev domain DLIST DataList
++ Author: Mark Botch
++ Description:
++ This domain provides some nice functions on lists

DataList(S:OrderedSet) : Exports == Implementation where
  Exports == ListAggregate(S) with
    coerce: List S -> %
      ++ coerce(l) creates a datalist from l
    coerce: % -> List S
      ++ coerce(x) returns the list of elements in x
    datalist: List S -> %
      ++ datalist(l) creates a datalist from l
    elt: (%,"unique") -> %
      ++ \axiom{l.unique} returns \axiom{l} with duplicates removed.
      ++ Note: \axiom{l.unique} = removeDuplicates(l)}.
    elt: (%,"sort") -> %
      ++ \axiom{l.sort} returns \axiom{l} with elements sorted.

```

```

    ++ Note: \axiom{l.sort = sort(l)}
    elt: (%,"count") -> NonNegativeInteger
    ++ \axiom{l."count"} returns the number of elements in \axiom{l}.
Implementation == List(S) add
  elt(x,"unique") == removeDuplicates(x)
  elt(x,"sort") == sort(x)
  elt(x,"count") == #x
  coerce(x:List S) == x pretend %
  coerce(x:%):List S == x pretend (List S)
  coerce(x:%): OutputForm == (x :: List S) :: OutputForm
  datalist(x:List S) == x::%

```

— DLIST.dotabb —

```

"DLIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DLIST"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"DLIST" -> "FLAGG"

```

domain DECIMAL DecimalExpansion

— DecimalExpansion.input —

```

)set break resume
)sys rm -f DecimalExpansion.output
)spool DecimalExpansion.output
)set message test on
)set message auto off
)clear all
--S 1 of 7
r := decimal(22/7)
--R
--R
--R
--R      (1)  3.142857
--R
--R                                          Type: DecimalExpansion
--E 1

--S 2 of 7
r + decimal(6/7)
--R

```



```

--R
--R (2) 4
--R
--R                                         Type: DecimalExpansion
--E 2

--S 3 of 7
[decimal(1/i) for i in 350..354]
--R
--R
--R (3)
--R
--R [0.00285714, 0.002849, 0.0028409, 0.00283286118980169971671388101983,
--R 0.00282485875706214689265536723163841807909604519774011299435]
--R
--R                                         Type: List(DecimalExpansion)
--E 3

--S 4 of 7
decimal(1/2049)
--R
--R
--R (4)
--R 0.
--R OVERBAR
--R 00048804294777940458760370912640312347486578818936066373840897999023914
--R 104441190824792581747193753050268423621278672523182040019521717911176
--R 183504148365056124938994631527574426549536359199609565641776476329917
--R 032698877501220107369448511469009272816007808687164470473401659346022
--R 449975597852611029770619814543679843826256710590531966813079551
--R
--R                                         Type: DecimalExpansion
--E 4

--S 5 of 7
p := decimal(1/4)*x**2 + decimal(2/3)*x + decimal(4/9)
--R
--R
--R
--R (5)  $0.25x^2 + 0.6x + 0.4$ 
--R
--R                                         Type: Polynomial(DecimalExpansion)
--E 5

--S 6 of 7
q := differentiate(p, x)
--R
--R
--R
--R (6)  $0.5x + 0.6$ 
--R
--R                                         Type: Polynomial(DecimalExpansion)
--E 6

```

```

--S 7 of 7
g := gcd(p, q)
--R
--R
--R
--R      (7)  x + 1.3
--R
--R                                          Type: Polynomial(DecimalExpansion)
--E 7
)spool
)lisp (bye)

```

— DecimalExpansion.help —

=====

DecimalExpansion examples

=====

All rationals have repeating decimal expansions. Operations to access the individual digits of a decimal expansion can be obtained by converting the value to RadixExpansion(10).

The operation decimal is used to create this expansion of type DecimalExpansion.

```

r := decimal(22/7)
-----
3.142857
Type: DecimalExpansion

```

Arithmetic is exact.

```

r + decimal(6/7)
4
Type: DecimalExpansion

```

The period of the expansion can be short or long ...

```

[decimal(1/i) for i in 350..354]
-----
[0.00285714, 0.002849, 0.0028409, 0.00283286118980169971671388101983,
-----
0.00282485875706214689265536723163841807909604519774011299435]
Type: List DecimalExpansion

```

or very long.

```

decimal(1/2049)

```

```

-----
0.00048804294777940458760370912640312347486578818936066373840897999023914
-----
104441190824792581747193753050268423621278672523182040019521717911176
-----
183504148365056124938994631527574426549536359199609565641776476329917
-----
032698877501220107369448511469009272816007808687164470473401659346022
-----
449975597852611029770619814543679843826256710590531966813079551
Type: DecimalExpansion

```

These numbers are bona fide algebraic objects.

```

p := decimal(1/4)*x**2 + decimal(2/3)*x + decimal(4/9)
      2      -      -
0.25x  + 0.6x + 0.4
Type: Polynomial DecimalExpansion

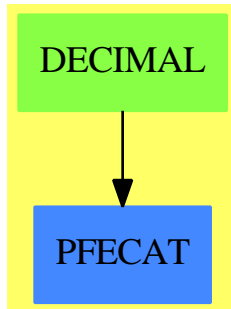
q := differentiate(p, x)
      -
0.5x + 0.6
Type: Polynomial DecimalExpansion

g := gcd(p, q)
      -
x + 1.3
Type: Polynomial DecimalExpansion

```

See Also:

- o)help RadixExpansion
- o)help BinaryExpansion
- o)help HexadecimalExpansion
- o)show DecimalExpansion

DecimalExpansion (DECIMAL)**See**⇒ “RadixExpansion” (RADIX) [19](#) on page [2205](#)⇒ “BinaryExpansion” (BINARY) [3](#) on page [292](#)⇒ “HexadecimalExpansion” (HEXADEC) [9](#) on page [1142](#)**Exports:**

0	1	abs
associates?	ceiling	characteristic
charthRoot	coerce	conditionP
convert	D	decimal
denom	denominator	differentiate
divide	euclideanSize	eval
expressIdealMember	exquo	extendedEuclidean
factor	factorPolynomial	factorSquareFreePolynomial
floor	fractionPart	gcd
gcdPolynomial	hash	init
inv	latex	lcm
map	max	min
multiEuclidean	negative?	nextItem
numer	numerator	one?
patternMatch	positive?	prime?
principalIdeal	random	recip
reducedSystem	retract	retractIfCan
sample	sign	sizeLess?
solveLinearPolynomialEquation	squareFree	squareFreePart
squareFreePolynomial	subtractIfCan	unit?
unitCanonical	unitNormal	wholePart
zero?	?*?	?**?
?+?	?-?	-?
?/?	?=?	?^?
?~=?	?<?	?<=?
?>?	?>=?	?..?
?quo?	?rem?	

— domain DECIMAL DecimalExpansion —

```

)abbrev domain DECIMAL DecimalExpansion
++ Author: Stephen M. Watt
++ Date Created: October, 1986
++ Date Last Updated: May 15, 1991
++ Description:
++ This domain allows rational numbers to be presented as repeating
++ decimal expansions.

DecimalExpansion(): Exports == Implementation where
  Exports ==> QuotientFieldCategory(Integer) with
    coerce: % -> Fraction Integer
      ++ coerce(d) converts a decimal expansion to a rational number.
    coerce: % -> RadixExpansion(10)
      ++ coerce(d) converts a decimal expansion to a radix expansion
      ++ with base 10.
    fractionPart: % -> Fraction Integer
      ++ fractionPart(d) returns the fractional part of a decimal expansion.
    decimal: Fraction Integer -> %
      ++ decimal(r) converts a rational number to a decimal expansion.

Implementation ==> RadixExpansion(10) add
  decimal r == r :: %
  coerce(x:%): RadixExpansion(10) == x pretend RadixExpansion(10)

```

— DECIMAL.dotabb —

```

"DECIMAL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DECIMAL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"DECIMAL" -> "PFECAT"

```

Denavit-Hartenberg Matrices

Homogeneous Transformations

The study of robot manipulation is concerned with the relationship between objects, and between objects and manipulators. In this chapter we will develop the representation necessary to describe these relationships. Similar problems of representation have already been solved in the field of computer graphics, where the relationship between objects must also be

described. Homogeneous transformations are used in this field and in computer vision [Duda] [Roberts63] [Roberts65]. These transformations were employed by Denavit to describe linkages [Denavit] and are now used to describe manipulators [Pieper] [Paul72] [Paul77b].

We will first establish notation for vectors and planes and then introduce transformations on them. These transformations consist primarily of translation and rotation. We will then show that these transformations can also be considered as coordinate frames in which to represent objects, including the manipulator. The inverse transformation will then be introduced. A later section describes the general rotation transformation representing a rotation about a vector. An algorithm is then described to find the equivalent axis and angle of rotations represented by any given transformation. A brief section on stretching and scaling transforms is included together with a section on the perspective transformation. The chapter concludes with a section on transformation equations.

Notation

In describing the relationship between objects we will make use of point vectors, planes, and coordinate frames. Point vectors are denoted by lower case, bold face characters. Planes are denoted by script characters, and coordinate frames by upper case, bold face characters. For example:

vectors	$\mathbf{v}, \mathbf{x1}, \mathbf{x}$
planes	\mathcal{P}, \mathcal{Q}
coordinate frames	$\mathbf{I}, \mathbf{A}, \mathbf{CONV}$

We will use point vectors, planes, and coordinate frames as variables which have associated values. For example, a point vector has as value its three Cartesian coordinate components.

If we wish to describe a point in space, which we will call p , with respect to a coordinate frame \mathbf{E} , we will use a vector which we will call \mathbf{v} . We will write this as

$${}^E\mathbf{v}$$

The leading superscript describes the defining coordinate frame.

We might also wish to describe this same point, p , with respect to a different coordinate frame, for example \mathbf{H} , using a vector \mathbf{w} as

$${}^H\mathbf{w}$$

\mathbf{v} and \mathbf{w} are two vectors which probably have different component values and $\mathbf{v} \neq \mathbf{w}$ even though both vectors describe the same point p . The case might also exist of a vector \mathbf{a} describing a point 3 inches above any frame

$${}^{F^1}\mathbf{a} \quad {}^{F^2}\mathbf{a}$$

In this case the vectors are identical but describe different points. Frequently, the defining frame will be obvious from the text and the superscripts will be left off. In many cases the

name of the vector will be the same as the name of the object described, for example, the tip of a pin might be described by a vector **tip** with respect to a frame **BASE** as

$${}^{BASE}\mathbf{tip}$$

If it were obvious from the text that we were describing the vector with respect to **BASE** then we might simply write

$$\mathbf{tip}$$

If we also wish to describe this point with respect to another coordinate frame say, **HAND**, then we must use another vector to describe this relationship, for example

$${}^{HAND}\mathbf{tv}$$

${}^{HAND}\mathbf{tv}$ and **tip** both describe the same feature but have different values. In order to refer to individual components of coordinate frames, point vectors, or planes, we add subscripts to indicate the particular component. For example, the vector ${}^{HAND}\mathbf{tv}$ has components ${}^{HAND}\mathbf{tv}_x$, ${}^{HAND}\mathbf{tv}_y$, ${}^{HAND}\mathbf{tv}_z$.

Vectors

The homogeneous coordinate representation of objects in n -space is an $(n + 1)$ -space entity such that a particular perspective projection recreates the n -space. This can also be viewed as the addition of an extra coordinate to each vector, a scale factor, such that the vector has the same meaning if each component, including the scale factor, is multiplied by a constant.

A point vector

$$\mathbf{v} = a\mathbf{i} + b\mathbf{j} + c\mathbf{k} \quad (1.1)$$

where **i**, **j**, and **k** are unit vectors along the x , y , and z coordinate axes, respectively, is represented in homogeneous coordinates as a column matrix

$$\mathbf{v} = \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{z} \\ \mathbf{w} \end{bmatrix} \quad (1.2)$$

where

$$\begin{aligned} \mathbf{a} &= \mathbf{x}/\mathbf{w} \\ \mathbf{b} &= \mathbf{y}/\mathbf{w} \\ \mathbf{c} &= \mathbf{z}/\mathbf{w} \end{aligned} \quad (1.3)$$

Thus the vector $3\mathbf{i} + 4\mathbf{j} + 5\mathbf{k}$ can be represented as $[3, 4, 5, 1]^T$ or as $[6, 8, 10, 2]^T$ or again as $[-30, -40, -50, -10]^T$, etc. The superscript T indicates the transpose of the row vector into a column vector. The vector at the origin, the null vector, is represented as $[0, 0, 0, n]^T$ where n is any non-zero scale factor. The vector $[0, 0, 0, 0]^T$ is undefined. Vectors of the form $[a, b, c, 0]^T$ represent vectors at infinity and are used to represent directions; the addition of any other finite vector does not change their value in any way.

We will also make use of the vector dot and cross products. Given two vectors

$$\begin{aligned}\mathbf{a} &= a_x\mathbf{i} + a_y\mathbf{j} + a_z\mathbf{k} \\ \mathbf{b} &= b_x\mathbf{i} + b_y\mathbf{j} + b_z\mathbf{k}\end{aligned}\tag{1.4}$$

we define the vector dot product, indicated by “.” as

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z\tag{1.5}$$

The dot product of two vectors is a scalar. The cross product, indicated by an “ \times ”, is another vector perpendicular to the plane formed by the vectors of the product and is defined by

$$\mathbf{a} \times \mathbf{b} = (a_y b_z - a_z b_y)\mathbf{i} + (a_z b_x - a_x b_z)\mathbf{j} + (a_x b_y - a_y b_x)\mathbf{k}\tag{1.6}$$

This definition is easily remembered as the expansion of the determinant

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}\tag{1.7}$$

Planes

A plane is represented as a row matrix

$$\mathcal{P} = [a, b, c, d]\tag{1.8}$$

such that if a point \mathbf{v} lies in a plane \mathcal{P} the matrix product

$$\mathcal{P} \mathbf{v} = 0\tag{1.9}$$

or in expanded form

$$xa + yb + zc + wd = 0\tag{1.10}$$

If we define a constant

$$m = +\sqrt{a^2 + b^2 + c^2}\tag{1.11}$$

and divide Equation 1.10 by wm we obtain

$$\frac{x}{w} \frac{a}{m} + \frac{y}{w} \frac{b}{m} + \frac{z}{w} \frac{c}{m} = -\frac{d}{m} \quad (1.12)$$

The left hand side of Equation 1.12 is the vector dot product of two vectors $(x/w)\mathbf{i} + (y/w)\mathbf{j} + (z/w)\mathbf{k}$ and $(a/m)\mathbf{i} + (b/m)\mathbf{j} + (c/m)\mathbf{k}$ and represents the directed distance of the point $(x/w)\mathbf{i} + (y/w)\mathbf{j} + (z/w)\mathbf{k}$ along the vector $(a/m)\mathbf{i} + (b/m)\mathbf{j} + (c/m)\mathbf{k}$. The vector $(a/m)\mathbf{i} + (b/m)\mathbf{j} + (c/m)\mathbf{k}$ can be interpreted as the outward pointing normal of a plane situated a distance $-d/m$ from the origin in the direction of the normal. Thus a plane \mathcal{P} parallel to the x,y plane, one unit along the z axis, is represented as

$$\mathcal{P} = [0, 0, 1, -1] \quad (1.13)$$

$$\text{or as } \mathcal{P} = [0, 0, 2, -2] \quad (1.14)$$

$$\text{or as } \mathcal{P} = [0, 0, -100, 100] \quad (1.15)$$

A point $\mathbf{v} = [10, 20, 1, 1]$ should lie in this plane

$$[0, 0, -100, 100] \begin{bmatrix} 10 \\ 20 \\ 1 \\ 1 \end{bmatrix} = 0 \quad (1.16)$$

or

$$[0, 0, 1, -1] \begin{bmatrix} -5 \\ -10 \\ -.5 \\ -.5 \end{bmatrix} = 0 \quad (1.17)$$

The point $\mathbf{v} = [0, 0, 2, 1]$ lies above the plane

$$[0, 0, 2, -2] \begin{bmatrix} 0 \\ 0 \\ 2 \\ 1 \end{bmatrix} = 2 \quad (1.18)$$

and $\mathcal{P} \mathbf{v}$ is indeed positive, indicating that the point is outside the plane in the direction of the outward pointing normal. A point $\mathbf{v} = [0, 0, 0, 1]$ lies below the plane

$$[0, 0, 1, -1] \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = -1 \quad (1.19)$$

The plane $[0, 0, 0, 0]$ is undefined.

Transformations

A transformation of the space \mathbf{H} is a 4x4 matrix and can represent translation, rotation, stretching, and perspective transformations. Given a point \mathbf{u} , its transformation \mathbf{v} is represented by the matrix product

$$\mathbf{v} = \mathbf{H}\mathbf{u} \quad (1.20)$$

The corresponding plane transformation \mathcal{P} to \mathcal{Q} is

$$\mathcal{Q} = \mathcal{P} \mathbf{H}^{-1} \quad (1.21)$$

as we require that the condition

$$\mathcal{Q} \mathbf{v} = \mathcal{P} \mathbf{u} \quad (1.22)$$

is invariant under all transformations. To verify this we substitute from Equations 1.20 and 1.21 into the left hand side of 1.22 and we obtain on the right hand side $\mathbf{H}^{-1}\mathbf{H}$ which is the identity matrix \mathbf{I}

$$\mathcal{P} \mathbf{H}^{-1} \mathbf{H} \mathbf{u} = \mathcal{P} \mathbf{u} \quad (1.23)$$

Translation Transformation

The transformation \mathbf{H} corresponding to a translation by a vector $a\mathbf{i} + b\mathbf{j} + c\mathbf{k}$ is

$$\mathbf{H} = \mathbf{Trans}(\mathbf{a}, \mathbf{b}, \mathbf{c}) = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.24)$$

Given a vector $\mathbf{u} = [x, y, z, w]^T$ the transformed vector \mathbf{v} is given by

$$\mathbf{H} = \mathbf{Trans}(\mathbf{a}, \mathbf{b}, \mathbf{c}) = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \quad (1.25)$$

$$\mathbf{v} = \begin{bmatrix} x + aw \\ y + bw \\ z + cw \\ w \end{bmatrix} = \begin{bmatrix} x/w + a \\ y/w + b \\ z/w + c \\ 1 \end{bmatrix} \quad (1.26)$$

The translation may also be interpreted as the addition of the two vectors $(x/w)\mathbf{i} + (y/w)\mathbf{j} + (z/w)\mathbf{k}$ and $a\mathbf{i} + b\mathbf{j} + c\mathbf{k}$.

Every element of a transformation matrix may be multiplied by a non-zero constant without changing the transformation, in the same manner as points and planes. Consider the vector $2\mathbf{i} + 3\mathbf{j} + 2\mathbf{k}$ translated by, or added to $4\mathbf{i} - 3\mathbf{j} + 7\mathbf{k}$

$$\begin{bmatrix} 6 \\ 0 \\ 9 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & -3 \\ 0 & 0 & 1 & 7 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 2 \\ 1 \end{bmatrix} \quad (1.27)$$

If we multiply the transformation matrix elements by, say, -5, and the vector elements by 2, we obtain

$$\begin{bmatrix} -60 \\ 0 \\ -90 \\ -10 \end{bmatrix} = \begin{bmatrix} -5 & 0 & 0 & -20 \\ 0 & -5 & 0 & 15 \\ 0 & 0 & -5 & -35 \\ 0 & 0 & 0 & -5 \end{bmatrix} \begin{bmatrix} 4 \\ 6 \\ 4 \\ 2 \end{bmatrix} \quad (1.28)$$

which corresponds to the vector $[6, 0, 9, 1]^T$ as before. The point $[2, 3, 2, 1]$ lies in the plane $[1, 0, 0, -2]$

$$[1, 0, 0, -2] \begin{bmatrix} 2 \\ 3 \\ 2 \\ 1 \end{bmatrix} = 0 \quad (1.29)$$

The transformed point is, as we have already found, $[6, 0, 9, 1]^T$. We will now compute the transformed plane. The inverse of the transform is

$$\begin{bmatrix} 1 & 0 & 0 & -4 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -7 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and the transformed plane

$$[1 \ 0 \ 0 \ -6] = [1 \ 0 \ 0 \ -2] \begin{bmatrix} 1 & 0 & 0 & -4 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -7 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.30)$$

Once again the transformed point lies in the transformed plane

$$[1 \ 0 \ 0 \ -6] \begin{bmatrix} 6 \\ 0 \\ 9 \\ 1 \end{bmatrix} = 0 \quad (1.31)$$

The general translation operation can be represented in Axiom as

— **translate** —

```
translate(x,y,z) ==
matrix(
  [[1,0,0,x],_
   [0,1,0,y],_
   [0,0,1,z],_
   [0,0,0,1]])
```

Rotation Transformations

The transformations corresponding to rotations about the x , y , and z axes by an angle θ are

$$\mathbf{Rot}(\mathbf{x}, \theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.32)$$

Rotations can be described in Axiom as functions that return matrices. We can define a function for each of the rotation matrices that correspond to the rotations about each axis. Note that the sine and cosine functions in Axiom expect their argument to be in radians rather than degrees. This conversion is

$$radians = \frac{degrees * \pi}{180}$$

The Axiom code for **Rot(x, degree)** is

— **rotatex** —

```
rotatex(degree) ==
angle := degree * pi() / 180::R
cosAngle := cos(angle)
sinAngle := sin(angle)
matrix(
  [[1, 0, 0, 0], _
   [0, cosAngle, -sinAngle, 0], _
   [0, sinAngle, cosAngle, 0], _
   [0, 0, 0, 1]])
```

$$\mathbf{Rot}(\mathbf{y}, \theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.33)$$

The Axiom code for $\mathbf{Rot}(\mathbf{y}, \text{degree})$ is

```

— rotatey —

rotatey(degree) ==
angle := degree * pi() / 180::R
cosAngle := cos(angle)
sinAngle := sin(angle)
matrix(_
[[ cosAngle, 0, sinAngle, 0], _
[ 0, 1, 0, 0], _
[-sinAngle, 0, cosAngle, 0], _
[ 0, 0, 0, 1]])

```

$$\mathbf{Rot}(\mathbf{z}, \theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.34)$$

And the Axiom code for $\mathbf{Rot}(\mathbf{z}, \text{degree})$ is

```

— rotatez —

rotatez(degree) ==
angle := degree * pi() / 180::R
cosAngle := cos(angle)
sinAngle := sin(angle)
matrix(_
[[cosAngle, -sinAngle, 0, 0], _
[sinAngle, cosAngle, 0, 0], _
[ 0, 0, 1, 0], _
[ 0, 0, 0, 1]])

```

Let us interpret these rotations by means of an example. Given a point $\mathbf{u} = 7\mathbf{i} + 3\mathbf{j} + 2\mathbf{k}$ what is the effect of rotating it 90° about the \mathbf{z} axis to \mathbf{v} ? The transform is obtained from Equation 1.34 with $\sin \theta = 1$ and $\cos \theta = 0$.

$$\begin{bmatrix} -3 \\ 7 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 7 \\ 3 \\ 2 \\ 1 \end{bmatrix} \quad (1.35)$$

Let us now rotate \mathbf{v} 90° about the y axis to \mathbf{w} . The transform is obtained from Equation 1.33 and we have

$$\begin{bmatrix} 2 \\ 7 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -3 \\ 7 \\ 2 \\ 1 \end{bmatrix} \quad (1.36)$$

If we combine these two rotations we have

$$\mathbf{v} = \mathbf{Rot}(\mathbf{z}, 90)\mathbf{u} \quad (1.37)$$

$$\text{and } \mathbf{w} = \mathbf{Rot}(\mathbf{y}, 90)\mathbf{v} \quad (1.38)$$

Substituting for \mathbf{v} from Equation 1.37 into Equation 1.38 we obtain

$$\mathbf{w} = \mathbf{Rot}(\mathbf{y}, 90) \mathbf{Rot}(\mathbf{z}, 90) \mathbf{u} \quad (1.39)$$

$$\mathbf{Rot}(\mathbf{y}, 90) \mathbf{Rot}(\mathbf{z}, 90) = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.40)$$

$$\mathbf{Rot}(\mathbf{y}, 90) \mathbf{Rot}(\mathbf{z}, 90) = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.41)$$

thus

$$\mathbf{w} = \begin{bmatrix} 2 \\ 7 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 7 \\ 3 \\ 2 \\ 1 \end{bmatrix} \quad (1.42)$$

as we obtained before.

If we reverse the order of rotations and first rotate 90° about the y axis and then 90° about the z axis, we obtain a different position

$$\mathbf{Rot}(\mathbf{z}, 90)\mathbf{Rot}(\mathbf{y}, 90) = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.43)$$

and the point \mathbf{u} transforms into \mathbf{w} as

$$\begin{bmatrix} -3 \\ 2 \\ -7 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 7 \\ 3 \\ 2 \\ 1 \end{bmatrix} \quad (1.44)$$

We should expect this, as matrix multiplication is noncommutative.

$$\mathbf{AB} \neq \mathbf{BA} \quad (1.45)$$

We will now combine the original rotation with a translation $4\mathbf{i} - 3\mathbf{j} + 7\mathbf{k}$. We obtain the translation from Equation 1.27 and the rotation from Equation 1.41. The matrix expression is

$$\mathbf{Trans}(4, -3, 7)\mathbf{Rot}(\mathbf{y}, 90)\mathbf{Rot}(\mathbf{z}, 90) = \begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & -3 \\ 0 & 0 & 1 & 7 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 4 \\ 1 & 0 & 0 & -3 \\ 0 & 1 & 0 & 7 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.46)$$

and our point $\mathbf{w} = 7\mathbf{i} + 3\mathbf{j} + 2\mathbf{k}$ transforms into \mathbf{x} as

$$\begin{bmatrix} 6 \\ 4 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 4 \\ 1 & 0 & 0 & -3 \\ 0 & 1 & 0 & 7 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 7 \\ 3 \\ 2 \\ 1 \end{bmatrix} \quad (1.47)$$

Coordinate Frames

We can interpret the elements of the homogeneous transformation as four vectors describing a second coordinate frame. The vector $[0, 0, 0, 1]^T$ lies at the origin of the second coordinate frame. Its transformation corresponds to the right hand column of the transformation matrix. Consider the transform in Equation 1.47

$$\begin{bmatrix} 4 \\ -3 \\ 7 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 4 \\ 1 & 0 & 0 & -3 \\ 0 & 1 & 0 & 7 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad (1.48)$$

The transform of the null vector is $[4, -3, 7, 1]^T$, the right hand column. If we transform vectors corresponding to unit vectors along the x , y , and z axes, we obtain $[4, -2, 7, 1]^T$, $[4, -3, 8, 1]^T$, and $[5, -3, 7, 1]^T$, respectively. Those four vectors form a coordinate frame.

The direction of these unit vectors is formed by subtracting the vector representing the origin of this coordinate frame and extending the vectors to infinity by reducing their scale factors to zero. The direction of the x , y , and z axes of this frame are $[0, 1, 0, 0]^T$, $[0, 0, 1, 0]^T$, and $[1, 0, 0, 0]^T$, respectively. These direction vectors correspond to the first three columns of the transformation matrix. The transformation matrix thus describes the three axis directions and the position of the origin of a coordinate frame rotated and translated away from the reference coordinate frame. When a vector is transformed, as in Equation 1.47, the original vector can be considered as a vector described in the coordinate frame. The transformed vector is the same vector described with respect to the reference coordinate frame.

Relative Transformations

The rotations and translations we have been describing have all been made with respect to the fixed reference coordinate frame. Thus, in the example given,

$$\mathbf{Trans}(4, -3, 7)\mathbf{Rot}(y, 90)\mathbf{Rot}(z, 90) = \begin{bmatrix} 0 & 0 & 1 & 4 \\ 1 & 0 & 0 & -3 \\ 0 & 1 & 0 & 7 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.49)$$

the frame is first rotated around the reference z axis by 90° , then rotated 90° around the reference y axis, and finally translated by $4\mathbf{i} - 3\mathbf{j} + 7\mathbf{k}$. We may also interpret the operation in the reverse order, from left to right, as follows: the object is first translated by $4\mathbf{i} - 3\mathbf{j} + 7\mathbf{k}$; it is then rotated 90° around the current frames axes, which in this case are the same as the reference axes; it is then rotated 90° about the newly rotated (current) frames axes.

In general, if we postmultiply a transform representing a frame by a second transformation describing a rotation and/or translation, we make that translation and/or rotation with respect to the frame axes described by the first transformation. If we premultiply the frame transformation by a transformation representing a translation and/or rotation, then that translation and/or rotation is made with respect to the base reference coordinate frame. Thus, given a frame \mathbf{C} and a transformation \mathbf{T} , corresponding to a rotation of 90° about the z axis, and a translation of 10 units in the x direction, we obtain a new position \mathbf{X} when the change is made in the base coordinates $\mathbf{X} = \mathbf{TC}$

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 20 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 & 10 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 20 \\ 0 & 0 & -1 & 10 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.50)$$

and a new position \mathbf{Y} when the change is made relative to the frame axes as $\mathbf{Y} = \mathbf{CT}$

$$\begin{bmatrix} 0 & -1 & 0 & 30 \\ 0 & 0 & -1 & 10 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 20 \\ 0 & 0 & -1 & 10 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 & 10 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.51)$$

Objects

Transformations are used to describe the position and orientation of objects. An object is described by six points with respect to a coordinate frame fixed in the object.

If we rotate the object 90° about the z axis and then 90° about the y axis, followed by a translation of four units in the x direction, we can describe the transformation as

$$\mathbf{Trans}(4, 0, 0)\mathbf{Rot}(y, 90)\mathbf{Rot}(z, 90) = \begin{bmatrix} 0 & 0 & 1 & 4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.52)$$

The transformation matrix represents the operation of rotation and translation on a coordinate frame originally aligned with the reference coordinate frame. We may transform the six points of the object as

$$\begin{bmatrix} 4 & 4 & 6 & 6 & 4 & 4 \\ 1 & -1 & -1 & 1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 4 & 4 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & -1 & -1 & 1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 4 & 4 \\ 0 & 0 & 2 & 2 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \quad (1.53)$$

It can be seen that the object described bears the same fixed relationship to its coordinate frame, whose position and orientation are described by the transformation. Given an object described by a reference coordinate frame, and a transformation representing the position and orientation of the object's axes, the object can be simply reconstructed, without the necessity of transforming all the points, by noting the direction and orientation of key features with respect to the describing frame's coordinate axes. By drawing the transformed coordinate frame, the object can be related to the new axis directions.

Inverse Transformations

We are now in a position to develop the inverse transformation as the transform which carries the transformed coordinate frame back to the original frame. This is simply the description of the reference coordinate frame with respect to the transformed frame. Suppose the direction of the reference frame x axis is $[0, 0, 1, 0]^T$ with respect to the transformed frame. The y and z axes are $[1, 0, 0, 0]^T$ and $[0, 1, 0, 0]^T$, respectively. The location of the origin is $[0, 0, -4, 1]^T$ with respect to the transformed frame and thus the inverse transformation is

$$\mathbf{T}^{-1} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & -4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.54)$$

That this is indeed the transform inverse is easily verified by multiplying it by the transform \mathbf{T} to obtain the identity transform

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & -4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.55)$$

In general, given a transform with elements

$$\mathbf{T} = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.56)$$

then the inverse is

$$\mathbf{T}^{-1} = \begin{bmatrix} n_x & n_y & n_z & -\mathbf{p} \cdot \mathbf{n} \\ o_x & o_y & o_z & -\mathbf{p} \cdot \mathbf{o} \\ a_x & a_y & a_z & -\mathbf{p} \cdot \mathbf{a} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.57)$$

where \mathbf{p} , \mathbf{n} , \mathbf{o} , and \mathbf{a} are the four column vectors and “ \cdot ” represents the vector dot product. This result is easily verified by postmultiplying Equation 1.56 by Equation 1.57.

General Rotation Transformation

We state the rotation transformations for rotations about the x , y , and z axes (Equations 1.32, 1.33 and 1.34). These transformations have a simple geometric interpretation. For example, in the case of a rotation about the z axis, the column representing the z axis will remain constant, while the column elements representing the x and y axes will vary.

We will now develop the transformation matrix representing a rotation around an arbitrary vector \mathbf{k} located at the origin. In order to do this we will imagine that \mathbf{k} is the z axis unit vector of a coordinate frame \mathbf{C}

$$\mathbf{C} = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.58)$$

$$\mathbf{k} = a_x \mathbf{i} + a_y \mathbf{j} + a_z \mathbf{k} \quad (1.59)$$

Rotating around the vector \mathbf{k} is then equivalent to rotating around the z axis of the frame \mathbf{C} .

$$\mathbf{Rot}(\mathbf{k}, \theta) = \mathbf{Rot}({}^{\mathbf{C}}\mathbf{z}, \theta) \quad (1.60)$$

If we are given a frame \mathbf{T} described with respect to the reference coordinate frame, we can find a frame \mathbf{X} which describes the same frame with respect to frame \mathbf{C} as

$$\mathbf{T} = \mathbf{C}\mathbf{X} \quad (1.61)$$

where \mathbf{X} describes the position of \mathbf{T} with respect to frame \mathbf{C} . Solving for \mathbf{X} we obtain

$$\mathbf{X} = \mathbf{C}^{-1}\mathbf{T} \quad (1.62)$$

Rotation \mathbf{T} around \mathbf{k} is equivalent to rotating \mathbf{X} around the z axis of frame \mathbf{C}

$$\mathbf{Rot}(\mathbf{k}, \theta)\mathbf{T} = \mathbf{C}\mathbf{Rot}(\mathbf{z}, \theta)\mathbf{X} \quad (1.63)$$

$$\mathbf{Rot}(\mathbf{k}, \theta)\mathbf{T} = \mathbf{C}\mathbf{Rot}(\mathbf{z}, \theta)\mathbf{C}^{-1}\mathbf{T}. \quad (1.64)$$

Thus

$$\mathbf{Rot}(\mathbf{k}, \theta) = \mathbf{C}\mathbf{Rot}(\mathbf{z}, \theta)\mathbf{C}^{-1} \quad (1.65)$$

However, we have only \mathbf{k} , the z axis of the frame \mathbf{C} . By expanding equation 1.65 we will discover that $\mathbf{C}\mathbf{Rot}(\mathbf{z}, \theta)\mathbf{C}^{-1}$ is a function of \mathbf{k} only.

Multiplying $\mathbf{Rot}(\mathbf{z}, \theta)$ on the right by \mathbf{C}^{-1} we obtain

$$\begin{aligned} \mathbf{Rot}(\mathbf{z}, \theta)\mathbf{C}^{-1} &= \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n_x & n_y & n_z & 0 \\ o_x & o_x & o_z & 0 \\ a_x & a_y & a_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} n_x \cos\theta - o_x \sin\theta & n_y \cos\theta - o_y \sin\theta & n_z \cos\theta - o_z \sin\theta & 0 \\ n_x \sin\theta + o_x \cos\theta & n_y \sin\theta + o_y \cos\theta & n_z \sin\theta + o_z \cos\theta & 0 \\ a_x & a_y & a_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \quad (1.66)$$

premultiplying by

$$\mathbf{C} = \begin{bmatrix} n_x & o_x & a_x & 0 \\ n_y & o_y & a_y & 0 \\ n_z & o_z & a_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.67)$$

we obtain $\mathbf{C}\mathbf{Rot}(\mathbf{z}, \theta)\mathbf{C}^{-1}$

$$\begin{bmatrix} n_x n_x \cos \theta - n_x o_x \sin \theta + n_x o_x \sin \theta + o_x o_x \cos \theta + a_x a_x \\ n_y n_x \cos \theta - n_y o_x \sin \theta + n_x o_y \sin \theta + o_x o_y \cos \theta + a_y a_x \\ n_z n_x \cos \theta - n_z o_x \sin \theta + n_x o_z \sin \theta + o_x o_z \cos \theta + a_z a_x \\ 0 \\ n_x n_y \cos \theta - n_x o_y \sin \theta + n_y o_x \sin \theta + o_y o_x \cos \theta + a_x a_y \\ n_y n_y \cos \theta - n_y o_y \sin \theta + n_y o_y \sin \theta + o_y o_y \cos \theta + a_y a_y \\ n_z n_y \cos \theta - n_z o_y \sin \theta + n_y o_z \sin \theta + o_y o_z \cos \theta + a_z a_y \\ 0 \\ n_x n_z \cos \theta - n_x o_z \sin \theta + n_z o_x \sin \theta + o_z o_x \cos \theta + a_x a_z \\ n_y n_z \cos \theta - n_y o_z \sin \theta + n_z o_y \sin \theta + o_z o_y \cos \theta + a_y a_z \\ n_z n_z \cos \theta - n_z o_z \sin \theta + n_z o_z \sin \theta + o_z o_z \cos \theta + a_z a_z \\ 0 \end{bmatrix} \quad (1.68)$$

Simplifying, using the following relationships:

the dot product of any row or column of \mathbf{C} with any other row or column is zero, as the vectors are orthogonal;

the dot product of any row or column of \mathbf{C} with itself is $\mathbf{1}$ as the vectors are of unit magnitude;

the z unit vector is the vector cross product of the x and y vectors or

$$\mathbf{a} = \mathbf{n} \times \mathbf{o} \quad (1.69)$$

which has components

$$a_x = n_y o_z - n_z o_y$$

$$a_y = n_z o_x - n_x o_z$$

$$a_z = n_x o_y - n_y o_x$$

the versine, abbreviated **vers** θ , is defined as **vers** $\theta = (1 - \cos \theta)$, $k_x = a_x$, $k_y = a_y$ and $k_z = a_z$. We obtain $\mathbf{Rot}(\mathbf{k}, \theta) =$

$$\begin{bmatrix} k_x k_x \text{vers} \theta + \cos \theta & k_y k_x \text{vers} \theta - k_z \sin \theta & k_z k_x \text{vers} \theta + k_y \sin \theta & 0 \\ k_x k_y \text{vers} \theta + k_z \sin \theta & k_y k_y \text{vers} \theta + \cos \theta & k_z k_y \text{vers} \theta - k_x \sin \theta & 0 \\ k_x k_z \text{vers} \theta - k_y \sin \theta & k_y k_z \text{vers} \theta + k_x \sin \theta & k_z k_z \text{vers} \theta + \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.70)$$

This is an important result and should be thoroughly understood before proceeding further.

From this general rotation transformation we can obtain each of the elementary rotation transforms. For example $\mathbf{Rot}(\mathbf{x}, \theta)$ is $\mathbf{Rot}(\mathbf{k}, \theta)$ where $k_x = 1$, $k_y = 0$, and $k_z = 0$. Substituting these values of \mathbf{k} into Equation 1.70 we obtain

$$\mathbf{Rot}(\mathbf{x}, \theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.71)$$

as before.

Equivalent Angle and Axis of Rotation

Given any arbitrary rotational transformation, we can use Equation 1.70 to obtain an axis about which an equivalent rotation θ is made as follows. Given a rotational transformation \mathbf{R}

$$\mathbf{R} = \begin{bmatrix} n_x & o_x & a_x & 0 \\ n_y & o_y & a_y & 0 \\ n_z & o_z & a_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.72)$$

we may equate \mathbf{R} to $\mathbf{Rot}(\mathbf{k}, \theta)$

$$\begin{bmatrix} n_x & o_x & a_x & 0 \\ n_y & o_y & a_y & 0 \\ n_z & o_z & a_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} k_x k_x \text{vers}\theta + \cos\theta & k_y k_x \text{vers}\theta - k_z \sin\theta & k_z k_x \text{vers}\theta + k_y \sin\theta & 0 \\ k_x k_y \text{vers}\theta + k_z \sin\theta & k_y k_y \text{vers}\theta + \cos\theta & k_z k_y \text{vers}\theta - k_x \sin\theta & 0 \\ k_x k_z \text{vers}\theta - k_y \sin\theta & k_y k_z \text{vers}\theta + k_x \sin\theta & k_z k_z \text{vers}\theta + \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.73)$$

Summing the diagonal terms of Equation 1.73 we obtain

$$n_x + o_y + a_z + 1 = k_x^2 \text{vers}\theta + \cos\theta + k_y^2 \text{vers}\theta + \cos\theta + k_z^2 \text{vers}\theta + \cos\theta + 1 \quad (1.74)$$

$$\begin{aligned} n_x + o_y + a_z &= (k_x^2 + k_y^2 + k_z^2) \text{vers}\theta + 3\cos\theta \\ &= 1 + 2\cos\theta \end{aligned} \quad (1.75)$$

and the cosine of the angle of rotation is

$$\cos\theta = \frac{1}{2}(n_x + o_y + a_z - 1) \quad (1.76)$$

Differencing pairs of off-diagonal terms in Equation 1.73 we obtain

$$o_z - a_y = 2k_x \sin \theta \quad (1.77)$$

$$a_x - n_z = 2k_y \sin \theta \quad (1.78)$$

$$n_y - o_x = 2k_z \sin \theta \quad (1.79)$$

Squaring and adding Equations 1.77-1.79 we obtain an expression for $\sin \theta$

$$(o_z - a_y)^2 + (a_x - n_z)^2 + (n_y - o_x)^2 = 4\sin^2 \theta \quad (1.80)$$

and the sine of the angle of rotation is

$$\sin \theta = \pm \frac{1}{2} \sqrt{(o_z - a_y)^2 + (a_x - n_z)^2 + (n_y - o_x)^2} \quad (1.81)$$

We may define the rotation to be positive about the vector \mathbf{k} such that $0 \leq \theta \leq 180^\circ$. In this case the $+$ sign is appropriate in Equation 1.81 and thus the angle of rotation θ is uniquely defined as

$$\tan \theta = \frac{\sqrt{(o_z - a_y)^2 + (a_x - n_z)^2 + (n_y - o_x)^2}}{(n_x + o_y + a_z - 1)} \quad (1.82)$$

The components of \mathbf{k} may be obtained from Equations 1.77-1.79 as

$$k_x = \frac{o_z - a_y}{2\sin \theta} \quad (1.83)$$

$$k_y = \frac{a_x - n_z}{2\sin \theta} \quad (1.84)$$

$$k_z = \frac{n_y - o_x}{2\sin \theta} \quad (1.85)$$

When the angle of rotation is very small, the axis of rotation is physically not well defined due to the small magnitude of both numerator and denominator in Equations 1.83-1.85. If the resulting angle is small, the vector \mathbf{k} should be renormalized to ensure that $|\mathbf{k}| = 1$. When the angle of rotation approaches 180° the vector \mathbf{k} is once again poorly defined by Equation 1.83-1.85 as the magnitude of the sine is again decreasing. The axis of rotation is, however, physically well defined in this case. When $\theta < 150^\circ$, the denominator of Equations 1.83-1.85 is less than 1. As the angle increases to 180° the rapidly decreasing magnitude of both numerator and denominator leads to considerable inaccuracies in the determination of \mathbf{k} . At $\theta = 180^\circ$, Equations 1.83-1.85 are of the form $0/0$, yielding no information at all about a physically well defined vector \mathbf{k} . If the angle of rotation is greater than 90° , then we must follow a different approach in determining \mathbf{k} . Equating the diagonal elements of Equation 1.73 we obtain

$$k_x^2 \cos \theta + \cos \theta = n_x \quad (1.86)$$

$$k_y^2 \text{vers}\theta + \cos\theta = o_y \quad (1.87)$$

$$k_z^2 \text{vers}\theta + \cos\theta = a_z \quad (1.88)$$

Substituting for $\cos\theta$ and $\text{vers}\theta$ from Equation 1.76 and solving for the elements of \mathbf{k} we obtain further

$$k_x = \pm \sqrt{\frac{n_x - \cos\theta}{1 - \cos\theta}} \quad (1.89)$$

$$k_y = \pm \sqrt{\frac{o_y - \cos\theta}{1 - \cos\theta}} \quad (1.90)$$

$$k_z = \pm \sqrt{\frac{a_z - \cos\theta}{1 - \cos\theta}} \quad (1.91)$$

The largest component of \mathbf{k} defined by Equations 1.89-1.91 corresponds to the most positive component of n_x , o_y , and a_z . For this largest element, the sign of the radical can be obtained from Equations 1.77-1.79. As the sine of the angle of rotation θ must be positive, then the sign of the component of \mathbf{k} defined by Equations 1.77-1.79 must be the same as the sign of the left hand side of these equations. Thus we may combine Equations 1.89-1.91 with the information contained in Equations 1.77-1.79 as follows

$$k_x = \text{sgn}(o_z - a_y) \sqrt{\frac{(n_x - \cos\theta)}{1 - \cos\theta}} \quad (1.92)$$

$$k_y = \text{sgn}(a_x - n_z) \sqrt{\frac{(o_y - \cos\theta)}{1 - \cos\theta}} \quad (1.93)$$

$$k_z = \text{sgn}(n_y - o_x) \sqrt{\frac{(a_z - \cos\theta)}{1 - \cos\theta}} \quad (1.94)$$

where $\text{sgn}(e) = +1$ if $e \geq 0$ and $\text{sgn}(e) = -1$ if $e \leq 0$.

Only the largest element of \mathbf{k} is determined from Equations 1.92-1.94, corresponding to the most positive element of n_x , o_y , and a_z . The remaining elements are more accurately determined by the following equations formed by summing pairs of off-diagonal elements of Equation 1.73

$$n_y + o_x = 2k_x k_y \text{vers}\theta \quad (1.95)$$

$$o_z + a_y = 2k_y k_z \text{vers}\theta \quad (1.96)$$

$$n_z + a_x = 2k_z k_x \text{vers}\theta \quad (1.97)$$

If k_x is largest then

$$k_y = \frac{n_y + o_x}{2k_x \text{vers}\theta} \quad \text{from Equation 1.95} \quad (1.98)$$

$$k_z = \frac{a_x + n_z}{2k_x \text{vers}\theta} \quad \text{from Equation 1.97} \quad (1.99)$$

If k_y is largest then

$$k_x = \frac{n_y + o_x}{2k_y \text{vers}\theta} \quad \text{from Equation 1.95} \quad (1.100)$$

$$k_z = \frac{o_z + a_y}{2k_y \text{vers}\theta} \quad \text{from Equation 1.96} \quad (1.101)$$

If k_z is largest then

$$k_x = \frac{a_x + n_z}{2k_z \text{vers}\theta} \quad \text{from Equation 1.97} \quad (1.102)$$

$$k_y = \frac{o_z + a_y}{2k_z \text{vers}\theta} \quad \text{from Equation 1.96} \quad (1.103)$$

Example 1.1

Determine the equivalent axis and angle of rotation for the matrix given in Equations 1.41

$$\mathbf{Rot}(\mathbf{y}, 90)\mathbf{Rot}(\mathbf{z}, 90) = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.104)$$

We first determine $\cos \theta$ from Equation 1.76

$$\cos \theta = \frac{1}{2}(0 + 0 + 0 - 1) = -\frac{1}{2} \quad (1.105)$$

and $\sin \theta$ from Equation 1.81

$$\sin \theta = \frac{1}{2}\sqrt{(1-0)^2 + (1-0)^2 + (1-0)^2} = \frac{\sqrt{3}}{2} \quad (1.106)$$

Thus

$$\theta = \tan^{-1} \left(\frac{\frac{\sqrt{3}}{2}}{-\frac{1}{2}} \right) = 120^\circ \quad (1.107)$$

As $\theta > 90$, we determine the largest component of \mathbf{k} corresponding to the largest element on the diagonal. As all diagonal elements are equal in this example we may pick any one. We will pick k_x given by Equation 1.92

$$k_x = + \sqrt{(0 + \frac{1}{2}) / (1 + \frac{1}{2})} = \frac{1}{\sqrt{3}} \quad (1.108)$$

As we have determined k_x we may now determine k_y and k_z from Equations 1.98 and 1.99, respectively

$$k_y = \frac{1+0}{\sqrt{3}} = \frac{1}{\sqrt{3}} \quad (1.109)$$

$$k_z = \frac{1+0}{\sqrt{3}} = \frac{1}{\sqrt{3}} \quad (1.110)$$

In summary, then

$$\mathbf{Rot}(\mathbf{y}, 90)\mathbf{Rot}(\mathbf{z}, 90) = \mathbf{Rot}(\mathbf{k}, 120) \quad (1.111)$$

where

$$\mathbf{k} = \frac{1}{\sqrt{3}}\mathbf{i} + \frac{1}{\sqrt{3}}\mathbf{j} + \frac{1}{\sqrt{3}}\mathbf{k} \quad (1.112)$$

Any combination of rotations is always equivalent to a single rotation about some axis \mathbf{k} by an angle θ , an important result that we will make use of later.

Stretching and Scaling

A transform \mathbf{T}

$$\mathbf{T} = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.113)$$

will stretch objects uniformly along the x axis by a factor a , along the y axis by a factor b , and along the z axis by a factor c . Consider any point on an object $x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$; its transform is

$$\begin{bmatrix} ax \\ by \\ cz \\ 1 \end{bmatrix} = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (1.114)$$

indicating stretching as stated. Thus a cube could be transformed into a rectangular parallelepiped by such a transform.

The Axiom code to perform this scale change is:

```

— scale —

scale(scalex, scaley, scalez) ==
matrix(
  [[scalex, 0, 0, 0],
   [0, scaley, 0, 0],
   [0, 0, scalez, 0],
   [0, 0, 0, 1]])

```

The transform \mathbf{S} where

$$\mathbf{S} = \begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.115)$$

will scale any object by the factor s .

Perspective Transformations

Consider the image formed of an object by a simple lens.

The axis of the lens is along the y axis for convenience. An object point x, y, z is imaged at x', y', z' if the lens has a focal length f (f is considered positive). y' represents the image distance and varies with object distance y . If we plot points on a plane perpendicular to the y axis located at y' (the film plane in a camera), then a perspective image is formed.

We will first obtain values of x' , y' , and z' , then introduce a perspective transformation and show that the same values are obtained.

Based on the fact that a ray passing through the center of the lens is undeviated we may write

$$\frac{z}{y} = \frac{z'}{y'} \quad (1.116)$$

$$\text{and } \frac{x}{y} = \frac{x'}{y'} \quad (1.117)$$

Based on the additional fact that a ray parallel to the lens axis passes through the focal point f , we may write

$$\frac{z}{f} = \frac{z'}{y' + f} \quad (1.118)$$

$$\text{and } \frac{x}{f} = \frac{x'}{y' + f} \quad (1.119)$$

Notice that x' , y' , and z' are negative and that f is positive. Eliminating y' between Equations 1.116 and 1.118 we obtain

$$\frac{z}{f} = \frac{z'}{\left(\frac{z'y}{z} + f\right)} \quad (1.120)$$

and solving for z' we obtain the result

$$z' = \frac{z}{\left(1 - \frac{y}{f}\right)} \quad (1.121)$$

Working with Equations 1.117 and 1.119 we can similarly obtain

$$x' = \frac{x}{\left(1 - \frac{y}{f}\right)} \quad (1.122)$$

In order to obtain the image distance y' we rewrite Equations 1.116 and 1.118 as

$$\frac{z}{z'} = \frac{y}{y'} \quad (1.123)$$

and

$$\frac{z}{z'} = \frac{f}{y' + f} \quad (1.124)$$

thus

$$\frac{y}{y'} = \frac{f}{y' + f} \quad (1.125)$$

and solving for y' we obtain the result

$$y' = \frac{y}{\left(1 - \frac{y}{f}\right)} \quad (1.126)$$

The homogeneous transformation \mathbf{P} which produces the same result is

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -\frac{1}{f} & 0 & 1 \end{bmatrix} \quad (1.127)$$

as any point $x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$ transforms as

$$\begin{bmatrix} x \\ y \\ z \\ 1 - \frac{y}{f} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -\frac{1}{f} & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (1.128)$$

The image point x', y', z' , obtained by dividing through by the weight factor $(1 - \frac{y}{f})$, is

$$\frac{x}{(1 - \frac{y}{f})} \mathbf{i} + \frac{y}{(1 - \frac{y}{f})} \mathbf{j} + \frac{z}{(1 - \frac{y}{f})} \mathbf{k} \quad (1.129)$$

This is the same result that we obtained above.

A transform similar to \mathbf{P} but with $-\frac{1}{f}$ at the bottom of the first column produces a perspective transformation along the x axis. If the $-\frac{1}{f}$ term is in the third column then the projection is along the z axis.

Transform Equations

We will frequently be required to deal with transform equations in which a coordinate frame is described in two or more ways. A manipulator is positioned with respect to base coordinates by a transform \mathbf{Z} . The end of the manipulator is described by a transform ${}^Z\mathbf{T}_6$, and the end effector is described by ${}^{T_6}\mathbf{E}$. An object is positioned with respect to base coordinates by a transform \mathbf{B} , and finally the manipulator end effector is positioned with respect to the object by ${}^B\mathbf{G}$. We have two descriptions of the position of the end effector, one with respect to the object and one with respect to the manipulator. As both positions are the same, we may equate the two descriptions

$$\mathbf{Z} {}^Z\mathbf{T}_6 {}^{T_6}\mathbf{E} = \mathbf{B} {}^B\mathbf{G} \quad (1.130)$$

If we wish to solve Equation 1.130 for the manipulator transform \mathbf{T}_6 we must premultiply Equation 1.130 by \mathbf{Z}^{-1} and postmultiply by \mathbf{E}^{-1} to obtain

$$\mathbf{T}_6 = \mathbf{Z}^{-1} \mathbf{B} \mathbf{G} \mathbf{E}^{-1} \quad (1.131)$$

As a further example, consider that the position of the object \mathbf{B} is unknown, but that the manipulator is moved such that the end effector is positioned over the object correctly. We may then solve for \mathbf{B} from Equation 1.130 by postmultiplying by \mathbf{G}^{-1} .

$$\mathbf{B} = \mathbf{Z} \mathbf{T}_6 \mathbf{E} \mathbf{G}^{-1} \quad (1.133)$$

Summary

Homogeneous transformations may be readily used to describe the positions and orientations of coordinate frames in space. If a coordinate frame is embedded in an object then the position and orientation of the object are also readily described.

The description of object A in terms of object B by means of a homogeneous transformation may be inverted to obtain the description of object B in terms of object A. This is not a property of a simple vector description of the relative displacement of one object with respect to another.

Transformations may be interpreted as a product of rotation and translation transformations. If they are interpreted from left to right, then the rotations and translations are in terms of the currently defined coordinate frame. If they are interpreted from right to left, then the rotations and translations are described with respect to the reference coordinate frame.

Homogeneous transformations describe coordinate frames in terms of rectangular components, which are the sines and cosines of angles. This description may be related to rotations in which case the description is in terms of a vector and angle of rotation.

— DenavitHartenbergMatrix.input —

```
)set break resume
)sys rm -f DenavitHartenbergMatrix.output
)spool DenavitHartenbergMatrix.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DenavitHartenbergMatrix
--E 1

)spool
)lisp (bye)
```

— DenavitHartenbergMatrix.help —

```
=====
DenavitHartenbergMatrix examples
=====
```

4x4 Matrices for coordinate transformations

This package contains functions to create 4x4 matrices useful for rotating and transforming coordinate systems. These matrices are useful for graphics and robotics. (Reference: Robot Manipulators Richard Paul MIT Press 1981)

A Denavit-Hartenberg Matrix is a 4x4 Matrix of the form:

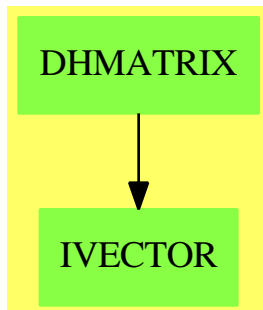
```
nx ox ax px
ny oy ay py
nz oz az pz
```

$\begin{pmatrix} 0 & 0 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$
 where n , o , and a are the direction cosines

See Also:

`o)show DenavitHartenbergMatrix`

DenavitHartenbergMatrix (DHMATRIX)



Exports:

antisymmetric?	any?	coerce	column	copy
count	determinant	diagonal?	diagonalMatrix	elt
empty	empty?	eq?	eval	every?
exquo	fill!	hash	horizConcat	identity
inverse	latex	less?	listOfLists	map
map!	matrix	maxColIndex	maxRowIndex	member?
members	minColIndex	minordet	minRowIndex	more?
ncols	new	nrows	nullSpace	nullity
parts	qelt	qsetelt!	rank	rotatex
rotatey	rotatez	row	rowEchelon	sample
scalarMatrix	scale	setColumn!	setRow!	setelt
setsubMatrix!	size?	square?	squareTop	subMatrix
swapColumns!	swapRows!	symmetric?	translate	transpose
vertConcat	zero	#?	?**?	?/?
?=?	?~=?	?*?	?+?	-?
?-?				

— domain DHMATRIX DenavitHartenbergMatrix —

`)abbrev domain DHMATRIX DenavitHartenbergMatrix`

```

++ Author: Timothy Daly
++ Date Created: June 26, 1991
++ Date Last Updated: 26 June 1991
++ Description:
++ 4x4 Matrices for coordinate transformations\br
++ This package contains functions to create 4x4 matrices
++ useful for rotating and transforming coordinate systems.
++ These matrices are useful for graphics and robotics.
++ (Reference: Robot Manipulators Richard Paul MIT Press 1981)
++
++ A Denavit-Hartenberg Matrix is a 4x4 Matrix of the form:\br
++ \tab{5}\spad{nx ox ax px}\br
++ \tab{5}\spad{ny oy ay py}\br
++ \tab{5}\spad{nz oz az pz}\br
++ \tab{5}\spad{0 0 0 1}\br
++ (n, o, and a are the direction cosines)

DenavitHartenbergMatrix(R): Exports == Implementation where
  R : Join(Field, TranscendentalFunctionCategory)

-- for the implementation of dhmatrix
  minrow ==> 1
  mincolumn ==> 1
--
  nx ==> x(1,1)::R
  ny ==> x(2,1)::R
  nz ==> x(3,1)::R
  ox ==> x(1,2)::R
  oy ==> x(2,2)::R
  oz ==> x(3,2)::R
  ax ==> x(1,3)::R
  ay ==> x(2,3)::R
  az ==> x(3,3)::R
  px ==> x(1,4)::R
  py ==> x(2,4)::R
  pz ==> x(3,4)::R
  row ==> Vector(R)
  col ==> Vector(R)
  radians ==> pi()/180

Exports ==> MatrixCategory(R,row,col) with
  "(": (% , Point R) -> Point R
  ++ t*p applies the dhmatrix t to point p
  identity: () -> %
  ++ identity() create the identity dhmatrix
  rotatex: R -> %
  ++ rotatex(r) returns a dhmatrix for rotation about axis x for r degrees
  rotatey: R -> %
  ++ rotatey(r) returns a dhmatrix for rotation about axis y for r degrees
  rotatez: R -> %

```

```

++ rotatez(r) returns a dhmatrix for rotation about axis z for r degrees
scale: (R,R,R) -> %
++ scale(sx,sy,sz) returns a dhmatrix for scaling in the x, y and z
++ directions
translate: (R,R,R) -> %
++ translate(x,y,z) returns a dhmatrix for translation by x, y, and z

Implementation ==> Matrix(R) add

identity() == matrix([[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])

-- inverse(x) == (inverse(x pretend (Matrix R))$Matrix(R)) pretend %
-- dhinverse(x) == matrix( _
--   [[nx,ny,nz,-(px*nx+py*ny+pz*nz)],_
--   [ox,oy,oz,-(px*ox+py*oy+pz*oz)],_
--   [ax,ay,az,-(px*ax+py*ay+pz*az)],_
--   [ 0, 0, 0, 1]])

d * p ==
  v := p pretend Vector R
  v := concat(v, 1$R)
  v := d * v
  point ([v.1, v.2, v.3]$List(R))

\getchunk{rotatex}

\getchunk{rotatey}

\getchunk{rotatez}

\getchunk{scale}

\getchunk{translate}

```

— DHMATRIX.dotabb —

```

"DHMATRIX" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DHMATRIX"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"DHMATRIX" -> "IVECTOR"

```

domain DEQUEUE Dequeue

— Dequeue.input —

```
)set break resume
)sys rm -f Dequeue.output
)spool Dequeue.output
)set message test on
)set message auto off
)clear all
```

```
--S 1 of 63
a:Dequeue INT:= dequeue [1,2,3,4,5]
--R
--R
--R (1) [1,2,3,4,5]
--R
--E 1
```

Type: Dequeue(Integer)

```
--S 2 of 63
dequeue! a
--R
--R
--R (2) 1
--R
--E 2
```

Type: PositiveInteger

```
--S 3 of 63
a
--R
--R
--R (3) [2,3,4,5]
--R
--E 3
```

Type: Dequeue(Integer)

```
--S 4 of 63
extract! a
--R
--R
--R (4) 2
--R
--E 4
```

Type: PositiveInteger

```
--S 5 of 63
a
--R
--R
--R (5) [3,4,5]
```

[illegible]

```

--S 12 of 63
bottom! a
--R
--R
--R (12)  8
--R
--R                                         Type: PositiveInteger
--E 12

--S 13 of 63
a
--R
--R
--R (13)  [3,4,5,9]
--R
--R                                         Type: Dequeue(Integer)
--E 13

--S 14 of 63
depth a
--R
--R
--R (14)  4
--R
--R                                         Type: PositiveInteger
--E 14

--S 15 of 63
height a
--R
--R
--R (15)  4
--R
--R                                         Type: PositiveInteger
--E 15

--S 16 of 63
insertBottom!(6,a)
--R
--R
--R (16)  6
--R
--R                                         Type: PositiveInteger
--E 16

--S 17 of 63
a
--R
--R
--R (17)  [3,4,5,9,6]
--R
--R                                         Type: Dequeue(Integer)
--E 17

--S 18 of 63

```

```

extractBottom! a
--R
--R
--R (18)  6
--R
--R                                         Type: PositiveInteger
--E 18

--S 19 of 63
a
--R
--R
--R (19)  [3,4,5,9]
--R
--R                                         Type: Dequeue(Integer)
--E 19

--S 20 of 63
insertTop!(7,a)
--R
--R
--R (20)  7
--R
--R                                         Type: PositiveInteger
--E 20

--S 21 of 63
a
--R
--R
--R (21)  [7,3,4,5,9]
--R
--R                                         Type: Dequeue(Integer)
--E 21

--S 22 of 63
extractTop! a
--R
--R
--R (22)  7
--R
--R                                         Type: PositiveInteger
--E 22

--S 23 of 63
a
--R
--R
--R (23)  [3,4,5,9]
--R
--R                                         Type: Dequeue(Integer)
--E 23

--S 24 of 63
top a
--R

```

```

--R
--R (24) 3
--R
--R                                         Type: PositiveInteger
--E 24

--S 25 of 63
a
--R
--R
--R (25) [3,4,5,9]
--R
--R                                         Type: Dequeue(Integer)
--E 25

--S 26 of 63
top! a
--R
--R
--R (26) 3
--R
--R                                         Type: PositiveInteger
--E 26

--S 27 of 63
a
--R
--R
--R (27) [4,5,9]
--R
--R                                         Type: Dequeue(Integer)
--E 27

--S 28 of 63
reverse! a
--R
--R
--R (28) [9,5,4]
--R
--R                                         Type: Dequeue(Integer)
--E 28

--S 29 of 63
rotate! a
--R
--R
--R (29) [5,4,9]
--R
--R                                         Type: Dequeue(Integer)
--E 29

--S 30 of 63
inspect a
--R
--R
--R (30) 5

```

[illegible]

```

--S 37 of 63
size?(a,9)
--R
--R
--R (37) false
--R
--R                                          Type: Boolean
--E 37

--S 38 of 63
parts a
--R
--R
--R (38) [5,4,9]
--R
--R                                          Type: List(Integer)
--E 38

--S 39 of 63
bag([1,2,3,4,5])$Dequeue(INT)
--R
--R
--R (39) [1,2,3,4,5]
--R
--R                                          Type: Dequeue(Integer)
--E 39

--S 40 of 63
b:=empty()$(Dequeue INT)
--R
--R
--R (40) []
--R
--R                                          Type: Dequeue(Integer)
--E 40

--S 41 of 63
empty? b
--R
--R
--R (41) true
--R
--R                                          Type: Boolean
--E 41

--S 42 of 63
sample()$Dequeue(INT)
--R
--R
--R (42) []
--R
--R                                          Type: Dequeue(Integer)
--E 42

--S 43 of 63

```

```

c:=copy a
--R
--R
--R (43) [5,4,9]
--R
--R                                         Type: Dequeue(Integer)
--E 43

--S 44 of 63
eq?(a,c)
--R
--R
--R (44) false
--R
--R                                         Type: Boolean
--E 44

--S 45 of 63
eq?(a,a)
--R
--R
--R (45) true
--R
--R                                         Type: Boolean
--E 45

--S 46 of 63
(a=c)@Boolean
--R
--R
--R (46) true
--R
--R                                         Type: Boolean
--E 46

--S 47 of 63
(a=a)@Boolean
--R
--R
--R (47) true
--R
--R                                         Type: Boolean
--E 47

--S 48 of 63
a~c
--R
--R
--R (48) false
--R
--R                                         Type: Boolean
--E 48

--S 49 of 63
any?(x+->(x=4),a)
--R

```



```

--R
--R (49) true
--R
--R                                         Type: Boolean
--E 49

--S 50 of 63
any?(x+-(x=11),a)
--R
--R
--R (50) false
--R
--R                                         Type: Boolean
--E 50

--S 51 of 63
every?(x+-(x=11),a)
--R
--R
--R (51) false
--R
--R                                         Type: Boolean
--E 51

--S 52 of 63
count(4,a)
--R
--R
--R (52) 1
--R
--R                                         Type: PositiveInteger
--E 52

--S 53 of 63
count(x+-(x>2),a)
--R
--R
--R (53) 3
--R
--R                                         Type: PositiveInteger
--E 53

--S 54 of 63
map(x+-(x+10),a)
--R
--R
--R (54) [15,14,19]
--R
--R                                         Type: Dequeue(Integer)
--E 54

--S 55 of 63
a
--R
--R
--R (55) [5,4,9]

```

[illegible]

```

--S 62 of 63
latex a
--R
--R
--R (62) "\mbox{\bf Unimplemented}"
--R
--R                                          Type: String
--E 62

--S 63 of 63
)show Dequeue
--R
--R Dequeue(S: SetCategory) is a domain constructor
--R Abbreviation for Dequeue is DEQUEUE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for DEQUEUE
--R
--R----- Operations -----
--R back : % -> S                                bag : List(S) -> %
--R bottom! : % -> S                            copy : % -> %
--R depth : % -> NonNegativeInteger            dequeue : List(S) -> %
--R dequeue : () -> %                          dequeue! : % -> S
--R empty : () -> %                            empty? : % -> Boolean
--R enqueue! : (S,%) -> S                      eq? : (%,%) -> Boolean
--R extract! : % -> S                          extractBottom! : % -> S
--R extractTop! : % -> S                      front : % -> S
--R height : % -> NonNegativeInteger           insert! : (S,%) -> %
--R insertBottom! : (S,%) -> S                 insertTop! : (S,%) -> S
--R inspect : % -> S                          length : % -> NonNegativeInteger
--R map : ((S -> S),%) -> %                   pop! : % -> S
--R push! : (S,%) -> S                       reverse! : % -> %
--R rotate! : % -> %                         sample : () -> %
--R top : % -> S                             top! : % -> S
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?? : (%,%) -> Boolean if S has SETCAT
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if S has SETCAT
--R count : (S,%) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R eval : (%,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R hash : % -> SingleInteger if S has SETCAT
--R latex : % -> String if S has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R member? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R members : % -> List(S) if $ has finiteAggregate

```

```

--R more? : (% , NonNegativeInteger) -> Boolean
--R parts : % -> List(S) if $ has finiteAggregate
--R size? : (% , NonNegativeInteger) -> Boolean
--R ?~=? : (% , %) -> Boolean if S has SETCAT
--R
--E 63

```

```

)spool
)lisp (bye)

```

— Dequeue.help —

=====

Dequeue examples

=====

A Dequeue is a double-ended queue so elements can be added to either end.

Here we create an dequeue of integers from a list. Notice that the order in the list is the order in the dequeue.

```

a:Dequeue INT:= dequeue [1,2,3,4,5]
[1,2,3,4,5]

```

We can remove the top of the dequeue using dequeue!:

```

dequeue! a
1

```

Notice that the use of dequeue! is destructive (destructive operations in Axiom usually end with ! to indicate that the underlying data structure is changed).

```

a
[2,3,4,5]

```

The extract! operation is another name for the dequeue! operation and has the same effect. This operation treats the dequeue as a BagAggregate:

```

extract! a
2

```

and you can see that it also has destructively modified the dequeue:

```

a

```

```
[3,4,5]
```

Next we use `enqueue!` to add a new element to the end of the dequeue:

```
enqueue!(9,a)
9
```

Again, the `enqueue!` operation is destructive so the dequeue is changed:

```
a
[3,4,5,9]
```

Another name for `enqueue!` is `insert!`, which treats the dequeue as a `BagAggregate`:

```
insert!(8,a)
[3,4,5,9,8]
```

and it modifies the dequeue:

```
a
[3,4,5,9,8]
```

The `front` operation returns the item at the front of the dequeue:

```
front a
3
```

The `back` operation returns the item at the back of the dequeue:

```
back a
8
```

The `bottom!` operation returns the item at the back of the dequeue:

```
bottom! a
8
```

and it modifies the dequeue:

```
a
[3,4,5,9]
```

The `depth` function returns the number of elements in the dequeue:

```
depth a
4
```

The `height` function returns the number of elements in the dequeue:

```
height a
4
```

The insertBottom! function adds the element at the end:

```
insertBottom!(6,a)
6
```

and it modifies the dequeue:

```
a
[3,4,5,9,6]
```

The extractBottom! function removes the element at the end:

```
extractBottom! a
6
```

and it modifies the dequeue:

```
a
[3,4,5,9]
```

The insertTop! function adds the element at the top:

```
insertTop!(7,a)
7
```

and it modifies the dequeue:

```
a
[7,3,4,5,9]
```

The extractTop! function adds the element at the top:

```
extractTop! a
7
```

and it modifies the dequeue:

```
a
[3,4,5,9]
```

The top function returns the top element:

```
top a
3
```

and it does not modifies the dequeue:

```
a
  [3,4,5,9]
```

The `top!` function returns the top element:

```
top! a
  3
```

and it modifies the dequeue:

```
a
  [4,5,9]
```

The `reverse!` operation destructively reverses the elements of the dequeue:

```
reverse! a
  [9,5,4]
```

The `rotate!` operation moves the top element to the bottom:

```
rotate! a
  [5,4,9]
```

The `inspect` function returns the top of the dequeue without modification, viewed as a `BagAggregate`:

```
inspect a
  5
```

The `empty?` operation returns true only if there are no element on the dequeue, otherwise it returns false:

```
empty? a
  false
```

The `#` (length) operation:

```
#a
  3
```

The `length` operation does the same thing:

```
length a
  3
```

The `less?` predicate will compare the dequeue length to an integer:

```
less?(a,9)
  true
```

The more? predicate will compare the dequeue length to an integer:

```
more?(a,9)
  false
```

The size? operation will compare the dequeue length to an integer:

```
size?(a,#a)
  true
```

and since the last computation must always be true we try:

```
size?(a,9)
  false
```

The parts function will return the dequeue as a list of its elements:

```
parts a
  [5,4,9]
```

If we have a BagAggregate of elements we can use it to construct a dequeue:

```
bag([1,2,3,4,5])$Dequeue(INT)
  [1,2,3,4,5]
```

The empty function will construct an empty dequeue of a given type:

```
b:=empty()$(Dequeue INT)
  []
```

and the empty? predicate allows us to find out if a dequeue is empty:

```
empty? b
  true
```

The sample function returns a sample, empty dequeue:

```
sample()$Dequeue(INT)
  []
```

We can copy a dequeue and it does not share storage so subsequent modifications of the original dequeue will not affect the copy:

```
c:=copy a
  [5,4,9]
```

The eq? function is only true if the lists are the same reference, so even though c is a copy of a, they are not the same:


```
eq?(a,c)
false
```

However, a clearly shares a reference with itself:

```
eq?(a,a)
true
```

But we can compare a and c for equality:

```
(a=c)@Boolean
true
```

and clearly a is equal to itself:

```
(a=a)@Boolean
true
```

and since a and c are equal, they are clearly NOT not-equal:

```
a~=c
false
```

We can use the any? function to see if a predicate is true for any element:

```
any?(x+>(x=4),a)
true
```

or false for every element:

```
any?(x+>(x=11),a)
false
```

We can use the every? function to check every element satisfies a predicate:

```
every?(x+>(x=11),a)
false
```

We can count the elements that are equal to an argument of this type:

```
count(4,a)
1
```

or we can count against a boolean function:

```
count(x+>(x>2),a)
3
```

You can also map a function over every element, returning a new dequeue:

```
map(x+>x+10,a)
[15,14,19]
```

Notice that the original dequeue is unchanged:

```
a
[5,4,9]
```

You can use `map!` to map a function over every element and change the original dequeue since `map!` is destructive:

```
map!(x+>x+10,a)
[15,14,19]
```

Notice that the original dequeue has been changed:

```
a
[15,14,19]
```

The `members` function can also get the element of the dequeue as a list:

```
members a
[15,14,19]
```

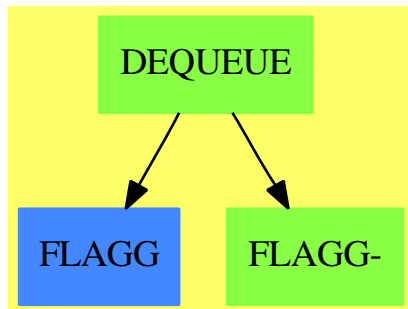
and using `member?` we can test if the dequeue holds a given element:

```
member?(14,a)
true
```

See Also:

- o `)show Stack`
- o `)show ArrayStack`
- o `)show Queue`
- o `)show Dequeue`
- o `)show Heap`
- o `)show BagAggregate`

Deque (DEQUEUE)



See

- ⇒ “Stack” (STACK) [20](#) on page [2563](#)
- ⇒ “ArrayStack” (ASTACK) [2](#) on page [65](#)
- ⇒ “Queue” (QUEUE) [18](#) on page [2184](#)
- ⇒ “Heap” (HEAP) [9](#) on page [1134](#)

Exports:

any?	back	bag	bottom!	coerce
copy	count	depth	dequeue	dequeue!
empty	empty?	enqueue!	eq?	eval
every?	extract!	extractBottom!	extractTop!	front
height	hash	insert!	insertBottom!	insertTop!
inspect	latex	length	less?	map
map!	member?	members	more?	parts
pop!	push!	reverse!	rotate!	sample
size?	top	top!	#?	?=?
?~=?				

— domain DEQUEUE Dequeue —

```

)abbrev domain DEQUEUE Dequeue
++ Author: Michael Monagan and Stephen Watt
++ Date Created: June 86 and July 87
++ Date Last Updated: Feb 92
++ Description:
++ Linked list implementation of a Dequeue

Deque(S:SetCategory): DequeueAggregate S with
  dequeue: List S -> %
    ++ dequeue([x,y,...,z]) creates a dequeue with first (top or front)
    ++ element x, second element y,...,and last (bottom or back) element z.
    ++
    ++E g:Dequeue INT:= dequeue [1,2,3,4,5]

-- Inherited Signatures repeated for examples documentation
  
```

```

dequeue_! : % -> S
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X dequeue! a
++X a
extract_! : % -> S
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X extract! a
++X a
enqueue_! : (S,%) -> S
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X enqueue! (9,a)
++X a
insert_! : (S,%) -> %
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X insert! (8,a)
++X a
inspect : % -> S
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X inspect a
front : % -> S
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X front a
back : % -> S
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X back a
rotate_! : % -> %
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X rotate! a
length : % -> NonNegativeInteger
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X length a
less? : (% ,NonNegativeInteger) -> Boolean
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X less?(a,9)
more? : (% ,NonNegativeInteger) -> Boolean
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X more?(a,9)
size? : (% ,NonNegativeInteger) -> Boolean

```

```

++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X size?(a,5)
bag : List S -> %
++
++X bag([1,2,3,4,5])$Dequeue(INT)
empty? : % -> Boolean
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X empty? a
empty : () -> %
++
++X b:=empty()$(Dequeue INT)
sample : () -> %
++
++X sample()$Dequeue(INT)
copy : % -> %
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X copy a
eq? : (%,%) -> Boolean
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X b:=copy a
++X eq?(a,b)
map : ((S -> S),%) -> %
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X map(x+->x+10,a)
++X a
depth : % -> NonNegativeInteger
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X depth a
dequeue : () -> %
++
++X a:Dequeue INT:= dequeue ()
height : % -> NonNegativeInteger
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X height a
top : % -> S
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X top a
bottom_! : % -> S
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X bottom! a
++X a

```

```

extractBottom_! : % -> S
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X extractBottom! a
++X a
extractTop_! : % -> S
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X extractTop! a
++X a
insertBottom_! : (S,%) -> S
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X insertBottom! a
++X a
insertTop_! : (S,%) -> S
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X insertTop! a
++X a
pop_! : % -> S
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X pop! a
++X a
push_! : (S,%) -> S
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X push! a
++X a
reverse_! : % -> %
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X reverse! a
++X a
top_! : % -> S
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X top! a
++X a
if $ has shallowlyMutable then
  map! : ((S -> S),%) -> %
  ++
  ++X a:Dequeue INT:= dequeue [1,2,3,4,5]
  ++X map!(x+>x+10,a)
  ++X a
if S has SetCategory then
  latex : % -> String
  ++
  ++X a:Dequeue INT:= dequeue [1,2,3,4,5]

```

```

++X latex a
hash : % -> SingleInteger
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X hash a
coerce : % -> OutputForm
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X coerce a
"=" : (%,% ) -> Boolean
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X b:Dequeue INT:= dequeue [1,2,3,4,5]
++X (a=b)@Boolean
"~=" : (%,% ) -> Boolean
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X b:=copy a
++X (a~=b)
if % has finiteAggregate then
every? : ((S -> Boolean),%) -> Boolean
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X every?(x+-(x=4),a)
any? : ((S -> Boolean),%) -> Boolean
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X any?(x+-(x=4),a)
count : ((S -> Boolean),%) -> NonNegativeInteger
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X count(x+-(x>2),a)
_# : % -> NonNegativeInteger
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X #a
parts : % -> List S
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X parts a
members : % -> List S
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X members a
if % has finiteAggregate and S has SetCategory then
member? : (S,% ) -> Boolean
++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X member?(3,a)
count : (S,% ) -> NonNegativeInteger

```

```

++
++X a:Dequeue INT:= dequeue [1,2,3,4,5]
++X count(4,a)

== Queue S add
Rep := Reference List S
bottom! d == extractBottom! d
dequeue d == ref copy d
extractBottom! d ==
  if empty? d then error "empty dequeue"
  p := deref d
  n := maxIndex p
  n = 1 =>
    r := first p
    setref(d,[])
    r
  q := rest(p,(n-2)::NonNegativeInteger)
  r := first rest q
  q.rest := []
  r
top! d == extractTop! d
extractTop! d ==
  if empty? d then error "empty dequeue"
  e := top d
  setref(d,rest deref d)
  e
height d == # deref d
depth d == # deref d
insertTop!(e,d) == (setref(d,cons(e,deref d)); e)
lastTail==> LAST$Lisp
insertBottom!(e,d) ==
  if empty? d then setref(d, list e)
  else lastTail.(deref d).rest := list e
  e
top d == if empty? d then error "empty dequeue" else first deref d
reverse! d == (setref(d,reverse deref d); d)

```

— DEQUEUE.dotabb —

```

"DEQUEUE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DEQUEUE"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"DEQUEUE" -> "FLAGG-"
"DEQUEUE" -> "FLAGG"

```

domain DERHAM DeRhamComplex

— DeRhamComplex.input —

```

)set break resume
)sys rm -f DeRhamComplex.output
)spool DeRhamComplex.output
)set message test on
)set message auto off
)clear all
--S 1 of 34
coefRing := Integer
--R
--R
--R (1) Integer
--R
--R                                          Type: Domain
--E 1

--S 2 of 34
lv : List Symbol := [x,y,z]
--R
--R
--R (2) [x,y,z]
--R
--R                                          Type: List(Symbol)
--E 2

--S 3 of 34
der := DERHAM(coefRing,lv)
--R
--R
--R (3) DeRhamComplex(Integer,[x,y,z])
--R
--R                                          Type: Domain
--E 3

--S 4 of 34
R := Expression coefRing
--R
--R
--R (4) Expression(Integer)
--R
--R                                          Type: Domain
--E 4

--S 5 of 34
f : R := x**2*y*z-5*x**3*y**2*z**5
--R
--R
--R
--R          3 2 5      2
--R (5) - 5x y z + x y z

```

```

--R                                                    Type: Expression(Integer)
--E 5

--S 6 of 34
g : R := z**2*y*cos(z)-7*sin(x**3*y**2)*z**2
--R
--R
--R      2      3 2      2
--R  (6)  - 7z sin(x y ) + y z cos(z)
--R                                                    Type: Expression(Integer)
--E 6

--S 7 of 34
h : R := x*y*z-2*x**3*y*z**2
--R
--R
--R      3      2
--R  (7)  - 2x y z + x y z
--R                                                    Type: Expression(Integer)
--E 7

--S 8 of 34
dx : der := generator(1)
--R
--R
--R  (8)  dx
--R                                                    Type: DeRhamComplex(Integer,[x,y,z])
--E 8

--S 9 of 34
dy : der := generator(2)
--R
--R
--R  (9)  dy
--R                                                    Type: DeRhamComplex(Integer,[x,y,z])
--E 9

--S 10 of 34
dz : der := generator(3)
--R
--R
--R  (10) dz
--R                                                    Type: DeRhamComplex(Integer,[x,y,z])
--E 10

--S 11 of 34
[dx,dy,dz] := [generator(i)$der for i in 1..3]
--R
--R
--R  (11)  [dx,dy,dz]

```

```

--R                                         Type: List(DeRhamComplex(Integer,[x,y,z]))
--E 11

--S 12 of 34
alpha : der := f*dx + g*dy + h*dz
--R
--R
--R (12)
--R      3 2      2 3 2      2
--R      (- 2x y z + x y z)dz + (- 7z sin(x y ) + y z cos(z))dy
--R      +
--R      3 2 5      2
--R      (- 5x y z + x y z)dx
--R                                         Type: DeRhamComplex(Integer,[x,y,z])
--E 12

--S 13 of 34
beta : der := cos(tan(x*y*z)+x*y*z)*dx + x*dy
--R
--R
--R (13) x dy + cos(tan(x y z) + x y z)dx
--R                                         Type: DeRhamComplex(Integer,[x,y,z])
--E 13

--S 14 of 34
exteriorDifferential alpha
--R
--R
--R (14)
--R      2      3 2      3 2
--R      (y z sin(z) + 14z sin(x y ) - 2y z cos(z) - 2x z + x z)dy dz
--R      +
--R      3 2 4      2 2      2
--R      (25x y z - 6x y z + y z - x y)dx dz
--R      +
--R      2 2 2      3 2      3 5      2
--R      (- 21x y z cos(x y ) + 10x y z - x z)dx dy
--R                                         Type: DeRhamComplex(Integer,[x,y,z])
--E 14

--S 15 of 34
exteriorDifferential %
--R
--R
--R (15) 0
--R                                         Type: DeRhamComplex(Integer,[x,y,z])
--E 15

--S 16 of 34
gamma := alpha * beta

```

```

--R
--R
--R (16)
--R      4 2 2      3 2
--R      (2x y z - x y z)dy dz + (2x y z - x y z)cos(tan(x y z) + x y z)dx dz
--R +
--R      2 3 2      2      4 2 5      3
--R      ((7z sin(x y ) - y z cos(z))cos(tan(x y z) + x y z) - 5x y z + x y z)dx dy
--R                                         Type: DeRhamComplex(Integer,[x,y,z])
--E 16

--S 17 of 34
exteriorDifferential(gamma) - (exteriorDifferential(alpha)*beta - alpha * exteriorDifferential(beta))
--R
--R
--R (17)  0
--R                                         Type: DeRhamComplex(Integer,[x,y,z])
--E 17

--S 18 of 34
a : BOP := operator('a)
--R
--R
--R (18)  a
--R                                         Type: BasicOperator
--E 18

--S 19 of 34
b : BOP := operator('b)
--R
--R
--R (19)  b
--R                                         Type: BasicOperator
--E 19

--S 20 of 34
c : BOP := operator('c)
--R
--R
--R (20)  c
--R                                         Type: BasicOperator
--E 20

--S 21 of 34
sigma := a(x,y,z) * dx + b(x,y,z) * dy + c(x,y,z) * dz
--R
--R
--R (21)  c(x,y,z)dz + b(x,y,z)dy + a(x,y,z)dx
--R                                         Type: DeRhamComplex(Integer,[x,y,z])
--E 21

```

```

--S 22 of 34
theta := a(x,y,z) * dx * dy + b(x,y,z) * dx * dz + c(x,y,z) * dy * dz
--R
--R
--R (22)  $c(x,y,z)dy \, dz + b(x,y,z)dx \, dz + a(x,y,z)dx \, dy$ 
--R                                         Type: DeRhamComplex(Integer,[x,y,z])
--E 22

--S 23 of 34
totalDifferential(a(x,y,z))$der
--R
--R
--R (23)  $a_{(3)}(x,y,z)dz + a_{(2)}(x,y,z)dy + a_{(1)}(x,y,z)dx$ 
--R                                         Type: DeRhamComplex(Integer,[x,y,z])
--E 23

--S 24 of 34
exteriorDifferential sigma
--R
--R
--R (24)
--R  $(c_{(2)}(x,y,z) - b_{(3)}(x,y,z))dy \, dz + (c_{(1)}(x,y,z) - a_{(3)}(x,y,z))dx \, dz$ 
--R +
--R  $(b_{(1)}(x,y,z) - a_{(2)}(x,y,z))dx \, dy$ 
--R                                         Type: DeRhamComplex(Integer,[x,y,z])
--E 24

--S 25 of 34
exteriorDifferential theta
--R
--R
--R (25)  $(c_{(1)}(x,y,z) - b_{(2)}(x,y,z) + a_{(3)}(x,y,z))dx \, dy \, dz$ 
--R                                         Type: DeRhamComplex(Integer,[x,y,z])
--E 25

--S 26 of 34
one : der := 1
--R
--R
--R (26) 1
--R                                         Type: DeRhamComplex(Integer,[x,y,z])
--E 26

--S 27 of 34
g1 : der := a([x,t,y,u,v,z,e]) * one

```

```

--R
--R
--R (27) a(x,t,y,u,v,z,e)
--R
--R                                         Type: DeRhamComplex(Integer,[x,y,z])
--E 27

```

```

--S 28 of 34
h1 : der := a([x,y,x,t,x,z,y,r,u,x]) * one
--R
--R
--R (28) a(x,y,x,t,x,z,y,r,u,x)
--R
--R                                         Type: DeRhamComplex(Integer,[x,y,z])
--E 28

```

```

--S 29 of 34
exteriorDifferential g1
--R
--R
--R (29) a (x,t,y,u,v,z,e)dz + a (x,t,y,u,v,z,e)dy + a (x,t,y,u,v,z,e)dx
--R      ,6                      ,3                      ,1
--R
--R                                         Type: DeRhamComplex(Integer,[x,y,z])
--E 29

```

```

--S 30 of 34
exteriorDifferential h1
--R
--R
--R (30)
--R      a (x,y,x,t,x,z,y,r,u,x)dz
--R      ,6
--R
--R      +
--R      (a (x,y,x,t,x,z,y,r,u,x) + a (x,y,x,t,x,z,y,r,u,x))dy
--R      ,7                      ,2
--R
--R      +
--R      a (x,y,x,t,x,z,y,r,u,x) + a (x,y,x,t,x,z,y,r,u,x)
--R      ,10                      ,5
--R
--R      +
--R      a (x,y,x,t,x,z,y,r,u,x) + a (x,y,x,t,x,z,y,r,u,x)
--R      ,3                      ,1
--R
--R      *
--R      dx
--R
--R                                         Type: DeRhamComplex(Integer,[x,y,z])
--E 30

```

```

--S 31 of 34
coefficient(gamma, dx*dy)
--R
--R
--R
--R      2      3 2      2      4 2 5      3
--R (31) (7z sin(x y ) - y z cos(z))cos(tan(x y z) + x y z) - 5x y z + x y z

```

```

--R                                                    Type: Expression(Integer)
--E 31

--S 32 of 34
coefficient(gamma, one)
--R
--R
--R (32)  0
--R
--R                                                    Type: Expression(Integer)
--E 32

--S 33 of 34
coefficient(g1,one)
--R
--R
--R (33)  a(x,t,y,u,v,z,e)
--R
--R                                                    Type: Expression(Integer)
--E 33

--S 34 of 34
gamma := alpha * beta
--R
--R
--R (34)
--R      4      2      2      3      2
--R      (2x y z - x y z)dy dz + (2x y z - x y z)cos(tan(x y z) + x y z)dx dz
--R      +
--R      2      3 2      2      4 2 5      3
--R      ((7z sin(x y ) - y z cos(z))cos(tan(x y z) + x y z) - 5x y z + x y z)dx dy
--R
--R                                                    Type: DeRhamComplex(Integer,[x,y,z])
--E 34
)spool
)lisp (bye)

```

— DeRhamComplex.help —

=====

DeRhamComplex examples

=====

The domain constructor DeRhamComplex creates the class of differential forms of arbitrary degree over a coefficient ring. The De Rham complex constructor takes two arguments: a ring, coefRing, and a list of coordinate variables.

This is the ring of coefficients.

```
coefRing := Integer
Integer
Type: Domain
```

These are the coordinate variables.

```
lv : List Symbol := [x,y,z]
[x,y,z]
Type: List Symbol
```

This is the De Rham complex of Euclidean three-space using coordinates x, y and z.

```
der := DERHAM(coefRing,lv)
DeRhamComplex(Integer,[x,y,z])
Type: Domain
```

This complex allows us to describe differential forms having expressions of integers as coefficients. These coefficients can involve any number of variables, for example, $f(x,t,r,y,u,z)$. As we've chosen to work with ordinary Euclidean three-space, expressions involving these forms are treated as functions of x, y and z with the additional arguments t, r and u regarded as symbolic constants.

Here are some examples of coefficients.

```
R := Expression coefRing
Expression Integer
Type: Domain

f : R := x**2*y*z-5*x**3*y**2*z**5
      3 2 5      2
- 5x y z + x y z
Type: Expression Integer

g : R := z**2*y*cos(z)-7*sin(x**3*y**2)*z**2
      2      3 2      2
- 7z sin(x y ) + y z cos(z)
Type: Expression Integer

h : R :=x*y*z-2*x**3*y*z**2
      3      2
- 2x y z + x y z
Type: Expression Integer
```

We now define the multiplicative basis elements for the exterior algebra over R.

```
dx : der := generator(1)
```



```

dx
                                     Type: DeRhamComplex(Integer,[x,y,z])

dy : der := generator(2)
dy
                                     Type: DeRhamComplex(Integer,[x,y,z])

dz : der := generator(3)
dz
                                     Type: DeRhamComplex(Integer,[x,y,z])

```

This is an alternative way to give the above assignments.

```

[dx,dy,dz] := [generator(i)$der for i in 1..3]
[dx,dy,dz]
                                     Type: List DeRhamComplex(Integer,[x,y,z])

```

Now we define some one-forms.

```

alpha : der := f*dx + g*dy + h*dz
          3 2          2 3 2          2
          (- 2x y z + x y z)dz + (- 7z sin(x y ) + y z cos(z))dy
+
          3 2 5 2
          (- 5x y z + x y z)dx
                                     Type: DeRhamComplex(Integer,[x,y,z])

beta : der := cos(tan(x*y*z)+x*y*z)*dx + x*dy
          x dy + cos(tan(x y z) + x y z)dx
                                     Type: DeRhamComplex(Integer,[x,y,z])

```

A well-known theorem states that the composition of exteriorDifferential with itself is the zero map for continuous forms. Let's verify this theorem for alpha.

```

exteriorDifferential alpha
          2          3 2          3 2
          (y z sin(z) + 14z sin(x y ) - 2y z cos(z) - 2x z + x z)dy dz
+
          3 2 4 2 2 2
          (25x y z - 6x y z + y z - x y)dx dz
+
          2 2 2 3 2 3 5 2
          (- 21x y z cos(x y ) + 10x y z - x z)dx dy
                                     Type: DeRhamComplex(Integer,[x,y,z])

```

We see a lengthy output of the last expression, but nevertheless, the composition is zero.

```

exteriorDifferential %

```

0

Type: DeRhamComplex(Integer,[x,y,z])

Now we check that exteriorDifferential is a "graded derivation" D,
that is, D satisfies:

$$D(a*b) = D(a)*b + (-1)**degree(a)*a*D(b)$$

```
gamma := alpha * beta
      4 2 2      3 2
      (2x y z - x y z)dy dz + (2x y z - x y z)cos(tan(x y z) + x y z)dx dz
+
      2 3 2      2      4 2 5      3
      ((7z sin(x y ) - y z cos(z))cos(tan(x y z) + x y z) - 5x y z + x y z)dx dy
      Type: DeRhamComplex(Integer,[x,y,z])
```

We try this for the one-forms alpha and beta.

```
exteriorDifferential(gamma) - (exteriorDifferential(alpha)*beta - alpha * exteriorDifferential(beta))
0
      Type: DeRhamComplex(Integer,[x,y,z])
```

Now we define some "basic operators"

```
a : BOP := operator('a)
a
      Type: BasicOperator
```

```
b : BOP := operator('b)
b
      Type: BasicOperator
```

```
c : BOP := operator('c)
c
      Type: BasicOperator
```

We also define some indeterminate one- and two-forms using these operators.

```
sigma := a(x,y,z) * dx + b(x,y,z) * dy + c(x,y,z) * dz
      c(x,y,z)dz + b(x,y,z)dy + a(x,y,z)dx
      Type: DeRhamComplex(Integer,[x,y,z])

theta := a(x,y,z) * dx * dy + b(x,y,z) * dx * dz + c(x,y,z) * dy * dz
      c(x,y,z)dy dz + b(x,y,z)dx dz + a(x,y,z)dx dy
      Type: DeRhamComplex(Integer,[x,y,z])
```

This allows us to get formal definitions for the "gradient" ...

```
totalDifferential(a(x,y,z))$der
```

```

(23)  a (x,y,z)dz + a (x,y,z)dy + a (x,y,z)dx
      ,3          ,2          ,1
      Type: DeRhamComplex(Integer,[x,y,z])
the "curl" ...

exteriorDifferential sigma
(c (x,y,z) - b (x,y,z))dy dz + (c (x,y,z) - a (x,y,z))dx dz
  ,2          ,3          ,1          ,3
+
(b (x,y,z) - a (x,y,z))dx dy
  ,1          ,2
      Type: DeRhamComplex(Integer,[x,y,z])

and the "divergence."

```

```

exteriorDifferential theta
(c (x,y,z) - b (x,y,z) + a (x,y,z))dx dy dz
  ,1          ,2          ,3
      Type: DeRhamComplex(Integer,[x,y,z])

```

Note that the De Rham complex is an algebra with unity. This element 1 is the basis for elements for zero-forms, that is, functions in our space.

```

one : der := 1
1
      Type: DeRhamComplex(Integer,[x,y,z])

```

To convert a function to a function lying in the De Rham complex, multiply the function by "one."

```

g1 : der := a([x,t,y,u,v,z,e]) * one
a(x,t,y,u,v,z,e)
      Type: DeRhamComplex(Integer,[x,y,z])

```

A current limitation of Axiom forces you to write functions with more than four arguments using square brackets in this way.

```

h1 : der := a([x,y,x,t,x,z,y,r,u,x]) * one
a(x,y,x,t,x,z,y,r,u,x)
      Type: DeRhamComplex(Integer,[x,y,z])

```

Now note how the system keeps track of where your coordinate functions are located in expressions.

```

exteriorDifferential g1
a (x,t,y,u,v,z,e)dz + a (x,t,y,u,v,z,e)dy + a (x,t,y,u,v,z,e)dx
  ,6          ,3          ,1
      Type: DeRhamComplex(Integer,[x,y,z])

```

```

exteriorDifferential h1
  a (x,y,x,t,x,z,y,r,u,x)dz
    ,6
+
  (a (x,y,x,t,x,z,y,r,u,x) + a (x,y,x,t,x,z,y,r,u,x))dy
    ,7                                ,2
+
  a (x,y,x,t,x,z,y,r,u,x) + a (x,y,x,t,x,z,y,r,u,x)
    ,10                                ,5
+
  a (x,y,x,t,x,z,y,r,u,x) + a (x,y,x,t,x,z,y,r,u,x)
    ,3                                ,1
*
dx

```

Type: DeRhamComplex(Integer,[x,y,z])

In this example of Euclidean three-space, the basis for the De Rham complex consists of the eight forms: 1, dx, dy, dz, dx*dy, dx*dz, dy*dz, and dx*dy*dz.

```

coefficient(gamma, dx*dy)
  2      3 2      2      4 2 5      3
  (7z sin(x y ) - y z cos(z))cos(tan(x y z) + x y z) - 5x y z + x y z

```

Type: Expression Integer

```

coefficient(gamma, one)
0

```

Type: Expression Integer

```

coefficient(g1,one)
a(x,t,y,u,v,z,e)

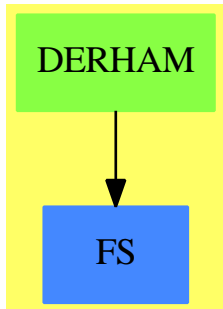
```

Type: Expression Integer

See Also:

- o)help Operator
- o)show DeRhamComplex

DeRhamComplex (DERHAM)



See

⇒ “ExtAlgBasis” (EAB) [6](#) on page [743](#)

⇒ “AntiSymm” (ANTISYM) [2](#) on page [40](#)

Exports:

0	1	characteristic	coefficient
coerce	degree	exteriorDifferential	generator
hash	homogeneous?	latex	leadingBasisTerm
leadingCoefficient	map	one?	recip
reductum	retract	retractable?	retractIfCan
sample	subtractIfCan	totalDifferential	zero?
?~=?	?*?	?**?	?^?
?+?	?-?	-?	?=?

— domain DERHAM DeRhamComplex —

```
)abbrev domain DERHAM DeRhamComplex
++ Author: Larry A. Lambe
++ Date   : 01/26/91.
++ Revised : 12/01/91.
++ Description:
++ The deRham complex of Euclidean space, that is, the
++ class of differential forms of arbitrary degree over a coefficient ring.
++ See Flanders, Harley, Differential Forms, With Applications to the Physical
++ Sciences, New York, Academic Press, 1963.
```

```
DeRhamComplex(CoefRing,listIndVar>List Symbol): Export == Implement where
  CoefRing : Join(Ring, OrderedSet)
  ASY      ==> AntiSymm(R,listIndVar)
  DIFRING ==> DifferentialRing
  LALG     ==> LeftAlgebra
  FMR      ==> FreeMod(R,EAB)
  I        ==> Integer
  L        ==> List
  EAB      ==> ExtAlgBasis -- these are exponents of basis elements in order
```

```

NNI      ==> NonNegativeInteger
0        ==> OutputForm
R        ==> Expression(CoefRing)

Export == Join(LALG(R), RetractableTo(R)) with
  leadingCoefficient : %          -> R
    ++ leadingCoefficient(df) returns the leading
    ++ coefficient of differential form df.
  leadingBasisTerm   : %          -> %
    ++ leadingBasisTerm(df) returns the leading
    ++ basis term of differential form df.
  reductum           : %          -> %
    ++ reductum(df), where df is a differential form,
    ++ returns df minus the leading
    ++ term of df if df has two or more terms, and
    ++ 0 otherwise.
  coefficient         : (%,% )    -> R
    ++ coefficient(df,u), where df is a differential form,
    ++ returns the coefficient of df containing the basis term u
    ++ if such a term exists, and 0 otherwise.
  generator          : NNI        -> %
    ++ generator(n) returns the nth basis term for a differential form.
  homogeneous?       : %          -> Boolean
    ++ homogeneous?(df) tests if all of the terms of
    ++ differential form df have the same degree.
  retractable?       : %          -> Boolean
    ++ retractable?(df) tests if differential form df is a 0-form,
    ++ i.e., if degree(df) = 0.
  degree             : %          -> I
    ++ degree(df) returns the homogeneous degree of differential form df.
  map                : (R -> R, %) -> %
    ++ map(f,df) replaces each coefficient x of differential
    ++ form df by \spad{f(x)}.
  totalDifferential   : R -> %
    ++ totalDifferential(x) returns the total differential
    ++ (gradient) form for element x.
  exteriorDifferential : % -> %
    ++ exteriorDifferential(df) returns the exterior
    ++ derivative (gradient, curl, divergence, ...) of
    ++ the differential form df.

Implement == ASY add
  Rep := ASY

  dim := #listIndVar

  totalDifferential(f) ==
    divs:=[differentiate(f,listIndVar.i)*generator(i)$ASY for i in 1..dim]
    reduce("+",divs)

```

```

termDiff : (R, %) -> %
termDiff(r,e) ==
  totalDifferential(r) * e

exteriorDifferential(x) ==
  x = 0 => 0
  termDiff(leadingCoefficient(x)$Rep,leadingBasisTerm x) + exteriorDifferential(reductum x)

lv := [concat("d",string(liv))$String::Symbol for liv in listIndVar]

displayList:EAB -> 0
displayList(x):0 ==
  le: L I := exponents(x)$EAB
--   reduce(*,[lv.i]::0 for i in 1..dim | le.i = 1])$L(0)
--   reduce(*,[lv.i]::0 for i in 1..dim | one?(le.i))$L(0)
  reduce(*,[lv.i]::0 for i in 1..dim | ((le.i) = 1))$L(0)

makeTerm:(R,EAB) -> 0
makeTerm(r,x) ==
-- we know that r ^= 0
  x = Nul(dim)$EAB => r::0
--   one? r => displayList(x)
  (r = 1) => displayList(x)
--   r = 1 => displayList(x)
  r::0 * displayList(x)

terms : % -> List Record(k: EAB, c: R)
terms(a) ==
  -- it is the case that there are at least two terms in a
  a pretend List Record(k: EAB, c: R)

coerce(a):0 ==
  a = 0$Rep => 0$I::0
  ta := terms a
--   reductum(a) = 0$Rep => makeTerm(leadingCoefficient a, a.first.k)
  null ta.rest => makeTerm(ta.first.c, ta.first.k)
  reduce(+,[makeTerm(t.c,t.k) for t in ta])$L(0)

```

— DERHAM.dotabb —

```

"DERHAM" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DERHAM"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"DERHAM" -> "FS"

```

domain DSTREE DesingTree

— DesingTree.input —

```

)set break resume
)sys rm -f DesingTree.output
)spool DesingTree.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DesingTree
--R
--R DesingTree(S: SetCategory) is a domain constructor
--R Abbreviation for DesingTree is DSTREE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for DSTREE
--R
--R----- Operations -----
--R children : % -> List(%)          copy : % -> %
--R cyclic? : % -> Boolean           distance : (%,%) -> Integer
--R ?.value : (%,value) -> S         empty : () -> %
--R empty? : % -> Boolean            encode : % -> String
--R eq? : (%,%) -> Boolean           fullOut : % -> OutputForm
--R fullOutput : () -> Boolean       fullOutput : Boolean -> Boolean
--R leaf? : % -> Boolean             leaves : % -> List(S)
--R map : ((S -> S),%) -> %          nodes : % -> List(%)
--R sample : () -> %                tree : List(S) -> %
--R tree : S -> %                   tree : (S,List(%)) -> %
--R value : % -> S
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ==? : (%,%) -> Boolean if S has SETCAT
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R child? : (%,%) -> Boolean if S has SETCAT
--R coerce : % -> OutputForm if S has SETCAT
--R count : (S,%) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R eval : (%,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R hash : % -> SingleInteger if S has SETCAT
--R latex : % -> String if S has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R member? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT

```



```

--R members : % -> List(S) if $ has finiteAggregate
--R more? : (% , NonNegativeInteger) -> Boolean
--R node? : (% , %) -> Boolean if S has SETCAT
--R parts : % -> List(S) if $ has finiteAggregate
--R setchildren! : (% , List(%)) -> % if $ has shallowlyMutable
--R setelt : (% , value, S) -> S if $ has shallowlyMutable
--R setvalue! : (% , S) -> S if $ has shallowlyMutable
--R size? : (% , NonNegativeInteger) -> Boolean
--R ~=? : (% , %) -> Boolean if S has SETCAT
--R
--E 1

)spool
)lisp (bye)

```

— DesingTree.help —

=====

DesingTree examples

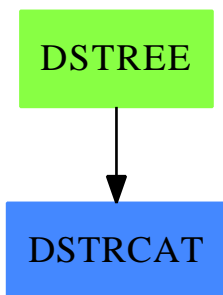
=====

This category is part of the PAFF package

See Also:

- o)show DesingTree

DesingTree (DSTREE)



Exports:

#?	?value	?=?	?~=?	any?
child?	children	coerce	copy	count
cyclic?	distance	empty	empty?	encode
eq?	eval	every?	fullOut	fullOutput
hash	latex	leaf?	leaves	less?
map	map!	member?	members	more?
node?	nodes	parts	sample	setchildren!
setelt	setvalue!	size?	tree	value

— domain DSTREE DesingTree —

```

)abbrev domain DSTREE DesingTree
++ Authors: Gaetan Hache
++ Date Created: jan 1998
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ This category is part of the PAFF package
DesingTree(S: SetCategory): T==C where

T == DesingTreeCategory(S) with

  encode: % -> String
    ++ encode(t) returns a string indicating the "shape" of the tree

  fullOut: % -> OutputForm
    ++ fullOut(tr) yields a full output of tr (see function fullOutput).

  fullOutput: Boolean -> Boolean
    ++ fullOutput(b) sets a flag such that when true,
    ++ a coerce to OutputForm yields the full output of
    ++ tr, otherwise encode(tr) is output (see encode function).
    ++ The default is false.

  fullOutput: () -> Boolean
    ++ fullOutput returns the value of the flag set by fullOutput(b).

C == add
Rep ==> Record(value: S, args: List %)

fullOut(t:%): OutputForm ==
  empty? children t => (value t) ::OutputForm
  prefix((value t)::OutputForm, [fullOut(tr) for tr in children t])

fullOutputFlag:Boolean:=false()

fullOutput(f)== fullOutputFlag:=f

fullOutput == fullOutputFlag

```

```

leaves(t)==
  empty?(chdr:=children(t)) => list(value(t))
  concat([leaves(subt) for subt in chdr])

t1=t2 == value t1 = value t2 and children t1 = children t2

coerce(t:%):OutputForm==
  ^fullOutput() => encode(t) :: OutputForm
  fullOut(t)

tree(s,ls) == ([s,ls]:Rep):%

tree(s:S) == ([s,[]]:Rep):%

tree(ls:List(S))==
  empty?(ls) =>
    error "Cannot create a tree with an empty list"
  f:=first(ls)
  empty?(rs:=rest(ls)) =>
    tree(f)
  tree(f,[tree(rs)])

value t == (t:Rep).value

children t == ((t:Rep).args):List %

setchildren_!(t,ls) == ((t:Rep).args:=ls;t pretend %)

setvalue_!(t,s) == ((t:Rep).value:=s;s)

encode(t)==
  empty?(chtr:=children(t)) => empty()$String
  concat([concat(["U",encode(arb),"."]) for arb in chtr])

```

— DSTREE.dotabb —

```

"DSTREE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DSTREE"];
"DSTRCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DSTRCAT"]
"DSTREE" -> "DSTRCAT"

```

domain DSMP DifferentialSparseMultivariatePolynomial

— DifferentialSparseMultivariatePolynomial.input —

```

)set break resume
)sys rm -f DifferentialSparseMultivariatePolynomial.output
)spool DifferentialSparseMultivariatePolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DifferentialSparseMultivariatePolynomial
--R
--R DifferentialSparseMultivariatePolynomial(R: Ring,S: OrderedSet,V: DifferentialVariableCategory(S)) is a
--R Abbreviation for DifferentialSparseMultivariatePolynomial is DSMP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for DSMP
--R
--R----- Operations -----
--R ??? : (%,R) -> %          ??? : (R,%) -> %
--R ??? : (%,%) -> %          ??? : (Integer,%) -> %
--R ??? : (PositiveInteger,%) -> %    ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %          ?-? : (%,%) -> %
--R -? : % -> %              ?=? : (%,%) -> Boolean
--R D : (%,(R -> R)) -> %        D : % -> % if R has DIFRING
--R D : (%,List(V)) -> %        D : (%,V) -> %
--R 1 : () -> %                0 : () -> %
--R ?? : (%,PositiveInteger) -> %    coefficients : % -> List(R)
--R coerce : S -> %              coerce : V -> %
--R coerce : R -> %              coerce : Integer -> %
--R coerce : % -> OutputForm      degree : % -> IndexedExponents(V)
--R differentiate : (%,List(V)) -> % differentiate : (%,V) -> %
--R eval : (%,List(V),List(%)) -> % eval : (%,V,%) -> %
--R eval : (%,List(V),List(R)) -> % eval : (%,V,R) -> %
--R eval : (%,%,%) -> %          eval : (%,Equation(%)) -> %
--R ground : % -> R              ground? : % -> Boolean
--R hash : % -> SingleInteger      initial : % -> %
--R isobaric? : % -> Boolean        latex : % -> String
--R leader : % -> V                leadingCoefficient : % -> R
--R leadingMonomial : % -> %        map : ((R -> R),%) -> %
--R monomial? : % -> Boolean        monomials : % -> List(%)
--R one? : % -> Boolean            order : % -> NonNegativeInteger
--R recip : % -> Union(%, "failed") reductum : % -> %
--R retract : % -> S              retract : % -> V
--R retract : % -> R              sample : () -> %
--R separant : % -> %              variables : % -> List(V)
--R weight : % -> NonNegativeInteger zero? : % -> Boolean

```

```

--R ?^=? : (%,% ) -> Boolean
--R ?*? : (Fraction(Integer),%) -> % if R has ALGEBRA(FRAC(INT))
--R ?*? : (% ,Fraction(Integer)) -> % if R has ALGEBRA(FRAC(INT))
--R ?*? : (NonNegativeInteger,%) -> %
--R ??? : (% ,NonNegativeInteger) -> %
--R ?/? : (% ,R) -> % if R has FIELD
--R ?<? : (% ,%) -> Boolean if R has ORDSET
--R ?<=? : (% ,%) -> Boolean if R has ORDSET
--R ?>? : (% ,%) -> Boolean if R has ORDSET
--R ?>=? : (% ,%) -> Boolean if R has ORDSET
--R D : (% ,(R -> R),NonNegativeInteger) -> %
--R D : (% ,List(Symbol),List(NonNegativeInteger)) -> % if R has PDRING(SYMBOL)
--R D : (% ,Symbol,NonNegativeInteger) -> % if R has PDRING(SYMBOL)
--R D : (% ,List(Symbol)) -> % if R has PDRING(SYMBOL)
--R D : (% ,Symbol) -> % if R has PDRING(SYMBOL)
--R D : (% ,NonNegativeInteger) -> % if R has DIFRING
--R D : (% ,List(V),List(NonNegativeInteger)) -> %
--R D : (% ,V,NonNegativeInteger) -> %
--R ?^? : (% ,NonNegativeInteger) -> %
--R associates? : (% ,%) -> Boolean if R has INTDOM
--R binomThmExpt : (% ,%,NonNegativeInteger) -> % if R has COMRING
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(% ,"failed") if $ has CHARNZ and R has PFECAT or R has CHARNZ
--R coefficient : (% ,List(V),List(NonNegativeInteger)) -> %
--R coefficient : (% ,V,NonNegativeInteger) -> %
--R coefficient : (% ,IndexedExponents(V)) -> R
--R coerce : % -> % if R has INTDOM
--R coerce : Fraction(Integer) -> % if R has ALGEBRA(FRAC(INT)) or R has RETRACT(FRAC(INT))
--R coerce : SparseMultivariatePolynomial(R,S) -> %
--R conditionP : Matrix(% ) -> Union(Vector(% ),"failed") if $ has CHARNZ and R has PFECAT
--R content : (% ,V) -> % if R has GCDDOM
--R content : % -> R if R has GCDDOM
--R convert : % -> InputForm if R has KONVERT(INFORM) and V has KONVERT(INFORM)
--R convert : % -> Pattern(Integer) if R has KONVERT(PATTERN(INT)) and V has KONVERT(PATTERN(INT))
--R convert : % -> Pattern(Float) if R has KONVERT(PATTERN(FLOAT)) and V has KONVERT(PATTERN(FLOAT))
--R degree : (% ,S) -> NonNegativeInteger
--R degree : (% ,List(V)) -> List(NonNegativeInteger)
--R degree : (% ,V) -> NonNegativeInteger
--R differentialVariables : % -> List(S)
--R differentiate : (% ,(R -> R)) -> %
--R differentiate : (% ,(R -> R),NonNegativeInteger) -> %
--R differentiate : (% ,List(Symbol),List(NonNegativeInteger)) -> % if R has PDRING(SYMBOL)
--R differentiate : (% ,Symbol,NonNegativeInteger) -> % if R has PDRING(SYMBOL)
--R differentiate : (% ,List(Symbol)) -> % if R has PDRING(SYMBOL)
--R differentiate : (% ,Symbol) -> % if R has PDRING(SYMBOL)
--R differentiate : (% ,NonNegativeInteger) -> % if R has DIFRING
--R differentiate : % -> % if R has DIFRING
--R differentiate : (% ,List(V),List(NonNegativeInteger)) -> %
--R differentiate : (% ,V,NonNegativeInteger) -> %
--R discriminant : (% ,V) -> % if R has COMRING

```

```

--R eval : (% ,List(S),List(R)) -> % if R has DIFRING
--R eval : (% ,S,R) -> % if R has DIFRING
--R eval : (% ,List(S),List(%)) -> % if R has DIFRING
--R eval : (% ,S,%) -> % if R has DIFRING
--R eval : (% ,List(%),List(%)) -> %
--R eval : (% ,List(Equation(%))) -> %
--R exquo : (% ,%) -> Union(%,"failed") if R has INTDOM
--R exquo : (% ,R) -> Union(%,"failed") if R has INTDOM
--R factor : % -> Factored(%) if R has PFECAT
--R factorPolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if R has PF
--R factorSquareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if
--R gcd : (% ,%) -> % if R has GCDDOM
--R gcd : List(%) -> % if R has GCDDOM
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynom
--R isExpt : % -> Union(Record(var: V,exponent: NonNegativeInteger),"failed")
--R isPlus : % -> Union(List(%),"failed")
--R isTimes : % -> Union(List(%),"failed")
--R lcm : (% ,%) -> % if R has GCDDOM
--R lcm : List(%) -> % if R has GCDDOM
--R mainVariable : % -> Union(V,"failed")
--R makeVariable : % -> (NonNegativeInteger -> %) if R has DIFRING
--R makeVariable : S -> (NonNegativeInteger -> %)
--R mapExponents : ((IndexedExponents(V) -> IndexedExponents(V)),%) -> %
--R max : (% ,%) -> % if R has ORDSET
--R min : (% ,%) -> % if R has ORDSET
--R minimumDegree : (% ,List(V)) -> List(NonNegativeInteger)
--R minimumDegree : (% ,V) -> NonNegativeInteger
--R minimumDegree : % -> IndexedExponents(V)
--R monicDivide : (% ,%,V) -> Record(quotient: %,remainder: %)
--R monomial : (% ,List(V),List(NonNegativeInteger)) -> %
--R monomial : (% ,V,NonNegativeInteger) -> %
--R monomial : (R,IndexedExponents(V)) -> %
--R multivariate : (SparseUnivariatePolynomial(%),V) -> %
--R multivariate : (SparseUnivariatePolynomial(R),V) -> %
--R numberOfMonomials : % -> NonNegativeInteger
--R order : (% ,S) -> NonNegativeInteger
--R patternMatch : (% ,Pattern(Integer),PatternMatchResult(Integer,%)) -> PatternMatchResult(Integer,%) if R
--R patternMatch : (% ,Pattern(Float),PatternMatchResult(Float,%)) -> PatternMatchResult(Float,%) if R has P
--R pomopo! : (% ,R,IndexedExponents(V),%) -> %
--R prime? : % -> Boolean if R has PFECAT
--R primitiveMonomials : % -> List(%)
--R primitivePart : (% ,V) -> % if R has GCDDOM
--R primitivePart : % -> % if R has GCDDOM
--R reducedSystem : Matrix(%) -> Matrix(R)
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(R),vec: Vector(R))
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) if R has LIN
--R reducedSystem : Matrix(%) -> Matrix(Integer) if R has LINEXP(INT)
--R resultant : (% ,%,V) -> % if R has COMRING
--R retract : % -> SparseMultivariatePolynomial(R,S)
--R retract : % -> Integer if R has RETRACT(INT)

```

```

--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(SparseMultivariatePolynomial(R,S),"failed")
--R retractIfCan : % -> Union(S,"failed")
--R retractIfCan : % -> Union(V,"failed")
--R retractIfCan : % -> Union(Integer,"failed") if R has RETRACT(INT)
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(R,"failed")
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(%)),SparseUnivariatePolynomial(%)) -> List(SparseUnivariatePolynomial(%))
--R squareFree : % -> Factored(%) if R has GCDDOM
--R squareFreePart : % -> % if R has GCDDOM
--R squareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%))
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R totalDegree : (%,List(V)) -> NonNegativeInteger
--R totalDegree : % -> NonNegativeInteger
--R unit? : % -> Boolean if R has INTDOM
--R unitCanonical : % -> % if R has INTDOM
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %) if R has INTDOM
--R univariate : % -> SparseUnivariatePolynomial(R)
--R univariate : (%,V) -> SparseUnivariatePolynomial(%)
--R weight : (%,S) -> NonNegativeInteger
--R weights : (%,S) -> List(NonNegativeInteger)
--R weights : % -> List(NonNegativeInteger)
--R
--E 1

)spool
)lisp (bye)

```

— DifferentialSparseMultivariatePolynomial.help —

```

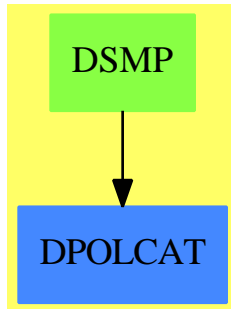
=====
DifferentialSparseMultivariatePolynomial examples
=====

```

DifferentialSparseMultivariatePolynomial implements an ordinary differential polynomial ring by combining a domain belonging to the category DifferentialVariableCategory with the domain SparseMultivariatePolynomial.

See Also:

o)show DifferentialSparseMultivariatePolynomial

DifferentialSparseMultivariatePolynomial (DSMP)**See**

- ⇒ “OrderlyDifferentialVariable” (ODVAR) [16](#) on page [1855](#)
- ⇒ “SequentialDifferentialVariable” (SDVAR) [20](#) on page [2389](#)
- ⇒ “OrderlyDifferentialPolynomial” (ODPOL) [16](#) on page [1851](#)
- ⇒ “SequentialDifferentialPolynomial” (SDPOL) [20](#) on page [2385](#)

Exports:

0	1	associates?
binomThmExpt	characteristic	charthRoot
coefficient	coefficients	coerce
conditionP	content	convert
D	degree	differentialVariables
differentiate	discriminant	eval
exquo	factor	factorPolynomial
factorSquareFreePolynomial	gcd	gcdPolynomial
ground	ground?	hash
initial	isExpt	isobaric?
isPlus	isTimes	latex
lcm	leader	leadingCoefficient
leadingMonomial	makeVariable	map
mapExponents	max	min
minimumDegree	monicDivide	monomial
monomials	monomial?	multivariate
numberOfMonomials	one?	order
patternMatch	pomopo!	prime?
primitiveMonomials	primitivePart	recip
reducedSystem	reductum	resultant
retract	retractIfCan	sample
separant	solveLinearPolynomialEquation	squareFree
squareFreePart	squareFreePolynomial	subtractIfCan
totalDegree	unit?	unitCanonical
unitNormal	univariate	variables
weight	weights	zero?
?*?	?**?	?+?
?-?	-?	?=?
?^?	?~=?	?/?
?<?	?<=?	?>?
?>=?		

— domain DSMP DifferentialSparseMultivariatePolynomial —

```

)abbrev domain DSMP DifferentialSparseMultivariatePolynomial
++ Author: William Sit
++ Date Created: 19 July 1990
++ Date Last Updated: 13 September 1991
++ References: Kolchin, E.R. "Differential Algebra and Algebraic Groups"
++ (Academic Press, 1973).
++ Description:
++ \spadtype{DifferentialSparseMultivariatePolynomial} implements
++ an ordinary differential polynomial ring by combining a
++ domain belonging to the category \spadtype{DifferentialVariableCategory}
++ with the domain \spadtype{SparseMultivariatePolynomial}.

```

```
DifferentialSparseMultivariatePolynomial(R, S, V):
```

```

Exports == Implementation where
R: Ring
S: OrderedSet
V: DifferentialVariableCategory S
E ==> IndexedExponents(V)
PC ==> PolynomialCategory(R,IndexedExponents(V),V)
PCL ==> PolynomialCategoryLifting
P ==> SparseMultivariatePolynomial(R, V)
SUP ==> SparseUnivariatePolynomial
SMP ==> SparseMultivariatePolynomial(R, S)

Exports ==> Join(DifferentialPolynomialCategory(R,S,V,E),
                 RetractableTo SMP)

Implementation ==> P add
  retractIfCan(p:$):Union(SMP, "failed") ==
    zero? order p =>
      map(x+>retract(x)@S :: SMP,y+>y::SMP, p)$PCL(
        IndexedExponents V, V, R, $, SMP)
    "failed"

  coerce(p:SMP):$ ==
    map(x+>x::V::$, y+>y::$, p)$PCL(IndexedExponents S, S, R, SMP, $)

```

— DSMP.dotabb —

```

"DSMP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DSMP"]
"DPOLCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DPOLCAT"]
"DSMP" -> "DPOLCAT"

```

domain DIRPROD DirectProduct

— DirectProduct.input —

```

)set break resume
)sys rm -f DirectProduct.output
)spool DirectProduct.output
)set message test on
)set message auto off
)clear all

```

```

--S 1 of 1
)show DirectProduct
--R DirectProduct(dim: NonNegativeInteger,R: Type) is a domain constructor
--R Abbreviation for DirectProduct is DIRPROD
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for DIRPROD
--R
--R----- Operations -----
--R -? : % -> % if R has RING          1 : () -> % if R has MONOID
--R 0 : () -> % if R has CABMON        coerce : % -> Vector(R)
--R copy : % -> %                      directProduct : Vector(R) -> %
--R ?.? : (%,Integer) -> R             elt : (%,Integer,R) -> R
--R empty : () -> %                   empty? : % -> Boolean
--R entries : % -> List(R)             eq? : (%,% ) -> Boolean
--R index? : (Integer,% ) -> Boolean    indices : % -> List(Integer)
--R map : ((R -> R),%) -> %            qelt : (%,Integer) -> R
--R sample : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?? : (PositiveInteger,% ) -> % if R has ABELSG
--R ?? : (NonNegativeInteger,% ) -> % if R has CABMON
--R ?? : (R,% ) -> % if R has MONOID
--R ?? : (% ,R) -> % if R has MONOID
--R ?? : (% ,%) -> % if R has MONOID
--R ?? : (Integer,% ) -> % if R has RING
--R ??? : (% ,PositiveInteger) -> % if R has MONOID
--R ??? : (% ,NonNegativeInteger) -> % if R has MONOID
--R ?? : (% ,%) -> % if R has ABELSG
--R ?? : (% ,%) -> % if R has RING
--R ?? : (% ,R) -> % if R has FIELD
--R ?? : (% ,%) -> Boolean if R has OAMONS or R has ORDRING
--R ?? : (% ,%) -> Boolean if R has OAMONS or R has ORDRING
--R ?? : (% ,%) -> Boolean if R has SETCAT
--R ?? : (% ,%) -> Boolean if R has OAMONS or R has ORDRING
--R ?? : (% ,%) -> Boolean if R has OAMONS or R has ORDRING
--R D : (% ,(R -> R)) -> % if R has RING
--R D : (% ,(R -> R),NonNegativeInteger) -> % if R has RING
--R D : (% ,List(Symbol),List(NonNegativeInteger)) -> % if R has PDRING(SYMBOL) and R has RING
--R D : (% ,Symbol,NonNegativeInteger) -> % if R has PDRING(SYMBOL) and R has RING
--R D : (% ,List(Symbol)) -> % if R has PDRING(SYMBOL) and R has RING
--R D : (% ,Symbol) -> % if R has PDRING(SYMBOL) and R has RING
--R D : (% ,NonNegativeInteger) -> % if R has DIFRING and R has RING
--R D : % -> % if R has DIFRING and R has RING
--R ?? : (% ,PositiveInteger) -> % if R has MONOID
--R ?? : (% ,NonNegativeInteger) -> % if R has MONOID
--R abs : % -> % if R has ORDRING
--R any? : ((R -> Boolean),%) -> Boolean if $ has finiteAggregate
--R characteristic : () -> NonNegativeInteger if R has RING
--R coerce : R -> % if R has SETCAT
--R coerce : Fraction(Integer) -> % if R has RETRACT(FRAC(INT)) and R has SETCAT

```

```

--R coerce : Integer -> % if R has RETRACT(INT) and R has SETCAT or R has RING
--R coerce : % -> OutputForm if R has SETCAT
--R count : (R,%) -> NonNegativeInteger if $ has finiteAggregate and R has SETCAT
--R count : ((R -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R differentiate : (%,(R -> R)) -> % if R has RING
--R differentiate : (%,(R -> R),NonNegativeInteger) -> % if R has RING
--R differentiate : (%,List(Symbol),List(NonNegativeInteger)) -> % if R has PDRING(SYMBOL) and R has RING
--R differentiate : (%,Symbol,NonNegativeInteger) -> % if R has PDRING(SYMBOL) and R has RING
--R differentiate : (%,List(Symbol)) -> % if R has PDRING(SYMBOL) and R has RING
--R differentiate : (%,Symbol) -> % if R has PDRING(SYMBOL) and R has RING
--R differentiate : (%,NonNegativeInteger) -> % if R has DIFRING and R has RING
--R differentiate : % -> % if R has DIFRING and R has RING
--R dimension : () -> CardinalNumber if R has FIELD
--R dot : (%,%) -> R if R has RING
--R entry? : (R,%) -> Boolean if $ has finiteAggregate and R has SETCAT
--R enumerate : () -> List(%) if R has FINITE
--R eval : (%,List(R),List(R)) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (%,R,R) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (%,Equation(R)) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (%,List(Equation(R))) -> % if R has EVALAB(R) and R has SETCAT
--R every? : ((R -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (%,R) -> % if $ has shallowlyMutable
--R first : % -> R if Integer has ORDSET
--R hash : % -> SingleInteger if R has SETCAT
--R index : PositiveInteger -> % if R has FINITE
--R latex : % -> String if R has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R lookup : % -> PositiveInteger if R has FINITE
--R map! : ((R -> R),%) -> % if $ has shallowlyMutable
--R max : (%,%) -> % if R has OAMONS or R has ORDRING
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (R,%) -> Boolean if $ has finiteAggregate and R has SETCAT
--R members : % -> List(R) if $ has finiteAggregate
--R min : (%,%) -> % if R has OAMONS or R has ORDRING
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (%,NonNegativeInteger) -> Boolean
--R negative? : % -> Boolean if R has ORDRING
--R one? : % -> Boolean if R has MONOID
--R parts : % -> List(R) if $ has finiteAggregate
--R positive? : % -> Boolean if R has ORDRING
--R qsetelt! : (%,Integer,R) -> R if $ has shallowlyMutable
--R random : () -> % if R has FINITE
--R recip : % -> Union(%, "failed") if R has MONOID
--R reducedSystem : Matrix(%) -> Matrix(R) if R has RING
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(R),vec: Vector(R)) if R has RING
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) if R has LIN
--R reducedSystem : Matrix(%) -> Matrix(Integer) if R has LINEXP(INT) and R has RING
--R retract : % -> R if R has SETCAT
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT)) and R has SETCAT
--R retract : % -> Integer if R has RETRACT(INT) and R has SETCAT

```

```

--R retractIfCan : % -> Union(R,"failed") if R has SETCAT
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if R has RETRACT(FRAC(INT)) and R has
--R retractIfCan : % -> Union(Integer,"failed") if R has RETRACT(INT) and R has SETCAT
--R setelt : (%,Integer,R) -> R if $ has shallowlyMutable
--R sign : % -> Integer if R has ORDRING
--R size : () -> NonNegativeInteger if R has FINITE
--R size? : (%NonNegativeInteger) -> Boolean
--R subtractIfCan : (%%) -> Union(%,"failed") if R has CABMON
--R sup : (%%) -> % if R has OAMONS
--R swap! : (%Integer,Integer) -> Void if $ has shallowlyMutable
--R unitVector : PositiveInteger -> % if R has RING
--R zero? : % -> Boolean if R has CABMON
--R ?=? : (%%) -> Boolean if R has SETCAT
--R
--E 1

)spool
)lisp (bye)

```

— DirectProduct.help —

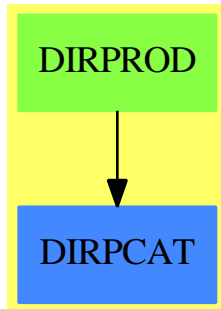
```

=====
DirectProduct examples
=====

```

This type represents the finite direct or cartesian product of an underlying component type. This contrasts with simple vectors in that the members can be viewed as having constant length. Thus many categorical properties can be lifted from the underlying component type. Component extraction operations are provided but no updating operations. Thus new direct product elements can either be created by converting vector elements using the `directProduct` function or by taking appropriate linear combinations of basis vectors provided by the `unitVector` operation.

See Also:
o `)show DirectProduct`

DirectProduct (DIRPROD)**Exports:**

0	1	abs	any?	characteristic
coerce	copy	count	D	differentiate
dimension	directProduct	dot	elt	empty
empty?	entries	entry?	eq?	eval
every?	fill!	first	hash	index
index?	indices	latex	less?	lookup
map	map!	max	maxIndex	member?
members	min	minIndex	more?	negative?
one?	parts	positive?	qelt	qsetelt!
random	recip	reducedSystem	retract	retractIfCan
sample	setelt	sign	size	size?
subtractIfCan	sup	swap!	unitVector	zero?
#?	?*?	?**?	?+?	?-?
?/?	?<?	?<=?	?=?	?>?
?>=?	?^?	?~=?	-?	?.?

— domain DIRPROD DirectProduct —

```

)abbrev domain DIRPROD DirectProduct
++ Author: Mark Botch
++ Description:
++ This type represents the finite direct or cartesian product of an
++ underlying component type. This contrasts with simple vectors in that
++ the members can be viewed as having constant length. Thus many
++ categorical properties can be lifted from the underlying component type.
++ Component extraction operations are provided but no updating operations.
++ Thus new direct product elements can either be created by converting
++ vector elements using the \spadfun{directProduct} function
++ or by taking appropriate linear combinations of basis vectors provided
++ by the \spad{unitVector} operation.

```

```

DirectProduct(dim:NonNegativeInteger, R:Type):
  DirectProductCategory(dim, R) == Vector R add

```

```

Rep := Vector R

coerce(z:%):Vector(R)      == copy(z)$Rep pretend Vector(R)
coerce(r:R):%              == new(dim, r)$Rep

parts x == VEC2LIST(x)$Lisp

directProduct z ==
  size?(z, dim) => copy(z)$Rep
  error "Not of the correct length"

if R has SetCategory then
  same?: % -> Boolean
  same? z == every?(x +-> x = z(minIndex z), z)

  x = y == _and/[qelt(x,i)$Rep = qelt(y,i)$Rep for i in 1..dim]

  retract(z:%):R ==
    same? z => z(minIndex z)
    error "Not retractable"

  retractIfCan(z:%):Union(R, "failed") ==
    same? z => z(minIndex z)
    "failed"

if R has AbelianSemiGroup then
  u:% + v:% == map(_+ , u, v)$Rep

if R has AbelianMonoid then
  0 == zero(dim)$Vector(R) pretend %

if R has Monoid then
  1 == new(dim, 1)$Vector(R) pretend %
  u:% * r:R      == map(x +-> x * r, u)
  r:R * u:%      == map(x +-> r * x, u)
  x:% * y:% == [x.i * y.i for i in 1..dim]$Vector(R) pretend %

if R has CancellationAbelianMonoid then
  subtractIfCan(u:%, v:%):Union(%, "failed") ==
    w := new(dim, 0)$Vector(R)
    for i in 1..dim repeat
      (c:=subtractIfCan(qelt(u, i)$Rep, qelt(v,i)$Rep)) case "failed" =>
        return "failed"
      qsetelt_!(w, i, c::R)$Rep
    w pretend %

if R has Ring then

```

```

u:% * v:%                                == map(_* , u, v)$Rep

recip z ==
  w := new(dim,0)$Vector(R)
  for i in minIndex w .. maxIndex w repeat
    (u := recip qelt(z, i)) case "failed" => return "failed"
    qsetelt_!(w, i, u::R)
  w pretend %

unitVector i ==
  v:= new(dim,0)$Vector(R)
  v.i := 1
  v pretend %

if R has OrderedSet then
  x < y ==
    for i in 1..dim repeat
      qelt(x,i) < qelt(y,i) => return true
      qelt(x,i) > qelt(y,i) => return false
    false

if R has OrderedAbelianMonoidSup then sup(x, y) == map(sup, x, y)

--bo $noSubsumption := false

```

— DIRPROD.dotabb —

```

"DIRPROD" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DIRPROD"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"DIRPROD" -> "DIRPCAT"

```

domain DPMM DirectProductMatrixModule

— DirectProductMatrixModule.input —

```

)set break resume
)sys rm -f DirectProductMatrixModule.output
)spool DirectProductMatrixModule.output
)set message test on

```



```

)set message auto off
)clear all

--S 1 of 1
)show DirectProductMatrixModule
--R DirectProductMatrixModule(n: PositiveInteger,R: Ring,M: SquareMatrixCategory(n,R,DirectProdu
--R Abbreviation for DirectProductMatrixModule is DPMM
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for DPMM
--R
--R----- Operations -----
--R ?? : (M,%) -> %                ?? : (PositiveInteger,%) -> %
--R ?? : (Integer,%) -> %          ?? : (R,%) -> %
--R ?? : (%,%) -> %                -? : % -> %
--R ?? : (%,%) -> %                ?? : (%,%) -> Boolean
--R 0 : () -> %                    coerce : % -> OutputForm
--R coerce : % -> Vector(S)        copy : % -> %
--R directProduct : Vector(S) -> %  ?.? : (%,Integer) -> S
--R elt : (%,Integer,S) -> S        empty : () -> %
--R empty? : % -> Boolean           entries : % -> List(S)
--R eq? : (%,%) -> Boolean          hash : % -> SingleInteger
--R index? : (Integer,%) -> Boolean indices : % -> List(Integer)
--R latex : % -> String             map : ((S -> S),%) -> %
--R qelt : (%,Integer) -> S         sample : () -> %
--R zero? : % -> Boolean            ?~=? : (%,%) -> Boolean
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?? : (%,%) -> % if S has DIFRING and S has RING or S has LINEP(INT) and S has RING or S ha
--R ?? : (S,%) -> % if S has MONOID
--R ?? : (%,S) -> % if S has MONOID
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%,PositiveInteger) -> % if S has DIFRING and S has RING or S has LINEP(INT) and S h
--R ??? : (%,NonNegativeInteger) -> % if S has DIFRING and S has RING or S has LINEP(INT) and
--R ?/? : (%,S) -> % if S has FIELD
--R ?<? : (%,%) -> Boolean if S has OAMONS or S has ORDRING
--R ?<=? : (%,%) -> Boolean if S has OAMONS or S has ORDRING
--R ?>? : (%,%) -> Boolean if S has OAMONS or S has ORDRING
--R ?>=? : (%,%) -> Boolean if S has OAMONS or S has ORDRING
--R D : % -> % if S has DIFRING and S has RING
--R D : (%,NonNegativeInteger) -> % if S has DIFRING and S has RING
--R D : (%,Symbol) -> % if S has PDRING(SYMBOL) and S has RING
--R D : (%,List(Symbol)) -> % if S has PDRING(SYMBOL) and S has RING
--R D : (%,Symbol,NonNegativeInteger) -> % if S has PDRING(SYMBOL) and S has RING
--R D : (%,List(Symbol),List(NonNegativeInteger)) -> % if S has PDRING(SYMBOL) and S has RING
--R D : (%,(S -> S)) -> % if S has RING
--R D : (%,(S -> S),NonNegativeInteger) -> % if S has RING
--R 1 : () -> % if S has DIFRING and S has RING or S has LINEP(INT) and S has RING or S has MON
--R ?? : (%,PositiveInteger) -> % if S has DIFRING and S has RING or S has LINEP(INT) and S ha
--R ?? : (%,NonNegativeInteger) -> % if S has DIFRING and S has RING or S has LINEP(INT) and S
--R abs : % -> % if S has ORDRING
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate

```

```

--R characteristic : () -> NonNegativeInteger if S has RING
--R coerce : Fraction(Integer) -> % if S has RETRACT(FRAC(INT)) and S has SETCAT
--R coerce : Integer -> % if S has RETRACT(INT) and S has SETCAT or S has RING
--R coerce : S -> % if S has SETCAT
--R count : (S,%) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R differentiate : % -> % if S has DIFRING and S has RING
--R differentiate : (%,NonNegativeInteger) -> % if S has DIFRING and S has RING
--R differentiate : (%,Symbol) -> % if S has PDRING(SYMBOL) and S has RING
--R differentiate : (%,List(Symbol)) -> % if S has PDRING(SYMBOL) and S has RING
--R differentiate : (%,Symbol,NonNegativeInteger) -> % if S has PDRING(SYMBOL) and S has RING
--R differentiate : (%,List(Symbol),List(NonNegativeInteger)) -> % if S has PDRING(SYMBOL) and S has RING
--R differentiate : (%,(S -> S)) -> % if S has RING
--R differentiate : (%,(S -> S),NonNegativeInteger) -> % if S has RING
--R dimension : () -> CardinalNumber if S has FIELD
--R dot : (%,%) -> S if S has RING
--R entry? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R enumerate : () -> List(%) if S has FINITE
--R eval : (%,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (%,S) -> % if $ has shallowlyMutable
--R first : % -> S if Integer has ORDSET
--R index : PositiveInteger -> % if S has FINITE
--R less? : (%,NonNegativeInteger) -> Boolean
--R lookup : % -> PositiveInteger if S has FINITE
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R max : (%,%) -> % if S has OAMONS or S has ORDRING
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R members : % -> List(S) if $ has finiteAggregate
--R min : (%,%) -> % if S has OAMONS or S has ORDRING
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (%,NonNegativeInteger) -> Boolean
--R negative? : % -> Boolean if S has ORDRING
--R one? : % -> Boolean if S has DIFRING and S has RING or S has LINEXP(INT) and S has RING or S has MONOID
--R parts : % -> List(S) if $ has finiteAggregate
--R positive? : % -> Boolean if S has ORDRING
--R qsetelt! : (%,Integer,S) -> S if $ has shallowlyMutable
--R random : () -> % if S has FINITE
--R recip : % -> Union(%, "failed") if S has DIFRING and S has RING or S has LINEXP(INT) and S has RING or S
--R reducedSystem : Matrix(%) -> Matrix(Integer) if S has LINEXP(INT) and S has RING
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) if S has LINEXP(INT) and S has RING
--R reducedSystem : Matrix(%) -> Matrix(S) if S has RING
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(S),vec: Vector(S)) if S has RING
--R retract : % -> Fraction(Integer) if S has RETRACT(FRAC(INT)) and S has SETCAT
--R retract : % -> Integer if S has RETRACT(INT) and S has SETCAT
--R retract : % -> S if S has SETCAT

```

```

--R retractIfCan : % -> Union(Fraction(Integer),"failed") if S has RETRACT(FRAC(INT)) and S has
--R retractIfCan : % -> Union(Integer,"failed") if S has RETRACT(INT) and S has SETCAT
--R retractIfCan : % -> Union(S,"failed") if S has SETCAT
--R setelt : (%,Integer,S) -> S if $ has shallowlyMutable
--R sign : % -> Integer if S has ORDRING
--R size : () -> NonNegativeInteger if S has FINITE
--R size? : (%NonNegativeInteger) -> Boolean
--R subtractIfCan : (%,% ) -> Union(%,"failed")
--R sup : (%,% ) -> % if S has OAMONS
--R swap! : (%Integer,Integer) -> Void if $ has shallowlyMutable
--R unitVector : PositiveInteger -> % if S has RING
--R
--E 1

)spool
)lisp (bye)

```

— DirectProductMatrixModule.help —

=====

DirectProductMatrixModule examples

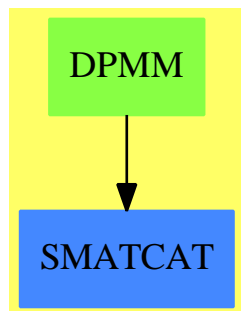
=====

This constructor provides a direct product type with a left matrix-module view.

See Also:

o)show DirectProductMatrixModule

DirectProductMatrixModule (DPMM)



See

⇒ “OppositeMonogenicLinearOperator” (OMLO) [16](#) on page [1806](#)

⇒ “OrdinaryDifferentialRing” (ODR) [16](#) on page [1858](#)

⇒ “DirectProductModule” (DPMO) [5](#) on page [557](#)

Exports:

0	1	coerce	copy	directProduct
elt	empty	empty?	entries	eq?
hash	index?	indices	latex	map
qelt	sample	zero?	D	abs
any?	characteristic	coerce	count	differentiate
dimension	dot	entry?	eval	every?
fill!	first	index	less?	lookup
map!	max	maxIndex	member?	members
min	minIndex	more?	negative?	one?
parts	positive?	qsetelt!	random	recip
reducedSystem	retract	retractIfCan	setelt	sign
size	size?	subtractIfCan	sup	swap!
unitVector	#?	?*?	?**?	?/?
?<?	?<=?	?>?	?>=?	?^?
?^?	-?	?-?	?=?	?..?
?~=?				

— domain DPMM DirectProductMatrixModule —

```

)abbrev domain DPMM DirectProductMatrixModule
++ Author: Stephen M. Watt
++ Date Created: 1986
++ Date Last Updated: June 4, 1991
++ Description:
++ This constructor provides a direct product type with a
++ left matrix-module view.

DirectProductMatrixModule(n, R, M, S): DPcategory == DPcapsule where
  n: PositiveInteger
  R: Ring
  RowCol ==> DirectProduct(n,R)
  M: SquareMatrixCategory(n,R,RowCol,RowCol)
  S: LeftModule(R)

DPcategory == Join(DirectProductCategory(n,S), LeftModule(R), LeftModule(M))

DPcapsule == DirectProduct(n, S) add
  Rep := Vector(S)
  r:R * x:$ == [r*x.i for i in 1..n]
  m:M * x:$ == [ +/[m(i,j)*x.j for j in 1..n] for i in 1..n]

```

— DPMM.dotabb —

```
"DPMM" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DPMM"]
"SMATCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SMATCAT"]
"DPMM" -> "SMATCAT"
```

domain DPMO DirectProductModule

— DirectProductModule.input —

```
)set break resume
)sys rm -f DirectProductModule.output
)spool DirectProductModule.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DirectProductModule
--R DirectProductModule(n: NonNegativeInteger,R: Ring,S: LeftModule(R)) is a domain constructor
--R Abbreviation for DirectProductModule is DPMO
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for DPMO
--R
--R----- Operations -----
--R ???: (PositiveInteger,%) -> %      ??? : (Integer,%) -> %
--R ??? : (R,%) -> %                  ?+: (% ,%) -> %
--R -? : % -> %                       ?-: (% ,%) -> %
--R ?? : (% ,%) -> Boolean            0 : () -> %
--R coerce : % -> OutputForm          coerce : % -> Vector(S)
--R copy : % -> %                     directProduct : Vector(S) -> %
--R ?.? : (% ,Integer) -> S           elt : (% ,Integer,S) -> S
--R empty : () -> %                  empty? : % -> Boolean
--R entries : % -> List(S)            eq? : (% ,%) -> Boolean
--R hash : % -> SingleInteger          index? : (Integer,%) -> Boolean
--R indices : % -> List(Integer)       latex : % -> String
--R map : ((S -> S),%) -> %           qelt : (% ,Integer) -> S
--R sample : () -> %                 zero? : % -> Boolean
--R ~=?: (% ,%) -> Boolean
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ??? : (% ,%) -> % if S has DIFRING and S has RING or S has LINEXP(INT) and S has RING or S has MONOID
--R ??? : (S,%) -> % if S has MONOID
--R ??? : (% ,S) -> % if S has MONOID
```

```

--R ?? : (NonNegativeInteger,%) -> %
--R ***? : (%,PositiveInteger) -> % if S has DIFRING and S has RING or S has LINEXP(INT) and S has RING or S
--R ***? : (%,NonNegativeInteger) -> % if S has DIFRING and S has RING or S has LINEXP(INT) and S has RING or S
--R ?/? : (%,S) -> % if S has FIELD
--R ?<? : (%,%) -> Boolean if S has OAMONS or S has ORDRING
--R ?<=? : (%,%) -> Boolean if S has OAMONS or S has ORDRING
--R ?>? : (%,%) -> Boolean if S has OAMONS or S has ORDRING
--R ?>=? : (%,%) -> Boolean if S has OAMONS or S has ORDRING
--R D : % -> % if S has DIFRING and S has RING
--R D : (%,NonNegativeInteger) -> % if S has DIFRING and S has RING
--R D : (%,Symbol) -> % if S has PDRING(SYMBOL) and S has RING
--R D : (%,List(Symbol)) -> % if S has PDRING(SYMBOL) and S has RING
--R D : (%,Symbol,NonNegativeInteger) -> % if S has PDRING(SYMBOL) and S has RING
--R D : (%,List(Symbol),List(NonNegativeInteger)) -> % if S has PDRING(SYMBOL) and S has RING
--R D : (%,(S -> S)) -> % if S has RING
--R D : (%,(S -> S),NonNegativeInteger) -> % if S has RING
--R 1 : () -> % if S has DIFRING and S has RING or S has LINEXP(INT) and S has RING or S has MONOID or S has
--R ?? : (%,PositiveInteger) -> % if S has DIFRING and S has RING or S has LINEXP(INT) and S has RING or S
--R ?? : (%,NonNegativeInteger) -> % if S has DIFRING and S has RING or S has LINEXP(INT) and S has RING or S
--R abs : % -> % if S has ORDRING
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R characteristic : () -> NonNegativeInteger if S has RING
--R coerce : Fraction(Integer) -> % if S has RETRACT(FRAC(INT)) and S has SETCAT
--R coerce : Integer -> % if S has RETRACT(INT) and S has SETCAT or S has RING
--R coerce : S -> % if S has SETCAT
--R count : (S,%) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R differentiate : % -> % if S has DIFRING and S has RING
--R differentiate : (%,NonNegativeInteger) -> % if S has DIFRING and S has RING
--R differentiate : (%,Symbol) -> % if S has PDRING(SYMBOL) and S has RING
--R differentiate : (%,List(Symbol)) -> % if S has PDRING(SYMBOL) and S has RING
--R differentiate : (%,Symbol,NonNegativeInteger) -> % if S has PDRING(SYMBOL) and S has RING
--R differentiate : (%,List(Symbol),List(NonNegativeInteger)) -> % if S has PDRING(SYMBOL) and S has RING
--R differentiate : (%,(S -> S)) -> % if S has RING
--R differentiate : (%,(S -> S),NonNegativeInteger) -> % if S has RING
--R dimension : () -> CardinalNumber if S has FIELD
--R dot : (%,%) -> S if S has RING
--R entry? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R enumerate : () -> List(%) if S has FINITE
--R eval : (%,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (%,S) -> % if $ has shallowlyMutable
--R first : % -> S if Integer has ORDSET
--R index : PositiveInteger -> % if S has FINITE
--R less? : (%,NonNegativeInteger) -> Boolean
--R lookup : % -> PositiveInteger if S has FINITE
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable

```

```

--R max : (%,% ) -> % if S has OAMONS or S has ORDRING
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (S,% ) -> Boolean if $ has finiteAggregate and S has SETCAT
--R members : % -> List(S) if $ has finiteAggregate
--R min : (%,% ) -> % if S has OAMONS or S has ORDRING
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (% ,NonNegativeInteger) -> Boolean
--R negative? : % -> Boolean if S has ORDRING
--R one? : % -> Boolean if S has DIFRING and S has RING or S has LINEXP(INT) and S has RING or S
--R parts : % -> List(S) if $ has finiteAggregate
--R positive? : % -> Boolean if S has ORDRING
--R qsetelt! : (% ,Integer,S) -> S if $ has shallowlyMutable
--R random : () -> % if S has FINITE
--R recip : % -> Union(% ,"failed") if S has DIFRING and S has RING or S has LINEXP(INT) and S has
--R reducedSystem : Matrix(% ) -> Matrix(Integer) if S has LINEXP(INT) and S has RING
--R reducedSystem : (Matrix(% ),Vector(% )) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) if
--R reducedSystem : Matrix(% ) -> Matrix(S) if S has RING
--R reducedSystem : (Matrix(% ),Vector(% )) -> Record(mat: Matrix(S),vec: Vector(S)) if S has RING
--R retract : % -> Fraction(Integer) if S has RETRACT(FRAC(INT)) and S has SETCAT
--R retract : % -> Integer if S has RETRACT(INT) and S has SETCAT
--R retract : % -> S if S has SETCAT
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if S has RETRACT(FRAC(INT)) and S has
--R retractIfCan : % -> Union(Integer,"failed") if S has RETRACT(INT) and S has SETCAT
--R retractIfCan : % -> Union(S,"failed") if S has SETCAT
--R setelt : (% ,Integer,S) -> S if $ has shallowlyMutable
--R sign : % -> Integer if S has ORDRING
--R size : () -> NonNegativeInteger if S has FINITE
--R size? : (% ,NonNegativeInteger) -> Boolean
--R subtractIfCan : (%,% ) -> Union(% ,"failed")
--R sup : (%,% ) -> % if S has OAMONS
--R swap! : (% ,Integer,Integer) -> Void if $ has shallowlyMutable
--R unitVector : PositiveInteger -> % if S has RING
--R
--E 1

```

```

)spool
)lisp (bye)

```

— DirectProductModule.help —

```

=====
DirectProductModule examples
=====

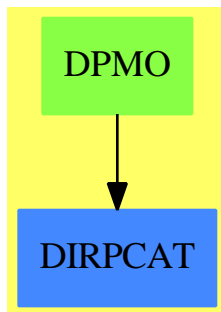
```

This constructor provides a direct product of R-modules with an R-module view.

See Also:

o)show DirectProductModule

DirectProductModule (DPMO)



See

⇒ “OppositeMonogenicLinearOperator” (OMLO) [16](#) on page [1806](#)

⇒ “OrdinaryDifferentialRing” (ODR) [16](#) on page [1858](#)

⇒ “DirectProductMatrixModule” (DPMM) [5](#) on page [552](#)

Exports:

0	1	abs	any?	characteristic
coerce	copy	count	D	differentiate
dimension	directProduct	dot	elt	empty
empty?	entries	entry?	eval	every?
eq?	fill!	first	hash	index
index?	indices	latex	less?	lookup
map	map!	max	maxIndex	member?
members	min	minIndex	more?	negative?
one?	parts	positive?	qelt	qsetelt!
random	recip	reducedSystem	retract	retractIfCan
sample	setelt	sign	size	size?
subtractIfCan	sup	swap!	unitVector	zero?
#?	?*?	?**?	?/?	?<?
?<=?	?>?	?>=?	?^?	?+?
-?	?-?	?=?	?..?	?~=?

— domain DPMO DirectProductModule —

)abbrev domain DPMO DirectProductModule

++ Author: Stephen M. Watt

++ Date Created: 1986


```

++ Date Last Updated: June 4, 1991
++ Description:
++ This constructor provides a direct product of R-modules
++ with an R-module view.

DirectProductModule(n, R, S): DPcategory == DPcapsule where
  n: NonNegativeInteger
  R: Ring
  S: LeftModule(R)

DPcategory == Join(DirectProductCategory(n,S), LeftModule(R))
-- with if S has Algebra(R) then Algebra(R)
-- <above line leads to matchMmCond: unknown form of condition>

DPcapsule == DirectProduct(n,S) add
  Rep := Vector(S)
  r:R * x:$ == [r * x.i for i in 1..n]

```

— DPMO.dotabb —

```

"DPMO" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DPMO"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"DPMO" -> "DIRPCAT"

```

domain DIRRING DirichletRing

The Dirichlet Ring is the ring of arithmetical functions

$$f : \mathbb{N}_+ \rightarrow R$$

(see http://en.wikipedia.org/wiki/Arithmetic_function) together with the Dirichlet convolution (see http://en.wikipedia.org/wiki/Dirichlet_convolution) as multiplication and component-wise addition. Since we can consider the values an arithmetic functions assumes as the coefficients of a Dirichlet generating series, we call R the coefficient ring of a function.

In general we only assume that the coefficient ring R is a ring. If R happens to be commutative, then so is the Dirichlet ring, and in this case it is even an algebra.

Apart from the operations inherited from those categories, we only provide some convenient coercion functions.

— DirichletRing.input —

```

)set break resume
)sys rm -f DirichletRing.output
)spool DirichletRing.output
)set message test on
)set message auto off
)clear all

--S 1 of 21
t1:DIRRING INT := (n:PI):INT --> moebiusMu n
--R
--R
--R (1) [1,- 1,- 1,0,- 1,1,- 1,0,0,1,...]
--R
--R                                          Type: DirichletRing(Integer)
--E 1

--S 2 of 21
[t1.i for i in 1..4]
--R
--R
--R (2) [1,- 1,- 1,0]
--R
--R                                          Type: List(Integer)
--E 2

--S 3 of 21
t2:DIRRING INT := [moebiusMu n for n in 1..]
--R
--R
--R (3) [1,- 1,- 1,0,- 1,1,- 1,0,0,1,...]
--R
--R                                          Type: DirichletRing(Integer)
--E 3

--S 4 of 21
[t2.i for i in 1..4]
--R
--R
--R (4) [1,- 1,- 1,0]
--R
--R                                          Type: List(Integer)
--E 4

--S 5 of 21
DIRRING INT has CommutativeRing
--R
--R
--R (5) true
--R
--R                                          Type: Boolean
--E 5

--S 6 of 21
mu:DIRRING FRAC INT := (n:PI):FRAC INT --> moebiusMu n
--R

```

```

--R
--R (6) [1,- 1,- 1,0,- 1,1,- 1,0,0,1,...]
--R                                         Type: DirichletRing(Fraction(Integer))
--E 6

```

```

--S 7 of 21
phi:DIRRING FRAC INT := (n:PI):FRAC INT +-> eulerPhi n
--R
--R
--R (7) [1,1,2,2,4,2,6,4,6,4,...]
--R                                         Type: DirichletRing(Fraction(Integer))
--E 7

```

```

--S 8 of 21
t3:=[(recip mu * phi).n for n in 1..10]
--R
--R
--R (8) [1,2,3,4,5,6,7,8,9,10]
--R                                         Type: List(Fraction(Integer))
--E 8

```

```

--S 9 of 21
t4:=[(phi * recip mu).n for n in 1..10]
--R
--R
--R (9) [1,2,3,4,5,6,7,8,9,10]
--R                                         Type: List(Fraction(Integer))
--E 9

```

```

--S 10 of 21
reduce(_and,[(x = y)@Boolean for x in t3 for y in t4])
--R
--R
--R (10) true
--R                                         Type: Boolean
--E 10

```

```

--S 11 of 21
DIRRING FRAC INT has Algebra FRAC INT
--R
--R
--R (11) true
--R                                         Type: Boolean
--E 11

```

```

--S 12 of 21
t5:=[(1/2 * phi).n for n in 1..10]
--R
--R
--R 1 1

```

```

--R (12) [-, -, 1, 1, 2, 1, 3, 2, 3, 2]
--R      2 2
--R
--R                                         Type: List(Fraction(Integer))
--E 12

--S 13 of 21
t6:=[eulerPhi n/2 for n in 1..10]
--R
--R
--R      1 1
--R (13) [-, -, 1, 1, 2, 1, 3, 2, 3, 2]
--R      2 2
--R
--R                                         Type: List(Fraction(Integer))
--E 13

--S 14 of 21
reduce(_and, [(x = y)@Boolean for x in t5 for y in t6])
--R
--R
--R (14) true
--R
--R                                         Type: Boolean
--E 14

--S 15 of 21
t7:=[(recip mu).n for n in 1..10]
--R
--R
--R (15) [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
--R
--R                                         Type: List(Fraction(Integer))
--E 15

--S 16 of 21
t8:=[1 for n in 1..10]
--R
--R
--R (16) [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
--R
--R                                         Type: List(PositiveInteger)
--E 16

--S 17 of 21
reduce(_and, [(x = y)@Boolean for x in t7 for y in t8])
--R
--R
--R (17) true
--R
--R                                         Type: Boolean
--E 17

--S 18 of 21
t9:=[(recip mu * phi).n for n in 1..10]
--R

```

```

--R
--R (18) [1,2,3,4,5,6,7,8,9,10]
--R
--R                                         Type: List(Fraction(Integer))
--E 18

--S 19 of 21
t10:=[n for n in 1..10]
--R
--R
--R (19) [1,2,3,4,5,6,7,8,9,10]
--R
--R                                         Type: List(PositiveInteger)
--E 19

--S 20 of 21
reduce(_and,[(x = y)@Boolean for x in t9 for y in t10])
--R
--R
--R (20) true
--R
--R                                         Type: Boolean
--E 20

--S 21 of 21
)show DirichletRing
--R
--R DirichletRing(Coef: Ring) is a domain constructor
--R Abbreviation for DirichletRing is DIRRING
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for DIRRING
--R
--R----- Operations -----
--R ?? : (%,%) -> %                                ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %                ??? : (%,PositiveInteger) -> %
--R +? : (%,%) -> %                                ?-? : (%,%) -> %
--R -? : % -> %                                    ?=? : (%,%) -> Boolean
--R 1 : () -> %                                    0 : () -> %
--R ^? : (%,PositiveInteger) -> %                coerce : % -> Stream(Coef)
--R coerce : Stream(Coef) -> %                    coerce : Integer -> %
--R coerce : % -> OutputForm                      ?..? : (%,PositiveInteger) -> Coef
--R hash : % -> SingleInteger                    latex : % -> String
--R one? : % -> Boolean                          recip : % -> Union(%, "failed")
--R sample : () -> %                             zero? : % -> Boolean
--R zeta : () -> %                               ?^=? : (%,%) -> Boolean
--R ?? : (%,Coef) -> % if Coef has COMRING
--R ?? : (Coef,%) -> % if Coef has COMRING
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R ^? : (%,NonNegativeInteger) -> %
--R additive? : (%,PositiveInteger) -> Boolean
--R associates? : (%,%) -> Boolean if Coef has COMRING
--R characteristic : () -> NonNegativeInteger

```

```

--R coerce : % -> % if Coef has COMRING
--R coerce : Coef -> % if Coef has COMRING
--R coerce : % -> (PositiveInteger -> Coef)
--R coerce : (PositiveInteger -> Coef) -> %
--R exquo : (%,% ) -> Union(%, "failed") if Coef has COMRING
--R multiplicative? : (% , PositiveInteger) -> Boolean
--R subtractIfCan : (% , %) -> Union(%, "failed")
--R unit? : % -> Boolean if Coef has COMRING
--R unitCanonical : % -> % if Coef has COMRING
--R unitNormal : % -> Record(unit: %, canonical: %, associate: %) if Coef has COMRING
--R
--E 21

)spool
)lisp (bye)

```

—————

— DirichletRing.help —

```

=====
DirichletRing examples
=====

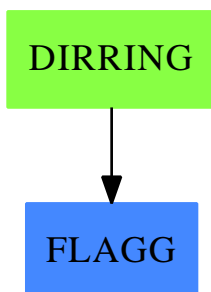
```

DirichletRing is the ring of arithmetical functions with Dirichlet convolution as multiplication

See Also:
o)show DirichletRing

—————

DirichletRing (DIRRING)



Exports:

0	1	?*?	?**?	?+?
?-?	-?	?=?	?^?	?..?
? =?	additive?	associates?	characteristic	coerce
exquo	hash	latex	multiplicative?	one?
recip	sample	subtractIfCan	unit?	unitCanonical
unitNormal	zero?	zeta		

— domain DIRRING DirichletRing —

```

)abbrev domain DIRRING DirichletRing
++ Author: Martin Rubey
++ Description:
++ DirichletRing is the ring of arithmetical functions
++ with Dirichlet convolution as multiplication
DirichletRing(Coef: Ring):
  Exports == Implementation where

  PI ==> PositiveInteger
  FUN ==> PI -> Coef

  Exports ==> Join(Ring, Eltable(PI, Coef)) with

    if Coef has CommutativeRing then
      IntegralDomain

    if Coef has CommutativeRing then
      Algebra Coef

    coerce: FUN -> %
    coerce: % -> FUN
    coerce: Stream Coef -> %
    coerce: % -> Stream Coef

    zeta: constant -> %
    ++ zeta() returns the function which is constantly one

    multiplicative?: (% , PI) -> Boolean
    ++ multiplicative?(a, n) returns true if the first
    ++ n coefficients of a are multiplicative

    additive?: (% , PI) -> Boolean
    ++ additive?(a, n) returns true if the first
    ++ n coefficients of a are additive

  Implementation ==> add

  Rep := Record(function: FUN)

```

```

per(f: Rep): % == f pretend %
rep(a: %): Rep == a pretend Rep

elt(a: %, n: PI): Coef ==
  f: FUN := (rep a).function
  f n

coerce(a: %): FUN == (rep a).function

coerce(f: FUN): % == per [f]

indices: Stream Integer
  := integers(1)$StreamTaylorSeriesOperations(Integer)

coerce(a: %): Stream Coef ==
  f: FUN := (rep a).function
  map((n: Integer): Coef +-> f(n::PI), indices)
  $StreamFunctions2(Integer, Coef)

coerce(f: Stream Coef): % ==
  ((n: PI): Coef +-> f.(n::Integer))::%

coerce(f: %): OutputForm == f::Stream Coef::OutputForm

1: % ==
  ((n: PI): Coef +-> (if one? n then 1$Coef else 0$Coef))::%

0: % ==
  ((n: PI): Coef +-> 0$Coef)::%

zeta: % ==
  ((n: PI): Coef +-> 1$Coef)::%

(f: %) + (g: %) ==
  ((n: PI): Coef +-> f(n)+g(n))::%

- (f: %) ==
  ((n: PI): Coef +-> -f(n))::%

(a: Integer) * (f: %) ==
  ((n: PI): Coef +-> a*f(n))::%

(a: Coef) * (f: %) ==
  ((n: PI): Coef +-> a*f(n))::%

import IntegerNumberTheoryFunctions

(f: %) * (g: %) ==
  conv := (n: PI): Coef +-> _
  reduce((a: Coef, b: Coef): Coef +-> a + b, _

```



```

[f(d::PI) * g((n quo d)::PI) for d in divisors(n::Integer)], 0)
    $ListFunctions2(Coef, Coef)
conv::%

unit?(a: %): Boolean == not (recip(a(1$PI))$Coef case "failed")

qrecip: (% , Coef, PI) -> Coef
qrecip(f: %, flinv: Coef, n: PI): Coef ==
  if one? n then flinv
  else
    -flinv * reduce(+, [f(d::PI) * qrecip(f, flinv, (n quo d)::PI) _
                        for d in rest divisors(n)], 0) _
    $ListFunctions2(Coef, Coef)

recip f ==
  if (flinv := recip(f(1$PI))$Coef) case "failed" then "failed"
  else
    mp := (n: PI): Coef +-> qrecip(f, flinv, n)

    mp::%::Union(%, "failed")

multiplicative?(a, n) ==
  for i in 2..n repeat
    fl := factors(factor i)$Factored(Integer)
    rl := [a.(((f.factor)::PI)**((f.exponent)::PI)) for f in fl]
    if a.(i::PI) ~= reduce((r:Coef, s:Coef):Coef +-> r*s, rl)
    then
      output(i::OutputForm)$OutputPackage
      output(rl::OutputForm)$OutputPackage
      return false
  true

additive?(a, n) ==
  for i in 2..n repeat
    fl := factors(factor i)$Factored(Integer)
    rl := [a.(((f.factor)::PI)**((f.exponent)::PI)) for f in fl]
    if a.(i::PI) ~= reduce((r:Coef, s:Coef):Coef +-> r+s, rl)
    then
      output(i::OutputForm)$OutputPackage
      output(rl::OutputForm)$OutputPackage
      return false
  true

```

— DIRRING.dotabb —

"DIRRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DIRRING"]

"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
 "DIRRING" -> "FLAGG"

domain DMP DistributedMultivariatePolynomial

— DistributedMultivariatePolynomial.input —

```
)set break resume
)sys rm -f DistributedMultivariatePolynomial.output
)spool DistributedMultivariatePolynomial.output
)set message test on
)set message auto off
)clear all
--S 1 of 10
(d1,d2,d3) : DMP([z,y,x],FRAC INT)
--R
--R
--R                                          Type: Void
--E 1

--S 2 of 10
d1 := -4*z + 4*y**2*x + 16*x**2 + 1
--R
--R
--R          2      2
--R   (2)  - 4z + 4y x + 16x  + 1
--R          Type: DistributedMultivariatePolynomial([z,y,x],Fraction(Integer))
--E 2

--S 3 of 10
d2 := 2*z*y**2 + 4*x + 1
--R
--R
--R          2
--R   (3)  2z y  + 4x + 1
--R          Type: DistributedMultivariatePolynomial([z,y,x],Fraction(Integer))
--E 3

--S 4 of 10
d3 := 2*z*x**2 - 2*y**2 - x
--R
--R
--R          2      2
--R   (4)  2z x  - 2y  - x
--R          Type: DistributedMultivariatePolynomial([z,y,x],Fraction(Integer))
```

```

--E 4

--S 5 of 10
groebner [d1,d2,d3]
--R
--R
--R (5)
--R      1568 6    1264 5    6 4    182 3    2047 2    103    2857
--R [z - ---- x - ---- x + --- x + --- x - ---- x - ---- x - ----,
--R      2745      305      305      549      610      2745      10980
--R      2      112 6    84 5    1264 4    13 3    84 2    1772      2
--R y + ---- x - ---- x - ---- x - ---- x + --- x + ---- x + ----,
--R      2745      305      305      549      305      2745      2745
--R      7      29 6    17 4    11 3    1 2    15      1
--R x + -- x - -- x - -- x + -- x + -- x + -]
--R      4      16      8      32      16      4
--R      Type: List(DistributedMultivariatePolynomial([z,y,x],Fraction(Integer)))
--E 5

--S 6 of 10
(n1,n2,n3) : HDMP([z,y,x],FRAC INT)
--R
--R
--R                                          Type: Void
--E 6

--S 7 of 10
n1 := d1
--R
--R
--R      2      2
--R (7)  4y x + 16x - 4z + 1
--RType: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction(Integer))
--E 7

--S 8 of 10
n2 := d2
--R
--R
--R      2
--R (8)  2z y + 4x + 1
--RType: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction(Integer))
--E 8

--S 9 of 10
n3 := d3
--R
--R
--R      2      2
--R (9)  2z x - 2y - x
--RType: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction(Integer))

```

```

--E 9

--S 10 of 10
groebner [n1,n2,n3]
--R
--R
--R (10)
--R      4      3      3      2      1      1      4      29      3      1      2      7      9      1
--R      [y  + 2x  - - x  + - z - -, x  + - x  - - y  - - z x - - x - -,
--R      2      2      2      8      4      8      4      16      4
--R      z y  + 2x + -, y x + 4x  - z + -, z x  - y  - - x,
--R      2      2      2      2      4      2      2
--R      z  - 4y  + 2x  - - z - - x]
--R      4      2
--RType: List(HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction(Integer)))
--E 10
)spool
)lisp (bye)

```

— DistributedMultivariatePolynomial.help —

```

=====
MultivariatePolynomial
DistributedMultivariatePolynomial
HomogeneousDistributedMultivariatePolynomial
GeneralDistributedMultivariatePolynomial
=====

```

DistributedMultivariatePolynomial which is abbreviated as DMP and HomogeneousDistributedMultivariatePolynomial, which is abbreviated as HDMP, are very similar to MultivariatePolynomial except that they are represented and displayed in a non-recursive manner.

```

(d1,d2,d3) : DMP([z,y,x],FRAC INT)
Type: Void

```

The constructor DMP orders its monomials lexicographically while HDMP orders them by total order refined by reverse lexicographic order.

```

d1 := -4*z + 4*y**2*x + 16*x**2 + 1
      2      2
- 4z + 4y x + 16x  + 1
Type: DistributedMultivariatePolynomial([z,y,x],Fraction Integer)

```

```

d2 := 2*z*y**2 + 4*x + 1
      2
2z y  + 4x + 1
      Type: DistributedMultivariatePolynomial([z,y,x],Fraction Integer)

d3 := 2*z*x**2 - 2*y**2 - x
      2      2
2z x  - 2y  - x
      Type: DistributedMultivariatePolynomial([z,y,x],Fraction Integer)

```

These constructors are mostly used in Groebner basis calculations.

```

groebner [d1,d2,d3]
      1568 6 1264 5 6 4 182 3 2047 2 103 2857
[z - ---- x - ---- x + --- x + --- x - ---- x - ---- x - ----,
      2745 305 305 549 610 2745 10980
 2 112 6 84 5 1264 4 13 3 84 2 1772 2
y + ---- x - --- x - ---- x - --- x + --- x + ---- x + ----,
      2745 305 305 549 305 2745 2745
 7 29 6 17 4 11 3 1 2 15 1
x + -- x - -- x - -- x + -- x + -- x + -]
 4 16 8 32 16 4
      Type: List DistributedMultivariatePolynomial([z,y,x],Fraction Integer)

(n1,n2,n3) : HDMP([z,y,x],FRAC INT)
      Type: Void

```

```

n1 := d1
      2      2
4y x + 16x - 4z + 1
      Type: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction Integer)

n2 := d2
      2
2z y  + 4x + 1
      Type: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction Integer)

n3 := d3
      2      2
2z x  - 2y  - x
      Type: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction Integer)

```

Note that we get a different Groebner basis when we use the HDMP polynomials, as expected.

```

groebner [n1,n2,n3]
      4 3 3 2 1 1 4 29 3 1 2 7 9 1
[y + 2x - - x + - z - -, x + -- x - - y - - z x - -- x - -,
      2 2 8 4 8 4 16 4
 2 1 2 2 1 2 2 1

```

$$z^2 y^2 + 2x^2 + \frac{1}{2}, y^2 x^2 + 4x^2 - z + \frac{1}{4}, z^2 x^2 - y^2 - \frac{1}{2}x,$$

$$z^2 - 4y^2 + 2x^2 - \frac{1}{4}z - \frac{3}{2}x]$$

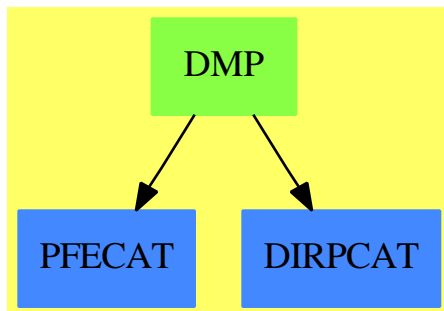
Type: List HomogeneousDistributedMultivariatePolynomial([z,y,x],
Fraction Integer)

GeneralDistributedMultivariatePolynomial is somewhat more flexible in the sense that as well as accepting a list of variables to specify the variable ordering, it also takes a predicate on exponent vectors to specify the term ordering. With this polynomial type the user can experiment with the effect of using completely arbitrary term orderings. This flexibility is mostly important for algorithms such as Groebner basis calculations which can be very sensitive to term ordering.

See Also:

- o)help Polynomial
- o)help UnivariatePolynomial
- o)help MultivariatePolynomial
- o)help HomogeneousDistributedMultivariatePolynomial
- o)help GeneralDistributedMultivariatePolynomial
- o)show DistributedMultivariatePolynomial

DistributedMultivariatePolynomial (DMP)



See

⇒ “GeneralDistributedMultivariatePolynomial” (GDMP) 8 on page 1052

⇒ “HomogeneousDistributedMultivariatePolynomial” (HDMP) 9 on page 1179

Exports:

0	1	associates?
binomThmExpt	characteristic	charthRoot
coefficient	coefficients	coerce
conditionP	convert	D
degree	differentiate	discriminant
eval	exquo	factor
factorPolynomial	factorSquareFreePolynomial	gcd
gcdPolynomial	ground	ground?
hash	isExpt	isPlus
isTimes	latex	lcm
leadingCoefficient	leadingMonomial	mainVariable
map	mapExponents	max
min	minimumDegree	monicDivide
monomial	monomial?	monomials
multivariate	numberOfMonomials	one?
patternMatch	pomopo!	prime?
primitiveMonomials	primitivePart	recip
reducedSystem	reductum	resultant
retract	retractIfCan	reorder
retract	solveLinearPolynomialEquation	sample
squareFree	squareFreePolynomial	squareFreePart
subtractIfCan	totalDegree	unit?
unitCanonical	unitNormal	univariate
variables	zero?	?*?
?**?	?+?	?-?
-?	?=?	?^?
?~=?	?/?	?<?
?<=?	?>?	?>=?

— domain DMP DistributedMultivariatePolynomial —

```

)abbrev domain DMP DistributedMultivariatePolynomial
++ Author: Barry Trager
++ Description:
++ This type supports distributed multivariate polynomials
++ whose variables are from a user specified list of symbols.
++ The coefficient ring may be non commutative,
++ but the variables are assumed to commute.
++ The term ordering is lexicographic specified by the variable
++ list parameter with the most significant variable first in the list.

DistributedMultivariatePolynomial(vl,R): public == private where
  vl : List Symbol
  R  : Ring
  E  ==> DirectProduct(#vl,NonNegativeInteger)
  OV ==> OrderedVariableList(vl)
  public == PolynomialCategory(R,E,OV) with
    reorder: (% ,List Integer) -> %

```

```

++ reorder(p, perm) applies the permutation perm to the variables
++ in a polynomial and returns the new correctly ordered polynomial

private ==
  GeneralDistributedMultivariatePolynomial(v1,R,E)

_____

— DMP.dotabb —

"DMP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DMP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"DMP" -> "PFECAT"
"DMP" -> "DIRPCAT"

_____

```

domain DIV Divisor

```

— Divisor.input —

)set break resume
)sys rm -f Divisor.output
)spool Divisor.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Divisor
--R
--R Divisor(S: SetCategoryWithDegree) is a domain constructor
--R Abbreviation for Divisor is DIV
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for DIV
--R
--R----- Operations -----
--R ??? : (Integer,S) -> %          ??? : (Integer,%) -> %
--R ??? : (%,Integer) -> %         ??? : (Integer,%) -> %
--R ??? : (PositiveInteger,%) -> %  ?+? : (S,%) -> %
--R ?+? : (%,%) -> %              ?-? : (%,%) -> %
--R -? : % -> %                   ?<=? : (%,%) -> Boolean
--R ?=? : (%,%) -> Boolean         0 : () -> %

```



```

--R coefficient : (S,%) -> Integer
--R coerce : % -> OutputForm
--R concat : (%,%) -> %
--R divOfPole : % -> %
--R effective? : % -> Boolean
--R incr : % -> %
--R mapGen : ((S -> S),%) -> %
--R nthFactor : (%,Integer) -> S
--R retract : % -> S
--R size : % -> NonNegativeInteger
--R supp : % -> List(S)
--R suppOfZero : % -> List(S)
--R ?^=? : (%,%) -> Boolean
--R ?*? : (NonNegativeInteger,%) -> %
--R head : % -> Record(gen: S,exp: Integer)
--R highCommonTerms : (%,%) -> % if Integer has OAMON
--R mapCoef : ((Integer -> Integer),%) -> %
--R retractIfCan : % -> Union(S,"failed")
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R terms : % -> List(Record(gen: S,exp: Integer))
--R
--E 1

)spool
)lisp (bye)

```

— Divisor.help —

=====

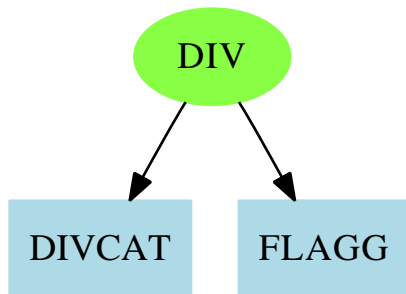
Divisor examples

=====

The following is part of the PAFF package

See Also:
o)show Divisor

Divisor (DIV)

**Exports:**

0	-?	?*?	?+?
?-?	?<=?	?=?	?~=?
coefficient	coerce	collect	concat
degree	divOfPole	divOfZero	effective?
hash	head	highCommonTerms	incr
latex	mapCoef	mapGen	nthCoef
nthFactor	reductum	retract	retractIfCan
sample	size	split	subtractIfCan
supp	suppOfPole	suppOfZero	terms
zero?			

— domain DIV Divisor —

```

)abbrev domain DIV Divisor
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
Divisor(S:SetCategoryWithDegree):Exports == Implementation where

```

```

PT      ==> Record(gen:S,exp: Integer)
INT      ==> Integer
BOOLEAN ==> Boolean
LIST     ==> List

```

```

Exports == DivisorCategory(S) with

```

```

  head: % -> PT

```

```

  reductum: % -> %

```

```

Implementation == List PT add

```

```

Rep := List PT

incr(d)==
  [ [ pt.gen , pt.exp + 1 ] for pt in d ]

inOut: PT -> OutputForm

inOut(pp)==
  one?(pp.exp) => pp.gen :: OutputForm
  bl:OutputForm:= " " ::OutputForm
  (pp.exp :: OutputForm) * hconcat(bl,pp.gen :: OutputForm)

coerce(d:%):OutputForm==
  zero?(d) => ("0"::OutputForm)
  ll:List OutputForm:=[inOut df for df in d]
  reduce("+",ll)

reductum(d)==
  zero?(d) => d
  dl:Rep:= d pretend Rep
  dlr := rest dl
  empty?(dlr) => 0
  dlr

head(d)==
  zero?(d) => error "Cannot take head of zero"
  dl:Rep:= d pretend Rep
  first dl

coerce(s:S) == [[s,1]$PT]::%

split(a) ==
  zero?(a) => []
  [[r]::% for r in a]

coefficient(s,a)==
  r:INT:=0
  for pt in terms(a) repeat
    if pt.gen=s then
      r:=pt.exp
  r

terms(a)==a::Rep

0==empty()$Rep

supp(a)==
  aa:=terms(collect(a))
  [p.gen for p in aa | ^zero?(p.exp)]

```

```

suppOfZero(a)==
  aa:=terms(collect(a))
  [p.gen for p in aa | (p.exp) > 0 ]

suppOfPole(a)==
  aa:=terms(collect(a))
  [p.gen for p in aa | p.exp < 0 ]

divOfZero(a)==
  aa:=terms(collect(a))
  [p for p in aa | (p.exp) > 0 ]::%

divOfPole(a)==
  aa:=terms(collect(a))
  [p for p in aa | p.exp < 0 ]::%

zero?(a)==
  ((collect(a)::Rep)=empty()$Rep)::BOOLEAN

collect(d)==
  a:=d::Rep
  empty?(a) => 0
  t:Rep:=empty()
  ff:PT:=first(a)
  one?(#(a)) =>
    if zero?(ff.exp) then
      t::%
    else
      a::%
  inList?:Boolean:=false()
  newC:INT
  restred:=terms(collect((rest(a)::%)))
  zero?(ff.exp) =>
    restred::%
  for bb in restred repeat
    b:=bb::PT
    if b.gen=ff.gen then
      newC:=b.exp+ff.exp
      if ^zero?(newC) then
        t:=concat(t,[b.gen,newC]$PT)
        inList?:=true()
    else
      t:=concat(t,b)
  if ^inList? then
    t:=cons(ff,t)
  t::%

a:% + b:% ==
  collect(concat(a pretend Rep,b pretend Rep))

```

```

a:% - b:% ==
  a + (-1)*b

-a:% == (-1)*a

n:INT * a:% ==
  zero?(n) => 0
  t:Rep:=empty()
  for p in a pretend Rep repeat
    t:=concat(t,[ p.gen, n*p.exp]$PT)
  t::%

a:% <= b:% ==
  bma:= b - a
  effective? bma => true
  false

effective?(a)== empty?(suppOfPole(a))

degree(d:%):Integer==
  reduce("+",[ (p.exp * degree(p.gen)) for p in d @ Rep],0$INT)

```

— DIV.dotabb —

```

"DIV" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DIV",shape=ellipse]
"DIVCAT" [color=lightblue,href="bookvol10.2.pdf#nameddest=DIVCAT"];
"FLAGG" [color=lightblue,href="bookvol10.2.pdf#nameddest=FLAGG"];
"DIV" -> "DIVCAT"
"DIV" -> "FLAGG"

```

domain DFLOAT DoubleFloat

Greg Vanuxem has added some functionality to allow the user to modify the printed format of floating point numbers. The format of the numbers follows the common lisp format specification for floats. First we include Greg's email to show the use of this feature:

```

PS: For those who use the Doublefloat domain
    there is an another (undocumented) patch that adds a
    lisp format to the DoubleFloat output routine. Copy
    int/algebra/DFLOAT.spad to your working directory,
    patch it, compile it and ")lib" it when necessary.

```

```

(1) -> )boot $useBFasDefault:=false

(SPADLET |$useBFasDefault| NIL)
Value = NIL
(1) -> a:= matrix [ [0.5978,0.2356], [0.4512,0.2355] ]

      +      0.5978      0.2356      +
(1)  |
      +0.4511999999999999  0.2354999999999999+
                                Type: Matrix DoubleFloat

(2) -> )lib DFLOAT
      DoubleFloat is now explicitly exposed in frame initial
      DoubleFloat will be automatically loaded when needed
from /home/greg/Axiom/DFLOAT.nrllib/code
(2) -> doubleFloatFormat("~",4,,F")

      (2)  "~G"
                                Type: String

(3) -> a

      +0.5978  0.2356+
(3)  |
      +0.4512  0.2355+
                                Type: Matrix DoubleFloat

```

So it is clear that he has added a new function called `doubleFloatFormat` which takes a string argument that specifies the common lisp format control string ("`~",4,,F`"). For reference we quote from the common lisp manual [?]. On page 582 we find:

A format directive consists of a tilde (`~`), optional prefix parameters separated by commas, optional colon (`:`) and at-sign (`@`) modifiers, and a single character indicating what kind of directive this is. The alphabetic case of the directive character is ignored. The prefix parameters are generally integers, notated as optionally signed decimal numbers.

X3J13 voted in June 1987 (80) to specify that if both colon and at-sign modifiers are present, they may appear in either order; thus `~:@R` and `~@:R` mean the same thing. However, it is traditional to put the colon first, and all examples in the book put colon before at-signs.

On page 588 we find:

`~F`

Fixed-format floating-point. The next *arg* is printed as a floating point number.

The full form is `~w,d,k,overflowchar,padcharF`. The parameter *w* is the width of the field to be printed; *d* is the number of digits to print after the decimal point; *k* is a scale factor that defaults to zero.

Exactly w characters will be output. First, leading copies of the character *padchar* (which defaults to a space) are printed, if necessary, to pad the field on the left. If the *arg* is negative, then a minus sign is printed; if the *arg* is not negative, then a plus sign is printed if and only if the `@` modifier was specified. Then a sequence of digits, containing a single embedded decimal point, is printed; this represents the magnitude of the value of *arg* times 10^k , rounded to d fractional digits. (When rounding up and rounding down would produce printed values equidistant from the scaled value of *arg*, then the implementation is free to use either one. For example, printing the argument 6.375 using the format `~4.2F` may correctly produce either 6.37 or 6.38.) Leading zeros are not permitted, except that a single zero digit is output before the decimal point if the printed value is less than 1, and this single zero digit is not output after all if $w = d + 1$.

If it is impossible to print the value in the required format in the field of width w , then one of two actions is taken. If the parameter *overflowchar* is specified, then w copies of that parameter are printed instead of the scaled value of *arg*. If the *overflowchar* parameter is omitted, then the scaled value is printed using more than w characters, as many more as may be needed.

If the w parameter is omitted, then the field is of variable width. In effect, a value is chosen for w in such a way that no leading pad characters need to be printed and exactly d characters will follow the decimal point. For example, the directive `~,2F` will print exactly two digits after the decimal point and as many as necessary before the decimal point.

If the parameter d is omitted, then there is no constraint on the number of digits to appear after the decimal point. A value is chosen for d in such a way that as many digits as possible may be printed subject to the width constraint imposed by the parameter w and the constraint that no trailing zero digits may appear in the fraction, except that if the fraction to be printed is zero, then a single zero digit should appear after the decimal point if permitted by the width constraint.

If both w and d are omitted, then the effect is to print the value using ordinary free-format output; `prin1` uses this format for any number whose magnitude is either zero or between 10^{-3} (inclusive) and 10^7 (exclusive).

If w is omitted, then if the magnitude of *arg* is so large (or, if d is also omitted, so small) that more than 100 digits would have to be printed, then an implementation is free, at its discretion, to print the number using exponential notation instead, as if by the directive `~E` (with all parameters of `~E` defaulted, not taking their values from the `~F` directive).

If *arg* is a rational number, then it is coerced to be a **single-float** and then printed. (Alternatively, an implementation is permitted to process a rational number by any other method that has essentially the same behavior but avoids such hazards as loss of precision or overflow because of the coercion. However, note that if w and d are unspecified and the number has no exact decimal representation, for example $1/3$, some precision cutoff must be chosen by the implementation; only a finite number of digits may be printed.)

If *arg* is a complex number or some non-numeric object, then it is printed using the format directive `~wD`, thereby printing it in decimal radix and a minimum field width of *w*. (If it is desired to print each of the real part and imaginary part of a complex number using a `~F` directive, then this must be done explicitly with two `~F` directives and code to extract the two parts of the complex number.)

A key difficulty is creating test suites which generate DoubleFloats. We need to be bit-exact in the results. This can be achieved using the common lisp `integer-decode-float` function. We cover that function in two different ways, with `machineFraction` which translates the results into an Axiom **Fraction(Integer)** and `integerDecode` which just returns a list of the multiple values from the `integer-decode-float` function.

— DoubleFloat.input —

```
)set break resume
)sys rm -f DoubleFloat.output
)spool DoubleFloat.output
)set message test on
)set message auto off
)clear all

--S 1 of 13
2.71828
--R
--R
--R (1) 2.71828
--R
--R                                          Type: Float
--E 1

--S 2 of 13
2.71828@DoubleFloat
--R
--R
--R (2) 2.71828
--R
--R                                          Type: DoubleFloat
--E 2

--S 3 of 13
2.71828 :: DoubleFloat
--R
--R
--R (3) 2.71828
--R
--R                                          Type: DoubleFloat
--E 3

--S 4 of 13
eApprox : DoubleFloat := 2.71828
--R
```



```
--R
--R (4) 2.71828
--R
--R Type: DoubleFloat
--E 4

--S 5 of 13
avg : List DoubleFloat -> DoubleFloat
--R
--R
--R Type: Void
--E 5

--S 6 of 13
avg l ==
  empty? l => 0 :: DoubleFloat
  reduce(+,l) / #l
--R
--R
--R Type: Void
--E 6

--S 7 of 13
avg []
--R
--R Compiling function avg with type List(DoubleFloat) -> DoubleFloat
--R
--R (7) 0.
--R
--R Type: DoubleFloat
--E 7

--S 8 of 13
avg [3.4,9.7,-6.8]
--R
--R
--R (8) 2.0999999999999996
--R
--R Type: DoubleFloat
--E 8

--S 9 of 13
cos(3.1415926)$DoubleFloat
--R
--R
--R (9) - 0.99999999999999856
--R
--R Type: DoubleFloat
--E 9

--S 10 of 13
cos(3.1415926 :: DoubleFloat)
--R
--R
--R (10) - 0.99999999999999856
--R
--R Type: DoubleFloat
```

--E 10

--S 11 of 13

a:DFLOAT := -1.0/3.0

--R

--R

--R (11) - 0.3333333333333337

--R

Type: DoubleFloat

--E 11

--S 12 of 13

integerDecode a

--R

--R

--R (12) [6004799503160662,- 54,- 1]

--R

Type: List(Integer)

--E 12

--S 13 of 13

machineFraction a

--R

--R

--R 3002399751580331

--R (13) - -----

--R 9007199254740992

--R

Type: Fraction(Integer)

--E 13

)spool

)lisp (bye)

— DoubleFloat.help —

=====

DoubleFloat examples

=====

Axiom provides two kinds of floating point numbers. The domain Float (abbreviation FLOAT) implements a model of arbitrary precision floating point numbers. The domain DoubleFloat (abbreviation DFLOAT) is intended to make available hardware floating point arithmetic in Axiom. The actual model of floating point DoubleFloat that provides is system-dependent. For example, on the IBM system 370 Axiom uses IBM double precision which has fourteen hexadecimal digits of precision or roughly sixteen decimal digits. Arbitrary precision floats allow the user to specify the precision at which arithmetic

operations are computed. Although this is an attractive facility, it comes at a cost. Arbitrary-precision floating-point arithmetic typically takes twenty to two hundred times more time than hardware floating point.

The usual arithmetic and elementary functions are available for DoubleFloat. By default, floating point numbers that you enter into Axiom are of type Float.

```
2.71828
2.71828
Type: Float
```

You must therefore tell Axiom that you want to use DoubleFloat values and operations. The following are some conservative guidelines for getting Axiom to use DoubleFloat.

To get a value of type DoubleFloat, use a target with @, ...

```
2.71828@DoubleFloat
2.71828
Type: DoubleFloat
```

a conversion, ...

```
2.71828 :: DoubleFloat
2.71828
Type: DoubleFloat
```

or an assignment to a declared variable. It is more efficient if you use a target rather than an explicit or implicit conversion.

```
eApprox : DoubleFloat := 2.71828
2.71828
Type: DoubleFloat
```

You also need to declare functions that work with DoubleFloat.

```
avg : List DoubleFloat -> DoubleFloat
Type: Void
```

```
avg l ==
  empty? l => 0 :: DoubleFloat
  reduce(+,l) / #l
Type: Void
```

```
avg []
0.
Type: DoubleFloat
```

```
avg [3.4,9.7,-6.8]
2.1000000000000001
Type: DoubleFloat
```

Use package-calling for operations from DoubleFloat unless the arguments themselves are already of type DoubleFloat.

```
cos(3.1415926)$DoubleFloat
-0.99999999999999856
Type: DoubleFloat
```

```
cos(3.1415926 :: DoubleFloat)
-0.99999999999999856
Type: DoubleFloat
```

By far, the most common usage of DoubleFloat is for functions to be graphed.

You can get the exact machine-specific bits of a DoubleFloat in two ways. The first is to use the integerDecode function to break the DoubleFloat into components.

```
a := -1.0/3.0
-0.3333333333333331
integerDecode a
[6004799503160661,- 54,- 1]
```

This is the mantissa, exact to 54 bits, the power of 2, and the sign. Thus it is:

```
6004799503160661 * 2^-54
```

You can get the same information as a fraction with

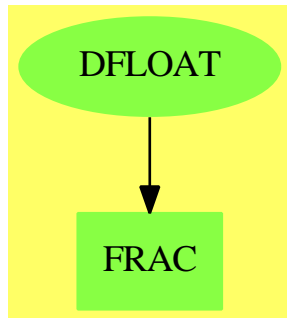
```
machineFraction a
6004799503160661
- -----
18014398509481984
```

where the denominator 18014398509481984 is 2^{54}

See Also:

- o)help Float
- o)show DoubleFloat

DoubleFloat (DFLOAT)



Exports:

0	1	abs	acos
acosh	acot	acoth	acsc
acsch	airyAi	airyBi	asec
asech	asin	asinh	associates?
atan	atanh	base	besselI
besselJ	besselK	besselY	Beta
bits	ceiling	characteristic	coerce
convert	cos	cosh	cot
coth	csc	csch	D
decreasePrecision	differentiate	digamma	digits
divide	doubleFloatFormat	euclideanSize	exp
expressIdealMember	exp1	exponent	exquo
extendedEuclidean	factor	float	floor
fractionPart	Gamma	gcd	gcdPolynomial
hash	increasePrecision	integerDecode	inv
latex	lcm	log	log10
log2	machineFraction	mantissa	max
min	multiEuclidean	negative?	norm
nthRoot	OMwrite	one?	order
patternMatch	pi	polygamma	positive?
precision	prime?	principalIdeal	rationalApproximation
recip	retract	retractIfCan	round
sample	sec	sech	sign
sin	sinh	sizeLess?	sqrt
squareFree	squareFreePart	subtractIfCan	tan
tanh	truncate	unit?	unitCanonical
unitNormal	wholePart	zero?	?*?
?**?	?+?	?-?	-?
?/?	?<?	?<=?	?=?
?>?	?>=?	?^?	?quo?
?rem?	?~=?		

— domain DFLOAT DoubleFloat —

```

)abbrev domain DFLOAT DoubleFloat
++ Author: Michael Monagan
++ Date Created: January 1988
++ Description:
++ \spadtype{DoubleFloat} is intended to make accessible
++ hardware floating point arithmetic in Axiom, either native double
++ precision, or IEEE. On most machines, there will be hardware support for
++ the arithmetic operations: ++ +, *, / and possibly also the
++ sqrt operation.
++ The operations exp, log, sin, cos, atan are normally coded in
++ software based on minimax polynomial/rational approximations.
++

```

```

++ Some general comments about the accuracy of the operations:
++ the operations +, *, / and sqrt are expected to be fully accurate.
++ The operations exp, log, sin, cos and atan are not expected to be
++ fully accurate. In particular, sin and cos
++ will lose all precision for large arguments.
++
++ The Float domain provides an alternative to the DoubleFloat domain.
++ It provides an arbitrary precision model of floating point arithmetic.
++ This means that accuracy problems like those above are eliminated
++ by increasing the working precision where necessary. \spadtype{Float}
++ provides some special functions such as erf, the error function
++ in addition to the elementary functions. The disadvantage of Float is that
++ it is much more expensive than small floats when the latter can be used.

DoubleFloat(): Join(FloatingPointSystem, DifferentialRing, OpenMath,
  TranscendentalFunctionCategory, SpecialFunctionCategory, _
  ConvertibleTo InputForm) with

  _/ : (%, Integer) -> %
    ++ x / i computes the division from x by an integer i.

  _*_ : (%,%) -> %
    ++ x ** y returns the yth power of x (equal to \spad{exp(y log x)}).

  exp1 : () -> %
    ++ exp1() returns the natural log base \spad{2.718281828...}.

  hash : % -> Integer
    ++ hash(x) returns the hash key for x

  log2 : % -> %
    ++ log2(x) computes the logarithm with base 2 for x.

  log10: % -> %
    ++ log10(x) computes the logarithm with base 10 for x.

  atan : (%,%) -> %
    ++ atan(x,y) computes the arc tangent from x with phase y.

  Gamma: % -> %
    ++ Gamma(x) is the Euler Gamma function.

  Beta : (%,%) -> %
    ++ Beta(x,y) is \spad{Gamma(x) * Gamma(y)/Gamma(x+y)}.

  doubleFloatFormat : String -> String
    ++doubleFloatFormat changes the output format for doublefloats
    ++using lisp format strings

  rationalApproximation: (%, NonNegativeInteger) -> Fraction Integer

```

```

++rationalApproximation(f, n) computes a rational approximation
++r to f with relative error \spad{< 10**(-n)}.

rationalApproximation: (% , NonNegativeInteger, NonNegativeInteger) -> _
    Fraction Integer
++rationalApproximation(f, n, b) computes a rational
++approximation r to f with relative error \spad{< b**(-n)}
++(that is, \spad{|(r-f)/f| < b**(-n)}).

machineFraction : % -> Fraction Integer
++machineFraction(x) returns a bit-exact fraction of the machine
++floating point number using the common lisp integer-decode-float
++function. See Steele, ISBN 0-13-152414-3 p354
++This function can be used to print results which do not depend
++on binary-to-decimal conversions
++
++X a:DFLOAT:=-1.0/3.0
++X machineFraction a

integerDecode : % -> List Integer
++integerDecode(x) returns the multiple values of the common
++lisp integer-decode-float function.
++See Steele, ISBN 0-13-152414-3 p354. This function can be used
++to ensure that the results are bit-exact and do not depend on
++the binary-to-decimal conversions.
++
++X a:DFLOAT:=-1.0/3.0
++X integerDecode a

== add
format: String := "~G"
MER ==> Record(MANTISSA:Integer,EXPONENT:Integer)

manexp: % -> MER

doubleFloatFormat(s:String): String ==
    ss: String := format
    format := s
    ss

OMwrite(x: %): String ==
    s: String := ""
    sp := OM_STRINGTOSTRINGPTR(s)$Lisp
    dev: OpenMathDevice := OMopenString(sp @ String, OMencodingXML)
    OMputObject(dev)
    OMputFloat(dev, convert x)
    OMputEndObject(dev)
    OMclose(dev)
    s := OM_STRINGPTRTOSTRING(sp)$Lisp @ String
    s

```



```

OMwrite(x: %, wholeObj: Boolean): String ==
  s: String := ""
  sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
  dev: OpenMathDevice := OMopenString(sp @ String, OMencodingXML)
  if wholeObj then
    OMputObject(dev)
  OMputFloat(dev, convert x)
  if wholeObj then
    OMputEndObject(dev)
  OMclose(dev)
  s := OM_-STRINGPTRTOSTRING(sp)$Lisp @ String
  s

OMwrite(dev: OpenMathDevice, x: %): Void ==
  OMputObject(dev)
  OMputFloat(dev, convert x)
  OMputEndObject(dev)

OMwrite(dev: OpenMathDevice, x: %, wholeObj: Boolean): Void ==
  if wholeObj then
    OMputObject(dev)
  OMputFloat(dev, convert x)
  if wholeObj then
    OMputEndObject(dev)

checkComplex(x:%):% == C_-TO_-R(x)$Lisp
-- In AKCL we used to have to make the arguments to ASIN ACOS ACOSH ATANH
-- complex to get the correct behaviour.
--makeComplex(x: %):% == COMPLEX(x, 0$%)$Lisp

machineFraction(df: %):Fraction(Integer) ==
  numer:Integer:=INTEGER_-DECODE_-FLOAT_-NUMERATOR(df)$Lisp
  denom:Integer:=INTEGER_-DECODE_-FLOAT_-DENOMINATOR(df)$Lisp
  sign:Integer:=INTEGER_-DECODE_-FLOAT_-SIGN(df)$Lisp
  sign*numer/denom

integerDecode(df: %):List(Integer) ==
  numer:Integer:=INTEGER_-DECODE_-FLOAT_-NUMERATOR(df)$Lisp
  exp:Integer:=INTEGER_-DECODE_-FLOAT_-EXPONENT(df)$Lisp
  sign:Integer:=INTEGER_-DECODE_-FLOAT_-SIGN(df)$Lisp
  [numer,exp,sign]

base()          == FLOAT_-RADIX(0$%)$Lisp
mantissa x      == manexp(x).MANTISSA
exponent x      == manexp(x).EXPONENT
precision()     == FLOAT_-DIGITS(0$%)$Lisp
bits()          ==
  base() = 2 => precision()
  base() = 16 => 4*precision()

```

```

wholePart(precision()*log2(base():%))::PositiveInteger
max()      == MOST_-POSITIVE_-DOUBLE_-FLOAT$Lisp
min()      == MOST_-NEGATIVE_-DOUBLE_-FLOAT$Lisp
order(a) == precision() + exponent a - 1
0          == FLOAT(0$Lisp,MOST_-POSITIVE_-DOUBLE_-FLOAT$Lisp)$Lisp
1          == FLOAT(1$Lisp,MOST_-POSITIVE_-DOUBLE_-FLOAT$Lisp)$Lisp
-- rational approximation to e accurate to 23 digits
exp1() == FLOAT(534625820200,MOST_-POSITIVE_-DOUBLE_-FLOAT$Lisp)$Lisp / _
        FLOAT(196677847971,MOST_-POSITIVE_-DOUBLE_-FLOAT$Lisp)$Lisp
pi()     == FLOAT(PI$Lisp,MOST_-POSITIVE_-DOUBLE_-FLOAT$Lisp)$Lisp
coerce(x:%):OutputForm ==
  x >= 0 => message(FORMAT(NIL$Lisp,format,x)$Lisp @ String)
  - (message(FORMAT(NIL$Lisp,format,-x)$Lisp @ String))
convert(x:%):InputForm == convert(x pretend DoubleFloat)$InputForm
x < y          == DFLESSTHAN(x,y)$Lisp
- x           == DFUNARYMINUS(x)$Lisp
x + y         == DFADD(x,y)$Lisp
x:% - y:%     == DFSUBTRACT(x,y)$Lisp
x:% * y:%     == DFMULTIPLY(x,y)$Lisp
i:Integer * x:% == DFINTEGERMULTIPLY(i,x)$Lisp
max(x,y)      == DFMAX(x,y)$Lisp
min(x,y)      == DFMIN(x,y)$Lisp
x = y         == DFEQL(x,y)$Lisp
x:% / i:Integer == DFINTEGERDIVIDE(x,i)$Lisp
sqrt x       == checkComplex DFSQRT(x)$Lisp
log10 x      == checkComplex DFLOG(x,10)$Lisp
x:% ** i:Integer == DFINTEGEREXPT(x,i)$Lisp
x:% ** y:%     == checkComplex DFEXPT(x,y)$Lisp
coerce(i:Integer):% == FLOAT(i,MOST_-POSITIVE_-DOUBLE_-FLOAT$Lisp)$Lisp
exp x        == DFEXP(x)$Lisp
log x        == checkComplex DFLOGE(x)$Lisp
log2 x       == checkComplex DFLOG(x,2)$Lisp
sin x        == DFSIN(x)$Lisp
cos x        == DFCOS(x)$Lisp
tan x        == DFTAN(x)$Lisp
cot x        == COT(x)$Lisp
sec x        == SEC(x)$Lisp
csc x        == CSC(x)$Lisp
asin x       == checkComplex DFASIN(x)$Lisp -- can be complex
acos x       == checkComplex DFACOS(x)$Lisp -- can be complex
atan x       == DFATAN(x)$Lisp
acsc x       == checkComplex ACSC(x)$Lisp
acot x       == ACOT(x)$Lisp
asec x       == checkComplex ASEC(x)$Lisp
sinh x       == SINH(x)$Lisp
cosh x       == COSH(x)$Lisp
tanh x       == TANH(x)$Lisp
csch x       == CSCH(x)$Lisp
coth x       == COTH(x)$Lisp
sech x       == SECH(x)$Lisp

```

```

asinh x      == DFASINH(x)$Lisp
acosh x      == checkComplex DFACOSH(x)$Lisp -- can be complex
atanh x      == checkComplex DFATANH(x)$Lisp -- can be complex
acsch x      == ACSCH(x)$Lisp
acoth x      == checkComplex ACOTH(x)$Lisp
asech x      == checkComplex ASECH(x)$Lisp
x:% / y:%    == DFDIVIDE(x,y)$Lisp
negative? x  == DFMINUSP(x)$Lisp
zero? x      == ZEROP(x)$Lisp
hash x       == SXHASH(x)$Lisp
recip(x)     == (zero? x => "failed"; 1 / x)
differentiate x == 0

SFSFUN       ==> DoubleFloatSpecialFunctions()
sfx          ==> x pretend DoubleFloat
sfy          ==> y pretend DoubleFloat
airyAi x     == airyAi(sfx)$SFSFUN pretend %
airyBi x     == airyBi(sfx)$SFSFUN pretend %
besselI(x,y) == besselI(sfx,sfy)$SFSFUN pretend %
besselJ(x,y) == besselJ(sfx,sfy)$SFSFUN pretend %
besselK(x,y) == besselK(sfx,sfy)$SFSFUN pretend %
besselY(x,y) == besselY(sfx,sfy)$SFSFUN pretend %
Beta(x,y)    == Beta(sfx,sfy)$SFSFUN pretend %
digamma x    == digamma(sfx)$SFSFUN pretend %
Gamma x      == Gamma(sfx)$SFSFUN pretend %
-- not implemented in SFSFUN
-- Gamma(x,y) == Gamma(sfx,sfy)$SFSFUN pretend %
polygamma(x,y) ==
  if (n := retractIfCan(x@%)@Union(Integer, "failed")) case Integer _
    and n >= 0
  then polygamma(n::Integer::NonNegativeInteger,sfy)$SFSFUN pretend %
  else error "polygamma: first argument should be a nonnegative integer"

wholePart x      == TRUNCATE(x)$Lisp
float(ma,ex,b)   == ma*(b::%)**ex
convert(x:%):DoubleFloat == x pretend DoubleFloat
convert(x:%):Float == convert(x pretend DoubleFloat)$Float
rationalApproximation(x, d) == rationalApproximation(x, d, 10)

atan(x,y) ==
  x = 0 =>
    y > 0 => pi()/2
    y < 0 => -pi()/2
    0
  -- Only count on first quadrant being on principal branch.
  theta := atan abs(y/x)
  if x < 0 then theta := pi() - theta
  if y < 0 then theta := - theta
  theta

```

```

retract(x: %): Fraction(Integer) ==
  rationalApproximation(x, (precision() - 1)::NonNegativeInteger, base())

retractIfCan(x: %): Union(Fraction Integer, "failed") ==
  rationalApproximation(x, (precision() - 1)::NonNegativeInteger, base())

retract(x: %): Integer ==
  x = ((n := wholePart x)::%) => n
  error "Not an integer"

retractIfCan(x: %): Union(Integer, "failed") ==
  x = ((n := wholePart x)::%) => n
  "failed"

sign(x) == retract FLOAT_-SIGN(x,1)$Lisp

abs x == FLOAT_-SIGN(1,x)$Lisp

manexp(x) ==
  zero? x => [0,0]
  s := sign x; x := abs x
  if x > max()$% then return [s*mantissa(max())+1,exponent max()]
  me:Record(man:%,exp:Integer) := MANEXP(x)$Lisp
  two53 := base()**precision()
  [s*wholePart(two53 * me.man ),me.exp-precision()]

-- rationalApproximation(y,d,b) ==
--   this is the quotient remainder algorithm (requires wholePart operation)
--   x := y
--   if b < 2 then error "base must be > 1"
--   tol := (b::%)**d
--   p0,p1,q0,q1 : Integer
--   p0 := 0; p1 := 1; q0 := 1; q1 := 0
--   repeat
--     a := wholePart x
--     x := fractionPart x
--     p2 := p0+a*p1
--     q2 := q0+a*q1
--     if x = 0 or tol*abs(q2*y-(p2::%)) < abs(q2*y) then
--       return (p2/q2)
--     (p0,p1) := (p1,p2)
--     (q0,q1) := (q1,q2)
--   x := 1/x

rationalApproximation(f,d,b) ==
  -- this algorithm expresses f as n / d where d = BASE ** k
  -- then all arithmetic operations are done over the integers
  (nu, ex) := manexp f
  BASE := base()
  ex >= 0 => (nu * BASE ** (ex::NonNegativeInteger))::Fraction(Integer)

```

```

de :Integer := BASE**((-ex)::NonNegativeInteger)
b < 2 => error "base must be > 1"
tol := b**d
s := nu; t := de
p0:Integer := 0; p1:Integer := 1; q0:Integer := 1; q1:Integer := 0
repeat
  (q,r) := divide(s, t)
  p2 := q*p1+p0
  q2 := q*q1+q0
  r = 0 or tol*abs(nu*q2-de*p2) < de*abs(p2) => return(p2/q2)
  (p0,p1) := (p1,p2)
  (q0,q1) := (q1,q2)
  (s,t) := (t,r)

x:% ** r:Fraction Integer ==
zero? x =>
  zero? r => error "0**0 is undefined"
  negative? r => error "division by 0"
  0
--      zero? r or one? x => 1
zero? r or (x = 1) => 1
--      one? r => x
(r = 1) => x
n := numer r
d := denom r
negative? x =>
  odd? d =>
    odd? n => return -((-x)**r)
    return ((-x)**r)
  error "negative root"
d = 2 => sqrt(x) ** n
x ** (n:% / d:%)

```

— DFLOAT.dotabb —

```

"DFLOAT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DFLOAT",
          shape=ellipse]
"FRAC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FRAC"]
"DFLOAT" -> "FRAC"

```

domain DFMAT DoubleFloatMatrix

— DoubleFloatMatrix.input —

```

)set break resume
)sys rm -f DoubleFloatMatrix.output
)spool DoubleFloatMatrix.output
)set message test on
)set message auto off
)clear all

--S 1 of 6
)show DoubleFloatMatrix
--R
--R DoubleFloatMatrix is a domain constructor
--R Abbreviation for DoubleFloatMatrix is DFMAT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for DFMAT
--R
--R----- Operations -----
--R ??? : (Integer,%) -> %           ??? : (%,DoubleFloat) -> %
--R ??? : (DoubleFloat,%) -> %       ??? : (%,%) -> %
--R ?+? : (%,%) -> %                 -? : % -> %
--R ?-? : (%,%) -> %                 antisymmetric? : % -> Boolean
--R coerce : DoubleFloatVector -> % copy : % -> %
--R diagonal? : % -> Boolean          diagonalMatrix : List(%) -> %
--R empty : () -> %                  empty? : % -> Boolean
--R eq? : (%,%) -> Boolean           fill! : (%,DoubleFloat) -> %
--R horizConcat : (%,%) -> %         maxColIndex : % -> Integer
--R maxRowIndex : % -> Integer        minColIndex : % -> Integer
--R minRowIndex : % -> Integer        ncols : % -> NonNegativeInteger
--R nrows : % -> NonNegativeInteger   parts : % -> List(DoubleFloat)
--R qnew : (Integer,Integer) -> %     sample : () -> %
--R square? : % -> Boolean            squareTop : % -> %
--R symmetric? : % -> Boolean         transpose : % -> %
--R vertConcat : (%,%) -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ??? : (DoubleFloatVector,%) -> DoubleFloatVector
--R ??? : (%,DoubleFloatVector) -> DoubleFloatVector
--R ??? : (%,Integer) -> % if DoubleFloat has FIELD
--R ??? : (%,NonNegativeInteger) -> %
--R ?/? : (%,DoubleFloat) -> % if DoubleFloat has FIELD
--R ?=? : (%,%) -> Boolean if DoubleFloat has SETCAT
--R any? : ((DoubleFloat -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if DoubleFloat has SETCAT
--R column : (%,Integer) -> DoubleFloatVector
--R columnSpace : % -> List(DoubleFloatVector) if DoubleFloat has EUCLDOM
--R count : (DoubleFloat,%) -> NonNegativeInteger if $ has finiteAggregate and DoubleFloat has SETCAT

```

```

--R count : ((DoubleFloat -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R determinant : % -> DoubleFloat if DoubleFloat has commutative(*)
--R diagonalMatrix : List(DoubleFloat) -> %
--R elt : (%,List(Integer),List(Integer)) -> %
--R elt : (%,Integer,Integer,DoubleFloat) -> DoubleFloat
--R elt : (%,Integer,Integer) -> DoubleFloat
--R eval : (%,List(DoubleFloat),List(DoubleFloat)) -> % if DoubleFloat has EVALAB(DFLOAT) and Do
--R eval : (%,DoubleFloat,DoubleFloat) -> % if DoubleFloat has EVALAB(DFLOAT) and DoubleFloat has
--R eval : (%,Equation(DoubleFloat)) -> % if DoubleFloat has EVALAB(DFLOAT) and DoubleFloat has
--R eval : (%,List(Equation(DoubleFloat))) -> % if DoubleFloat has EVALAB(DFLOAT) and DoubleFloa
--R every? : ((DoubleFloat -> Boolean),%) -> Boolean if $ has finiteAggregate
--R exquo : (%,DoubleFloat) -> Union(%, "failed") if DoubleFloat has INTDOM
--R hash : % -> SingleInteger if DoubleFloat has SETCAT
--R inverse : % -> Union(%, "failed") if DoubleFloat has FIELD
--R latex : % -> String if DoubleFloat has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R listOfLists : % -> List(List(DoubleFloat))
--R map : (((DoubleFloat,DoubleFloat) -> DoubleFloat),%,%,DoubleFloat) -> %
--R map : (((DoubleFloat,DoubleFloat) -> DoubleFloat),%,%) -> %
--R map : ((DoubleFloat -> DoubleFloat),%) -> %
--R map! : ((DoubleFloat -> DoubleFloat),%) -> %
--R matrix : (NonNegativeInteger,NonNegativeInteger,((Integer,Integer) -> DoubleFloat)) -> %
--R matrix : List(List(DoubleFloat)) -> %
--R member? : (DoubleFloat,%) -> Boolean if $ has finiteAggregate and DoubleFloat has SETCAT
--R members : % -> List(DoubleFloat) if $ has finiteAggregate
--R minordet : % -> DoubleFloat if DoubleFloat has commutative(*)
--R more? : (%,NonNegativeInteger) -> Boolean
--R new : (NonNegativeInteger,NonNegativeInteger,DoubleFloat) -> %
--R nullSpace : % -> List(DoubleFloatVector) if DoubleFloat has INTDOM
--R nullity : % -> NonNegativeInteger if DoubleFloat has INTDOM
--R pfaffian : % -> DoubleFloat if DoubleFloat has COMRING
--R qelt : (%,Integer,Integer) -> DoubleFloat
--R qsetelt! : (%,Integer,Integer,DoubleFloat) -> DoubleFloat
--R rank : % -> NonNegativeInteger if DoubleFloat has INTDOM
--R row : (%,Integer) -> DoubleFloatVector
--R rowEchelon : % -> % if DoubleFloat has EUCDOM
--R scalarMatrix : (NonNegativeInteger,DoubleFloat) -> %
--R setColumn! : (%,Integer,DoubleFloatVector) -> %
--R setRow! : (%,Integer,DoubleFloatVector) -> %
--R setelt : (%,List(Integer),List(Integer),%) -> %
--R setelt : (%,Integer,Integer,DoubleFloat) -> DoubleFloat
--R setsubMatrix! : (%,Integer,Integer,%) -> %
--R size? : (%,NonNegativeInteger) -> Boolean
--R subMatrix : (%,Integer,Integer,Integer,Integer) -> %
--R swapColumns! : (%,Integer,Integer) -> %
--R swapRows! : (%,Integer,Integer) -> %
--R transpose : DoubleFloatVector -> %
--R zero : (NonNegativeInteger,NonNegativeInteger) -> %
--R ?=? : (%,%) -> Boolean if DoubleFloat has SETCAT
--R

```

```
--E 1
```

```
--S 2 of 6
```

```
a:DFMAT:=qnew(2,3)
```

```
--R
```

```
--R      +0.  0.  0.+
```

```
--R  (1) |      |
```

```
--R      +0.  0.  0.+
```

```
--R
```

```
Type: DoubleFloatMatrix
```

```
--E 2
```

```
--S 3 of 6
```

```
qsetelt!(a,1,1,1.0)
```

```
--R
```

```
--R  (2) 1.
```

```
--R
```

```
Type: DoubleFloat
```

```
--E 3
```

```
--S 4 of 6
```

```
a
```

```
--R
```

```
--R      +0.  0.  0.+
```

```
--R  (3) |      |
```

```
--R      +0.  1.  0.+
```

```
--R
```

```
Type: DoubleFloatMatrix
```

```
--E 4
```

```
--S 5 of 6
```

```
qsetelt!(a,0,0,2.0)
```

```
--R
```

```
--R  (4) 2.
```

```
--R
```

```
Type: DoubleFloat
```

```
--E 5
```

```
--S 6 of 6
```

```
a
```

```
--R
```

```
--R      +2.  0.  0.+
```

```
--R  (5) |      |
```

```
--R      +0.  1.  0.+
```

```
--R
```

```
Type: DoubleFloatMatrix
```

```
--E 6
```

```
)spool
```

```
)lisp (bye)
```

— DoubleFloatMatrix.help —

```
=====
DoubleFloatMatrix examples
=====
```

This domain creates a lisp simple array of machine doublefloats.
It provides one new function called qnew which takes an integer
that gives the array length.

NOTE: Unlike normal Axiom arrays the DoubleFloatMatrix arrays
are 0-based so the first element is 0. Axiom arrays normally
start at 1.

```
a:DFMAT:=qnew(2,3)
      +0.  0.  0.+
      |      |
      +0.  0.  0.+
```

```
qsetelt!(a,1,1,1.0)
      1.
```

```
a
      +0.  0.  0.+
      |      |
      +0.  1.  0.+
```

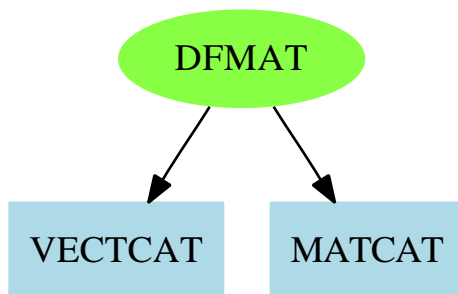
```
qsetelt!(a,0,0,2.0)
      2.
```

```
a
      +2.  0.  0.+
      |      |
      +0.  1.  0.+
```

See Also:

- o)help Float
- o)help DoubleFloat
- o)show DoubleFloatMatrix

DoubleFloatMatrix (DFMAT)

**Exports:**

#?	-?	?**?	?*?
?+?	?-?	?/?	?=?
?~=?	antisymmetric?	any?	coerce
coerce	column	columnSpace	copy
count	count	determinant	diagonal?
diagonalMatrix	diagonalMatrix	elt	elt
elt	empty	empty?	eq?
eval	eval	eval	eval
every?	exquo	fill!	hash
horizConcat	inverse	latex	less?
listOfLists	map	map	map
map!	matrix	maxColIndex	maxRowIndex
member?	members	minColIndex	minRowIndex
minordet	more?	ncols	new
nrows	nullSpace	nullity	parts
pfaffian	qelt	qnew	qsetelt!
rank	row	rowEchelon	sample
scalarMatrix	setColumn!	setRow!	setelt
setelt	setsubMatrix!	size?	square?
squareTop	subMatrix	swapColumns!	swapRows!
symmetric?	transpose	transpose	vertConcat
zero			

— domain DFMAT DoubleFloatMatrix —

```

)abbrev domain DFMAT DoubleFloatMatrix
++ Author: Waldek Hebisch
++ Description:
++ This is a low-level domain which implements matrices
++ (two dimensional arrays) of double precision floating point
++ numbers. Indexing is 0 based, there is no bound checking (unless
++ provided by lower level).

```

```

DoubleFloatMatrix : MatrixCategory(DoubleFloat,
                                   DoubleFloatVector,
                                   DoubleFloatVector) with
  qnew : (Integer, Integer) -> %
    ++ qnew(n, m) creates a new uninitialized n by m matrix.
    ++
    ++X t1:DFMAT:=qnew(3,4)

== add

Qelt2 ==> DAREF2$Lisp
Qsetelt2 ==> DSETAREF2$Lisp
Qnrows ==> DANROWS$Lisp
Qncols ==> DANCOLS$Lisp
Qnew ==> MAKE_-DOUBLE_-MATRIX$Lisp
Qnew1 ==> MAKE_-DOUBLE_-MATRIX1$Lisp

minRowIndex x == 0
minColIndex x == 0
nrows x == Qnrows(x)
ncols x == Qncols(x)
maxRowIndex x == Qnrows(x) - 1
maxColIndex x == Qncols(x) - 1

qelt(m, i, j) == Qelt2(m, i, j)
qsetelt_!(m, i, j, r) == Qsetelt2(m, i, j, r)

empty() == Qnew(0$Integer, 0$Integer)
qnew(rows, cols) == Qnew(rows, cols)
new(rows, cols, a) == Qnew1(rows, cols, a)

```

— DFMAT.dotabb —

```

"DFMAT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DFMAT",
        shape=ellipse]
"VECTCAT" [color=lightblue,href="bookvol10.2.pdf#nameddest=VECTCAT"];
"MATCAT" [color=lightblue,href="bookvol10.2.pdf#nameddest=MATCAT"];
"DFMAT" -> "VECTCAT"
"DFMAT" -> "MATCAT"

```

domain DFVEC DoubleFloatVector

— DoubleFloatVector.input —

```

)set break resume
)sys rm -f DoubleFloatVector.output
)spool DoubleFloatVector.output
)set message test on
)set message auto off
)clear all

--S 1 of 6
)show DoubleFloatVector
--R
--R DoubleFloatVector is a domain constructor
--R Abbreviation for DoubleFloatVector is DFVEC
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for DFVEC
--R
--R----- Operations -----
--R concat : List(%) -> %          concat : (%,%) -> %
--R concat : (DoubleFloat,%) -> %  concat : (%,DoubleFloat) -> %
--R copy : % -> %                  delete : (%,Integer) -> %
--R ?? : (%,Integer) -> DoubleFloat empty : () -> %
--R empty? : % -> Boolean          entries : % -> List(DoubleFloat)
--R eq? : (%,%) -> Boolean         index? : (Integer,%) -> Boolean
--R indices : % -> List(Integer)   insert : (%,%,Integer) -> %
--R qelt : (%,Integer) -> DoubleFloat qnew : Integer -> %
--R reverse : % -> %              sample : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?? : (%,DoubleFloat) -> % if DoubleFloat has MONOID
--R ?? : (DoubleFloat,%) -> % if DoubleFloat has MONOID
--R ?? : (Integer,%) -> % if DoubleFloat has ABELGRP
--R ?? : (%,%) -> % if DoubleFloat has ABELSG
--R ?? : (%,%) -> % if DoubleFloat has ABELGRP
--R -? : % -> % if DoubleFloat has ABELGRP
--R ?<? : (%,%) -> Boolean if DoubleFloat has ORDSET
--R ?<=? : (%,%) -> Boolean if DoubleFloat has ORDSET
--R ?=? : (%,%) -> Boolean if DoubleFloat has SETCAT
--R ?>? : (%,%) -> Boolean if DoubleFloat has ORDSET
--R ?>=? : (%,%) -> Boolean if DoubleFloat has ORDSET
--R any? : ((DoubleFloat -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if DoubleFloat has SETCAT
--R construct : List(DoubleFloat) -> %
--R convert : % -> InputForm if DoubleFloat has KONVERT(INFORM)
--R copyInto! : (%,%,Integer) -> % if $ has shallowlyMutable
--R count : (DoubleFloat,%) -> NonNegativeInteger if $ has finiteAggregate and DoubleFloat has SETCAT
--R count : ((DoubleFloat -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate

```

```

--R cross : (%,%) -> % if DoubleFloat has RING
--R delete : (%,UniversalSegment(Integer)) -> %
--R dot : (%,%) -> DoubleFloat if DoubleFloat has RING
--R ?.? : (%,UniversalSegment(Integer)) -> %
--R elt : (%,Integer,DoubleFloat) -> DoubleFloat
--R entry? : (DoubleFloat,%) -> Boolean if $ has finiteAggregate and DoubleFloat has SETCAT
--R eval : (%,List(DoubleFloat),List(DoubleFloat)) -> % if DoubleFloat has EVALAB(DFLOAT) and DoubleFloat has
--R eval : (%,DoubleFloat,DoubleFloat) -> % if DoubleFloat has EVALAB(DFLOAT) and DoubleFloat has
--R eval : (%,Equation(DoubleFloat)) -> % if DoubleFloat has EVALAB(DFLOAT) and DoubleFloat has
--R eval : (%,List(Equation(DoubleFloat))) -> % if DoubleFloat has EVALAB(DFLOAT) and DoubleFloat has
--R every? : ((DoubleFloat -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (%,DoubleFloat) -> % if $ has shallowlyMutable
--R find : ((DoubleFloat -> Boolean),%) -> Union(DoubleFloat,"failed")
--R first : % -> DoubleFloat if Integer has ORDSET
--R hash : % -> SingleInteger if DoubleFloat has SETCAT
--R insert : (DoubleFloat,%,Integer) -> %
--R latex : % -> String if DoubleFloat has SETCAT
--R length : % -> DoubleFloat if DoubleFloat has RADCAT and DoubleFloat has RING
--R less? : (%,NonNegativeInteger) -> Boolean
--R magnitude : % -> DoubleFloat if DoubleFloat has RADCAT and DoubleFloat has RING
--R map : (((DoubleFloat,DoubleFloat) -> DoubleFloat),%,%) -> %
--R map : ((DoubleFloat -> DoubleFloat),%) -> %
--R map! : ((DoubleFloat -> DoubleFloat),%) -> % if $ has shallowlyMutable
--R max : (%,%) -> % if DoubleFloat has ORDSET
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (DoubleFloat,%) -> Boolean if $ has finiteAggregate and DoubleFloat has SETCAT
--R members : % -> List(DoubleFloat) if $ has finiteAggregate
--R merge : (%,%) -> % if DoubleFloat has ORDSET
--R merge : (((DoubleFloat,DoubleFloat) -> Boolean),%,%) -> %
--R min : (%,%) -> % if DoubleFloat has ORDSET
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (%,NonNegativeInteger) -> Boolean
--R new : (NonNegativeInteger,DoubleFloat) -> %
--R outerProduct : (%,%) -> Matrix(DoubleFloat) if DoubleFloat has RING
--R parts : % -> List(DoubleFloat) if $ has finiteAggregate
--R position : (DoubleFloat,%,Integer) -> Integer if DoubleFloat has SETCAT
--R position : (DoubleFloat,%) -> Integer if DoubleFloat has SETCAT
--R position : ((DoubleFloat -> Boolean),%) -> Integer
--R qsetelt! : (%,Integer,DoubleFloat) -> DoubleFloat if $ has shallowlyMutable
--R reduce : (((DoubleFloat,DoubleFloat) -> DoubleFloat),%) -> DoubleFloat if $ has finiteAggregate
--R reduce : (((DoubleFloat,DoubleFloat) -> DoubleFloat),%,DoubleFloat) -> DoubleFloat if $ has
--R reduce : (((DoubleFloat,DoubleFloat) -> DoubleFloat),%,DoubleFloat,DoubleFloat) -> DoubleFloat
--R remove : ((DoubleFloat -> Boolean),%) -> % if $ has finiteAggregate
--R remove : (DoubleFloat,%) -> % if $ has finiteAggregate and DoubleFloat has SETCAT
--R removeDuplicates : % -> % if $ has finiteAggregate and DoubleFloat has SETCAT
--R reverse! : % -> % if $ has shallowlyMutable
--R select : ((DoubleFloat -> Boolean),%) -> % if $ has finiteAggregate
--R setelt : (%,UniversalSegment(Integer),DoubleFloat) -> DoubleFloat if $ has shallowlyMutable
--R setelt : (%,Integer,DoubleFloat) -> DoubleFloat if $ has shallowlyMutable
--R size? : (%,NonNegativeInteger) -> Boolean

```

```

--R sort : % -> % if DoubleFloat has ORDSET
--R sort : (((DoubleFloat,DoubleFloat) -> Boolean),%) -> %
--R sort! : % -> % if $ has shallowlyMutable and DoubleFloat has ORDSET
--R sort! : (((DoubleFloat,DoubleFloat) -> Boolean),%) -> % if $ has shallowlyMutable
--R sorted? : % -> Boolean if DoubleFloat has ORDSET
--R sorted? : (((DoubleFloat,DoubleFloat) -> Boolean),%) -> Boolean
--R swap! : (%,Integer,Integer) -> Void if $ has shallowlyMutable
--R zero : NonNegativeInteger -> % if DoubleFloat has ABELMON
--R ?=? : (%,%) -> Boolean if DoubleFloat has SETCAT
--R
--E 1

--S 2 of 6
t1:DFVEC:=qnew 17
--R
--R (1) [0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.]
--R                                          Type: DoubleFloatVector
--E 2

-- NOTE: DFVEC arrays are 0-based, normal Axiom arrays are 1-based
--S 3 of 6
t1.1:=1.0
(3) 1.
                                          Type: DoubleFloat
--E 3

--S 4 of 6
t1
(4) [0.,1.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.]
                                          Type: DoubleFloatVector
--E 4

--S 5 of 6
t1.0:=2.0
(5) 2.
                                          Type: DoubleFloat
--E 5

--S 6 of 6
t1
(6) [2.,1.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.]
                                          Type: DoubleFloatVector
--E 6

)spool
)lisp (bye)

```

— DoubleFloatVector.help —

```
=====
DoubleFloatVector examples
=====
```

This domain creates a lisp simple array of machine doublefloats.
It provides one new function called qnew which takes an integer
that gives the array length.

NOTE: Unlike normal Axiom arrays the DoubleFloatVector arrays
are 0-based so the first element is 0. Axiom arrays normally
start at 1.

```
a:DFVEC:=qnew 17
```

```
[0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.]
```

```
a.1:=1.0
```

```
1.
```

```
a
```

```
[0.,1.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.]
```

```
a.0:=2.0
```

```
2.
```

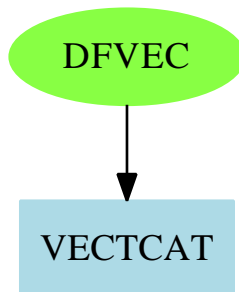
```
a
```

```
[2.,1.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.]
```

See Also:

- o)help Float
 - o)help DoubleFloat
 - o)show DoubleFloatVector
-

DoubleFloatVector (DFVEC)



Exports:

#?	-?	?*?	?+?	?-?
?.?	?<=?	?<?	?=?	?>=?
?>?	?~=?	any?	coerce	concat
construct	convert	copy	copyInto!	count
cross	delete	dot	elt	empty
empty?	entries	entry?	eq?	eval
every?	fill!	find	first	hash
index?	indices	insert	latex	length
less?	magnitude	map	map!	max
maxIndex	member?	members	merge	min
minIndex	more?	new	outerProduct	parts
position	qelt	qnew	qsetelt!	reduce
remove	removeDuplicates	reverse	reverse!	sample
select	setelt	size?	sort	sort!
sorted?	swap!	zero		

— domain DFVEC DoubleFloatVector —

```

)abbrev domain DFVEC DoubleFloatVector
++ Author: Waldek Hebisch
++ Description:
++ This is a low-level domain which implements vectors
++ (one dimensional arrays) of double precision floating point
++ numbers. Indexing is 0 based, there is no bound checking (unless
++ provided by lower level).
DoubleFloatVector : VectorCategory DoubleFloat with
  qnew : Integer -> %
    ++ qnew(n) creates a new uninitialized vector of length n.
    ++
    ++X t1:DFVEC:=qnew(7)
== add

Qelt1 ==> DELT$Lisp

```



```

Qsetelt1 ==> DSETELT$Lisp

qelt(x, i) == Qelt1(x, i)
qsetelt_!(x, i, s) == Qsetelt1(x, i, s)
Qsize ==> DLEN$Lisp
Qnew ==> MAKE_-DOUBLE_-VECTOR$Lisp
Qnew1 ==> MAKE_-DOUBLE_-VECTOR1$Lisp

#x == Qsize x
minIndex x == 0
empty() == Qnew(0$Lisp)
qnew(n) == Qnew(n)
new(n, x) == Qnew1(n, x)
qelt(x, i) == Qelt1(x, i)
elt(x:%, i:Integer) == Qelt1(x, i)
qsetelt_!(x, i, s) == Qsetelt1(x, i, s)
setelt(x : %, i : Integer, s : DoubleFloat) == Qsetelt1(x, i, s)
fill_!(x, s) ==
  for i in 0..((Qsize(x)) - 1) repeat Qsetelt1(x, i, s)
  x

```

— DFVEC.dotabb —

```

"DFVEC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DFVEC",
        shape=ellipse]
"VECTCAT" [color=lightblue,href="bookvol10.2.pdf#nameddest=VECTCAT"];
"DFVEC" -> "VECTCAT"

```

domain DROPT DrawOption

— DrawOption.input —

```

)set break resume
)sys rm -f DrawOption.output
)spool DrawOption.output
)set message test on
)set message auto off
)clear all

```

--S 1 of 1

```

)show DrawOption
--R
--R DrawOption is a domain constructor
--R Abbreviation for DrawOption is DROPT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for DROPT
--R
--R----- Operations -----
--R ==? : (%,% ) -> Boolean          adaptive : Boolean -> %
--R clip : List(Segment(Float)) -> % clip : Boolean -> %
--R coerce : % -> OutputForm         curveColor : Palette -> %
--R curveColor : Float -> %          hash : % -> SingleInteger
--R latex : % -> String              pointColor : Palette -> %
--R pointColor : Float -> %          range : List(Segment(Float)) -> %
--R style : String -> %              title : String -> %
--R toScale : Boolean -> %           tubePoints : PositiveInteger -> %
--R tubeRadius : Float -> %          unit : List(Float) -> %
--R var1Steps : PositiveInteger -> % var2Steps : PositiveInteger -> %
--R ~=? : (%,% ) -> Boolean
--R colorFunction : ((DoubleFloat,DoubleFloat,DoubleFloat) -> DoubleFloat) -> %
--R colorFunction : ((DoubleFloat,DoubleFloat) -> DoubleFloat) -> %
--R colorFunction : (DoubleFloat -> DoubleFloat) -> %
--R coord : (Point(DoubleFloat) -> Point(DoubleFloat)) -> %
--R coordinates : (Point(DoubleFloat) -> Point(DoubleFloat)) -> %
--R option : (List(%),Symbol) -> Union(Any,"failed")
--R option? : (List(%),Symbol) -> Boolean
--R range : List(Segment(Fraction(Integer))) -> %
--R ranges : List(Segment(Float)) -> %
--R space : ThreeSpace(DoubleFloat) -> %
--R viewpoint : Record(theta: DoubleFloat,phi: DoubleFloat,scale: DoubleFloat,scaleX: DoubleFloat,scaleY: D
--R
--E 1

)spool
)lisp (bye)

```

— DrawOption.help —

=====

DrawOption examples

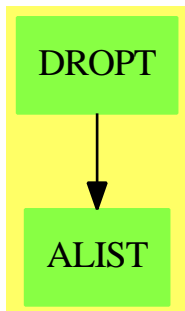
=====

DrawOption allows the user to specify defaults for the creation and rendering of plots.

See Also:

o)show DrawOption

DrawOption (DROPT)



Exports:

adaptive	clip	coerce	colorFunction	coord
coordinates	curveColor	hash	latex	option
option?	pointColor	range	ranges	space
style	title	toScale	tubePoints	tubeRadius
unit	var1Steps	var2Steps	viewpoint	?=?
?~=?				

— domain DROPT DrawOption —

```

)abbrev domain DROPT DrawOption
++ Author: Stephen Watt
++ Date Created: 1 March 1990
++ Date Last Updated: 31 Oct 1990, Jim Wen
++ Description:
++ DrawOption allows the user to specify defaults for the
++ creation and rendering of plots.

DrawOption(): Exports == Implementation where
RANGE ==> List Segment Float
UNIT ==> List Float
PAL ==> Palette
POINT ==> Point(DoubleFloat)
SEG ==> Segment Float
SF ==> DoubleFloat
SPACE3 ==> ThreeSpace(DoubleFloat)
VIEWPT ==> Record(theta:SF, phi:SF, scale:SF, scaleX:SF, scaleY:SF, scaleZ:SF, deltaX:SF, deltaY:SF, deltaZ:SF)

Exports ==> SetCategory with
adaptive : Boolean -> %
  
```

```

++ adaptive(b) turns adaptive 2D plotting on if b is true, or off if b is
++ false. This option is expressed in the form \spad{adaptive == b}.
clip : Boolean -> %
++ clip(b) turns 2D clipping on if b is true, or off if b is false.
++ This option is expressed in the form \spad{clip == b}.
viewpoint : VIEWPT -> %
++ viewpoint(vp) creates a viewpoint data structure corresponding to the
++ list of values. The values are interpreted as [theta, phi, scale,
++ scaleX, scaleY, scaleZ, deltaX, deltaY]. This option is expressed
++ in the form \spad{viewpoint == ls}.
title : String -> %
++ title(s) specifies a title for a plot by the indicated string s.
++ This option is expressed in the form \spad{title == s}.
style : String -> %
++ style(s) specifies the drawing style in which the graph will be plotted
++ by the indicated string s. This option is expressed in the
++ form \spad{style == s}.
toScale : Boolean -> %
++ toScale(b) specifies whether or not a plot is to be drawn to scale;
++ if b is true it is drawn to scale, if b is false it is not. This option
++ is expressed in the form \spad{toScale == b}.
clip : List SEG -> %
++ clip([l]) provides ranges for user-defined clipping as specified
++ in the list l. This option is expressed in the form \spad{clip == [l]}.
coordinates : (POINT -> POINT) -> %
++ coordinates(p) specifies a change of coordinate systems of point p.
++ This option is expressed in the form \spad{coordinates == p}.
pointColor : Float -> %
++ pointColor(v) specifies a color, v, for 2D graph points. This option
++ is expressed in the form \spad{pointColor == v}.
pointColor : PAL -> %
++ pointColor(p) specifies a color index for 2D graph points from the
++ spadcolors palette p. This option is expressed in the
++ form \spad{pointColor == p}.
curveColor : Float -> %
++ curveColor(v) specifies a color, v, for 2D graph curves.
++ This option is expressed in the form \spad{curveColor == v}.
curveColor : PAL -> %
++ curveColor(p) specifies a color index for 2D graph curves from the
++ spadcolors palette p.
++ This option is expressed in the form \spad{curveColor == p}.
colorFunction : (SF -> SF) -> %
++ colorFunction(f(z)) specifies the color based upon the z-component of
++ three dimensional plots. This option is expressed in the
++ form \spad{colorFunction == f(z)}.
colorFunction : ((SF,SF) -> SF) -> %
++ colorFunction(f(u,v)) specifies the color for three dimensional plots
++ as a function based upon the two parametric variables. This option
++ is expressed in the form \spad{colorFunction == f(u,v)}.
colorFunction : ((SF,SF,SF) -> SF) -> %

```

```

++ colorFunction(f(x,y,z)) specifies the color for three dimensional
++ plots as a function of x, y, and z coordinates. This option is
++ expressed in the form \spad{colorFunction == f(x,y,z)}.
tubeRadius : Float -> %
++ tubeRadius(r) specifies a radius, r, for a tube plot around a 3D curve;
++ is expressed in the form \spad{tubeRadius == 4}.
range : List SEG -> %
++ range([l]) provides a user-specified range l.
++ This option is expressed in the form \spad{range == [l]}.
range : List Segment Fraction Integer -> %
++ range([i]) provides a user-specified range i.
++ This option is expressed in the form \spad{range == [i]}.
ranges : RANGE -> %
++ ranges(l) provides a list of user-specified ranges l.
++ This option is expressed in the form \spad{ranges == l}.
space : SPACE3 -> %
++ space specifies the space into which we will draw. If none is given
++ then a new space is created.
var1Steps : PositiveInteger -> %
++ var1Steps(n) indicates the number of subdivisions, n, of the first
++ range variable. This option is expressed in the
++ form \spad{var1Steps == n}.
var2Steps : PositiveInteger -> %
++ var2Steps(n) indicates the number of subdivisions, n, of the second
++ range variable. This option is expressed in the
++ form \spad{var2Steps == n}.
tubePoints : PositiveInteger -> %
++ tubePoints(n) specifies the number of points, n, defining the circle
++ which creates the tube around a 3D curve, the default is 6.
++ This option is expressed in the form \spad{tubePoints == n}.
coord : (POINT->POINT) -> %
++ coord(p) specifies a change of coordinates of point p.
++ This option is expressed in the form \spad{coord == p}.
unit : UNIT -> %
++ unit(lf) will mark off the units according to the indicated list lf.
++ This option is expressed in the form \spad{unit == [f1,f2]}.
option : (List %, Symbol) -> Union(Any, "failed")
++ option() is not to be used at the top level;
++ option determines internally which drawing options are indicated in
++ a draw command.
option?: (List %, Symbol) -> Boolean
++ option?() is not to be used at the top level;
++ option? internally returns true for drawing options which are
++ indicated in a draw command, or false for those which are not.
Implementation ==> add
import AnyFunctions1(String)
import AnyFunctions1(Segment Float)
import AnyFunctions1(VIEWPT)
import AnyFunctions1(List Segment Float)
import AnyFunctions1(List Segment Fraction Integer)

```

```

import AnyFunctions1(List Integer)
import AnyFunctions1(PositiveInteger)
import AnyFunctions1(Boolean)
import AnyFunctions1(RANGE)
import AnyFunctions1(UNIT)
import AnyFunctions1(Float)
import AnyFunctions1(POINT -> POINT)
import AnyFunctions1(SF -> SF)
import AnyFunctions1((SF,SF) -> SF)
import AnyFunctions1((SF,SF,SF) -> SF)
import AnyFunctions1(POINT)
import AnyFunctions1(PAL)
import AnyFunctions1(SPACE3)

Rep := Record(keyword:Symbol, value:Any)

length:List SEG -> NonNegativeInteger
-- these lists will become tuples in a later version
length tup == # tup

lengthR:List Segment Fraction Integer -> NonNegativeInteger
-- these lists will become tuples in a later version
lengthR tup == # tup

lengthI:List Integer -> NonNegativeInteger
-- these lists will become tuples in a later version
lengthI tup == # tup

viewpoint vp ==
  ["viewpoint":Symbol, vp:Any]

title s == ["title":Symbol, s:Any]
style s == ["style":Symbol, s:Any]
toScale b == ["toScale":Symbol, b:Any]
clip(b:Boolean) == ["clipBoolean":Symbol, b:Any]
adaptive b == ["adaptive":Symbol, b:Any]

pointColor(x:Float) == ["pointColorFloat":Symbol, x:Any]
pointColor(c:PAL) == ["pointColorPalette":Symbol, c:Any]
curveColor(x:Float) == ["curveColorFloat":Symbol, x:Any]
curveColor(c:PAL) == ["curveColorPalette":Symbol, c:Any]
colorFunction(f:Sf -> Sf) == ["colorFunction1":Symbol, f:Any]
colorFunction(f:(Sf,Sf) -> Sf) == ["colorFunction2":Symbol, f:Any]
colorFunction(f:(Sf,Sf,Sf) -> Sf) == ["colorFunction3":Symbol, f:Any]
clip(tup:List SEG) ==
  length tup > 3 =>
    error "clip: at most 3 segments may be specified"
  ["clipSegment":Symbol, tup:Any]
coordinates f == ["coordinates":Symbol, f:Any]
tubeRadius x == ["tubeRadius":Symbol, x:Any]

```

```

range(tup:List Segment Float) ==
  ((n := length tup) > 3) =>
    error "range: at most 3 segments may be specified"
  n < 2 =>
    error "range: at least 2 segments may be specified"
  ["rangeFloat"::Symbol, tup::Any]
range(tup:List Segment Fraction Integer) ==
  ((n := lengthR tup) > 3) =>
    error "range: at most 3 segments may be specified"
  n < 2 =>
    error "range: at least 2 segments may be specified"
  ["rangeRat"::Symbol, tup::Any]

ranges s          == ["ranges"::Symbol, s::Any]
space s           == ["space"::Symbol, s::Any]
var1Steps s       == ["var1Steps"::Symbol, s::Any]
var2Steps s       == ["var2Steps"::Symbol, s::Any]
tubePoints s      == ["tubePoints"::Symbol, s::Any]
coord s           == ["coord"::Symbol, s::Any]
unit s            == ["unit"::Symbol, s::Any]
coerce(x:%):OutputForm == x.keyword::OutputForm = x.value::OutputForm
x:% = y:%         == x.keyword = y.keyword and x.value = y.value

option?(l, s) ==
  for x in l repeat
    x.keyword = s => return true
  false

option(l, s) ==
  for x in l repeat
    x.keyword = s => return(x.value)
  "failed"

```

— DROPT.dotabb —

```

"DROPT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=DROPT"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"DROPT" -> "ALIST"

```

domain D01AJFA d01ajfAnnaType

— d01ajfAnnaType.input —

```

)set break resume
)sys rm -f d01ajfAnnaType.output
)spool d01ajfAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d01ajfAnnaType
--R
--R d01ajfAnnaType is a domain constructor
--R Abbreviation for d01ajfAnnaType is D01AJFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for D01AJFA
--R
--R----- Operations -----
--R ?? : (% ,%) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger      latex : % -> String
--R ?~=? : (% ,%) -> Boolean
--R measure : (RoutinesTable,Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedCompletion(Double
--R measure : (RoutinesTable,Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedCompletion
--R numericalIntegration : (Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedCompletion(Double
--R numericalIntegration : (Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedCompletion
--R
--E 1

)spool
)lisp (bye)

```

— d01ajfAnnaType.help —

```

=====
d01ajfAnnaType examples
=====

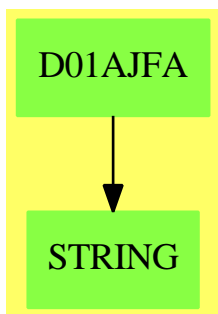
```

d01ajfAnnaType is a domain of NumericalIntegrationCategory for the NAG routine D01AJF, a general numerical integration routine which can handle some singularities in the input function. The function measure measures the usefulness of the routine D01AJF for the given problem. The function numericalIntegration performs the integration by using NagIntegrationPackage.

See Also:


```
o )show d01ajfAnnaType
```

d01ajfAnnaType (D01AJFA)



Exports:

coerce hash latex measure numericalIntegration ?~=? ?=?

— domain D01AJFA d01ajfAnnaType —

```

)abbrev domain D01AJFA d01ajfAnnaType
++ Author: Brian Dupee
++ Date Created: March 1994
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{d01ajfAnnaType} is a domain of
++ \axiomType{NumericalIntegrationCategory}
++ for the NAG routine D01AJF, a general numerical integration routine which
++ can handle some singularities in the input function. The function
++ \axiomFun{measure} measures the usefulness of the routine D01AJF
++ for the given problem. The function \axiomFun{numericalIntegration}
++ performs the integration by using \axiomType{NagIntegrationPackage}.

d01ajfAnnaType(): NumericalIntegrationCategory == Result add
  EF2 ==> ExpressionFunctions2
  EDF ==> Expression DoubleFloat
  LDF ==> List DoubleFloat
  SDF ==> Stream DoubleFloat
  DF ==> DoubleFloat
  FI ==> Fraction Integer
  EFI ==> Expression Fraction Integer
  SOCDF ==> Segment OrderedCompletion DoubleFloat
  NIA ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
  
```

```

MDNIA ==> Record(fn:EDF,range:List SOCDF,abserr:DF,relerr:DF)
INT  ==> Integer
BOP  ==> BasicOperator
S    ==> Symbol
ST   ==> String
LST  ==> List String
RT   ==> RoutinesTable
Rep:=Result
import Rep, NagIntegrationPackage, d01AgentsPackage

measure(R:RT,args:NIA) ==
  ext:Result := empty()$Result
  pp:SDF := singularitiesOf(args)
  not (empty?(pp)$SDF) =>
    [0.1,"d01ajf: There is a possible problem at the following point(s): "
      commaSeparate(sdf2lst(pp)) ,ext]
  [getMeasure(R,d01ajf :: S)$RT,
    "The general routine d01ajf is our default",ext]

numericalIntegration(args:NIA,hints:Result) ==
  ArgsFn := map(x+>convert(x)$DF,args.fn)$EF2(DF,Float)
  b:Float := getButtonValue("d01ajf","functionEvaluations")$AttributeButtons
  fEvals:INT := wholePart exp(1.1513*(1.0/(2.0*(1.0-b))))
  iw:INT := 75*fEvals
  f : Union(fn:FileName,fp:Asp1(F)) := [retract(ArgsFn)$Asp1(F)]
  d01ajf(getlo(args.range),gethi(args.range),args.abserr,_
    args.relerr,4*iw,iw,-1,f)

_____

— D01AJFA.dotabb —

"D01AJFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=D01AJFA"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"D01AJFA" -> "STRING"

_____

```

domain D01AKFA d01akfAnnaType

— d01akfAnnaType.input —

```

)set break resume
)sys rm -f d01akfAnnaType.output

```

```

)spool d01akfAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d01akfAnnaType
--R
--R d01akfAnnaType is a domain constructor
--R Abbreviation for d01akfAnnaType is D01AKFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for D01AKFA
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger        latex : % -> String
--R ?~=? : (%,% ) -> Boolean
--R measure : (RoutinesTable,Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedComple
--R measure : (RoutinesTable,Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(Order
--R numericalIntegration : (Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedComple
--R numericalIntegration : (Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(Ordere
--R
--E 1

```

```

)spool
)lisp (bye)

```

— d01akfAnnaType.help —

```

=====
d01akfAnnaType examples
=====

```

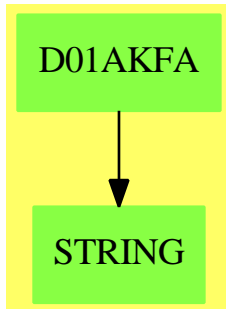
d01akfAnnaType is a domain of NumericalIntegrationCategory for the NAG routine D01AKF, a numerical integration routine which is suitable for oscillating, non-singular functions. The function measure measures the usefulness of the routine D01AKF for the given problem. The function numericalIntegration performs the integration by using NagIntegrationPackage.

See Also:

```

o )show d01akfAnnaType

```

d01akfAnnaType (D01AKFA)**Exports:**

coerce hash latex measure numericalIntegration ?=? ?~=?

— domain D01AKFA d01akfAnnaType —

```

)abbrev domain D01AKFA d01akfAnnaType
++ Author: Brian Dupee
++ Date Created: March 1994
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{d01akfAnnaType} is a domain of
++ \axiomType{NumericalIntegrationCategory}
++ for the NAG routine D01AKF, a numerical integration routine which is
++ is suitable for oscillating, non-singular functions. The function
++ \axiomFun{measure} measures the usefulness of the routine D01AKF
++ for the given problem. The function \axiomFun{numericalIntegration}
++ performs the integration by using \axiomType{NagIntegrationPackage}.

d01akfAnnaType(): NumericalIntegrationCategory == Result add
  EF2 ==> ExpressionFunctions2
  EDF ==> Expression DoubleFloat
  LDF ==> List DoubleFloat
  SDF ==> Stream DoubleFloat
  DF ==> DoubleFloat
  FI ==> Fraction Integer
  EFI ==> Expression Fraction Integer
  SOCDF ==> Segment OrderedCompletion DoubleFloat
  NIA ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
  MDNIA ==> Record(fn:EDF,range:List SOCDF,abserr:DF,relerr:DF)
  INT ==> Integer
  BOP ==> BasicOperator
  S ==> Symbol
  ST ==> String
  LST ==> List String
  RT ==> RoutinesTable
  
```

```

Rep:=Result
import Rep, d01AgentsPackage, NagIntegrationPackage

measure(R:RT,args:NIA) ==
  ext:Result := empty()$Result
  pp:SDF := singularitiesOf(args)
  not (empty?(pp)$SDF) =>
    [0.0,"d01akf: There is a possible problem at the following point(s): "
      commaSeparate(sdf2lst(pp)) ,ext]
  o:Float := functionIsOscillatory(args)
  one := 1.0
  m:Float := (getMeasure(R,d01akf@S)$RT)*(one-one/(one+sqrt(o)))*2
  m > 0.8 => [m,"d01akf: The expression shows much oscillation",ext]
  m > 0.6 => [m,"d01akf: The expression shows some oscillation",ext]
  m > 0.5 => [m,"d01akf: The expression shows little oscillation",ext]
  [m,"d01akf: The expression shows little or no oscillation",ext]

numericalIntegration(args:NIA,hints:Result) ==
  ArgsFn := map(x+>convert(x)$DF,args.fn)$EF2(DF,Float)
  b:Float := getButtonValue("d01akf","functionEvaluations")$AttributeButtons
  fEvals:INT := wholePart exp(1.1513*(1.0/(2.0*(1.0-b))))
  iw:INT := 75*fEvals
  f : Union(fn:FileName,fp:Asp1(F)) := [retract(ArgsFn)$Asp1(F)]
  d01akf(getlo(args.range),gethi(args.range),args.abserr,_
    args.relerr,4*iw,iw,-1,f)

```

— D01AKFA.dotabb —

```

"D01AKFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=D01AKFA"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"D01AKFA" -> "STRING"

```

domain D01ALFA d01alfAnnaType

— d01alfAnnaType.input —

```

)set break resume
)sys rm -f d01alfAnnaType.output
)spool d01alfAnnaType.output
)set message test on

```

```
)set message auto off
)clear all
```

```
--S 1 of 1
```

```
)show d01alfAnnaType
```

```
--R
```

```
--R d01alfAnnaType is a domain constructor
```

```
--R Abbreviation for d01alfAnnaType is D01ALFA
```

```
--R This constructor is exposed in this frame.
```

```
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for D01ALFA
```

```
--R
```

```
--R----- Operations -----
```

```
--R ?=? : (%,% ) -> Boolean          coerce : % -> OutputForm
```

```
--R hash : % -> SingleInteger        latex : % -> String
```

```
--R ?~=? : (%,% ) -> Boolean
```

```
--R measure : (RoutinesTable,Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedCompletion(DoubleFloat),
```

```
--R measure : (RoutinesTable,Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedCompletion(DoubleFloat),
```

```
--R numericalIntegration : (Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedCompletion(DoubleFloat),
```

```
--R numericalIntegration : (Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedCompletion(DoubleFloat),
```

```
--R
```

```
--E 1
```

```
)spool
```

```
)lisp (bye)
```

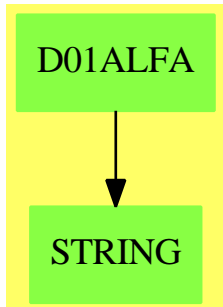
— d01alfAnnaType.help —

```
=====
d01alfAnnaType examples
=====
```

d01alfAnnaType is a domain of NumericalIntegrationCategory for the NAG routine D01ALF, a general numerical integration routine which can handle a list of singularities. The function measure measures the usefulness of the routine D01ALF for the given problem. The function numericalIntegration performs the integration by using NagIntegrationPackage.

See Also:

o)show d01alfAnnaType

d01alfAnnaType (D01ALFA)**Exports:**

coerce hash latex measure numericalIntegration ?~=? ?=?

— domain D01ALFA d01alfAnnaType —

```

)abbrev domain D01ALFA d01alfAnnaType
++ Author: Brian Dupee
++ Date Created: March 1994
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{d01alfAnnaType} is a domain of
++ \axiomType{NumericalIntegrationCategory}
++ for the NAG routine D01ALF, a general numerical integration routine which
++ can handle a list of singularities. The
++ function \axiomFun{measure} measures the usefulness of the routine D01ALF
++ for the given problem. The function \axiomFun{numericalIntegration}
++ performs the integration by using \axiomType{NagIntegrationPackage}.

d01alfAnnaType(): NumericalIntegrationCategory == Result add
  EF2 ==> ExpressionFunctions2
  EDF ==> Expression DoubleFloat
  LDF ==> List DoubleFloat
  SDF ==> Stream DoubleFloat
  DF ==> DoubleFloat
  FI ==> Fraction Integer
  EFI ==> Expression Fraction Integer
  SOCDF ==> Segment OrderedCompletion DoubleFloat
  NIA ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
  MDNIA ==> Record(fn:EDF,range:List SOCDF,abserr:DF,relerr:DF)
  INT ==> Integer
  BOP ==> BasicOperator
  S ==> Symbol
  ST ==> String
  LST ==> List String
  RT ==> RoutinesTable
  
```

```

Rep:=Result
import Rep, d01AgentsPackage, NagIntegrationPackage

measure(R:RT,args:NIA) ==
  ext:Result := empty()$Result
  streamOfZeros:SDF := singularitiesOf(args)
  listOfZeros:LST := removeDuplicates!(sdf2lst(streamOfZeros))
  numberOfZeros:INT := # listOfZeros
  (numberOfZeros > 15)@Boolean =>
    [0.0,"d01alf: The list of singularities is too long", ext]
  positive?(numberOfZeros) =>
    l:LDF := entries(complete(streamOfZeros)$SDF)$SDF
    lany:Any := coerce(1)$AnyFunctions1(LDF)
    ex:Record(key:S,entry:Any) := [d01alfextra@S,lany]
    ext := insert!(ex,ext)$Result
    st:ST := "Recommended is d01alf with the singularities "
              commaSeparate(listOfZeros)
    m :=
--      one?(numberOfZeros) => 0.4
      (numberOfZeros = 1) => 0.4
      getMeasure(R,d01alf@S)$RT
    [m, st, ext]
  [0.0, "d01alf: A list of suitable singularities has not been found", ext]

numericalIntegration(args:NIA,hints:Result) ==
  la:Any := coerce(search((d01alfextra@S),hints)$Result)@Any
  listOfZeros:LDF := retract(la)$AnyFunctions1(LDF)
  l:= removeDuplicates(listOfZeros)$LDF
  n:Integer := (#(l))$List(DF)
  M:Matrix DF := matrix([l])$(Matrix DF)
  b:Float := getButtonValue("d01alf","functionEvaluations")$AttributeButtons
  fEvals:INT := wholePart exp(1.1513*(1.0/(2.0*(1.0-b))))
  iw:INT := 75*fEvals
  ArgsFn := map(x->convert(x)$DF,args.fn)$EF2(DF,Float)
  f : Union(fn:FileName,fp:Asp1(F)) := [retract(ArgsFn)$Asp1(F)]
  d01alf(getlo(args.range),gethi(args.range),n,M,_
    args.abserr,args.relerr,2*n*iw,n*iw,-1,f)



---


— D01ALFA.dotabb —

"D01ALFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=D01ALFA"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"D01ALFA" -> "STRING"

```

domain D01AMFA d01amfAnnaType

— d01amfAnnaType.input —

```
)set break resume
)sys rm -f d01amfAnnaType.output
)spool d01amfAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d01amfAnnaType
--R
--R d01amfAnnaType is a domain constructor
--R Abbreviation for d01amfAnnaType is D01AMFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for D01AMFA
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger        latex : % -> String
--R ?~=? : (%,% ) -> Boolean
--R measure : (RoutinesTable,Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedComple
--R measure : (RoutinesTable,Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(Order
--R numericalIntegration : (Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedComple
--R numericalIntegration : (Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(Ordere
--R
--E 1

)spool
)lisp (bye)
```

— d01amfAnnaType.help —

=====

d01amfAnnaType examples

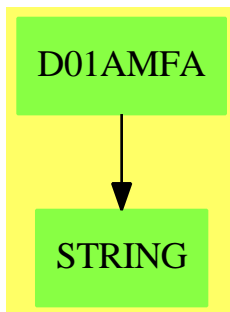
=====

d01amfAnnaType is a domain of NumericalIntegrationCategory for the NAG routine D01AMF, a general numerical integration routine which can handle infinite or semi-infinite range of the input function. The function measure measures the usefulness of the routine D01AMF for the given problem. The function numericalIntegration performs the integration by using NagIntegrationPackage.

See Also:

o)show d01amfAnnaType

d01amfAnnaType (D01AMFA)



Exports:

coerce hash latex measure numericalIntegration ?? ?~=?

— domain D01AMFA d01amfAnnaType —

```

)abbrev domain D01AMFA d01amfAnnaType
++ Author: Brian Dupee
++ Date Created: March 1994
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{d01amfAnnaType} is a domain of
++ \axiomType{NumericalIntegrationCategory}
++ for the NAG routine D01AMF, a general numerical integration routine which
++ can handle infinite or semi-infinite range of the input function. The
++ function \axiomFun{measure} measures the usefulness of the routine D01AMF
++ for the given problem. The function \axiomFun{numericalIntegration}
++ performs the integration by using \axiomType{NagIntegrationPackage}.
  
```

```

d01amfAnnaType(): NumericalIntegrationCategory == Result add
  EF2 ==> ExpressionFunctions2
  EDF ==> Expression DoubleFloat
  LDF ==> List DoubleFloat
  SDF ==> Stream DoubleFloat
  DF ==> DoubleFloat
  FI ==> Fraction Integer
  EFI ==> Expression Fraction Integer
  SOCDF ==> Segment OrderedCompletion DoubleFloat
  
```

```

NIA ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
MDNIA ==> Record(fn:EDF,range:List SOCDF,abserr:DF,relerr:DF)
INT ==> Integer
BOP ==> BasicOperator
S ==> Symbol
ST ==> String
LST ==> List String
RT ==> RoutinesTable
Rep:=Result
import Rep, d01AgentsPackage, NagIntegrationPackage

measure(R:RT,args:NIA) ==
  ext:Result := empty()$Result
  Range:=rangeIsFinite(args)
  pp:SDF := singularitiesOf(args)
  not (empty?(pp)$SDF) =>
    [0.0,"d01amf: There is a possible problem at the following point(s): "
      commaSeparate(sdf2lst(pp)), ext]
  [getMeasure(R,d01amf@S)$RT, "d01amf is a reasonable choice if the "
    "integral is infinite or semi-infinite and d01transform cannot "
    "do better than using general routines",ext]

numericalIntegration(args:NIA,hints:Result) ==
  r:INT
  bound:DF
  ArgsFn := map(x-->convert(x)$DF,args.fn)$EF2(DF,Float)
  b:Float := getButtonValue("d01amf","functionEvaluations")$AttributeButtons
  fEvals:INT := wholePart exp(1.1513*(1.0/(2.0*(1.0-b))))
  iw:INT := 150*fEvals
  f : Union(fn:FileName,fp:Asp1(F)) := [retract(ArgsFn)$Asp1(F)]
  Range:=rangeIsFinite(args)
  if (Range case upperInfinite) then
    bound := getlo(args.range)
    r := 1
  else if (Range case lowerInfinite) then
    bound := gethi(args.range)
    r := -1
  else
    bound := 0$DF
    r := 2
  d01amf(bound,r,args.abserr,args.relerr,4*iw,iw,-1,f)

```

— D01AMFA.dotabb —

```

"D01AMFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=D01AMFA"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]

```

"D01AMFA" -> "STRING"

domain D01ANFA d01anfAnnaType

— d01anfAnnaType.input —

```
)set break resume
)sys rm -f d01anfAnnaType.output
)spool d01anfAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d01anfAnnaType
--R
--R d01anfAnnaType is a domain constructor
--R Abbreviation for d01anfAnnaType is D01ANFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for D01ANFA
--R
--R----- Operations -----
--R ?? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger      latex : % -> String
--R ?~=? : (%,% ) -> Boolean
--R measure : (RoutinesTable,Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedCompletion(DoubleFloat),
--R measure : (RoutinesTable,Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedCompletion(DoubleFloat),
--R numericalIntegration : (Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedCompletion(DoubleFloat),
--R numericalIntegration : (Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedCompletion(DoubleFloat),
--R
--E 1

)spool
)lisp (bye)
```

— d01anfAnnaType.help —

```
=====
d01anfAnnaType examples
=====
```

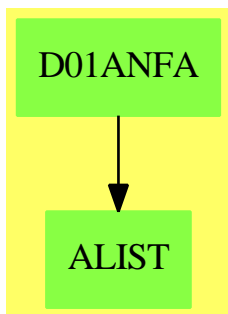
`d01anfAnnaType` is a domain of `NumericalIntegrationCategory` for the NAG routine `D01ANF`, a numerical integration routine which can handle weight functions of the form `cos(omega x)` or `sin(omega x)`. The function `measure` measures the usefulness of the routine `D01ANF` for the given problem. The function `numericalIntegration` performs the integration by using `NagIntegrationPackage`.

See Also:

o `)show d01anfAnnaType`

—————→

`d01anfAnnaType` (`D01ANFA`)



Exports:

`coerce` `hash` `latex` `measure` `numericalIntegration` `?~=?` `?=?`

— domain `D01ANFA` `d01anfAnnaType` —

```

)abbrev domain D01ANFA d01anfAnnaType
++ Author: Brian Dupee
++ Date Created: March 1994
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{d01anfAnnaType} is a domain of
++ \axiomType{NumericalIntegrationCategory}
++ for the NAG routine D01ANF, a numerical integration routine which can
++ handle weight functions of the form cos(\omega x) or sin(\omega x). The
++ function \axiomFun{measure} measures the usefulness of the routine D01ANF
++ for the given problem. The function \axiomFun{numericalIntegration}
++ performs the integration by using \axiomType{NagIntegrationPackage}.

d01anfAnnaType(): NumericalIntegrationCategory == Result add
EF2 ==> ExpressionFunctions2
  
```

```

EDF ==> Expression DoubleFloat
LDF ==> List DoubleFloat
SDF ==> Stream DoubleFloat
DF ==> DoubleFloat
FI ==> Fraction Integer
EFI ==> Expression Fraction Integer
SOCDF ==> Segment OrderedCompletion DoubleFloat
NIA ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
MDNIA ==> Record(fn:EDF,range:List SOCDF,abserr:DF,relerr:DF)
INT ==> Integer
BOP ==> BasicOperator
S ==> Symbol
ST ==> String
LST ==> List String
RT ==> RoutinesTable
Rep:=Result
import Rep, d01WeightsPackage, d01AgentsPackage, NagIntegrationPackage

measure(R:RT,args:NIA) ==
  ext:Result := empty()$Result
  weight:Union(Record(op:BOP,w:DF),"failed") :=
    exprHasWeightCosWXorSinWX(args)
  weight case "failed" =>
    [0.0,"d01anf: A suitable weight has not been found", ext]
  weight case Record(op:BOP,w:DF) =>
    wany := coerce(weight)$AnyFunctions1(Record(op:BOP,w:DF))
    ex:Record(key:S,entry:Any) := [d01anfextra@S,wany]
    ext := insert!(ex,ext)$Result
    ws:ST := string(name(weight.op)$BOP)$S "(" df2st(weight.w)
              string(args.var)$S ")"
    [getMeasure(R,d01anf@S)$RT,
     "d01anf: The expression has a suitable weight:- " ws, ext]

numericalIntegration(args:NIA,hints:Result) ==
  a:INT
  r:Any := coerce(search((d01anfextra@S),hints)$Result)$Any
  rec:Record(op:BOP,w:DF) := retract(r)$AnyFunctions1(Record(op:BOP,w:DF))
  Var := args.var :: EDF
  o:BOP := rec.op
  den:EDF := o((rec.w*Var)$EDF)
  Argsfn:EDF := args.fn/den
  if (name(o) = cos@S)$Boolean then a := 1
  else a := 2
  b:Float := getButtonValue("d01anf","functionEvaluations")$AttributeButtons
  fEvals:INT := wholePart exp(1.1513*(1.0/(2.0*(1.0-b))))
  iw:INT := 75*fEvals
  ArgsFn := map(x+->convert(x)$DF,Argsfn)$EF2(DF,Float)
  f : Union(fn:FileName,fp:Asp1(G)) := [retract(ArgsFn)$Asp1(G)]
  d01anf(getlo(args.range),gethi(args.range),rec.w,a,
        args.abserr,args.relerr,4*iw,iw,-1,f)

```

— D01ANFA.dotabb —

```
"D01ANFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=D01ANFA"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"D01ANFA" -> "ALIST"
```

domain D01APFA d01apfAnnaType

— d01apfAnnaType.input —

```
)set break resume
)sys rm -f d01apfAnnaType.output
)spool d01apfAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d01apfAnnaType
--R
--R d01apfAnnaType is a domain constructor
--R Abbreviation for d01apfAnnaType is D01APFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for D01APFA
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger        latex : % -> String
--R ?~=? : (%,% ) -> Boolean
--R measure : (RoutinesTable,Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedComple
--R measure : (RoutinesTable,Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(Order
--R numericalIntegration : (Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedComple
--R numericalIntegration : (Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(Ordere
--R
--E 1

)spool
)lisp (bye)
```

— d01apfAnnaType.help —

=====

d01apfAnnaType examples

=====

d01apfAnnaType is a domain of NumericalIntegrationCategory for the NAG routine D01APF, a general numerical integration routine which can handle end point singularities of the algebraico-logarithmic form

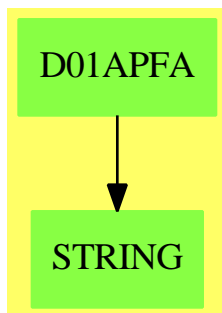
$$w(x) = (x-a)^c * (b-x)^d.$$

The function measure measures the usefulness of the routine D01APF for the given problem. The function numericalIntegration performs the integration by using NagIntegrationPackage.

See Also:

o)show d01apfAnnaType

d01apfAnnaType (D01APFA)



Exports:

coerce hash latex measure numericalIntegration ?=? ?~=?

— domain D01APFA d01apfAnnaType —

```

)abbrev domain D01APFA d01apfAnnaType
++ Author: Brian Dupee
++ Date Created: March 1994
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{d01apfAnnaType} is a domain of
  
```



```

++ \axiomType{NumericalIntegrationCategory}
++ for the NAG routine D01APF, a general numerical integration routine which
++ can handle end point singularities of the algebraico-logarithmic form
++  $w(x) = (x-a)^c * (b-x)^d$ . The
++ function \axiomFun{measure} measures the usefulness of the routine D01APF
++ for the given problem. The function \axiomFun{numericalIntegration}
++ performs the integration by using \axiomType{NagIntegrationPackage}.

```

```

d01apfAnnaType(): NumericalIntegrationCategory == Result add
  EF2 ==> ExpressionFunctions2
  EDF ==> Expression DoubleFloat
  LDF ==> List DoubleFloat
  SDF ==> Stream DoubleFloat
  DF ==> DoubleFloat
  FI ==> Fraction Integer
  EFI ==> Expression Fraction Integer
  SOCDF ==> Segment OrderedCompletion DoubleFloat
  NIA ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
  MDNIA ==> Record(fn:EDF,range:List SOCDF,abserr:DF,relerr:DF)
  INT ==> Integer
  BOP ==> BasicOperator
  S ==> Symbol
  ST ==> String
  LST ==> List String
  RT ==> RoutinesTable
  Rep:=Result
  import Rep, NagIntegrationPackage, d01AgentsPackage, d01WeightsPackage

  measure(R:RT,args:NIA) ==
    ext:Result := empty()$Result
    d := (c := 0$DF)
    if ((a := exprHasAlgebraicWeight(args)) case LDF) then
      if (a.1 > -1) then c := a.1
      if (a.2 > -1) then d := a.2
    l:INT := exprHasLogarithmicWeights(args)
    -- (zero? c) and (zero? d) and (one? l) =>
    (zero? c) and (zero? d) and (l = 1) =>
      [0.0,"d01apf: A suitable singularity has not been found", ext]
    out:LDF := [c,d,l :: DF]
    outany:Any := coerce(out)$AnyFunctions1(LDF)
    ex:Record(key:S,entry:Any) := [d01apfextra@S,outany]
    ext := insert!(ex,ext)$Result
    st:ST := "Recommended is d01apf with c = " df2st(c) ", d = "
              df2st(d) " and l = " string(1)$ST
    [getMeasure(R,d01apf@S)$RT, st, ext]

  numericalIntegration(args:NIA,hints:Result) ==

    Var:EDF := coerce(args.var)$EDF
    la:Any := coerce(search((d01apfextra@S),hints)$Result)$Any

```

```

list:LDF := retract(1a)$AnyFunctions1(LDF)
Fac1:EDF := (Var - (getlo(args.range) :: EDF))$EDF
Fac2:EDF := ((gethi(args.range) :: EDF) - Var)$EDF
c := first(list)$LDF
d := second(list)$LDF
l := (retract(third(list)$LDF)@INT)$DF
thebiz:EDF := (Fac1**(c :: EDF))*(Fac2**(d :: EDF))
if l > 1 then
  if l = 2 then
    thebiz := thebiz*log(Fac1)
  else if l = 3 then
    thebiz := thebiz*log(Fac2)
  else
    thebiz := thebiz*log(Fac1)*log(Fac2)
Fn := (args.fn/thebiz)$EDF
ArgsFn := map(x-->convert(x)$DF,Fn)$EF2(DF,Float)
b:Float := getButtonValue("d01apf","functionEvaluations")$AttributeButtons
fEvals:INT := wholePart exp(1.1513*(1.0/(2.0*(1.0-b))))
iw:INT := 75*fEvals
f : Union(fn:FileName,fp:Asp1(G)) := [retract(ArgsFn)$Asp1(G)]
d01apf(getlo(args.range),gethi(args.range),c,d,l,_,
args.abserr,args.relerr,4*iw,iw,-1,f)

```

— D01APFA.dotabb —

```

"D01APFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=D01APFA"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"D01APFA" -> "STRING"

```

domain D01AQFA d01aqfAnnaType

— d01aqfAnnaType.input —

```

)set break resume
)sys rm -f d01aqfAnnaType.output
)spool d01aqfAnnaType.output
)set message test on
)set message auto off
)clear all

```

```

--S 1 of 1
)show d01aqfAnnaType
--R
--R d01aqfAnnaType is a domain constructor
--R Abbreviation for d01aqfAnnaType is D01AQFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for D01AQFA
--R
--R----- Operations -----
--R ?? : (%,%) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger      latex : % -> String
--R ~=? : (%,%) -> Boolean
--R measure : (RoutinesTable,Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedComple
--R measure : (RoutinesTable,Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(Ordere
--R numericalIntegration : (Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedComple
--R numericalIntegration : (Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(Ordere
--R
--E 1

)spool
)lisp (bye)

```

— d01aqfAnnaType.help —

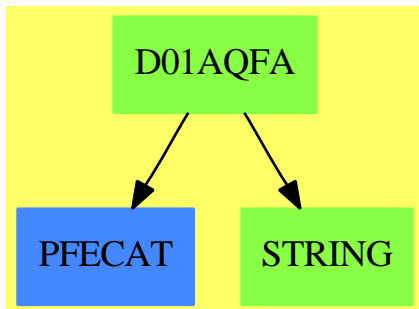
```

=====
d01aqfAnnaType examples
=====

```

d01aqfAnnaType is a domain of NumericalIntegrationCategory for the NAG routine D01AQF, a general numerical integration routine which can solve an integral of the form d01aqf.xbm. The function measure measures the usefulness of the routine D01AQF for the given problem. The function numericalIntegration performs the integration by using NagIntegrationPackage.

See Also:
o)show d01aqfAnnaType

d01aqfAnnaType (D01AQFA)**Exports:**

coerce hash latex measure numericalIntegration ?? ?~=?

— domain D01AQFA d01aqfAnnaType —

```

)abbrev domain D01AQFA d01aqfAnnaType
++ Author: Brian Dupee
++ Date Created: March 1994
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{d01aqfAnnaType} is a domain of
++ \axiomType{NumericalIntegrationCategory}
++ for the NAG routine D01AQF, a general numerical integration routine which
++ can solve an integral of the form
++ /home/bjd/Axiom/anna/hypertext/bitmaps/d01aqf.xbm
++ The function \axiomFun{measure} measures the usefulness of the routine
++ D01AQF for the given problem. The function \axiomFun{numericalIntegration}
++ performs the integration by using \axiomType{NagIntegrationPackage}.

```

```

d01aqfAnnaType(): NumericalIntegrationCategory == Result add
  EF2 ==> ExpressionFunctions2
  EDF ==> Expression DoubleFloat
  LDF ==> List DoubleFloat
  SDF ==> Stream DoubleFloat
  DF ==> DoubleFloat
  FI ==> Fraction Integer
  EFI ==> Expression Fraction Integer
  SOCDF ==> Segment OrderedCompletion DoubleFloat
  NIA ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
  MDNIA ==> Record(fn:EDF,range:List SOCDF,abserr:DF,relerr:DF)
  INT ==> Integer
  BOP ==> BasicOperator
  S ==> Symbol
  ST ==> String
  LST ==> List String

```

```

RT ==> RoutinesTable
Rep:=Result
import Rep, d01AgentsPackage, NagIntegrationPackage

measure(R:RT,args:NIA) ==
  ext:Result := empty()$Result
  Den := denominator(args.fn)
--  one? Den =>
  (Den = 1) =>
    [0.0,"d01aqf: A suitable weight function has not been found", ext]
  listOfZeros:LDF := problemPoints(args.fn,args.var,args.range)
  numberOfZeros := (#(listOfZeros))$LDF
  zero?(numberOfZeros) =>
    [0.0,"d01aqf: A suitable weight function has not been found", ext]
  numberOfZeros = 1 =>
    s:SDF := singularitiesOf(args)
    more?(s,1)$SDF =>
      [0.0,"d01aqf: Too many singularities have been found", ext]
    cFloat:Float := (convert(first(listOfZeros)$LDF)$Float)$DF
    cString:ST := (convert(cFloat)$ST)$Float
    lany:Any := coerce(listOfZeros)$AnyFunctions1(LDF)
    ex:Record(key:S,entry:Any) := [d01aqfextra@S,lany]
    ext := insert!(ex,ext)$Result
    [getMeasure(R,d01aqf@S)$RT, "Recommended is d01aqf with the "
      "hilbertian weight function of 1/(x-c) where c = " cString, ext]
  [0.0,"d01aqf: More than one factor has been found and so does not "
    "have a suitable weight function",ext]

numericalIntegration(args:NIA,hints:Result) ==
  Args := copy args
  ca:Any := coerce(search((d01aqfextra@S),hints)$Result)$Any
  c:DF := first(retract(ca)$AnyFunctions1(LDF))$LDF
  ci:FI := df2fi(c)$ExpertSystemToolsPackage
  Var:EFI := Args.var :: EFI
  Gx:EFI := (Var-(ci::EFI))*(edf2efi(Args.fn)$ExpertSystemToolsPackage)
  ArgsFn := map(x+>convert(x)$FI,Gx)$EF2(FI,Float)
  b:Float := getButtonValue("d01aqf","functionEvaluations")$AttributeButtons
  fEvals:INT := wholePart exp(1.1513*(1.0/(2.0*(1.0-b))))
  iw:INT := 75*fEvals
  f : Union(fn:FileName,fp:Asp1(G)) := [retract(ArgsFn)$Asp1(G)]
  d01aqf(getlo(Args.range),gethi(Args.range),c,_
    Args.abserr,Args.relerr,4*iw,iw,-1,f)

```

— D01AQFA.dotabb —

"D01AQFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=D01AQFA"]

— d01asfAnnaType.input —

=====

d01asfAnnaType examples

=====

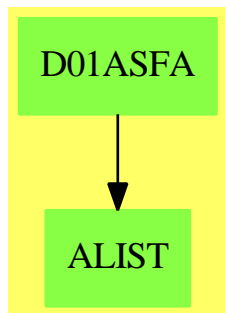
d01asfAnnaType is a domain of NumericalIntegrationCategory for the NAG routine D01ASF, a numerical integration routine which can handle weight functions of the form $\cos(\omega x)$ or $\sin(\omega x)$ on an semi-infinite range. The function measure measures the usefulness of the routine D01ASF for the given problem. The function numericalIntegration performs the integration by using NagIntegrationPackage.

See Also:

o)show d01asfAnnaType

—————

d01asfAnnaType (D01ASFA)



Exports:

coerce hash latex measure numericalIntegration ==? ?~=?

— domain D01ASFA d01asfAnnaType —

```

)abbrev domain D01ASFA d01asfAnnaType
++ Author: Brian Dupee
++ Date Created: March 1994
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{d01asfAnnaType} is a domain of
++ \axiomType{NumericalIntegrationCategory}
++ for the NAG routine D01ASF, a numerical integration routine which can
++ handle weight functions of the form  $\cos(\omega x)$  or  $\sin(\omega x)$  on an
++ semi-infinite range. The
++ function \axiomFun{measure} measures the usefulness of the routine D01ASF
++ for the given problem. The function \axiomFun{numericalIntegration}
  
```

```

++ performs the integration by using \axiomType{NagIntegrationPackage}.

d01asfAnnaType(): NumericalIntegrationCategory == Result add
  EF2 ==> ExpressionFunctions2
  EDF ==> Expression DoubleFloat
  LDF ==> List DoubleFloat
  SDF ==> Stream DoubleFloat
  DF ==> DoubleFloat
  FI ==> Fraction Integer
  EFI ==> Expression Fraction Integer
  SOCDF ==> Segment OrderedCompletion DoubleFloat
  NIA ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
  MDNIA ==> Record(fn:EDF,range:List SOCDF,abserr:DF,relerr:DF)
  INT ==> Integer
  BOP ==> BasicOperator
  S ==> Symbol
  ST ==> String
  LST ==> List String
  RT ==> RoutinesTable
  Rep:=Result
  import Rep, d01WeightsPackage, d01AgentsPackage, NagIntegrationPackage

measure(R:RT,args:NIA) ==
  ext:Result := empty()$Result
  Range := rangeIsFinite(args)
  not(Range case upperInfinite) =>
    [0.0,"d01asf is not a suitable routine for infinite integrals",ext]
  weight: Union(Record(op:BOP,w:DF),"failed") :=
    exprHasWeightCosWXorSinWX(args)
  weight case "failed" =>
    [0.0,"d01asf: A suitable weight has not been found", ext]
  weight case Record(op:BOP,w:DF) =>
    wany := coerce(weight)$AnyFunctions1(Record(op:BOP,w:DF))
    ex:Record(key:S,entry:Any) := [d01asfextra@S,wany]
    ext := insert!(ex,ext)$Result
    ws:ST := string(name(weight.op)$BOP)$S "(" df2st(weight.w)
              string(args.var)$S ")"
    [getMeasure(R,d01asf@S)$RT,
     "d01asf: A suitable weight has been found:- " ws, ext]

numericalIntegration(args:NIA,hints:Result) ==
  i:INT
  r:Any := coerce(search((d01asfextra@S),hints)$Result)$Any
  rec:Record(op:BOP,w:DF) := retract(r)$AnyFunctions1(Record(op:BOP,w:DF))
  Var := args.var :: EDF
  o:BOP := rec.op
  den:EDF := o((rec.w*Var)$EDF)
  Argsfn:EDF := args.fn/den
  if (name(o) = cos@S)@Boolean then i := 1
  else i := 2

```



```

b:Float := getButtonValue("d01asf","functionEvaluations")$AttributeButtons
fEvals:INT := wholePart exp(1.1513*(1.0/(2.0*(1.0-b))))
iw:INT := 75*fEvals
ArgsFn := map(x +-> convert(x)$DF,Argsfn)$EF2(DF,Float)
f : Union(fn:FileName,fp:Asp1(G)) := [retract(ArgsFn)$Asp1(G)]
err :=
  positive?(args.abserr) => args.abserr
  args.relerr
d01asf(getlo(args.range),rec.w,i,err,50,4*iw,2*iw,-1,f)

```

— D01ASFA.dotabb —

```

"D01ASFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=D01ASFA"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"D01ASFA" -> "ALIST"

```

domain D01FCFA d01fcfAnnaType

— d01fcfAnnaType.input —

```

)set break resume
)sys rm -f d01fcfAnnaType.output
)spool d01fcfAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d01fcfAnnaType
--R
--R d01fcfAnnaType is a domain constructor
--R Abbreviation for d01fcfAnnaType is D01FCFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for D01FCFA
--R
--R----- Operations -----
--R ?? : (%,%) -> Boolean               coerce : % -> OutputForm
--R hash : % -> SingleInteger          latex : % -> String
--R ?? : (%,%) -> Boolean
--R measure : (RoutinesTable,Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedCompl

```

```
--R measure : (RoutinesTable,Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedCompletion
--R numericalIntegration : (Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedCompletion(Double
--R numericalIntegration : (Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedCompletion
--R
--E 1

)spool
)lisp (bye)
```

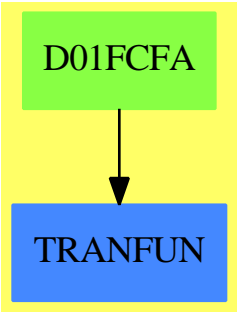
— d01fcfAnnaType.help —

=====
d01fcfAnnaType examples
=====

d01fcfAnnaType is a domain of NumericalIntegrationCategory for the NAG routine D01FCF, a numerical integration routine which can handle multi-dimensional quadrature over a finite region. The function measure measures the usefulness of the routine D01GBF for the given problem. The function numericalIntegration performs the integration by using NagIntegrationPackage.

See Also:
o)show d01fcfAnnaType

d01fcfAnnaType (D01FCFA)



Exports:
coerce hash latex measure numericalIntegration ?? ?~=?

— domain D01FCFA d01fcfAnnaType —

```

)abbrev domain D01FCFA d01fcfAnnaType
++ Author: Brian Dupee
++ Date Created: March 1994
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{d01fcfAnnaType} is a domain of
++ \axiomType{NumericalIntegrationCategory}
++ for the NAG routine D01FCF, a numerical integration routine which can
++ handle multi-dimensional quadrature over a finite region. The
++ function \axiomFun{measure} measures the usefulness of the routine D01GBF
++ for the given problem. The function \axiomFun{numericalIntegration}
++ performs the integration by using \axiomType{NagIntegrationPackage}.

d01fcfAnnaType(): NumericalIntegrationCategory == Result add
  EF2 ==> ExpressionFunctions2
  EDF ==> Expression DoubleFloat
  LDF ==> List DoubleFloat
  SDF ==> Stream DoubleFloat
  DF ==> DoubleFloat
  FI ==> Fraction Integer
  EFI ==> Expression Fraction Integer
  SOCDF ==> Segment OrderedCompletion DoubleFloat
  NIA ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
  MDNIA ==> Record(fn:EDF,range:List SOCDF,abserr:DF,relerr:DF)
  INT ==> Integer
  BOP ==> BasicOperator
  S ==> Symbol
  ST ==> String
  LST ==> List String
  RT ==> RoutinesTable
  Rep:=Result
  import Rep, d01AgentsPackage, NagIntegrationPackage

measure(R:RT,args:MDNIA) ==
  ext:Result := empty()$Result
  segs := args.range
  vars := variables(args.fn)$EDF
  for i in 1..# vars repeat
    nia:NIA := [vars.i,args.fn,segs.i,args.abserr,args.relerr]
    not rangeIsFinite(nia) case finite => return
      [0.0,"d01fcf is not a suitable routine for infinite integrals",ext]
  [getMeasure(R,d01fcf@S)$RT, "Recommended is d01fcf", ext]

numericalIntegration(args:MDNIA,hints:Result) ==
  import Integer
  segs := args.range
  dim := # segs

```

```

err := args.reterr
low:Matrix DF := matrix([[getlo(segs.i) for i in 1..dim]])$(Matrix DF)
high:Matrix DF := matrix([[gethi(segs.i) for i in 1..dim]])$(Matrix DF)
b:Float := getButtonValue("d01fcf", "functionEvaluations")$AttributeButtons
a:Float:= exp(1.1513*(1.0/(2.0*(1.0-b))))
alpha:INT := 2**dim+2*dim**2+2*dim+1
d:Float := max(1.e-3,nthRoot(convert(err)@Float,4))$Float
minpts:INT := (fEvals := wholePart(a))*wholePart(alpha::Float/d)
maxpts:INT := 5*minpts
lenwrk:INT := (dim+2)*(1+(33*fEvals))
ArgsFn := map(x+>convert(x)$DF,args.fn)$EF2(DF,Float)
f : Union(fn:FileName,fp:Asp4(FUNCTN)) := [retract(ArgsFn)$Asp4(FUNCTN)]
out:Result := d01fcf(dim,low,high,maxpts,err,lenwrk,minpts,-1,f)
changeName(finval@Symbol,result@Symbol,out)

```

— D01FCFA.dotabb —

```

"D01FCFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=D01FCFA"]
"TRANFUN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TRANFUN"]
"D01FCFA" -> "TRANFUN"

```

domain D01GBFA d01gbfAnnaType

— d01gbfAnnaType.input —

```

)set break resume
)sys rm -f d01gbfAnnaType.output
)spool d01gbfAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d01gbfAnnaType
--R
--R d01gbfAnnaType is a domain constructor
--R Abbreviation for d01gbfAnnaType is D01GBFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for D01GBFA
--R

```

```

--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger        latex : % -> String
--R ?~=? : (%,% ) -> Boolean
--R measure : (RoutinesTable,Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedComple
--R measure : (RoutinesTable,Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(Order
--R numericalIntegration : (Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedComple
--R numericalIntegration : (Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(Ordere
--R
--E 1

```

```

)spool
)lisp (bye)

```

— d01gbfAnnaType.help —

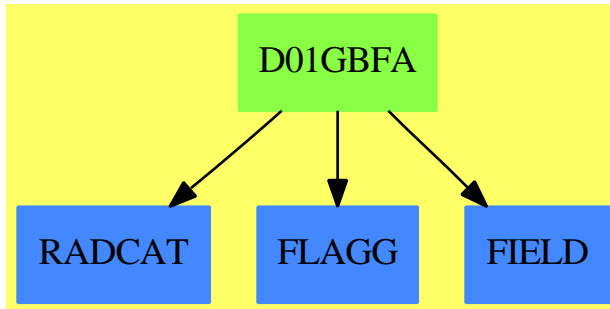
```

=====
d01gbfAnnaType examples
=====

```

d01gbfAnnaType is a domain of NumericalIntegrationCategory for the NAG routine D01GBF, a numerical integration routine which can handle multi-dimensional quadrature over a finite region. The function measure measures the usefulness of the routine D01GBF for the given problem. The function numericalIntegration performs the integration by using NagIntegrationPackage.

See Also:
o)show d01gbfAnnaType

d01gbfAnnaType (D01GBFA)**Exports:**

coerce hash latex measure numericalIntegration ?? ?~=?

— domain D01GBFA d01gbfAnnaType —

```

)abbrev domain D01GBFA d01gbfAnnaType
++ Author: Brian Dupee
++ Date Created: March 1994
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{d01gbfAnnaType} is a domain of
++ \axiomType{NumericalIntegrationCategory}
++ for the NAG routine D01GBF, a numerical integration routine which can
++ handle multi-dimensional quadrature over a finite region. The
++ function \axiomFun{measure} measures the usefulness of the routine D01GBF
++ for the given problem. The function \axiomFun{numericalIntegration}
++ performs the integration by using \axiomType{NagIntegrationPackage}.

d01gbfAnnaType(): NumericalIntegrationCategory == Result add
  EF2 ==> ExpressionFunctions2
  EDF ==> Expression DoubleFloat
  LDF ==> List DoubleFloat
  SDF ==> Stream DoubleFloat
  DF ==> DoubleFloat
  FI ==> Fraction Integer
  EFI ==> Expression Fraction Integer
  SOCDF ==> Segment OrderedCompletion DoubleFloat
  NIA ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
  MDNIA ==> Record(fn:EDF,range:List SOCDF,abserr:DF,relerr:DF)
  INT ==> Integer
  BOP ==> BasicOperator
  S ==> Symbol
  ST ==> String
  LST ==> List String
  RT ==> RoutinesTable
  
```

```

Rep:=Result
import Rep, d01AgentsPackage, NagIntegrationPackage

measure(R:RT,args:MDNIA) ==
  ext:Result := empty()$Result
  (rel := args.relerr) < 0.01 :: DF =>
    [0.1, "d01gbf: The relative error requirement is too small",ext]
  segs := args.range
  vars := variables(args.fn)$EDF
  for i in 1..# vars repeat
    nia:NIA := [vars.i,args.fn,segs.i,args.abterr,rel]
    not rangeIsFinite(nia) case finite => return
    [0.0,"d01gbf is not a suitable routine for infinite integrals",ext]
  [getMeasure(R,d01gbf@S)$RT, "Recommended is d01gbf", ext]

numericalIntegration(args:MDNIA,hints:Result) ==
  import Integer
  segs := args.range
  dim:INT := # segs
  low:Matrix DF := matrix([[getlo(segs.i) for i in 1..dim]])$(Matrix DF)
  high:Matrix DF := matrix([[gethi(segs.i) for i in 1..dim]])$(Matrix DF)
  b:Float := getButtonValue("d01gbf","functionEvaluations")$AttributeButtons
  a:Float:= exp(1.1513*(1.0/(2.0*(1.0-b))))
  maxcls:INT := 1500*(dim+1)*(fEvals:INT := wholePart(a))
  mincls:INT := 300*fEvals
  c:Float := nthRoot((maxcls:Float)/4.0,dim)$Float
  lenwrk:INT := 3*dim*(d:INT := wholePart(c))+10*dim
  wrkstr:Matrix DF := matrix([[0$DF for i in 1..lenwrk]])$(Matrix DF)
  ArgsFn := map(x-->convert(x)$DF,args.fn)$EF2(DF,Float)
  f : Union(fn:FileName,fp:Asp4(FUNCTN)) := [retract(ArgsFn)$Asp4(FUNCTN)]
  out:Result := _
    d01gbf(dim,low,high,maxcls,args.relerr,lenwrk,mincls,wrkstr,-1,f)
  changeName(finest@Symbol,result@Symbol,out)

```

— D01GBFA.dotabb —

```

"D01GBFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=D01GBFA"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"D01GBFA" -> "FIELD"
"D01GBFA" -> "RADCAT"
"D01GBFA" -> "FLAGG"

```

domain D01TRNS d01TransformFunctionType

— d01TransformFunctionType.input —

```

)set break resume
)sys rm -f d01TransformFunctionType.output
)spool d01TransformFunctionType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d01TransformFunctionType
--R
--R d01TransformFunctionType is a domain constructor
--R Abbreviation for d01TransformFunctionType is D01TRNS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for D01TRNS
--R
--R----- Operations -----
--R ?? : (% ,%) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger      latex : % -> String
--R ~=? : (% ,%) -> Boolean
--R measure : (RoutinesTable,Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedCompletion(DoubleFloat),
--R measure : (RoutinesTable,Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedCompletion(DoubleFloat),
--R numericalIntegration : (Record(fn: Expression(DoubleFloat),range: List(Segment(OrderedCompletion(DoubleFloat),
--R numericalIntegration : (Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedCompletion(DoubleFloat),
--R
--E 1

)spool
)lisp (bye)

```

— d01TransformFunctionType.help —

```

=====
d01TransformFunctionType examples
=====

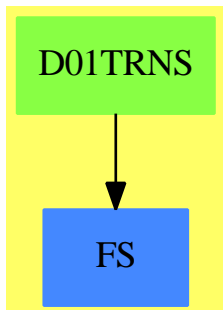
```

Since an infinite integral cannot be evaluated numerically it is necessary to transform the integral onto finite ranges. d01TransformFunctionType uses the mapping $x \rightarrow 1/x$ and contains the functions `measure` and `numericalIntegration`.

See Also:

o)show d01TransformFunctionType

d01TransformFunctionType (D01TRNS)



Exports:

coerce hash latex measure numericalIntegration ==? ?~=?

— domain D01TRNS d01TransformFunctionType —

```

)abbrev domain D01TRNS d01TransformFunctionType
++ Author: Brian Dupee
++ Date Created: April 1994
++ Date Last Updated: December 1997
++ Description:
++ Since an infinite integral cannot be evaluated numerically
++ it is necessary to transform the integral onto finite ranges.
++ \axiomType{d01TransformFunctionType} uses the mapping \spad{x -> 1/x}
++ and contains the functions \axiomFun{measure} and
++ \axiomFun{numericalIntegration}.

EDF ==> Expression DoubleFloat
EEDF ==> Equation Expression DoubleFloat
FI ==> Fraction Integer
EFI ==> Expression Fraction Integer
EEFI ==> Equation Expression Fraction Integer
EF2 ==> ExpressionFunctions2
DF ==> DoubleFloat
F ==> Float
SOCDF ==> Segment OrderedCompletion DoubleFloat
OCDF ==> OrderedCompletion DoubleFloat
NIA ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
INT ==> Integer
PI ==> PositiveInteger
  
```

```

HINT ==> Record(str:String,fn:EDF,range:SOCDF,ext:Result)
S ==> Symbol
ST ==> String
LST ==> List String
Measure ==> Record(measure:F,explanations:ST,extra:Result)
MS ==> Record(measure:F,name:ST,explanations:LST,extra:Result)

d01TransformFunctionType():NumericalIntegrationCategory == Result add
  Rep:=Result
  import d01AgentsPackage,Rep

rec2any(re:Record(str:ST,fn:EDF,range:SOCDF)):Any ==
  coerce(re)$AnyFunctions1(Record(str:ST,fn:EDF,range:SOCDF))

changeName(ans:Result,name:ST):Result ==
  sy:S := coerce(name "Answer")$S
  anyAns:Any := coerce(ans)$AnyFunctions1(Result)
  construct([[sy,anyAns]])$Result

getIntegral(args:NIA,hint:HINT) : Result ==
  Args := copy args
  Args.fn := hint.fn
  Args.range := hint.range
  integrate(Args::NumericalIntegrationProblem)$AnnaNumericalIntegrationPackage

transformFunction(args:NIA) : NIA ==
  Args := copy args
  Var := Args.var :: EFI
  NewVar:EFI := inv(Var)$EFI
  VarEqn:EEFI:=equation(Var,NewVar)$EEFI -- turn it into an equation
  Afn:EFI := edf2efi(Args.fn)$ExpertSystemToolsPackage
  Afn := subst(Afn,VarEqn)$EFI
  Var2:EFI := Var**2
  Afn:= simplify(Afn/Var2)$TranscendentalManipulations(FI,EFI)
  Args.fn:= map(x+>convert(x)$FI,Afn)$EF2(FI,DF)
  Args

doit(seg:SOCDF,args:NIA):MS ==
  Args := copy args
  Args.range := seg
  measure(Args::NumericalIntegrationProblem)$AnnaNumericalIntegrationPackage

transform(c:Boolean,args:NIA):Measure ==
  if c then
    l := coerce(recip(lo(args.range)))@OCDF
    Seg:SOCDF := segment(0$OCDF,l)
  else
    h := coerce(recip(hi(args.range)))@OCDF
    Seg:SOCDF := segment(h,0$OCDF)
  Args := transformFunction(args)

```

```

m:MS := doit(Seg,Args)
out1:ST :=
  "The recommendation is to transform the function and use " m.name
out2:List(HINT) := [[m.name,Args.fn,Seg,m.extra]]
out2Any:Any := coerce(out2)$AnyFunctions1(List(HINT))
ex:Record(key:S,entry:Any) := [d01transformextra@S,out2Any]
extr:Result := construct([ex])$Result
[m.measure,out1,extr]

split(c:PI,args:NIA):Measure ==
  Args := copy args
  Args.relerr := Args.relerr/2
  Args.abserr := Args.abserr/2
  if (c = 1)@Boolean then
    seg1:SOCDF := segment(-1$OCDF,1$OCDF)
  else if (c = 2)@Boolean then
    seg1 := segment(lo(Args.range),1$OCDF)
  else
    seg1 := segment(-1$OCDF,hi(Args.range))
  m1:MS := doit(seg1,Args)
  Args := transformFunction Args
  if (c = 2)@Boolean then
    seg2:SOCDF := segment(0$OCDF,1$OCDF)
  else if (c = 3)@Boolean then
    seg2 := segment(-1$OCDF,0$OCDF)
  else seg2 := seg1
  m2:MS := doit(seg2,Args)
  m1m:F := m1.measure
  m2m:F := m2.measure
  m:F := m1m*m2m/((m1m*m2m)+(1.0-m1m)*(1.0-m2m))
  out1:ST := "The recommendation is to transform the function and use "
    m1.name " and " m2.name
  out2:List(HINT) :=
    [[m1.name,args.fn,seg1,m1.extra],[m2.name,Args.fn,seg2,m2.extra]]
  out2Any:Any := coerce(out2)$AnyFunctions1(List(HINT))
  ex:Record(key:S,entry:Any) := [d01transformextra@S,out2Any]
  extr:Result := construct([ex])$Result
  [m,out1,extr]

measure(R:RoutinesTable,args:NIA) ==
  Range:=rangeIsFinite(args)
  Range case bothInfinite => split(1,args)
  Range case upperInfinite =>
    positive?(lo(args.range))$OCDF =>
      transform(true,args)
      split(2,args)
  Range case lowerInfinite =>
    negative?(hi(args.range))$OCDF =>
      transform(false,args)
      split(3,args)

```

```

numericalIntegration(args:NIA,hints:Result) ==
  mainResult:DF := mainAbserr:DF := 0$DF
  ans:Result := empty()$Result
  hla:Any := coerce(search((d01transformextra@S),hints)$Result)@Any
  hintList := retract(hla)$AnyFunctions1(List(HINT))
  methodName:ST := empty()$ST
  repeat
    if (empty?(hintList)$(List(HINT)))
      then leave
    item := first(hintList)$List(HINT)
    a:Result := getIntegral(args,item)
    anyRes := coerce(search((result@S),a)$Result)@Any
    midResult := retract(anyRes)$AnyFunctions1(DF)
    anyErr := coerce(search((abserr pretend S),a)$Result)@Any
    midAbserr := retract(anyErr)$AnyFunctions1(DF)
    mainResult := mainResult+midResult
    mainAbserr := mainAbserr+midAbserr
    if (methodName = item.str)@Boolean then
      methodName := concat([item.str,"1"])$ST
    else
      methodName := item.str
    ans := concat(ans,changeName(a,methodName))$ExpertSystemToolsPackage
    hintList := rest(hintList)$(List(HINT))
  anyResult := coerce(mainResult)$AnyFunctions1(DF)
  anyAbserr := coerce(mainAbserr)$AnyFunctions1(DF)
  recResult:Record(key:S,entry:Any):=[result@S,anyResult]
  recAbserr:Record(key:S,entry:Any):=[abserr pretend S,anyAbserr]
  insert!(recAbserr,insert!(recResult,ans))$Result

```

— D01TRNS.dotabb —

"D01TRNS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=D01TRNS"]
 "FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
 "D01TRNS" -> "FS"

domain D02BBFA d02bbfAnnaType

— d02bbfAnnaType.input —

```
)set break resume
```

```

)sys rm -f d02bbfAnnaType.output
)spool d02bbfAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d02bbfAnnaType
--R
--R d02bbfAnnaType is a domain constructor
--R Abbreviation for d02bbfAnnaType is D02BBFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for D02BBFA
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger        latex : % -> String
--R ?~=? : (%,% ) -> Boolean
--R ODESolve : Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(DoubleFloat)),y
--R measure : (RoutinesTable,Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(D
--R
--E 1

)spool
)lisp (bye)

```

— d02bbfAnnaType.help —

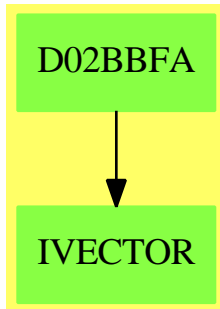
=====

d02bbfAnnaType examples

=====

d02bbfAnnaType is a domain of
 OrdinaryDifferentialEquationsInitialValueProblemSolverCategory
 for the NAG routine D02BBF, a ODE routine which uses an Runge-Kutta method
 to solve a system of differential equations. The function measure
 measures the usefulness of the routine D02BBF for the given problem.
 The function ODESolve performs the integration by using
 NagOrdinaryDifferentialEquationsPackage.

See Also:
 o)show d02bbfAnnaType

d02bbfAnnaType (D02BBFA)**Exports:**

coerce hash latex measure ODEsolve ?=? ?~=?

— domain D02BBFA d02bbfAnnaType —

```

)abbrev domain D02BBFA d02bbfAnnaType
++ Author: Brian Dupee
++ Date Created: February 1995
++ Date Last Updated: January 1996
++ Description:
++ \axiomType{d02bbfAnnaType} is a domain of
++ \axiomType{OrdinaryDifferentialEquationsInitialValueProblemSolverCategory}
++ for the NAG routine D02BBF, a ODE routine which uses an
++ Runge-Kutta method to solve a system of differential
++ equations. The function \axiomFun{measure} measures the
++ usefulness of the routine D02BBF for the given problem. The
++ function \axiomFun{ODEsolve} performs the integration by using
++ \axiomType{NagOrdinaryDifferentialEquationsPackage}.

d02bbfAnnaType():OrdinaryDifferentialEquationsSolverCategory == Result add
-- Runge Kutta

EDF ==> Expression DoubleFloat
LDF ==> List DoubleFloat
MDF ==> Matrix DoubleFloat
DF ==> DoubleFloat
F ==> Float
FI ==> Fraction Integer
EFI ==> Expression Fraction Integer
SOCDF ==> Segment OrderedCompletion DoubleFloat
VEDF ==> Vector Expression DoubleFloat
VEF ==> Vector Expression Float
EF ==> Expression Float
VDF ==> Vector DoubleFloat
VMF ==> Vector MachineFloat
  
```

```

MF ==> MachineFloat
ODEA ==> Record(xinit:DF,xend:DF,fn:VEDF,yinit:LDF,intvals:LDF,_
               g:EDF,abserr:DF,relerr:DF)
RSS ==> Record(stiffnessFactor:F,stabilityFactor:F)
INT ==> Integer
EF2 ==> ExpressionFunctions2

import d02AgentsPackage, NagOrdinaryDifferentialEquationsPackage
import AttributeButtons

accuracyCF(ode:ODEA):F ==
  b := getButtonValue("d02bbf","accuracy")$AttributeButtons
  accuracyIntensityValue := combineFeatureCompatibility(b,accuracyIF(ode))
  accuracyIntensityValue > 0.999 => 0$F
  0.8*exp(-((10*accuracyIntensityValue)**3)$F/266)$F

stiffnessCF(stiffnessIntensityValue:F):F ==
  b := getButtonValue("d02bbf","stiffness")$AttributeButtons
  0.5*exp(-(2*combineFeatureCompatibility(b,stiffnessIntensityValue)**2)$F

stabilityCF(stabilityIntensityValue:F):F ==
  b := getButtonValue("d02bbf","stability")$AttributeButtons
  0.5 * cos(combineFeatureCompatibility(b,stabilityIntensityValue))$F

expenseOfEvaluationCF(ode:ODEA):F ==
  b := getButtonValue("d02bbf","expense")$AttributeButtons
  expenseOfEvaluationIntensityValue :=
    combineFeatureCompatibility(b,expenseOfEvaluationIF(ode))
  0.35+0.2*exp(-(2.0*expenseOfEvaluationIntensityValue)**3)$F

measure(R:RoutinesTable,args:ODEA) ==
  m := getMeasure(R,d02bbf :: Symbol)$RoutinesTable
  ssf := stiffnessAndStabilityOfODEIF args
  m := combineFeatureCompatibility(m,[accuracyCF(args),
    stiffnessCF(ssf.stiffnessFactor),
    expenseOfEvaluationCF(args),
    stabilityCF(ssf.stabilityFactor)])
  [m,"Runge-Kutta Merson method"]

ODESolve(ode:ODEA) ==
  i:LDF := ode.intvals
  M := inc(# i)$INT
  irelab := 0$INT
  if positive?(a := ode.abserr) then
    inc(irelab)$INT
  if positive?(r := ode.relerr) then
    inc(irelab)$INT
  if positive?(a+r) then
    tol:DF := a + r
  else

```

```

    tol := float(1,-4,10)$DF
asp7:Union(fn:FileName,fp:Asp7(FCN)) :=
    [retract(vedf2vef(ode.fn)$ExpertSystemToolsPackage)$Asp7(FCN)]
asp8:Union(fn:FileName,fp:Asp8(OUTPUT)) :=
    [coerce(ldf2vmf(i)$ExpertSystemToolsPackage)$Asp8(OUTPUT)]
d02bbf(ode.xend,M,# ode.fn,irelab,ode.xinit,matrix([ode.yinit])$MDF,
    tol,-1,asp7,asp8)

```

— D02BBFA.dotabb —

```

"D02BBFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=D02BBFA"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"D02BBFA" -> "IVECTOR"

```

domain D02BHFA d02bhfAnnaType

— d02bhfAnnaType.input —

```

)set break resume
)sys rm -f d02bhfAnnaType.output
)spool d02bhfAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d02bhfAnnaType
--R
--R d02bhfAnnaType is a domain constructor
--R Abbreviation for d02bhfAnnaType is D02BHFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for D02BHFA
--R
--R----- Operations -----
--R ?? : (% ,%) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger      latex : % -> String
--R ?~=? : (% ,%) -> Boolean
--R ODESolve : Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(DoubleFloat)),yinit: List(List(DoubleFloat)))
--R measure : (RoutinesTable,Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(DoubleFloat))))
--R

```



```
--E 1
```

```
)spool
)lisp (bye)
```

— d02bhfAnnaType.help —

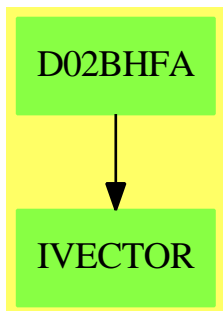
```
=====
d02bhfAnnaType examples
=====
```

d02bhfAnnaType is a domain of
 OrdinaryDifferentialEquationsInitialValueProblemSolverCategory
 for the NAG routine D02BHF, a ODE routine which uses a Runge-Kutta method
 to solve a system of differential equations. The function measure measures
 the usefulness of the routine D02BHF for the given problem. The function
 ODESolve performs the integration by using
 NagOrdinaryDifferentialEquationsPackage.

See Also:

o)show d02bhfAnnaType

d02bhfAnnaType (D02BHFA)



Exports:

coerce hash latex measure ODESolve ?=? ?~=?

— domain D02BHFA d02bhfAnnaType —

```
)abbrev domain D02BHFA d02bhfAnnaType
```

```

++ Author: Brian Dupee
++ Date Created: February 1995
++ Date Last Updated: January 1996
++ Description:
++ \axiomType{d02bhfAnnaType} is a domain of
++ \axiomType{OrdinaryDifferentialEquationsInitialValueProblemSolverCategory}
++ for the NAG routine D02BHF, a ODE routine which uses an
++ Runge-Kutta method to solve a system of differential
++ equations. The function \axiomFun{measure} measures the
++ usefulness of the routine D02BHF for the given problem. The
++ function \axiomFun{ODESolve} performs the integration by using
++ \axiomType{NagOrdinaryDifferentialEquationsPackage}.

d02bhfAnnaType():OrdinaryDifferentialEquationsSolverCategory == Result add
-- Runge Kutta
EDF ==> Expression DoubleFloat
LDF ==> List DoubleFloat
MDF ==> Matrix DoubleFloat
DF ==> DoubleFloat
F ==> Float
FI ==> Fraction Integer
EFI ==> Expression Fraction Integer
SOCDF ==> Segment OrderedCompletion DoubleFloat
VEDF ==> Vector Expression DoubleFloat
VEF ==> Vector Expression Float
EF ==> Expression Float
VDF ==> Vector DoubleFloat
VMF ==> Vector MachineFloat
MF ==> MachineFloat
ODEA ==> Record(xinit:DF,xend:DF,fn:VEDF,yinit:LDF,intvals:LDF,
               g:EDF,abserr:DF,relerr:DF)
RSS ==> Record(stiffnessFactor:F,stabilityFactor:F)
INT ==> Integer
EF2 ==> ExpressionFunctions2

import d02AgentsPackage, NagOrdinaryDifferentialEquationsPackage
import AttributeButtons

accuracyCF(ode:ODEA):F ==
  b := getButtonValue("d02bhf","accuracy")$AttributeButtons
  accuracyIntensityValue := combineFeatureCompatibility(b,accuracyIF(ode))
  accuracyIntensityValue > 0.999 => 0$F
  0.8*exp(-(10*accuracyIntensityValue)**3)$F/266)$F

stiffnessCF(stiffnessIntensityValue:F):F ==
  b := getButtonValue("d02bhf","stiffness")$AttributeButtons
  0.5*exp(-(2*combineFeatureCompatibility(b,stiffnessIntensityValue)**2)$F

stabilityCF(stabilityIntensityValue:F):F ==
  b := getButtonValue("d02bhf","stability")$AttributeButtons

```

```

0.5 * cos(combineFeatureCompatibility(b,stabilityIntensityValue))$F

expenseOfEvaluationCF(ode:ODEA):F ==
  b := getButtonValue("d02bhf","expense")$AttributeButtons
  expenseOfEvaluationIntensityValue :=
    combineFeatureCompatibility(b,expenseOfEvaluationIF(ode))
  0.35+0.2*exp(-(2.0*expenseOfEvaluationIntensityValue)**3)$F

measure(R:RoutinesTable,args:ODEA) ==
  m := getMeasure(R,d02bhf :: Symbol)$RoutinesTable
  ssf := stiffnessAndStabilityOfODEIF args
  m := combineFeatureCompatibility(m,[accuracyCF(args),
    stiffnessCF(ssf.stiffnessFactor),
    expenseOfEvaluationCF(args),
    stabilityCF(ssf.stabilityFactor)])
  [m,"Runge-Kutta Merson method"]

ODESolve(ode:ODEA) ==
  irelab := 0$INT
  if positive?(a := ode.abserr) then
    inc(irelab)$INT
  if positive?(r := ode.relerr) then
    inc(irelab)$INT
  if positive?(a+r) then
    tol := max(a,r)$DF
  else
    tol:DF := float(1,-4,10)$DF
  asp7:Union(fn:FileName,fp:Asp7(FCN)) :=
    [retract(e:VEF := vedf2vef(ode.fn)$ExpertSystemToolsPackage)$Asp7(FCN)]
  asp9:Union(fn:FileName,fp:Asp9(G)) :=
    [retract(edf2ef(ode.g)$ExpertSystemToolsPackage)$Asp9(G)]
  d02bhf(ode.xend,# e,irelab,0$DF,ode.xinit,matrix([ode.yinit])$MDF,
    tol,-1,asp9,asp7)

```

— D02BHFA.dotabb —

```

"D02BHFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=D02BHFA"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"D02BHFA" -> "IVECTOR"

```

domain D02CJFA d02cjfAnnaType

— d02cjfAnnaType.input —

```

)set break resume
)sys rm -f d02cjfAnnaType.output
)spool d02cjfAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d02cjfAnnaType
--R
--R d02cjfAnnaType is a domain constructor
--R Abbreviation for d02cjfAnnaType is D02CJFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for D02CJFA
--R
--R----- Operations -----
--R ?? : (% ,%) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger      latex : % -> String
--R ?~=? : (% ,%) -> Boolean
--R ODESolve : Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(DoubleFloat)),yinit: List(
--R measure : (RoutinesTable,Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(DoubleFloat))
--R
--E 1

)spool
)lisp (bye)

```

— d02cjfAnnaType.help —

```

=====
d02cjfAnnaType examples
=====

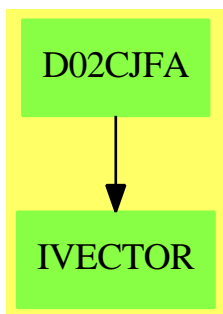
```

d02cjfAnnaType is a domain of OrdinaryDifferentialEquationsInitialValueProblemSolverCategory for the NAG routine D02CJF, a ODE routine which uses an Adams-Moulton-Bashworth method to solve a system of differential equations. The function measure measures the usefulness of the routine D02CJF for the given problem. The function ODESolve performs the integration by using NagOrdinaryDifferentialEquationsPackage.

See Also:

```
o )show d02cjfAnnaType
```

d02cjfAnnaType (D02CJFA)



Exports:

```
coerce hash latex measure ODESolve ==? ?~=?
```

— domain D02CJFA d02cjfAnnaType —

```

)abbrev domain D02CJFA d02cjfAnnaType
++ Author: Brian Dupee
++ Date Created: February 1995
++ Date Last Updated: January 1996
++ Description:
++ \axiomType{d02cjfAnnaType} is a domain of
++ \axiomType{OrdinaryDifferentialEquationsInitialValueProblemSolverCategory}
++ for the NAG routine D02CJF, a ODE routine which uses an
++ Adams-Moulton-Bashworth method to solve a system of differential
++ equations. The function \axiomFun{measure} measures the
++ usefulness of the routine D02CJF for the given problem. The
++ function \axiomFun{ODESolve} performs the integration by using
++ \axiomType{NagOrdinaryDifferentialEquationsPackage}.

d02cjfAnnaType():OrdinaryDifferentialEquationsSolverCategory == Result add
-- Adams
EDF ==> Expression DoubleFloat
LDF ==> List DoubleFloat
MDF ==> Matrix DoubleFloat
DF ==> DoubleFloat
F ==> Float
FI ==> Fraction Integer
EFI ==> Expression Fraction Integer
  
```

```

SOCDF ==> Segment OrderedCompletion DoubleFloat
VEDF ==> Vector Expression DoubleFloat
VEF ==> Vector Expression Float
EF ==> Expression Float
VDF ==> Vector DoubleFloat
VMF ==> Vector MachineFloat
MF ==> MachineFloat
ODEA ==> Record(xinit:DF,xend:DF,fn:VEDF,yinit:LDF,intvals:LDF,_
                g:EDF,abserr:DF,relerr:DF)
RSS ==> Record(stiffnessFactor:F,stabilityFactor:F)
INT ==> Integer
EF2 ==> ExpressionFunctions2

import d02AgentsPackage, NagOrdinaryDifferentialEquationsPackage

accuracyCF(ode:ODEA):F ==
  b := getButtonValue("d02cjf","accuracy")$AttributeButtons
  accuracyIntensityValue := combineFeatureCompatibility(b,accuracyIF(ode))
  accuracyIntensityValue > 0.9999 => 0$F
  0.6*(cos(accuracyIntensityValue*(pi()$F)/2)$F)**0.755

stiffnessCF(ode:ODEA):F ==
  b := getButtonValue("d02cjf","stiffness")$AttributeButtons
  ssf := stiffnessAndStabilityOfODEIF ode
  stiffnessIntensityValue :=
    combineFeatureCompatibility(b,ssf.stiffnessFactor)
  0.5*exp(-(1.1*stiffnessIntensityValue)**3)$F

measure(R:RoutinesTable,args:ODEA) ==
  m := getMeasure(R,d02cjf :: Symbol)$RoutinesTable
  m := combineFeatureCompatibility(m,[accuracyCF(args), stiffnessCF(args)])
  [m,"Adams method"]

ODESolve(ode:ODEA) ==
  i:LDF := ode.intvals
  if empty?(i) then
    i := [ode.xend]
  M := inc(# i)$INT
  if positive?((a := ode.abserr)*(r := ode.relerr))$DF then
    ire:String := "D"
  else
    if positive?(a) then
      ire:String := "A"
    else
      ire:String := "R"
  tol := max(a,r)$DF
  asp7:Union(fn:FileName,fp:Asp7(FCN)) :=
    [retract(e:VEF := vedf2vef(ode.fn)$ExpertSystemToolsPackage)$Asp7(FCN)]
  asp8:Union(fn:FileName,fp:Asp8(OUTPUT)) :=
    [coerce(ldf2vmf(i)$ExpertSystemToolsPackage)$Asp8(OUTPUT)]

```

```

asp9:Union(fn:FileName,fp:Asp9(G)) :=
  [retract(edf2ef(ode.g)$ExpertSystemToolsPackage)$Asp9(G)]
d02cjf(ode.xend,M,# e,tol,ire,ode.xinit,matrix([ode.yinit])$MDF,
      -1,asp9,asp7,asp8)

```

— D02CJFA.dotabb —

```

"D02CJFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=D02CJFA"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"D02CJFA" -> "IVECTOR"

```

domain D02EJFA d02ejfAnnaType

— d02ejfAnnaType.input —

```

)set break resume
)sys rm -f d02ejfAnnaType.output
)spool d02ejfAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d02ejfAnnaType
--R
--R d02ejfAnnaType is a domain constructor
--R Abbreviation for d02ejfAnnaType is D02EJFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for D02EJFA
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger        latex : % -> String
--R ?^=? : (%,% ) -> Boolean
--R ODESolve : Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(DoubleFloat)),y
--R measure : (RoutinesTable,Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(D
--R
--E 1

)spool
)lisp (bye)

```

— d02ejfAnnaType.help —

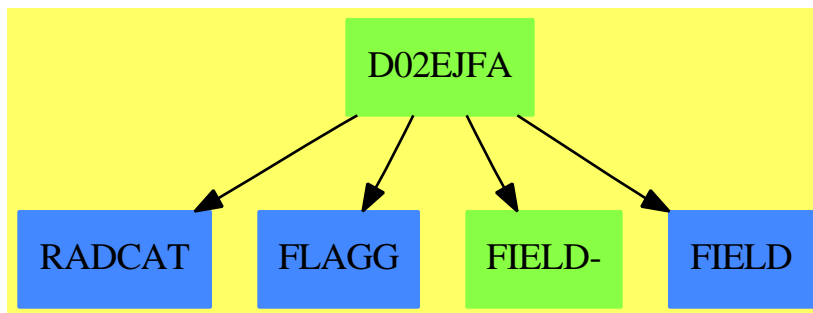
```
=====
d02ejfAnnaType examples
=====
```

d02ejfAnnaType is a domain of OrdinaryDifferentialEquationsInitialValueProblemSolverCategory for the NAG routine D02EJF, a ODE routine which uses a backward differentiation formula method to handle a stiff system of differential equations. The function measure measures the usefulness of the routine D02EJF for the given problem. The function ODESolve performs the integration by using NagOrdinaryDifferentialEquationsPackage.

See Also:

o)show d02ejfAnnaType

d02ejfAnnaType (D02EJFA)



Exports:

coerce hash latex measure ODESolve ?=? ?~=?

— domain D02EJFA d02ejfAnnaType —

```
)abbrev domain D02EJFA d02ejfAnnaType
++ Author: Brian Dupee
++ Date Created: February 1995
++ Date Last Updated: January 1996
++ Description:
++ \axiomType{d02ejfAnnaType} is a domain of
```



```

++ \axiomType{OrdinaryDifferentialEquationsInitialValueProblemSolverCategory}
++ for the NAG routine D02EJF, a ODE routine which uses a backward
++ differentiation formulae method to handle a stiff system
++ of differential equations. The function \axiomFun{measure} measures
++ the usefulness of the routine D02EJF for the given problem. The
++ function \axiomFun{ODESolve} performs the integration by using
++ \axiomType{NagOrdinaryDifferentialEquationsPackage}.

d02ejfAnnaType():OrdinaryDifferentialEquationsSolverCategory == Result add
-- BDF "Stiff"
EDF ==> Expression DoubleFloat
LDF ==> List DoubleFloat
MDF ==> Matrix DoubleFloat
DF ==> DoubleFloat
F ==> Float
FI ==> Fraction Integer
EFI ==> Expression Fraction Integer
SOCDF ==> Segment OrderedCompletion DoubleFloat
VEDF ==> Vector Expression DoubleFloat
VEF ==> Vector Expression Float
EF ==> Expression Float
VDF ==> Vector DoubleFloat
VMF ==> Vector MachineFloat
MF ==> MachineFloat
ODEA ==> Record(xinit:DF,xend:DF,fn:VEDF,yinit:LDF,intvals:LDF,
               g:EDF,abserr:DF,relerr:DF)
RSS ==> Record(stiffnessFactor:F,stabilityFactor:F)
INT ==> Integer
EF2 ==> ExpressionFunctions2

import d02AgentsPackage, NagOrdinaryDifferentialEquationsPackage

accuracyCF(ode:ODEA):F ==
  b := getButtonValue("d02ejf","accuracy")$AttributeButtons
  accuracyIntensityValue := combineFeatureCompatibility(b,accuracyIF(ode))
  accuracyIntensityValue > 0.999 => 0$F
  0.5*exp(-(10*accuracyIntensityValue)**3)$F/250)$F

intermediateResultsCF(ode:ODEA):F ==
  intermediateResultsIntensityValue := intermediateResultsIF(ode)
  i := 0.5 * exp(-(intermediateResultsIntensityValue/1.649)**3)$F
  a := accuracyCF(ode)
  i+(0.5-i)*(0.5-a)

stabilityCF(ode:ODEA):F ==
  b := getButtonValue("d02ejf","stability")$AttributeButtons
  ssf := stiffnessAndStabilityOfODEIF ode
  stabilityIntensityValue :=
    combineFeatureCompatibility(b,ssf.stabilityFactor)
  0.68 - 0.5 * exp(-(stabilityIntensityValue)**3)$F

```

```

expenseOfEvaluationCF(ode:ODEA):F ==
  b := getButtonValue("d02ejf","expense")$AttributeButtons
  expenseOfEvaluationIntensityValue :=
    combineFeatureCompatibility(b,expenseOfEvaluationIF(ode))
  0.5 * exp(-(1.7*expenseOfEvaluationIntensityValue)**3)$F

systemSizeCF(args:ODEA):F ==
  (1$F - systemSizeIF(args))/2.0

measure(R:RoutinesTable,args:ODEA) ==
  arg := copy args
  m := getMeasure(R,d02ejf :: Symbol)$RoutinesTable
  m := combineFeatureCompatibility(m,[intermediateResultsCF(arg),
    accuracyCF(arg),
    systemSizeCF(arg),
    expenseOfEvaluationCF(arg),
    stabilityCF(arg)])
  [m,"BDF method for Stiff Systems"]

ODESolve(ode:ODEA) ==
  i:LDF := ode.intvals
  m := inc(# i)$INT
  if positive?((a := ode.abserr)*(r := ode.relerr))$DF then
    ire:String := "D"
  else
    if positive?(a) then
      ire:String := "A"
    else
      ire:String := "R"
  if positive?(a+r)$DF then
    tol := max(a,r)$DF
  else
    tol := float(1,-4,10)$DF
  asp7:Union(fn:FileName,fp:Asp7(FCN)) :=
    [retract(e:VEF := vedf2vef(ode.fn)$ExpertSystemToolsPackage)$Asp7(FCN)]
  asp31:Union(fn:FileName,fp:Asp31(PEDERV)) :=
    [retract(e)$Asp31(PEDERV)]
  asp8:Union(fn:FileName,fp:Asp8(OUTPUT)) :=
    [coerce(ldf2vmf(i)$ExpertSystemToolsPackage)$Asp8(OUTPUT)]
  asp9:Union(fn:FileName,fp:Asp9(G)) :=
    [retract(edf2ef(ode.g)$ExpertSystemToolsPackage)$Asp9(G)]
  n:INT := # ode.yinit
  iw:INT := (12+n)*n+50
  ans := d02ejf(ode.xend,m,n,ire,iw,ode.xinit,matrix([ode.yinit])$MDF,
    tol,-1,asp9,asp7,asp31,asp8)

```

— D02EJFA.dotabb —

```
"D02EJFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=D02EJFA"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FIELD-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FIELD"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"D02EJFA" -> "FLAGG"
"D02EJFA" -> "FIELD-"
"D02EJFA" -> "FIELD"
"D02EJFA" -> "RADCAT"
```

domain D03EEFA d03eefAnnaType

— d03eefAnnaType.input —

```
)set break resume
)sys rm -f d03eefAnnaType.output
)spool d03eefAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d03eefAnnaType
--R
--R d03eefAnnaType is a domain constructor
--R Abbreviation for d03eefAnnaType is D03EEFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for D03EEFA
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean               coerce : % -> OutputForm
--R hash : % -> SingleInteger             latex : % -> String
--R ?~=? : (%,% ) -> Boolean
--R PDESolve : Record(pde: List(Expression(DoubleFloat)),constraints: List(Record(start: DoubleF
--R measure : (RoutinesTable,Record(pde: List(Expression(DoubleFloat)),constraints: List(Record(
--R
--E 1

)spool
)lisp (bye)
```

— d03eefAnnaType.help —

=====

d03eefAnnaType examples

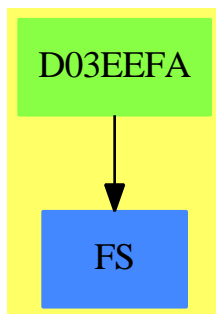
=====

d03eefAnnaType is a domain of PartialDifferentialEquationsSolverCategory
for the NAG routines D03EEF/D03EDF.

See Also:

o)show d03eefAnnaType

d03eefAnnaType (D03EEFA)



Exports:

coerce hash latex measure PDESolve ?=? ?~=?

— domain D03EEFA d03eefAnnaType —

```

)abbrev domain D03EEFA d03eefAnnaType
++ Author: Brian Dupee
++ Date Created: June 1996
++ Date Last Updated: June 1996
++ Description:
++ \axiomType{d03eefAnnaType} is a domain of
++ \axiomType{PartialDifferentialEquationsSolverCategory}
++ for the NAG routines D03EEF/D03EDF.
  
```

```

d03eefAnnaType():PartialDifferentialEquationsSolverCategory == Result add
-- 2D Elliptic PDE
  
```

```

LEDF ==> List Expression DoubleFloat
EDF ==> Expression DoubleFloat
LDF ==> List DoubleFloat
MDF ==> Matrix DoubleFloat
DF ==> DoubleFloat
F ==> Float
FI ==> Fraction Integer
VEF ==> Vector Expression Float
EF ==> Expression Float
MEF ==> Matrix Expression Float
NNI ==> NonNegativeInteger
INT ==> Integer
PDEC ==> Record(start:DF, finish:DF, grid:NNI, boundaryType:INT,
                dStart:MDF, dFinish:MDF)
PDEB ==> Record(pde:LEDF, constraints:List PDEC,
                f:List LEDF, st:String, tol:DF)

import d03AgentsPackage, NagPartialDifferentialEquationsPackage
import ExpertSystemToolsPackage

measure(R:RoutinesTable, args:PDEB) ==
  (# (args.constraints) > 2)@Boolean =>
    [0$F,"d03eef/d03edf is unsuitable for PDEs of order more than 2"]
  elliptic?(args) =>
    m := getMeasure(R,d03eef :: Symbol)$RoutinesTable
    [m,"d03eef/d03edf is suitable"]
    [0$F,"d03eef/d03edf is unsuitable for hyperbolic or parabolic PDEs"]

PDESolve(args:PDEB) ==
  xcon := first(args.constraints)
  ycon := second(args.constraints)
  nx := xcon.grid
  ny := ycon.grid
  p := args.pde
  x1 := xcon.start
  x2 := xcon.finish
  y1 := ycon.start
  y2 := ycon.finish
  lda := ((4*(nx+1)*(ny+1)+2) quo 3)$INT
  scheme:String :=
    central?((x2-x1)/2,(y2-y1)/2,args.pde) => "C"
    "U"
  asp73:Union(fn:FileName,fp:Asp73(PDEF)) :=
    [retract(vector([edf2ef u for u in p])$VEF)$Asp73(PDEF)]
  asp74:Union(fn:FileName,fp:Asp74(BNDY)) :=
    [retract(matrix([[edf2ef v for v in w] for w in args.f])$MEF)$Asp74(BNDY)]
  fde := d03eef(x1,x2,y1,y2,nx,ny,lda,scheme,-1,asp73,asp74)
  ub := new(1,nx*ny,0$DF)$MDF
  A := search(a::Symbol,fde)$Result
  A case "failed" => empty()$Result

```

```

AA := A::Any
fdea := retract(AA)$AnyFunctions1(MDF)
r := search(rhs::Symbol,fde)$Result
r case "failed" => empty()$Result
rh := r::Any
fderhs := retract(rh)$AnyFunctions1(MDF)
d03edf(nx,ny,lda,15,args.tol,0,fdea,fderhs,ub,-1)

```

— D03EEFA.dotabb —

```

"D03EEFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=D03EEFA"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"D03EEFA" -> "FS"

```

domain D03FAFA d03fafAnnaType

— d03fafAnnaType.input —

```

)set break resume
)sys rm -f d03fafAnnaType.output
)spool d03fafAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d03fafAnnaType
--R
--R d03fafAnnaType is a domain constructor
--R Abbreviation for d03fafAnnaType is D03FAFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for D03FAFA
--R
--R----- Operations -----
--R ?? : (%,% ) -> Boolean               coerce : % -> OutputForm
--R hash : % -> SingleInteger           latex : % -> String
--R ?~=? : (%,% ) -> Boolean
--R PDESolve : Record(pde: List(Expression(DoubleFloat)),constraints: List(Record(start: DoubleFloat,finish
--R measure : (RoutinesTable,Record(pde: List(Expression(DoubleFloat)),constraints: List(Record(start: Doub
--R

```

```
--E 1
```

```
)spool
)lisp (bye)
```

```
_____
```

```
— d03fafAnnaType.help —
```

```
=====
d03fafAnnaType examples
=====
```

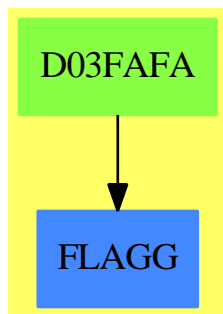
```
d03fafAnnaType is a domain of PartialDifferentialEquationsSolverCategory
for the NAG routine D03FAF.
```

```
See Also:
```

```
o )show d03fafAnnaType
```

```
_____
```

d03fafAnnaType (D03FAFA)



Exports:

```
coerce hash latex measure PDESolve ?=? ?~=?
```

```
— domain D03FAFA d03fafAnnaType —
```

```
)abbrev domain D03FAFA d03fafAnnaType
++ Author: Brian Dupee
++ Date Created: July 1996
++ Date Last Updated: July 1996
++ Description:
```

```

++ \axiomType{d03fafAnnaType} is a domain of
++ \axiomType{PartialDifferentialEquationsSolverCategory}
++ for the NAG routine D03FAF.

d03fafAnnaType():PartialDifferentialEquationsSolverCategory == Result add
-- 3D Helmholtz PDE
LEDF ==> List Expression DoubleFloat
EDF ==> Expression DoubleFloat
LDF ==> List DoubleFloat
MDF ==> Matrix DoubleFloat
DF ==> DoubleFloat
F ==> Float
FI ==> Fraction Integer
VEF ==> Vector Expression Float
EF ==> Expression Float
MEF ==> Matrix Expression Float
NNI ==> NonNegativeInteger
INT ==> Integer
PDEC ==> Record(start:DF, finish:DF, grid:NNI, boundaryType:INT,
                dStart:MDF, dFinish:MDF)
PDEB ==> Record(pde:LEDF, constraints:List PDEC,
                f:List LEDF, st:String, tol:DF)

import d03AgentsPackage, NagPartialDifferentialEquationsPackage
import ExpertSystemToolsPackage

measure(R:RoutinesTable,args:PDEB) ==
  (# (args.constraints) < 3)@Boolean =>
    [0$F,"d03faf is unsuitable for PDEs of order other than 3"]
    [0$F,"d03faf isn't finished"]

```

— D03FAFAs.dotabb —

```

"D03FAFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=D03FAFA"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"D03FAFA" -> "FLAGG"

```

Chapter 6

Chapter E

domain EFULS ElementaryFunctionsUnivariateLaurentSeries

— ElementaryFunctionsUnivariateLaurentSeries.input —

```
)set break resume
)sys rm -f ElementaryFunctionsUnivariateLaurentSeries.output
)spool ElementaryFunctionsUnivariateLaurentSeries.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ElementaryFunctionsUnivariateLaurentSeries
--E 1

)spool
)lisp (bye)
```

— ElementaryFunctionsUnivariateLaurentSeries.help —

```
=====
ElementaryFunctionsUnivariateLaurentSeries examples
=====
```

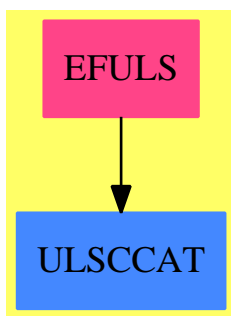
This domain provides elementary functions on any Laurent series domain over a field which was constructed from a Taylor series domain. These functions are implemented by calling the corresponding functions on the Taylor series domain. We also

provide 'partial functions' which compute transcendental functions of Laurent series when possible and return "failed" when this is not possible.

See Also:

o)show ElementaryFunctionsUnivariateLaurentSeries

ElementaryFunctionsUnivariateLaurentSeries (EFULS)



Exports:

acos	acosIfCan	acosh	acoshIfCan	acot
acotIfCan	acoth	acothIfCan	acsc	acscIfCan
acsch	acschIfCan	asec	asecIfCan	asech
asechIfCan	asin	asinIfCan	asinh	asinhIfCan
atan	atanIfCan	atanh	atanhIfCan	cos
cosIfCan	cosh	coshIfCan	cot	cotIfCan
coth	cothIfCan	csc	cscIfCan	csch
cschIfCan	exp	expIfCan	log	logIfCan
nthRootIfCan	sec	secIfCan	sech	sechIfCan
sin	sinIfCan	sinh	sinhIfCan	tan
tanIfCan	tanh	tanhIfCan	***	

— domain EFULS ElementaryFunctionsUnivariateLaurentSeries —

```

)abbrev domain EFULS ElementaryFunctionsUnivariateLaurentSeries
++ Author: Clifton J. Williamson
++ Date Created: 6 February 1990
++ Date Last Updated: 25 February 1990
++ Description:
++ This domain provides elementary functions on any Laurent series
++ domain over a field which was constructed from a Taylor series
  
```

```

++ domain. These functions are implemented by calling the
++ corresponding functions on the Taylor series domain. We also
++ provide 'partial functions' which compute transcendental
++ functions of Laurent series when possible and return "failed"
++ when this is not possible.

```

```

ElementaryFunctionsUnivariateLaurentSeries(Coef,UTS,ULS):_
Exports == Implementation where
  Coef   : Algebra Fraction Integer
  UTS    : UnivariateTaylorSeriesCategory Coef
  ULS    : UnivariateLaurentSeriesConstructorCategory(Coef,UTS)
  I      ==> Integer
  NNI    ==> NonNegativeInteger
  RN     ==> Fraction Integer
  S      ==> String
  STTF   ==> StreamTranscendentalFunctions(Coef)

Exports ==> PartialTranscendentalFunctions(ULS) with

  if Coef has Field then
    "***: (ULS,RN) -> ULS
      ++ s ** r raises a Laurent series s to a rational power r

--% Exponentials and Logarithms

exp: ULS -> ULS
  ++ exp(z) returns the exponential of Laurent series z.
log: ULS -> ULS
  ++ log(z) returns the logarithm of Laurent series z.

--% TrigonometricFunctionCategory

sin: ULS -> ULS
  ++ sin(z) returns the sine of Laurent series z.
cos: ULS -> ULS
  ++ cos(z) returns the cosine of Laurent series z.
tan: ULS -> ULS
  ++ tan(z) returns the tangent of Laurent series z.
cot: ULS -> ULS
  ++ cot(z) returns the cotangent of Laurent series z.
sec: ULS -> ULS
  ++ sec(z) returns the secant of Laurent series z.
csc: ULS -> ULS
  ++ csc(z) returns the cosecant of Laurent series z.

--% ArcTrigonometricFunctionCategory

asin: ULS -> ULS
  ++ asin(z) returns the arc-sine of Laurent series z.
acos: ULS -> ULS

```

```

    ++ acos(z) returns the arc-cosine of Laurent series z.
atan: ULS -> ULS
    ++ atan(z) returns the arc-tangent of Laurent series z.
acot: ULS -> ULS
    ++ acot(z) returns the arc-cotangent of Laurent series z.
asec: ULS -> ULS
    ++ asec(z) returns the arc-secant of Laurent series z.
acsc: ULS -> ULS
    ++ acsc(z) returns the arc-cosecant of Laurent series z.

--% HyperbolicFunctionCategory

sinh: ULS -> ULS
    ++ sinh(z) returns the hyperbolic sine of Laurent series z.
cosh: ULS -> ULS
    ++ cosh(z) returns the hyperbolic cosine of Laurent series z.
tanh: ULS -> ULS
    ++ tanh(z) returns the hyperbolic tangent of Laurent series z.
coth: ULS -> ULS
    ++ coth(z) returns the hyperbolic cotangent of Laurent series z.
sech: ULS -> ULS
    ++ sech(z) returns the hyperbolic secant of Laurent series z.
csch: ULS -> ULS
    ++ csch(z) returns the hyperbolic cosecant of Laurent series z.

--% ArcHyperbolicFunctionCategory

asinh: ULS -> ULS
    ++ asinh(z) returns the inverse hyperbolic sine of Laurent series z.
acosh: ULS -> ULS
    ++ acosh(z) returns the inverse hyperbolic cosine of Laurent series z.
atanh: ULS -> ULS
    ++ atanh(z) returns the inverse hyperbolic tangent of Laurent series z.
acoth: ULS -> ULS
    ++ acoth(z) returns the inverse hyperbolic cotangent of Laurent series z.
asech: ULS -> ULS
    ++ asech(z) returns the inverse hyperbolic secant of Laurent series z.
acsch: ULS -> ULS
    ++ acsch(z) returns the inverse hyperbolic cosecant of Laurent series z.

Implementation ==> add

--% roots

RATPOWERS : Boolean := Coef has "***":(Coef,RN) -> Coef
TRANSFCN  : Boolean := Coef has TranscendentalFunctionCategory
RATS      : Boolean := Coef has retractIfCan: Coef -> Union(RN,"failed")

nthRootUTS:(UTS,I) -> Union(UTS,"failed")
nthRootUTS(uts,n) ==

```

```

-- assumed: n > 1, uts has non-zero constant term
--   one? coefficient(uts,0) => uts ** inv(n::RN)
   coefficient(uts,0) = 1 => uts ** inv(n::RN)
   RATPOWERS => uts ** inv(n::RN)
   "failed"

nthRootIfCan(uls,nn) ==
  (n := nn :: I) < 1 => error "nthRootIfCan: n must be positive"
  n = 1 => uls
  deg := degree uls
  if zero? (coef := coefficient(uls,deg)) then
    uls := removeZeroes(1000,uls); deg := degree uls
    zero? (coef := coefficient(uls,deg)) =>
      error "root of series with many leading zero coefficients"
  (k := deg exquo n) case "failed" => "failed"
  uts := taylor(uls * monomial(1,-deg))
  (root := nthRootUTS(uts,n)) case "failed" => "failed"
  monomial(1,k :: I) * (root :: UTS :: ULS)

if Coef has Field then
  (uls:ULS) ** (r:RN) ==
    num := numer r; den := denom r
--   one? den => uls ** num
  den = 1 => uls ** num
  deg := degree uls
  if zero? (coef := coefficient(uls,deg)) then
    uls := removeZeroes(1000,uls); deg := degree uls
    zero? (coef := coefficient(uls,deg)) =>
      error "power of series with many leading zero coefficients"
  (k := deg exquo den) case "failed" =>
    error "**: rational power does not exist"
  uts := taylor(uls * monomial(1,-deg)) ** r
  monomial(1,(k :: I) * num) * (uts :: ULS)

--% transcendental functions

applyIfCan: (UTS -> UTS,ULS) -> Union(ULS,"failed")
applyIfCan(fcn,uls) ==
  uts := taylorIfCan uls
  uts case "failed" => "failed"
  fcn(uts :: UTS) :: ULS

expIfCan   uls == applyIfCan(exp,uls)
sinIfCan   uls == applyIfCan(sin,uls)
cosIfCan   uls == applyIfCan(cos,uls)
asinIfCan  uls == applyIfCan(asin,uls)
acosIfCan  uls == applyIfCan(acos,uls)
asecIfCan  uls == applyIfCan(asec,uls)
acscIfCan  uls == applyIfCan(acsc,uls)
sinhIfCan  uls == applyIfCan(sinh,uls)

```

```

coshIfCan uls == applyIfCan(cosh,uls)
asinhIfCan uls == applyIfCan(asinh,uls)
acoshIfCan uls == applyIfCan(acosh,uls)
atanhIfCan uls == applyIfCan(atanh,uls)
acothIfCan uls == applyIfCan(acoth,uls)
asechIfCan uls == applyIfCan(asech,uls)
acschIfCan uls == applyIfCan(acsch,uls)

logIfCan uls ==
  uts := taylorIfCan uls
  uts case "failed" => "failed"
  zero? coefficient(ts := uts :: UTS,0) => "failed"
  log(ts) :: ULS

tanIfCan uls ==
  -- don't call 'tan' on a UTS (tan(uls) may have a singularity)
  uts := taylorIfCan uls
  uts case "failed" => "failed"
  sc := sincos(coefficients(uts :: UTS))$STTF
  (cosInv := recip(series(sc.cos) :: ULS)) case "failed" => "failed"
  (series(sc.sin) :: ULS) * (cosInv :: ULS)

cotIfCan uls ==
  -- don't call 'cot' on a UTS (cot(uls) may have a singularity)
  uts := taylorIfCan uls
  uts case "failed" => "failed"
  sc := sincos(coefficients(uts :: UTS))$STTF
  (sinInv := recip(series(sc.sin) :: ULS)) case "failed" => "failed"
  (series(sc.cos) :: ULS) * (sinInv :: ULS)

secIfCan uls ==
  cos := cosIfCan uls
  cos case "failed" => "failed"
  (cosInv := recip(cos :: ULS)) case "failed" => "failed"
  cosInv :: ULS

cscIfCan uls ==
  sin := sinIfCan uls
  sin case "failed" => "failed"
  (sinInv := recip(sin :: ULS)) case "failed" => "failed"
  sinInv :: ULS

atanIfCan uls ==
  coef := coefficient(uls,0)
  (ord := order(uls,0)) = 0 and coef * coef = -1 => "failed"
  cc : Coef :=
    ord < 0 =>
      TRANSFCN =>
        RATS =>
          lc := coefficient(uls,ord)

```

```

        (rat := retractIfCan(lc)@Union(RN,"failed")) case "failed" =>
            (1/2) * pi()
        (rat :: RN) > 0 => (1/2) * pi()
        (-1/2) * pi()
        (1/2) * pi()
        return "failed"
    coef = 0 => 0
    TRANSFCN => atan coef
    return "failed"
(z := recip(1 + uls*uls)) case "failed" => "failed"
(cc :: ULS) + integrate(differentiate(uls) * (z :: ULS))

acotIfCan uls ==
    coef := coefficient(uls,0)
    (ord := order(uls,0)) = 0 and coef * coef = -1 => "failed"
    cc : Coef :=
        ord < 0 =>
            RATS =>
                lc := coefficient(uls,ord)
                (rat := retractIfCan(lc)@Union(RN,"failed")) case "failed" => 0
                (rat :: RN) > 0 => 0
                TRANSFCN => pi()
                return "failed"
            0
        TRANSFCN => acot coef
        return "failed"
    (z := recip(1 + uls*uls)) case "failed" => "failed"
    (cc :: ULS) - integrate(differentiate(uls) * (z :: ULS))

tanhIfCan uls ==
    -- don't call 'tanh' on a UTS (tanh(uls) may have a singularity)
    uts := taylorIfCan uls
    uts case "failed" => "failed"
    sc := sinhcosh(coefficients(uts :: UTS))$STTF
    (coshInv := recip(series(sc.cosh) :: ULS)) case "failed" =>
        "failed"
    (series(sc.sinh) :: ULS) * (coshInv :: ULS)

cothIfCan uls ==
    -- don't call 'coth' on a UTS (coth(uls) may have a singularity)
    uts := taylorIfCan uls
    uts case "failed" => "failed"
    sc := sinhcosh(coefficients(uts :: UTS))$STTF
    (sinhInv := recip(series(sc.sinh) :: ULS)) case "failed" =>
        "failed"
    (series(sc.cosh) :: ULS) * (sinhInv :: ULS)

sechIfCan uls ==
    cosh := coshIfCan uls
    cosh case "failed" => "failed"

```



```

(coshInv := recip(cosh :: ULS)) case "failed" => "failed"
coshInv :: ULS

cschIfCan uls ==
sinh := sinhIfCan uls
sinh case "failed" => "failed"
(sinhInv := recip(sinh :: ULS)) case "failed" => "failed"
sinhInv :: ULS

applyOnError:(ULS -> Union(ULS,"failed"),S,ULS) -> ULS
applyOnError(fcn,name,uls) ==
ans := fcn uls
ans case "failed" =>
error concat(name," of function with singularity")
ans :: ULS

exp uls == applyOnError(expIfCan,"exp",uls)
log uls == applyOnError(logIfCan,"log",uls)
sin uls == applyOnError(sinIfCan,"sin",uls)
cos uls == applyOnError(cosIfCan,"cos",uls)
tan uls == applyOnError(tanIfCan,"tan",uls)
cot uls == applyOnError(cotIfCan,"cot",uls)
sec uls == applyOnError(secIfCan,"sec",uls)
csc uls == applyOnError(cscIfCan,"csc",uls)
asin uls == applyOnError(asinIfCan,"asin",uls)
acos uls == applyOnError(acosIfCan,"acos",uls)
asec uls == applyOnError(asecIfCan,"asec",uls)
acsc uls == applyOnError(acscIfCan,"acsc",uls)
sinh uls == applyOnError(sinhIfCan,"sinh",uls)
cosh uls == applyOnError(coshIfCan,"cosh",uls)
tanh uls == applyOnError(tanhIfCan,"tanh",uls)
coth uls == applyOnError(cothIfCan,"coth",uls)
sech uls == applyOnError(sechIfCan,"sech",uls)
csch uls == applyOnError(cschIfCan,"csch",uls)
asinh uls == applyOnError(asinhIfCan,"asinh",uls)
acosh uls == applyOnError(acoshIfCan,"acosh",uls)
atanh uls == applyOnError(atanhIfCan,"atanh",uls)
acoth uls == applyOnError(acothIfCan,"acoth",uls)
asech uls == applyOnError(asechIfCan,"asech",uls)
acsch uls == applyOnError(acschIfCan,"acsch",uls)

atan uls ==
-- code is duplicated so that correct error messages will be returned
coef := coefficient(uls,0)
(ord := order(uls,0)) = 0 and coef * coef = -1 =>
error "atan: series expansion has logarithmic term"
cc : Coef :=
ord < 0 =>
TRANSFCN =>
RATS =>

```

```

      lc := coefficient(uls,ord)
      (rat := retractIfCan(lc)@Union(RN,"failed")) case "failed" =>
        (1/2) * pi()
      (rat :: RN) > 0 => (1/2) * pi()
      (-1/2) * pi()
      (1/2) * pi()
      error "atan: series expansion involves transcendental constants"
      coef = 0 => 0
      TRANSFCN => atan coef
      error "atan: series expansion involves transcendental constants"
      (z := recip(1 + uls*uls)) case "failed" =>
        error "atan: leading coefficient not invertible"
      (cc :: ULS) + integrate(differentiate(uls) * (z :: ULS))

acot uls ==
-- code is duplicated so that correct error messages will be returned
coef := coefficient(uls,0)
(ord := order(uls,0)) = 0 and coef * coef = -1 =>
  error "acot: series expansion has logarithmic term"
cc : Coef :=
  ord < 0 =>
    RATS =>
      lc := coefficient(uls,ord)
      (rat := retractIfCan(lc)@Union(RN,"failed")) case "failed" => 0
      (rat :: RN) > 0 => 0
      TRANSFCN => pi()
      error "acot: series expansion involves transcendental constants"
    0
  TRANSFCN => acot coef
  error "acot: series expansion involves transcendental constants"
  (z := recip(1 + uls*uls)) case "failed" =>
    error "acot: leading coefficient not invertible"
  (cc :: ULS) - integrate(differentiate(uls) * (z :: ULS))

```

— EFULS.dotabb —

```

"EFULS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EFULS"]
"ULSCCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ULSCCAT"]
"EFULS" -> "ULSCCAT"

```

domain EFUPXS ElementaryFunctionsUnivariatePuisseuxSeries

— ElementaryFunctionsUnivariatePuisseuxSeries.input —

```
)set break resume
)sys rm -f ElementaryFunctionsUnivariatePuisseuxSeries.output
)spool ElementaryFunctionsUnivariatePuisseuxSeries.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ElementaryFunctionsUnivariatePuisseuxSeries
--E 1

)spool
)lisp (bye)
```

— ElementaryFunctionsUnivariatePuisseuxSeries.help —

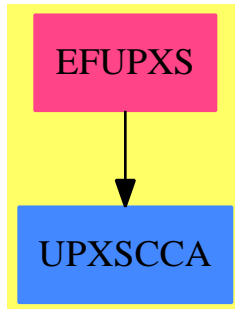
```
=====
ElementaryFunctionsUnivariatePuisseuxSeries examples
=====
```

This domain provides elementary functions on any Laurent series domain over a field which was constructed from a Taylor series domain. These functions are implemented by calling the corresponding functions on the Taylor series domain. We also provide 'partial functions' which compute transcendental functions of Laurent series when possible and return "failed" when this is not possible.

See Also:

o)show ElementaryFunctionsUnivariatePuisseuxSeries

ElementaryFunctionsUnivariatePuisseuxSeries (EFUPXS)

**Exports:**

acos	acosIfCan	acosh	acoshIfCan	acot
acotIfCan	acoth	acothIfCan	acsc	acscIfCan
acsch	acschIfCan	asec	asecIfCan	asech
asechIfCan	asin	asinIfCan	asinh	asinhIfCan
atan	atanIfCan	atanh	atanhIfCan	cos
cosIfCan	cosh	coshIfCan	cot	cotIfCan
coth	cothIfCan	csc	cscIfCan	csch
cschIfCan	exp	expIfCan	log	logIfCan
nthRootIfCan	sec	secIfCan	sech	sechIfCan
sin	sinIfCan	sinh	sinhIfCan	tan
tanIfCan	tanh	tanhIfCan	***	

— domain EFUPXS ElementaryFunctionsUnivariatePuisseuxSeries —

```

)abbrev domain EFUPXS ElementaryFunctionsUnivariatePuisseuxSeries
++ Author: Clifton J. Williamson
++ Date Created: 20 February 1990
++ Date Last Updated: 20 February 1990
++ Description:
++ This package provides elementary functions on any Laurent series
++ domain over a field which was constructed from a Taylor series
++ domain. These functions are implemented by calling the
++ corresponding functions on the Taylor series domain. We also
++ provide 'partial functions' which compute transcendental
++ functions of Laurent series when possible and return "failed"
++ when this is not possible.

```

```

ElementaryFunctionsUnivariatePuisseuxSeries(Coef,ULS,UPXS,EFULS):_
Exports == Implementation where
  Coef  : Algebra Fraction Integer
  ULS   : UnivariateLaurentSeriesCategory Coef
  UPXS  : UnivariatePuisseuxSeriesConstructorCategory(Coef,ULS)

```

```

EFULS : PartialTranscendentalFunctions(ULS)
I      ==> Integer
NNI    ==> NonNegativeInteger
RN      ==> Fraction Integer

Exports ==> PartialTranscendentalFunctions(UPXS) with

if Coef has Field then
  "***": (UPXS,RN) -> UPXS
  ++ z ** r raises a Puiseux series z to a rational power r

--% Exponentials and Logarithms

exp: UPXS -> UPXS
++ exp(z) returns the exponential of a Puiseux series z.
log: UPXS -> UPXS
++ log(z) returns the logarithm of a Puiseux series z.

--% TrigonometricFunctionCategory

sin: UPXS -> UPXS
++ sin(z) returns the sine of a Puiseux series z.
cos: UPXS -> UPXS
++ cos(z) returns the cosine of a Puiseux series z.
tan: UPXS -> UPXS
++ tan(z) returns the tangent of a Puiseux series z.
cot: UPXS -> UPXS
++ cot(z) returns the cotangent of a Puiseux series z.
sec: UPXS -> UPXS
++ sec(z) returns the secant of a Puiseux series z.
csc: UPXS -> UPXS
++ csc(z) returns the cosecant of a Puiseux series z.

--% ArcTrigonometricFunctionCategory

asin: UPXS -> UPXS
++ asin(z) returns the arc-sine of a Puiseux series z.
acos: UPXS -> UPXS
++ acos(z) returns the arc-cosine of a Puiseux series z.
atan: UPXS -> UPXS
++ atan(z) returns the arc-tangent of a Puiseux series z.
acot: UPXS -> UPXS
++ acot(z) returns the arc-cotangent of a Puiseux series z.
asec: UPXS -> UPXS
++ asec(z) returns the arc-secant of a Puiseux series z.
acsc: UPXS -> UPXS
++ acsc(z) returns the arc-cosecant of a Puiseux series z.

--% HyperbolicFunctionCategory

```

```

sinh: UPXS -> UPXS
  ++ sinh(z) returns the hyperbolic sine of a Puiseux series z.
cosh: UPXS -> UPXS
  ++ cosh(z) returns the hyperbolic cosine of a Puiseux series z.
tanh: UPXS -> UPXS
  ++ tanh(z) returns the hyperbolic tangent of a Puiseux series z.
coth: UPXS -> UPXS
  ++ coth(z) returns the hyperbolic cotangent of a Puiseux series z.
sech: UPXS -> UPXS
  ++ sech(z) returns the hyperbolic secant of a Puiseux series z.
csch: UPXS -> UPXS
  ++ csch(z) returns the hyperbolic cosecant of a Puiseux series z.

--% ArcHyperbolicFunctionCategory

asinh: UPXS -> UPXS
  ++ asinh(z) returns the inverse hyperbolic sine of a Puiseux series z.
acosh: UPXS -> UPXS
  ++ acosh(z) returns the inverse hyperbolic cosine of a Puiseux series z.
atanh: UPXS -> UPXS
  ++ atanh(z) returns the inverse hyperbolic tangent of a Puiseux series z.
acoth: UPXS -> UPXS
  ++ acoth(z) returns the inverse hyperbolic cotangent
  ++ of a Puiseux series z.
asech: UPXS -> UPXS
  ++ asech(z) returns the inverse hyperbolic secant of a Puiseux series z.
acsch: UPXS -> UPXS
  ++ acsch(z) returns the inverse hyperbolic cosecant
  ++ of a Puiseux series z.

Implementation ==> add

TRANSFCN : Boolean := Coef has TranscendentalFunctionCategory

--% roots

nthRootIfCan(upxs,n) ==
--   one? n => upxs
   n = 1 => upxs
   r := rationalPower upxs; uls := laurentRep upxs
   deg := degree uls
   if zero?(coef := coefficient(uls,deg)) then
     deg := order(uls,deg + 1000)
     zero?(coef := coefficient(uls,deg)) =>
       error "root of series with many leading zero coefficients"
   uls := uls * monomial(1,-deg)$ULS
   (ulsRoot := nthRootIfCan(uls,n)) case "failed" => "failed"
   puiseux(r,ulsRoot :: ULS) * monomial(1,deg * r * inv(n :: RN))

if Coef has Field then

```

```

(upxs:UPXS) ** (q:RN) ==
  num := numer q; den := denom q
--  one? den => upxs ** num
  den = 1 => upxs ** num
  r := rationalPower upxs; uls := laurentRep upxs
  deg := degree uls
  if zero?(coef := coefficient(uls,deg)) then
    deg := order(uls,deg + 1000)
    zero?(coef := coefficient(uls,deg)) =>
      error "power of series with many leading zero coefficients"
  ulsPow := (uls * monomial(1,-deg)$ULS) ** q
  puioux(r,ulsPow) * monomial(1,deg*q*r)

--% transcendental functions

applyIfCan: (ULS -> Union(ULS,"failed"),UPXS) -> Union(UPXS,"failed")
applyIfCan(fcn,upxs) ==
  uls := fcn laurentRep upxs
  uls case "failed" => "failed"
  puioux(rationalPower upxs,uls :: ULS)

expIfCan   upxs == applyIfCan(expIfCan,upxs)
logIfCan   upxs == applyIfCan(logIfCan,upxs)
sinIfCan   upxs == applyIfCan(sinIfCan,upxs)
cosIfCan   upxs == applyIfCan(cosIfCan,upxs)
tanIfCan   upxs == applyIfCan(tanIfCan,upxs)
cotIfCan   upxs == applyIfCan(cotIfCan,upxs)
secIfCan   upxs == applyIfCan(secIfCan,upxs)
cscIfCan   upxs == applyIfCan(cscIfCan,upxs)
atanIfCan  upxs == applyIfCan(atanIfCan,upxs)
acotIfCan  upxs == applyIfCan(acotIfCan,upxs)
sinhIfCan  upxs == applyIfCan(sinhIfCan,upxs)
coshIfCan  upxs == applyIfCan(coshIfCan,upxs)
tanhIfCan  upxs == applyIfCan(tanhIfCan,upxs)
cothIfCan  upxs == applyIfCan(cothIfCan,upxs)
sechIfCan  upxs == applyIfCan(sechIfCan,upxs)
cschIfCan  upxs == applyIfCan(cschIfCan,upxs)
asinhIfCan upxs == applyIfCan(asinhIfCan,upxs)
acoshIfCan upxs == applyIfCan(acoshIfCan,upxs)
atanhIfCan upxs == applyIfCan(atanhIfCan,upxs)
acothIfCan upxs == applyIfCan(acothIfCan,upxs)
asechIfCan upxs == applyIfCan(asechIfCan,upxs)
acschIfCan upxs == applyIfCan(acschIfCan,upxs)

asinIfCan upxs ==
  order(upxs,0) < 0 => "failed"
  (coef := coefficient(upxs,0)) = 0 =>
    integrate((1 - upxs*upxs)**(-1/2) * (differentiate upxs))
  TRANSFCN =>
    cc := asin(coef) :: UPXS

```

```

    cc + integrate((1 - upxs*upxs)**(-1/2) * (differentiate upxs))
    "failed"

acosIfCan upxs ==
  order(upxs,0) < 0 => "failed"
  TRANSFCN =>
    cc := acos(coefficent(upxs,0)) :: UPXS
    cc + integrate(-(1 - upxs*upxs)**(-1/2) * (differentiate upxs))
    "failed"

asecIfCan upxs ==
  order(upxs,0) < 0 => "failed"
  TRANSFCN =>
    cc := asec(coefficent(upxs,0)) :: UPXS
    f := (upxs*upxs - 1)**(-1/2) * (differentiate upxs)
    (rec := recip upxs) case "failed" => "failed"
    cc + integrate(f * (rec :: UPXS))
    "failed"

acscIfCan upxs ==
  order(upxs,0) < 0 => "failed"
  TRANSFCN =>
    cc := acsc(coefficent(upxs,0)) :: UPXS
    f := -(upxs*upxs - 1)**(-1/2) * (differentiate upxs)
    (rec := recip upxs) case "failed" => "failed"
    cc + integrate(f * (rec :: UPXS))
    "failed"

asinhIfCan upxs ==
  order(upxs,0) < 0 => "failed"
  TRANSFCN or (coefficent(upxs,0) = 0) =>
    log(upxs + (1 + upxs*upxs)**(1/2))
    "failed"

acoshIfCan upxs ==
  TRANSFCN =>
    order(upxs,0) < 0 => "failed"
    log(upxs + (upxs*upxs - 1)**(1/2))
    "failed"

asechIfCan upxs ==
  TRANSFCN =>
    order(upxs,0) < 0 => "failed"
    (rec := recip upxs) case "failed" => "failed"
    log((1 + (1 - upxs*upxs)*(1/2)) * (rec :: UPXS))
    "failed"

acschIfCan upxs ==
  TRANSFCN =>
    order(upxs,0) < 0 => "failed"

```



```

      (rec := recip upxs) case "failed" => "failed"
      log((1 + (1 + upxs*upxs)*(1/2)) * (rec :: UPXS))
      "failed"

applyOnError:(UPXS -> Union(UPXS,"failed"),String,UPXS) -> UPXS
applyOnError(fcn,name,upxs) ==
  ans := fcn upxs
  ans case "failed" =>
    error concat(name," of function with singularity")
  ans :: UPXS

exp upxs    == applyOnError(expIfCan,"exp",upxs)
log upxs    == applyOnError(logIfCan,"log",upxs)
sin upxs    == applyOnError(sinIfCan,"sin",upxs)
cos upxs    == applyOnError(cosIfCan,"cos",upxs)
tan upxs    == applyOnError(tanIfCan,"tan",upxs)
cot upxs    == applyOnError(cotIfCan,"cot",upxs)
sec upxs    == applyOnError(secIfCan,"sec",upxs)
csc upxs    == applyOnError(cscIfCan,"csc",upxs)
asin upxs   == applyOnError(asinIfCan,"asin",upxs)
acos upxs   == applyOnError(acosIfCan,"acos",upxs)
atan upxs   == applyOnError(atanIfCan,"atan",upxs)
acot upxs   == applyOnError(acotIfCan,"acot",upxs)
asec upxs   == applyOnError(asecIfCan,"asec",upxs)
acsc upxs   == applyOnError(acscIfCan,"acsc",upxs)
sinh upxs   == applyOnError(sinhIfCan,"sinh",upxs)
cosh upxs   == applyOnError(coshIfCan,"cosh",upxs)
tanh upxs   == applyOnError(tanhIfCan,"tanh",upxs)
coth upxs   == applyOnError(cothIfCan,"coth",upxs)
sech upxs   == applyOnError(sechIfCan,"sech",upxs)
csch upxs   == applyOnError(cschIfCan,"csch",upxs)
asinh upxs  == applyOnError(asinhIfCan,"asinh",upxs)
acosh upxs  == applyOnError(acoshIfCan,"acosh",upxs)
atanh upxs  == applyOnError(atanhIfCan,"atanh",upxs)
acoth upxs  == applyOnError(acothIfCan,"acoth",upxs)
asech upxs  == applyOnError(asechIfCan,"asech",upxs)
acsch upxs  == applyOnError(acschIfCan,"acsch",upxs)

```

— EFUPXS.dotabb —

```

"EFUPXS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EFUPXS"]
"UPXSCCA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UPXSCCA"]
"EFUPXS" -> "UPXSCCA"

```

— Equation.input —

[illegible]

--E 11

--S 12 of 12

eqpol::Boolean

--R

--R

--R (12) false

--R

Type: Boolean

--E 12

)spool

)lisp (bye)

— Equation.help —

Equation examples

The Equation domain provides equations as mathematical objects. These are used, for example, as the input to various solve operations.

Equations are created using the equals symbol, =.

eq1 := 3*x + 4*y = 5
4y + 3x= 5

Type: Equation Polynomial Integer

eq2 := 2*x + 2*y = 3
2y + 2x= 3

Type: Equation Polynomial Integer

The left- and right-hand sides of an equation are accessible using the operations lhs and rhs.

lhs eq1
4y + 3x

Type: Polynomial Integer

rhs eq1
5

Type: Polynomial Integer

Arithmetic operations are supported and operate on both sides of the equation.

eq1 + eq2
6y + 5x= 8

Type: Equation Polynomial Integer

```
eq1 * eq2
      2      2
      8y  + 14x y + 6x = 15
```

Type: Equation Polynomial Integer

```
2*eq2 - eq1
x= 1
```

Type: Equation Polynomial Integer

Equations may be created for any type so the arithmetic operations will be defined only when they make sense. For example, exponentiation is not defined for equations involving non-square matrices.

```
eq1**2
      2      2
      16y  + 24x y + 9x = 25
```

Type: Equation Polynomial Integer

Note that an equals symbol is also used to test for equality of values in certain contexts. For example, $x+1$ and y are unequal as polynomials.

```
if x+1 = y then "equal" else "unequal"
"unequal"
```

Type: String

```
eqpol := x+1 = y
x + 1= y
```

Type: Equation Polynomial Integer

If an equation is used where a Boolean value is required, then it is evaluated using the equality test from the operand type.

```
if eqpol then "equal" else "unequal"
"unequal"
```

Type: String

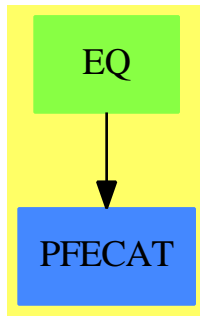
If one wants a Boolean value rather than an equation, all one has to do is ask!

```
eqpol::Boolean
false
```

Type: Boolean

See Also:

o)show Equation

Equation (EQ)**Exports:**

0	1	characteristic	coerce	commutator
conjugate	D	differentiate	dimension	equation
eval	factorAndSplit	hash	inv	latex
leftOne	leftZero	lhs	map	one?
recip	rhs	rightOne	rightZero	sample
subst	subtractIfCan	swap	zero?	?~=?
-?	?=?	?*?	?**?	?+?
?-?	?/?	?=?	?^?	

— domain EQ Equation —

```

)abbrev domain EQ Equation
--FOR THE BENEFIT OF LIBAXO GENERATION
++ Author: Stephen M. Watt, enhancements by Johannes Grabmeier
++ Date Created: April 1985
++ Date Last Updated: June 3, 1991; September 2, 1992
++ Description:
++ Equations as mathematical objects. All properties of the basis domain,
++ e.g. being an abelian group are carried over the equation domain, by
++ performing the structural operations on the left and on the
++ right hand side.
-- The interpreter translates "=" to "equation". Otherwise, it will
-- find a modemap for "=" in the domain of the arguments.

```

```

Equation(S: Type): public == private where
  Ex ==> OutputForm
  public ==> Type with
    "=": (S, S) -> $
      ++ a=b creates an equation.
    equation: (S, S) -> $
      ++ equation(a,b) creates an equation.

```

```

swap: $ -> $
  ++ swap(eq) interchanges left and right hand side of equation eq.
lhs: $ -> S
  ++ lhs(eq) returns the left hand side of equation eqn.
rhs: $ -> S
  ++ rhs(eq) returns the right hand side of equation eqn.
map: (S -> S, $) -> $
  ++ map(f,eqn) constructs a new equation by applying f to both
  ++ sides of eqn.
if S has InnerEvalable(Symbol,S) then
  InnerEvalable(Symbol,S)
if S has SetCategory then
  SetCategory
  CoercibleTo Boolean
  if S has Evalable(S) then
    eval: ($, $) -> $
      ++ eval(eq, x=f) replaces x by f in equation eqn.
    eval: ($, List $) -> $
      ++ eval(eq, [x1=v1, ... xn=vn]) replaces xi by vi in equation eqn.
if S has AbelianSemiGroup then
  AbelianSemiGroup
  "+": (S, $) -> $
    ++ x+eqn produces a new equation by adding x to both sides of
    ++ equation eqn.
  "+": ($, S) -> $
    ++ eqn+x produces a new equation by adding x to both sides of
    ++ equation eqn.
if S has AbelianGroup then
  AbelianGroup
  leftZero : $ -> $
    ++ leftZero(eq) subtracts the left hand side.
  rightZero : $ -> $
    ++ rightZero(eq) subtracts the right hand side.
  "-": (S, $) -> $
    ++ x-eqn produces a new equation by subtracting both sides of
    ++ equation eqn from x.
  "-": ($, S) -> $
    ++ eqn-x produces a new equation by subtracting x from both sides of
    ++ equation eqn.
if S has SemiGroup then
  SemiGroup
  "*": (S, $) -> $
    ++ x*eqn produces a new equation by multiplying both sides of
    ++ equation eqn by x.
  "*": ($, S) -> $
    ++ eqn*x produces a new equation by multiplying both sides of
    ++ equation eqn by x.
if S has Monoid then
  Monoid
  leftOne : $ -> Union($,"failed")

```

```

    ++ leftOne(eq) divides by the left hand side, if possible.
    rightOne : $ -> Union($,"failed")
    ++ rightOne(eq) divides by the right hand side, if possible.
  if S has Group then
    Group
    leftOne : $ -> Union($,"failed")
    ++ leftOne(eq) divides by the left hand side.
    rightOne : $ -> Union($,"failed")
    ++ rightOne(eq) divides by the right hand side.
  if S has Ring then
    Ring
    BiModule(S,S)
  if S has CommutativeRing then
    Module(S)
    --Algebra(S)
  if S has IntegralDomain then
    factorAndSplit : $ -> List $
    ++ factorAndSplit(eq) make the right hand side 0 and
    ++ factors the new left hand side. Each factor is equated
    ++ to 0 and put into the resulting list without repetitions.
  if S has PartialDifferentialRing(Symbol) then
    PartialDifferentialRing(Symbol)
  if S has Field then
    VectorSpace(S)
    "/": ($, $) -> $
    ++ e1/e2 produces a new equation by dividing the left and right
    ++ hand sides of equations e1 and e2.
    inv: $ -> $
    ++ inv(x) returns the multiplicative inverse of x.
  if S has ExpressionSpace then
    subst: ($, $) -> $
    ++ subst(eq1,eq2) substitutes eq2 into both sides of eq1
    ++ the lhs of eq2 should be a kernel

private ==> add
Rep := Record(lhs: S, rhs: S)
eq1,eq2: $
s : S
if S has IntegralDomain then
  factorAndSplit eq ==
    (S has factor : S -> Factored S) =>
      eq0 := rightZero eq
      [equation(rcf.factor,0) for rcf in factors factor lhs eq0]
  [eq]
l:S = r:S      == [l, r]
equation(l, r) == [l, r]  -- hack! See comment above.
lhs eqn        == eqn.lhs
rhs eqn        == eqn.rhs
swap eqn       == [rhs eqn, lhs eqn]
map(fn, eqn)   == equation(fn(eqn.lhs), fn(eqn.rhs))

```



```

if S has InnerEvalable(Symbol,S) then
  s:Symbol
  ls:List Symbol
  x:S
  lx:List S
  eval(eqn,s,x) == eval(eqn.lhs,s,x) = eval(eqn.rhs,s,x)
  eval(eqn,ls,lx) == eval(eqn.lhs,ls,lx) = eval(eqn.rhs,ls,lx)
if S has Evalable(S) then
  eval(eqn1:$, eqn2:$):$ ==
    eval(eqn1.lhs, eqn2 pretend Equation S) =
      eval(eqn1.rhs, eqn2 pretend Equation S)
  eval(eqn1:$, leqn2:List $):$ ==
    eval(eqn1.lhs, leqn2 pretend List Equation S) =
      eval(eqn1.rhs, leqn2 pretend List Equation S)
if S has SetCategory then
  eq1 = eq2 == (eq1.lhs = eq2.lhs)@Boolean and
               (eq1.rhs = eq2.rhs)@Boolean
  coerce(eqn:$):Ex == eqn.lhs::Ex = eqn.rhs::Ex
  coerce(eqn:$):Boolean == eqn.lhs = eqn.rhs
if S has AbelianSemiGroup then
  eq1 + eq2 == eq1.lhs + eq2.lhs = eq1.rhs + eq2.rhs
  s + eq2 == [s,s] + eq2
  eq1 + s == eq1 + [s,s]
if S has AbelianGroup then
  - eq == (- lhs eq) = (-rhs eq)
  s - eq2 == [s,s] - eq2
  eq1 - s == eq1 - [s,s]
  leftZero eq == 0 = rhs eq - lhs eq
  rightZero eq == lhs eq - rhs eq = 0
  0 == equation(0$S,0$S)
  eq1 - eq2 == eq1.lhs - eq2.lhs = eq1.rhs - eq2.rhs
if S has SemiGroup then
  eq1:$ * eq2:$ == eq1.lhs * eq2.lhs = eq1.rhs * eq2.rhs
  1:S * eqn:$ == 1 * eqn.lhs = 1 * eqn.rhs
  1:S * eqn:$ == 1 * eqn.lhs = 1 * eqn.rhs
  eqn:$ * 1:S == eqn.lhs * 1 = eqn.rhs * 1
  -- We have to be a bit careful here: raising to a +ve integer is OK
  -- (since it's the equivalent of repeated multiplication)
  -- but other powers may cause contradictions
  -- Watch what else you add here! JHD 2/Aug 1990
if S has Monoid then
  1 == equation(1$S,1$S)
  recip eq ==
    (lh := recip lhs eq) case "failed" => "failed"
    (rh := recip rhs eq) case "failed" => "failed"
    [lh :: S, rh :: S]
  leftOne eq ==
    (re := recip lhs eq) case "failed" => "failed"
    1 = rhs eq * re

```

```

rightOne eq ==
  (re := recip rhs eq) case "failed" => "failed"
  lhs eq * re = 1
if S has Group then
  inv eq == [inv lhs eq, inv rhs eq]
  leftOne eq == 1 = rhs eq * inv rhs eq
  rightOne eq == lhs eq * inv rhs eq = 1
if S has Ring then
  characteristic() == characteristic()$S
  i:Integer * eq:$ == (i::S) * eq
if S has IntegralDomain then
  factorAndSplit eq ==
    (S has factor : S -> Factored S) =>
      eq0 := rightZero eq
      [equation(rcf.factor,0) for rcf in factors factor lhs eq0]
    (S has Polynomial Integer) =>
      eq0 := rightZero eq
      MF ==> MultivariateFactorize(Symbol, IndexedExponents Symbol, _
        Integer, Polynomial Integer)
      p : Polynomial Integer := (lhs eq0) pretend Polynomial Integer
      [equation((rcf.factor) pretend S,0) for rcf in factors factor(p)$MF]
    [eq]
if S has PartialDifferentialRing(Symbol) then
  differentiate(eq:$, sym:Symbol):$ ==
    [differentiate(lhs eq, sym), differentiate(rhs eq, sym)]
if S has Field then
  dimension() == 2 :: CardinalNumber
  eq1:$ / eq2:$ == eq1.lhs / eq2.lhs = eq1.rhs / eq2.rhs
  inv eq == [inv lhs eq, inv rhs eq]
if S has ExpressionSpace then
  subst(eq1,eq2) ==
    eq3 := eq2 pretend Equation S
    [subst(lhs eq1,eq3),subst(rhs eq1,eq3)]

```

— EQ.dotabb —

"EQ" [color="#88FF44",href="bookvol10.3.pdf#nameddest=EQ"]
 "PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
 "EQ" -> "PFECAT"

— EqTable.input —

```

)set break resume
)sys rm -f EqTable.output
)spool EqTable.output
)set message test on
)set message auto off
)clear all

--S 1 of 6
e: EqTable(List Integer, Integer) := table()
--R
--R
--R (1) table()
--R
--R Type: EqTable(List(Integer),Integer)
--E 1

--S 2 of 6
l1 := [1,2,3]
--R
--R
--R (2) [1,2,3]
--R
--R Type: List(PositiveInteger)
--E 2

--S 3 of 6
l2 := [1,2,3]
--R
--R
--R (3) [1,2,3]
--R
--R Type: List(PositiveInteger)
--E 3

--S 4 of 6
e.l1 := 111
--R
--R
--R (4) 111
--R
--R Type: PositiveInteger
--E 4

--S 5 of 6
e.l2 := 222
--R
--R
--R (5) 222
--R
--R Type: PositiveInteger

```

```
--E 5
```

```
--S 6 of 6
```

```
e.l1
```

```
--R
```

```
--R
```

```
--R (6) 111
```

```
--R
```

```
Type: PositiveInteger
```

```
--E 6
```

```
)spool
```

```
)lisp (bye)
```

— EqTable.help —

=====

EqTable examples

=====

The EqTable domain provides tables where the keys are compared using eq?. Keys are considered equal only if they are the same instance of a structure. This is useful if the keys are themselves updatable structures. Otherwise, all operations are the same as for type Table.

The operation table is here used to create a table where the keys are lists of integers.

```
e: EqTable(List Integer, Integer) := table()
table()
Type: EqTable(List Integer,Integer)
```

These two lists are equal according to =, but not according to eq?.

```
l1 := [1,2,3]
[1,2,3]
Type: List PositiveInteger
```

```
l2 := [1,2,3]
[1,2,3]
Type: List PositiveInteger
```

Because the two lists are not eq?, separate values can be stored under each.

```
e.l1 := 111
111
Type: PositiveInteger
```

```
e.l2 := 222
```

222

Type: PositiveInteger

e.l1

111

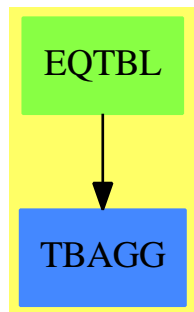
Type: PositiveInteger

See Also:

o)help Table

o)show EqTable

EqTable (EQTBL)



See

- ⇒ “HashTable” (HASHTBL) [9](#) on page [1120](#)
- ⇒ “InnerTable” (INTABL) [10](#) on page [1335](#)
- ⇒ “Table” (TABLE) [21](#) on page [2663](#)
- ⇒ “StringTable” (STRTBL) [20](#) on page [2610](#)
- ⇒ “GeneralSparseTable” (GSTBL) [8](#) on page [1078](#)
- ⇒ “SparseTable” (STBL) [20](#) on page [2450](#)

Exports:

any?	bag	coerce	construct	convert
copy	count	dictionary	elt	empty
empty?	entries	entry?	eq?	eval
every?	extract!	fill!	find	first
hash	index?	indices	insert!	inspect
key?	keys	latex	less?	map
map!	maxIndex	member?	members	minIndex
more?	parts	qelt	qsetelt!	reduce
remove	remove!	removeDuplicates	sample	search
select	select!	setelt	size?	swap!
table	table	#?	?=?	?~=?
?.?				

— domain EQTBL EqTable —

```
)abbrev domain EQTBL EqTable
++ Author: Stephen M. Watt
++ Date Created:
++ Date Last Updated: June 21, 1991
++ Description:
++ This domain provides tables where the keys are compared using
++ \spadfun{eq?}. Thus keys are considered equal only if they
++ are the same instance of a structure.
```

```
EqTable(Key: SetCategory, Entry: SetCategory) ==
  HashTable(Key, Entry, "EQ")
```

— EQTBL.dotabb —

```
"EQTBL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=EQTBL"]
"TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]
"EQTBL" -> "TBAGG"
```

domain EMR EuclideanModularRing

— EuclideanModularRing.input —

```
)set break resume
```

```

)sys rm -f EuclideanModularRing.output
)spool EuclideanModularRing.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show EuclideanModularRing
--R
--R EuclideanModularRing(S: CommutativeRing,R: UnivariatePolynomialCategory(S),Mod: AbelianMonoid)
--R Abbreviation for EuclideanModularRing is EMR
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for EMR
--R
--R----- Operations -----
--R ?? : (%,%) -> %                ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R +? : (%,%) -> %                ?-? : (%,%) -> %
--R -? : % -> %                    ?=? : (%,%) -> Boolean
--R 1 : () -> %                    0 : () -> %
--R ?? : (%,PositiveInteger) -> %   associates? : (%,%) -> Boolean
--R coerce : % -> R                 coerce : % -> %
--R coerce : Integer -> %            coerce : % -> OutputForm
--R ?? : (%,R) -> R                  gcd : List(%) -> %
--R gcd : (%,%) -> %                 hash : % -> SingleInteger
--R inv : % -> %                      latex : % -> String
--R lcm : List(%) -> %                lcm : (%,%) -> %
--R modulus : % -> Mod                one? : % -> Boolean
--R ?quo? : (%,%) -> %                recip : % -> Union(%, "failed")
--R reduce : (R,Mod) -> %             ?rem? : (%,%) -> %
--R sample : () -> %                  sizeLess? : (%,%) -> Boolean
--R unit? : % -> Boolean                unitCanonical : % -> %
--R zero? : % -> Boolean                ?~=? : (%,%) -> Boolean
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R ?? : (%,NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R divide : (%,%) -> Record(quotient: %,remainder: %)
--R euclideanSize : % -> NonNegativeInteger
--R exQuo : (%,%) -> Union(%, "failed")
--R expressIdealMember : (List(%),%) -> Union(List(%), "failed")
--R exquo : (%,%) -> Union(%, "failed")
--R extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %), "failed")
--R extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %)
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R multiEuclidean : (List(%),%) -> Union(List(%), "failed")
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %)
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R

```

```
--E 1
```

```
)spool
```

```
)lisp (bye)
```

— EuclideanModularRing.help —

=====

EuclideanModularRing examples

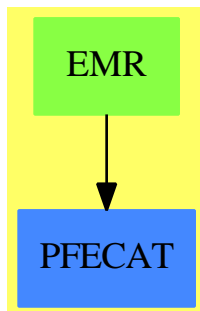
=====

These domains are used for the factorization and gcds of univariate polynomials over the integers in order to work modulo different primes.

See Also:

- o)show EuclideanModularRing
- o)show ModularRing
- o)show ModularField

EuclideanModularRing (EMR)



See

⇒ “ModularRing” (MODRING) [14](#) on page [1641](#)

⇒ “ModularField” (MODFIELD) [14](#) on page [1638](#)

Exports:

0	1	associates?	characteristic	coerce
divide	euclideanSize	exQuo	expressIdealMember	exquo
extendedEuclidean	gcd	gcdPolynomial	hash	inv
latex	lcm	modulus	multiEuclidean	one?
principalIdeal	recip	reduce	sample	sizeLess?
subtractIfCan	unit?	unitCanonical	unitNormal	zero?
?^=?	?*?	?**?	?^?	?+?
?-?	-?	?=?	?..?	?quo?
?rem?				

— domain EMR EuclideanModularRing —

```

)abbrev domain EMR EuclideanModularRing
++ Author: Mark Botch
++ Description:
++ These domains are used for the factorization and gcds
++ of univariate polynomials over the integers in order to work modulo
++ different primes.
++ See \spadtype{ModularRing}, \spadtype{ModularField}

EuclideanModularRing(S,R,Mod,reduction:(R,Mod) -> R,
    merge:(Mod,Mod) -> Union(Mod,"failed"),
    exactQuo : (R,R,Mod) -> Union(R,"failed")) : C == T

where
  S    : CommutativeRing
  R    : UnivariatePolynomialCategory S
  Mod  : AbelianMonoid

C == EuclideanDomain with
  modulus : % -> Mod
    ++ modulus(x) is not documented
  coerce : % -> R
    ++ coerce(x) is not documented
  reduce : (R,Mod) -> %
    ++ reduce(r,m) is not documented
  exQuo : (%,% ) -> Union(%, "failed")
    ++ exQuo(x,y) is not documented
  recip : % -> Union(%, "failed")
    ++ recip(x) is not documented
  inv : % -> %
    ++ inv(x) is not documented
  elt : (% , R) -> R
    ++ elt(x,r) or x.r is not documented

T == ModularRing(R,Mod,reduction,merge,exactQuo) add

--representation
Rep:= Record(val:R,module:Mod)

```

```

--declarations
x,y,z: %

divide(x,y) ==
  t:=merge(x.modulo,y.modulo)
  t case "failed" => error "incompatible moduli"
  xm:=t::Mod
  yv:=y.val
  invlcy:R
--   if one? leadingCoefficient yv then invlcy:=1
  if (leadingCoefficient yv = 1) then invlcy:=1
  else
    invlcy:=(inv reduce((leadingCoefficient yv)::R,xm)).val
    yv:=reduction(invlcy*yv,xm)
  r:=monicDivide(x.val,yv)
  [reduce(invlcy*r.quotient,xm),reduce(r.remainder,xm)]

if R has fme cg:(R,NonNegativeInteger,S,R)->R
then x rem y ==
  t:=merge(x.modulo,y.modulo)
  t case "failed" => error "incompatible moduli"
  xm:=t::Mod
  yv:=y.val
  invlcy:R
--   if not one? leadingCoefficient yv then
  if not (leadingCoefficient yv = 1) then
    invlcy:=(inv reduce((leadingCoefficient yv)::R,xm)).val
    yv:=reduction(invlcy*yv,xm)
  dy:=degree yv
  xv:=x.val
  while (d:=degree xv - dy)>=0 repeat
    xv:=reduction(fme cg(xv,d::NonNegativeInteger,
      leadingCoefficient xv,yv),xm)
    xv = 0 => return [xv,xm]$Rep
  [xv,xm]$Rep
else x rem y ==
  t:=merge(x.modulo,y.modulo)
  t case "failed" => error "incompatible moduli"
  xm:=t::Mod
  yv:=y.val
  invlcy:R
--   if not one? leadingCoefficient yv then
  if not (leadingCoefficient yv = 1) then
    invlcy:=(inv reduce((leadingCoefficient yv)::R,xm)).val
    yv:=reduction(invlcy*yv,xm)
  r:=monicDivide(x.val,yv)
  reduce(r.remainder,xm)

euclideanSize x == degree x.val

```

```

unitCanonical x ==
  zero? x => x
  degree(x.val) = 0 => 1
--
  one? leadingCoefficient(x.val) => x
  (leadingCoefficient(x.val) = 1) => x
  invlcx:=%:=inv reduce((leadingCoefficient(x.val))::R,x.modulo)
  invlcx * x

unitNormal x ==
--
  zero?(x) or one?(leadingCoefficient(x.val)) => [1, x, 1]
  zero?(x) or ((leadingCoefficient(x.val)) = 1) => [1, x, 1]
  lcx := reduce((leadingCoefficient(x.val))::R,x.modulo)
  invlcx:=inv lcx
  degree(x.val) = 0 => [lcx, 1, invlcx]
  [lcx, invlcx * x, invlcx]

elt(x : %,s : R) : R == reduction(elt(x.val,s),x.modulo)

```

— EMR.dotabb —

```

"EMR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=EMR"]
"PFECA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECA"]
"EMR" -> "PFECA"

```

domain EXIT Exit

— Exit.input —

```

)set break resume
)sys rm -f Exit.output
)spool Exit.output
)set message test on
)set message auto off
)clear all
--S 1
n := 0
--R
--R
--R
--R (1) 0
--R

```

Type: NonNegativeInteger

```

--E 1

--S 2
gasp(): Exit ==
  free n
  n := n + 1
  error "Oh no!"
--R
--R   Function declaration gasp : () -> Exit has been added to workspace.
--R                                                    Type: Void
--E 2

--S 3
half(k) ==
  if odd? k then gasp()
  else k quo 2
--R
--R                                                    Type: Void
--E 3

--S 4
half 4
--R
--R   Compiling function gasp with type () -> Exit
--R   Compiling function half with type PositiveInteger -> Integer
--R
--R   (4)  2
--R
--R                                                    Type: PositiveInteger
--E 4

--S 5
half 3
--R
--R
--RDaly Bug
--R   Error signalled from user code in function gasp:
--R   Oh no!
--E 5

--S 6
n
--R
--R
--R   (5)  1
--R
--R                                                    Type: NonNegativeInteger
--E 6
)spool
)lisp (bye)

```

— Exit.help —

```
=====
Exit examples
=====
```

A function that does not return directly to its caller has Exit as its return type. The operation error is an example of one which does not return to its caller. Instead, it causes a return to top-level.

```
n := 0
```

The function gasp is given return type Exit since it is guaranteed never to return a value to its caller.

```
gasp(): Exit ==
  free n
  n := n + 1
  error "Oh no!"
```

The return type of half is determined by resolving the types of the two branches of the if.

```
half(k) ==
  if odd? k then gasp()
  else k quo 2
```

Because gasp has the return type Exit, the type of if in half is resolved to be Integer.

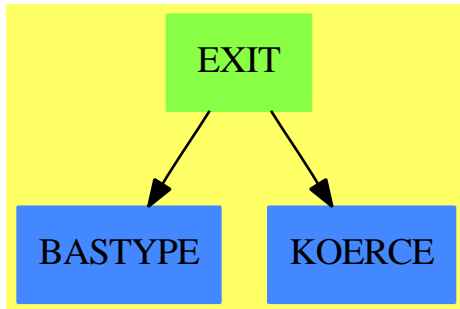
```
half 4
```

```
half 3
```

```
n
```

See Also:
o)show Exit

—————▶

Exit (EXIT)**Exports:**

coerce hash latex ?=? ?~=?

— domain EXIT Exit —

```

)abbrev domain EXIT Exit
++ Author: Stephen M. Watt
++ Date Created: 1986
++ Date Last Updated: May 30, 1991
++ Description:
++ A function which does not return directly to its caller should
++ have Exit as its return type.
++
++ Note that It is convenient to have a formal \spad{coerce} into each type
++ from type Exit. This allows, for example, errors to be raised in
++ one half of a type-balanced \spad{if}.
  
```

```

Exit: SetCategory == add
      coerce(n:%) == error "Cannot use an Exit value."
      n1 = n2      == error "Cannot use an Exit value."
  
```

— EXIT.dotabb —

```

"EXIT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=EXIT"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"EXIT" -> "BASTYPE"
"EXIT" -> "KOERCE"
  
```

domain EXPEXPAN ExponentialExpansion

— ExponentialExpansion.input —

```

)set break resume
)sys rm -f ExponentialExpansion.output
)spool ExponentialExpansion.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExponentialExpansion
--R
--R ExponentialExpansion(R: Join(OrderedSet,RetractableTo(Integer),LinearlyExplicitRingOver(Integer)))
--R Abbreviation for ExponentialExpansion is EXPEXPAN
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for EXPEXPAN
--R
--R----- Operations -----
--R ?? : (Fraction(Integer),%) -> %      ?? : (%,Fraction(Integer)) -> %
--R ?? : (%,%) -> %                      ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %        ??? : (%,Integer) -> %
--R ??? : (%,PositiveInteger) -> %      ?+? : (%,%) -> %
--R ?-? : (%,%) -> %                    -? : % -> %
--R ?/? : (%,%) -> %                    ?=? : (%,%) -> Boolean
--R 1 : () -> %                         0 : () -> %
--R ?? : (%,Integer) -> %               ?? : (%,PositiveInteger) -> %
--R associates? : (%,%) -> Boolean      coerce : Fraction(Integer) -> %
--R coerce : % -> %                     coerce : Integer -> %
--R coerce : % -> OutputForm            denominator : % -> %
--R factor : % -> Factored(%)           gcd : List(%) -> %
--R gcd : (%,%) -> %                   hash : % -> SingleInteger
--R inv : % -> %                        latex : % -> String
--R lcm : List(%) -> %                  lcm : (%,%) -> %
--R numerator : % -> %                  one? : % -> Boolean
--R prime? : % -> Boolean                ?quo? : (%,%) -> %
--R recip : % -> Union(%, "failed")      ?rem? : (%,%) -> %
--R sample : () -> %                     sizeLess? : (%,%) -> Boolean
--R squareFree : % -> Factored(%)        squareFreePart : % -> %
--R unit? : % -> Boolean                 unitCanonical : % -> %
--R zero? : % -> Boolean                 ?~=? : (%,%) -> Boolean
--R ?? : (%,UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen)) -> %
--R ?? : (UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen),%) -> %
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R ?/? : (UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen),UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen)) -> %
--R ?<? : (%,%) -> Boolean if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) ha

```

```

--R ?<=? : (%,% ) -> Boolean if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has ORDSET
--R ?>? : (%,% ) -> Boolean if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has ORDSET
--R ?>=? : (%,% ) -> Boolean if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has ORDSET
--R D : (%,(UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) -> UnivariatePuisseuxSeriesWithE
--R D : (%,(UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) -> UnivariatePuisseuxSeriesWithE
--R D : (%,(List(Symbol),List(NonNegativeInteger))) -> % if UnivariatePuisseuxSeriesWithExponentialSingularity
--R D : (%,(Symbol,NonNegativeInteger)) -> % if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen)
--R D : (%,(List(Symbol))) -> % if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has PDRING
--R D : (%,(Symbol)) -> % if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has PDRING(SYMBOL)
--R D : (%,(NonNegativeInteger)) -> % if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has P
--R D : % -> % if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has DIFRING
--R ?? : (%,(NonNegativeInteger)) -> %
--R abs : % -> % if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has OINTDOM
--R ceiling : % -> UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) if UnivariatePuisseuxSeries
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%,"failed") if $ has CHARNZ and UnivariatePuisseuxSeriesWithExponentialSingularity
--R coerce : UnivariatePuisseuxSeries(FE,var,cen) -> %
--R coerce : Symbol -> % if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has RETRACT(SYMBOL)
--R coerce : UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) -> %
--R conditionP : Matrix(%) -> Union(Vector(%),"failed") if $ has CHARNZ and UnivariatePuisseuxSeriesWithExpon
--R convert : % -> DoubleFloat if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has REAL
--R convert : % -> Float if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has REAL
--R convert : % -> InputForm if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has KONVERT
--R convert : % -> Pattern(Float) if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has KO
--R convert : % -> Pattern(Integer) if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has KO
--R denom : % -> UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen)
--R differentiate : (%,(UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) -> UnivariatePuisseux
--R differentiate : (%,(UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) -> UnivariatePuisseux
--R differentiate : (%,(List(Symbol),List(NonNegativeInteger))) -> % if UnivariatePuisseuxSeriesWithExponential
--R differentiate : (%,(Symbol,NonNegativeInteger)) -> % if UnivariatePuisseuxSeriesWithExponentialSingularity
--R differentiate : (%,(List(Symbol))) -> % if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen)
--R differentiate : (%,(Symbol)) -> % if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has P
--R differentiate : (%,(NonNegativeInteger)) -> % if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,v
--R differentiate : % -> % if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has DIFRING
--R divide : (%,% ) -> Record(quotient: %,remainder: %)
--R ?.? : (%,(UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen))) -> % if UnivariatePuisseuxSeries
--R euclideanSize : % -> NonNegativeInteger
--R eval : (%,(Symbol,UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen))) -> % if UnivariatePuis
--R eval : (%,(List(Symbol),List(UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen)))) -> % if Un
--R eval : (%,(List(Equation(UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen)))) -> % if Univa
--R eval : (%,(Equation(UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen))) -> % if Univariate
--R eval : (%,(UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen),UnivariatePuisseuxSeriesWithExp
--R eval : (%,(List(UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen)),List(UnivariatePuisseuxS
--R expressIdealMember : (List(%),%) -> Union(List(%),"failed")
--R exquo : (%,% ) -> Union(%,"failed")
--R extendedEuclidean : (%,%,% ) -> Union(Record(coef1: %,coef2: %),"failed")
--R extendedEuclidean : (%,% ) -> Record(coef1: %,coef2: %,generator: %)
--R factorPolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if Univariate
--R factorSquareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if
--R floor : % -> UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) if UnivariatePuisseuxSeries

```



```

--R fractionPart : % -> % if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUniva
--R init : () -> % if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has STEP
--R limitPlus : % -> Union(OrderedCompletion(FE),"failed")
--R map : ((UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) -> UnivariatePuisseux
--R max : (%,% ) -> % if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has ORDS
--R min : (%,% ) -> % if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has ORDS
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed")
--R negative? : % -> Boolean if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen)
--R nextItem : % -> Union(%,"failed") if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,
--R numer : % -> UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen)
--R patternMatch : (% ,Pattern(Float),PatternMatchResult(Float,%)) -> PatternMatchResult(Float,% )
--R patternMatch : (% ,Pattern(Integer),PatternMatchResult(Integer,%)) -> PatternMatchResult(Inte
--R positive? : % -> Boolean if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen)
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %)
--R random : () -> % if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has INS
--R reducedSystem : Matrix(%) -> Matrix(UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,v
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(UnivariatePuisseuxSeriesWithExpon
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) i
--R reducedSystem : Matrix(%) -> Matrix(Integer) if UnivariatePuisseuxSeriesWithExponentialSingul
--R retract : % -> UnivariatePuisseuxSeries(FE,var,cen)
--R retract : % -> Integer if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) ha
--R retract : % -> Fraction(Integer) if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,v
--R retract : % -> Symbol if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has
--R retract : % -> UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen)
--R retractIfCan : % -> Union(UnivariatePuisseuxSeries(FE,var,cen),"failed")
--R retractIfCan : % -> Union(Integer,"failed") if UnivariatePuisseuxSeriesWithExponentialSingula
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if UnivariatePuisseuxSeriesWithExponent
--R retractIfCan : % -> Union(Symbol,"failed") if UnivariatePuisseuxSeriesWithExponentialSingular
--R retractIfCan : % -> Union(UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen),"f
--R sign : % -> Integer if UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) has 0
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(%)),SparseUnivariatePolynom
--R squareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(
--R subtractIfCan : (%,% ) -> Union(%,"failed")
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R wholePart : % -> UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen) if Univaria
--R
--E 1

```

```

)spool
)lisp (bye)

```

— ExponentialExpansion.help —

```

=====
ExponentialExpansion examples
=====

```

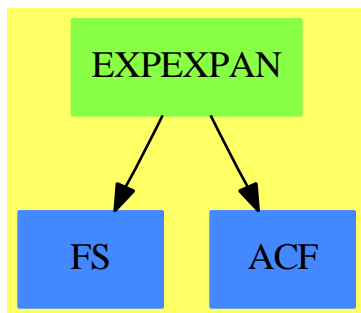
UnivariatePuisseuxSeriesWithExponentialSingularity is a domain used to represent essential singularities of functions. Objects in this domain are quotients of sums, where each term in the sum is a univariate Puiseux series times the exponential of a univariate Puiseux series.

See Also:

o)show ExponentialExpansion

—————→

ExponentialExpansion (EXPEXPAN)



See

⇒ “ExponentialOfUnivariatePuisseuxSeries” (EXPUPXS) [6](#) on page [739](#)

⇒ “UnivariatePuisseuxSeriesWithExponentialSingularity” (UPXSSING) [22](#) on page [2852](#)

Exports:

0	1	associates?
abs	ceiling	characteristic
charthRoot	coerce	conditionP
convert	D	denom
denominator	differentiate	divide
euclideanSize	expressIdealMember	eval
exquo	extendedEuclidean	factor
factorSquareFreePolynomial	factorPolynomial	floor
fractionPart	gcd	gcdPolynomial
hash	init	inv
latex	lcm	limitPlus
map	max	min
multiEuclidean	negative?	nextItem
numer	numerator	one?
patternMatch	positive?	prime?
principalIdeal	random	recip
reducedSystem	retract	retractIfCan
sample	sign	sizeLess?
solveLinearPolynomialEquation	squareFree	squareFreePart
squareFreePolynomial	subtractIfCan	unit?
unitCanonical	unitNormal	wholePart
zero?	?*?	?**?
?+?	?-?	-?
?/?	?=?	?^?
?~=?	?<?	?<=?
?>?	?>=?	?..?
?quo?	?rem?	

— domain EXPEXPAN ExponentialExpansion —

```

)abbrev domain EXPEXPAN ExponentialExpansion
++ Author: Clifton J. Williamson
++ Date Created: 13 August 1992
++ Date Last Updated: 27 August 1992
++ Description:
++ UnivariatePuisseuxSeriesWithExponentialSingularity is a domain used to
++ represent essential singularities of functions. Objects in this domain
++ are quotients of sums, where each term in the sum is a univariate Puiseux
++ series times the exponential of a univariate Puiseux series.

ExponentialExpansion(R,FE,var,cen): Exports == Implementation where
  R   : Join(OrderedSet,RetractableTo Integer,_,
            LinearlyExplicitRingOver Integer,GcdDomain)
  FE  : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,_,
            FunctionSpace R)
  var : Symbol
  cen : FE

```

```

RN      ==> Fraction Integer
UPXS    ==> UnivariatePuisseuxSeries(FE,var,cen)
EXPUPXS ==> ExponentialOfUnivariatePuisseuxSeries(FE,var,cen)
UPXSSING ==> UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen)
OFE     ==> OrderedCompletion FE
Result  ==> Union(OFE,"failed")
PxRec   ==> Record(k: Fraction Integer,c:FE)
Term    ==> Record(%coef:UPXS,%expon:EXPUPXS,%expTerms:List PxRec)
TypedTerm ==> Record(%term:Term,%type:String)
SIGNEF  ==> ElementaryFunctionSign(R,FE)

Exports ==> Join(QuotientFieldCategory UPXSSING,RetractableTo UPXS) with
  limitPlus : % -> Union(OFE,"failed")
    ++ limitPlus(f(var)) returns \spad{limit(var -> a+,f(var))}.
  coerce: UPXS -> %
    ++ coerce(f) converts a \spadtype{UnivariatePuisseuxSeries} to
    ++ an \spadtype{ExponentialExpansion}.

Implementation ==> Fraction(UPXSSING) add
  coeff : Term -> UPXS
  exponent : Term -> EXPUPXS
  upxssingIfCan : % -> Union(UPXSSING,"failed")
  seriesQuotientLimit: (UPXS,UPXS) -> Union(OFE,"failed")
  seriesQuotientInfinity: (UPXS,UPXS) -> Union(OFE,"failed")

Rep := Fraction UPXSSING

ZEROCOUNT : RN := 1000/1

coeff term == term.%coef
exponent term == term.%expon

--!! why is this necessary?
--!! code can run forever in retractIfCan if original assignment
--!! for 'ff' is used
upxssingIfCan f ==
--   one? denom f => numer f
   (denom f = 1) => numer f
   "failed"

retractIfCan(f:%):Union(UPXS,"failed") ==
  --ff := (retractIfCan$Rep)(f)@Union(UPXSSING,"failed")
  --ff case "failed" => "failed"
  (ff := upxssingIfCan f) case "failed" => "failed"
  (fff := retractIfCan(ff::UPXSSING)@Union(UPXS,"failed")) case "failed" =>
    "failed"
  fff :: UPXS

f:UPXSSING / g:UPXSSING ==
  (rec := recip g) case "failed" => f /$Rep g

```

```

f * (rec :: UPXSSING) :: %

f:% / g:% ==
  (rec := recip numer g) case "failed" => f / $Rep g
  (rec :: UPXSSING) * (denom g) * f

coerce(f:UPXS) == f :: UPXSSING :: %

seriesQuotientLimit(num,den) ==
  -- limit of the quotient of two series
  series := num / den
  (ord := order(series,1)) > 0 => 0
  coef := coefficient(series,ord)
  member?(var,variables coef) => "failed"
  ord = 0 => coef :: OFE
  (sig := sign(coef)$SIGNEF) case "failed" => return "failed"
  (sig :: Integer) = 1 => plusInfinity()
  minusInfinity()

seriesQuotientInfinity(num,den) ==
  -- infinite limit: plus or minus?
  -- look at leading coefficients of series to tell
  (numOrd := order(num,ZEROCOUNT)) = ZEROCOUNT => "failed"
  (denOrd := order(den,ZEROCOUNT)) = ZEROCOUNT => "failed"
  cc := coefficient(num,numOrd)/coefficient(den,denOrd)
  member?(var,variables cc) => "failed"
  (sig := sign(cc)$SIGNEF) case "failed" => return "failed"
  (sig :: Integer) = 1 => plusInfinity()
  minusInfinity()

limitPlus f ==
  zero? f => 0
  (den := denom f) = 1 => limitPlus numer f
  (numTerm := dominantTerm(num := numer f)) case "failed" => "failed"
  numType := (numTerm := numerTerm :: TypedTerm).%type
  (denomTerm := dominantTerm den) case "failed" => "failed"
  denType := (denTerm := denomTerm :: TypedTerm).%type
  numExpon := exponent numTerm.%term; denExpon := exponent denTerm.%term
  numCoef := coeff numTerm.%term; denCoef := coeff denTerm.%term
  -- numerator tends to zero exponentially
  (numType = "zero") =>
    -- denominator tends to zero exponentially
    (denType = "zero") =>
      (exponDiff := numExpon - denExpon) = 0 =>
        seriesQuotientLimit(numCoef,denCoef)
      expCoef := coefficient(exponDiff,order exponDiff)
      (sig := sign(expCoef)$SIGNEF) case "failed" => return "failed"
      (sig :: Integer) = -1 => 0
      seriesQuotientInfinity(numCoef,denCoef)
    0 -- otherwise limit is zero

```

```

-- numerator is a Puiseux series
(numType = "series") =>
  -- denominator tends to zero exponentially
  (denType = "zero") =>
    seriesQuotientInfinity(numCoef,denCoef)
  -- denominator is a series
  (denType = "series") => seriesQuotientLimit(numCoef,denCoef)
  0
-- remaining case: numerator tends to infinity exponentially
-- denominator tends to infinity exponentially
(denType = "infinity") =>
  (exponDiff := numExpon - denExpon) = 0 =>
    seriesQuotientLimit(numCoef,denCoef)
  expCoef := coefficient(exponDiff,order exponDiff)
  (sig := sign(expCoef)$SIGNEF) case "failed" => return "failed"
  (sig :: Integer) = -1 => 0
  seriesQuotientInfinity(numCoef,denCoef)
-- denominator tends to zero exponentially or is a series
seriesQuotientInfinity(numCoef,denCoef)

```

— EXPEXPAN.dotabb —

```

"EXPEXPAN" [color="#88FF44",href="bookvol10.3.pdf#nameddest=EXPEXPAN"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"EXPEXPAN" -> "ACF"
"EXPEXPAN" -> "FS"

```

domain EXPR Expression

— Expression.input —

```

)set break resume
)sys rm -f Expression.output
)spool Expression.output
)set message test on
)set message auto off
)clear all
--S 1 of 23
sin(x) + 3*cos(x)**2

```

```

--R
--R
--R      2
--R      (1)  sin(x) + 3cos(x)
--R
--R                                          Type: Expression(Integer)
--E 1

--S 2 of 23
tan(x) - 3.45*x
--R
--R
--R      (2)  tan(x) - 3.45 x
--R
--R                                          Type: Expression(Float)
--E 2

--S 3 of 23
(tan sqrt 7 - sin sqrt 11)**2 / (4 - cos(x - y))
--R
--R
--R      +-+ 2      +---+      +-+      +---+ 2
--R      - tan(\|7 ) + 2sin(\|11 )tan(\|7 ) - sin(\|11 )
--R      (3)  -----
--R                      cos(y - x) - 4
--R
--R                                          Type: Expression(Integer)
--E 3

--S 4 of 23
log(exp x)@Expression(Integer)
--R
--R
--R      (4)  x
--R
--R                                          Type: Expression(Integer)
--E 4

--S 5 of 23
log(exp x)@Expression(Complex Integer)
--R
--R
--R      x
--R      (5)  log(%e )
--R
--R                                          Type: Expression(Complex(Integer))
--E 5

--S 6 of 23
sqrt 3 + sqrt(2 + sqrt(-5))
--R
--R
--R      +-----+
--R      | +---+      +-+
--R      (6)  \||- 5  + 2  + \|3

```

```

--R                                                    Type: AlgebraicNumber
--E 6

--S 7 of 23
% :: Expression Integer
--R
--R
--R      +-----+
--R      | +---+      +-+
--R  (7)  \|\|- 5  + 2  + \3
--R
--R                                                    Type: Expression(Integer)
--E 7

--S 8 of 23
height mainKernel sin(x + 4)
--R
--R
--R  (8)  2
--R
--R                                                    Type: PositiveInteger
--E 8

--S 9 of 23
e := (sin(x) - 4)**2 / ( 1 - 2*y*sqrt(- y) )
--R
--R
--R      2
--R      - sin(x)  + 8sin(x) - 16
--R  (9)  -----
--R      +---+
--R      2y\|- y  - 1
--R
--R                                                    Type: Expression(Integer)
--E 9

--S 10 of 23
numer e
--R
--R
--R      2
--R  (10) - sin(x)  + 8sin(x) - 16
--R      Type: SparseMultivariatePolynomial(Integer,Kernel(Expression(Integer)))
--E 10

--S 11 of 23
denom e
--R
--R
--R      +---+
--R  (11) 2y\|- y  - 1
--R      Type: SparseMultivariatePolynomial(Integer,Kernel(Expression(Integer)))
--E 11

```


--S 12 of 23

D(e, x)

--R

--R

--R

--R
$$(4y \cos(x)\sin(x) - 16y \cos(x))\sqrt{-y} - 2\cos(x)\sin(x) + 8\cos(x)$$

--R (12) -----

--R
$$4y\sqrt{-y} + 4y^3 - 1$$

Type: Expression(Integer)

--E 12

--S 13 of 23

D(e, [x, y], [1, 2])

--R

--R

--R (13)

--R
$$((-2304y^7 + 960y^4)\cos(x)\sin(x) + (9216y^7 - 3840y^4)\cos(x))\sqrt{-y}$$

--R +

--R
$$(-960y^9 + 2160y^6 - 180y^3 - 3)\cos(x)\sin(x)$$

--R +

--R
$$(3840y^9 - 8640y^6 + 720y^3 + 12)\cos(x)$$

--R /

--R
$$(256y^{12} - 1792y^9 + 1120y^6 - 112y^3 + 1)\sqrt{-y} - 1024y^{11} + 1792y^8 - 448y^5$$

--R +

--R
$$16y^2$$

Type: Expression(Integer)

--E 13

--S 14 of 23

complexNumeric(cos(2 - 3*i))

--R

--R

--R (14) - 4.1896256909 688072301 + 9.1092278937 55336598 %i

Type: Complex(Float)

--E 14

--S 15 of 23

numeric(tan 3.8)

--R

--R

--R (15) 0.7735560905 0312607286

--R

Type: Float

--E 15

```

--S 16 of 23
e2 := cos(x**2 - y + 3)
--R
--R
--R      2
--R (16) cos(y - x - 3)
--R
--R                                          Type: Expression(Integer)
--E 16

--S 17 of 23
e3 := asin(e2) - %pi/2
--R
--R
--R      2
--R (17) - y + x + 3
--R
--R                                          Type: Expression(Integer)
--E 17

--S 18 of 23
e3 :: Polynomial Integer
--R
--R
--R      2
--R (18) - y + x + 3
--R
--R                                          Type: Polynomial(Integer)
--E 18

--S 19 of 23
e3 :: DMP([x, y], Integer)
--R
--R
--R      2
--R (19) x - y + 3
--R
--R                                          Type: DistributedMultivariatePolynomial([x,y],Integer)
--E 19

--S 20 of 23
sin %pi
--R
--R
--R (20) 0
--R
--R                                          Type: Expression(Integer)
--E 20

--S 21 of 23
cos(%pi / 4)
--R
--R
--R      +-+

```

```

--R      \|2
--R (21) ----
--R      2
--R
--R                                          Type: Expression(Integer)
--E 21

--S 22 of 23
tan(x)**6 + 3*tan(x)**4 + 3*tan(x)**2 + 1
--R
--R
--R      6      4      2
--R (22) tan(x) + 3tan(x) + 3tan(x) + 1
--R
--R                                          Type: Expression(Integer)
--E 22

--S 23 of 23
simplify %
--R
--R
--R      1
--R (23) -----
--R      6
--R      cos(x)
--R
--R                                          Type: Expression(Integer)
--E 23
)spool
)lisp (bye)

```

— Expression.help —

=====

Expression examples

=====

Expression is a constructor that creates domains whose objects can have very general symbolic forms. Here are some examples:

This is an object of type Expression Integer.

```
sin(x) + 3*cos(x)**2
```

This is an object of type Expression Float.

```
tan(x) - 3.45*x
```

This object contains symbolic function applications, sums, products, square roots, and a quotient.

```
(tan sqrt 7 - sin sqrt 11)**2 / (4 - cos(x - y))
```

As you can see, Expression actually takes an argument domain. The coefficients of the terms within the expression belong to the argument domain. Integer and Float, along with Complex Integer and Complex Float are the most common coefficient domains.

The choice of whether to use a Complex coefficient domain or not is important since Axiom can perform some simplifications on real-valued objects

```
log(exp x)@Expression(Integer)
```

... which are not valid on complex ones.

```
log(exp x)@Expression(Complex Integer)
```

Many potential coefficient domains, such as AlgebraicNumber, are not usually used because Expression can subsume them.

```
sqrt 3 + sqrt(2 + sqrt(-5))
```

```
% :: Expression Integer
```

Note that we sometimes talk about "an object of type Expression." This is not really correct because we should say, for example, "an object of type Expression Integer" or "an object of type Expression Float." By a similar abuse of language, when we refer to an "expression" in this section we will mean an object of type Expression R for some domain R.

The Axiom documentation contains many examples of the use of Expression. For the rest of this section, we'll give you some pointers to those examples plus give you some idea of how to manipulate expressions.

It is important for you to know that Expression creates domains that have category Field. Thus you can invert any non-zero expression and you shouldn't expect an operation like factor to give you much information. You can imagine expressions as being represented as quotients of "multivariate" polynomials where the "variables" are kernels. A kernel can either be a symbol such as x or a symbolic function application like sin(x + 4). The second example is actually a nested kernel since the argument to sin contains the kernel x.

```
height mainKernel sin(x + 4)
```

Actually, the argument to sin is an expression, and so the structure of Expression is recursive. See Kernel which demonstrates how to

extract the kernels in an expression.

Use the HyperDoc Browse facility to see what operations are applicable to expression. At the time of this writing, there were 262 operations with 147 distinct name in Expression Integer. For example, numer and denom extract the numerator and denominator of an expression.

```
e := (sin(x) - 4)**2 / ( 1 - 2*y*sqrt(- y) )
```

```
numer e
```

```
denom e
```

Use D to compute partial derivatives.

```
D(e, x)
```

```
D(e, [x, y], [1, 2])
```

When an expression involves no ‘symbol kernels’ (for example, x), it may be possible to numerically evaluate the expression.

If you suspect the evaluation will create a complex number, use complexNumeric.

```
complexNumeric(cos(2 - 3*i))
```

If you know it will be real, use numeric.

```
numeric(tan 3.8)
```

The numeric operation will display an error message if the evaluation yields a value with a non-zero imaginary part. Both of these operations have an optional second argument n which specifies that the accuracy of the approximation be up to n decimal places.

When an expression involves no "symbolic application" kernels, it may be possible to convert it a polynomial or rational function in the variables that are present.

```
e2 := cos(x**2 - y + 3)
```

```
e3 := asin(e2) - %pi/2
```

```
e3 :: Polynomial Integer
```

This also works for the polynomial types where specific variables and their ordering are given.

```
e3 :: DMP([x, y], Integer)
```

Finally, a certain amount of simplification takes place as expressions are constructed.

```
sin %pi
```

```
cos(%pi / 4)
```

For simplifications that involve multiple terms of the expression, use `simplify`.

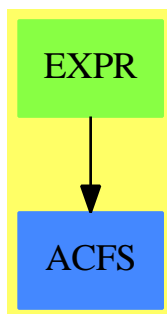
```
tan(x)**6 + 3*tan(x)**4 + 3*tan(x)**2 + 1
```

```
simplify %
```

See Also:

- o `)show Kernel`
- o `)show Expression`

Expression (EXPR)



See

⇒ “Pi” (HACKPI) [17](#) on page [1977](#)

Exports:

0	1	abs	acos
acosh	acot	acoth	acsc
acsch	airyAi	airyBi	applyQuote
asec	asech	asin	asinh
associates?	atan	atanh	belong?
besselI	besselJ	besselK	besselY
Beta	binomial	box	characteristic
charthRoot	Ci	coerce	commutator
conjugate	convert	cos	cosh
cot	coth	csc	csch
D	definingPolynomial	denom	denominator
differentiate	digamma	dilog	distribute
divide	Ei	elt	erf
euclideanSize	eval	even?	exp
expressIdealMember	exquo	extendedEuclidean	factor
factorPolynomial	factorial	factorials	freeOf?
Gamma	gcd	gcdPolynomial	ground
ground?	hash	height	integral
inv	is?	isExpt	isMult
isPlus	isPower	isTimes	kernel
kernels	latex	lcm	li
log	mainKernel	map	max
min	minPoly	multiEuclidean	nthRoot
number?	numer	numerator	odd?
one?	operator	operators	paren
patternMatch	permutation	pi	polygamma
prime?	principalIdeal	product	recip
reduce	reducedSystem	retract	retractIfCan
rootOf	rootsOf	sample	sec
sech	Si	simplifyPower	sin
sinh	sizeLess?	sqrt	squareFree
squareFreePart	squareFreePolynomial	subst	subtractIfCan
summation	tan	tanh	tower
unit?	unitCanonical	unitNormal	univariate
variables	zero?	zeroOf	zerosOf
?<?	?<=?	?=?	?>?
?>=?	?~=?	?*?	?**?
?+?	-?	?-?	?/?
?^?	?quo?	?rem?	

— domain **EXPR** Expression —)abbrev domain **EXPR** Expression

++ Author: Manuel Bronstein

++ Date Created: 19 July 1988

```

++ Date Last Updated: October 1993 (P.Gianni), February 1995 (MB)
++ Description:
++ Top-level mathematical expressions involving symbolic functions.

```

```

Expression(R:OrderedSet): Exports == Implementation where

```

```

  Q ==> Fraction Integer
  K ==> Kernel %
  MP ==> SparseMultivariatePolynomial(R, K)
  AF ==> AlgebraicFunction(R, %)
  EF ==> ElementaryFunction(R, %)
  CF ==> CombinatorialFunction(R, %)
  LF ==> LiouvillianFunction(R, %)
  AN ==> AlgebraicNumber
  KAN ==> Kernel AN
  FSF ==> FunctionalSpecialFunction(R, %)
  ESD ==> ExpressionSpace_&(&())
  FSD ==> FunctionSpace_&(&(), R)
  SYMBOL ==> "%symbol"
  ALGOP ==> "%alg"
  POWER ==> "%power"::Symbol
  SUP ==> SparseUnivariatePolynomial

```

```

Exports ==> FunctionSpace R with

```

```

  if R has IntegralDomain then
    AlgebraicallyClosedFunctionSpace R
    TranscendentalFunctionCategory
    CombinatorialOpsCategory
    LiouvillianFunctionCategory
    SpecialFunctionCategory
    reduce: % -> %
      ++ reduce(f) simplifies all the unreduced algebraic quantities
      ++ present in f by applying their defining relations.
    number?: % -> Boolean
      ++ number?(f) tests if f is rational
    simplifyPower: (%,Integer) -> %
      ++ simplifyPower?(f,n) is not documented
    if R has GcdDomain then
      factorPolynomial : SUP % -> Factored SUP %
        ++ factorPolynomial(p) is not documented
      squareFreePolynomial : SUP % -> Factored SUP %
        ++ squareFreePolynomial(p) is not documented
    if R has RetractableTo Integer then RetractableTo AN

```

```

Implementation ==> add

```

```

  import KernelFunctions2(R, %)

  retNotUnit      : % -> R
  retNotUnitIfCan: % -> Union(R, "failed")

  belong? op == true

```



```

retNotUnit x ==
  (u := constantIfCan(k := retract(x)@K)) case R => u::R
  error "Not retractable"

retNotUnitIfCan x ==
  (r := retractIfCan(x)@Union(K,"failed")) case "failed" => "failed"
  constantIfCan(r::K)

if R has IntegralDomain then
  reduc : (% , List Kernel %) -> %
  commonk : (% , %) -> List K
  commonk0 : (List K, List K) -> List K
  toptrat : % -> %
  algkernels: List K -> List K
  evl : (MP, K, SparseUnivariatePolynomial %) -> Fraction MP
  evl0 : (MP, K) -> SparseUnivariatePolynomial Fraction MP

Rep := Fraction MP
0 == 0$Rep
1 == 1$Rep
-- one? x == one?(x)$Rep
one? x == (x = 1)$Rep
zero? x == zero?(x)$Rep
- x:% == -$Rep x
n:Integer * x:% == n *$Rep x
coerce(n:Integer) == coerce(n)$Rep@Rep::%
x:% * y:% == reduc(x *$Rep y, commonk(x, y))
x:% + y:% == reduc(x +$Rep y, commonk(x, y))
(x:% - y:%):% == reduc(x -$Rep y, commonk(x, y))
x:% / y:% == reduc(x /$Rep y, commonk(x, y))

number?(x:%):Boolean ==
  if R has RetractableTo(Integer) then
    ground?(x) or ((retractIfCan(x)@Union(Q,"failed")) case Q)
  else
    ground?(x)

simplifyPower(x:%,n:Integer):% ==
  k : List K := kernels x
  is?(x,POWER) =>
    -- Look for a power of a number in case we can do a simplification
    args : List % := argument first k
    not(#args = 2) => error "Too many arguments to **"
    number?(args.1) =>
      reduc((args.1) **$Rep n, algkernels kernels (args.1))**(args.2)
      (first args)**(n*second(args))
      reduc(x **$Rep n, algkernels k)

x:% ** n:NonNegativeInteger ==

```

```

n = 0 => 1$%
n = 1 => x
simplifyPower(numerator x,n pretend Integer) /
  simplifyPower(denominator x,n pretend Integer)

x:% ** n:Integer ==
  n = 0 => 1$%
  n = 1 => x
  n = -1 => 1/x
  simplifyPower(numerator x,n) /
    simplifyPower(denominator x,n)

x:% ** n:PositiveInteger ==
  n = 1 => x
  simplifyPower(numerator x,n pretend Integer) /
    simplifyPower(denominator x,n pretend Integer)

x:% < y:%      == x <$Rep y
x:% = y:%      == x =$Rep y
numer x        == numer(x)$Rep
denom x        == denom(x)$Rep
coerce(p:MP):% == coerce(p)$Rep
reduce x       == reduc(x, algkernels kernels x)
commonk(x, y)  == commonk0(algkernels kernels x, algkernels kernels y)
algkernels l   == select_!(x +-> has?(operator x, ALGOP), l)
toprat f == ratDenom(f,algkernels kernels f)$AlgebraicManipulations(R, %)

x:MP / y:MP ==
  reduc(x /$Rep y,commonk0(algkernels variables x,algkernels variables y))

-- since we use the reduction from FRAC SMP which assumes that the
-- variables are independent, we must remove algebraic from the denominators
reducedSystem(m:Matrix %):Matrix(R) ==
  mm:Matrix(MP) := reducedSystem(map(toprat, m))$Rep
  reducedSystem(mm)$MP

-- since we use the reduction from FRAC SMP which assumes that the
-- variables are independent, we must remove algebraic from the denominators
reducedSystem(m:Matrix %, v:Vector %):
  Record(mat:Matrix R, vec:Vector R) ==
  r:Record(mat:Matrix MP, vec:Vector MP) :=
    reducedSystem(map(toprat, m), map(toprat, v))$Rep
  reducedSystem(r.mat, r.vec)$MP

-- The result MUST be left sorted deepest first    MB 3/90
commonk0(x, y) ==
  ans := empty()$List(K)
  for k in reverse_! x repeat if member?(k, y) then ans := concat(k, ans)
  ans

```

```

rootOf(x: SparseUnivariatePolynomial %, v: Symbol) == rootOf(x, v)$AF
pi() == pi()$EF
exp x == exp(x)$EF
log x == log(x)$EF
sin x == sin(x)$EF
cos x == cos(x)$EF
tan x == tan(x)$EF
cot x == cot(x)$EF
sec x == sec(x)$EF
csc x == csc(x)$EF
asin x == asin(x)$EF
acos x == acos(x)$EF
atan x == atan(x)$EF
acot x == acot(x)$EF
asec x == asec(x)$EF
acsc x == acsc(x)$EF
sinh x == sinh(x)$EF
cosh x == cosh(x)$EF
tanh x == tanh(x)$EF
coth x == coth(x)$EF
sech x == sech(x)$EF
csch x == csch(x)$EF
asinh x == asinh(x)$EF
acosh x == acosh(x)$EF
atanh x == atanh(x)$EF
acoth x == acoth(x)$EF
asech x == asech(x)$EF
acsch x == acsch(x)$EF

abs x == abs(x)$FSF
Gamma x == Gamma(x)$FSF
Gamma(a, x) == Gamma(a, x)$FSF
Beta(x, y) == Beta(x, y)$FSF
digamma x == digamma(x)$FSF
polygamma(k, x) == polygamma(k, x)$FSF
besselJ(v, x) == besselJ(v, x)$FSF
besselY(v, x) == besselY(v, x)$FSF
besselI(v, x) == besselI(v, x)$FSF
besselK(v, x) == besselK(v, x)$FSF
airyAi x == airyAi(x)$FSF
airyBi x == airyBi(x)$FSF

x:% ** y:% == x **$CF y
factorial x == factorial(x)$CF
binomial(n, m) == binomial(n, m)$CF
permutation(n, m) == permutation(n, m)$CF
factorials x == factorials(x)$CF
factorials(x, n) == factorials(x, n)$CF
summation(x:%, n: Symbol) == summation(x, n)$CF
summation(x:%, s: SegmentBinding %) == summation(x, s)$CF

```

```

product(x:%, n:Symbol)      == product(x, n)$CF
product(x:%, s:SegmentBinding %) == product(x, s)$CF

erf x                        == erf(x)$LF
Ei x                        == Ei(x)$LF
Si x                        == Si(x)$LF
Ci x                        == Ci(x)$LF
li x                        == li(x)$LF
dilog x                     == dilog(x)$LF
fresnelS x                  == fresnelS(x)$LF
fresnelC x                  == fresnelC(x)$LF
integral(x:%, n:Symbol)     == integral(x, n)$LF
integral(x:%, s:SegmentBinding %) == integral(x, s)$LF

operator op ==
  belong?(op)$AF => operator(op)$AF
  belong?(op)$EF => operator(op)$EF
  belong?(op)$CF => operator(op)$CF
  belong?(op)$LF => operator(op)$LF
  belong?(op)$FSF => operator(op)$FSF
  belong?(op)$FSD => operator(op)$FSD
  belong?(op)$ESD => operator(op)$ESD
  nullary? op and has?(op, SYMBOL) => operator(kernel(name op)$K)
  (n := arity op) case "failed" => operator name op
  operator(name op, n::NonNegativeInteger)

reduc(x, l) ==
  for k in l repeat
    p := minPoly k
    x := evl( numer x, k, p) /$Rep evl(denom x, k, p)
  x

evl0(p, k) ==
  numer univariate(p::Fraction(MP),
    k)$PolynomialCategoryQuotientFunctions(IndexedExponents K,
      K,R,MP,Fraction MP)

-- uses some operations from Rep instead of % in order not to
-- reduce recursively during those operations.
evl(p, k, m) ==
  degree(p, k) < degree m => p::Fraction(MP)
  (((evl0(p, k) pretend SparseUnivariatePolynomial($)) rem m)
    pretend SparseUnivariatePolynomial Fraction MP) (k::MP::Fraction(MP))

if R has GcdDomain then
  noalg?: SUP % -> Boolean

noalg? p ==
  while p ^= 0 repeat
    not empty? algkernels kernels leadingCoefficient p => return false

```

```

    p := reductum p
    true

gcdPolynomial(p:SUP %, q:SUP %) ==
  noalg? p and noalg? q => gcdPolynomial(p, q)$Rep
gcdPolynomial(p, q)$GcdDomain_&(&)

factorPolynomial(x:SUP %) : Factored SUP % ==
  uf:= factor(x pretend SUP(Rep))$SupFractionFactorizer(
    IndexedExponents K,K,R,MP)
  uf pretend Factored SUP %

squareFreePolynomial(x:SUP %) : Factored SUP % ==
  uf:= squareFree(x pretend SUP(Rep))$SupFractionFactorizer(
    IndexedExponents K,K,R,MP)
  uf pretend Factored SUP %

if R is AN then
  -- this is to force the coercion R -> EXPR R to be used
  -- instead of the coercion AN -> EXPR R which loops.
  -- simpler looking code will fail! MB 10/91
  coerce(x:AN):% == (monomial(x, 0$IndexedExponents(K))$MP)::%

if (R has RetractableTo Integer) then
  x:% ** r:Q == x **$AF r
  minPoly k == minPoly(k)$AF
  definingPolynomial x == definingPolynomial(x)$AF
  retract(x:%):Q == retract(x)$Rep
  retractIfCan(x:%):Union(Q, "failed") == retractIfCan(x)$Rep

if not(R is AN) then
  k2expr : KAN -> %
  smp2expr: SparseMultivariatePolynomial(Integer, KAN) -> %
  R2AN : R -> Union(AN, "failed")
  k2an : K -> Union(AN, "failed")
  smp2an : MP -> Union(AN, "failed")

  coerce(x:AN):% == smp2expr(num x) / smp2expr(denom x)
  k2expr k == map(x+>x::%, k)$ExpressionSpaceFunctions2(AN, %)

  smp2expr p ==
    map(k2expr,x+>x::%,p)_
    $PolynomialCategoryLifting(IndexedExponents KAN,
      KAN, Integer, SparseMultivariatePolynomial(Integer, KAN), %)

  retractIfCan(x:%):Union(AN, "failed") ==
    ((n:= smp2an num x) case AN) and ((d:= smp2an denom x) case AN)
    => (n::AN) / (d::AN)
    "failed"

```

```

R2AN r ==
  (u := retractIfCan(r::%)@Union(Q, "failed")) case Q => u::Q::AN
  "failed"

k2an k ==
  not(belong?(op := operator k)$AN) => "failed"
  arg:List(AN) := empty()
  for x in argument k repeat
    if (a := retractIfCan(x)@Union(AN, "failed")) case "failed" then
      return "failed"
    else arg := concat(a::AN, arg)
  (operator(op)$AN) reverse_!(arg)

smp2an p ==
  (x1 := mainVariable p) case "failed" => R2AN leadingCoefficient p
  up := univariate(p, k := x1::K)
  (t := k2an k) case "failed" => "failed"
  ans:AN := 0
  while not ground? up repeat
    (c:=smp2an leadingCoefficient up) case "failed" => return "failed"
    ans := ans + (c::AN) * (t::AN) ** (degree up)
    up := reductum up
  (c := smp2an leadingCoefficient up) case "failed" => "failed"
  ans + c::AN

if R has ConvertibleTo InputForm then
  convert(x::%):InputForm == convert(x)$Rep
  import MakeUnaryCompiledFunction(%, %, %)
  eval(f::%, op: BasicOperator, g::%, x:Symbol)::% ==
    eval(f,[op],[g],x)
  eval(f::%, ls:List BasicOperator, lg:List %, x:Symbol) ==
    -- handle subscripted symbols by renaming -> eval -> renaming back
    llsym:List List Symbol:=[variables g for g in lg]
    lsym:List Symbol:= removeDuplicates concat llsym
    lsd:List Symbol:=select (scripted?,lsym)
    empty? lsd=> eval(f,ls,[compiledFunction(g, x) for g in lg])
    ns:List Symbol:=[new()$Symbol for i in lsd]
    lforwardSubs:List Equation % := [(i::%)= (j::%) for i in lsd for j in ns]
    lbackwardSubs:List Equation % := [(j::%)= (i::%) for i in lsd for j in ns]
    nlgl:List % :=[subst(g,lforwardSubs) for g in lg]
    res::% :=eval(f, ls, [compiledFunction(g, x) for g in nlgl])
    subst(res,lbackwardSubs)

if R has PatternMatchable Integer then
  patternMatch(x::%, p:Pattern Integer,
    l:PatternMatchResult(Integer, %)) ==
    patternMatch(x, p, l)$PatternMatchFunctionSpace(Integer, R, %)

if R has PatternMatchable Float then
  patternMatch(x::%, p:Pattern Float,

```

```

l:PatternMatchResult(Float, %) ==
  patternMatch(x, p, l)$PatternMatchFunctionSpace(Float, R, %)

else -- R is not an integral domain
operator op ==
  belong?(op)$FSD => operator(op)$FSD
  belong?(op)$ESD => operator(op)$ESD
  nullary? op and has?(op, SYMBOL) => operator(kernel(name op)$K)
  (n := arity op) case "failed" => operator name op
  operator(name op, n::NonNegativeInteger)

if R has Ring then
  Rep := MP
  0 == 0$Rep
  1 == 1$Rep
  - x:% == -$Rep x
  n:Integer *x:% == n *$Rep x
  x:% * y:% == x *$Rep y
  x:% + y:% == x +$Rep y
  x:% = y:% == x =$Rep y
  x:% < y:% == x <$Rep y
  numer x == x@Rep
  coerce(p:MP):% == p

  reducedSystem(m:Matrix %):Matrix(R) ==
    reducedSystem(m)$Rep

  reducedSystem(m:Matrix %, v:Vector %):
  Record(mat:Matrix R, vec:Vector R) ==
    reducedSystem(m, v)$Rep

if R has ConvertibleTo InputForm then
  convert(x:%):InputForm == convert(x)$Rep

if R has PatternMatchable Integer then
  kintmatch: (K,Pattern Integer,PatternMatchResult(Integer,Rep))
    -> PatternMatchResult(Integer, Rep)

  kintmatch(k, p, l) ==
    patternMatch(k, p, l pretend PatternMatchResult(Integer, %)
    )$PatternMatchKernel(Integer, %)
    pretend PatternMatchResult(Integer, Rep)

  patternMatch(x:%, p:Pattern Integer,
  l:PatternMatchResult(Integer, %)) ==
    patternMatch(x@Rep, p,
    l pretend PatternMatchResult(Integer, Rep),
    kintmatch
    )$PatternMatchPolynomialCategory(Integer,
    IndexedExponents K, K, R, Rep)

```

```

                                pretend PatternMatchResult(Integer, %)

if R has PatternMatchable Float then
  kfltmach: (K, Pattern Float, PatternMatchResult(Float, Rep))
            -> PatternMatchResult(Float, Rep)

  kfltmach(k, p, l) ==
    patternMatch(k, p, l pretend PatternMatchResult(Float, %)
    )$PatternMatchKernel(Float, %)
    pretend PatternMatchResult(Float, Rep)

  patternMatch(x:%, p:Pattern Float,
  l:PatternMatchResult(Float, %)) ==
    patternMatch(x@Rep, p,
    l pretend PatternMatchResult(Float, Rep),
    kfltmach
    )$PatternMatchPolynomialCategory(Float,
    IndexedExponents K, K, R, Rep)
    pretend PatternMatchResult(Float, %)

else -- R is not even a ring
  if R has AbelianMonoid then
    import ListToMap(K, %)

    kereval      : (K, List K, List %) -> %
    subeval      : (K, List K, List %) -> %

    Rep := FreeAbelianGroup K

    0 == 0$Rep
    x:% + y:% == x +$Rep y
    x:% = y:% == x =$Rep y
    x:% < y:% == x <$Rep y
    coerce(k:K):% == coerce(k)$Rep
    kernels x == [f.gen for f in terms x]
    coerce(x:R):% == (zero? x => 0; constantKernel(x)::%)
    retract(x:%):R == (zero? x => 0; retNotUnit x)
    coerce(x:%):OutputForm == coerce(x)$Rep
    kereval(k, lk, lv) ==
      match(lk, lv, k, (x2:K):% +-> map(x1 +-> eval(x1, lk, lv), x2))

    subeval(k, lk, lv) ==
      match(lk, lv, k,
      (x:K):% +->
      kernel(operator x, [subst(a, lk, lv) for a in argument x]))

  isPlus x ==
    empty?(l := terms x) or empty? rest l => "failed"
    [t.exp *$Rep t.gen for t in l]$List(%)

```



```

isMult x ==
  empty?(l := terms x) or not empty? rest l => "failed"
  t := first l
  [t.exp, t.gen]

eval(x:%, lk:List K, lv:List %) ==
  _+/[t.exp * kereval(t.gen, lk, lv) for t in terms x]

subst(x:%, lk:List K, lv:List %) ==
  _+/[t.exp * subeval(t.gen, lk, lv) for t in terms x]

retractIfCan(x:%):Union(R, "failed") ==
  zero? x => 0
  retNotUnitIfCan x

if R has AbelianGroup then -(x:%) == -$Rep x

--      else      -- R is not an AbelianMonoid
--      if R has SemiGroup then
--      Rep := FreeGroup K
--      1      == 1$Rep
--      x:% * y:%      == x *$Rep y
--      x:% = y:%      == x =$Rep y
--      coerce(k:K):%  == k:$Rep
--      kernels x      == [f.gen for f in factors x]
--      coerce(x:R):%  == (one? x => 1; constantKernel x)
--      retract(x:%):R == (one? x => 1; retNotUnit x)
--      coerce(x:%):OutputForm == coerce(x)$Rep

--      retractIfCan(x:%):Union(R, "failed") ==
--      one? x => 1
--      retNotUnitIfCan x

--      if R has Group then inv(x:%):% == inv(x)$Rep

--      else      -- R is nothing
--      import ListToMap(K, %)

--      Rep := K

--      x:% < y:%      == x <$Rep y
--      x:% = y:%      == x =$Rep y
--      coerce(k:K):%  == k
--      kernels x      == [x pretend K]
--      coerce(x:R):%  == constantKernel x
--      retract(x:%):R == retNotUnit x
--      retractIfCan(x:%):Union(R, "failed") == retNotUnitIfCan x
--      coerce(x:%):OutputForm      == coerce(x)$Rep

eval(x:%, lk:List K, lv:List %) ==

```

```

      match(lk, lv, x pretend K,
        (x1:K):% +-> map(x2 +-> eval(x2, lk, lv), x1))

subst(x, lk, lv) ==
  match(lk, lv, x pretend K,
    (x1:K):% +->
      kernel(operator x1, [subst(a, lk, lv) for a in argument x1]))

if R has ConvertibleTo InputForm then
  convert(x:%):InputForm == convert(x)$Rep

--
if R has PatternMatchable Integer then
--
  convert(x:%):Pattern(Integer) == convert(x)$Rep
--
--
  patternMatch(x:%, p:Pattern Integer,
--
    l:PatternMatchResult(Integer, %)) ==
--
    patternMatch(x pretend K,p,l)$PatternMatchKernel(Integer, %)
--
--
if R has PatternMatchable Float then
--
  convert(x:%):Pattern(Float) == convert(x)$Rep
--
--
  patternMatch(x:%, p:Pattern Float,
--
    l:PatternMatchResult(Float, %)) ==
--
    patternMatch(x pretend K, p, l)$PatternMatchKernel(Float, %)
--

-----

— EXPR.dotabb —

"EXPR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=EXPR"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"EXPR" -> "ACFS"

-----

```

domain EXPUPXS ExponentialOfUnivariatePuisseuxSeries

— **ExponentialOfUnivariatePuisseuxSeries.input** —

```

)set break resume
)sys rm -f ExponentialOfUnivariatePuisseuxSeries.output
)spool ExponentialOfUnivariatePuisseuxSeries.output
)set message test on
)set message auto off

```

```

)clear all

--S 1 of 1
)show ExponentialOfUnivariatePuisseuxSeries
--R
--R ExponentialOfUnivariatePuisseuxSeries(FE: Join(Field,OrderedSet),var: Symbol,cen: FE) is a d
--R Abbreviation for ExponentialOfUnivariatePuisseuxSeries is EXPUPXS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for EXPUPXS
--R
--R----- Operations -----
--R ?? : (FE,%) -> %          ?? : (%,FE) -> %
--R ?? : (%,%) -> %          ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %    ??? : (%,PositiveInteger) -> %
--R ?? : (%,%) -> %          ?-? : (%,%) -> %
--R -? : % -> %              ?<? : (%,%) -> Boolean
--R ?<=? : (%,%) -> Boolean    ?=? : (%,%) -> Boolean
--R ?>? : (%,%) -> Boolean    ?>=? : (%,%) -> Boolean
--R 1 : () -> %              0 : () -> %
--R ?? : (%,PositiveInteger) -> %    center : % -> FE
--R coerce : Integer -> %          coerce : % -> OutputForm
--R complete : % -> %          degree : % -> Fraction(Integer)
--R .? : (%,Fraction(Integer)) -> FE  hash : % -> SingleInteger
--R inv : % -> % if FE has FIELD      latex : % -> String
--R leadingCoefficient : % -> FE      leadingMonomial : % -> %
--R map : ((FE -> FE),%) -> %        max : (%,%) -> %
--R min : (%,%) -> %              monomial? : % -> Boolean
--R one? : % -> Boolean            order : % -> Fraction(Integer)
--R pole? : % -> Boolean           recip : % -> Union(%, "failed")
--R reductum : % -> %             sample : () -> %
--R variable : % -> Symbol         zero? : % -> Boolean
--R ?^=? : (%,%) -> Boolean
--R ?? : (%,Fraction(Integer)) -> % if FE has ALGEBRA(FRAC(INT))
--R ?? : (Fraction(Integer),%) -> % if FE has ALGEBRA(FRAC(INT))
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%,Fraction(Integer)) -> % if FE has ALGEBRA(FRAC(INT))
--R ??? : (%,%) -> % if FE has ALGEBRA(FRAC(INT))
--R ??? : (%,Integer) -> % if FE has FIELD
--R ??? : (%,NonNegativeInteger) -> %
--R ?/? : (%,%) -> % if FE has FIELD
--R ?/? : (%,FE) -> % if FE has FIELD
--R D : % -> % if FE has *: (Fraction(Integer),FE) -> FE
--R D : (%,NonNegativeInteger) -> % if FE has *: (Fraction(Integer),FE) -> FE
--R D : (%,Symbol) -> % if FE has *: (Fraction(Integer),FE) -> FE and FE has PDRING(SYMBOL)
--R D : (%,List(Symbol)) -> % if FE has *: (Fraction(Integer),FE) -> FE and FE has PDRING(SYMBOL)
--R D : (%,Symbol,NonNegativeInteger) -> % if FE has *: (Fraction(Integer),FE) -> FE and FE has
--R D : (%,List(Symbol),List(NonNegativeInteger)) -> % if FE has *: (Fraction(Integer),FE) -> FE
--R ?? : (%,Integer) -> % if FE has FIELD
--R ?? : (%,NonNegativeInteger) -> %
--R acos : % -> % if FE has ALGEBRA(FRAC(INT))

```

```

--R acosh : % -> % if FE has ALGEBRA(FRAC(INT))
--R acot : % -> % if FE has ALGEBRA(FRAC(INT))
--R acoth : % -> % if FE has ALGEBRA(FRAC(INT))
--R acsc : % -> % if FE has ALGEBRA(FRAC(INT))
--R acsch : % -> % if FE has ALGEBRA(FRAC(INT))
--R approximate : (% , Fraction(Integer)) -> FE if FE has **: (FE, Fraction(Integer)) -> FE and FE has coerce:
--R asec : % -> % if FE has ALGEBRA(FRAC(INT))
--R asech : % -> % if FE has ALGEBRA(FRAC(INT))
--R asin : % -> % if FE has ALGEBRA(FRAC(INT))
--R asinh : % -> % if FE has ALGEBRA(FRAC(INT))
--R associates? : (% , %) -> Boolean if FE has INTDOM
--R atan : % -> % if FE has ALGEBRA(FRAC(INT))
--R atanh : % -> % if FE has ALGEBRA(FRAC(INT))
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(% , "failed") if FE has CHARNZ
--R coefficient : (% , Fraction(Integer)) -> FE
--R coerce : % -> % if FE has INTDOM
--R coerce : Fraction(Integer) -> % if FE has ALGEBRA(FRAC(INT))
--R coerce : FE -> % if FE has COMRING
--R cos : % -> % if FE has ALGEBRA(FRAC(INT))
--R cosh : % -> % if FE has ALGEBRA(FRAC(INT))
--R cot : % -> % if FE has ALGEBRA(FRAC(INT))
--R coth : % -> % if FE has ALGEBRA(FRAC(INT))
--R csc : % -> % if FE has ALGEBRA(FRAC(INT))
--R csch : % -> % if FE has ALGEBRA(FRAC(INT))
--R differentiate : % -> % if FE has *: (Fraction(Integer), FE) -> FE
--R differentiate : (% , NonNegativeInteger) -> % if FE has *: (Fraction(Integer), FE) -> FE
--R differentiate : (% , Symbol) -> % if FE has *: (Fraction(Integer), FE) -> FE and FE has PDRING(SYMBOL)
--R differentiate : (% , List(Symbol)) -> % if FE has *: (Fraction(Integer), FE) -> FE and FE has PDRING(SYMBOL)
--R differentiate : (% , Symbol, NonNegativeInteger) -> % if FE has *: (Fraction(Integer), FE) -> FE and FE has
--R differentiate : (% , List(Symbol), List(NonNegativeInteger)) -> % if FE has *: (Fraction(Integer), FE) -> FE
--R divide : (% , %) -> Record(quotient: % , remainder: %) if FE has FIELD
--R ?? : (% , %) -> % if Fraction(Integer) has SGROUP
--R euclideanSize : % -> NonNegativeInteger if FE has FIELD
--R eval : (% , FE) -> Stream(FE) if FE has **: (FE, Fraction(Integer)) -> FE
--R exp : % -> % if FE has ALGEBRA(FRAC(INT))
--R exponent : % -> UnivariatePuisseuxSeries(FE, var, cen)
--R exponential : UnivariatePuisseuxSeries(FE, var, cen) -> %
--R exponentialOrder : % -> Fraction(Integer)
--R expressIdealMember : (List(% , %) -> Union(List(% , "failed") if FE has FIELD
--R exquo : (% , %) -> Union(% , "failed") if FE has INTDOM
--R extend : (% , Fraction(Integer)) -> %
--R extendedEuclidean : (% , %) -> Record(coef1: % , coef2: % , generator: %) if FE has FIELD
--R extendedEuclidean : (% , % , %) -> Union(Record(coef1: % , coef2: % , "failed") if FE has FIELD
--R factor : % -> Factored(%) if FE has FIELD
--R gcd : (% , %) -> % if FE has FIELD
--R gcd : List(%) -> % if FE has FIELD
--R gcdPolynomial : (SparseUnivariatePolynomial(% , SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial
--R integrate : (% , Symbol) -> % if FE has integrate: (FE, Symbol) -> FE and FE has variables: FE -> List(Symbol)
--R integrate : % -> % if FE has ALGEBRA(FRAC(INT))

```

```

--R lcm : (% , %) -> % if FE has FIELD
--R lcm : List(%) -> % if FE has FIELD
--R log : % -> % if FE has ALGEBRA(FRAC(INT))
--R monomial : (% , List(SingletonAsOrderedSet), List(Fraction(Integer))) -> %
--R monomial : (% , SingletonAsOrderedSet, Fraction(Integer)) -> %
--R monomial : (FE, Fraction(Integer)) -> %
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed") if FE has FIELD
--R multiplyExponents : (% , Fraction(Integer)) -> %
--R multiplyExponents : (% , PositiveInteger) -> %
--R nthRoot : (% , Integer) -> % if FE has ALGEBRA(FRAC(INT))
--R order : (% , Fraction(Integer)) -> Fraction(Integer)
--R pi : () -> % if FE has ALGEBRA(FRAC(INT))
--R prime? : % -> Boolean if FE has FIELD
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %) if FE has FIELD
--R ?quo? : (% , %) -> % if FE has FIELD
--R ?rem? : (% , %) -> % if FE has FIELD
--R sec : % -> % if FE has ALGEBRA(FRAC(INT))
--R sech : % -> % if FE has ALGEBRA(FRAC(INT))
--R series : (NonNegativeInteger, Stream(Record(k: Fraction(Integer), c: FE))) -> %
--R sin : % -> % if FE has ALGEBRA(FRAC(INT))
--R sinh : % -> % if FE has ALGEBRA(FRAC(INT))
--R sizeLess? : (% , %) -> Boolean if FE has FIELD
--R sqrt : % -> % if FE has ALGEBRA(FRAC(INT))
--R squareFree : % -> Factored(%) if FE has FIELD
--R squareFreePart : % -> % if FE has FIELD
--R subtractIfCan : (% , %) -> Union(% , "failed")
--R tan : % -> % if FE has ALGEBRA(FRAC(INT))
--R tanh : % -> % if FE has ALGEBRA(FRAC(INT))
--R terms : % -> Stream(Record(k: Fraction(Integer), c: FE))
--R truncate : (% , Fraction(Integer), Fraction(Integer)) -> %
--R truncate : (% , Fraction(Integer)) -> %
--R unit? : % -> Boolean if FE has INTDOM
--R unitCanonical : % -> % if FE has INTDOM
--R unitNormal : % -> Record(unit: % , canonical: % , associate: %) if FE has INTDOM
--R variables : % -> List(SingletonAsOrderedSet)
--R
--E 1

)spool
)lisp (bye)

```

— ExponentialOfUnivariatePuisseuxSeries.help —

```

=====
ExponentialOfUnivariatePuisseuxSeries examples
=====

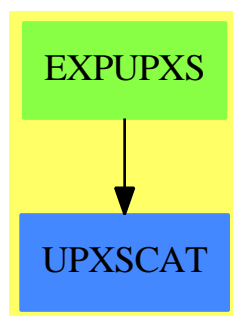
```

`ExponentialOfUnivariatePuisseuxSeries` is a domain used to represent essential singularities of functions. An object in this domain is a function of the form $\exp(f(x))$, where $f(x)$ is a Puiseux series with no terms of non-negative degree. Objects are ordered according to order of singularity, with functions which tend more rapidly to zero or infinity considered to be larger. Thus, if $\text{order}(f(x)) < \text{order}(g(x))$, i.e. the first non-zero term of $f(x)$ has lower degree than the first non-zero term of $g(x)$, then $\exp(f(x)) > \exp(g(x))$. If $\text{order}(f(x)) = \text{order}(g(x))$, then the ordering is essentially random. This domain is used in computing limits involving functions with essential singularities.

See Also:

o `show ExponentialOfUnivariatePuisseuxSeries`

ExponentialOfUnivariatePuisseuxSeries (EXPUPXS)



See

⇒ “UnivariatePuisseuxSeriesWithExponentialSingularity” (UPXSSING) [22](#) on page [2852](#)

⇒ “ExponentialExpansion” (EXPEXPAN) [6](#) on page [711](#)

Exports:

0	1	acos	acosh
acot	acoth	acsc	acsch
approximate	asec	asech	asin
asinh	associates?	atan	atanh
center	characteristic	charthRoot	coefficient
coerce	complete	cos	cosh
cot	coth	csc	csch
D	degree	differentiate	divide
euclideanSize	eval	exp	exponent
exponential	exponentialOrder	expressIdealMember	exquo
extend	extendedEuclidean	factor	gcd
gcdPolynomial	hash	integrate	inv
latex	lcm	leadingCoefficient	leadingMonomial
log	map	max	min
monomial	monomial?	multiEuclidean	multiplyExponents
multiplyExponents	nthRoot	one?	order
pi	pole?	prime?	principalIdeal
recip	reductum	sample	sec
sech	series	sin	sinh
sizeLess?	sqrt	squareFree	squareFreePart
subtractIfCan	tan	tanh	terms
truncate	unit?	unitCanonical	unitNormal
variable	variables	zero?	?*?
?**?	?+?	?-?	-?
?<?	?<=?	?=?	?>?
?>=?	?^?	?..?	?~=?
?/?	?quo?	?rem?	

— domain EXPUPXS ExponentialOfUnivariatePuisseuxSeries —

```

)abbrev domain EXPUPXS ExponentialOfUnivariatePuisseuxSeries
++ Author: Clifton J. Williamson
++ Date Created: 4 August 1992
++ Date Last Updated: 27 August 1992
++ References:
++ Description:
++ ExponentialOfUnivariatePuisseuxSeries is a domain used to represent
++ essential singularities of functions. An object in this domain is a
++ function of the form \spad{exp(f(x))}, where \spad{f(x)} is a Puiseux
++ series with no terms of non-negative degree. Objects are ordered
++ according to order of singularity, with functions which tend more
++ rapidly to zero or infinity considered to be larger. Thus, if
++ \spad{order(f(x)) < order(g(x))}, i.e. the first non-zero term of
++ \spad{f(x)} has lower degree than the first non-zero term of \spad{g(x)},
++ then \spad{exp(f(x)) > exp(g(x))}. If \spad{order(f(x)) = order(g(x))},
++ then the ordering is essentially random. This domain is used
++ in computing limits involving functions with essential singularities.

```

```

ExponentialOfUnivariatePuisseuxSeries(FE,var,cen):_
  Exports == Implementation where
  FE : Join(Field,OrderedSet)
  var : Symbol
  cen : FE
  UPXS ==> UnivariatePuisseuxSeries(FE,var,cen)

Exports ==> Join(UnivariatePuisseuxSeriesCategory(FE),OrderedAbelianMonoid) _
  with
  exponential : UPXS -> %
    ++ exponential(f(x)) returns \spad{exp(f(x))}.
    ++ Note: the function does NOT check that \spad{f(x)} has no
    ++ non-negative terms.
  exponent : % -> UPXS
    ++ exponent(exp(f(x))) returns \spad{f(x)}
  exponentialOrder: % -> Fraction Integer
    ++ exponentialOrder(exp(c * x **(-n) + ...)) returns \spad{-n}.
    ++ exponentialOrder(0) returns \spad{0}.

Implementation ==> UPXS add

Rep := UPXS

exponential f == complete f
exponent f == f pretend UPXS
exponentialOrder f == order(exponent f,0)

zero? f == empty? entries complete terms f

f = g ==
-- we redefine equality because we know that we are dealing with
-- a FINITE series, so there is no danger in computing all terms
(entries complete terms f) = (entries complete terms g)

f < g ==
  zero? f => not zero? g
  zero? g => false
  (ordf := exponentialOrder f) > (ordg := exponentialOrder g) => true
  ordf < ordg => false
  (fCoef := coefficient(f,ordf)) = (gCoef := coefficient(g,ordg)) =>
    reductum(f) < reductum(g)
  fCoef < gCoef -- this is "random" if FE is EXPR INT

coerce(f:%):OutputForm ==
  ("%e" :: OutputForm) ** ((coerce$Rep)(complete f)@OutputForm)

```

— EXPUPXS.dotabb —

```
"EXPUPXS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=EXPUPXS"]
"UPXSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UPXSCAT"]
"EXPUPXS" -> "UPXSCAT"
```

domain EAB ExtAlgBasis

— ExtAlgBasis.input —

```
)set break resume
)sys rm -f ExtAlgBasis.output
)spool ExtAlgBasis.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExtAlgBasis
--R
--R ExtAlgBasis is a domain constructor
--R Abbreviation for ExtAlgBasis is EAB
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for EAB
--R
--R----- Operations -----
--R ?<? : (%,% ) -> Boolean          ?<=? : (%,% ) -> Boolean
--R ?=? : (%,% ) -> Boolean          ?>? : (%,% ) -> Boolean
--R ?>=? : (%,% ) -> Boolean          Nul : NonNegativeInteger -> %
--R coerce : List(Integer) -> %      coerce : % -> OutputForm
--R degree : % -> NonNegativeInteger exponents : % -> List(Integer)
--R hash : % -> SingleInteger         latex : % -> String
--R max : (%,% ) -> %                 min : (%,% ) -> %
--R ?~=? : (%,% ) -> Boolean
--R
--E 1

)spool
)lisp (bye)
```

— ExtAlgBasis.help —

```
=====
ExtAlgBasis examples
=====
```

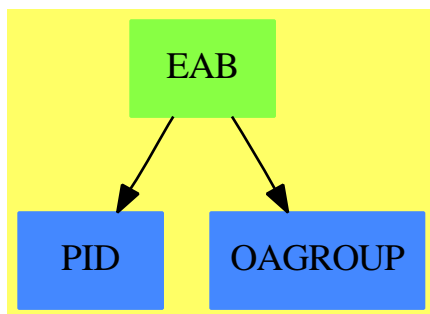
A domain used in the construction of the exterior algebra on a set X over a ring R . This domain represents the set of all ordered subsets of the set X , assumed to be in correspondance with $\{1,2,3, \dots\}$. The ordered subsets are themselves ordered lexicographically and are in bijective correspondance with an ordered basis of the exterior algebra. In this domain we are dealing strictly with the exponents of basis elements which can only be 0 or 1. Thus we really have $L(\{0,1\})$.

The multiplicative identity element of the exterior algebra corresponds to the empty subset of X . A coerce from List Integer to an ordered basis element is provided to allow the convenient input of expressions. Another exported function forgets the ordered structure and simply returns the list corresponding to an ordered subset.

See Also:

```
o )show ExtAlgBasis
```

ExtAlgBasis (EAB)



See

⇒ “AntiSymm” (ANTISYM) [2](#) on page [40](#)

⇒ “DeRhamComplex” (DERHAM) [5](#) on page [530](#)

Exports:

coerce	degree	exponents	hash	latex
max	min	Nul	?~=?	?<?
?<=?	?=?	?>?	?>=?	

— domain EAB ExtAlgBasis —

```

)abbrev domain EAB ExtAlgBasis
++ Author: Larry Lambe
++ Date created: 03/14/89
++ Description:
++ A domain used in the construction of the exterior algebra on a set
++ X over a ring R. This domain represents the set of all ordered
++ subsets of the set X, assumed to be in correspondance with
++ {1,2,3, ...}. The ordered subsets are themselves ordered
++ lexicographically and are in bijective correspondance with an ordered
++ basis of the exterior algebra. In this domain we are dealing strictly
++ with the exponents of basis elements which can only be 0 or 1.
-- Thus we really have L({0,1}).
++
++ The multiplicative identity element of the exterior algebra corresponds
++ to the empty subset of X. A coerce from List Integer to an
++ ordered basis element is provided to allow the convenient input of
++ expressions. Another exported function forgets the ordered structure
++ and simply returns the list corresponding to an ordered subset.

ExtAlgBasis(): Export == Implement where
  I   ==> Integer
  L   ==> List
  NNI ==> NonNegativeInteger

  Export == OrderedSet with
    coerce      : L I -> %
      ++ coerce(l) converts a list of 0's and 1's into a basis
      ++ element, where 1 (respectively 0) designates that the
      ++ variable of the corresponding index of l is (respectively, is not)
      ++ present.
      ++ Error: if an element of l is not 0 or 1.
    degree      : % -> NNI
      ++ degree(x) gives the numbers of 1's in x, i.e., the number
      ++ of non-zero exponents in the basis element that x represents.
    exponents   : % -> L I
      ++ exponents(x) converts a domain element into a list of zeros
      ++ and ones corresponding to the exponents in the basis element
      ++ that x represents.
-- subscripts : % -> L I
-- subscripts(x) looks at the exponents in x and converts
-- them to the proper subscripts
  Nul          : NNI -> %
      ++ Nul() gives the basis element 1 for the algebra generated
      ++ by n generators.

  Implement == add
    Rep := L I
    x,y : %

    x = y == x =$Rep y

```

```

x < y ==
  null x          => not null y
  null y          => false
  first x = first y => rest x < rest y
  first x > first y

coerce(li:(L I)) ==
  for x in li repeat
    if x ^= 1 and x ^= 0 then error "coerce: values can only be 0 and 1"
  li

degree x          == (_+/x)::NNI

exponents x       == copy(x @ Rep)

-- subscripts x   ==
--   cntr:I := 1
--   result: L I := []
--   for j in x repeat
--     if j = 1 then result := cons(cntr,result)
--     cntr:=cntr+1
--   reverse_! result

Nul n             == [0 for i in 1..n]

coerce x          == coerce(x @ Rep)$(L I)

```

— EAB.dotabb —

```

"EAB" [color="#88FF44",href="bookvol10.3.pdf#nameddest=EAB"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"EAB" -> "PID"
"EAB" -> "OAGROUP"

```

domain E04DGFA e04dgfAnnaType

— e04dgfAnnaType.input —

```
)set break resume
```

```

)sys rm -f e04dgfAnnaType.output
)spool e04dgfAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show e04dgfAnnaType
--R
--R e04dgfAnnaType is a domain constructor
--R Abbreviation for e04dgfAnnaType is E04DGFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for E04DGFA
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger        latex : % -> String
--R ?~=? : (%,% ) -> Boolean
--R measure : (RoutinesTable,Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat)))
--R measure : (RoutinesTable,Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: List
--R numericalOptimization : Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: List(
--R numericalOptimization : Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat)) -
--R
--E 1

)spool
)lisp (bye)

```

— e04dgfAnnaType.help —

=====

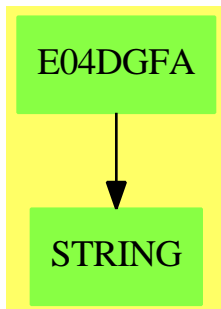
e04dgfAnnaType examples

=====

e04dgfAnnaType is a domain of NumericalOptimization for the NAG routine E04DGF, a general optimization routine which can handle some singularities in the input function. The function measure measures the usefulness of the routine E04DGF for the given problem. The function numericalOptimization performs the optimization by using NagOptimisationPackage.

See Also:

o)show e04dgfAnnaType

e04dgfAnnaType (E04DGFA)**See**

- ⇒ “E04FDFA” (e04fdfAnnaType) [6](#) on page [750](#)
- ⇒ “E04GCFA” (e04gcfAnnaType) [6](#) on page [754](#)
- ⇒ “E04JAFA” (e04jafAnnaType) [6](#) on page [758](#)
- ⇒ “E04MBFA” (e04mbfAnnaType) [6](#) on page [762](#)
- ⇒ “E04NAFA” (e04nafAnnaType) [6](#) on page [765](#)
- ⇒ “E04UCFA” (e04ucfAnnaType) [6](#) on page [769](#)

Exports:

coerce hash latex measure numericalOptimization ?? ?~=?

— domain E04DGFA e04dgfAnnaType —

```

)abbrev domain E04DGFA e04dgfAnnaType
++ Author: Brian Dupee
++ Date Created: February 1996
++ Date Last Updated: February 1996
++ Description:
++ \axiomType{e04dgfAnnaType} is a domain of \axiomType{NumericalOptimization}
++ for the NAG routine E04DGF, a general optimization routine which
++ can handle some singularities in the input function. The function
++ \axiomFun{measure} measures the usefulness of the routine E04DGF
++ for the given problem. The function \axiomFun{numericalOptimization}
++ performs the optimization by using \axiomType{NagOptimisationPackage}.

```

```

e04dgfAnnaType(): NumericalOptimizationCategory == Result add
  DF ==> DoubleFloat
  EF ==> Expression Float
  EDF ==> Expression DoubleFloat
  PDF ==> Polynomial DoubleFloat
  VPDF ==> Vector Polynomial DoubleFloat
  LDF ==> List DoubleFloat
  LOCDF ==> List OrderedCompletion DoubleFloat
  MDF ==> Matrix DoubleFloat
  MPDF ==> Matrix Polynomial DoubleFloat

```

```

MF ==> Matrix Float
MEF ==> Matrix Expression Float
LEDF ==> List Expression DoubleFloat
VEF ==> Vector Expression Float
NOA ==> Record(fn:EDF, init:LDF, lb:LOCDF, cf:LEDF, ub:LOCDF)
LSA ==> Record(lfn:LEDF, init:LDF)
EF2 ==> ExpressionFunctions2
MI ==> Matrix Integer
INT ==> Integer
F ==> Float
NNI ==> NonNegativeInteger
S ==> Symbol
LS ==> List Symbol
MVCF ==> MultiVariableCalculusFunctions
ESTOOLS2 ==> ExpertSystemToolsPackage2
SDF ==> Stream DoubleFloat
LSDF ==> List Stream DoubleFloat
SOCDF ==> Segment OrderedCompletion DoubleFloat
OCDF ==> OrderedCompletion DoubleFloat

Rep:=Result
import Rep, NagOptimisationPackage, ExpertSystemToolsPackage

measure(R:RoutinesTable,args:NOA) ==
  string:String := "e04dgm is "
  positive?((args.cf) + (args.lb) + (args.ub)) =>
    string := concat(string,"unsuitable for constrained problems. ")
    [0.0,string]
  string := concat(string,"recommended")
  [getMeasure(R,e04dgm@Symbol)$RoutinesTable, string]

numericalOptimization(args:NOA) ==
  argsFn:EDF := args.fn
  n:NNI := #(variables(argsFn)$EDF)
  fu:DF := float(4373903597,-24,10)$DF
  it:INT := max(50,5*n)
  lin:DF := float(9,-1,10)$DF
  ma:DF := float(1,20,10)$DF
  op:DF := float(326,-14,10)$DF
  x:MDF := mat(args.init,n)
  ArgsFn:Expression Float := edf2ef(argsFn)
  f:Union(fn:FileName,fp:Asp49(OBJFUN)) := [retract(ArgsFn)$Asp49(OBJFUN)]
  e04dgm(n,1$DF,fu,it,lin,true,ma,op,1,1,n,0,x,-1,f)

```

```
"E04DGFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=E04DGFA"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"E04DGFA" -> "STRING"
```

domain E04FDFA e04fdfAnnaType

— e04fdfAnnaType.input —

```
)set break resume
)sys rm -f e04fdfAnnaType.output
)spool e04fdfAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show e04fdfAnnaType
--R
--R e04fdfAnnaType is a domain constructor
--R Abbreviation for e04fdfAnnaType is E04FDFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for E04FDFA
--R
--R----- Operations -----
--R ?? : (% ,%) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger      latex : % -> String
--R ?~=? : (% ,%) -> Boolean
--R measure : (RoutinesTable,Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat))) -> Record(
--R measure : (RoutinesTable,Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: List(OrderedComp
--R numericalOptimization : Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: List(OrderedComp
--R numericalOptimization : Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat)) -> Result
--R
--E 1

)spool
)lisp (bye)
```

— e04fdfAnnaType.help —

```
=====
e04fdfAnnaType examples
```



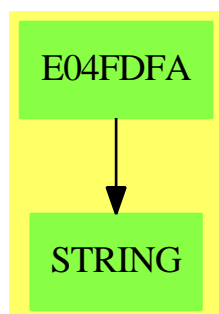
```
=====

e04fdfAnnaType is a domain of NumericalOptimization for the NAG routine
E04FDF, a general optimization routine which can handle some singularities
in the input function. The function measure measures the usefulness of the
routine E04FDF for the given problem. The function numericalOptimization
performs the optimization by using NagOptimisationPackage.
```

See Also:

```
o )show e04fdfAnnaType
```

e04fdfAnnaType (E04FDFA)



See

```
⇒ “E04DGFa” (e04dgfAnnaType) 6 on page 747
⇒ “E04GCFA” (e04gcfAnnaType) 6 on page 754
⇒ “E04JAFA” (e04jafAnnaType) 6 on page 758
⇒ “E04MBFA” (e04mbfAnnaType) 6 on page 762
⇒ “E04NAFA” (e04nafAnnaType) 6 on page 765
⇒ “E04UCFA” (e04ucfAnnaType) 6 on page 769
```

Exports:

```
coerce hash latex measure numericalOptimization ?? ?~=?
```

— domain E04FDFA e04fdfAnnaType —

```
)abbrev domain E04FDFA e04fdfAnnaType
++ Author: Brian Dupee
++ Date Created: February 1996
++ Date Last Updated: February 1996
++ Description:
++ \axiomType{e04fdfAnnaType} is a domain of \axiomType{NumericalOptimization}
++ for the NAG routine E04FDF, a general optimization routine which
```

```

++ can handle some singularities in the input function. The function
++ \axiomFun{measure} measures the usefulness of the routine E04FDF
++ for the given problem. The function \axiomFun{numericalOptimization}
++ performs the optimization by using \axiomType{NagOptimisationPackage}.

```

```

e04fdfAnnaType(): NumericalOptimizationCategory == Result add
  DF ==> DoubleFloat
  EF ==> Expression Float
  EDF ==> Expression DoubleFloat
  PDF ==> Polynomial DoubleFloat
  VPDF ==> Vector Polynomial DoubleFloat
  LDF ==> List DoubleFloat
  LOCDF ==> List OrderedCompletion DoubleFloat
  MDF ==> Matrix DoubleFloat
  MPDF ==> Matrix Polynomial DoubleFloat
  MF ==> Matrix Float
  MEF ==> Matrix Expression Float
  LEDF ==> List Expression DoubleFloat
  VEF ==> Vector Expression Float
  NOA ==> Record(fn:EDF, init:LDF, lb:LOCDF, cf:LEDF, ub:LOCDF)
  LSA ==> Record(lfn:LEDF, init:LDF)
  EF2 ==> ExpressionFunctions2
  MI ==> Matrix Integer
  INT ==> Integer
  F ==> Float
  NNI ==> NonNegativeInteger
  S ==> Symbol
  LS ==> List Symbol
  MVCF ==> MultiVariableCalculusFunctions
  ESTOOLS2 ==> ExpertSystemToolsPackage2
  SDF ==> Stream DoubleFloat
  LSDF ==> List Stream DoubleFloat
  SOCDF ==> Segment OrderedCompletion DoubleFloat
  OCDF ==> OrderedCompletion DoubleFloat

Rep:=Result
import Rep, NagOptimisationPackage
import e04AgentsPackage,ExpertSystemToolsPackage

measure(R:RoutinesTable,args:NOA) ==
  argsFn := args.fn
  string:String := "e04fdf is "
  positive?((#(args.cf) + #(args.lb) + #(args.ub)) =>
    string := concat(string,"unsuitable for constrained problems. ")
    [0.0,string]
  n:NNI := #(variables(argsFn)$EDF)
  (n>1)@Boolean =>
    string := concat(string,"unsuitable for single instances of multivariate problems. ")
    [0.0,string]
  sumOfSquares(argsFn) case "failed" =>

```

```

    string := concat(string,"unsuitable.")
    [0.0,string]
    string := concat(string,"recommended since the function is a sum of squares.")
    [getMeasure(R,e04fdf@Symbol)$RoutinesTable, string]

measure(R:RoutinesTable,args:LSA) ==
    string:String := "e04fdf is recommended"
    [getMeasure(R,e04fdf@Symbol)$RoutinesTable, string]

numericalOptimization(args:NOA) ==
    argsFn := args.fn
    lw:INT := 14
    x := mat(args.init,1)
    (a := sumOfSquares(argsFn)) case EDF =>
        ArgsFn := vector([edf2ef(a)])$VEF
        f : Union(fn:FileName,fp:Asp50(LSFUN1)) := [retract(ArgsFn)$Asp50(LSFUN1)]
        out:Result := e04fdf(1,1,1,lw,x,-1,f)
        changeNameToObjf(fsumsq@Symbol,out)
    empty()$Result

numericalOptimization(args:LSA) ==
    argsFn := copy args.lfn
    m:INT := #(argsFn)
    n:NNI := #(variables(args))
    nn:INT := n
    lw:INT :=
--      one?(nn) => 9+5*m
      (nn = 1) => 9+5*m
      nn*(7+n+2*m+((nn-1) quo 2)$INT)+3*m
    x := mat(args.init,n)
    ArgsFn := vector([edf2ef(i)$ExpertSystemToolsPackage for i in argsFn])$VEF
    f : Union(fn:FileName,fp:Asp50(LSFUN1)) := [retract(ArgsFn)$Asp50(LSFUN1)]
    out:Result := e04fdf(m,n,1,lw,x,-1,f)
    changeNameToObjf(fsumsq@Symbol,out)

-----

— E04FDFA.dotabb —

"E04FDFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=E04FDFA"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"E04FDFA" -> "STRING"

-----

```

domain E04GCFA e04gcfAnnaType**— e04gcfAnnaType.input —**

```

)set break resume
)sys rm -f e04gcfAnnaType.output
)spool e04gcfAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show e04gcfAnnaType
--R
--R e04gcfAnnaType is a domain constructor
--R Abbreviation for e04gcfAnnaType is E04GCFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for E04GCFA
--R
--R----- Operations -----
--R ==? : (% ,%) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger        latex : % -> String
--R ~=? : (% ,%) -> Boolean
--R measure : (RoutinesTable,Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat))) -> Record(
--R measure : (RoutinesTable,Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: List(OrderedComp
--R numericalOptimization : Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: List(OrderedComp
--R numericalOptimization : Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat)) -> Result
--R
--E 1

)spool
)lisp (bye)

```

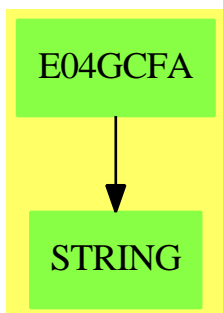
— e04gcfAnnaType.help —**=====**
e04gcfAnnaType examples
=====

e04gcfAnnaType is a domain of NumericalOptimization for the NAG routine E04GCF, a general optimization routine which can handle some singularities in the input function. The function measure measures the usefulness of the routine E04GCF for the given problem. The function numericalOptimization performs the optimization by using NagOptimisationPackage.

See Also:

```
o )show e04gcfAnnaType
```

e04gcfAnnaType (E04GCFA)



See

⇒ “E04DGFA” (e04dgfAnnaType) [6](#) on page [747](#)
 ⇒ “E04FDFA” (e04fdfAnnaType) [6](#) on page [750](#)
 ⇒ “E04JAFA” (e04jafAnnaType) [6](#) on page [758](#)
 ⇒ “E04MBFA” (e04mbfAnnaType) [6](#) on page [762](#)
 ⇒ “E04NAFA” (e04nafAnnaType) [6](#) on page [765](#)
 ⇒ “E04UCFA” (e04ucfAnnaType) [6](#) on page [769](#)

Exports:

coerce hash latex measure numericalOptimization ?=? ?~=?

— domain E04GCFA e04gcfAnnaType —

```
)abbrev domain E04GCFA e04gcfAnnaType
++ Author: Brian Dupee
++ Date Created: February 1996
++ Date Last Updated: February 1996
++ Description:
++ \axiomType{e04gcfAnnaType} is a domain of \axiomType{NumericalOptimization}
++ for the NAG routine E04GCF, a general optimization routine which
++ can handle some singularities in the input function. The function
++ \axiomFun{measure} measures the usefulness of the routine E04GCF
++ for the given problem. The function \axiomFun{numericalOptimization}
++ performs the optimization by using \axiomType{NagOptimisationPackage}.

e04gcfAnnaType(): NumericalOptimizationCategory == Result add
  DF ==> DoubleFloat
  EF ==> Expression Float
  EDF ==> Expression DoubleFloat
```

```

PDF ==> Polynomial DoubleFloat
VPDF ==> Vector Polynomial DoubleFloat
LDF ==> List DoubleFloat
LOCDF ==> List OrderedCompletion DoubleFloat
MDF ==> Matrix DoubleFloat
MPDF ==> Matrix Polynomial DoubleFloat
MF ==> Matrix Float
MEF ==> Matrix Expression Float
LEDF ==> List Expression DoubleFloat
VEF ==> Vector Expression Float
NOA ==> Record(fn:EDF, init:LDF, lb:LOCDF, cf:LEDF, ub:LOCDF)
LSA ==> Record(lfn:LEDF, init:LDF)
EF2 ==> ExpressionFunctions2
MI ==> Matrix Integer
INT ==> Integer
F ==> Float
NNI ==> NonNegativeInteger
S ==> Symbol
LS ==> List Symbol
MVCF ==> MultiVariableCalculusFunctions
ESTOOLS2 ==> ExpertSystemToolsPackage2
SDF ==> Stream DoubleFloat
LSDF ==> List Stream DoubleFloat
SOCDF ==> Segment OrderedCompletion DoubleFloat
OCDF ==> OrderedCompletion DoubleFloat

Rep:=Result
import Rep, NagOptimisationPackage,ExpertSystemContinuityPackage
import e04AgentsPackage,ExpertSystemToolsPackage

measure(R:RoutinesTable,args:NOA) ==
  argsFn:EDF := args.fn
  string:String := "e04gcf is "
  positive?((#(args.cf) + #(args.lb) + #(args.ub)) =>
    string := concat(string,"unsuitable for constrained problems. ")
    [0.0,string]
  n:NNI := #(variables(argsFn)$EDF)
  (n>1)@Boolean =>
    string := concat(string,"unsuitable for single instances of multivariate problems. ")
    [0.0,string]
  a := coerce(float(10,0,10))$OCDF
  seg:SOCDF := -a..a
  sings := singularitiesOf(argsFn,variables(argsFn)$EDF,seg)
  s := #(sdf2lst(sings))
  positive? s =>
    string := concat(string,"not recommended for discontinuous functions.")
    [0.0,string]
  sumOfSquares(args.fn) case "failed" =>
    string := concat(string,"unsuitable.")
    [0.0,string]

```

```

string := concat(string,"recommended since the function is a sum of squares.")
[getMeasure(R,e04gcf@Symbol)$RoutinesTable, string]

measure(R:RoutinesTable,args:LSA) ==
  string:String := "e04gcf is "
  a := coerce(float(10,0,10))$OCDF
  seg:SOCDF := -a..a
  sings := concat([singularitiesOf(i,variables(args),seg) for i in args.lfn])$SDF
  s := #(sdf2lst(sings))
  positive? s =>
    string := concat(string,"not recommended for discontinuous functions.")
    [0.0,string]
  string := concat(string,"recommended.")
  m := getMeasure(R,e04gcf@Symbol)$RoutinesTable
  m := m-(1-exp(-(expenseOfEvaluation(args))*3))
  [m, string]

numericalOptimization(args:NOA) ==
  argsFn:EDF := args.fn
  lw:INT := 16
  x := mat(args.init,1)
  (a := sumOfSquares(argsFn)) case EDF =>
    ArgsFn := vector([edf2ef(a)$ExpertSystemToolsPackage])$VEF
    f : Union(fn:FileName,fp:Asp19(LSFUN2)) := [retract(ArgsFn)$Asp19(LSFUN2)]
    out:Result := e04gcf(1,1,1,lw,x,-1,f)
    changeNameToObjf(fsumsq@Symbol,out)
  empty()$Result

numericalOptimization(args:LSA) ==
  argsFn := copy args.lfn
  m:NNI := #(argsFn)
  n:NNI := #(variables(args))
  lw:INT :=
--   one?(n) => 11+5*m
    (n = 1) => 11+5*m
    2*n*(4+n+m)+3*m
  x := mat(args.init,n)
  ArgsFn := vector([edf2ef(i)$ExpertSystemToolsPackage for i in argsFn])$VEF
  f : Union(fn:FileName,fp:Asp19(LSFUN2)) := [retract(ArgsFn)$Asp19(LSFUN2)]
  out:Result := e04gcf(m,n,1,lw,x,-1,f)
  changeNameToObjf(fsumsq@Symbol,out)

```

— E04GCFA.dotabb —

"E04GCFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=E04GCFA"]
 "STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]

"E04GCFA" -> "STRING"

domain E04JAFA e04jafAnnaType

— e04jafAnnaType.input —

```
)set break resume
)sys rm -f e04jafAnnaType.output
)spool e04jafAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show e04jafAnnaType
--R
--R e04jafAnnaType is a domain constructor
--R Abbreviation for e04jafAnnaType is E04JAFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for E04JAFA
--R
--R----- Operations -----
--R ?? : (% ,%) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger      latex : % -> String
--R ~=? : (% ,%) -> Boolean
--R measure : (RoutinesTable,Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat))) -> Record(
--R measure : (RoutinesTable,Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: List(OrderedComp
--R numericalOptimization : Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: List(OrderedComp
--R numericalOptimization : Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat)) -> Result
--R
--E 1

)spool
)lisp (bye)
```

— e04jafAnnaType.help —

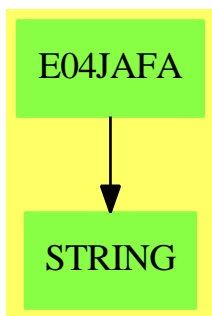
```
=====
e04jafAnnaType examples
=====
```


`e04jafAnnaType` is a domain of `NumericalOptimization` for the NAG routine `E04JAF`, a general optimization routine which can handle some singularities in the input function. The function `measure` measures the usefulness of the routine `E04JAF` for the given problem. The function `numericalOptimization` performs the optimization by using `NagOptimisationPackage`.

See Also:

o `)show e04jafAnnaType`

`e04jafAnnaType` (E04JAFA)



See

⇒ “E04DGF” (`e04dgfAnnaType`) [6](#) on page [747](#)
 ⇒ “E04FDFA” (`e04fdfAnnaType`) [6](#) on page [750](#)
 ⇒ “E04GCFA” (`e04gcfAnnaType`) [6](#) on page [754](#)
 ⇒ “E04MBFA” (`e04mbfAnnaType`) [6](#) on page [762](#)
 ⇒ “E04NAFA” (`e04nafAnnaType`) [6](#) on page [765](#)
 ⇒ “E04UCFA” (`e04ucfAnnaType`) [6](#) on page [769](#)

Exports:

`coerce` `hash` `latex` `measure` `numericalOptimization` `??` `?~=?`

— domain E04JAFA `e04jafAnnaType` —

```

)abbrev domain E04JAFA e04jafAnnaType
++ Author: Brian Dupee
++ Date Created: February 1996
++ Date Last Updated: February 1996
++ Description:
++ \axiomType{e04jafAnnaType} is a domain of \axiomType{NumericalOptimization}
++ for the NAG routine E04JAF, a general optimization routine which
++ can handle some singularities in the input function. The function
++ \axiomFun{measure} measures the usefulness of the routine E04JAF
  
```

```

++ for the given problem. The function \axiomFun{numericalOptimization}
++ performs the optimization by using \axiomType{NagOptimisationPackage}.

```

```

e04jafAnnaType(): NumericalOptimizationCategory == Result add
  DF ==> DoubleFloat
  EF ==> Expression Float
  EDF ==> Expression DoubleFloat
  PDF ==> Polynomial DoubleFloat
  VPDF ==> Vector Polynomial DoubleFloat
  LDF ==> List DoubleFloat
  LOCDF ==> List OrderedCompletion DoubleFloat
  MDF ==> Matrix DoubleFloat
  MPDF ==> Matrix Polynomial DoubleFloat
  MF ==> Matrix Float
  MEF ==> Matrix Expression Float
  LEDF ==> List Expression DoubleFloat
  VEF ==> Vector Expression Float
  NOA ==> Record(fn:EDF, init:LDF, lb:LOCDF, cf:LEDF, ub:LOCDF)
  LSA ==> Record(lfn:LEDF, init:LDF)
  EF2 ==> ExpressionFunctions2
  MI ==> Matrix Integer
  INT ==> Integer
  F ==> Float
  NNI ==> NonNegativeInteger
  S ==> Symbol
  LS ==> List Symbol
  MVCF ==> MultiVariableCalculusFunctions
  ESTOOLS2 ==> ExpertSystemToolsPackage2
  SDF ==> Stream DoubleFloat
  LSDF ==> List Stream DoubleFloat
  SOCDF ==> Segment OrderedCompletion DoubleFloat
  OCDF ==> OrderedCompletion DoubleFloat

Rep:=Result
import Rep, NagOptimisationPackage
import e04AgentsPackage,ExpertSystemToolsPackage

bound(a:LOCDF,b:LOCDF):Integer ==
  empty?(concat(a,b)) => 1
--   one?((#(removeDuplicates(a))) and zero?(first(a))) => 2
--   ((#(removeDuplicates(a)) = 1) and zero?(first(a))) => 2
--   one?((#(removeDuplicates(a))) and one?((#(removeDuplicates(b)))) => 3
--   ((#(removeDuplicates(a)) = 1) and ((#(removeDuplicates(b)) = 1) => 3
  0

measure(R:RoutinesTable,args:NOA) ==
  string:String := "e04jaf is "
  if positive?((#(args.cf))) then
    if not simpleBounds?(args.cf) then
      string :=

```

```

        concat(string,"suitable for simple bounds only, not constraint functions.")
(# string) < 20 =>
    if zero?((args.lb) + (args.ub)) then
        string := concat(string, "usable if there are no constraints")
        [getMeasure(R,e04jaf@Symbol)$RoutinesTable*0.5,string]
    else
        string := concat(string,"recommended")
        [getMeasure(R,e04jaf@Symbol)$RoutinesTable, string]
[0.0,string]

numericalOptimization(args:NOA) ==
    argsFn:EDF := args.fn
    n:NNI := #(variables(argsFn)$EDF)
    ibound:INT := bound(args.lb,args.ub)
    m:INT := n
    lw:INT := max(13,12 * m + ((m * (m - 1)) quo 2)$INT)$INT
    bl := mat(finiteBound(args.lb,float(1,6,10)$DF),n)
    bu := mat(finiteBound(args.ub,float(1,6,10)$DF),n)
    x := mat(args.init,n)
    ArgsFn:EF := edf2ef(argsFn)
    fr:Union(fn:FileName,fp:Asp24(FUNCT1)) := [retract(ArgsFn)$Asp24(FUNCT1)]
    out:Result := e04jaf(n,ibound,n+2,lw,bl,bu,x,-1,fr)
    changeNameToObjf(f@Symbol,out)

```

— E04JAFA.dotabb —

```

"E04JAFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=E04JAFA"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"E04JAFA" -> "STRING"

```

domain E04MBFA e04mbfAnnaType

— e04mbfAnnaType.input —

```

)set break resume
)sys rm -f e04mbfAnnaType.output
)spool e04mbfAnnaType.output
)set message test on
)set message auto off
)clear all

```

```

--S 1 of 1
)show e04mbfAnnaType
--R
--R e04mbfAnnaType is a domain constructor
--R Abbreviation for e04mbfAnnaType is E04MBFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for E04MBFA
--R
--R----- Operations -----
--R ?? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger      latex : % -> String
--R ?~=? : (%,% ) -> Boolean
--R measure : (RoutinesTable,Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat))) -> Record(
--R measure : (RoutinesTable,Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: List(OrderedComp
--R numericalOptimization : Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: List(OrderedComp
--R numericalOptimization : Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat)) -> Result
--R
--E 1

)spool
)lisp (bye)

```

— e04mbfAnnaType.help —

```

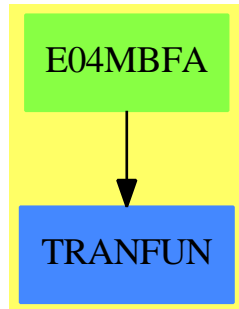
=====
e04mbfAnnaType examples
=====

```

e04mbfAnnaType is a domain of NumericalOptimization for the NAG routine E04MBF, an optimization routine for Linear functions. The function measure measures the usefulness of the routine E04MBF for the given problem. The function numericalOptimization performs the optimization by using NagOptimisationPackage.

See Also:

o)show e04mbfAnnaType

e04mbfAnnaType (E04MBFA)**See**

- ⇒ “E04DGFA” (e04dgfAnnaType) [6](#) on page [747](#)
- ⇒ “E04FDFA” (e04fdfAnnaType) [6](#) on page [750](#)
- ⇒ “E04GCFA” (e04gcfAnnaType) [6](#) on page [754](#)
- ⇒ “E04JAFA” (e04jafAnnaType) [6](#) on page [758](#)
- ⇒ “E04NAFA” (e04nafAnnaType) [6](#) on page [765](#)
- ⇒ “E04UCFA” (e04ucfAnnaType) [6](#) on page [769](#)

Exports:

coerce hash latex measure numericalOptimization ?? ?~=?

— domain E04MBFA e04mbfAnnaType —

```

)abbrev domain E04MBFA e04mbfAnnaType
++ Author: Brian Dupee
++ Date Created: February 1996
++ Date Last Updated: February 1996
++ Description:
++ \axiomType{e04mbfAnnaType} is a domain of \axiomType{NumericalOptimization}
++ for the NAG routine E04MBF, an optimization routine for Linear functions.
++ The function
++ \axiomFun{measure} measures the usefulness of the routine E04MBF
++ for the given problem. The function \axiomFun{numericalOptimization}
++ performs the optimization by using \axiomType{NagOptimisationPackage}.

e04mbfAnnaType(): NumericalOptimizationCategory == Result add
  DF ==> DoubleFloat
  EF ==> Expression Float
  EDF ==> Expression DoubleFloat
  PDF ==> Polynomial DoubleFloat
  VPDF ==> Vector Polynomial DoubleFloat
  LDF ==> List DoubleFloat
  LOCDF ==> List OrderedCompletion DoubleFloat
  MDF ==> Matrix DoubleFloat
  MPDF ==> Matrix Polynomial DoubleFloat
  
```

```

MF ==> Matrix Float
MEF ==> Matrix Expression Float
LEDF ==> List Expression DoubleFloat
VEF ==> Vector Expression Float
NOA ==> Record(fn:EDF, init:LDF, lb:LOCDF, cf:LEDF, ub:LOCDF)
LSA ==> Record(lfn:LEDF, init:LDF)
EF2 ==> ExpressionFunctions2
MI ==> Matrix Integer
INT ==> Integer
F ==> Float
NNI ==> NonNegativeInteger
S ==> Symbol
LS ==> List Symbol
MVCF ==> MultiVariableCalculusFunctions
ESTOOLS2 ==> ExpertSystemToolsPackage2
SDF ==> Stream DoubleFloat
LSDF ==> List Stream DoubleFloat
SOCDF ==> Segment OrderedCompletion DoubleFloat
OCDF ==> OrderedCompletion DoubleFloat

Rep:=Result
import Rep, NagOptimisationPackage
import e04AgentsPackage,ExpertSystemToolsPackage

measure(R:RoutinesTable,args:NOA) ==
  (not linear?([args.fn])) or (not linear?(args.cf)) =>
    [0.0,"e04mbf is for a linear objective function and constraints only."]
  [getMeasure(R,e04mbf@Symbol)$RoutinesTable,"e04mbf is recommended" ]

numericalOptimization(args:NOA) ==
  argsFn:EDF := args.fn
  c := args.cf
  listVars:List LS := concat(variables(argsFn)$EDF,[variables(z)$EDF for z in c])
  n:NNI := #(v := removeDuplicates(concat(listVars)$LS)$LS)
  A:MDF := linearMatrix(args.cf,n)
  nclin:NNI := # linearPart(c)
  nrowa:NNI := max(1,nclin)
  bl:MDF := mat(finiteBound(args.lb,float(1,21,10)$DF),n)
  bu:MDF := mat(finiteBound(args.ub,float(1,21,10)$DF),n)
  cvec:MDF := mat(coefficients(retract(argsFn)@PDF)$PDF,n)
  x := mat(args.init,n)
  lwork:INT :=
    nclin < n => 2*nclin*(nclin+4)+2+6*n+nrowa
    2*(n+3)*n+4*nclin+nrowa
  out:Result := e04mbf(20,1,n,nclin,n+nclin,nrowa,A,bl,bu,cvec,true,2*n,lwork,x,-1)
  changeNameToObjf(objlp@Symbol,out)

```

— E04MBFA.dotabb —

```
"E04MBFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=E04MBFA"]
"TRANFUN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TRANFUN"]
"E04MBFA" -> "TRANFUN"
```

domain E04NAFA e04nafAnnaType

— e04nafAnnaType.input —

```
)set break resume
)sys rm -f e04nafAnnaType.output
)spool e04nafAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show e04nafAnnaType
--R
--R e04nafAnnaType is a domain constructor
--R Abbreviation for e04nafAnnaType is E04NAFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for E04NAFA
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger        latex : % -> String
--R ?~=? : (%,% ) -> Boolean
--R measure : (RoutinesTable,Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat)))
--R measure : (RoutinesTable,Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: List
--R numericalOptimization : Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: List
--R numericalOptimization : Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat)) -
--R
--E 1

)spool
)lisp (bye)
```

— e04nafAnnaType.help —

```
=====
e04nafAnnaType examples
=====
```

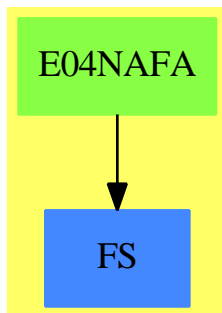
e04nafAnnaType is a domain of NumericalOptimization for the NAG routine E04NAF, an optimization routine for Quadratic functions. The function measure measures the usefulness of the routine E04NAF for the given problem. The function numericalOptimization performs the optimization by using NagOptimisationPackage.

See Also:

o)show e04nafAnnaType

—————

e04nafAnnaType (E04NAFA)



See

- ⇒ “E04DGFA” (e04dgfAnnaType) 6 on page 747
- ⇒ “E04FDFA” (e04fdfAnnaType) 6 on page 750
- ⇒ “E04GCFA” (e04gcfAnnaType) 6 on page 754
- ⇒ “E04JAFA” (e04jafAnnaType) 6 on page 758
- ⇒ “E04MBFA” (e04mbfAnnaType) 6 on page 762
- ⇒ “E04UCFA” (e04ucfAnnaType) 6 on page 769

Exports:

coerce hash latex measure numericalOptimization ?=? ?~=?

— domain E04NAFA e04nafAnnaType —

```
)abbrev domain E04NAFA e04nafAnnaType
++ Author: Brian Dupee
++ Date Created: February 1996
++ Date Last Updated: February 1996
++ Description:
```



```

++ \axiomType{e04nafAnnaType} is a domain of \axiomType{NumericalOptimization}
++ for the NAG routine E04NAF, an optimization routine for Quadratic functions.
++ The function
++ \axiomFun{measure} measures the usefulness of the routine E04NAF
++ for the given problem. The function \axiomFun{numericalOptimization}
++ performs the optimization by using \axiomType{NagOptimisationPackage}.

```

```

e04nafAnnaType(): NumericalOptimizationCategory == Result add
  DF ==> DoubleFloat
  EF ==> Expression Float
  EDF ==> Expression DoubleFloat
  PDF ==> Polynomial DoubleFloat
  VPDF ==> Vector Polynomial DoubleFloat
  LDF ==> List DoubleFloat
  LOCDF ==> List OrderedCompletion DoubleFloat
  MDF ==> Matrix DoubleFloat
  MPDF ==> Matrix Polynomial DoubleFloat
  MF ==> Matrix Float
  MEF ==> Matrix Expression Float
  LEDF ==> List Expression DoubleFloat
  VEF ==> Vector Expression Float
  NOA ==> Record(fn:EDF, init:LDF, lb:LOCDF, cf:LEDF, ub:LOCDF)
  LSA ==> Record(lfn:LEDF, init:LDF)
  EF2 ==> ExpressionFunctions2
  MI ==> Matrix Integer
  INT ==> Integer
  F ==> Float
  NNI ==> NonNegativeInteger
  S ==> Symbol
  LS ==> List Symbol
  MVCF ==> MultiVariableCalculusFunctions
  ESTOOLS2 ==> ExpertSystemToolsPackage2
  SDF ==> Stream DoubleFloat
  LSDF ==> List Stream DoubleFloat
  SOCDF ==> Segment OrderedCompletion DoubleFloat
  OCDF ==> OrderedCompletion DoubleFloat

```

```
Rep:=Result
```

```
import Rep, NagOptimisationPackage
```

```
import e04AgentsPackage,ExpertSystemToolsPackage
```

```
measure(R:RoutinesTable,args:NOA) ==
```

```
  string:String := "e04naf is "
```

```
  argsFn:EDF := args.fn
```

```
  if not (quadratic?(argsFn) and linear?(args.cf)) then
```

```
    string :=
```

```
      concat(string,"for a quadratic function with linear constraints only.")
```

```
  (# string) < 20 =>
```

```
    string := concat(string,"recommended")
```

```
  [getMeasure(R,e04naf@Symbol)$RoutinesTable, string]
```

```

[0.0,string]

numericalOptimization(args:NOA) ==
  argsFn:EDF := args.fn
  c := args.cf
  listVars:List LS := concat(variables(argsFn)$EDF,[variables(z)$EDF for z in c])
  n:NNI := #(v := sort(removeDuplicates(concat(listVars)$LS)$LS)$LS)
  A:MDF := linearMatrix(c,n)
  nclin:NNI := # linearPart(c)
  nrowa:NNI := max(1,nclin)
  big:DF := float(1,10,10)$DF
  fea:MDF := new(1,n+nclin,float(1053,-11,10)$DF)$MDF
  bl:MDF := mat(finiteBound(args.lb,float(1,21,10)$DF),n)
  bu:MDF := mat(finiteBound(args.ub,float(1,21,10)$DF),n)
  alin:EDF := splitLinear(argsFn)
  p:PDF := retract(alin)@PDF
  pl:List PDF := [coefficient(p,i,1)$PDF for i in v]
  cvec:MDF := mat([pdf2df j for j in pl],n)
  h1:MPDF := hessian(p,v)$MVCf(S,PDF,VPDF,LS)
  hess:MDF := map(pdf2df,h1)$ESTOOLS2(PDF,DF)
  h2:MEF := map(df2ef,hess)$ESTOOLS2(DF,EF)
  x := mat(args.init,n)
  istate:MI := zero(1,n+nclin)$MI
  lwork:INT := 2*n*(n+2*nclin)+nrowa
  qp Hess:Union(fn:FileName,fp:Asp20(QPHESS)) := [retract(h2)$Asp20(QPHESS)]
  out:Result := e04naf(20,1,n,nclin,n+nclin,nrowa,n,n,big,A,bl,bu,cvec,fea,
    hess,true,false,true,2*n,lwork,x,istate,-1,qp Hess)
  changeNameToObjf(obj@Symbol,out)

```

— E04NAFA.dotabb —

```

"E04NAFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=E04NAFA"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"E04NAFA" -> "FS"

```

domain E04UCFA e04ucfAnnaType

— e04ucfAnnaType.input —

```

)set break resume

```

```

)sys rm -f e04ucfAnnaType.output
)spool e04ucfAnnaType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show e04ucfAnnaType
--R
--R e04ucfAnnaType is a domain constructor
--R Abbreviation for e04ucfAnnaType is E04UCFA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for E04UCFA
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger        latex : % -> String
--R ?~=? : (%,% ) -> Boolean
--R measure : (RoutinesTable,Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat)))
--R measure : (RoutinesTable,Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: List
--R numericalOptimization : Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: List(
--R numericalOptimization : Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat)) -
--R
--E 1

)spool
)lisp (bye)

```

— e04ucfAnnaType.help —

=====

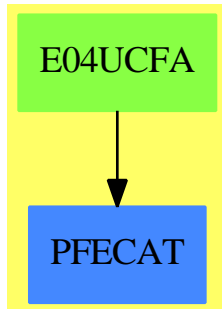
e04ucfAnnaType examples

=====

e04ucfAnnaType is a domain of NumericalOptimization for the NAG routine E04UCF, a general optimization routine which can handle some singularities in the input function. The function measure measures the usefulness of the routine E04UCF for the given problem. The function numericalOptimization performs the optimization by using NagOptimisationPackage.

See Also:

o)show e04ucfAnnaType

e04ucfAnnaType (E04UCFA)**See**

- ⇒ “E04DGFA” (e04dgfAnnaType) [6](#) on page [747](#)
- ⇒ “E04FDFA” (e04fdfAnnaType) [6](#) on page [750](#)
- ⇒ “E04GCFA” (e04gcfAnnaType) [6](#) on page [754](#)
- ⇒ “E04JAFA” (e04jafAnnaType) [6](#) on page [758](#)
- ⇒ “E04MBFA” (e04mbfAnnaType) [6](#) on page [762](#)
- ⇒ “E04NAFA” (e04nafAnnaType) [6](#) on page [765](#)

Exports:

coerce hash latex measure numericalOptimization ?? ?~=?

— domain E04UCFA e04ucfAnnaType —

```

)abbrev domain E04UCFA e04ucfAnnaType
++ Author: Brian Dupee
++ Date Created: February 1996
++ Date Last Updated: November 1997
++ Description:
++ \axiomType{e04ucfAnnaType} is a domain of \axiomType{NumericalOptimization}
++ for the NAG routine E04UCF, a general optimization routine which
++ can handle some singularities in the input function. The function
++ \axiomFun{measure} measures the usefulness of the routine E04UCF
++ for the given problem. The function \axiomFun{numericalOptimization}
++ performs the optimization by using \axiomType{NagOptimisationPackage}.

```

```

e04ucfAnnaType(): NumericalOptimizationCategory == Result add
  DF ==> DoubleFloat
  EF ==> Expression Float
  EDF ==> Expression DoubleFloat
  PDF ==> Polynomial DoubleFloat
  VPDF ==> Vector Polynomial DoubleFloat
  LDF ==> List DoubleFloat
  LOCDF ==> List OrderedCompletion DoubleFloat
  MDF ==> Matrix DoubleFloat
  MPDF ==> Matrix Polynomial DoubleFloat

```

```

MF ==> Matrix Float
MEF ==> Matrix Expression Float
LEDF ==> List Expression DoubleFloat
VEF ==> Vector Expression Float
NOA ==> Record(fn:EDF, init:LDF, lb:LOCDF, cf:LEDF, ub:LOCDF)
LSA ==> Record(lfn:LEDF, init:LDF)
EF2 ==> ExpressionFunctions2
MI ==> Matrix Integer
INT ==> Integer
F ==> Float
NNI ==> NonNegativeInteger
S ==> Symbol
LS ==> List Symbol
MVCF ==> MultiVariableCalculusFunctions
ESTOOLS2 ==> ExpertSystemToolsPackage2
SDF ==> Stream DoubleFloat
LSDF ==> List Stream DoubleFloat
SOCDF ==> Segment OrderedCompletion DoubleFloat
OCDF ==> OrderedCompletion DoubleFloat

Rep:=Result
import Rep,NagOptimisationPackage
import e04AgentsPackage,ExpertSystemToolsPackage

measure(R:RoutinesTable,args:NOA) ==
  zero?((args.lb) + (args.ub)) =>
    [0.0,"e04ucf is not recommended if there are no bounds specified"]
  zero?((args.cf)) =>
    string:String := "e04ucf is usable but not always recommended if there are no constraints"
    [getMeasure(R,e04ucf@Symbol)$RoutinesTable*0.5,string]
    [getMeasure(R,e04ucf@Symbol)$RoutinesTable,"e04ucf is recommended"]

numericalOptimization(args:NOA) ==
  Args := sortConstraints(args)
  argsFn := Args.fn
  c := Args.cf
  listVars:List LS := concat(variables(argsFn)$EDF,[variables(z)$EDF for z in c])
  n:NNI := #(v := sort(removeDuplicates(concat(listVars)$LS)$LS)$LS)
  lin:NNI := #(linearPart(c))
  nlcf := nonLinearPart(c)
  nonlin:NNI := #(nlcf)
  if empty?(nlcf) then
    nlcf := new(n,coerce(first(v)$LS)$EDF)$LEDF
  nrowa:NNI := max(1,lin)
  nrowj:NNI := max(1,nonlin)
  A:MDF := linearMatrix(c,n)
  bl:MDF := mat(finiteBound(Args.lb,float(1,25,10)$DF),n)
  bu:MDF := mat(finiteBound(Args.ub,float(1,25,10)$DF),n)
  liwork:INT := 3*n+lin+2*nonlin
  lwork:INT :=

```

```

zero?(lin+nonlin) => 20*n
zero?(nonlin) => 2*n*(n+10)+11*lin
2*n*(n+nonlin+10)+(11+n)*lin + 21*nonlin
cra:DF := float(1,-2,10)$DF
fea:DF := float(1053671201,-17,10)$DF
fun:DF := float(4373903597,-24,10)$DF
infb:DF := float(1,15,10)$DF
lint:DF := float(9,-1,10)$DF
maji:INT := max(50,3*(n+lin)+10*nonlin)
mini:INT := max(50,3*(n+lin+nonlin))
nonf:DF := float(105,-10,10)$DF
opt:DF := float(326,-10,10)$DF
ste:DF := float(2,0,10)$DF
istate:MI := zero(1,n+lin+nonlin)$MI
cjac:MDF :=
  positive?(nonlin) => zero(nrowj,n)$MDF
  zero(nrowj,1)$MDF
clambda:MDF := zero(1,n+lin+nonlin)$MDF
r:MDF := zero(n,n)$MDF
x:MDF := mat(Args.init,n)
VectCF:VEF := vector([edf2ef e for e in nlcf])$VEF
ArgsFn:EF := edf2ef(argsFn)
fasp : Union(fn:FileName,fp:Asp49(OBJFUN)) := [retract(ArgsFn)$Asp49(OBJFUN)]
casp : Union(fn:FileName,fp:Asp55(CONFUN)) := [retract(VectCF)$Asp55(CONFUN)]
e04ucf(n,lin,nonlin,nrowa,nrowj,n,A,bl,bu,liwork,lwork,false,cra,3,fea,
  fun,true,infb,infb,fea,lint,true,maji,1,mini,0,-1,nonf,opt,ste,1,
  1,n,n,3,istate,cjac,clambda,r,x,-1,casp,fasp)

```

— E04UCFA.dotabb —

```

"E04UCFA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=E04UCFA"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"E04UCFA" -> "PFECAT"

```

Chapter 7

Chapter F

domain FR Factored

— Factored.input —

```
)set break resume
)sys rm -f Factored.output
)spool Factored.output
)set message test on
)set message auto off
)clear all
```

```
--S 1 of 38
```

```
g := factor(4312)
```

```
--R
```

```
--R
```

```
--R      3 2
```

```
--R (1)  2 7 11
```

```
--R
```

Type: Factored(Integer)

```
--E 1
```

```
--S 2 of 38
```

```
unit(g)
```

```
--R
```

```
--R
```

```
--R (2)  1
```

```
--R
```

Type: PositiveInteger

```
--E 2
```

```
--S 3 of 38
```

```
numberOfFactors(g)
```

```
--R
```

```
--R
```



```

--R (3) 3
--R                                         Type: PositiveInteger
--E 3

--S 4 of 38
[nthFactor(g,i) for i in 1..numberOfFactors(g)]
--R
--R
--R (4) [2,7,11]
--R                                         Type: List(Integer)
--E 4

--S 5 of 38
[nthExponent(g,i) for i in 1..numberOfFactors(g)]
--R
--R
--R (5) [3,2,1]
--R                                         Type: List(Integer)
--E 5

--S 6 of 38
[nthFlag(g,i) for i in 1..numberOfFactors(g)]
--R
--R
--R (6) ["prime","prime","prime"]
--R                                         Type: List(Union("nil","sqfr","irred","prime"))
--E 6

--S 7 of 38
factorList(g)
--R
--R
--R (7)
--R [[flg= "prime",fctr= 2,xpnt= 3], [flg= "prime",fctr= 7,xpnt= 2],
--R  [flg= "prime",fctr= 11,xpnt= 1]]
--RType: List(Record(flg: Union("nil","sqfr","irred","prime"),fctr: Integer,xpnt: Integer))
--E 7

--S 8 of 38
factors(g)
--R
--R
--R (8)
--R [[factor= 2,exponent= 3],[factor= 7,exponent= 2],[factor= 11,exponent= 1]]
--R                                         Type: List(Record(factor: Integer,exponent: Integer))
--E 8

--S 9 of 38
first(%).factor
--R

```

```

--R
--R (9)  2
--R
--R                                         Type: PositiveInteger
--E 9

--S 10 of 38
g := factor(4312)
--R
--R
--R          3 2
--R (10)  2 7 11
--R
--R                                         Type: Factored(Integer)
--E 10

--S 11 of 38
expand(g)
--R
--R
--R (11)  4312
--R
--R                                         Type: PositiveInteger
--E 11

--S 12 of 38
reduce(*,[t.factor for t in factors(g)])
--R
--R
--R (12)  154
--R
--R                                         Type: PositiveInteger
--E 12

--S 13 of 38
g := factor(4312)
--R
--R
--R          3 2
--R (13)  2 7 11
--R
--R                                         Type: Factored(Integer)
--E 13

--S 14 of 38
f := factor(246960)
--R
--R
--R          4 2 3
--R (14)  2 3 5 7
--R
--R                                         Type: Factored(Integer)
--E 14

--S 15 of 38
f * g

```

Type: Factored(Integer)

Type: Factored(Integer)

Type: Factored(Integer)

Type: Factored(Integer)

Type: Factored(Integer)

Type: Factored(Integer)

--E 20

--S 21 of 38

zero?(factor(0))

--R

--R

--R (21) true

--R

Type: Boolean

--E 21

--S 22 of 38

zero?(g)

--R

--R

--R (22) false

--R

Type: Boolean

--E 22

--S 23 of 38

one?(factor(1))

--R

--R

--R (23) true

--R

Type: Boolean

--E 23

--S 24 of 38

one?(f)

--R

--R

--R (24) false

--R

Type: Boolean

--E 24

--S 25 of 38

0\$Factored(Integer)

--R

--R

--R (25) 0

--R

Type: Factored(Integer)

--E 25

--S 26 of 38

1\$Factored(Integer)

--R

--R

--R (26) 1

--R

Type: Factored(Integer)

--E 26

```
--S 27 of 38
nilFactor(24,2)
--R
--R
--R      2
--R (27) 24
--R
--E 27
```

Type: Factored(Integer)

```
--S 28 of 38
nthFlag(%,1)
--R
--R
--R (28) "nil"
--R
--E 28
```

Type: Union("nil",...)

```
--S 29 of 38
sqfrFactor(30,2)
--R
--R
--R      2
--R (29) 30
--R
--E 29
```

Type: Factored(Integer)

```
--S 30 of 38
irreducibleFactor(13,10)
--R
--R
--R      10
--R (30) 13
--R
--E 30
```

Type: Factored(Integer)

```
--S 31 of 38
primeFactor(11,5)
--R
--R
--R      5
--R (31) 11
--R
--E 31
```

Type: Factored(Integer)

```
--S 32 of 38
h := factor(-720)
--R
--R
--R      4 2
--R (32) - 2 3 5
```

```

--R                                                    Type: Factored(Integer)
--E 32

--S 33 of 38
h - makeFR(unit(h),factorList(h))
--R
--R
--R (33)  0
--R
--R                                                    Type: Factored(Integer)
--E 33

--S 34 of 38
p := (4*x*x-12*x+9)*y*y + (4*x*x-12*x+9)*y + 28*x*x - 84*x + 63
--R
--R
--R (34)   $(4x^2 - 12x + 9)y^2 + (4x^2 - 12x + 9)y + 28x^2 - 84x + 63$ 
--R
--R                                                    Type: Polynomial(Integer)
--E 34

--S 35 of 38
fp := factor(p)
--R
--R
--R (35)   $(2x^2 - 3)(y^2 + y + 7)$ 
--R
--R                                                    Type: Factored(Polynomial(Integer))
--E 35

--S 36 of 38
D(p,x)
--R
--R
--R (36)   $(8x^2 - 12)y^2 + (8x^2 - 12)y + 56x - 84$ 
--R
--R                                                    Type: Polynomial(Integer)
--E 36

--S 37 of 38
D(fp,x)
--R
--R
--R (37)   $4(2x^2 - 3)(y^2 + y + 7)$ 
--R
--R                                                    Type: Factored(Polynomial(Integer))
--E 37

--S 38 of 38
numberOfFactors(%)
--R

```

```

--R
--R (38) 3
--R
--R                                         Type: PositiveInteger
--E 38
)spool
)lisp (bye)

```

— Factored.help —

=====

Factored examples

=====

Factored creates a domain whose objects are kept in factored form as long as possible. Thus certain operations like * (multiplication) and gcd are relatively easy to do. Others, such as addition, require somewhat more work, and the result may not be completely factored unless the argument domain R provides a factor operation. Each object consists of a unit and a list of factors, where each factor consists of a member of R (the base), an exponent, and a flag indicating what is known about the base. A flag may be one of "nil", "sqfr", "irred" or "prime", which mean that nothing is known about the base, it is square-free, it is irreducible, or it is prime, respectively. The current restriction to factored objects of integral domains allows simplification to be performed without worrying about multiplication order.

=====

Decomposing Factored Objects

=====

In this section we will work with a factored integer.

```

g := factor(4312)
      3 2
      2 7 11

```

Type: Factored Integer

Let's begin by decomposing g into pieces. The only possible units for integers are 1 and -1.

```

unit(g)
      1

```

Type: PositiveInteger

There are three factors.

```

numberOfFactors(g)
3
                                Type: PositiveInteger

```

We can make a list of the bases, ...

```

[nthFactor(g,i) for i in 1..numberOfFactors(g)]
[2,7,11]
                                Type: List Integer

```

and the exponents, ...

```

[nthExponent(g,i) for i in 1..numberOfFactors(g)]
[3,2,1]
                                Type: List Integer

```

and the flags. You can see that all the bases (factors) are prime.

```

[nthFlag(g,i) for i in 1..numberOfFactors(g)]
["prime","prime","prime"]
                                Type: List Union("nil","sqfr","irred","prime")

```

A useful operation for pulling apart a factored object into a list of records of the components is `factorList`.

```

factorList(g)
[[flg= "prime",fctr= 2,xpnt= 3], [flg= "prime",fctr= 7,xpnt= 2],
 [flg= "prime",fctr= 11,xpnt= 1]]
                                Type: List Record(flg: Union("nil","sqfr","irred","prime"),
                                fctr: Integer,xpnt: Integer)

```

If you don't care about the flags, use `factors`.

```

factors(g)
[[factor= 2,exponent= 3],[factor= 7,exponent= 2],[factor= 11,exponent= 1]]
                                Type: List Record(factor: Integer,exponent: Integer)

```

Neither of these operations returns the unit.

```

first(%).factor
2
                                Type: PositiveInteger

```

Expanding Factored Objects

Recall that we are working with this factored integer.

```

g := factor(4312)

```



```

3 2
2 7 11

```

Type: Factored Integer

To multiply out the factors with their multiplicities, use `expand`.

```

expand(g)
4312

```

Type: PositiveInteger

If you would like, say, the distinct factors multiplied together but with multiplicity one, you could do it this way.

```

reduce(*,[t.factor for t in factors(g)])
154

```

Type: PositiveInteger

=====

Arithmetic with Factored Objects

=====

We're still working with this factored integer.

```

g := factor(4312)
3 2
2 7 11

```

Type: Factored Integer

We'll also define this factored integer.

```

f := factor(246960)
4 2 3
2 3 5 7

```

Type: Factored Integer

Operations involving multiplication and division are particularly easy with factored objects.

```

f * g
7 2 5
2 3 5 7 11

```

Type: Factored Integer

```

f**500
2000 1000 500 1500
2 3 5 7

```

Type: Factored Integer

```

gcd(f,g)
3 2

```

```
2 7
Type: Factored Integer
```

```
lcm(f,g)
  4 2 3
  2 3 5 7 11
Type: Factored Integer
```

If we use addition and subtraction things can slow down because we may need to compute greatest common divisors.

```
f + g
  3 2
  2 7 641
Type: Factored Integer
```

```
f - g
  3 2
  2 7 619
Type: Factored Integer
```

Test for equality with 0 and 1 by using zero? and one?, respectively.

```
zero?(factor(0))
true
Type: Boolean
```

```
zero?(g)
false
Type: Boolean
```

```
one?(factor(1))
true
Type: Boolean
```

```
one?(f)
false
Type: Boolean
```

Another way to get the zero and one factored objects is to use package calling.

```
0$Factored(Integer)
0
Type: Factored Integer
```

```
1$Factored(Integer)
1
Type: Factored Integer
```

```
=====
Creating New Factored Objects
=====
```

The map operation is used to iterate across the unit and bases of a factored object.

The following four operations take a base and an exponent and create a factored object. They differ in handling the flag component.

```
nilFactor(24,2)
      2
     24
                                Type: Factored Integer
```

This factor has no associated information.

```
nthFlag(%,1)
"nil"
                                Type: Union("nil",...)
```

This factor is asserted to be square-free.

```
sqfrFactor(30,2)
      2
     30
                                Type: Factored Integer
```

This factor is asserted to be irreducible.

```
irreducibleFactor(13,10)
      10
     13
                                Type: Factored Integer
```

This factor is asserted to be prime.

```
primeFactor(11,5)
      5
     11
                                Type: Factored Integer
```

A partial inverse to factorList is makeFR.

```
h := factor(-720)
      4 2
     - 2 3 5
                                Type: Factored Integer
```

The first argument is the unit and the second is a list of records as

returned by factorList.

```
h - makeFR(unit(h),factorList(h))
0
Type: Factored Integer
```

=====

Factored Objects with Variables

=====

Some of the operations available for polynomials are also available for factored polynomials.

```
p := (4*x*x-12*x+9)*y*y + (4*x*x-12*x+9)*y + 28*x*x - 84*x + 63
      2      2      2      2
(4x - 12x + 9)y + (4x - 12x + 9)y + 28x - 84x + 63
Type: Polynomial Integer
```

```
fp := factor(p)
      2 2
(2x - 3) (y + y + 7)
Type: Factored Polynomial Integer
```

You can differentiate with respect to a variable.

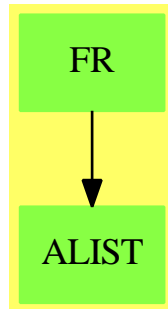
```
D(p,x)
      2
(8x - 12)y + (8x - 12)y + 56x - 84
Type: Polynomial Integer
```

```
D(fp,x)
      2
4(2x - 3)(y + y + 7)
Type: Factored Polynomial Integer
```

```
numberOfFactors(%)
3
Type: PositiveInteger
```

See Also:

- o)help FactoredFunctions2
- o)show Factored

Factored (FR)**Exports:**

0	1	associates?	characteristic	coerce
convert	D	differentiate	eval	expand
exponent	exquo	factor	factorList	factors
flagFactor	gcd	gcdPolynomial	hash	irreducibleFactor
latex	lcm	makeFR	map	nilFactor
nthExponent	nthFactor	nthFlag	numberOfFactors	one?
prime?	primeFactor	rational	rational?	rationalIfCan
recip	retract	retractIfCan	sample	sqfrFactor
squareFree	squareFreePart	subtractIfCan	unit	unit?
unitCanonical	unitNormal	unitNormalize	zero?	?*?
?**?	?+?	?-?	-?	?=?
?^?	?~=?	?..?		

— domain FR Factored —

```

)abbrev domain FR Factored
++ Author: Robert S. Sutor, J. Grabmeier
++ Date Created: 1985
++ Change History: 21 Jan 1991 (J Grabmeier) 16 Aug 1994 (R S Sutor)
++ Description:
++ \spadtype{Factored} creates a domain whose objects are kept in
++ factored form as long as possible. Thus certain operations like
++ multiplication and gcd are relatively easy to do. Others, like
++ addition require somewhat more work, and unless the argument
++ domain provides a factor function, the result may not be
++ completely factored. Each object consists of a unit and a list of
++ factors, where a factor has a member of R (the "base"), and
++ exponent and a flag indicating what is known about the base. A
++ flag may be one of "nil", "sqfr", "irred" or "prime", which respectively mean
++ that nothing is known about the base, it is square-free, it is
++ irreducible, or it is prime. The current
++ restriction to integral domains allows simplification to be
++ performed without worrying about multiplication order.

```

```

Factored(R: IntegralDomain): Exports == Implementation where
  fUnion ==> Union("nil", "sqfr", "irred", "prime")
  FF      ==> Record(flag: fUnion, fctr: R, xpnt: Integer)
  SRFE    ==> Set(Record(factor:R, exponent:Integer))

Exports ==> Join(IntegralDomain, DifferentialExtension R, Algebra R,
  FullyEvalableOver R, FullyRetractableTo R) with
expand: % -> R
  ++ expand(f) multiplies the unit and factors together, yielding an
  ++ "unfactored" object. Note: this is purposely not called
  ++ \spadfun{coerce} which would cause the interpreter to do this
  ++ automatically.
  ++
  ++X f:=nilFactor(y-x,3)
  ++X expand(f)

exponent: % -> Integer
  ++ exponent(u) returns the exponent of the first factor of
  ++ \spadvar{u}, or 0 if the factored form consists solely of a unit.
  ++
  ++X f:=nilFactor(y-x,3)
  ++X exponent(f)

makeFR : (R, List FF) -> %
  ++ makeFR(unit,listOfFactors) creates a factored object (for
  ++ use by factoring code).
  ++
  ++X f:=nilFactor(x-y,3)
  ++X g:=factorList f
  ++X makeFR(z,g)

factorList : % -> List FF
  ++ factorList(u) returns the list of factors with flags (for
  ++ use by factoring code).
  ++
  ++X f:=nilFactor(x-y,3)
  ++X factorList f

nilFactor: (R, Integer) -> %
  ++ nilFactor(base,exponent) creates a factored object with
  ++ a single factor with no information about the kind of
  ++ base (flag = "nil").
  ++
  ++X nilFactor(24,2)
  ++X nilFactor(x-y,3)

factors: % -> List Record(factor:R, exponent:Integer)
  ++ factors(u) returns a list of the factors in a form suitable
  ++ for iteration. That is, it returns a list where each element

```

```

++ is a record containing a base and exponent. The original
++ object is the product of all the factors and the unit (which
++ can be extracted by \axiom{unit(u)}).
++
++X f:=x*y^3-3*x^2*y^2+3*x^3*y-x^4
++X factors f
++X g:=makeFR(z,factorList f)
++X factors g

irreducibleFactor: (R, Integer) -> %
++ irreducibleFactor(base,exponent) creates a factored object with
++ a single factor whose base is asserted to be irreducible
++ (flag = "irred").
++
++X a:=irreducibleFactor(3,1)
++X nthFlag(a,1)

nthExponent: (% , Integer) -> Integer
++ nthExponent(u,n) returns the exponent of the nth factor of
++ \spadvar{u}. If \spadvar{n} is not a valid index for a factor
++ (for example, less than 1 or too big), 0 is returned.
++
++X a:=factor 9720000
++X nthExponent(a,2)

nthFactor: (% ,Integer) -> R
++ nthFactor(u,n) returns the base of the nth factor of
++ \spadvar{u}. If \spadvar{n} is not a valid index for a factor
++ (for example, less than 1 or too big), 1 is returned. If
++ \spadvar{u} consists only of a unit, the unit is returned.
++
++X a:=factor 9720000
++X nthFactor(a,2)

nthFlag: (% ,Integer) -> fUnion
++ nthFlag(u,n) returns the information flag of the nth factor of
++ \spadvar{u}. If \spadvar{n} is not a valid index for a factor
++ (for example, less than 1 or too big), "nil" is returned.
++
++X a:=factor 9720000
++X nthFlag(a,2)

numberOfFactors : % -> NonNegativeInteger
++ numberOfFactors(u) returns the number of factors in \spadvar{u}.
++
++X a:=factor 9720000
++X numberOfFactors a

primeFactor: (R,Integer) -> %
++ primeFactor(base,exponent) creates a factored object with

```

```

++ a single factor whose base is asserted to be prime
++ (flag = "prime").
++
++X a:=primeFactor(3,4)
++X nthFlag(a,1)

sqfrFactor: (R,Integer) -> %
++ sqfrFactor(base,exponent) creates a factored object with
++ a single factor whose base is asserted to be square-free
++ (flag = "sqfr").
++
++X a:=sqfrFactor(3,5)
++X nthFlag(a,1)

flagFactor: (R,Integer, fUnion) -> %
++ flagFactor(base,exponent,flag) creates a factored object with
++ a single factor whose base is asserted to be properly
++ described by the information flag.

unit: % -> R
++ unit(u) extracts the unit part of the factorization.
++
++X f:=x*y^3-3*x^2*y^2+3*x^3*y-x^4
++X unit f
++X g:=makeFR(z,factorList f)
++X unit g

unitNormalize: % -> %
++ unitNormalize(u) normalizes the unit part of the factorization.
++ For example, when working with factored integers, this operation will
++ ensure that the bases are all positive integers.

map: (R -> R, %) -> %
++ map(fn,u) maps the function \userfun{fn} across the factors of
++ \spadvar{u} and creates a new factored object. Note: this clears
++ the information flags (sets them to "nil") because the effect of
++ \userfun{fn} is clearly not known in general.
++
++X m(a:Factored Polynomial Integer):Factored Polynomial Integer == a^2
++X f:=x*y^3-3*x^2*y^2+3*x^3*y-x^4
++X map(m,f)
++X g:=makeFR(z,factorList f)
++X map(m,g)

-- the following operations are conditional on R

if R has GcdDomain then GcdDomain
if R has RealConstant then RealConstant
if R has UniqueFactorizationDomain then UniqueFactorizationDomain

```



```

if R has ConvertibleTo InputForm then ConvertibleTo InputForm

if R has IntegerNumberSystem then
  rational?      : % -> Boolean
  ++ rational?(u) tests if \spadvar{u} is actually a
  ++ rational number (see \spadtype{Fraction Integer}).
  rational       : % -> Fraction Integer
  ++ rational(u) assumes spadvar{u} is actually a rational number
  ++ and does the conversion to rational number
  ++ (see \spadtype{Fraction Integer}).
  rationalIfCan : % -> Union(Fraction Integer, "failed")
  ++ rationalIfCan(u) returns a rational number if u
  ++ really is one, and "failed" otherwise.

if R has Eltable(%, %) then Eltable(%, %)
if R has Evaluable(%) then Evaluable(%)
if R has InnerEvaluable(Symbol, %) then InnerEvaluable(Symbol, %)

Implementation ==> add

-- Representation:
-- Note: exponents are allowed to be integers so that some special cases
-- may be used in simplifications
Rep := Record(unt:R, fct:List FF)

if R has ConvertibleTo InputForm then
  convert(x:%):InputForm ==
    empty?(lf := reverse factorList x) => convert(unit x)@InputForm
    l := empty()$List(InputForm)
    for rec in lf repeat
--      one?(rec.fctr) => l
      ((rec.fctr) = 1) => l
      iFactor : InputForm := binary( convert("::" :: Symbol)@InputForm, [convert(rec.fctr)@I
      iExpon  : InputForm := convert(rec.xpnt)@InputForm
      iFun    : List InputForm :=
        rec.flg case "nil" =>
          [convert("nilFactor" :: Symbol)@InputForm, iFactor, iExpon]$List(InputForm)
        rec.flg case "sqfr" =>
          [convert("sqfrFactor" :: Symbol)@InputForm, iFactor, iExpon]$List(InputForm)
        rec.flg case "prime" =>
          [convert("primeFactor" :: Symbol)@InputForm, iFactor, iExpon]$List(InputForm)
        rec.flg case "irred" =>
          [convert("irreducibleFactor" :: Symbol)@InputForm, iFactor, iExpon]$List(InputForm)
        nil$List(InputForm)
      l := concat( iFun pretend InputForm, l )
--      one?(rec.xpnt) =>
--        l := concat(convert(rec.fctr)@InputForm, l)
--      l := concat(convert(rec.fctr)@InputForm ** rec.xpnt, l)
    empty? l => convert(unit x)@InputForm
    if unit x ^= 1 then l := concat(convert(unit x)@InputForm,l)

```

```

empty? rest l => first l
binary(convert(_*::Symbol)@InputForm, l)@InputForm

orderedR? := R has OrderedSet

-- Private function signatures:
reciprocal      : % -> %
qexpand        : % -> R
negexp?        : % -> Boolean
SimplifyFactorization : List FF -> List FF
LispLessP      : (FF, FF) -> Boolean
mkFF           : (R, List FF) -> %
SimplifyFactorization1 : (FF, List FF) -> List FF
stricterFlag   : (fUnion, fUnion) -> fUnion

nilFactor(r, i)      == flagFactor(r, i, "nil")
sqfrFactor(r, i)     == flagFactor(r, i, "sqfr")
irreducibleFactor(r, i) == flagFactor(r, i, "irred")
primeFactor(r, i)    == flagFactor(r, i, "prime")
unit? u              == (empty? u.fct) and (not zero? u.unt)
factorList u         == u.fct
unit u               == u.unt
numberOfFactors u    == # u.fct
0                    == [1, [{"nil", 0, 1}$FF]]
zero? u              == # u.fct = 1 and
                      (first u.fct).flg case "nil" and
                      zero? (first u.fct).fctr and
                      one? u.unt
--                      (u.unt = 1)
1                    == [1, empty()]
one? u               == empty? u.fct and u.unt = 1
mkFF(r, x)           == [r, x]
coerce(j:Integer):%  == (j::R)::%
characteristic()     == characteristic()$R
i:Integer * u:%      == (i :: %) * u
r:R * u:%            == (r :: %) * u
factors u            == [[fe.fctr, fe.xpnt] for fe in factorList u]
expand u             == retract u
negexp? x            == "or"/[negative?(y.xpnt) for y in factorList x]

makeFR(u, l) ==
-- normalizing code to be installed when contents are handled better
-- current squareFree returns the content as a unit part.
--   if (not unit?(u)) then
--       l := cons(["nil", u, 1]$FF, l)
--       u := 1
--   unitNormalize mkFF(u, SimplifyFactorization l)

if R has IntegerNumberSystem then
  rational? x == true

```

```

rationalIfCan x == rational x

rational x ==
  convert(unit x)@Integer *
    _*/[(convert(f.fctr)@Integer)::Fraction(Integer)
        ** f.xpnt for f in factorList x]

if R has Eltable(R, R) then
  elt(x:%, v:%) == x(expand v)

if R has Evaluable(R) then
  eval(x:%, l:List Equation %) ==
    eval(x, [expand lhs e = expand rhs e for e in l]$List(Equation R))

if R has InnerEvaluable(Symbol, R) then
  eval(x:%, ls:List Symbol, lv:List %) ==
    eval(x, ls, [expand v for v in lv]$List(R))

if R has RealConstant then
--! negcount and rest commented out since RealConstant doesn't support
--! positive? or negative?
-- negcount: % -> Integer
-- positive?(x:%):Boolean == not(zero? x) and even?(negcount x)
-- negative?(x:%):Boolean == not(zero? x) and odd?(negcount x)
-- negcount x ==
--   n := count(negative?(#1.fctr), factorList x)$List(FF)
--   negative? unit x => n + 1
--   n

convert(x:%):Float ==
  convert(unit x)@Float *
    _*/[convert(f.fctr)@Float ** f.xpnt for f in factorList x]

convert(x:%):DoubleFloat ==
  convert(unit x)@DoubleFloat *
    _*/[convert(f.fctr)@DoubleFloat ** f.xpnt for f in factorList x]

u:% * v:% ==
  zero? u or zero? v => 0
--   one? u => v
  (u = 1) => v
--   one? v => u
  (v = 1) => u
  mkFF(unit u * unit v,
    SimplifyFactorization concat(factorList u, copy factorList v))

u:% ** n:NonNegativeInteger ==
  mkFF(unit(u)**n, [[x.flg, x.fctr, n * x.xpnt] for x in factorList u])

SimplifyFactorization x ==

```

```

empty? x => empty()
x := sort_!(LispLessP, x)
x := SimplifyFactorization1(first x, rest x)
if orderedR? then x := sort_!(LispLessP, x)
x

SimplifyFactorization1(f, x) ==
  empty? x =>
    zero?(f.xpnt) => empty()
    list f
  f1 := first x
  f.fctr = f1.fctr =>
    SimplifyFactorization1([stricterFlag(f.flg, f1.flg),
                          f.fctr, f.xpnt + f1.xpnt], rest x)
  l := SimplifyFactorization1(first x, rest x)
  zero?(f.xpnt) => l
  concat(f, l)

coerce(x:%):OutputForm ==
  empty?(lf := reverse factorList x) => (unit x)::OutputForm
  l := empty()$List(OutputForm)
  for rec in lf repeat
--    one?(rec.fctr) => l
    ((rec.fctr) = 1) => l
--    one?(rec.xpnt) =>
    ((rec.xpnt) = 1) =>
      l := concat(rec.fctr :: OutputForm, l)
      l := concat(rec.fctr::OutputForm ** rec.xpnt::OutputForm, l)
  empty? l => (unit x) :: OutputForm
  e :=
    empty? rest l => first l
    reduce(*, l)
  l = unit x => e
  (unit x)::OutputForm * e

retract(u:%):R ==
  negexp? u => error "Negative exponent in factored object"
  qexpand u

qexpand u ==
  unit u *
    _*/[y.fctr ** (y.xpnt::NonNegativeInteger) for y in factorList u]

retractIfCan(u:%):Union(R, "failed") ==
  negexp? u => "failed"
  qexpand u

LispLessP(y, y1) ==
  orderedR? => y.fctr < y1.fctr

```

```

GGGREATERP(y.fctr, y1.fctr)$Lisp => false
true

stricterFlag(fl1, fl2) ==
  fl1 case "prime"    => fl1
  fl1 case "irred"    =>
    fl2 case "prime" => fl2
    fl1
  fl1 case "sqfr"     =>
    fl2 case "nil"   => fl1
    fl2
  fl2

if R has IntegerNumberSystem
then
  coerce(r:R):% ==
    factor(r)$IntegerFactorizationPackage(R) pretend %
else
  if R has UniqueFactorizationDomain
  then
    coerce(r:R):% ==
      zero? r => 0
      unit? r => mkFF(r, empty())
      unitNormalize(squareFree(r) pretend %)
  else
    coerce(r:R):% ==
--      one? r => 1
      (r = 1) => 1
      unitNormalize mkFF(1, [{"nil", r, 1}$FF])

u = v ==
  (unit u = unit v) and # u.fct = # v.fct and
  set(factors u)$SRFE =$SRFE set(factors v)$SRFE

- u ==
  zero? u => u
  mkFF(- unit u, factorList u)

recip u ==
  not empty? factorList u => "failed"
  (r := recip unit u) case "failed" => "failed"
  mkFF(r::R, empty())

reciprocal u ==
  mkFF((recip unit u)::R,
    [[y.flg, y.fctr, - y.xpnt]$FF for y in factorList u])

exponent u == -- exponent of first factor
  empty?(fl := factorList u) or zero? u => 0
  first(fl).xpnt

```

```

nthExponent(u, i) ==
  l := factorList u
  zero? u or i < 1 or i > #l => 0
  (l.(minIndex(l) + i - 1)).xpnt

nthFactor(u, i) ==
  zero? u => 0
  zero? i => unit u
  l := factorList u
  negative? i or i > #l => 1
  (l.(minIndex(l) + i - 1)).fctr

nthFlag(u, i) ==
  l := factorList u
  zero? u or i < 1 or i > #l => "nil"
  (l.(minIndex(l) + i - 1)).flg

flagFactor(r, i, fl) ==
  zero? i => 1
  zero? r => 0
  unitNormalize mkFF(1, [[fl, r, i]$FF])

differentiate(u:%, deriv: R -> R) ==
  ans := deriv(unit u) * ((u exquo unit(u)::%)::%)
  ans + (_+/[fact.xpnt * deriv(fact.fctr) *
    ((u exquo nilFactor(fact.fctr, 1))::%) for fact in factorList u])

map(fn, u) ==
  fn(unit u) * _*/[irreducibleFactor(fn(f.fctr),f.xpnt) for f in factorList u]

u exquo v ==
  empty?(x1 := factorList v) => unitNormal(retract v).associate * u
  empty? factorList u => "failed"
  v1 := u * reciprocal v
  goodQuotient:Boolean := true
  while (goodQuotient and (not empty? x1)) repeat
    if x1.first.xpnt < 0
      then goodQuotient := false
    else x1 := rest x1
  goodQuotient => v1
  "failed"

unitNormal u == -- does a bunch of work, but more canonical
  (ur := recip(un := unit u)) case "failed" => [1, u, 1]
  as := ur::R
  v1 := empty()$List(FF)
  for x in factorList u repeat
    ucar := unitNormal(x.fctr)
    e := abs(x.xpnt)::NonNegativeInteger

```

```

if x.xpnt < 0
then -- associate is recip of unit
  un := un * (ucar.associate ** e)
  as := as * (ucar.unit ** e)
else
  un := un * (ucar.unit ** e)
  as := as * (ucar.associate ** e)
--
if not one?(ucar.canonical) then
if not ((ucar.canonical) = 1) then
  v1 := concat([x.flg, ucar.canonical, x.xpnt], v1)
[mkFF(un, empty()), mkFF(1, reverse_! v1), mkFF(as, empty())]

unitNormalize u ==
  uca := unitNormal u
  mkFF(unit(uca.unit)*unit(uca.canonical),factorList(uca.canonical))

if R has GcdDomain then
  u + v ==
    zero? u => v
    zero? v => u
    v1 := reciprocal(u1 := gcd(u, v))
    (expand(u * v1) + expand(v * v1)) * u1

gcd(u, v) ==
--
  one? u or one? v => 1
  (u = 1) or (v = 1) => 1
  zero? u => v
  zero? v => u
  f1 := empty()$List(Integer) -- list of used factor indices in x
  f2 := f1 -- list of indices corresponding to a given factor
  f3 := empty()$List(List Integer) -- list of f2-like lists
  x := concat(factorList u, factorList v)
  for i in minIndex x .. maxIndex x repeat
    if not member?(i, f1) then
      f1 := concat(i, f1)
      f2 := [i]
      for j in i+1..maxIndex x repeat
        if x.i.fctr = x.j.fctr then
          f1 := concat(j, f1)
          f2 := concat(j, f2)
      f3 := concat(f2, f3)
  x1 := empty()$List(FF)
  while not empty? f3 repeat
    f1 := first f3
    if #f1 > 1 then
      i := first f1
      y := copy x.i
      f1 := rest f1
      while not empty? f1 repeat
        i := first f1

```

```

        if x.i.xpnt < y.xpnt then y.xpnt := x.i.xpnt
        f1 := rest f1
        x1 := concat(y, x1)
        f3 := rest f3
        if orderedR? then x1 := sort_!(LispLessP, x1)
        mkFF(1, x1)

    else -- R not a GCD domain
        u + v ==
        zero? u => v
        zero? v => u
        irreducibleFactor(expand u + expand v, 1)

    if R has UniqueFactorizationDomain then
        prime? u ==
        not(empty?(l := factorList u)) and (empty? rest l) and
        one?(l.first.xpnt) and (l.first.flg case "prime")
--      ((l.first.xpnt) = 1) and (l.first.flg case "prime")

```

— FR.dotabb —

```

"FR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FR"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"FR" -> "ALIST"

```

domain FILE File

— File.input —

```

)set break resume
)sys rm -f File.output
)spool File.output
)set message test on
)set message auto off
)clear all
--S 1 of 12
ifile:File List Integer:=open("jazz1","output")
--R
--R
--R (1) "jazz1"

```


[illegible]

```

--S 8 of 12
readIfCan! ifile
--R
--R
--R (8) [7]
--R
--R                                         Type: Union(List(Integer),...)
--E 8

--S 9 of 12
readIfCan! ifile
--R
--R
--R (9) "failed"
--R
--R                                         Type: Union("failed",...)
--E 9

--S 10 of 12
iomode ifile
--R
--R
--R (10) "input"
--R
--R                                         Type: String
--E 10

--S 11 of 12
name ifile
--R
--R
--R (11) "jazz1"
--R
--R                                         Type: FileName
--E 11

--S 12 of 12
close! ifile
--R
--R
--R (12) "jazz1"
--R
--R                                         Type: File(List(Integer))
--E 12
)system rm jazz1
)spool
)lisp (bye)

```

— File.help —

=====

File examples

=====

The File(S) domain provides a basic interface to read and write values of type S in files.

Before working with a file, it must be made accessible to Axiom with the open operation.

```
ifile:File List Integer:=open("/tmp/jazz1","output")
    "jazz1"
                                Type: File List Integer
```

The open function arguments are a FileNam} and a String specifying the mode. If a full pathname is not specified, the current default directory is assumed. The mode must be one of "input" or "output". If it is not specified, "input" is assumed. Once the file has been opened, you can read or write data.

The operations read and write are provided.

```
write!(ifile, [-1,2,3])
    [- 1,2,3]
                                Type: List Integer
```

```
write!(ifile, [10,-10,0,111])
    [10,- 10,0,111]
                                Type: List Integer
```

```
write!(ifile, [7])
    [7]
                                Type: List Integer
```

You can change from writing to reading (or vice versa) by reopening a file.

```
reopen!(ifile, "input")
    "jazz1"
                                Type: File List Integer
```

```
read! ifile
    [- 1,2,3]
                                Type: List Integer
```

```
read! ifile
    [10,- 10,0,111]
                                Type: List Integer
```

The read operation can cause an error if one tries to read more data than is in the file. To guard against this possibility the readIfCan operation should be used.

```
readIfCan! ifile
[7]
Type: Union(List Integer,...)
```

```
readIfCan! ifile
"failed"
Type: Union("failed",...)
```

You can find the current mode of the file, and the file's name.

```
iomode ifile
"input"
Type: String
```

```
name ifile
"jazz1"
Type: FileName
```

When you are finished with a file, you should close it.

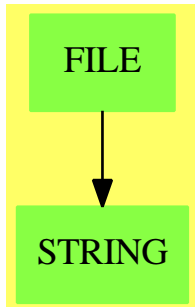
```
close! ifile
"jazz1"
Type: File List Integer
```

```
)system rm /tmp/jazz1
```

A limitation of the underlying LISP system is that not all values can be represented in a file. In particular, delayed values containing compiled functions cannot be saved.

See Also:

- o)help TextFile
- o)help KeyedAccessFile
- o)help Library
- o)help Filename
- o)show File

File (FILE)**See**

- ⇒ “TextFile” (TEXTFILE) [21](#) on page [2692](#)
- ⇒ “BinaryFile” (BINFILE) [3](#) on page [295](#)
- ⇒ “KeyedAccessFile” (KAFfile) [12](#) on page [1413](#)
- ⇒ “Library” (LIB) [13](#) on page [1428](#)

Exports:

close!	coerce	flush	hash	iomode
latex	name	open	readIfCan!	read!
reopen!	write!	?=?	?~=?	

— domain **FILE** File —

```

)abbrev domain FILE File
++ Author: Stephen M. Watt, Victor Miller
++ Date Created: 1984
++ Date Last Updated: June 4, 1991
++ Description:
++ This domain provides a basic model of files to save arbitrary values.
++ The operations provide sequential access to the contents.

File(S:SetCategory): FileCategory(FileName, S) with
  readIfCan_!: % -> Union(S, "failed")
    ++ readIfCan!(f) returns a value from the file f, if possible.
    ++ If f is not open for reading, or if f is at the end of file
    ++ then \spad{"failed"} is the result.
== add
  FileState ==> SExpression
  IOMode    ==> String

  Rep:=Record(fileName:  FileName,  _
               fileState:  FileState, _
               fileIOMode: IOMode)

  defstream(fn: FileName, mode: IOMode): FileState ==
    mode = "input" =>

```

```

        not readable? fn => error ["File is not readable", fn]
        MAKE_-INSTREAM(fn::String)$Lisp
    mode = "output" =>
        not writable? fn => error ["File is not writable", fn]
        MAKE_-OUTSTREAM(fn::String)$Lisp
    error ["IO mode must be input or output", mode]

f1 = f2 ==
    f1.fileName = f2.fileName
coerce(f: %): OutputForm ==
    f.fileName::OutputForm

open fname ==
    open(fname, "input")
open(fname, mode) ==
    fstream := defstream(fname, mode)
    [fname, fstream, mode]
reopen_!(f, mode) ==
    fname := f.fileName
    f.fileState := defstream(fname, mode)
    f.fileIOmode := mode
    f
close_! f ==
    SHUT(f.fileState)$Lisp
    f.fileIOmode := "closed"
    f
name f ==
    f.fileName
iomode f ==
    f.fileIOmode
read_! f ==
    f.fileIOmode ^= "input" =>
        error "File not in read state"
    x := VMREAD(f.fileState)$Lisp
    PLACEP(x)$Lisp =>
        error "End of file"
    x
readIfCan_! f ==
    f.fileIOmode ^= "input" =>
        error "File not in read state"
    x: S := VMREAD(f.fileState)$Lisp
    PLACEP(x)$Lisp => "failed"
    x
write_!(f, x) ==
    f.fileIOmode ^= "output" =>
        error "File not in write state"
    z := PRINT_-FULL(x, f.fileState)$Lisp
    TERPRI(f.fileState)$Lisp
    x

```

```
flush f ==
  f.fileIOmode ^= "output" => error "File not in write state"
  FORCE_-OUTPUT(f.fileState)$Lisp
```

— FILE.dotabb —

```
"FILE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FILE"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"FILE" -> "STRING"
```

domain FNAME FileName

— FileName.input —

```
)set break resume
)sys rm -f FileName.output
)spool FileName.output
)set message test on
)set message auto off
)clear all
--S 1 of 18
fn: FileName
--R
--R
--E 1
Type: Void

--S 2 of 18
fn := "fname.input"
--R
--R
--R (2) "fname.input"
--R
--E 2
Type: FileName

--S 3 of 18
directory fn
--R
--R
--R (3) ""
--R
Type: String
```

--E 3

--S 4 of 18

name fn

--R

--R

--R (4) "fname"

--R

Type: String

--E 4

--S 5 of 18

extension fn

--R

--R

--R (5) "input"

--R

Type: String

--E 5

--S 6 of 18

fn := filename("/tmp", "fname", "input")

--R

--R

--R (6) "/tmp/fname.input"

--R

Type: FileName

--E 6

--S 7 of 18

objdir := "/tmp"

--R

--R

--R (7) "/tmp"

--R

Type: String

--E 7

--S 8 of 18

fn := filename(objdir, "table", "spad")

--R

--R

--R (8) "/tmp/table.spad"

--R

Type: FileName

--E 8

--S 9 of 18

fn := filename("", "letter", "")

--R

--R

--R (9) "letter"

--R

Type: FileName

--E 9


```
--S 10 of 18
exists? "/etc/passwd"
--R
--R
--R (10) true
--R
--R                                          Type: Boolean
--E 10

--S 11 of 18
readable? "/etc/passwd"
--R
--R
--R (11) true
--R
--R                                          Type: Boolean
--E 11

--S 12 of 18
readable? "/etc/security/passwd"
--R
--R
--R (12) false
--R
--R                                          Type: Boolean
--E 12

--S 13 of 18
readable? "/ect/passwd"
--R
--R
--R (13) false
--R
--R                                          Type: Boolean
--E 13

--S 14 of 18
writable? "/etc/passwd"
--R
--R
--R (14) true
--R
--R                                          Type: Boolean
--E 14

--S 15 of 18
writable? "/dev/null"
--R
--R
--R (15) true
--R
--R                                          Type: Boolean
--E 15

--S 16 of 18
writable? "/etc/DoesNotExist"
```

```

--R
--R
--R (16) true
--R
--R                                          Type: Boolean
--E 16

--S 17 of 18
writable? "/tmp/DoesNotExist"
--R
--R
--R (17) true
--R
--R                                          Type: Boolean
--E 17

--S 18 of 18
fn := new(objdir, "xxx", "yy")
--R
--R
--R (18) "/tmp/xxx1419.yy"
--R
--R                                          Type: FileName
--E 18
)spool
)lisp (bye)

```

— **FileName.help** —

=====
 FileName examples
 =====

The FileName domain provides an interface to the computer's file system. Functions are provided to manipulate file names and to test properties of files.

The simplest way to use file names in the Axiom interpreter is to rely on conversion to and from strings. The syntax of these strings depends on the operating system.

```

fn: FileName
                                Type: Void

```

On Linux, this is a proper file syntax:

```

fn := "fname.input"
    "fname.input"
                                Type: FileName

```

Although it is very convenient to be able to use string notation for file names in the interpreter, it is desirable to have a portable way of creating and manipulating file names from within programs.

A measure of portability is obtained by considering a file name to consist of three parts: the directory, the name, and the extension.

```

directory fn
    ""
                                Type: String

name fn
    "fname"
                                Type: String

extension fn
    "input"
                                Type: String

```

The meaning of these three parts depends on the operating system. For example, on CMS the file "SPADPROF INPUT M" would have directory "M", name "SPADPROF" and extension "INPUT".

It is possible to create a filename from its parts.

```

fn := filename("/tmp", "fname", "input")
    "/tmp/fname.input"
                                Type: FileName

```

When writing programs, it is helpful to refer to directories via variables.

```

objdir := "/tmp"
    "/tmp"
                                Type: String

fn := filename(objdir, "table", "spad")
    "/tmp/table.spad"
                                Type: FileName

```

If the directory or the extension is given as an empty string, then a default is used. On AIX, the defaults are the current directory and no extension.

```

fn := filename("", "letter", "")
    "letter"
                                Type: FileName

```

Three tests provide information about names in the file system.

The exists? operation tests whether the named file exists.

```
exists? "/etc/passwd"
true
Type: Boolean
```

The operation readable? tells whether the named file can be read. If the file does not exist, then it cannot be read.

```
readable? "/etc/passwd"
true
Type: Boolean
```

```
readable? "/etc/security/passwd"
false
Type: Boolean
```

```
readable? "/ect/passwd"
false
Type: Boolean
```

Likewise, the operation writable? tells whether the named file can be written. If the file does not exist, the test is determined by the properties of the directory.

```
writable? "/etc/passwd"
true
Type: Boolean
```

```
writable? "/dev/null"
true
Type: Boolean
```

```
writable? "/etc/DoesNotExist"
true
Type: Boolean
```

```
writable? "/tmp/DoesNotExist"
true
Type: Boolean
```

The new operation constructs the name of a new writable file. The argument sequence is the same as for filename, except that the name part is actually a prefix for a constructed unique name.

The resulting file is in the specified directory with the given extension, and the same defaults are used.

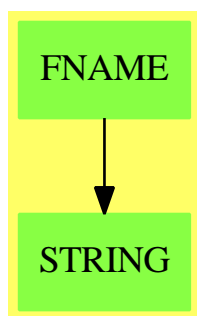
```
fn := new(objdir, "xxx", "yy")
"/tmp/xxx1419.yy"
```

Type: FileName

See Also:

o)show FileName

FileName (FNAME)



Exports:

coerce	directory	exists?	extension	filename
hash	latex	name	new	readable?
writable?	?=?	?~=?		

— domain FNAME FileName —

```

)abbrev domain FNAME FileName
++ Author: Stephen M. Watt
++ Date Created: 1985
++ Date Last Updated: June 20, 1991
++ Description:
++ This domain provides an interface to names in the file system.
  
```

```

FileName(): FileNameCategory == add
  
```

```

    f1 = f2                                == EQUAL(f1, f2)$Lisp
    coerce(f: %): OutputForm == f::String::OutputForm

    coerce(f: %): String == NAMESTRING(f)$Lisp
    coerce(s: String): % == PARSE_-NAMESTRING(s)$Lisp

    filename(d,n,e) == fnameMake(d,n,e)$Lisp

    directory(f:%): String == fnameDirectory(f)$Lisp
  
```

```

name(f: %): String      == fnameName(f)$Lisp
extension(f: %): String == fnameType(f)$Lisp

exists? f                == fnameExists?(f)$Lisp
readable? f              == fnameReadable?(f)$Lisp
writable? f              == fnameWritable?(f)$Lisp

new(d, pref, e)          == fnameNew(d, pref, e)$Lisp

```

— FNAME.dotabb —

```

"FNAME" [color="#88FF44", href="bookvol10.3.pdf#nameddest=FNAME"]
"STRING" [color="#88FF44", href="bookvol10.3.pdf#nameddest=STRING"]
"FNAME" -> "STRING"

```

domain FDIV FiniteDivisor

— FiniteDivisor.input —

```

)set break resume
)sys rm -f FiniteDivisor.output
)spool FiniteDivisor.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteDivisor
--R
--R FiniteDivisor(F: Field, UP: UnivariatePolynomialCategory(F), UPUP: UnivariatePolynomialCategory(Fraction(F)))
--R Abbreviation for FiniteDivisor is FDIV
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FDIV
--R
--R----- Operations -----
--R ?? : (Integer, %) -> %          ?? : (PositiveInteger, %) -> %
--R +? : (% , %) -> %             ?- : (% , %) -> %
--R -? : % -> %                   ==? : (% , %) -> Boolean
--R 0 : () -> %                   coerce : % -> OutputForm
--R divisor : (R, UP, UP, UP, F) -> %   divisor : (F, F, Integer) -> %

```

```

--R divisor : (F,F) -> %
--R finiteBasis : % -> Vector(R)
--R lSpaceBasis : % -> Vector(R)
--R principal? : % -> Boolean
--R sample : () -> %
--R ~=? : (%,% ) -> Boolean
--R ?? : (NonNegativeInteger,% ) -> %
--R decompose : % -> Record(id: FractionalIdeal(UP,Fraction(UP),UPUP,R),principalPart: R)
--R divisor : FractionalIdeal(UP,Fraction(UP),UPUP,R) -> %
--R generator : % -> Union(R,"failed")
--R ideal : % -> FractionalIdeal(UP,Fraction(UP),UPUP,R)
--R subtractIfCan : (%,% ) -> Union(%,"failed")
--R
--E 1

)spool
)lisp (bye)

```

— FiniteDivisor.help —

=====

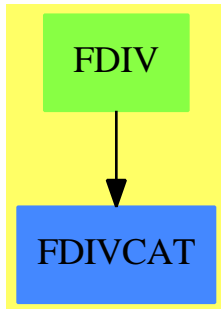
FiniteDivisor examples

=====

This domains implements finite rational divisors on a curve, that is finite formal sums $\sum(n * P)$ where the n 's are integers and the P 's are finite rational points on the curve.

See Also:

o)show FiniteDivisor

FiniteDivisor (FDIV)**See**

- ⇒ “FractionalIdeal” (FRIDEAL) [7](#) on page [994](#)
- ⇒ “FramedModule” (FRMOD) [7](#) on page [1000](#)
- ⇒ “HyperellipticFiniteDivisor” (HELLFDIV) [9](#) on page [1183](#)

Exports:

0	coerce	decompose	divisor	finiteBasis
generator	hash	ideal	lSpaceBasis	latex
principal?	reduce	sample	subtractIfCan	zero?
?~=?	?*?	?+?	?-?	-?
?=?				

— domain **FDIV** FiniteDivisor —

```

)abbrev domain FDIV FiniteDivisor
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 29 July 1993
++ Description:
++ This domains implements finite rational divisors on a curve, that
++ is finite formal sums SUM(n * P) where the n's are integers and the
++ P's are finite rational points on the curve.

```

```

FiniteDivisor(F, UP, UPUP, R): Exports == Implementation where
  F   : Field
  UP  : UnivariatePolynomialCategory F
  UPUP: UnivariatePolynomialCategory Fraction UP
  R   : FunctionFieldCategory(F, UP, UPUP)

```

```

N ==> NonNegativeInteger
RF ==> Fraction UP
ID ==> FractionalIdeal(UP, RF, UPUP, R)

```

```

Exports ==> FiniteDivisorCategory(F, UP, UPUP, R) with
  finiteBasis: % -> Vector R
  ++ finiteBasis(d) returns a basis for d as a module over K[x].

```



```

lSpaceBasis: % -> Vector R
++ lSpaceBasis(d) returns a basis for \spad{L(d) = {f | (f) >= -d}}
++ as a module over \spad{K[x]}.

Implementation ==> add
if hyperelliptic()$R case UP then
  Rep := HyperellipticFiniteDivisor(F, UP, UPUP, R)

  0 == 0$Rep
  coerce(d:$):OutputForm == coerce(d)$Rep
  d1 = d2 == d1 =$Rep d2
  n * d == n *$Rep d
  d1 + d2 == d1 +$Rep d2
  - d == -$Rep d
  ideal d == ideal(d)$Rep
  reduce d == reduce(d)$Rep
  generator d == generator(d)$Rep
  decompose d == decompose(d)$Rep
  divisor(i:ID) == divisor(i)$Rep
  divisor(f:R) == divisor(f)$Rep
  divisor(a, b) == divisor(a, b)$Rep
  divisor(a, b, n) == divisor(a, b, n)$Rep
  divisor(h, d, dp, g, r) == divisor(h, d, dp, g, r)$Rep
else
  Rep := Record(id:ID, fbasis:Vector(R))

  import CommonDenominator(UP, RF, Vector RF)
  import UnivariatePolynomialCommonDenominator(UP, RF, UPUP)

  makeDivisor : (UP, UPUP, UP) -> %
  intReduce : (R, UP) -> R

  ww := integralBasis()$R

  0 == [1, empty()]
  divisor(i:ID) == [i, empty()]
  divisor(f:R) == divisor ideal [f]
  coerce(d:%):OutputForm == ideal(d)::OutputForm
  ideal d == d.id
  decompose d == [ideal d, 1]
  d1 = d2 == basis(ideal d1) = basis(ideal d2)
  n * d == divisor(ideal(d) ** n)
  d1 + d2 == divisor(ideal d1 * ideal d2)
  - d == divisor inv ideal d
  divisor(h, d, dp, g, r) == makeDivisor(d, lift h - (r * dp)::RF::UPUP, g)

  intReduce(h, b) ==
    v := integralCoordinates(h).num
    integralRepresents(

```

```

[qelt(v, i) rem b for i in minIndex v .. maxIndex v], 1)

divisor(a, b) ==
  x := monomial(1, 1)$UP
  not ground? gcd(d := x - a::UP, retract(discriminant())@UP) =>
    error "divisor: point is singular"
  makeDivisor(d, monomial(1, 1)$UPUP - b::UP::RF::UPUP, 1)

divisor(a, b, n) ==
  not(ground? gcd(d := monomial(1, 1)$UP - a::UP,
    retract(discriminant())@UP)) and
    ((n exquo rank()) case "failed") =>
      error "divisor: point is singular"
  m:N :=
    n < 0 => (-n)::N
  n::N
  g := makeDivisor(d**m, (monomial(1, 1)$UPUP - b::UP::RF::UPUP)**m, 1)
  n < 0 => -g
  g

reduce d ==
  (i := minimize(j := ideal d)) = j => d
  #(n := numer i) ^= 2 => divisor i
  cd := splitDenominator lift n(1 + minIndex n)
  b := gcd(cd.den * retract(retract(n minIndex n)@RF)@UP,
    retract(norm reduce(cd.num))@UP)
  e := cd.den * denom i
  divisor ideal([(b / e)::R,
    reduce map((s:RF):RF+>(retract(s)@UP rem b)/e, cd.num)]$Vector(R))

finiteBasis d ==
  if empty?(d.fbasis) then
    d.fbasis := normalizeAtInfinity
      basis module(ideal d)$FramedModule(UP, RF, UPUP, R, ww)
  d.fbasis

generator d ==
  bsis := finiteBasis d
  for i in minIndex bsis .. maxIndex bsis repeat
    integralAtInfinity? qelt(bsis, i) =>
      return primitivePart qelt(bsis, i)
  "failed"

lSpaceBasis d ==
  map_!(primitivePart, reduceBasisAtInfinity finiteBasis(-d))

-- b = center, hh = integral function, g = gcd(b, discriminant)
makeDivisor(b, hh, g) ==
  b := gcd(b, retract(norm(h := reduce hh))@UP)
  h := intReduce(h, b)

```

```

if not ground? gcd(g, b) then h := intReduce(h ** rank(), b)
divisor ideal [b::RF::R, h]$Vector(R)

```

— FDIV.dotabb —

```

"FDIV" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FDIV"]
"FDIVCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FDIVCAT"]
"FDIV" -> "FDIVCAT"

```

domain FF FiniteField

— FiniteField.input —

```

)set break resume
)sys rm -f FiniteField.output
)spool FiniteField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteField
--R FiniteField(p: PositiveInteger,n: PositiveInteger) is a domain constructor
--R Abbreviation for FiniteField is FF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FF
--R
--R----- Operations -----
--R ??? : (PrimeField(p),%) -> %          ??? : (%,PrimeField(p)) -> %
--R ??? : (Fraction(Integer),%) -> %      ??? : (%,Fraction(Integer)) -> %
--R ??? : (%,%) -> %                     ??? : (Integer,%) -> %
--R ??? : (PositiveInteger,%) -> %        ??? : (%,Integer) -> %
--R ??? : (%,PositiveInteger) -> %       ?+? : (%,%) -> %
--R ?-? : (%,%) -> %                     -? : % -> %
--R ?/? : (%,PrimeField(p)) -> %         ?/? : (%,%) -> %
--R ?=? : (%,%) -> Boolean                1 : () -> %
--R 0 : () -> %                           ??? : (%,Integer) -> %
--R ??? : (%,PositiveInteger) -> %       algebraic? : % -> Boolean
--R associates? : (%,%) -> Boolean        basis : () -> Vector(%)
--R coerce : PrimeField(p) -> %          coerce : Fraction(Integer) -> %

```

```

--R coerce : % -> %
--R coerce : % -> OutputForm
--R dimension : () -> CardinalNumber
--R gcd : List(%) -> %
--R hash : % -> SingleInteger
--R inv : % -> %
--R lcm : List(%) -> %
--R norm : % -> PrimeField(p)
--R prime? : % -> Boolean
--R recip : % -> Union(%, "failed")
--R retract : % -> PrimeField(p)
--R sizeLess? : (%, %) -> Boolean
--R squareFreePart : % -> %
--R transcendental? : % -> Boolean
--R unitCanonical : % -> %
--R ?~=? : (%, %) -> Boolean
--R ?*? : (NonNegativeInteger, %) -> %
--R ?**? : (%, NonNegativeInteger) -> %
--R D : (%, NonNegativeInteger) -> % if PrimeField(p) has FINITE
--R D : % -> % if PrimeField(p) has FINITE
--R Frobenius : (%, NonNegativeInteger) -> % if PrimeField(p) has FINITE
--R Frobenius : % -> % if PrimeField(p) has FINITE
--R ??? : (%, NonNegativeInteger) -> %
--R basis : PositiveInteger -> Vector(%)
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if PrimeField(p) has CHARNZ or PrimeField(p) has FINITE
--R charthRoot : % -> % if PrimeField(p) has FINITE
--R conditionP : Matrix(%) -> Union(Vector(%, "failed") if PrimeField(p) has FINITE
--R coordinates : Vector(%) -> Matrix(PrimeField(p))
--R coordinates : % -> Vector(PrimeField(p))
--R createNormalElement : () -> % if PrimeField(p) has FINITE
--R createPrimitiveElement : () -> % if PrimeField(p) has FINITE
--R definingPolynomial : () -> SparseUnivariatePolynomial(PrimeField(p))
--R degree : % -> OnePointCompletion(PositiveInteger)
--R differentiate : (%, NonNegativeInteger) -> % if PrimeField(p) has FINITE
--R differentiate : % -> % if PrimeField(p) has FINITE
--R discreteLog : (%, %) -> Union(NonNegativeInteger, "failed") if PrimeField(p) has CHARNZ or PrimeField(p) has FINITE
--R discreteLog : % -> NonNegativeInteger if PrimeField(p) has FINITE
--R divide : (%, %) -> Record(quotient: %, remainder: %)
--R enumerate : () -> List(%) if PrimeField(p) has FINITE
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(%, %) -> Union(List(%, "failed")
--R exquo : (%, %) -> Union(%, "failed")
--R extendedEuclidean : (%, %, %) -> Union(Record(coef1: %, coef2: %), "failed")
--R extendedEuclidean : (%, %) -> Record(coef1: %, coef2: %, generator: %)
--R extensionDegree : () -> PositiveInteger
--R extensionDegree : () -> OnePointCompletion(PositiveInteger)
--R factorsOfCyclicGroupSize : () -> List(Record(factor: Integer, exponent: Integer)) if PrimeField(p) has FINITE
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R generator : () -> % if PrimeField(p) has FINITE

```

```

--R index : PositiveInteger -> % if PrimeField(p) has FINITE
--R init : () -> % if PrimeField(p) has FINITE
--R linearAssociatedExp : (%,SparseUnivariatePolynomial(PrimeField(p))) -> % if PrimeField(p) has FINITE
--R linearAssociatedLog : (%,%) -> Union(SparseUnivariatePolynomial(PrimeField(p)),"failed") if
--R linearAssociatedLog : % -> SparseUnivariatePolynomial(PrimeField(p)) if PrimeField(p) has FINITE
--R linearAssociatedOrder : % -> SparseUnivariatePolynomial(PrimeField(p)) if PrimeField(p) has FINITE
--R lookup : % -> PositiveInteger if PrimeField(p) has FINITE
--R minimalPolynomial : (%,PositiveInteger) -> SparseUnivariatePolynomial(PrimeField(p)) if PrimeField(p) has FINITE
--R minimalPolynomial : % -> SparseUnivariatePolynomial(PrimeField(p)) if PrimeField(p) has FINITE
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed")
--R nextItem : % -> Union(%,"failed") if PrimeField(p) has FINITE
--R norm : (%,PositiveInteger) -> % if PrimeField(p) has FINITE
--R normal? : % -> Boolean if PrimeField(p) has FINITE
--R normalElement : () -> % if PrimeField(p) has FINITE
--R order : % -> OnePointCompletion(PositiveInteger) if PrimeField(p) has CHARNZ or PrimeField(p) has FINITE
--R order : % -> PositiveInteger if PrimeField(p) has FINITE
--R primeFrobenius : % -> % if PrimeField(p) has CHARNZ or PrimeField(p) has FINITE
--R primeFrobenius : (%,NonNegativeInteger) -> % if PrimeField(p) has CHARNZ or PrimeField(p) has FINITE
--R primitive? : % -> Boolean if PrimeField(p) has FINITE
--R primitiveElement : () -> % if PrimeField(p) has FINITE
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %)
--R random : () -> % if PrimeField(p) has FINITE
--R representationType : () -> Union("prime",polynomial,normal,cyclic) if PrimeField(p) has FINITE
--R represents : Vector(PrimeField(p)) -> %
--R retractIfCan : % -> Union(PrimeField(p),"failed")
--R size : () -> NonNegativeInteger if PrimeField(p) has FINITE
--R subtractIfCan : (%,%) -> Union(%,"failed")
--R tableForDiscreteLogarithm : Integer -> Table(PositiveInteger,NonNegativeInteger) if PrimeField(p) has FINITE
--R trace : (%,PositiveInteger) -> % if PrimeField(p) has FINITE
--R transcendenceDegree : () -> NonNegativeInteger
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R
--E 1

```

```

)spool
)lisp (bye)

```

— FiniteField.help —

=====

FiniteField examples

=====

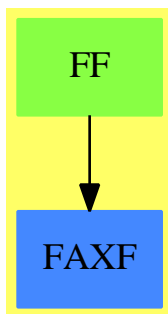
FiniteField(p,n) implements finite fields with p^n elements.
This package checks that p is prime.
For a non-checking version, see InnerFiniteField.

See Also:

- o `)show FiniteField`
- o `)show InnerFiniteField`

—————→

FiniteField (FF)



See

- ⇒ “FiniteFieldExtensionByPolynomial” (FFP) [7](#) on page [850](#)
- ⇒ “FiniteFieldExtension” (FFX) [7](#) on page [845](#)
- ⇒ “InnerFiniteField” (IFF) [10](#) on page [1283](#)

Exports:

0	1	algebraic?
associates?	basis	characteristic
charthRoot	coerce	conditionP
coordinates	createNormalElement	createPrimitiveElement
D	definingPolynomial	degree
differentiate	dimension	discreteLog
divide	euclideanSize	expressIdealMember
exquo	extendedEuclidean	extensionDegree
factor	factorsOfCyclicGroupSize	Frobenius
gcd	gcdPolynomial	generator
hash	index	inGroundField?
init	inv	latex
lcm	linearAssociatedExp	linearAssociatedLog
linearAssociatedOrder	lookup	minimalPolynomial
multiEuclidean	nextItem	norm
normal?	normalElement	one?
order	prime?	primeFrobenius
primitive?	primitiveElement	principalIdeal
random	recip	representationType
represents	retract	retractIfCan
sample	size	sizeLess?
squareFree	squareFreePart	subtractIfCan
tableForDiscreteLogarithm	trace	transcendenceDegree
transcendent?	unit?	unitCanonical
unitNormal	zero?	?*?
***?	?+?	?-?
-?	?/?	?=?
?^?	?~=?	?quo?
?rem?		

— domain **FF** **FiniteField** —

```

)abbrev domain FF FiniteField
++ Author: Mark Botch
++ Date Created: ???
++ Date Last Updated: 29 May 1990
++ Reference:
++ R.Lidl, H.Niederreiter: Finite Field, Encyclopedia of Mathematics and
++ Its Applications, Vol. 20, Cambridge Univ. Press, 1983, ISBN 0 521 30240 4
++ J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++ AXIOM Technical Report Series, ATR/5 NP2522.
++ Description:
++ FiniteField(p,n) implements finite fields with p**n elements.
++ This package checks that p is prime.
++ For a non-checking version, see \spadtype{InnerFiniteField}.

```

```

FiniteField(p:PositiveInteger, n:PositiveInteger): _
  FiniteAlgebraicExtensionField(PrimeField p) ==_
  FiniteFieldExtensionByPolynomial(PrimeField p, _
    createIrreduciblePoly(n)$FiniteFieldPolynomialPackage(PrimeField p))
-- old code for generating irreducible polynomials:
-- now "better" order (sparse polys first)
-- generateIrredPoly(n)$IrredPolyOverFiniteField(GF))

```

— FF.dotabb —

```

"FF" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FF"]
"FAXF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAXF"]
"FF" -> "FAXF"

```

domain FFCG FiniteFieldCyclicGroup

— FiniteFieldCyclicGroup.input —

```

)set break resume
)sys rm -f FiniteFieldCyclicGroup.output
)spool FiniteFieldCyclicGroup.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteFieldCyclicGroup
--R FiniteFieldCyclicGroup(p: PositiveInteger,extdeg: PositiveInteger) is a domain constructor
--R Abbreviation for FiniteFieldCyclicGroup is FFCG
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FFCG
--R
--R----- Operations -----
--R ??? : (PrimeField(p),%) -> %      ??? : (%,PrimeField(p)) -> %
--R ??? : (Fraction(Integer),%) -> %  ??? : (%,Fraction(Integer)) -> %
--R ??? : (%,%) -> %                  ??? : (Integer,%) -> %
--R ??? : (PositiveInteger,%) -> %     ??? : (%,Integer) -> %
--R ??? : (%,PositiveInteger) -> %     ?+? : (%,%) -> %
--R ?-? : (%,%) -> %                   -? : % -> %
--R ?/? : (%,PrimeField(p)) -> %       ?/? : (%,%) -> %

```



```

--R ?=? : (%,% ) -> Boolean
--R 0 : () -> %
--R ?^? : (% ,PositiveInteger) -> %
--R associates? : (% ,%) -> Boolean
--R coerce : PrimeField(p) -> %
--R coerce : % -> %
--R coerce : % -> OutputForm
--R dimension : () -> CardinalNumber
--R gcd : List(%) -> %
--R hash : % -> SingleInteger
--R inv : % -> %
--R lcm : List(%) -> %
--R norm : % -> PrimeField(p)
--R prime? : % -> Boolean
--R recip : % -> Union(%, "failed")
--R retract : % -> PrimeField(p)
--R sizeLess? : (% ,%) -> Boolean
--R squareFreePart : % -> %
--R transcendental? : % -> Boolean
--R unitCanonical : % -> %
--R ?~=? : (% ,%) -> Boolean
--R ?*? : (NonNegativeInteger ,%) -> %
--R ??? : (% ,NonNegativeInteger) -> %
--R D : (% ,NonNegativeInteger) -> % if PrimeField(p) has FINITE
--R D : % -> % if PrimeField(p) has FINITE
--R Frobenius : (% ,NonNegativeInteger) -> % if PrimeField(p) has FINITE
--R Frobenius : % -> % if PrimeField(p) has FINITE
--R ?^? : (% ,NonNegativeInteger) -> %
--R basis : PositiveInteger -> Vector(%)
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if PrimeField(p) has CHARNZ or PrimeField(p) has FINITE
--R charthRoot : % -> % if PrimeField(p) has FINITE
--R conditionP : Matrix(%) -> Union(Vector(%, "failed") if PrimeField(p) has FINITE
--R coordinates : Vector(%) -> Matrix(PrimeField(p))
--R coordinates : % -> Vector(PrimeField(p))
--R createNormalElement : () -> % if PrimeField(p) has FINITE
--R createPrimitiveElement : () -> % if PrimeField(p) has FINITE
--R definingPolynomial : () -> SparseUnivariatePolynomial(PrimeField(p))
--R degree : % -> OnePointCompletion(PositiveInteger)
--R differentiate : (% ,NonNegativeInteger) -> % if PrimeField(p) has FINITE
--R differentiate : % -> % if PrimeField(p) has FINITE
--R discreteLog : (% ,%) -> Union(NonNegativeInteger, "failed") if PrimeField(p) has CHARNZ or Pri
--R discreteLog : % -> NonNegativeInteger if PrimeField(p) has FINITE
--R divide : (% ,%) -> Record(quotient: %, remainder: %)
--R enumerate : () -> List(%) if PrimeField(p) has FINITE
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(%) ,%) -> Union(List(%, "failed")
--R exquo : (% ,%) -> Union(%, "failed")
--R extendedEuclidean : (% ,%,%) -> Union(Record(coef1: %,coef2: %), "failed")
--R extendedEuclidean : (% ,%) -> Record(coef1: %,coef2: %,generator: %)
1 : () -> %
?^? : (% ,Integer) -> %
algebraic? : % -> Boolean
basis : () -> Vector(%)
coerce : Fraction(Integer) -> %
coerce : Integer -> %
degree : % -> PositiveInteger
factor : % -> Factored(%)
gcd : (% ,%) -> %
inGroundField? : % -> Boolean
latex : % -> String
lcm : (% ,%) -> %
one? : % -> Boolean
?quo? : (% ,%) -> %
?rem? : (% ,%) -> %
sample : () -> %
squareFree : % -> Factored(%)
trace : % -> PrimeField(p)
unit? : % -> Boolean
zero? : % -> Boolean

```

```

--R extensionDegree : () -> PositiveInteger
--R extensionDegree : () -> OnePointCompletion(PositiveInteger)
--R factorsOfCyclicGroupSize : () -> List(Record(factor: Integer,exponent: Integer)) if PrimeField(p) has F
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynom
--R generator : () -> % if PrimeField(p) has FINITE
--R getZechTable : () -> PrimitiveArray(SingleInteger)
--R index : PositiveInteger -> % if PrimeField(p) has FINITE
--R init : () -> % if PrimeField(p) has FINITE
--R linearAssociatedExp : (% ,SparseUnivariatePolynomial(PrimeField(p))) -> % if PrimeField(p) has FINITE
--R linearAssociatedLog : (% ,%) -> Union(SparseUnivariatePolynomial(PrimeField(p)),"failed") if PrimeField(p)
--R linearAssociatedLog : % -> SparseUnivariatePolynomial(PrimeField(p)) if PrimeField(p) has FINITE
--R linearAssociatedOrder : % -> SparseUnivariatePolynomial(PrimeField(p)) if PrimeField(p) has FINITE
--R lookup : % -> PositiveInteger if PrimeField(p) has FINITE
--R minimalPolynomial : (% ,PositiveInteger) -> SparseUnivariatePolynomial(%) if PrimeField(p) has FINITE
--R minimalPolynomial : % -> SparseUnivariatePolynomial(PrimeField(p))
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed")
--R nextItem : % -> Union(%,"failed") if PrimeField(p) has FINITE
--R norm : (% ,PositiveInteger) -> % if PrimeField(p) has FINITE
--R normal? : % -> Boolean if PrimeField(p) has FINITE
--R normalElement : () -> % if PrimeField(p) has FINITE
--R order : % -> OnePointCompletion(PositiveInteger) if PrimeField(p) has CHARNZ or PrimeField(p) has FINITE
--R order : % -> PositiveInteger if PrimeField(p) has FINITE
--R primeFrobenius : % -> % if PrimeField(p) has CHARNZ or PrimeField(p) has FINITE
--R primeFrobenius : (% ,NonNegativeInteger) -> % if PrimeField(p) has CHARNZ or PrimeField(p) has FINITE
--R primitive? : % -> Boolean if PrimeField(p) has FINITE
--R primitiveElement : () -> % if PrimeField(p) has FINITE
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %)
--R random : () -> % if PrimeField(p) has FINITE
--R representationType : () -> Union("prime",polynomial,normal,cyclic) if PrimeField(p) has FINITE
--R represents : Vector(PrimeField(p)) -> %
--R retractIfCan : % -> Union(PrimeField(p),"failed")
--R size : () -> NonNegativeInteger if PrimeField(p) has FINITE
--R subtractIfCan : (% ,%) -> Union(%,"failed")
--R tableForDiscreteLogarithm : Integer -> Table(PositiveInteger,NonNegativeInteger) if PrimeField(p) has F
--R trace : (% ,PositiveInteger) -> % if PrimeField(p) has FINITE
--R transcendenceDegree : () -> NonNegativeInteger
--R unitNormal : % -> Record(unit: % ,canonical: % ,associate: %)
--R
--E 1

)spool
)lisp (bye)

```

— FiniteFieldCyclicGroup.help —

=====

FiniteFieldCyclicGroup examples

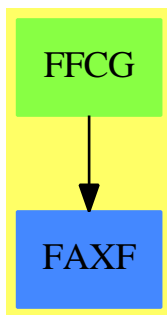
=====

`FiniteFieldCyclicGroup(p,n)` implements a finite field extension of degree n over the prime field with p elements. Its elements are represented by powers of a primitive element, i.e. a generator of the multiplicative (cyclic) group. As primitive element we choose the root of the extension polynomial, which is created by `createPrimitivePoly` from `FiniteFieldPolynomialPackage`. The Zech logarithms are stored in a table of size half of the field size, and use `SingleInteger` for representing field elements, hence, there are restrictions on the size of the field.

See Also:

o `show FiniteFieldCyclicGroup`

FiniteFieldCyclicGroup (FFCG)



See

⇒ “FiniteFieldCyclicGroupExtensionByPolynomial” (FFCGP) 7 on page 834

⇒ “FiniteFieldCyclicGroupExtension” (FFCGX) 7 on page 829

Exports:

0	1	algebraic?
associates?	basis	characteristic
charthRoot	coerce	conditionP
coordinates	createNormalElement	createPrimitiveElement
D	definingPolynomial	degree
differentiate	dimension	discreteLog
divide	euclideanSize	expressIdealMember
exquo	extendedEuclidean	extensionDegree
factor	factorsOfCyclicGroupSize	Frobenius
gcd	gcdPolynomial	generator
getZechTable	hash	index
inGroundField?	init	inv
latex	lcm	linearAssociatedExp
linearAssociatedLog	linearAssociatedOrder	lookup
minimalPolynomial	multiEuclidean	nextItem
norm	normal?	normalElement
one?	order	prime?
primeFrobenius	primitive?	primitiveElement
principalIdeal	random	recip
representationType	represents	retract
retractIfCan	sample	size
sizeLess?	squareFree	squareFreePart
subtractIfCan	tableForDiscreteLogarithm	trace
transcendenceDegree	transcendent?	unit?
unitCanonical	unitNormal	zero?
?*?	?**?	?+?
?-?	?-	?/?
?=?	?^?	?~=?
?quo?	?rem?	

— domain **FFCG** FiniteFieldCyclicGroup —

```

)abbrev domain FFCG FiniteFieldCyclicGroup
++ Authors: J.Grabmeier, A.Scheerhorn
++ Date Created: 04.04.1991
++ References:
++ R.Lidl, H.Niederreiter: Finite Field, Encycoldia of Mathematics and
++ Its Applications, Vol. 20, Cambridge Univ. Press, 1983, ISBN 0 521 30240 4
++ Description:
++ FiniteFieldCyclicGroup(p,n) implements a finite field extension of degee n
++ over the prime field with p elements. Its elements are represented by
++ powers of a primitive element, i.e. a generator of the multiplicative
++ (cyclic) group. As primitive element we choose the root of the extension
++ polynomial, which is created by createPrimitivePoly from
++ \spadtype{FiniteFieldPolynomialPackage}. The Zech logarithms are stored
++ in a table of size half of the field size, and use \spadtype{SingleInteger}

```

++ for representing field elements, hence, there are restrictions
 ++ on the size of the field.

```
FiniteFieldCyclicGroup(p,extdeg):_
  Exports == Implementation where
  p : PositiveInteger
  extdeg : PositiveInteger
  PI      ==> PositiveInteger
  FFPOLY   ==> FiniteFieldPolynomialPackage(PrimeField(p))
  SI      ==> SingleInteger
  Exports ==> FiniteAlgebraicExtensionField(PrimeField(p)) with
    getZechTable:() -> PrimitiveArray(SingleInteger)
    ++ getZechTable() returns the zech logarithm table of the field.
    ++ This table is used to perform additions in the field quickly.
  Implementation ==> FiniteFieldCyclicGroupExtensionByPolynomial(PrimeField(p),_
    createPrimitivePoly(extdeg)$FFPOLY)
```

— FFCG.dotabb —

```
"FFCG" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FFCG"]
"FAXF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAXF"]
"FFCG" -> "FAXF"
```

domain FFCGX FiniteFieldCyclicGroupExtension

— FiniteFieldCyclicGroupExtension.input —

```
)set break resume
)sys rm -f FiniteFieldCyclicGroupExtension.output
)spool FiniteFieldCyclicGroupExtension.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteFieldCyclicGroupExtension
--R FiniteFieldCyclicGroupExtension(GF: FiniteFieldCategory,extdeg: PositiveInteger) is a domain
--R Abbreviation for FiniteFieldCyclicGroupExtension is FFCGX
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FFCGX
```

```

--R
--R----- Operations -----
--R ?? : (GF,%) -> %                ?? : (%,GF) -> %
--R ?? : (Fraction(Integer),%) -> %  ?? : (%,Fraction(Integer)) -> %
--R ?? : (%,%) -> %                  ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %    ??? : (%,Integer) -> %
--R ??? : (%,PositiveInteger) -> %   ?+? : (%,%) -> %
--R ?-? : (%,%) -> %                 -? : % -> %
--R ?/? : (%,GF) -> %                ?/? : (%,%) -> %
--R ?? : (%,%) -> Boolean             D : % -> % if GF has FINITE
--R 1 : () -> %                       0 : () -> %
--R ?? : (%,Integer) -> %             ?? : (%,PositiveInteger) -> %
--R algebraic? : % -> Boolean          associates? : (%,%) -> Boolean
--R basis : () -> Vector(%)            coerce : GF -> %
--R coerce : Fraction(Integer) -> %    coerce : % -> %
--R coerce : Integer -> %              coerce : % -> OutputForm
--R coordinates : % -> Vector(GF)      degree : % -> PositiveInteger
--R dimension : () -> CardinalNumber    factor : % -> Factored(%)
--R gcd : List(%) -> %                 gcd : (%,%) -> %
--R hash : % -> SingleInteger           inGroundField? : % -> Boolean
--R inv : % -> %                       latex : % -> String
--R lcm : List(%) -> %                 lcm : (%,%) -> %
--R norm : % -> GF                     one? : % -> Boolean
--R prime? : % -> Boolean               ?quo? : (%,%) -> %
--R recip : % -> Union(%, "failed")     ?rem? : (%,%) -> %
--R represents : Vector(GF) -> %        retract : % -> GF
--R sample : () -> %                   sizeLess? : (%,%) -> Boolean
--R squareFree : % -> Factored(%)       squareFreePart : % -> %
--R trace : % -> GF                    transcendent? : % -> Boolean
--R unit? : % -> Boolean                unitCanonical : % -> %
--R zero? : % -> Boolean                ?~=? : (%,%) -> Boolean
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R D : (%,NonNegativeInteger) -> % if GF has FINITE
--R Frobenius : (%,NonNegativeInteger) -> % if GF has FINITE
--R Frobenius : % -> % if GF has FINITE
--R ?? : (%,NonNegativeInteger) -> %
--R basis : PositiveInteger -> Vector(%)
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if GF has CHARNZ or GF has FINITE
--R charthRoot : % -> % if GF has FINITE
--R conditionP : Matrix(%) -> Union(Vector(%, "failed") if GF has FINITE
--R coordinates : Vector(%) -> Matrix(GF)
--R createNormalElement : () -> % if GF has FINITE
--R createPrimitiveElement : () -> % if GF has FINITE
--R definingPolynomial : () -> SparseUnivariatePolynomial(GF)
--R degree : % -> OnePointCompletion(PositiveInteger)
--R differentiate : (%,NonNegativeInteger) -> % if GF has FINITE
--R differentiate : % -> % if GF has FINITE
--R discreteLog : (%,%) -> Union(NonNegativeInteger, "failed") if GF has CHARNZ or GF has FINITE

```

```

--R discreteLog : % -> NonNegativeInteger if GF has FINITE
--R divide : (%,% ) -> Record(quotient: %,remainder: %)
--R enumerate : () -> List(%) if GF has FINITE
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(%),%) -> Union(List(%),"failed")
--R exquo : (%,% ) -> Union(%,"failed")
--R extendedEuclidean : (%,%,% ) -> Union(Record(coef1: %,coef2: %),"failed")
--R extendedEuclidean : (%,% ) -> Record(coef1: %,coef2: %,generator: %)
--R extensionDegree : () -> PositiveInteger
--R extensionDegree : () -> OnePointCompletion(PositiveInteger)
--R factorsOfCyclicGroupSize : () -> List(Record(factor: Integer,exponent: Integer)) if GF has F
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUniva
--R generator : () -> % if GF has FINITE
--R getZechTable : () -> PrimitiveArray(SingleInteger)
--R index : PositiveInteger -> % if GF has FINITE
--R init : () -> % if GF has FINITE
--R linearAssociatedExp : (% ,SparseUnivariatePolynomial(GF)) -> % if GF has FINITE
--R linearAssociatedLog : (%,% ) -> Union(SparseUnivariatePolynomial(GF),"failed") if GF has FINI
--R linearAssociatedLog : % -> SparseUnivariatePolynomial(GF) if GF has FINITE
--R linearAssociatedOrder : % -> SparseUnivariatePolynomial(GF) if GF has FINITE
--R lookup : % -> PositiveInteger if GF has FINITE
--R minimalPolynomial : (% ,PositiveInteger) -> SparseUnivariatePolynomial(%) if GF has FINITE
--R minimalPolynomial : % -> SparseUnivariatePolynomial(GF)
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed")
--R nextItem : % -> Union(%,"failed") if GF has FINITE
--R norm : (% ,PositiveInteger) -> % if GF has FINITE
--R normal? : % -> Boolean if GF has FINITE
--R normalElement : () -> % if GF has FINITE
--R order : % -> OnePointCompletion(PositiveInteger) if GF has CHARNZ or GF has FINITE
--R order : % -> PositiveInteger if GF has FINITE
--R primeFrobenius : % -> % if GF has CHARNZ or GF has FINITE
--R primeFrobenius : (% ,NonNegativeInteger) -> % if GF has CHARNZ or GF has FINITE
--R primitive? : % -> Boolean if GF has FINITE
--R primitiveElement : () -> % if GF has FINITE
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %)
--R random : () -> % if GF has FINITE
--R representationType : () -> Union("prime",polynomial,normal,cyclic) if GF has FINITE
--R retractIfCan : % -> Union(GF,"failed")
--R size : () -> NonNegativeInteger if GF has FINITE
--R subtractIfCan : (%,% ) -> Union(%,"failed")
--R tableForDiscreteLogarithm : Integer -> Table(PositiveInteger,NonNegativeInteger) if GF has F
--R trace : (% ,PositiveInteger) -> % if GF has FINITE
--R transcendenceDegree : () -> NonNegativeInteger
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R
--E 1

)spool
)lisp (bye)

```

— FiniteFieldCyclicGroupExtension.help —

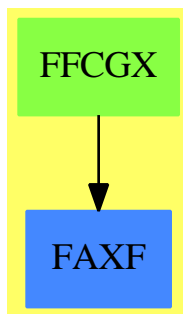
```
=====
FiniteFieldCyclicGroupExtension examples
=====
```

FiniteFieldCyclicGroupExtension(GF,n) implements a extension of degree n over the ground field GF. Its elements are represented by powers of a primitive element, i.e. a generator of the multiplicative (cyclic) group. As primitive element we choose the root of the extension polynomial, which is created by createPrimitivePoly from FiniteFieldPolynomialPackage. Zech logarithms are stored in a table of size half of the field size, and use SingleInteger for representing field elements, hence, there are restrictions on the size of the field.

See Also:

o)show FiniteFieldCyclicGroupExtension

FiniteFieldCyclicGroupExtension (FFCGX)



See

⇒ “FiniteFieldCyclicGroupExtensionByPolynomial” (FFCGP) [7](#) on page [834](#)

⇒ “FiniteFieldCyclicGroup” (FFCG) [7](#) on page [824](#)

Exports:

0	1	algebraic?
associates?	basis	characteristic
charthRoot	coerce	conditionP
coordinates	createNormalElement	createPrimitiveElement
D	definingPolynomial	degree
differentiate	dimension	discreteLog
divide	euclideanSize	expressIdealMember
exquo	extendedEuclidean	extensionDegree
factor	factorsOfCyclicGroupSize	Frobenius
gcd	gcdPolynomial	generator
getZechTable	hash	index
inGroundField?	init	inv
latex	lcm	linearAssociatedExp
linearAssociatedLog	linearAssociatedOrder	lookup
minimalPolynomial	multiEuclidean	nextItem
norm	normal?	normalElement
one?	order	prime?
primeFrobenius	primitive?	primitiveElement
principalIdeal	random	recip
representationType	represents	retract
retractIfCan	sample	size
sizeLess?	squareFree	squareFreePart
subtractIfCan	tableForDiscreteLogarithm	trace
transcendenceDegree	transcendent?	unit?
unitCanonical	unitNormal	zero?
?*?	?**?	?+?
?-?	?-?	?/?
?=?	?^?	?~=?
?quo?	?rem?	

— domain **FFCGX** **FiniteFieldCyclicGroupExtension** —

```

)abbrev domain FFCGX FiniteFieldCyclicGroupExtension
++ Authors: J.Grabmeier, A.Scheerhorn
++ Date Created: 04.04.1991
++ References:
++ R.Lidl, H.Niederreiter: Finite Field, Encycoldia of Mathematics and
++ Its Applications, Vol. 20, Cambridge Univ. Press, 1983, ISBN 0 521 30240 4
++ J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++ AXIOM Technical Report Series, ATR/5 NP2522.
++ Description:
++ FiniteFieldCyclicGroupExtension(GF,n) implements a extension of degree n
++ over the ground field GF. Its elements are represented by powers of
++ a primitive element, i.e. a generator of the multiplicative (cyclic) group.
++ As primitive element we choose the root of the extension polynomial, which
++ is created by createPrimitivePoly from

```

```

++ \spadtype{FiniteFieldPolynomialPackage}. Zech logarithms are stored
++ in a table of size half of the field size, and use \spadtype{SingleInteger}
++ for representing field elements, hence, there are restrictions
++ on the size of the field.

```

```

FiniteFieldCyclicGroupExtension(GF,extdeg):_
  Exports == Implementation where
  GF      : FiniteFieldCategory
  extdeg  : PositiveInteger
  PI      ==> PositiveInteger
  FFPOLY  ==> FiniteFieldPolynomialPackage(GF)
  SI      ==> SingleInteger
  Exports ==> FiniteAlgebraicExtensionField(GF) with
    getZechTable:() -> PrimitiveArray(SingleInteger)
    ++ getZechTable() returns the zech logarithm table of the field.
    ++ This table is used to perform additions in the field quickly.
  Implementation ==> FiniteFieldCyclicGroupExtensionByPolynomial(GF,_
    createPrimitivePoly(extdeg)$FFPOLY)

```

— FFCGX.dotabb —

```

"FFCGX" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FFCGX"]
"FAXF"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAXF"]
"FFCGX" -> "FAXF"

```

domain FFCGP FiniteFieldCyclicGroupExtensionByPolynomial

— FiniteFieldCyclicGroupExtensionByPolynomial.input —

```

)set break resume
)sys rm -f FiniteFieldCyclicGroupExtensionByPolynomial.output
)spool FiniteFieldCyclicGroupExtensionByPolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteFieldCyclicGroupExtensionByPolynomial

```

```

--R FiniteFieldCyclicGroupExtensionByPolynomial(GF: FiniteFieldCategory, defpol: SparseUnivariate
--R Abbreviation for FiniteFieldCyclicGroupExtensionByPolynomial is FFCGP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FFCGP
--R
--R----- Operations -----
--R ?? : (GF, %) -> %           ?? : (% , GF) -> %
--R ?? : (Fraction(Integer), %) -> %   ?? : (% , Fraction(Integer)) -> %
--R ?? : (% , %) -> %           ?? : (Integer, %) -> %
--R ?? : (PositiveInteger, %) -> %   ??? : (% , Integer) -> %
--R ??? : (% , PositiveInteger) -> %  ?+? : (% , %) -> %
--R ?-? : (% , %) -> %           -? : % -> %
--R ?/? : (% , GF) -> %           ?/? : (% , %) -> %
--R ?? : (% , %) -> Boolean       D : % -> % if GF has FINITE
--R 1 : () -> %                   0 : () -> %
--R ?? : (% , Integer) -> %       ?? : (% , PositiveInteger) -> %
--R algebraic? : % -> Boolean     associates? : (% , %) -> Boolean
--R basis : () -> Vector(%)       coerce : GF -> %
--R coerce : Fraction(Integer) -> %   coerce : % -> %
--R coerce : Integer -> %         coerce : % -> OutputForm
--R coordinates : % -> Vector(GF)   degree : % -> PositiveInteger
--R dimension : () -> CardinalNumber factor : % -> Factored(%)
--R gcd : List(%) -> %           gcd : (% , %) -> %
--R hash : % -> SingleInteger     inGroundField? : % -> Boolean
--R inv : % -> %                 latex : % -> String
--R lcm : List(%) -> %           lcm : (% , %) -> %
--R norm : % -> GF               one? : % -> Boolean
--R prime? : % -> Boolean         ?quo? : (% , %) -> %
--R recip : % -> Union(%, "failed") ?rem? : (% , %) -> %
--R represents : Vector(GF) -> %   retract : % -> GF
--R sample : () -> %             sizeLess? : (% , %) -> Boolean
--R squareFree : % -> Factored(%)  squareFreePart : % -> %
--R trace : % -> GF              transcendent? : % -> Boolean
--R unit? : % -> Boolean          unitCanonical : % -> %
--R zero? : % -> Boolean          ?~=? : (% , %) -> Boolean
--R ?? : (NonNegativeInteger, %) -> %
--R ??? : (% , NonNegativeInteger) -> %
--R D : (% , NonNegativeInteger) -> % if GF has FINITE
--R Frobenius : (% , NonNegativeInteger) -> % if GF has FINITE
--R Frobenius : % -> % if GF has FINITE
--R ?? : (% , NonNegativeInteger) -> %
--R basis : PositiveInteger -> Vector(%)
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if GF has CHARNZ or GF has FINITE
--R charthRoot : % -> % if GF has FINITE
--R conditionP : Matrix(%) -> Union(Vector(%, "failed") if GF has FINITE
--R coordinates : Vector(%) -> Matrix(GF)
--R createNormalElement : () -> % if GF has FINITE
--R createPrimitiveElement : () -> % if GF has FINITE
--R definingPolynomial : () -> SparseUnivariatePolynomial(GF)

```

```

--R degree : % -> OnePointCompletion(PositiveInteger)
--R differentiate : (% , NonNegativeInteger) -> % if GF has FINITE
--R differentiate : % -> % if GF has FINITE
--R discreteLog : (% , %) -> Union(NonNegativeInteger, "failed") if GF has CHARNZ or GF has FINITE
--R discreteLog : % -> NonNegativeInteger if GF has FINITE
--R divide : (% , %) -> Record(quotient: %, remainder: %)
--R enumerate : () -> List(%) if GF has FINITE
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(%), %) -> Union(List(%), "failed")
--R exquo : (% , %) -> Union(% , "failed")
--R extendedEuclidean : (% , %, %) -> Union(Record(coef1: %, coef2: %), "failed")
--R extendedEuclidean : (% , %) -> Record(coef1: %, coef2: %, generator: %)
--R extensionDegree : () -> PositiveInteger
--R extensionDegree : () -> OnePointCompletion(PositiveInteger)
--R factorsOfCyclicGroupSize : () -> List(Record(factor: Integer, exponent: Integer)) if GF has FINITE
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R generator : () -> % if GF has FINITE
--R getZechTable : () -> PrimitiveArray(SingleInteger)
--R index : PositiveInteger -> % if GF has FINITE
--R init : () -> % if GF has FINITE
--R linearAssociatedExp : (% , SparseUnivariatePolynomial(GF)) -> % if GF has FINITE
--R linearAssociatedLog : (% , %) -> Union(SparseUnivariatePolynomial(GF), "failed") if GF has FINITE
--R linearAssociatedLog : % -> SparseUnivariatePolynomial(GF) if GF has FINITE
--R linearAssociatedOrder : % -> SparseUnivariatePolynomial(GF) if GF has FINITE
--R lookup : % -> PositiveInteger if GF has FINITE
--R minimalPolynomial : (% , PositiveInteger) -> SparseUnivariatePolynomial(%) if GF has FINITE
--R minimalPolynomial : % -> SparseUnivariatePolynomial(GF)
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed")
--R nextItem : % -> Union(% , "failed") if GF has FINITE
--R norm : (% , PositiveInteger) -> % if GF has FINITE
--R normal? : % -> Boolean if GF has FINITE
--R normalElement : () -> % if GF has FINITE
--R order : % -> OnePointCompletion(PositiveInteger) if GF has CHARNZ or GF has FINITE
--R order : % -> PositiveInteger if GF has FINITE
--R primeFrobenius : % -> % if GF has CHARNZ or GF has FINITE
--R primeFrobenius : (% , NonNegativeInteger) -> % if GF has CHARNZ or GF has FINITE
--R primitive? : % -> Boolean if GF has FINITE
--R primitiveElement : () -> % if GF has FINITE
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %)
--R random : () -> % if GF has FINITE
--R representationType : () -> Union("prime", "polynomial", "normal", "cyclic") if GF has FINITE
--R retractIfCan : % -> Union(GF, "failed")
--R size : () -> NonNegativeInteger if GF has FINITE
--R subtractIfCan : (% , %) -> Union(% , "failed")
--R tableForDiscreteLogarithm : Integer -> Table(PositiveInteger, NonNegativeInteger) if GF has FINITE
--R trace : (% , PositiveInteger) -> % if GF has FINITE
--R transcendenceDegree : () -> NonNegativeInteger
--R unitNormal : % -> Record(unit: %, canonical: %, associate: %)
--R
--E 1

```

```
)spool
)lisp (bye)
```

— **FiniteFieldCyclicGroupExtensionByPolynomial.help** —

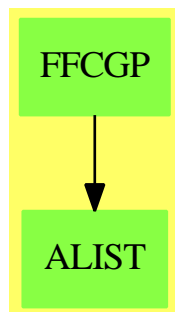
```
=====
FiniteFieldCyclicGroupExtensionByPolynomial examples
=====
```

`FiniteFieldCyclicGroupExtensionByPolynomial(GF,defpol)` implements a finite extension field of the ground field GF. Its elements are represented by powers of a primitive element, i.e. a generator of the multiplicative (cyclic) group. As primitive element we choose the root of the extension polynomial `defpol`, which **MUST** be primitive (user responsibility). Zech logarithms are stored in a table of size half of the field size, and use `SingleInteger` for representing field elements, hence, there are restrictions on the size of the field.

See Also:

```
o )show FiniteFieldCyclicGroupExtensionByPolynomial
```

FiniteFieldCyclicGroupExtensionByPolynomial (FFCGP)



See

- ⇒ “FiniteFieldCyclicGroupExtension” (FFCGX) [7](#) on page [829](#)
- ⇒ “FiniteFieldCyclicGroup” (FFCG) [7](#) on page [824](#)

Exports:

0	1	algebraic?
associates?	basis	characteristic
charthRoot	coerce	conditionP
coordinates	createNormalElement	createPrimitiveElement
D	definingPolynomial	degree
differentiate	dimension	discreteLog
divide	euclideanSize	expressIdealMember
exquo	extendedEuclidean	extensionDegree
factor	factorsOfCyclicGroupSize	Frobenius
gcd	gcdPolynomial	generator
getZechTable	hash	index
inGroundField?	init	inv
latex	lcm	linearAssociatedExp
linearAssociatedLog	linearAssociatedOrder	lookup
minimalPolynomial	multiEuclidean	nextItem
norm	normal?	normalElement
one?	order	prime?
primeFrobenius	primitive?	primitiveElement
principalIdeal	random	recip
representationType	represents	retract
retractIfCan	sample	size
sizeLess?	squareFree	squareFreePart
subtractIfCan	tableForDiscreteLogarithm	trace
transcendenceDegree	transcendent?	unit?
unitCanonical	unitNormal	zero?
?*?	?**?	?+?
?-?	?-	?/?
?=?	?^?	?~=?
?quo?	?rem?	

— domain **FFCGP** FiniteFieldCyclicGroupExtensionByPolynomial

```

)abbrev domain FFCGP FiniteFieldCyclicGroupExtensionByPolynomial
++ Authors: J.Grabmeier, A.Scheerhorn
++ Date Created: 26.03.1991
++ Date Last Updated: 31 March 1991
++ References:
++ R.Lidl, H.Niederreiter: Finite Field, Encycoldia of Mathematics and
++ Its Applications, Vol. 20, Cambridge Univ. Press, 1983, ISBN 0 521 30240 4
++ J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++ AXIOM Technical Report Series, ATR/5 NP2522.
++ Description:
++ FiniteFieldCyclicGroupExtensionByPolynomial(GF,defpol) implements a
++ finite extension field of the ground field GF. Its elements are
++ represented by powers of a primitive element, i.e. a generator of the

```

```

++ multiplicative (cyclic) group. As primitive
++ element we choose the root of the extension polynomial defpol,
++ which MUST be primitive (user responsibility). Zech logarithms are stored
++ in a table of size half of the field size, and use \spadtype{SingleInteger}
++ for representing field elements, hence, there are restrictions
++ on the size of the field.

```

```

FiniteFieldCyclicGroupExtensionByPolynomial(GF,defpol):_
  Exports == Implementation where
  GF      : FiniteFieldCategory          -- the ground field
  defpol: SparseUnivariatePolynomial GF -- the extension polynomial
  -- the root of defpol is used as the primitive element

```

```

PI    ==> PositiveInteger
NNI   ==> NonNegativeInteger
I     ==> Integer
SI    ==> SingleInteger
SUP   ==> SparseUnivariatePolynomial
SAE   ==> SimpleAlgebraicExtension(GF,SUP GF,defpol)
V     ==> Vector GF
FFP   ==> FiniteFieldExtensionByPolynomial(GF,defpol)
FFF   ==> FiniteFieldFunctions(GF)
OUT   ==> OutputForm
ARR   ==> PrimitiveArray(SI)
TBL   ==> Table(PI,NNI)

```

```

Exports ==> FiniteAlgebraicExtensionField(GF) with

```

```

  getZechTable:() -> ARR
  ++ getZechTable() returns the zech logarithm table of the field
  ++ it is used to perform additions in the field quickly.
  Implementation ==> add

```

```

-- global variables =====

```

```

Rep:= SI
-- elements are represented by small integers in the range
-- (-1)..(size()-2). The (-1) representing the field element zero,
-- the other small integers representing the corresponding power
-- of the primitive element, the root of the defining polynomial

```

```

-- it would be very nice if we could use the representation
-- Rep:= Union("zero", IntegerMod(size())$GF ** degree(defpol) -1)),
-- why doesn't the compiler like this ?

```

```

extdeg:NNI :=degree(defpol)$(SUP GF)::NNI
-- the extension degree

```

```

sizeFF:NNI:=(size())$GF ** extdeg) pretend NNI

```

```

-- the size of the field

if sizeFF > 2**20 then
  error "field too large for this representation"

sizeCG:SI:=(sizeFF - 1) pretend SI
-- the order of the cyclic group

sizeFG:SI:=(sizeCG quo (size()$GF-1)) pretend SI
-- the order of the factor group

zechlog:ARR:=new(((sizeFF+1) quo 2)::NNI,-1::SI)$ARR
-- the table for the zech logarithm

alpha :=new()$Symbol :: OutputForm
-- get a new symbol for the output representation of
-- the elements

primEltGF:GF:=
  odd?(extdeg)$I => -$GF coefficient(defpol,0)$(SUP GF)
  coefficient(defpol,0)$(SUP GF)
-- the corresponding primitive element of the groundfield
-- equals the trace of the primitive element w.r.t. the groundfield

facOfGroupSize := nil()$(List Record(factor:Integer,exponent:Integer))
-- the factorization of sizeCG

initzech?:Boolean:=true
-- gets false after initialization of the zech logarithm array

initelt?:Boolean:=true
-- gets false after initialization of the normal element

normalElt:SI:=0
-- the global variable containing a normal element

-- functions =====

-- for completeness we have to give a dummy implementation for
-- 'tableForDiscreteLogarithm', although this function is not
-- necessary in the cyclic group representation case

tableForDiscreteLogarithm(fac) == table()$TBL

getZechTable() == zechlog
initializeZech:() -> Void
initializeElt: () -> Void

```



```

order(x:$):PI ==
  zero?(x) =>
    error"order: order of zero undefined"
  (sizeCG quo gcd(sizeCG,x pretend NNI))::PI

primitive?(x:$) ==
--   zero?(x) or one?(x) => false
   zero?(x) or (x = 1) => false
   gcd(x::Rep,sizeCG)$Rep = 1$Rep => true
   false

coordinates(x:$) ==
  x=0 => new(extdeg,0)$(Vector GF)
  primElement:SAE:=convert(monomial(1,1)$(SUP GF))$SAE
-- the primitive element in the corresponding algebraic extension
  coordinates(primElement **$SAE (x pretend SI))$SAE

x:$ + y:$ ==
  if initzech? then initializeZech()
  zero? x => y
  zero? y => x
  d:Rep:=positiveRemainder(y -$Rep x,sizeCG)$Rep
  (d pretend SI) <= shift(sizeCG,-$SI (1$SI)) =>
    zechlog.(d pretend SI) =$SI -1::SI => 0
    addmod(x,zechlog.(d pretend SI) pretend Rep,sizeCG)$Rep
  --d:Rep:=positiveRemainder(x -$Rep y,sizeCG)$Rep
  d:Rep:=(sizeCG -$SI d)::Rep
  addmod(y,zechlog.(d pretend SI) pretend Rep,sizeCG)$Rep
  --positiveRemainder(x +$Rep zechlog.(d pretend SI) -$Rep d,sizeCG)$Rep

initializeZech() ==
  zechlog:=createZechTable(defpol)$FFF
  -- set initialization flag
  initzech? := false
  void()$Void

basis(n:PI) ==
  extensionDegree() rem n ^= 0 =>
    error("argument must divide extension degree")
  m:=sizeCG quo (size()$GF**n-1)
  [index((1+i*m) ::PI) for i in 0..(n-1)]::Vector $

n:I * x:$ == ((n::GF)::)$ * x

minimalPolynomial(a) ==
  f:SUP $:=monomial(1,1)$(SUP $) - monomial(a,0)$(SUP $)
  u:$:=Frobenius(a)
  while not(u = a) repeat
    f:=f * (monomial(1,1)$(SUP $) - monomial(u,0)$(SUP $))

```

```

    u:=Frobenius(u)
    p:SUP GF:=0$(SUP GF)
    while not zero?(f)$(SUP $) repeat
        g:GF:=retract(leadingCoefficient(f)$(SUP $))
        p:=p+monomial(g,_
            degree(f)$(SUP $))$(SUP GF)
        f:=reductum(f)$(SUP $)
    p

factorsOfCyclicGroupSize() ==
    if empty? facOfGroupSize then initializeElt()
    facOfGroupSize

representationType() == "cyclic"

definingPolynomial() == defpol

random() ==
    positiveRemainder(random()$Rep,sizeFF pretend Rep)$Rep -$Rep 1$Rep

represents(v) ==
    u:FFP:=represents(v)$FFP
    u =$FFP 0$FFP => 0
    discreteLog(u)$FFP pretend Rep

coerce(e:GF):$ ==
    zero?(e)$GF => 0
    log:I:=discreteLog(primEltGF,e)$GF::NNI *$I sizeFG
    -- version before 10.20.92: log pretend Rep
    -- 1$GF is coerced to sizeCG pretend Rep by old version
    -- now 1$GF is coerced to 0$Rep which is correct.
    positiveRemainder(log,sizeCG) pretend Rep

retractIfCan(x:$) ==
    zero? x => 0$GF
    u:= (x::Rep) exquo$Rep (sizeFG pretend Rep)
    u = "failed" => "failed"
    primEltGF **$GF ((u::$) pretend SI)

retract(x:$) ==
    a:=retractIfCan(x)
    a="failed" => error "element not in groundfield"
    a :: GF

basis() == [index(i :: PI) for i in 1..extdeg]::Vector $

```

```

inGroundField?(x) ==
  zero? x=> true
  positiveRemainder(x::Rep,sizeFG pretend Rep)$Rep =$Rep 0$Rep => true
  false

discreteLog(b:$,x:$) ==
  zero? x => "failed"
  e:= extendedEuclidean(b,sizeCG,x)$Rep
  e = "failed" => "failed"
  e1:Record(coef1:$,coef2:$) := e :: Record(coef1:$,coef2:$)
  positiveRemainder(e1.coef1,sizeCG)$Rep pretend NNI

- x:$ ==
  zero? x => 0
  characteristic() =$I 2 => x
  addmod(x,shift(sizeCG,-1)$SI pretend Rep,sizeCG)

generator() == 1$SI
createPrimitiveElement() == 1$SI
primitiveElement() == 1$SI

discreteLog(x:$) ==
  zero? x => error "discrete logarithm error"
  x pretend NNI

normalElement() ==
  if initelt? then initializeElt()
  normalElt::$

initializeElt() ==
  facOfGroupSize := factors(factor(sizeCG)$Integer)
  normalElt:=createNormalElement() pretend SI
  initelt?:=false
  void()$Void

extensionDegree() == extdeg pretend PI

characteristic() == characteristic()$GF

lookup(x:$) ==
  x =$Rep (-$Rep 1$Rep) => sizeFF pretend PI
  (x +$Rep 1$Rep) pretend PI

index(a:PI) ==
  positiveRemainder(a,sizeFF)$I pretend Rep -$Rep 1$Rep

0 == (-$Rep 1$Rep)

1 == 0$Rep

```

```

-- to get a "exponent like" output form
coerce(x:$):OUT ==
  x =$Rep (-$Rep 1$Rep) => "0":OUT
  x =$Rep 0$Rep => "1":OUT
  y:I:=lookup(x)-1
  alpha **$OUT (y::OUT)

x:$ = y:$ == x =$Rep y

x:$ * y:$ ==
  x = 0 => 0
  y = 0 => 0
  addmod(x,y,sizeCG)$Rep

a:GF * x:$ == coerce(a)@$ * x
x:$/a:GF == x/coerce(a)@$

-- x:$ / a:GF ==
-- a = 0$GF => error "division by zero"
-- x * inv(coerce(a))

inv(x:$) ==
  zero?(x) => error "inv: not invertible"
-- one?(x) => 1
  (x = 1) => 1
  sizeCG -$Rep x

x:$ ** n:PI == x ** n::I

x:$ ** n:NNI == x ** n::I

x:$ ** n:I ==
  m:Rep:=positiveRemainder(n,sizeCG)$I pretend Rep
  m =$Rep 0$Rep => 1
  x = 0 => 0
  mulmod(m,x,sizeCG::Rep)$Rep

```

— FFCGP.dotabb —

```

"FFCGP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FFCGP"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"FFCGP" -> "ALIST"

```

domain FFX FiniteFieldExtension

— FiniteFieldExtension.input —

```

)set break resume
)sys rm -f FiniteFieldExtension.output
)spool FiniteFieldExtension.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteFieldExtension
--R FiniteFieldExtension(GF: FiniteFieldCategory,n: PositiveInteger) is a domain constructor
--R Abbreviation for FiniteFieldExtension is FFX
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FFX
--R
--R----- Operations -----
--R ?? : (GF,%) -> %
--R ?? : (Fraction(Integer),%) -> %
--R ?? : (%,%) -> %
--R ?? : (PositiveInteger,%) -> %
--R ??? : (%,PositiveInteger) -> %
--R ?-? : (%,%) -> %
--R ?/? : (%,GF) -> %
--R ?? : (%,%) -> Boolean
--R 1 : () -> %
--R ?? : (%,Integer) -> %
--R algebraic? : % -> Boolean
--R basis : () -> Vector(%)
--R coerce : Fraction(Integer) -> %
--R coerce : Integer -> %
--R coordinates : % -> Vector(GF)
--R dimension : () -> CardinalNumber
--R gcd : List(%) -> %
--R hash : % -> SingleInteger
--R inv : % -> %
--R lcm : List(%) -> %
--R norm : % -> GF
--R prime? : % -> Boolean
--R recip : % -> Union(%, "failed")
--R represents : Vector(GF) -> %
--R sample : () -> %
--R squareFree : % -> Factored(%)
--R trace : % -> GF
--R unit? : % -> Boolean
--R zero? : % -> Boolean
--R ?? : (%,GF) -> %
--R ?? : (%,Fraction(Integer)) -> %
--R ?? : (Integer,%) -> %
--R ??? : (%,Integer) -> %
--R ?+? : (%,%) -> %
--R -? : % -> %
--R ?/? : (%,%) -> %
--R D : % -> % if GF has FINITE
--R 0 : () -> %
--R ?? : (%,PositiveInteger) -> %
--R associates? : (%,%) -> Boolean
--R coerce : GF -> %
--R coerce : % -> %
--R coerce : % -> OutputForm
--R degree : % -> PositiveInteger
--R factor : % -> Factored(%)
--R gcd : (%,%) -> %
--R inGroundField? : % -> Boolean
--R latex : % -> String
--R lcm : (%,%) -> %
--R one? : % -> Boolean
--R ?quo? : (%,%) -> %
--R ?rem? : (%,%) -> %
--R retract : % -> GF
--R sizeLess? : (%,%) -> Boolean
--R squareFreePart : % -> %
--R transcendent? : % -> Boolean
--R unitCanonical : % -> %
--R ?~=? : (%,%) -> Boolean

```

```

--R ?? : (NonNegativeInteger,%) -> %
--R ***? : (%,NonNegativeInteger) -> %
--R D : (%,NonNegativeInteger) -> % if GF has FINITE
--R Frobenius : (%,NonNegativeInteger) -> % if GF has FINITE
--R Frobenius : % -> % if GF has FINITE
--R ?? : (%,NonNegativeInteger) -> %
--R basis : PositiveInteger -> Vector(%)
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if GF has CHARNZ or GF has FINITE
--R charthRoot : % -> % if GF has FINITE
--R conditionP : Matrix(%) -> Union(Vector(%), "failed") if GF has FINITE
--R coordinates : Vector(%) -> Matrix(GF)
--R createNormalElement : () -> % if GF has FINITE
--R createPrimitiveElement : () -> % if GF has FINITE
--R definingPolynomial : () -> SparseUnivariatePolynomial(GF)
--R degree : % -> OnePointCompletion(PositiveInteger)
--R differentiate : (%,NonNegativeInteger) -> % if GF has FINITE
--R differentiate : % -> % if GF has FINITE
--R discreteLog : (%,%) -> Union(NonNegativeInteger, "failed") if GF has CHARNZ or GF has FINITE
--R discreteLog : % -> NonNegativeInteger if GF has FINITE
--R divide : (%,%) -> Record(quotient: %, remainder: %)
--R enumerate : () -> List(%) if GF has FINITE
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(%),%) -> Union(List(%), "failed")
--R exquo : (%,%) -> Union(%, "failed")
--R extendedEuclidean : (%,%,%) -> Union(Record(coef1: %, coef2: %), "failed")
--R extendedEuclidean : (%,%) -> Record(coef1: %, coef2: %, generator: %)
--R extensionDegree : () -> PositiveInteger
--R extensionDegree : () -> OnePointCompletion(PositiveInteger)
--R factorsOfCyclicGroupSize : () -> List(Record(factor: Integer, exponent: Integer)) if GF has FINITE
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(GF)
--R generator : () -> % if GF has FINITE
--R index : PositiveInteger -> % if GF has FINITE
--R init : () -> % if GF has FINITE
--R linearAssociatedExp : (%, SparseUnivariatePolynomial(GF)) -> % if GF has FINITE
--R linearAssociatedLog : (%,%) -> Union(SparseUnivariatePolynomial(GF), "failed") if GF has FINITE
--R linearAssociatedLog : % -> SparseUnivariatePolynomial(GF) if GF has FINITE
--R linearAssociatedOrder : % -> SparseUnivariatePolynomial(GF) if GF has FINITE
--R lookup : % -> PositiveInteger if GF has FINITE
--R minimalPolynomial : (%, PositiveInteger) -> SparseUnivariatePolynomial(%) if GF has FINITE
--R minimalPolynomial : % -> SparseUnivariatePolynomial(GF)
--R multiEuclidean : (List(%),%) -> Union(List(%), "failed")
--R nextItem : % -> Union(%, "failed") if GF has FINITE
--R norm : (%, PositiveInteger) -> % if GF has FINITE
--R normal? : % -> Boolean if GF has FINITE
--R normalElement : () -> % if GF has FINITE
--R order : % -> OnePointCompletion(PositiveInteger) if GF has CHARNZ or GF has FINITE
--R order : % -> PositiveInteger if GF has FINITE
--R primeFrobenius : % -> % if GF has CHARNZ or GF has FINITE
--R primeFrobenius : (%, NonNegativeInteger) -> % if GF has CHARNZ or GF has FINITE

```

```

--R primitive? : % -> Boolean if GF has FINITE
--R primitiveElement : () -> % if GF has FINITE
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %)
--R random : () -> % if GF has FINITE
--R representationType : () -> Union("prime",polynomial,normal,cyclic) if GF has FINITE
--R retractIfCan : % -> Union(GF,"failed")
--R size : () -> NonNegativeInteger if GF has FINITE
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R tableForDiscreteLogarithm : Integer -> Table(PositiveInteger,NonNegativeInteger) if GF has F
--R trace : (%,PositiveInteger) -> % if GF has FINITE
--R transcendenceDegree : () -> NonNegativeInteger
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R
--E 1

)spool
)lisp (bye)

```

— **FiniteFieldExtension.help** —

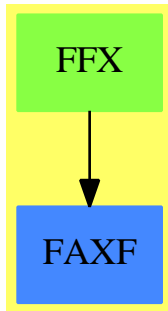
```

=====
FiniteFieldExtension examples
=====

```

FiniteFieldExtensionByPolynomial(GF, n) implements an extension of the finite field GF of degree n generated by the extension polynomial constructed by createIrreduciblePoly from FiniteFieldPolynomialPackage.

See Also:
o)show FiniteFieldExtension

FiniteFieldExtension (FFX)**See**

- ⇒ “FiniteFieldExtensionByPolynomial” (FFP) [7](#) on page [850](#)
- ⇒ “InnerFiniteField” (IFF) [10](#) on page [1283](#)
- ⇒ “FiniteField” (FF) [7](#) on page [819](#)

Exports:

0	1	algebraic?
associates?	basis	characteristic
charthRoot	coerce	conditionP
coordinates	createNormalElement	createPrimitiveElement
D	definingPolynomial	degree
differentiate	dimension	discreteLog
divide	euclideanSize	expressIdealMember
exquo	extendedEuclidean	extensionDegree
factor	factorsOfCyclicGroupSize	Frobenius
gcd	gcdPolynomial	generator
hash	index	inGroundField?
init	inv	latex
lcm	linearAssociatedExp	linearAssociatedLog
linearAssociatedOrder	lookup	minimalPolynomial
multiEuclidean	nextItem	norm
normal?	normalElement	one?
order	prime?	primeFrobenius
primitive?	primitiveElement	principalIdeal
random	recip	representationType
represents	retract	retractIfCan
sample	size	sizeLess?
squareFree	squareFreePart	subtractIfCan
tableForDiscreteLogarithm	trace	transcendenceDegree
transcendent?	unit?	unitCanonical
unitNormal	zero?	?*?
***?	?+?	?-?
-?	?/?	?=?
?^?	?~=?	?quo?
?rem?		

— domain FFX FiniteFieldExtension —

```

)abbrev domain FFX FiniteFieldExtension
++ Authors: R.Sutor, J. Grabmeier, A. Scheerhorn
++ Date Created:
++ Date Last Updated: 31 March 1991
++ Reference:
++ R.Lidl, H.Niederreiter: Finite Field, Encyclopedia of Mathematics and
++ Its Applications, Vol. 20, Cambridge Univ. Press, 1983, ISBN 0 521 30240 4
++ J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++ AXIOM Technical Report Series, ATR/5 NP2522.
++ Description:
++ FiniteFieldExtensionByPolynomial(GF, n) implements an extension
++ of the finite field GF of degree n generated by the extension
++ polynomial constructed by createIrreduciblePoly from
++ \spadtype{FiniteFieldPolynomialPackage}.
```

```

FiniteFieldExtension(GF, n): Exports == Implementation where
  GF: FiniteFieldCategory
  n : PositiveInteger
  Exports ==> FiniteAlgebraicExtensionField(GF)
  -- MonogenicAlgebra(GF, SUP) with -- have to check this
  Implementation ==> FiniteFieldExtensionByPolynomial(GF,
    createIrreduciblePoly(n)$FiniteFieldPolynomialPackage(GF))
  -- old code for generating irreducible polynomials:
  -- now "better" order (sparse polys first)
  -- generateIrredPoly(n)$IrredPolyOverFiniteField(GF))

```

— FFX.dotabb —

```

"FFX" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FFX"]
"FAXF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAXF"]
"FFX" -> "FAXF"

```

domain FFP FiniteFieldExtensionByPolynomial

— FiniteFieldExtensionByPolynomial.input —

```

)set break resume
)sys rm -f FiniteFieldExtensionByPolynomial.output
)spool FiniteFieldExtensionByPolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteFieldExtensionByPolynomial
--R FiniteFieldExtensionByPolynomial(GF: FiniteFieldCategory,defpol: SparseUnivariatePolynomial(GF)) is a
--R Abbreviation for FiniteFieldExtensionByPolynomial is FFP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FFP
--R
--R----- Operations -----
--R ??? : (GF,%) -> %          ??? : (%,GF) -> %
--R ??? : (Fraction(Integer),%) -> %    ??? : (%,Fraction(Integer)) -> %
--R ??? : (%,%) -> %          ??? : (Integer,%) -> %

```

```

--R ???: (PositiveInteger,%) -> %
--R ????: (%,PositiveInteger) -> %
--R ?-?: (%,%) -> %
--R ?/? : (%,GF) -> %
--R ==?: (%,%) -> Boolean
--R 1 : () -> %
--R ?^? : (%,Integer) -> %
--R algebraic? : % -> Boolean
--R basis : () -> Vector(%)
--R coerce : Fraction(Integer) -> %
--R coerce : Integer -> %
--R coordinates : % -> Vector(GF)
--R dimension : () -> CardinalNumber
--R gcd : List(%) -> %
--R hash : % -> SingleInteger
--R inv : % -> %
--R lcm : List(%) -> %
--R norm : % -> GF
--R prime? : % -> Boolean
--R recip : % -> Union(%, "failed")
--R represents : Vector(GF) -> %
--R sample : () -> %
--R squareFree : % -> Factored(%)
--R trace : % -> GF
--R unit? : % -> Boolean
--R zero? : % -> Boolean
--R ???: (NonNegativeInteger,%) -> %
--R ????: (%,NonNegativeInteger) -> %
--R D : (%,NonNegativeInteger) -> % if GF has FINITE
--R Frobenius : (%,NonNegativeInteger) -> % if GF has FINITE
--R Frobenius : % -> % if GF has FINITE
--R ?^? : (%,NonNegativeInteger) -> %
--R basis : PositiveInteger -> Vector(%)
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if GF has CHARNZ or GF has FINITE
--R charthRoot : % -> % if GF has FINITE
--R conditionP : Matrix(%) -> Union(Vector(%), "failed") if GF has FINITE
--R coordinates : Vector(%) -> Matrix(GF)
--R createNormalElement : () -> % if GF has FINITE
--R createPrimitiveElement : () -> % if GF has FINITE
--R definingPolynomial : () -> SparseUnivariatePolynomial(GF)
--R degree : % -> OnePointCompletion(PositiveInteger)
--R differentiate : (%,NonNegativeInteger) -> % if GF has FINITE
--R differentiate : % -> % if GF has FINITE
--R discreteLog : (%,%) -> Union(NonNegativeInteger, "failed") if GF has CHARNZ or GF has FINITE
--R discreteLog : % -> NonNegativeInteger if GF has FINITE
--R divide : (%,%) -> Record(quotient: %, remainder: %)
--R enumerate : () -> List(%) if GF has FINITE
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(%),%) -> Union(List(%), "failed")
???: (%,Integer) -> %
?+?: (%,%) -> %
-?: % -> %
?/? : (%,%) -> %
D : % -> % if GF has FINITE
0 : () -> %
?^? : (%,PositiveInteger) -> %
associates? : (%,%) -> Boolean
coerce : GF -> %
coerce : % -> %
coerce : % -> OutputForm
degree : % -> PositiveInteger
factor : % -> Factored(%)
gcd : (%,%) -> %
inGroundField? : % -> Boolean
latex : % -> String
lcm : (%,%) -> %
one? : % -> Boolean
?quo? : (%,%) -> %
?rem? : (%,%) -> %
retract : % -> GF
sizeLess? : (%,%) -> Boolean
squareFreePart : % -> %
transcendent? : % -> Boolean
unitCanonical : % -> %
?~?: (%,%) -> Boolean

```

```

--R exquo : (% ,%) -> Union(%,"failed")
--R extendedEuclidean : (% ,%,%) -> Union(Record(coef1: %,coef2: %),"failed")
--R extendedEuclidean : (% ,%) -> Record(coef1: %,coef2: %,generator: %)
--R extensionDegree : () -> PositiveInteger
--R extensionDegree : () -> OnePointCompletion(PositiveInteger)
--R factorsOfCyclicGroupSize : () -> List(Record(factor: Integer,exponent: Integer)) if GF has FINITE
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R generator : () -> % if GF has FINITE
--R index : PositiveInteger -> % if GF has FINITE
--R init : () -> % if GF has FINITE
--R linearAssociatedExp : (% ,SparseUnivariatePolynomial(GF)) -> % if GF has FINITE
--R linearAssociatedLog : (% ,%) -> Union(SparseUnivariatePolynomial(GF),"failed") if GF has FINITE
--R linearAssociatedLog : % -> SparseUnivariatePolynomial(GF) if GF has FINITE
--R linearAssociatedOrder : % -> SparseUnivariatePolynomial(GF) if GF has FINITE
--R lookup : % -> PositiveInteger if GF has FINITE
--R minimalPolynomial : (% ,PositiveInteger) -> SparseUnivariatePolynomial(%) if GF has FINITE
--R minimalPolynomial : % -> SparseUnivariatePolynomial(GF)
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed")
--R nextItem : % -> Union(%,"failed") if GF has FINITE
--R norm : (% ,PositiveInteger) -> % if GF has FINITE
--R normal? : % -> Boolean if GF has FINITE
--R normalElement : () -> % if GF has FINITE
--R order : % -> OnePointCompletion(PositiveInteger) if GF has CHARNZ or GF has FINITE
--R order : % -> PositiveInteger if GF has FINITE
--R primeFrobenius : % -> % if GF has CHARNZ or GF has FINITE
--R primeFrobenius : (% ,NonNegativeInteger) -> % if GF has CHARNZ or GF has FINITE
--R primitive? : % -> Boolean if GF has FINITE
--R primitiveElement : () -> % if GF has FINITE
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %)
--R random : () -> % if GF has FINITE
--R representationType : () -> Union("prime",polynomial,normal,cyclic) if GF has FINITE
--R retractIfCan : % -> Union(GF,"failed")
--R size : () -> NonNegativeInteger if GF has FINITE
--R subtractIfCan : (% ,%) -> Union(%,"failed")
--R tableForDiscreteLogarithm : Integer -> Table(PositiveInteger,NonNegativeInteger) if GF has FINITE
--R trace : (% ,PositiveInteger) -> % if GF has FINITE
--R transcendenceDegree : () -> NonNegativeInteger
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R
--E 1

)spool
)lisp (bye)

```

— FiniteFieldExtensionByPolynomial.help —

FiniteFieldExtensionByPolynomial examples

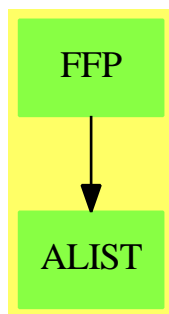
=====

FiniteFieldExtensionByPolynomial(GF, defpol) implements the extension of the finite field GF generated by the extension polynomial defpol which MUST be irreducible. Note: the user has the responsibility to ensure that defpol is irreducible.

See Also:

o)show FiniteFieldExtensionByPolynomial

FiniteFieldExtensionByPolynomial (FFP)



See

- ⇒ “FiniteFieldExtension” (FFX) [7](#) on page [845](#)
- ⇒ “InnerFiniteField” (IFF) [10](#) on page [1283](#)
- ⇒ “FiniteField” (FF) [7](#) on page [819](#)

Exports:

0	1	algebraic?
associates?	basis	characteristic
charthRoot	coerce	conditionP
coordinates	createNormalElement	createPrimitiveElement
D	definingPolynomial	degree
differentiate	dimension	discreteLog
divide	euclideanSize	expressIdealMember
exquo	extendedEuclidean	extensionDegree
factor	factorsOfCyclicGroupSize	Frobenius
gcd	gcdPolynomial	generator
hash	index	inGroundField?
init	inv	latex
lcm	linearAssociatedExp	linearAssociatedLog
linearAssociatedOrder	lookup	minimalPolynomial
multiEuclidean	nextItem	norm
normal?	normalElement	one?
order	prime?	primeFrobenius
primitive?	primitiveElement	principalIdeal
random	recip	representationType
represents	retract	retractIfCan
sample	size	sizeLess?
squareFree	squareFreePart	subtractIfCan
tableForDiscreteLogarithm	trace	transcendenceDegree
transcendent?	unit?	unitCanonical
unitNormal	zero?	?*?
***?	?+?	?-?
-?	?/?	?/?
?=?	?^?	?~=?
?quo?	?rem?	

— domain FFP FiniteFieldExtensionByPolynomial —

```

)abbrev domain FFP FiniteFieldExtensionByPolynomial
++ Authors: R.Sutor, J. Grabmeier, O. Gschnitzer, A. Scheerhorn
++ Date Created:
++ Date Last Updated: 31 March 1991
++ Reference:
++ R.Lidl, H.Niederreiter: Finite Field, Encyclopedia of Mathematics an
++ Its Applications, Vol. 20, Cambridge Univ. Press, 1983, ISBN 0 521 30240 4
++ J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++ AXIOM Technical Report Series, ATR/5 NP2522.
++ Description:
++ FiniteFieldExtensionByPolynomial(GF, defpol) implements the extension
++ of the finite field GF generated by the extension polynomial
++ defpol which MUST be irreducible.
++ Note: the user has the responsibility to ensure that

```

```

++ defpol is irreducible.

FiniteFieldExtensionByPolynomial(GF:FiniteFieldCategory, _
  defpol: SparseUnivariatePolynomial GF): Exports == Implementation where
-- GF      : FiniteFieldCategory
-- defpol  : SparseUnivariatePolynomial GF

PI  ==> PositiveInteger
NNI ==> NonNegativeInteger
SUP ==> SparseUnivariatePolynomial
I   ==> Integer
R   ==> Record(key:PI, entry:NNI)
TBL ==> Table(PI, NNI)
SAE ==> SimpleAlgebraicExtension(GF, SUP GF, defpol)
OUT ==> OutputForm

Exports ==> FiniteAlgebraicExtensionField(GF)

Implementation ==> add

-- global variables =====

Rep:=SAE

extdeg:PI      := degree(defpol)$(SUP GF) pretend PI
-- the extension degree

alpha          := new()$Symbol :: OutputForm
-- a new symbol for the output form of field elements

sizeCG:Integer := size()$GF**extdeg - 1
-- the order of the multiplicative group

facOfGroupSize := nil()$(List Record(factor:Integer, exponent:Integer))
-- the factorization of sizeCG

normalElt:PI:=1
-- for the lookup of the normal Element computed by
-- createNormalElement

primitiveElt:PI:=1
-- for the lookup of the primitive Element computed by
-- createPrimitiveElement()

initlog?:Boolean:=true
-- gets false after initialization of the discrete logarithm table

initelt?:Boolean:=true
-- gets false after initialization of the primitive and the
-- normal element

```

```

discLogTable:Table(PI,TBL):=table()$Table(PI,TBL)
-- tables indexed by the factors of sizeCG,
-- discLogTable(factor) is a table with keys
-- primitiveElement() ** (i * (sizeCG quo factor)) and entries i for
-- i in 0..n-1, n computed in initialize() in order to use
-- the minimal size limit 'limit' optimal.

-- functions =====

-- createNormalElement() ==
-- a:=primitiveElement()
-- nElt:=generator()
-- for i in 1.. repeat
--     normal? nElt => return nElt
--     nElt:=nElt*a
-- nElt

generator() == reduce(monomial(1,1)$SUP(GF))$Rep
norm x == resultant(defpol, lift x)

initializeElt: () -> Void
initializeLog: () -> Void
basis(n:PI) ==
    (extdeg rem n) ^= 0 => error "argument must divide extension degree"
    a:=$norm(primitiveElement(),n)
    vector [a**i for i in 0..n-1]

degree(x) ==
    y:=$=1
    m:=zero(extdeg,extdeg+1)$(Matrix GF)
    for i in 1..extdeg+1 repeat
        setColumn_!(m,i,coordinates(y))$(Matrix GF)
        y:=y*x
    rank(m)::PI

minimalPolynomial(x:$) ==
    y:=$=1
    m:=zero(extdeg,extdeg+1)$(Matrix GF)
    for i in 1..extdeg+1 repeat
        setColumn_!(m,i,coordinates(y))$(Matrix GF)
        y:=y*x
    v:=first nullSpace(m)$(Matrix GF)
    +/[monomial(v.(i+1),i)$(SUP GF) for i in 0..extdeg]

normal?(x) ==
    l:List List GF:=[entries coordinates x]
    a:=x

```



```

for i in 2..extdeg repeat
  a:=Frobenius(a)
  l:=concat(l,entries coordinates a)$(List List GF)
  ((rank matrix(l)$(Matrix GF)) = extdeg::NNI) => true
false

a:GF * x:$ == a *$Rep x
n:I * x:$ == n *$Rep x
-x == -$Rep x
random() == random()$Rep
coordinates(x:$) == coordinates(x)$Rep
represents(v) == represents(v)$Rep
coerce(x:GF):$ == coerce(x)$Rep
definingPolynomial() == defpol
retract(x) == retract(x)$Rep
retractIfCan(x) == retractIfCan(x)$Rep
index(x) == index(x)$Rep
lookup(x) == lookup(x)$Rep
x:$/y:$ == x /$Rep y
x:$/a:GF == x/coerce(a)
-- x:$ / a:GF ==
-- a = 0$GF => error "division by zero"
-- x * inv(coerce(a))
x:$ * y:$ == x *$Rep y
x:$ + y:$ == x +$Rep y
x:$ - y:$ == x -$Rep y
x:$ = y:$ == x =$Rep y
basis() == basis()$Rep
0 == 0$Rep
1 == 1$Rep

factorsOfCyclicGroupSize() ==
  if empty? facOfGroupSize then initializeElt()
  facOfGroupSize

representationType() == "polynomial"

tableForDiscreteLogarithm(fac) ==
  if initlog? then initializeLog()
  tbl:=search(fac::PI,discLogTable)$Table(PI,TBL)
  tbl case "failed" =>
    error "tableForDiscreteLogarithm: argument must be prime divisor_
of the order of the multiplicative group"
  tbl pretend TBL

primitiveElement() ==
  if initelt? then initializeElt()
  index(primitiveElt)

```

```

normalElement() ==
  if initelt? then initializeElt()
  index(normalElt)

initializeElt() ==
  facOfGroupSize:=factors(factor(sizeCG)$Integer)
  -- get a primitive element
  pE:=createPrimitiveElement()
  primitiveElt:=lookup(pE)
  -- create a normal element
  nElt:=generator()
  while not normal? nElt repeat
    nElt:=nElt*pE
  normalElt:=lookup(nElt)
  -- set elements initialization flag
  initelt? := false
  void()$Void

initializeLog() ==
  if initelt? then initializeElt()
-- set up tables for discrete logarithm
  limit:Integer:=30
  -- the minimum size for the discrete logarithm table
  for f in facOfGroupSize repeat
    fac:=f.factor
    base:=$:=primitiveElement() ** (sizeCG quo fac)
    l:Integer:=length(fac)$Integer
    n:Integer:=0
    if odd?(l)$Integer then n:=shift(fac,-(l quo 2))
      else n:=shift(1,(l quo 2))

    if n < limit then
      d:=(fac-1) quo limit + 1
      n:=(fac-1) quo d + 1
      tbl:TBL:=table()$TBL
      a:=$:=1
      for i in (0::NNI)..(n-1)::NNI repeat
        insert_!([lookup(a),i::NNI]$R,tbl)$TBL
        a:=a*base
      insert_!([fac::PI,copy(tbl)$TBL]_
        $Record(key:PI,entry:TBL),discLogTable)$Table(PI,TBL)
  -- set logarithm initialization flag
  initlog? := false
  -- tell user about initialization
  --print("discrete logarithm tables initialized":OUT)
  void()$Void

coerce(e:$):OutputForm == outputForm(lift(e),alpha)

extensionDegree() == extdeg

```

```

size() == (sizeCG + 1) pretend NNI

-- sizeOfGroundField() == size()$GF

inGroundField?(x) ==
  retractIfCan(x) = "failed" => false
  true

characteristic() == characteristic()$GF

```

— FFP.dotabb —

```

"FFP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FFP"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"FFP" -> "ALIST"

```

domain FFNB FiniteFieldNormalBasis

— FiniteFieldNormalBasis.input —

```

)set break resume
)sys rm -f FiniteFieldNormalBasis.output
)spool FiniteFieldNormalBasis.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteFieldNormalBasis
--R FiniteFieldNormalBasis(p: PositiveInteger,extdeg: PositiveInteger) is a domain constructor
--R Abbreviation for FiniteFieldNormalBasis is FFNB
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FFNB
--R
--R----- Operations -----
--R ???: (PrimeField(p),%) -> %      ??? : (%,PrimeField(p)) -> %
--R ??? : (Fraction(Integer),%) -> %  ??? : (%,Fraction(Integer)) -> %
--R ??? : (%,%) -> %                ??? : (Integer,%) -> %
--R ??? : (PositiveInteger,%) -> %    ??? : (%,Integer) -> %
--R ??? : (%,PositiveInteger) -> %    ?+? : (%,%) -> %

```

```

--R ?-? : (%,% ) -> %
--R ?/? : (% ,PrimeField(p)) -> %
--R ?=? : (%,% ) -> Boolean
--R 0 : () -> %
--R ?? : (% ,PositiveInteger) -> %
--R associates? : (%,% ) -> Boolean
--R coerce : PrimeField(p) -> %
--R coerce : % -> %
--R coerce : % -> OutputForm
--R dimension : () -> CardinalNumber
--R gcd : List(% ) -> %
--R hash : % -> SingleInteger
--R inv : % -> %
--R lcm : List(% ) -> %
--R norm : % -> PrimeField(p)
--R prime? : % -> Boolean
--R recip : % -> Union(% ,"failed")
--R retract : % -> PrimeField(p)
--R sizeLess? : (%,% ) -> Boolean
--R squareFreePart : % -> %
--R transcendental? : % -> Boolean
--R unitCanonical : % -> %
--R ?~=? : (%,% ) -> Boolean
--R ?*? : (NonNegativeInteger,% ) -> %
--R ?**? : (% ,NonNegativeInteger) -> %
--R D : (% ,NonNegativeInteger) -> % if PrimeField(p) has FINITE
--R D : % -> % if PrimeField(p) has FINITE
--R Frobenius : (% ,NonNegativeInteger) -> % if PrimeField(p) has FINITE
--R Frobenius : % -> % if PrimeField(p) has FINITE
--R ?? : (% ,NonNegativeInteger) -> %
--R basis : PositiveInteger -> Vector(% )
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(% ,"failed") if PrimeField(p) has CHARNZ or PrimeField(p) has FINITE
--R charthRoot : % -> % if PrimeField(p) has FINITE
--R conditionP : Matrix(% ) -> Union(Vector(% ),"failed") if PrimeField(p) has FINITE
--R coordinates : Vector(% ) -> Matrix(PrimeField(p))
--R coordinates : % -> Vector(PrimeField(p))
--R createNormalElement : () -> % if PrimeField(p) has FINITE
--R createPrimitiveElement : () -> % if PrimeField(p) has FINITE
--R definingPolynomial : () -> SparseUnivariatePolynomial(PrimeField(p))
--R degree : % -> OnePointCompletion(PositiveInteger)
--R differentiate : (% ,NonNegativeInteger) -> % if PrimeField(p) has FINITE
--R differentiate : % -> % if PrimeField(p) has FINITE
--R discreteLog : (%,% ) -> Union(NonNegativeInteger,"failed") if PrimeField(p) has CHARNZ or PrimeField(p) has FINITE
--R discreteLog : % -> NonNegativeInteger if PrimeField(p) has FINITE
--R divide : (%,% ) -> Record(quotient: %,remainder: %)
--R enumerate : () -> List(% ) if PrimeField(p) has FINITE
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(% ),%) -> Union(List(% ),"failed")
--R exquo : (%,% ) -> Union(% ,"failed")
--R ? : % -> %
--R ?/? : (%,% ) -> %
--R 1 : () -> %
--R ?? : (% ,Integer) -> %
--R algebraic? : % -> Boolean
--R basis : () -> Vector(% )
--R coerce : Fraction(Integer) -> %
--R coerce : Integer -> %
--R degree : % -> PositiveInteger
--R factor : % -> Factored(% )
--R gcd : (%,% ) -> %
--R inGroundField? : % -> Boolean
--R latex : % -> String
--R lcm : (%,% ) -> %
--R one? : % -> Boolean
--R ?quo? : (%,% ) -> %
--R ?rem? : (%,% ) -> %
--R sample : () -> %
--R squareFree : % -> Factored(% )
--R trace : % -> PrimeField(p)
--R unit? : % -> Boolean
--R zero? : % -> Boolean

```

```

--R extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %),"failed")
--R extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %)
--R extensionDegree : () -> PositiveInteger
--R extensionDegree : () -> OnePointCompletion(PositiveInteger)
--R factorsOfCyclicGroupSize : () -> List(Record(factor: Integer,exponent: Integer)) if PrimeField(p) has FINITE
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R generator : () -> % if PrimeField(p) has FINITE
--R getMultiplicationMatrix : () -> Matrix(PrimeField(p))
--R getMultiplicationTable : () -> Vector(List(Record(value: PrimeField(p),index: SingleInteger)))
--R index : PositiveInteger -> % if PrimeField(p) has FINITE
--R init : () -> % if PrimeField(p) has FINITE
--R linearAssociatedExp : (%,SparseUnivariatePolynomial(PrimeField(p))) -> % if PrimeField(p) has FINITE
--R linearAssociatedLog : (%,%) -> Union(SparseUnivariatePolynomial(PrimeField(p)),"failed") if PrimeField(p) has FINITE
--R linearAssociatedLog : % -> SparseUnivariatePolynomial(PrimeField(p)) if PrimeField(p) has FINITE
--R linearAssociatedOrder : % -> SparseUnivariatePolynomial(PrimeField(p)) if PrimeField(p) has FINITE
--R lookup : % -> PositiveInteger if PrimeField(p) has FINITE
--R minimalPolynomial : (%,PositiveInteger) -> SparseUnivariatePolynomial(%) if PrimeField(p) has FINITE
--R minimalPolynomial : % -> SparseUnivariatePolynomial(PrimeField(p))
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed")
--R nextItem : % -> Union(%,"failed") if PrimeField(p) has FINITE
--R norm : (%,PositiveInteger) -> % if PrimeField(p) has FINITE
--R normal? : % -> Boolean if PrimeField(p) has FINITE
--R normalElement : () -> % if PrimeField(p) has FINITE
--R order : % -> OnePointCompletion(PositiveInteger) if PrimeField(p) has CHARNZ or PrimeField(p) has FINITE
--R order : % -> PositiveInteger if PrimeField(p) has FINITE
--R primeFrobenius : % -> % if PrimeField(p) has CHARNZ or PrimeField(p) has FINITE
--R primeFrobenius : (%,NonNegativeInteger) -> % if PrimeField(p) has CHARNZ or PrimeField(p) has FINITE
--R primitive? : % -> Boolean if PrimeField(p) has FINITE
--R primitiveElement : () -> % if PrimeField(p) has FINITE
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %)
--R random : () -> % if PrimeField(p) has FINITE
--R representationType : () -> Union("prime",polynomial,normal,cyclic) if PrimeField(p) has FINITE
--R represents : Vector(PrimeField(p)) -> %
--R retractIfCan : % -> Union(PrimeField(p),"failed")
--R size : () -> NonNegativeInteger if PrimeField(p) has FINITE
--R sizeMultiplication : () -> NonNegativeInteger
--R subtractIfCan : (%,%) -> Union(%,"failed")
--R tableForDiscreteLogarithm : Integer -> Table(PositiveInteger,NonNegativeInteger) if PrimeField(p) has FINITE
--R trace : (%,PositiveInteger) -> % if PrimeField(p) has FINITE
--R transcendenceDegree : () -> NonNegativeInteger
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R
--E 1

)spool
)lisp (bye)

```

— FiniteFieldNormalBasis.help —

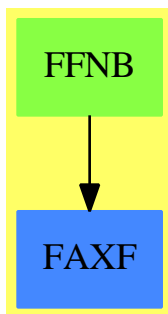
```
=====
FiniteFieldNormalBasis examples
=====
```

FiniteFieldNormalBasis(p,n) implements a finite extension field of degree n over the prime field with p elements. The elements are represented by coordinate vectors with respect to a normal basis, i.e. a basis consisting of the conjugates (q-powers) of an element, in this case called normal element. This is chosen as a root of the extension polynomial created by createNormalPoly

See Also:

o)show FiniteFieldNormalBasis

—————→

FiniteFieldNormalBasis (FFNB)

See

⇒ “FiniteFieldNormalBasisExtensionByPolynomial” (FFNBP) [7](#) on page [871](#)

⇒ “FiniteFieldNormalBasisExtension” (FFNBX) [7](#) on page [865](#)

Exports:

0	1	algebraic?
associates?	basis	characteristic
charthRoot	coerce	conditionP
coordinates	createNormalElement	createPrimitiveElement
D	definingPolynomial	degree
differentiate	dimension	discreteLog
divide	euclideanSize	expressIdealMember
exquo	extendedEuclidean	extensionDegree
factor	factorsOfCyclicGroupSize	Frobenius
gcd	gcdPolynomial	generator
getMultiplicationMatrix	getMultiplicationTable	hash
index	inGroundField?	init
inv	latex	lcm
linearAssociatedExp	linearAssociatedLog	linearAssociatedOrder
lookup	minimalPolynomial	multiEuclidean
nextItem	norm	normal?
normalElement	one?	order
prime?	primeFrobenius	primitive?
primitiveElement	principalIdeal	random
recip	representationType	represents
retract	retractIfCan	sample
size	sizeLess?	sizeMultiplication
squareFree	squareFreePart	subtractIfCan
tableForDiscreteLogarithm	trace	transcendenceDegree
transcendent?	unit?	unitCanonical
unitNormal	zero?	?*?
***?	?+?	?-?
-?	?/?	?=?
?^?	?~=?	?quo?
?rem?		

— domain FFNB FiniteFieldNormalBasis —

```

)abbrev domain FFNB FiniteFieldNormalBasis
++ Authors: J.Grabmeier, A.Scheerhorn
++ Date Created: 26.03.1991
++ References:
++ R.Lidl, H.Niederreiter: Finite Field, Encyclopedia of Mathematics and
++ Its Applications, Vol. 20, Cambridge Univ. Press, 1983, ISBN 0 521 30240 4
++ J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++ AXIOM Technical Report Series, ATR/5 NP2522.
++ Description:
++ FiniteFieldNormalBasis(p,n) implements a
++ finite extension field of degree n over the prime field with p elements.
++ The elements are represented by coordinate vectors with respect to
++ a normal basis,

```

```

++ i.e. a basis consisting of the conjugates (q-powers) of an element, in
++ this case called normal element.
++ This is chosen as a root of the extension polynomial
++ created by createNormalPoly

FiniteFieldNormalBasis(p,extdeg):_
  Exports == Implementation where
  p : PositiveInteger
  extdeg: PositiveInteger          -- the extension degree
  NNI    ==> NonNegativeInteger
  FFF    ==> FiniteFieldFunctions(PrimeField(p))
  TERM   ==> Record(value:PrimeField(p),index:SingleInteger)
  Exports ==> FiniteAlgebraicExtensionField(PrimeField(p)) with
    getMultiplicationTable: ()  -> Vector List TERM
    ++ getMultiplicationTable() returns the multiplication
    ++ table for the normal basis of the field.
    ++ This table is used to perform multiplications between field elements.
    getMultiplicationMatrix: () -> Matrix PrimeField(p)
    ++ getMultiplicationMatrix() returns the multiplication table in
    ++ form of a matrix.
    sizeMultiplication:()      -> NNI
    ++ sizeMultiplication() returns the number of entries in the
    ++ multiplication table of the field. Note: The time of multiplication
    ++ of field elements depends on this size.

  Implementation ==> FiniteFieldNormalBasisExtensionByPolynomial(PrimeField(p),_
    createLowComplexityNormalBasis(extdeg)$FFF)

-----

— FFNB.dotabb —

"FFNB" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FFNB"]
"FAXF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAXF"]
"FFNB" -> "FAXF"

-----

```

domain FFNBX FiniteFieldNormalBasisExtension

— FiniteFieldNormalBasisExtension.input —

```

)set break resume
)sys rm -f FiniteFieldNormalBasisExtension.output

```



```

)spool FiniteFieldNormalBasisExtension.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteFieldNormalBasisExtension
--R FiniteFieldNormalBasisExtension(GF: FiniteFieldCategory,extdeg: PositiveInteger) is a domain
--R Abbreviation for FiniteFieldNormalBasisExtension is FFNBX
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FFNBX
--R
--R----- Operations -----
--R ???: (GF,%) -> %          ??? : (%,GF) -> %
--R ???: (Fraction(Integer),%) -> %    ??? : (%,Fraction(Integer)) -> %
--R ???: (%,%) -> %            ??? : (Integer,%) -> %
--R ???: (PositiveInteger,%) -> %      ??? : (%,Integer) -> %
--R ??? : (%,PositiveInteger) -> %    ?+?: (%,%) -> %
--R ?-?: (%,%) -> %             -?: % -> %
--R ?/? : (%,GF) -> %           ?/? : (%,%) -> %
--R ?=: (%,%) -> Boolean         D : % -> % if GF has FINITE
--R 1 : () -> %                 0 : () -> %
--R ?? : (%,Integer) -> %       ??? : (%,PositiveInteger) -> %
--R algebraic? : % -> Boolean    associates? : (%,%) -> Boolean
--R basis : () -> Vector(%)      coerce : GF -> %
--R coerce : Fraction(Integer) -> %  coerce : % -> %
--R coerce : Integer -> %        coerce : % -> OutputForm
--R coordinates : % -> Vector(GF)  degree : % -> PositiveInteger
--R dimension : () -> CardinalNumber  factor : % -> Factored(%)
--R gcd : List(%) -> %           gcd : (%,%) -> %
--R hash : % -> SingleInteger      inGroundField? : % -> Boolean
--R inv : % -> %                  latex : % -> String
--R lcm : List(%) -> %            lcm : (%,%) -> %
--R norm : % -> GF                one? : % -> Boolean
--R prime? : % -> Boolean          ?quo? : (%,%) -> %
--R recip : % -> Union(%, "failed") ?rem? : (%,%) -> %
--R represents : Vector(GF) -> %    retract : % -> GF
--R sample : () -> %              sizeLess? : (%,%) -> Boolean
--R squareFree : % -> Factored(%)  squareFreePart : % -> %
--R trace : % -> GF                transcendent? : % -> Boolean
--R unit? : % -> Boolean           unitCanonical : % -> %
--R zero? : % -> Boolean           ?~=? : (%,%) -> Boolean
--R ??? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R D : (%,NonNegativeInteger) -> % if GF has FINITE
--R Frobenius : (%,NonNegativeInteger) -> % if GF has FINITE
--R Frobenius : % -> % if GF has FINITE
--R ?? : (%,NonNegativeInteger) -> %
--R basis : PositiveInteger -> Vector(%)
--R characteristic : () -> NonNegativeInteger

```

```

--R charthRoot : % -> Union(%, "failed") if GF has CHARNZ or GF has FINITE
--R charthRoot : % -> % if GF has FINITE
--R conditionP : Matrix(%) -> Union(Vector(%), "failed") if GF has FINITE
--R coordinates : Vector(%) -> Matrix(GF)
--R createNormalElement : () -> % if GF has FINITE
--R createPrimitiveElement : () -> % if GF has FINITE
--R definingPolynomial : () -> SparseUnivariatePolynomial(GF)
--R degree : % -> OnePointCompletion(PositiveInteger)
--R differentiate : (%, NonNegativeInteger) -> % if GF has FINITE
--R differentiate : % -> % if GF has FINITE
--R discreteLog : (%, %) -> Union(NonNegativeInteger, "failed") if GF has CHARNZ or GF has FINITE
--R discreteLog : % -> NonNegativeInteger if GF has FINITE
--R divide : (%, %) -> Record(quotient: %, remainder: %)
--R enumerate : () -> List(%) if GF has FINITE
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(%), %) -> Union(List(%), "failed")
--R exquo : (%, %) -> Union(%, "failed")
--R extendedEuclidean : (%, %, %) -> Union(Record(coef1: %, coef2: %), "failed")
--R extendedEuclidean : (%, %) -> Record(coef1: %, coef2: %, generator: %)
--R extensionDegree : () -> PositiveInteger
--R extensionDegree : () -> OnePointCompletion(PositiveInteger)
--R factorsOfCyclicGroupSize : () -> List(Record(factor: Integer, exponent: Integer)) if GF has FINITE
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(GF)
--R generator : () -> % if GF has FINITE
--R getMultiplicationMatrix : () -> Matrix(GF)
--R getMultiplicationTable : () -> Vector(List(Record(value: GF, index: SingleInteger)))
--R index : PositiveInteger -> % if GF has FINITE
--R init : () -> % if GF has FINITE
--R linearAssociatedExp : (%, SparseUnivariatePolynomial(GF)) -> % if GF has FINITE
--R linearAssociatedLog : (%, %) -> Union(SparseUnivariatePolynomial(GF), "failed") if GF has FINITE
--R linearAssociatedLog : % -> SparseUnivariatePolynomial(GF) if GF has FINITE
--R linearAssociatedOrder : % -> SparseUnivariatePolynomial(GF) if GF has FINITE
--R lookup : % -> PositiveInteger if GF has FINITE
--R minimalPolynomial : (%, PositiveInteger) -> SparseUnivariatePolynomial(%) if GF has FINITE
--R minimalPolynomial : % -> SparseUnivariatePolynomial(GF)
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed")
--R nextItem : % -> Union(%, "failed") if GF has FINITE
--R norm : (%, PositiveInteger) -> % if GF has FINITE
--R normal? : % -> Boolean if GF has FINITE
--R normalElement : () -> % if GF has FINITE
--R order : % -> OnePointCompletion(PositiveInteger) if GF has CHARNZ or GF has FINITE
--R order : % -> PositiveInteger if GF has FINITE
--R primeFrobenius : % -> % if GF has CHARNZ or GF has FINITE
--R primeFrobenius : (%, NonNegativeInteger) -> % if GF has CHARNZ or GF has FINITE
--R primitive? : % -> Boolean if GF has FINITE
--R primitiveElement : () -> % if GF has FINITE
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %)
--R random : () -> % if GF has FINITE
--R representationType : () -> Union("prime", "polynomial", "normal", "cyclic") if GF has FINITE
--R retractIfCan : % -> Union(GF, "failed")

```

```

--R size : () -> NonNegativeInteger if GF has FINITE
--R sizeMultiplication : () -> NonNegativeInteger
--R subtractIfCan : (%,%) -> Union(%,"failed")
--R tableForDiscreteLogarithm : Integer -> Table(PositiveInteger,NonNegativeInteger) if GF has F
--R trace : (%,PositiveInteger) -> % if GF has FINITE
--R transcendenceDegree : () -> NonNegativeInteger
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R
--E 1

```

```

)spool
)lisp (bye)

```

— FiniteFieldNormalBasisExtension.help —

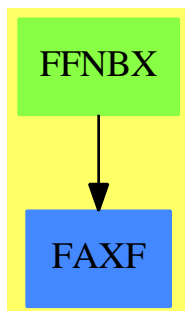
```

=====
FiniteFieldNormalBasisExtension examples
=====

```

FiniteFieldNormalBasisExtensionByPolynomial(GF,n) implements a finite extension field of degree n over the ground field GF. The elements are represented by coordinate vectors with respect to a normal basis, i.e. a basis consisting of the conjugates (q-powers) of an element, in this case called normal element. This is chosen as a root of the extension polynomial, created by createNormalPoly from FiniteFieldPolynomialPackage

See Also:
o)show FiniteFieldNormalBasisExtension

FiniteFieldNormalBasisExtension (FFNBX)

See

⇒ “FiniteFieldNormalBasisExtensionByPolynomial” (FFNBP) [7](#) on page [871](#)

⇒ “FiniteFieldNormalBasis” (FFNB) [7](#) on page [859](#)

Exports:

0	1	algebraic?
associates?	basis	characteristic
charthRoot	coerce	conditionP
coordinates	createNormalElement	createPrimitiveElement
D	definingPolynomial	degree
differentiate	dimension	discreteLog
divide	euclideanSize	expressIdealMember
exquo	extendedEuclidean	extensionDegree
factor	factorsOfCyclicGroupSize	Frobenius
gcd	gcdPolynomial	generator
getMultiplicationMatrix	getMultiplicationTable	hash
index	inGroundField?	init
inv	latex	lcm
linearAssociatedExp	linearAssociatedLog	linearAssociatedOrder
lookup	minimalPolynomial	multiEuclidean
nextItem	norm	normal?
normalElement	one?	order
prime?	primeFrobenius	primitive?
primitiveElement	principalIdeal	random
recip	representationType	represents
retract	retractIfCan	sample
size	sizeLess?	sizeMultiplication
squareFree	squareFreePart	subtractIfCan
tableForDiscreteLogarithm	trace	transcendenceDegree
transcendent?	unit?	unitCanonical
unitNormal	zero?	?*?
***?	?+?	?-?
-?	?/?	?=?
?^?	?~=?	?quo?
?rem?		

— domain FFNBX FiniteFieldNormalBasisExtension —

```

)abbrev domain FFNBX FiniteFieldNormalBasisExtension
++ Authors: J.Grabmeier, A.Scheerhorn
++ Date Created: 26.03.1991
++ References:
++ R.Lidl, H.Niederreiter: Finite Field, Encyclopedia of Mathematics and
++ Its Applications, Vol. 20, Cambridge Univ. Press, 1983, ISBN 0 521 30240 4
++ J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++ AXIOM Technical Report Series, ATR/5 NP2522.
++ Description:
++ FiniteFieldNormalBasisExtensionByPolynomial(GF,n) implements a
++ finite extension field of degree n over the ground field GF.
++ The elements are represented by coordinate vectors with respect
++ to a normal basis,

```

```

++ i.e. a basis consisting of the conjugates (q-powers) of an element, in
++ this case called normal element. This is chosen as a root of the extension
++ polynomial, created by createNormalPoly from
++ \spadtype{FiniteFieldPolynomialPackage}

```

```

FiniteFieldNormalBasisExtension(GF,extdeg):_
  Exports == Implementation where
  GF      : FiniteFieldCategory          -- the ground field
  extdeg: PositiveInteger                -- the extension degree
  NNI      ==> NonNegativeInteger
  FFF      ==> FiniteFieldFunctions(GF)
  TERM     ==> Record(value:GF,index:SingleInteger)
  Exports ==> FiniteAlgebraicExtensionField(GF) with
    getMultiplicationTable: () -> Vector List TERM
      ++ getMultiplicationTable() returns the multiplication
      ++ table for the normal basis of the field.
      ++ This table is used to perform multiplications between field elements.
    getMultiplicationMatrix: () -> Matrix GF
      ++ getMultiplicationMatrix() returns the multiplication table in
      ++ form of a matrix.
    sizeMultiplication:() -> NNI
      ++ sizeMultiplication() returns the number of entries in the
      ++ multiplication table of the field. Note: the time of multiplication
      ++ of field elements depends on this size.

  Implementation ==> FiniteFieldNormalBasisExtensionByPolynomial(GF,_
    createLowComplexityNormalBasis(extdeg)$FFF)

```

— FFNBX.dotabb —

```

"FFNBX" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FFNBX"]
"FAXF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAXF"]
"FFNBX" -> "FAXF"

```

domain FFNBP FiniteFieldNormalBasisExtensionByPolynomial

— FiniteFieldNormalBasisExtensionByPolynomial.input —

```

)set break resume

```

```

)sys rm -f FiniteFieldNormalBasisExtensionByPolynomial.output
)spool FiniteFieldNormalBasisExtensionByPolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteFieldNormalBasisExtensionByPolynomial
--R FiniteFieldNormalBasisExtensionByPolynomial(GF: FiniteFieldCategory, uni: Union(SparseUnivari
--R Abbreviation for FiniteFieldNormalBasisExtensionByPolynomial is FFNBP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FFNBP
--R
--R----- Operations -----
--R ???: (GF,%) -> %          ??? : (%,GF) -> %
--R ??? : (Fraction(Integer),%) -> %    ??? : (%,Fraction(Integer)) -> %
--R ??? : (%,%) -> %          ??? : (Integer,%) -> %
--R ??? : (PositiveInteger,%) -> %      ??? : (%,Integer) -> %
--R ??? : (%,PositiveInteger) -> %     ?+? : (%,%) -> %
--R ?-? : (%,%) -> %          -? : % -> %
--R ?/? : (%,GF) -> %         ?/? : (%,%) -> %
--R ?=? : (%,%) -> Boolean      D : % -> % if GF has FINITE
--R 1 : () -> %               0 : () -> %
--R ?^? : (%,Integer) -> %     ?^? : (%,PositiveInteger) -> %
--R algebraic? : % -> Boolean   associates? : (%,%) -> Boolean
--R basis : () -> Vector(%)    coerce : GF -> %
--R coerce : Fraction(Integer) -> %   coerce : % -> %
--R coerce : Integer -> %        coerce : % -> OutputForm
--R coordinates : % -> Vector(GF)    degree : % -> PositiveInteger
--R dimension : () -> CardinalNumber  factor : % -> Factored(%)
--R gcd : List(%) -> %          gcd : (%,%) -> %
--R hash : % -> SingleInteger    inGroundField? : % -> Boolean
--R inv : % -> %               latex : % -> String
--R lcm : List(%) -> %         lcm : (%,%) -> %
--R norm : % -> GF             one? : % -> Boolean
--R prime? : % -> Boolean       ?quo? : (%,%) -> %
--R recip : % -> Union(%, "failed")  ?rem? : (%,%) -> %
--R represents : Vector(GF) -> %    retract : % -> GF
--R sample : () -> %           sizeLess? : (%,%) -> Boolean
--R squareFree : % -> Factored(%)   squareFreePart : % -> %
--R trace : % -> GF             transcendent? : % -> Boolean
--R unit? : % -> Boolean         unitCanonical : % -> %
--R zero? : % -> Boolean        ?^=? : (%,%) -> Boolean
--R ??? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R D : (%,NonNegativeInteger) -> % if GF has FINITE
--R Frobenius : (%,NonNegativeInteger) -> % if GF has FINITE
--R Frobenius : % -> % if GF has FINITE
--R ?^? : (%,NonNegativeInteger) -> %
--R basis : PositiveInteger -> Vector(%)

```

```

--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if GF has CHARNZ or GF has FINITE
--R charthRoot : % -> % if GF has FINITE
--R conditionP : Matrix(%) -> Union(Vector(%), "failed") if GF has FINITE
--R coordinates : Vector(%) -> Matrix(GF)
--R createNormalElement : () -> % if GF has FINITE
--R createPrimitiveElement : () -> % if GF has FINITE
--R definingPolynomial : () -> SparseUnivariatePolynomial(GF)
--R degree : % -> OnePointCompletion(PositiveInteger)
--R differentiate : (%, NonNegativeInteger) -> % if GF has FINITE
--R differentiate : % -> % if GF has FINITE
--R discreteLog : (%, %) -> Union(NonNegativeInteger, "failed") if GF has CHARNZ or GF has FINITE
--R discreteLog : % -> NonNegativeInteger if GF has FINITE
--R divide : (%, %) -> Record(quotient: %, remainder: %)
--R enumerate : () -> List(%) if GF has FINITE
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(%), %) -> Union(List(%), "failed")
--R exquo : (%, %) -> Union(%, "failed")
--R extendedEuclidean : (%, %, %) -> Union(Record(coef1: %, coef2: %), "failed")
--R extendedEuclidean : (%, %) -> Record(coef1: %, coef2: %, generator: %)
--R extensionDegree : () -> PositiveInteger
--R extensionDegree : () -> OnePointCompletion(PositiveInteger)
--R factorsOfCyclicGroupSize : () -> List(Record(factor: Integer, exponent: Integer)) if GF has FINITE
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(GF)
--R generator : () -> % if GF has FINITE
--R getMultiplicationMatrix : () -> Matrix(GF)
--R getMultiplicationTable : () -> Vector(List(Record(value: GF, index: SingleInteger)))
--R index : PositiveInteger -> % if GF has FINITE
--R init : () -> % if GF has FINITE
--R linearAssociatedExp : (%, SparseUnivariatePolynomial(GF)) -> % if GF has FINITE
--R linearAssociatedLog : (%, %) -> Union(SparseUnivariatePolynomial(GF), "failed") if GF has FINITE
--R linearAssociatedLog : % -> SparseUnivariatePolynomial(GF) if GF has FINITE
--R linearAssociatedOrder : % -> SparseUnivariatePolynomial(GF) if GF has FINITE
--R lookup : % -> PositiveInteger if GF has FINITE
--R minimalPolynomial : (%, PositiveInteger) -> SparseUnivariatePolynomial(%) if GF has FINITE
--R minimalPolynomial : % -> SparseUnivariatePolynomial(GF)
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed")
--R nextItem : % -> Union(%, "failed") if GF has FINITE
--R norm : (%, PositiveInteger) -> % if GF has FINITE
--R normal? : % -> Boolean if GF has FINITE
--R normalElement : () -> % if GF has FINITE
--R order : % -> OnePointCompletion(PositiveInteger) if GF has CHARNZ or GF has FINITE
--R order : % -> PositiveInteger if GF has FINITE
--R primeFrobenius : % -> % if GF has CHARNZ or GF has FINITE
--R primeFrobenius : (%, NonNegativeInteger) -> % if GF has CHARNZ or GF has FINITE
--R primitive? : % -> Boolean if GF has FINITE
--R primitiveElement : () -> % if GF has FINITE
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %)
--R random : () -> % if GF has FINITE
--R representationType : () -> Union("prime", "polynomial", "normal", "cyclic") if GF has FINITE

```



```

--R retractIfCan : % -> Union(GF,"failed")
--R size : () -> NonNegativeInteger if GF has FINITE
--R sizeMultiplication : () -> NonNegativeInteger
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R tableForDiscreteLogarithm : Integer -> Table(PositiveInteger,NonNegativeInteger) if GF has F
--R trace : (%,PositiveInteger) -> % if GF has FINITE
--R transcendenceDegree : () -> NonNegativeInteger
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R
--E 1

)spool
)lisp (bye)

```

— FiniteFieldNormalBasisExtensionByPolynomial.help —

```

=====
FiniteFieldNormalBasisExtensionByPolynomial examples
=====

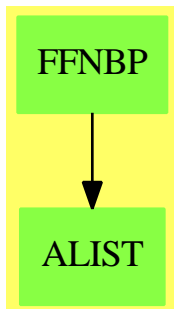
```

FiniteFieldNormalBasisExtensionByPolynomial(GF,uni) implements a finite extension of the ground field GF. The elements are represented by coordinate vectors with respect to. a normal basis, i.e. a basis consisting of the conjugates (q-powers) of an element, in this case called normal element, where q is the size of GF. The normal element is chosen as a root of the extension polynomial, which MUST be normal over GF (user responsibility)

See Also:

o)show FiniteFieldNormalBasisExtensionByPolynomial

FiniteFieldNormalBasisExtensionByPolynomial (FFNBP)



See

⇒ “FiniteFieldNormalBasisExtension” (FFNBX) [7](#) on page [865](#)

⇒ “FiniteFieldNormalBasis” (FFNB) [7](#) on page [859](#)

Exports:

0	1	algebraic?
associates?	basis	characteristic
charthRoot	coerce	conditionP
coordinates	createNormalElement	createPrimitiveElement
D	definingPolynomial	degree
differentiate	dimension	discreteLog
divide	euclideanSize	expressIdealMember
exquo	extendedEuclidean	extensionDegree
factor	factorsOfCyclicGroupSize	Frobenius
gcd	gcdPolynomial	generator
getMultiplicationMatrix	getMultiplicationTable	hash
index	inGroundField?	init
inv	latex	lcm
linearAssociatedExp	linearAssociatedLog	linearAssociatedOrder
lookup	minimalPolynomial	multiEuclidean
nextItem	norm	normal?
normalElement	one?	order
prime?	primeFrobenius	primitive?
primitiveElement	principalIdeal	recip
random	representationType	represents
retract	retractIfCan	sample
size	sizeLess?	sizeMultiplication
squareFree	squareFreePart	subtractIfCan
tableForDiscreteLogarithm	trace	transcendenceDegree
transcendent?	unit?	unitCanonical
unitNormal	zero?	?*?
***?	?+?	?-?
-?	?/?	?=?
?^?	?~=?	?quo?
?rem?		

— domain **FFNBP** **FiniteFieldNormalBasisExtensionByPolynomial**

```

)abbrev domain FFNBP FiniteFieldNormalBasisExtensionByPolynomial
++ Authors: J.Grabmeier, A.Scheerhorn
++ Date Created: 26.03.1991
++ Date Last Updated: 08 May 1991
++ References:
++ R.Lidl, H.Niederreiter: Finite Field, Encyclopedia of Mathematics and
++ Its Applications, Vol. 20, Cambridge Univ. Press, 1983, ISBN 0 521 30240 4
++ J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM .
++ AXIOM Technical Report Series, ATR/5 NP2522.
++ Description:
++ FiniteFieldNormalBasisExtensionByPolynomial(GF,uni) implements a
++ finite extension of the ground field GF. The elements are

```

```

++ represented by coordinate vectors with respect to. a normal basis,
++ i.e. a basis
++ consisting of the conjugates (q-powers) of an element, in this case
++ called normal element, where q is the size of GF.
++ The normal element is chosen as a root of the extension
++ polynomial, which MUST be normal over GF (user responsibility)

FiniteFieldNormalBasisExtensionByPolynomial(GF,uni): Exports == _
  Implementation where
  GF      : FiniteFieldCategory          -- the ground field
  uni     : Union(SparseUnivariatePolynomial GF, _
                  Vector List Record(value:GF,index:SingleInteger))

  PI  ==> PositiveInteger
  NNI ==> NonNegativeInteger
  I   ==> Integer
  SI  ==> SingleInteger
  SUP ==> SparseUnivariatePolynomial
  V   ==> Vector GF
  M   ==> Matrix GF
  OUT ==> OutputForm
  TERM ==> Record(value:GF,index:SI)
  R    ==> Record(key:PI,entry:NNI)
  TBL  ==> Table(PI,NNI)
  FFF  ==> FiniteFieldFunctions(GF)
  INBFF ==> InnerNormalBasisFieldFunctions(GF)

Exports ==> FiniteAlgebraicExtensionField(GF) with

  getMultiplicationTable: ()    -> Vector List TERM
    ++ getMultiplicationTable() returns the multiplication
    ++ table for the normal basis of the field.
    ++ This table is used to perform multiplications between field elements.
  getMultiplicationMatrix:()    -> M
    ++ getMultiplicationMatrix() returns the multiplication table in
    ++ form of a matrix.
  sizeMultiplication:()        -> NNI
    ++ sizeMultiplication() returns the number of entries in the
    ++ multiplication table of the field.
    ++ Note: the time of multiplication
    ++ of field elements depends on this size.
  Implementation ==> add

-- global variables =====

  Rep:= V      -- elements are represented by vectors over GF

  alpha      :=new()$Symbol :: OutputForm
  -- get a new Symbol for the output representation of the elements

```

```

initlog?:Boolean:=true
-- gets false after initialization of the logarithm table

initelt?:Boolean:=true
-- gets false after initialization of the primitive element

initmult?:Boolean:=true
-- gets false after initialization of the multiplication
-- table or the primitive element

extdeg:PI :=1

defpol:SUP(GF):=0$SUP(GF)
-- the defining polynomial

multTable:Vector List TERM:=new(1,nil())$(List TERM))
-- global variable containing the multiplication table

if uni case (Vector List TERM) then
  multTable:=uni :: (Vector List TERM)
  extdeg:= (#multTable) pretend PI
  vv:V:=new(extdeg,0)$V
  vv.1:=1$GF
  setFieldInfo(multTable,1$GF)$INBFF
  defpol:=minimalPolynomial(vv)$INBFF
  initmult?:=false
else
  defpol:=uni :: SUP(GF)
  extdeg:=degree(defpol)$(SUP GF) pretend PI
  multTable:Vector List TERM:=new(extdeg,nil())$(List TERM))

basisOutput : List OUT :=
  qs:OUT:=(q::Symbol)::OUT
  append([alpha, alpha **$OUT qs],_
    [alpha **$OUT (qs **$OUT i::OUT) for i in 2..extdeg-1] )

facOfGroupSize :=nil()$(List Record(factor:Integer,exponent:Integer))
-- the factorization of the cyclic group size

traceAlpha:GF:=-$GF coefficient(defpol,(degree(defpol)-1)::NNI)
-- the inverse of the trace of the normalElt
-- is computed here. It defines the imbedding of
-- GF in the extension field

primitiveElt:PI:=1
-- for the lookup the primitive Element computed by createPrimitiveElement()

discLogTable:Table(PI,TBL):=table()$Table(PI,TBL)

```

```

-- tables indexed by the factors of sizeCG,
-- discLogTable(factor) is a table with keys
-- primitiveElement() ** (i * (sizeCG quo factor)) and entries i for
-- i in 0..n-1, n computed in initialize() in order to use
-- the minimal size limit 'limit' optimal.

-- functions =====

initializeLog: ()      -> Void
initializeElt: ()      -> Void
initializeMult: ()     -> Void

coerce(v:GF):$ == new(extdeg,v /$GF traceAlpha)$Rep
represents(v)  == v::$

degree(a) ==
  d:PI:=1
  b:= qPot(a::Rep,1)$INBFF
  while (b^=a) repeat
    b:= qPot(b::Rep,1)$INBFF
    d:=d+1
  d

linearAssociatedExp(x,f) ==
  xm:SUP(GF):=monomial(1$GF,extdeg)$(SUP GF) - 1$(SUP GF)
  r:= (f * pol(x::Rep)$INBFF) rem xm
  vectorise(r,extdeg)$(SUP GF)
linearAssociatedLog(x) == pol(x::Rep)$INBFF
linearAssociatedOrder(x) ==
  xm:SUP(GF):=monomial(1$GF,extdeg)$(SUP GF) - 1$(SUP GF)
  xm quo gcd(xm,pol(x::Rep)$INBFF)
linearAssociatedLog(b,x) ==
  zero? x => 0
  xm:SUP(GF):=monomial(1$GF,extdeg)$(SUP GF) - 1$(SUP GF)
  e:= extendedEuclidean(pol(b::Rep)$INBFF,xm,pol(x::Rep)$INBFF)$(SUP GF)
  e = "failed" => "failed"
  e1:= e :: Record(coef1:(SUP GF),coef2:(SUP GF))
  e1.coef1

getMultiplicationTable() ==
  if initmult? then initializeMult()
  multTable
getMultiplicationMatrix() ==
  if initmult? then initializeMult()
  createMultiplicationMatrix(multTable)$FFF
sizeMultiplication() ==
  if initmult? then initializeMult()
  sizeMultiplication(multTable)$FFF

```

```

trace(a:$) == retract trace(a,1)
norm(a:$) == retract norm(a,1)
generator() == normalElement(extdeg)$INBFF
basis(n:PI) ==
  (extdeg rem n) ^= 0 => error "argument must divide extension degree"
  [Frobenius(trace(normalElement,n),i) for i in 0..(n-1)]::(Vector $)

a:GF * x:$ == a *$Rep x

x:$/a:GF == x/coerce(a)
-- x:$ / a:GF ==
-- a = 0$GF => error "division by zero"
-- x * inv(coerce(a))

coordinates(x:$) == x::Rep

Frobenius(e) == qPot(e::Rep,1)$INBFF
Frobenius(e,n) == qPot(e::Rep,n)$INBFF

retractIfCan(x) ==
  inGroundField?(x) =>
    x.1 *$GF traceAlpha
    "failed"

retract(x) ==
  inGroundField?(x) =>
    x.1 *$GF traceAlpha
    error("element not in ground field")

-- to get a "normal basis like" output form
coerce(x:$):OUT ==
  l:List OUT:=nil()$(List OUT)
  n : PI := extdeg
-- one? n => (x.1) :: OUT
  (n = 1) => (x.1) :: OUT
  for i in 1..n for b in basisOutput repeat
    if not zero? x.i then
      mon : OUT :=
-- one? x.i => b
      (x.i = 1) => b
      ((x.i)::OUT) *$OUT b
      l:=cons(mon,l)$(List OUT)
  null(l)$(List OUT) => (0::OUT)
  r:=reduce("+",l)$(List OUT)
  r

initializeElt() ==
  facOfGroupSize := factors factor(size())$GF**extdeg-1)$I
  -- get a primitive element

```

```

primitiveElt:=lookup(createPrimitiveElement())
initelt?:=false
void()$Void

initializeMult() ==
multTable:=createMultiplicationTable(defpol)$FFF
setFieldInfo(multTable,traceAlpha)$INBFF
-- reset initialize flag
initmult?:=false
void()$Void

initializeLog() ==
if initelt? then initializeElt()
-- set up tables for discrete logarithm
limit:Integer:=30
-- the minimum size for the discrete logarithm table
for f in facOfGroupSize repeat
  fac:=f.factor
  base:=index(primitiveElt)**((size())$GF**extdeg -$I 1$I) quo$I fac)
  l:Integer:=length(fac)$Integer
  n:Integer:=0
  if odd?(l)$I then n:=shift(fac,-$I (l quo$I 2))$I
    else n:=shift(1,l quo$I 2)$I
  if n < $I limit then
    d:=(fac -$I 1$I) quo$I limit +$I 1$I
    n:=(fac -$I 1$I) quo$I d +$I 1$I
  tbl:TBL:=table()$TBL
  a:=$=1
  for i in (0::NNI)..(n-1)::NNI repeat
    insert_!([lookup(a),i::NNI]$R,tbl)$TBL
    a:=a*base
  insert_!([fac::PI,copy(tbl)$TBL]_
    $Record(key:PI,entry:TBL),discLogTable)$Table(PI,TBL)
initlog?:=false
-- tell user about initialization
--print("discrete logarithm table initialized":OUT)
void()$Void

tableForDiscreteLogarithm(fac) ==
if initlog? then initializeLog()
tbl:=search(fac::PI,discLogTable)$Table(PI,TBL)
tbl case "failed" =>
  error "tableForDiscreteLogarithm: argument must be prime _
divisor of the order of the multiplicative group"
tbl :: TBL

primitiveElement() ==
if initelt? then initializeElt()
index(primitiveElt)

```



```

factorsOfCyclicGroupSize() ==
  if empty? facOfGroupSize then initializeElt()
  facOfGroupSize

extensionDegree() == extdeg

sizeOfGroundField() == size()$GF pretend NNI

definingPolynomial() == defpol

trace(a,d) ==
  v:=trace(a::Rep,d)$INBFF
  erg:=v
  for i in 2..(extdeg quo d) repeat
    erg:=concat(erg,v)$Rep
  erg

characteristic() == characteristic()$GF

random() == random(extdeg)$INBFF

x:$ * y:$ ==
  if initmult? then initializeMult()
  setFieldInfo(multTable,traceAlpha)$INBFF
  x::Rep *$INBFF y::Rep

1 == new(extdeg,inv(traceAlpha)$GF)$Rep

0 == zero(extdeg)$Rep

size() == size()$GF ** extdeg

index(n:PI) == index(extdeg,n)$INBFF

lookup(x:$) == lookup(x::Rep)$INBFF

basis() ==
  a:=basis(extdeg)$INBFF
  vector([e::$ for e in entries a])

x:$ ** e:I ==
  if initmult? then initializeMult()
  setFieldInfo(multTable,traceAlpha)$INBFF
  (x::Rep) **$INBFF e

normal?(x) == normal?(x::Rep)$INBFF

```

```

-(x:$) == -$Rep x
x:$ + y:$ == x +$Rep y
x:$ - y:$ == x -$Rep y
x:$ = y:$ == x =$Rep y
n:I * x:$ == x *$Rep (n::GF)

representationType() == "normal"

minimalPolynomial(a) ==
  if initmult? then initializeMult()
  setFieldInfo(multTable,traceAlpha)$INBFF
  minimalPolynomial(a::Rep)$INBFF

-- is x an element of the ground field GF ?
inGroundField?(x) ==
  erg:=true
  for i in 2..extdeg repeat
    not(x.i =$GF x.1) => erg:=false
  erg

x:$ / y:$ ==
  if initmult? then initializeMult()
  setFieldInfo(multTable,traceAlpha)$INBFF
  x::Rep /$INBFF y::Rep

inv(a) ==
  if initmult? then initializeMult()
  setFieldInfo(multTable,traceAlpha)$INBFF
  inv(a::Rep)$INBFF

norm(a,d) ==
  if initmult? then initializeMult()
  setFieldInfo(multTable,traceAlpha)$INBFF
  norm(a::Rep,d)$INBFF

normalElement() == normalElement(extdeg)$INBFF

```

— FFNBP.dotabb —

```

"FFNBP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FFNBP"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"FFNBP" -> "ALIST"

```

domain FARRAY FlexibleArray

— FlexibleArray.input —

```

)set break resume
)sys rm -f FlexibleArray.output
)spool FlexibleArray.output
)set message test on
)set message auto off
)clear all
--S 1 of 16
flexibleArray [i for i in 1..6]
--R
--R
--R (1) [1,2,3,4,5,6]
--R
--R                                          Type: FlexibleArray(PositiveInteger)
--E 1

--S 2 of 16
f : FARRAY INT := new(6,0)
--R
--R
--R (2) [0,0,0,0,0,0]
--R
--R                                          Type: FlexibleArray(Integer)
--E 2

--S 3 of 16
for i in 1..6 repeat f.i := i; f
--R
--R
--R (3) [1,2,3,4,5,6]
--R
--R                                          Type: FlexibleArray(Integer)
--E 3

--S 4 of 16
physicalLength f
--R
--R
--R (4) 6
--R
--R                                          Type: PositiveInteger
--E 4

--S 5 of 16
concat!(f,11)
--R

```

```

--R
--R (5) [1,2,3,4,5,6,11]
--R                                         Type: FlexibleArray(Integer)
--E 5

--S 6 of 16
physicalLength f
--R
--R
--R (6) 10
--R                                         Type: PositiveInteger
--E 6

--S 7 of 16
physicalLength!(f,15)
--R
--R
--R (7) [1,2,3,4,5,6,11]
--R                                         Type: FlexibleArray(Integer)
--E 7

--S 8 of 16
concat!(f,f)
--R
--R
--R (8) [1,2,3,4,5,6,11,1,2,3,4,5,6,11]
--R                                         Type: FlexibleArray(Integer)
--E 8

--S 9 of 16
insert!(22,f,1)
--R
--R
--R (9) [22,1,2,3,4,5,6,11,1,2,3,4,5,6,11]
--R                                         Type: FlexibleArray(Integer)
--E 9

--S 10 of 16
g := f(10..)
--R
--R
--R (10) [2,3,4,5,6,11]
--R                                         Type: FlexibleArray(Integer)
--E 10

--S 11 of 16
insert!(g,f,1)
--R
--R
--R (11) [2,3,4,5,6,11,22,1,2,3,4,5,6,11,1,2,3,4,5,6,11]

```

```

--R                                          Type: FlexibleArray(Integer)
--E 11

--S 12 of 16
merge!(sort! f, sort! g)
--R
--R
--R (12) [1,1,2,2,2,2,3,3,3,3,4,4,4,4,5,5,5,5,6,6,6,6,11,11,11,11,22]
--R                                          Type: FlexibleArray(Integer)
--E 12

--S 13 of 16
removeDuplicates! f
--R
--R
--R (13) [1,2,3,4,5,6,11,22]
--R                                          Type: FlexibleArray(Integer)
--E 13

--S 14 of 16
select!(i +-> even? i,f)
--R
--R
--R (14) [2,4,6,22]
--R                                          Type: FlexibleArray(Integer)
--E 14

--S 15 of 16
physicalLength f
--R
--R
--R (15) 8
--R                                          Type: PositiveInteger
--E 15

--S 16 of 16
shrinkable(false)$FlexibleArray(Integer)
--R
--R
--R (16) true
--R                                          Type: Boolean
--E 16
)spool
)lisp (bye)

```


See that its physical length has grown.

```
physicalLength f
10
Type: PositiveInteger
```

Make f grow to have room for 15 elements.

```
physicalLength!(f,15)
[1,2,3,4,5,6,11]
Type: FlexibleArray Integer
```

Concatenate the elements of f to itself. The physical length allows room for three more values at the end.

```
concat!(f,f)
[1,2,3,4,5,6,11,1,2,3,4,5,6,11]
Type: FlexibleArray Integer
```

Use insert! to add an element to the front of a flexible array.

```
insert!(22,f,1)
[22,1,2,3,4,5,6,11,1,2,3,4,5,6,11]
Type: FlexibleArray Integer
```

Create a second flexible array from f consisting of the elements from index 10 forward.

```
g := f(10..)
[2,3,4,5,6,11]
Type: FlexibleArray Integer
```

Insert this array at the front of f.

```
insert!(g,f,1)
[2,3,4,5,6,11,22,1,2,3,4,5,6,11,1,2,3,4,5,6,11]
Type: FlexibleArray Integer
```

Merge the flexible array f into g after sorting each in place.

```
merge!(sort! f, sort! g)
[1,1,2,2,2,2,3,3,3,3,4,4,4,4,5,5,5,5,6,6,6,6,11,11,11,11,22]
Type: FlexibleArray Integer
```

Remove duplicates in place.

```
removeDuplicates! f
[1,2,3,4,5,6,11,22]
Type: FlexibleArray Integer
```

Remove all odd integers.

```
select!(i +-> even? i,f)
[2,4,6,22]
Type: FlexibleArray Integer
```

All these operations have shrunk the physical length of f.

```
physicalLength f
8
Type: PositiveInteger
```

To force Axiom not to shrink flexible arrays call the shrinkable operation with the argument false. You must package call this operation. The previous value is returned.

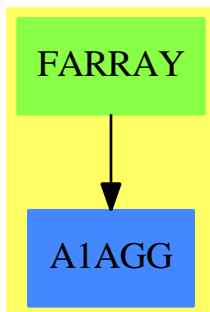
```
shrinkable(false)$FlexibleArray(Integer)
true
Type: Boolean
```

See Also:

- o)help OneDimensionalArray
- o)help Vector
- o)help ExtensibleLinearAggregate
- o)show FlexibleArray

—————

FlexibleArray (FARRAY)



See

- ⇒ “PrimitiveArray” (PRIMARR) [17](#) on page [2109](#)
- ⇒ “Tuple” (TUPLE) [21](#) on page [2752](#)

⇒ “IndexedFlexibleArray” (IFARRAY) [10](#) on page [1222](#)
 ⇒ “IndexedOneDimensionalArray” (IARRAY1) [10](#) on page [1243](#)
 ⇒ “OneDimensionalArray” (ARRAY1) [16](#) on page [1774](#)

Exports:

any?	coerce	concat	concat!	construct
convert	copy	copyInto!	count	delete
delete!	elt	empty	empty?	entries
entry?	eq?	eval	every?	fill!
find	first	flexibleArray	hash	index?
indices	insert	insert!	latex	less?
map	map!	max	maxIndex	member?
members	merge	merge!	min	minIndex
more?	new	parts	physicalLength	physicalLength!
position	qelt	qsetelt!	reduce	remove
remove!	removeDuplicates	removeDuplicates!	reverse	reverse!
sample	select	select!	setelt	shrinkable
size?	sort	sort!	sorted?	swap!
#?	?<?	?<=?	?=?	?>?
?>=?	?~=?	?..?		

— domain FARRAY FlexibleArray —

```
)abbrev domain FARRAY FlexibleArray
++ Author: Mark Botch
++ Description:
++ A FlexibleArray is the notion of an array intended to allow for growth
++ at the end only. Hence the following efficient operations
++ \spad{append(x,a)} meaning append item x at the end of the array \spad{a}
++ \spad{delete(a,n)} meaning delete the last item from the array \spad{a}
++ Flexible arrays support the other operations inherited from
++ \spadtype{ExtensibleLinearAggregate}. However, these are not efficient.
++ Flexible arrays combine the \spad{O(1)} access time property of arrays
++ with growing and shrinking at the end in \spad{O(1)} (average) time.
++ This is done by using an ordinary array which may have zero or more
++ empty slots at the end. When the array becomes full it is copied
++ into a new larger (50% larger) array. Conversely, when the array
++ becomes less than 1/2 full, it is copied into a smaller array.
++ Flexible arrays provide for an efficient implementation of many
++ data structures in particular heaps, stacks and sets.
```

```
FlexibleArray(S: Type) == Implementation where
  ARRAYMININDEX ==> 1      -- if you want to change this, be my guest
  Implementation ==> IndexedFlexibleArray(S, ARRAYMININDEX)
-- Join(OneDimensionalArrayAggregate S, ExtensibleLinearAggregate S)
```

— **FARRAY.dotabb** —

```
"FARRAY" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FARRAY"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"FARRAY" -> "A1AGG"
```

domain **Float**

As reported in bug number 4733 (rounding of negative numbers) errors were observed in operations such as

```
-> round(-3.9)
-> truncate(-3.9)
```

The problem is the unexpected behaviour of the shift with negative integer arguments.

```
-> shift(-7,-1)
```

returns -4 while the code here in float expects the value to be -3. shift uses the lisp function ASH 'arithmetic shift left' but the spad code expects an unsigned 'logical' shift. See

http://www.lispworks.com/reference/HyperSpec/Body/f_ash.htm#ash

A new internal function shift2 is defined in terms of shift to compensate for the use of ASH and provide the required function.

It is currently unknown whether the unexpected behaviour of shift for negative arguments will cause bugs in other parts of Axiom.

— **Float.input** —

```
)set break resume
)sys rm -f Float.output
)spool Float.output
)set message test on
)set message auto off
)clear all
--S 1 of 64
1.234
--R
--R
--R (1) 1.234
--R
--E 1
```

Type: Float

[illegible]


```
--S 14 of 64
fractionPart 3.6
--R
--R
--R (13) 0.6
--R
--E 14
```

Type: Float

```
--S 15 of 64
digits 40
--R
--R
--R (14) 20
--R
--E 15
```

Type: PositiveInteger

```
--S 16 of 64
sqrt 0.2
--R
--R
--R (15) 0.4472135954 9995793928 1834733746 2552470881
--R
--E 16
```

Type: Float

```
--S 17 of 64
pi()$Float
--R
--R
--R (16) 3.1415926535 8979323846 2643383279 502884197
--R
--E 17
```

Type: Float

```
--S 18 of 64
digits 500
--R
--R
--R (17) 40
--R
--E 18
```

Type: PositiveInteger

```
--S 19 of 64
pi()$Float
--R
--R
--R (18)
--R 3.1415926535 8979323846 2643383279 5028841971 6939937510 5820974944 592307816
--R 4 0628620899 8628034825 3421170679 8214808651 3282306647 0938446095 505822317
--R 2 5359408128 4811174502 8410270193 8521105559 6446229489 5493038196 442881097
--R 5 6659334461 2847564823 3786783165 2712019091 4564856692 3460348610 454326648
```

```

--R 2 1339360726 0249141273 7245870066 0631558817 4881520920 9628292540 917153643
--R 6 7892590360 0113305305 4882046652 1384146951 9415116094 3305727036 575959195
--R 3 0921861173 8193261179 3105118548 0744623799 6274956735 1885752724 891227938
--R 1 830119491
--R
--R                                          Type: Float
--E 19

```

```

--S 20 of 64
digits 20
--R
--R
--R (19) 500
--R
--R                                          Type: PositiveInteger
--E 20

```

```

--S 21 of 64
outputSpacing 0; x := sqrt 0.2
--R
--R
--R (20) 0.44721359549995793928
--R
--R                                          Type: Float
--E 21

```

```

--S 22 of 64
outputSpacing 5; x
--R
--R
--R (21) 0.44721 35954 99957 93928
--R
--R                                          Type: Float
--E 22

```

```

--S 23 of 64
y := x/10**10
--R
--R
--R (22) 0.44721 35954 99957 93928 E -10
--R
--R                                          Type: Float
--E 23

```

```

--S 24 of 64
outputFloating(); x
--R
--R
--R (23) 0.44721 35954 99957 93928 E 0
--R
--R                                          Type: Float
--E 24

```

```

--S 25 of 64
outputFixed(); y
--R

```

```
--R
--R (24) 0.00000 00000 44721 35954 99957 93928
--R
--R                                         Type: Float
--E 25
```

```
--S 26 of 64
outputFloating 2; y
--R
--R
--R (25) 0.45 E -10
--R
--R                                         Type: Float
--E 26
```

```
--S 27 of 64
outputFixed 2; x
--R
--R
--R (26) 0.45
--R
--R                                         Type: Float
--E 27
```

```
--S 28 of 64
outputGeneral()
--R
--R
--R                                         Type: Void
--E 28
```

```
--S 29 of 64
a: Matrix Fraction Integer := matrix [ [1/(i+j+1) for j in 0..9] for i in 0..9]
--R
--R
--R      +   1   1   1   1   1   1   1   1   1   1+
--R      |1   -   -   -   -   -   -   -   -   -  --|
--R      |   2   3   4   5   6   7   8   9   10|
--R      |
--R      |1   1   1   1   1   1   1   1   1   1|
--R      |-   -   -   -   -   -   -   -   -  -- --|
--R      |2   3   4   5   6   7   8   9   10  11|
--R      |
--R      |1   1   1   1   1   1   1   1   1   1|
--R      |-   -   -   -   -   -   -   -  -- -- --|
--R      |3   4   5   6   7   8   9   10  11  12|
--R      |
--R      |1   1   1   1   1   1   1   1   1   1|
--R      |-   -   -   -   -   -   -  -- -- -- --|
--R      |4   5   6   7   8   9   10  11  12  13|
--R      |
--R      |1   1   1   1   1   1   1   1   1   1|
--R      |-   -   -   -   -   -  -- -- -- -- --|
--R      |5   6   7   8   9   10  11  12  13  14|
```

```

--R (28) |
--R |1  1  1  1  1  1  1  1  1  1|
--R |-  -  -  -  -- -- -- -- --|
--R |6  7  8  9  10 11 12 13 14 15|
--R |
--R |1  1  1  1  1  1  1  1  1  1|
--R |-  -  -  -- -- -- -- --|
--R |7  8  9  10 11 12 13 14 15 16|
--R |
--R |1  1  1  1  1  1  1  1  1  1|
--R |-  -  -- -- -- -- --|
--R |8  9  10 11 12 13 14 15 16 17|
--R |
--R |1  1  1  1  1  1  1  1  1  1|
--R |-  -- -- -- -- --|
--R |9  10 11 12 13 14 15 16 17 18|
--R |
--R | 1  1  1  1  1  1  1  1  1  1|
--R |-- -- -- -- --|
--R +10 11 12 13 14 15 16 17 18 19+
--R                                         Type: Matrix(Fraction(Integer))
--E 29

--S 30 of 64
d:= determinant a
--R
--R
--R
--R
--R (29) -----
--R 46206893947914691316295628839036278726983680000000000
--R                                         Type: Fraction(Integer)
--E 30

--S 31 of 64
d :: Float
--R
--R
--R (30) 0.21641 79226 43149 18691 E -52
--R                                         Type: Float
--E 31

--S 32 of 64
b: Matrix DoubleFloat := matrix [ [1/(i+j+1$DoubleFloat) for j in 0..9] for i in 0..9]
--R
--R
--R (31)
--R [
--R [1., 0.5, 0.3333333333333331, 0.25, 0.20000000000000001,
--R 0.16666666666666666, 0.14285714285714285, 0.125, 0.11111111111111111,
--R 0.10000000000000001]

```


Type: Matrix(DoubleFloat)

```

--S 33 of 64
determinant b
--R
--R
--R (32) 2.1643677945721411E-53
--R
--R                                          Type: DoubleFloat
--E 33

--S 34 of 64
digits 40
--R
--R
--R (33) 20
--R
--R                                          Type: PositiveInteger
--E 34

--S 35 of 64
c: Matrix Float := matrix [ [1/(i+j+1$Float) for j in 0..9] for i in 0..9]
--R
--R
--R (34)
--R [
--R [1.0, 0.5, 0.33333 33333 33333 33333 33333 33333 33333 33333 33333, 0.25, 0.2,
--R 0.16666 66666 66666 66666 66666 66666 66666 66666 66667,
--R 0.14285 71428 57142 85714 28571 42857 14285 71429, 0.125,
--R 0.11111 11111 11111 11111 11111 11111 11111 11111 11111, 0.1]
--R ,
--R [0.5, 0.33333 33333 33333 33333 33333 33333 33333 33333 33333, 0.25, 0.2,
--R 0.16666 66666 66666 66666 66666 66666 66666 66666 66667,
--R 0.14285 71428 57142 85714 28571 42857 14285 71429, 0.125,
--R 0.11111 11111 11111 11111 11111 11111 11111 11111 11111, 0.1,
--R 0.09090 90909 09090 90909 09090 90909 09090 90909 90909 1]
--R ,
--R [0.33333 33333 33333 33333 33333 33333 33333 33333 33333 33333, 0.25, 0.2,
--R 0.16666 66666 66666 66666 66666 66666 66666 66666 66667,
--R 0.14285 71428 57142 85714 28571 42857 14285 71429, 0.125,
--R 0.11111 11111 11111 11111 11111 11111 11111 11111 11111, 0.1,
--R 0.09090 90909 09090 90909 09090 90909 09090 90909 90909 1,
--R 0.08333 33333 33333 33333 33333 33333 33333 33333 33333 4]
--R ,
--R [0.25, 0.2, 0.16666 66666 66666 66666 66666 66666 66666 66666 66667,
--R 0.14285 71428 57142 85714 28571 42857 14285 71429, 0.125,
--R 0.11111 11111 11111 11111 11111 11111 11111 11111 11111, 0.1,
--R 0.09090 90909 09090 90909 09090 90909 09090 90909 90909 1,
--R 0.08333 33333 33333 33333 33333 33333 33333 33333 33333 4,
--R 0.07692 30769 23076 92307 69230 76923 07692 30769 2]

```

```

--R ,
--R
--R [0.2, 0.16666 66666 66666 66666 66666 66666 66666 66667,
--R 0.14285 71428 57142 85714 28571 42857 14285 71429, 0.125,
--R 0.11111 11111 11111 11111 11111 11111 11111 11111, 0.1,
--R 0.09090 90909 09090 90909 09090 90909 09090 90909 1,
--R 0.08333 33333 33333 33333 33333 33333 33333 33333 4,
--R 0.07692 30769 23076 92307 69230 76923 07692 30769 2,
--R 0.07142 85714 28571 42857 14285 71428 57142 85714 3]
--R ,
--R
--R [0.16666 66666 66666 66666 66666 66666 66666 66667,
--R 0.14285 71428 57142 85714 28571 42857 14285 71429, 0.125,
--R 0.11111 11111 11111 11111 11111 11111 11111 11111, 0.1,
--R 0.09090 90909 09090 90909 09090 90909 09090 90909 1,
--R 0.08333 33333 33333 33333 33333 33333 33333 33333 4,
--R 0.07692 30769 23076 92307 69230 76923 07692 30769 2,
--R 0.07142 85714 28571 42857 14285 71428 57142 85714 3,
--R 0.06666 66666 66666 66666 66666 66666 66666 66666 7]
--R ,
--R
--R [0.14285 71428 57142 85714 28571 42857 14285 71429, 0.125,
--R 0.11111 11111 11111 11111 11111 11111 11111 11111, 0.1,
--R 0.09090 90909 09090 90909 09090 90909 09090 90909 1,
--R 0.08333 33333 33333 33333 33333 33333 33333 33333 4,
--R 0.07692 30769 23076 92307 69230 76923 07692 30769 2,
--R 0.07142 85714 28571 42857 14285 71428 57142 85714 3,
--R 0.06666 66666 66666 66666 66666 66666 66666 66666 7, 0.0625]
--R ,
--R
--R [0.125, 0.11111 11111 11111 11111 11111 11111 11111 11111, 0.1,
--R 0.09090 90909 09090 90909 09090 90909 09090 90909 1,
--R 0.08333 33333 33333 33333 33333 33333 33333 33333 4,
--R 0.07692 30769 23076 92307 69230 76923 07692 30769 2,
--R 0.07142 85714 28571 42857 14285 71428 57142 85714 3,
--R 0.06666 66666 66666 66666 66666 66666 66666 66666 7, 0.0625,
--R 0.05882 35294 11764 70588 23529 41176 47058 82352 9]
--R ,
--R
--R [0.11111 11111 11111 11111 11111 11111 11111 11111, 0.1,
--R 0.09090 90909 09090 90909 09090 90909 09090 90909 1,
--R 0.08333 33333 33333 33333 33333 33333 33333 33333 4,
--R 0.07692 30769 23076 92307 69230 76923 07692 30769 2,
--R 0.07142 85714 28571 42857 14285 71428 57142 85714 3,
--R 0.06666 66666 66666 66666 66666 66666 66666 66666 7, 0.0625,
--R 0.05882 35294 11764 70588 23529 41176 47058 82352 9,
--R 0.05555 55555 55555 55555 55555 55555 55555 55555 6]
--R ,
--R
--R [0.1, 0.09090 90909 09090 90909 09090 90909 09090 90909 1,

```

[illegible]

```
--S 41 of 64
c:=numeric pi()
--R
--R
--R (4) 3.14159 26535 89793 2385
--R
--R Type: Float
--E 41

--S 42 of 64
d:=0.0
--R
--R
--R (5) 0.0
--R
--R Type: Float
--E 42

--S 43 of 64
outputFixed 2
--R
--R
--R Type: Void
--E 43

--S 44 of 64
a
--R
--R
--R (7) 3.00
--R
--R Type: Float
--E 44

--S 45 of 64
b
--R
--R
--R (8) 3.10
--R
--R Type: Float
--E 45

--S 46 of 64
c
--R
--R
--R (9) 3.14
--R
--R Type: Float
--E 46

--S 47 of 64
d
--R
```

```
--R
--R (10) 0.00
--R
--R Type: Float
--E 47

--S 48 of 64
outputFixed 0
--R
--R
--R Type: Void
--E 48

--S 49 of 64
a
--R
--R
--R (12) 3.0
--R
--R Type: Float
--E 49

--S 50 of 64
b
--R
--R
--R (13) 3.
--R
--R Type: Float
--E 50

--S 51 of 64
c
--R
--R
--R (14) 3.
--R
--R Type: Float
--E 51

--S 52 of 64
31.1
--R
--R
--R (15) 31.
--R
--R Type: Float
--E 52

--S 53 of 64
310.1
--R
--R
--R (16) 310.
--R
--R Type: Float
--E 53
```

```
--S 54 of 64
```

```
d
```

```
--R
```

```
--R
```

```
--R (17) 0.0
```

```
--R
```

Type: Float

```
--E 54
```

```
--S 55 of 64
```

```
outputFixed(0)
```

```
--R
```

Type: Void

```
--E 55
```

```
--S 56 of 64
```

```
1.1
```

```
--R
```

```
--R (19) 1.
```

```
--R
```

Type: Float

```
--E 56
```

```
--S 57 of 64
```

```
3111.1
```

```
--R
```

```
--R (20) 3111.
```

```
--R
```

Type: Float

```
--E 57
```

```
--S 58 of 64
```

```
1234567890.1
```

```
--R
```

```
--R (21) 12345 67890.
```

```
--R
```

Type: Float

```
--E 58
```

```
--S 59 of 64
```

```
outputFixed(12)
```

```
--R
```

Type: Void

```
--E 59
```

```
--S 60 of 64
```

```
1234567890.1
```

```
--R
```

```
--R (23) 12345 67890.09999 99999 99
```

```
--R
```

Type: Float

```
--E 60
```

```
--S 61 of 64
```

```
outputFixed(15)
```

```
--R
```

Type: Void

--E 61

--S 62 of 64
1234567890.1

--R
--R (25) 12345 67890.09999 99999 98545
--R

Type: Float

--E 62

--S 63 of 64
outputFixed(2)

--R
--E 63

Type: Void

--S 64 of 64
1234567890.1

--R
--R (27) 12345 67890.10
--R
--E 64

Type: Float

)spool
)lisp (bye)

— Float.help —

Float

Axiom provides two kinds of floating point numbers. The domain Float implements a model of arbitrary precision floating point numbers. The domain DoubleFloat is intended to make available hardware floating point arithmetic in Axiom. The actual model of floating point that DoubleFloat provides is system-dependent. For example, on the IBM system 370 Axiom uses IBM double precision which has fourteen hexadecimal digits of precision or roughly sixteen decimal digits. Arbitrary precision floats allow the user to specify the precision at which arithmetic operations are computed. Although this is an attractive facility, it comes at a cost. Arbitrary-precision floating-point arithmetic typically takes twenty to two hundred times more time than hardware floating point.

Introduction to Float

Scientific notation is supported for input and output of floating point numbers. A floating point number is written as a string of digits containing a decimal point optionally followed by the letter "E", and then the exponent.

We begin by doing some calculations using arbitrary precision floats. The default precision is twenty decimal digits.

```
1.234
1.234
                                Type: Float
```

A decimal base for the exponent is assumed, so the number 1.234E2 denotes 1.234×10^2 .

```
1.234E2
123.4
                                Type: Float
```

The normal arithmetic operations are available for floating point numbers.

```
sqrt(1.2 + 2.3 / 3.4 ** 4.5)
1.0996972790 671286226
                                Type: Float
```

===== Conversion Functions =====

You can use conversion to go back and forth between Integer, Fraction Integer and Float, as appropriate.

```
i := 3 :: Float
3.0
                                Type: Float

i :: Integer
3
                                Type: Integer

i :: Fraction Integer
3
                                Type: Fraction Integer
```

Since you are explicitly asking for a conversion, you must take responsibility for any loss of exactness.

```
r := 3/7 :: Float
0.4285714285 7142857143
                                Type: Float
```

```

r :: Fraction Integer
  3
  -
  7

```

Type: Fraction Integer

This conversion cannot be performed: use truncate or round if that is what you intend.

```

r :: Integer
Cannot convert from type Float to Integer for value
0.4285714285 7142857143

```

The operations truncate and round truncate ...

```

truncate 3.6
3.0

```

Type: Float

and round to the nearest integral Float respectively.

```

round 3.6
4.0

```

Type: Float

```

truncate(-3.6)
- 3.0

```

Type: Float

```

round(-3.6)
- 4.0

```

Type: Float

The operation fractionPart computes the fractional part of x, that is, x - truncate x.

```

fractionPart 3.6
0.6

```

Type: Float

The operation digits allows the user to set the precision. It returns the previous value it was using.

```

digits 40
20

```

Type: PositiveInteger

```

sqrt 0.2
0.4472135954 9995793928 1834733746 2552470881

```

Type: Float

```
pi()$Float
3.1415926535 8979323846 2643383279 502884197
Type: Float
```

The precision is only limited by the computer memory available.
Calculations at 500 or more digits of precision are not difficult.

```
digits 500
40
Type: PositiveInteger
```

```
pi()$Float
3.1415926535 8979323846 2643383279 5028841971 6939937510 5820974944 592307816
4 0628620899 8628034825 3421170679 8214808651 3282306647 0938446095 505822317
2 5359408128 4811174502 8410270193 8521105559 6446229489 5493038196 442881097
5 6659334461 2847564823 3786783165 2712019091 4564856692 3460348610 454326648
2 1339360726 0249141273 7245870066 0631558817 4881520920 9628292540 917153643
6 7892590360 0113305305 4882046652 1384146951 9415116094 3305727036 575959195
3 0921861173 8193261179 3105118548 0744623799 6274956735 1885752724 891227938
1 830119491
Type: Float
```

Reset digits to its default value.

```
digits 20
500
Type: PositiveInteger
```

Numbers of type Float are represented as a record of two integers, namely, the mantissa and the exponent where the base of the exponent is binary. That is, the floating point number (m,e) represents the number $m \times 2^e$. A consequence of using a binary base is that decimal numbers can not, in general, be represented exactly.

```
=====
Output Functions
=====
```

A number of operations exist for specifying how numbers of type Float are to be displayed. By default, spaces are inserted every ten digits in the output for readability. Note that you cannot include spaces in the input form of a floating point number, though you can use underscores.

Output spacing can be modified with the outputSpacing operation. This inserts no spaces and then displays the value of x .

```
outputSpacing 0; x := sqrt 0.2
```

```
0.44721359549995793928
      Type: Float
```

Issue this to have the spaces inserted every 5 digits.

```
outputSpacing 5; x
0.44721 35954 99957 93928
      Type: Float
```

By default, the system displays floats in either fixed format or scientific format, depending on the magnitude of the number.

```
y := x/10**10
0.44721 35954 99957 93928 E -10
      Type: Float
```

A particular format may be requested with the operations `outputFloating` and `outputFixed`.

```
outputFloating(); x
0.44721 35954 99957 93928 E 0
      Type: Float
```

```
outputFixed(); y
0.00000 00000 44721 35954 99957 93928
      Type: Float
```

Additionally, you can ask for `n` digits to be displayed after the decimal point.

```
outputFloating 2; y
0.45 E -10
      Type: Float
```

```
outputFixed 2; x
0.45
      Type: Float
```

This resets the output printing to the default behavior.

```
outputGeneral()
      Type: Void
```

```
=====
An Example: Determinant of a Hilbert Matrix
=====
```

Consider the problem of computing the determinant of a 10 by 10 Hilbert matrix. The (i,j) -th entry of a Hilbert matrix is given by $1/(i+j+1)$.

First do the computation using rational numbers to obtain the exact result.

```
a: Matrix Fraction Integer:=matrix[ [1/(i+j+1) for j in 0..9] for i in 0..9]
+ 1 1 1 1 1 1 1 1 1 1+
|1 - - - - - - - - --|
| 2 3 4 5 6 7 8 9 10|
|
|1 1 1 1 1 1 1 1 1 1|
|- - - - - - - - -- --|
|2 3 4 5 6 7 8 9 10 11|
|
|1 1 1 1 1 1 1 1 1 1|
|- - - - - - - - -- -- --|
|3 4 5 6 7 8 9 10 11 12|
|
|1 1 1 1 1 1 1 1 1 1|
|- - - - - - - - -- -- --|
|4 5 6 7 8 9 10 11 12 13|
|
|1 1 1 1 1 1 1 1 1 1|
|- - - - - - - - -- -- --|
|5 6 7 8 9 10 11 12 13 14|
|
|1 1 1 1 1 1 1 1 1 1|
|- - - - - - - - -- -- --|
|6 7 8 9 10 11 12 13 14 15|
|
|1 1 1 1 1 1 1 1 1 1|
|- - - - - - - - -- -- --|
|7 8 9 10 11 12 13 14 15 16|
|
|1 1 1 1 1 1 1 1 1 1|
|- - - - - - - - -- -- --|
|8 9 10 11 12 13 14 15 16 17|
|
|1 1 1 1 1 1 1 1 1 1|
|- - - - - - - - -- -- --|
|9 10 11 12 13 14 15 16 17 18|
|
|1 1 1 1 1 1 1 1 1 1|
|- - - - - - - - -- -- --|
+10 11 12 13 14 15 16 17 18 19+
Type: Matrix Fraction Integer
```

This version of determinant uses Gaussian elimination.

```
d:= determinant a
```

```
-----
4620689394791469131629562883903627872698368000000000
Type: Fraction Integer
```

```
d :: Float
0.21641 79226 43149 18691 E -52
Type: Float
```

Now use hardware floats. Note that a semicolon (;) is used to prevent the display of the matrix.

```
b: Matrix DoubleFloat:=matrix[ [1/(i+j+1\DoubleFloat) for j in 0..9] for i in 0..9];
```

```
Type: Matrix DoubleFloat
```

The result given by hardware floats is correct only to four significant digits of precision. In the jargon of numerical analysis, the Hilbert matrix is said to be "ill-conditioned."

```
determinant b
2.1643677945721411E-53
Type: DoubleFloat
```

Now repeat the computation at a higher precision using Float.

```
digits 40
20
Type: PositiveInteger
```

```
c: Matrix Float := matrix [ [1/(i+j+1\Float) for j in 0..9] for i in 0..9];
Type: Matrix Float
```

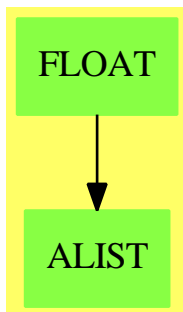
```
determinant c
0.21641 79226 43149 18690 60594 98362 26174 36159 E -52
Type: Float
```

Reset digits to its default value.

```
digits 20
40
Type: PositiveInteger
```

See Also:

- o)help DoubleFloat
- o)show Float

Float (FLOAT)**Exports:**

0	1	abs	acos
acosh	acot	acoth	acsc
acsch	asec	asech	asin
asinh	associates?	atan	atanh
base	bits	ceiling	characteristic
coerce	convert	cos	cosh
cot	coth	csc	csch
D	decreasePrecision	differentiate	digits
divide	euclideanSize	exp	expressIdealMember
exp1	exponent	exquo	extendedEuclidean
factor	float	floor	fractionPart
gcd	gcdPolynomial	hash	increasePrecision
inv	latex	lcm	log
log10	log2	mantissa	max
min	multiEuclidean	negative?	norm
normalize	nthRoot	OMwrite	one?
order	outputFixed	outputFloating	outputGeneral
outputSpacing	patternMatch	pi	positive?
precision	prime?	principalIdeal	rationalApproximation
recip	relerror	retract	retractIfCan
round	sample	sec	sech
shift	sign	sin	sinh
sizeLess?	sqrt	squareFree	squareFreePart
subtractIfCan	tan	tanh	truncate
unit?	unitCanonical	unitNormal	wholePart
zero?	?*?	?**?	?+?
?-?	-?	?/?	?<?
?<=?	?=?	?>?	?>=?
?^?	?~=?	?quo?	?rem?

— domain FLOAT Float —

```
)abbrev domain FLOAT Float
```

```
++ Author: Michael Monagan
++ Date Created: December 1987
++ Change History: 19 Jun 1990
++ Description:
++ \spadtype{Float} implements arbitrary precision floating point arithmetic.
++ The number of significant digits of each operation can be set
++ to an arbitrary value (the default is 20 decimal digits).
++ The operation \spad{float(mantissa,exponent,base)} for integer
++ \spad{mantissa}, \spad{exponent} specifies the number
++ \spad{mantissa * base ** exponent}
++ The underlying representation for floats is binary
++ not decimal. The implications of this are described below.
++
++ The model adopted is that arithmetic operations are rounded to
++ to nearest unit in the last place, that is, accurate to within
++ \spad{2**(-bits)}. Also, the elementary functions and constants are
++ accurate to one unit in the last place.
++ A float is represented as a record of two integers, the mantissa
++ and the exponent. The base of the representation is binary, hence
++ a \spad{Record(m:mantissa,e:exponent)} represents the number
++ \spad{m * 2 ** e}.
++ Though it is not assumed that the underlying integers are represented
++ with a binary base, the code will be most efficient when this is the
++ the case (this is true in most implementations of Lisp).
++ The decision to choose the base to be binary has some unfortunate
++ consequences. First, decimal numbers like 0.3 cannot be represented
++ exactly. Second, there is a further loss of accuracy during
++ conversion to decimal for output. To compensate for this, if d digits
++ of precision are specified, \spad{1 + ceiling(log2(10^d))} bits are used.
++ Two numbers that are displayed identically may therefore be
++ not equal. On the other hand, a significant efficiency loss would
++ be incurred if we chose to use a decimal base when the underlying
++ integer base is binary.
++
++ Algorithms used:
++ For the elementary functions, the general approach is to apply
++ identities so that the taylor series can be used, and, so
++ that it will converge within \spad{0( sqrt n )} steps. For example,
++ using the identity \spad{exp(x) = exp(x/2)**2}, we can compute
++ \spad{exp(1/3)} to n digits of precision as follows. We have
++ \spad{exp(1/3) = exp(2 ** (-sqrt s) / 3) ** (2 ** sqrt s)}.
++ The taylor series will converge in less than sqrt n steps and the
++ exponentiation requires sqrt n multiplications for a total of
++ \spad{2 sqrt n} multiplications. Assuming integer multiplication costs
++ \spad{0( n**2 )} the overall running time is \spad{0( sqrt(n) n**2 )}.
```



```

++ This approach is the best known approach for precisions up to
++ about 10,000 digits at which point the methods of Brent
++ which are \spad{0( log(n) n**2 )} become competitive. Note also that
++ summing the terms of the taylor series for the elementary
++ functions is done using integer operations. This avoids the
++ overhead of floating point operations and results in efficient
++ code at low precisions. This implementation makes no attempt
++ to reuse storage, relying on the underlying system to do
++ \spadgloss{garbage collection}. I estimate that the efficiency of this
++ package at low precisions could be improved by a factor of 2
++ if in-place operations were available.
++
++ Running times: in the following, n is the number of bits of precision\br
++ \spad{*}, \spad{ / }, \spad{sqrt}, \spad{pi}, \spad{exp1}, \spad{log2},
++ \spad{log10}: \spad{ 0( n**2 )} \br
++ \spad{exp}, \spad{log}, \spad{sin}, \spad{atan}: \spad{0(sqrt(n) n**2)}\br
++ The other elementary functions are coded in terms of the ones above.

B ==> Boolean
I ==> Integer
S ==> String
PI ==> PositiveInteger
RN ==> FractionInteger
SF ==> DoubleFloat
N ==> NonNegativeInteger

Float():
Join(FloatingPointSystem, DifferentialRing, ConvertibleTo String, OpenMath, _
CoercibleTo DoubleFloat, TranscendentalFunctionCategory, ConvertibleTo InputForm) with
_/ : (% , I) -> %
++ x / i computes the division from x by an integer i.
*_ : (% , %) -> %
++ x ** y computes \spad{exp(y log x)} where \spad{x >= 0}.
normalize: % -> %
++ normalize(x) normalizes x at current precision.
relererror : (% , %) -> I
++ relererror(x,y) computes the absolute value of \spad{x - y} divided by
++ y, when \spad{y ~= 0}.
shift: (% , I) -> %
++ shift(x,n) adds n to the exponent of float x.
rationalApproximation: (% , N) -> RN
++ rationalApproximation(f, n) computes a rational approximation
++ r to f with relative error \spad{< 10**(-n)}.
rationalApproximation: (% , N, N) -> RN
++ rationalApproximation(f, n, b) computes a rational
++ approximation r to f with relative error \spad{< b**(-n)}, that is
++ \spad{|(r-f)/f| < b**(-n)}.
log2 : () -> %
++ log2() returns \spad{ln 2}, i.e. \spad{0.6931471805...}.
log10: () -> %

```

```

    ++ log10() returns \spad{ln 10}: \spad{2.3025809299...}.
exp1 : () -> %
    ++ exp1() returns exp 1: \spad{2.7182818284...}.
atan : (%,% ) -> %
    ++ atan(x,y) computes the arc tangent from x with phase y.
log2 : % -> %
    ++ log2(x) computes the logarithm for x to base 2.
log10: % -> %
    ++ log10(x) computes the logarithm for x to base 10.
convert: SF -> %
    ++ convert(x) converts a \spadtype{DoubleFloat} x to a \spadtype{Float}.
outputFloating: () -> Void
    ++ outputFloating() sets the output mode to floating (scientific) notation, i.e.
    ++ \spad{mantissa * 10 exponent} is displayed as \spad{0.mantissa E exponent}.
outputFloating: N -> Void
    ++ outputFloating(n) sets the output mode to floating (scientific) notation
    ++ with n significant digits displayed after the decimal point.
outputFixed: () -> Void
    ++ outputFixed() sets the output mode to fixed point notation;
    ++ the output will contain a decimal point.
outputFixed: N -> Void
    ++ outputFixed(n) sets the output mode to fixed point notation,
    ++ with n digits displayed after the decimal point.
outputGeneral: () -> Void
    ++ outputGeneral() sets the output mode (default mode) to general
    ++ notation; numbers will be displayed in either fixed or floating
    ++ (scientific) notation depending on the magnitude.
outputGeneral: N -> Void
    ++ outputGeneral(n) sets the output mode to general notation
    ++ with n significant digits displayed.
outputSpacing: N -> Void
    ++ outputSpacing(n) inserts a space after n (default 10) digits on output;
    ++ outputSpacing(0) means no spaces are inserted.
arbitraryPrecision
arbitraryExponent
== add
BASE ==> 2
BITS:Reference(PI) := ref 68 -- 20 digits
LENGTH ==> INTEGER_-LENGTH$Lisp
ISQRT ==> approxSqrt$IntegerRoots(I)
Rep := Record( mantissa:I, exponent:I )
StoredConstant ==> Record( precision:PI, value:% )
UCA ==> Record( unit:%, coef:%, associate:% )
inc ==> increasePrecision
dec ==> decreasePrecision

-- local utility operations
shift2 : (I,I) -> I -- WSP: fix bug in shift
times : (%,% ) -> % -- multiply x and y with no rounding
itimes: (I,% ) -> % -- multiply by a small integer

```

```

chop: (% ,PI) -> %           -- chop x at p bits of precision
dvide: (% ,%) -> %          -- divide x by y with no rounding
square: (% ,I) -> %         -- repeated squaring with chopping
power: (% ,I) -> %          -- x ** n with chopping
plus: (% ,%) -> %           -- addition with no rounding
sub: (% ,%) -> %            -- subtraction with no rounding
negate: % -> %              -- negation with no rounding
ceillog10base2: PI -> PI    -- rational approximation
floorln2: PI -> PI          -- rational approximation

atanSeries: % -> %          -- atan(x) by taylor series |x| < 1/2
atanInverse: I -> %         -- atan(1/n) for n an integer > 1
expInverse: I -> %          -- exp(1/n) for n an integer
expSeries: % -> %           -- exp(x) by taylor series |x| < 1/2
logSeries: % -> %           -- log(x) by taylor series 1/2 < x < 2
sinSeries: % -> %           -- sin(x) by taylor series |x| < 1/2
cosSeries: % -> %           -- cos(x) by taylor series |x| < 1/2
piRamanujan: () -> %       -- pi using Ramanujans series

writeOMFloat(dev: OpenMathDevice, x: %): Void ==
  OMputApp(dev)
  OMputSymbol(dev, "bigfloat1", "bigfloat")
  OMputInteger(dev, mantissa x)
  OMputInteger(dev, 2)
  OMputInteger(dev, exponent x)
  OMputEndApp(dev)

OMwrite(x: %): String ==
  s: String := ""
  sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
  dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML)
  OMputObject(dev)
  writeOMFloat(dev, x)
  OMputEndObject(dev)
  OMclose(dev)
  s := OM_-STRINGPTRTOSTRING(sp)$Lisp pretend String
  s

OMwrite(x: %, wholeObj: Boolean): String ==
  s: String := ""
  sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
  dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML)
  if wholeObj then
    OMputObject(dev)
  writeOMFloat(dev, x)
  if wholeObj then
    OMputEndObject(dev)
  OMclose(dev)
  s := OM_-STRINGPTRTOSTRING(sp)$Lisp pretend String
  s

```

```

OMwrite(dev: OpenMathDevice, x: %): Void ==
  OMputObject(dev)
  writeOMFloat(dev, x)
  OMputEndObject(dev)

OMwrite(dev: OpenMathDevice, x: %, wholeObj: Boolean): Void ==
  if wholeObj then
    OMputObject(dev)
    writeOMFloat(dev, x)
  if wholeObj then
    OMputEndObject(dev)

shift2(x,y) == sign(x)*shift(sign(x)*x,y)

asin x ==
  zero? x => 0
  negative? x => -asin(-x)
--   one? x => pi()/2
  (x = 1) => pi()/2
  x > 1 => error "asin: argument > 1 in magnitude"
  inc 5; r := atan(x/sqrt(sub(1,times(x,x)))); dec 5
  normalize r

acos x ==
  zero? x => pi()/2
  negative? x => (inc 3; r := pi()-acos(-x); dec 3; normalize r)
--   one? x => 0
  (x = 1) => 0
  x > 1 => error "acos: argument > 1 in magnitude"
  inc 5; r := atan(sqrt(sub(1,times(x,x)))/x); dec 5
  normalize r

atan(x,y) ==
  x = 0 =>
    y > 0 => pi()/2
    y < 0 => -pi()/2
    0
  -- Only count on first quadrant being on principal branch.
  theta := atan abs(y/x)
  if x < 0 then theta := pi() - theta
  if y < 0 then theta := - theta
  theta

atan x ==
  zero? x => 0
  negative? x => -atan(-x)
  if x > 1 then
    inc 4
    r := if zero? fractionPart x and x < [bits(),0] then atanInverse wholePart x

```

```

        else atan(1/x)
    r := pi/2 - r
    dec 4
    return normalize r
-- make |x| < 0( 2**(-sqrt p) ) < 1/2 to speed series convergence
-- by using the formula atan(x) = 2*atan(x/(1+sqrt(1+x**2)))
k := ISQRT (bits()-100)::I quo 5
k := max(0,2 + k + order x)
inc(2*k)
for i in 1..k repeat x := x/(1+sqrt(1+x*x))
t := atanSeries x
dec(2*k)
t := shift(t,k)
normalize t

atanSeries x ==
-- atan(x) = x (1 - x**2/3 + x**4/5 - x**6/7 + ...) |x| < 1
p := bits() + LENGTH bits() + 2
s:I := d:I := shift(1,p)
y := times(x,x)
t := m := - shift2(y.mantissa,y.exponent+p)
for i in 3.. by 2 while t ^= 0 repeat
    s := s + t quo i
    t := (m * t) quo d
x * [s,-p]

atanInverse n ==
-- compute atan(1/n) for an integer n > 1
-- atan n = 1/n - 1/n**3/3 + 1/n**5/5 - ...
-- pi = 16 atan(1/5) - 4 atan(1/239)
n2 := -n*n
e:I := bits() + LENGTH bits() + LENGTH n + 1
s:I := shift(1,e) quo n
t:I := s quo n2
for k in 3.. by 2 while t ^= 0 repeat
    s := s + t quo k
    t := t quo n2
normalize [s,-e]

sin x ==
s := sign x; x := abs x; p := bits(); inc 4
if x > [6,0] then (inc p; x := 2*pi*fractionPart(x/pi/2); bits p)
if x > [3,0] then (inc p; s := -s; x := x - pi; bits p)
if x > [3,-1] then (inc p; x := pi - x; dec p)
-- make |x| < 0( 2**(-sqrt p) ) < 1/2 to speed series convergence
-- by using the formula sin(3*x/3) = 3 sin(x/3) - 4 sin(x/3)**3
-- the running time is 0( sqrt p M(p) ) assuming |x| < 1
k := ISQRT (bits()-100)::I quo 4
k := max(0,2 + k + order x)
if k > 0 then (inc k; x := x / 3**k::N)

```

```

r := sinSeries x
for i in 1..k repeat r := itimes(3,r)-shift(r**3,2)
bits p
s * r

sinSeries x ==
-- sin(x) = x (1 - x**2/3! + x**4/5! - x**6/7! + ... |x| < 1/2
p := bits() + LENGTH bits() + 2
y := times(x,x)
s:I := d:I := shift(1,p)
m:I := - shift2(y.mantissa,y.exponent+p)
t:I := m quo 6
for i in 4.. by 2 while t ^= 0 repeat
  s := s + t
  t := (m * t) quo (i*(i+1))
  t := t quo d
x * [s,-p]

cos x ==
s:I := 1; x := abs x; p := bits(); inc 4
if x > [6,0] then (inc p; x := 2*pi*fractionPart(x/pi/2); dec p)
if x > [3,0] then (inc p; s := -s; x := x-pi; dec p)
if x > [1,0] then
  -- take care of the accuracy problem near pi/2
  inc p; x := pi/2-x; bits p; x := normalize x
  return (s * sin x)
-- make |x| < 0( 2**(-sqrt p) ) < 1/2 to speed series convergence
-- by using the formula cos(2*x/2) = 2 cos(x/2)**2 - 1
-- the running time is O( sqrt p M(p) ) assuming |x| < 1
k := ISQRT (bits()-100)::I quo 3
k := max(0,2 + k + order x)
-- need to increase precision by more than k, otherwise recursion
-- causes loss of accuracy.
-- Michael Monagan suggests adding a factor of log(k)
if k > 0 then (inc(k+length(k)**2); x := shift(x,-k))
r := cosSeries x
for i in 1..k repeat r := shift(r*r,1)-1
bits p
s * r

cosSeries x ==
-- cos(x) = 1 - x**2/2! + x**4/4! - x**6/6! + ... |x| < 1/2
p := bits() + LENGTH bits() + 1
y := times(x,x)
s:I := d:I := shift(1,p)
m:I := - shift2(y.mantissa,y.exponent+p)
t:I := m quo 2
for i in 3.. by 2 while t ^= 0 repeat

```

```

    s := s + t
    t := (m * t) quo (i*(i+1))
    t := t quo d
    normalize [s,-p]

tan x ==
  s := sign x; x := abs x; p := bits(); inc 6
  if x > [3,0] then (inc p; x := pi()*fractionPart(x/pi())); dec p)
  if x > [3,-1] then (inc p; x := pi()-x; s := -s; dec p)
  if x > 1 then (c := cos x; t := sqrt(1-c*c)/c)
  else (c := sin x; t := c/sqrt(1-c*c))
  bits p
  s * t

P:StoredConstant := [1,[1,2]]
pi() ==
  -- We use Ramanujan's identity to compute pi.
  -- The running time is quadratic in the precision.
  -- This is about twice as fast as Machin's identity on Lisp/VM
  -- pi = 16 atan(1/5) - 4 atan(1/239)
  bits() <= P.precision => normalize P.value
  (P := [bits(), piRamanujan()]) value

piRamanujan() ==
  -- Ramanujans identity for 1/pi
  -- Reference: Shanks and Wrench, Math Comp, 1962
  -- "Calculation of pi to 100,000 Decimals".
  n := bits() + LENGTH bits() + 11
  t:I := shift(1,n) quo 882
  d:I := 4*882**2
  s:I := 0
  for i in 2.. by 2 for j in 1123.. by 21460 while t ^= 0 repeat
    s := s + j*t
    m := -(i-1)*(2*i-1)*(2*i-3)
    t := (m*t) quo (d*i**3)
  1 / [s,-n-2]

sinh x ==
  zero? x => 0
  lost:I := max(- order x,0)
  2*lost > bits() => x
  inc(5+lost); e := exp x; s := (e-1/e)/2; dec(5+lost)
  normalize s

cosh x ==
  (inc 5; e := exp x; c := (e+1/e)/2; dec 5; normalize c)

tanh x ==
  zero? x => 0
  lost:I := max(- order x,0)

```

```

2*lost > bits() => x
inc(6+lost); e := exp x; e := e*e; t := (e-1)/(e+1); dec(6+lost)
normalize t

asinh x ==
  p := min(0,order x)
  if zero? x or 2*p < -bits() then return x
  inc(5-p); r := log(x+sqrt(1+x*x)); dec(5-p)
  normalize r

acosh x ==
  if x < 1 then error "invalid argument to acosh"
  inc 5; r := log(x+sqrt(sub(times(x,x),1))); dec 5
  normalize r

atanh x ==
  if x > 1 or x < -1 then error "invalid argument to atanh"
  p := min(0,order x)
  if zero? x or 2*p < -bits() then return x
  inc(5-p); r := log((x+1)/(1-x))/2; dec(5-p)
  normalize r

log x ==
  negative? x => error "negative log"
  zero? x => error "log 0 generated"
  p := bits(); inc 5
  -- apply log(x) = n log 2 + log(x/2**n) so that 1/2 < x < 2
  if (n := order x) < 0 then n := n+1
  l := if n = 0 then 0 else (x := shift(x,-n); n * log2)
  -- speed the series convergence by finding m and k such that
  -- | exp(m/2**k) x - 1 | < 1 / 2 ** 0(sqrt p)
  -- write log(exp(m/2**k) x) as m/2**k + log x
  k := ISQRT (p-100)::I quo 3
  if k > 1 then
    k := max(1,k+order(x-1))
    inc k
    ek := expInverse (2**k::N)
    dec(p quo 2); m := order square(x,k); inc(p quo 2)
    m := (6847196937 * m) quo 9878417065 -- m := m log 2
    x := x * ek ** (-m)
    l := l + [m,-k]
  l := l + logSeries x
  bits p
  normalize l

logSeries x ==
  -- log(x) = 2 y (1 + y**2/3 + y**4/5 ...) for y = (x-1) / (x+1)
  -- given 1/2 < x < 2 on input we have -1/3 < y < 1/3
  p := bits() + (g := LENGTH bits() + 3)
  inc g; y := (x-1)/(x+1); dec g

```



```

s:I := d:I := shift(1,p)
z := times(y,y)
t := m := shift2(z.mantissa,z.exponent+p)
for i in 3.. by 2 while t ^= 0 repeat
  s := s + t quo i
  t := m * t quo d
y * [s,1-p]

L2:StoredConstant := [1,1]
log2() ==
-- log x = 2 * sum( ((x-1)/(x+1))**(2*k+1)/(2*k+1), k=1.. )
-- log 2 = 2 * sum( 1/9**k / (2*k+1), k=0..n ) / 3
n := bits() :: N
n <= L2.precision => normalize L2.value
n := n + LENGTH n + 3 -- guard bits
s:I := shift(1,n+1) quo 3
t:I := s quo 9
for k in 3.. by 2 while t ^= 0 repeat
  s := s + t quo k
  t := t quo 9
L2 := [bits(),[s,-n]]
normalize L2.value

L10:StoredConstant := [1,[1,1]]
log10() ==
-- log x = 2 * sum( ((x-1)/(x+1))**(2*k+1)/(2*k+1), k=0.. )
-- log 5/4 = 2 * sum( 1/81**k / (2*k+1), k=0.. ) / 9
n := bits() :: N
n <= L10.precision => normalize L10.value
n := n + LENGTH n + 5 -- guard bits
s:I := shift(1,n+1) quo 9
t:I := s quo 81
for k in 3.. by 2 while t ^= 0 repeat
  s := s + t quo k
  t := t quo 81
-- We have log 10 = log 5 + log 2 and log 5/4 = log 5 - 2 log 2
inc 2; L10 := [bits(),[s,-n] + 3*log2]; dec 2
normalize L10.value

log2(x) == (inc 2; r := log(x)/log2; dec 2; normalize r)
log10(x) == (inc 2; r := log(x)/log10; dec 2; normalize r)

exp(x) ==
-- exp(n+x) = exp(1)**n exp(x) for n such that |x| < 1
p := bits(); inc 5; e1:% := 1
if (n := wholePart x) ^= 0 then
  inc LENGTH n; e1 := exp1 ** n; dec LENGTH n
x := fractionPart x
if zero? x then (bits p; return normalize e1)
-- make |x| < 0( 2**(-sqrt p) ) < 1/2 to speed series convergence

```

```

-- by repeated use of the formula  $\exp(2x/2) = \exp(x/2)**2$ 
-- results in an overall running time of  $O(\sqrt{p} M(p))$ 
k := ISQRT (p-100)::I quo 3
k := max(0,2 + k + order x)
if k > 0 then (inc k; x := shift(x,-k))
e := expSeries x
if k > 0 then e := square(e,k)
bits p
e * e1

expSeries x ==
--  $\exp(x) = 1 + x + x^2/2 + \dots + x^i/i!$  valid for all x
p := bits() + LENGTH bits() + 1
s:I := d:I := shift(1,p)
t:I := n:I := shift2(x.mantissa,x.exponent+p)
for i in 2.. while t ^= 0 repeat
  s := s + t
  t := (n * t) quo i
  t := t quo d
normalize [s,-p]

expInverse k ==
-- computes  $\exp(1/k)$  via continued fraction
p0:I := 2*k+1; p1:I := 6*k*p0+1
q0:I := 2*k-1; q1:I := 6*k*q0+1
for i in 10*k.. by 4*k while 2 * LENGTH p0 < bits() repeat
  (p0,p1) := (p1,i*p1+p0)
  (q0,q1) := (q1,i*q1+q0)
ddivide([p1,0],[q1,0])

E:StoredConstant := [1,[1,1]]
exp1() ==
  if bits() > E.precision then E := [bits(),expInverse 1]
  normalize E.value

sqrt x ==
  negative? x => error "negative sqrt"
  m := x.mantissa; e := x.exponent
  l := LENGTH m
  p := 2 * bits() - l + 2
  if odd?(e-l) then p := p - 1
  i := shift2(x.mantissa,p)
  -- ISQRT uses a variable precision newton iteration
  i := ISQRT i
  normalize [i,(e-p) quo 2]

bits() == BITS()
bits(n) == (t := bits(); BITS() := n; t)
precision() == bits()
precision(n) == bits(n)

```

```

increasePrecision n == (b := bits(); bits((b + n)::PI); b)
decreasePrecision n == (b := bits(); bits((b - n)::PI); b)
ceilog10base2 n == ((13301 * n + 4003) quo 4004) :: PI
digits() == max(1,4004 * (bits()-1) quo 13301)::PI
digits(n) == (t := digits(); bits (1 + ceilog10base2 n); t)

order(a) == LENGTH a.mantissa + a.exponent - 1
relererror(a,b) == order((a-b)/b)
0 == [0,0]
1 == [1,0]
base() == BASE
mantissa x == x.mantissa
exponent x == x.exponent
one? a == a = 1
zero? a == zero?(a.mantissa)
negative? a == negative?(a.mantissa)
positive? a == positive?(a.mantissa)

chop(x,p) ==
  e : I := LENGTH x.mantissa - p
  if e > 0 then x := [shift2(x.mantissa,-e),x.exponent+e]
  x
float(m,e) == normalize [m,e]
float(m,e,b) ==
  m = 0 => 0
  inc 4; r := m * [b,0] ** e; dec 4
  normalize r
normalize x ==
  m := x.mantissa
  m = 0 => 0
  e : I := LENGTH m - bits()
  if e > 0 then
    y := shift2(m,1-e)
    if odd? y then
      y := (if y>0 then y+1 else y-1) quo 2
    if LENGTH y > bits() then
      y := y quo 2
      e := e+1
    else y := y quo 2
    x := [y,x.exponent+e]
  x
shift(x:%,n:I) == [x.mantissa,x.exponent+n]

x = y ==
  order x = order y and sign x = sign y and zero? (x - y)
x < y ==
  y.mantissa = 0 => x.mantissa < 0
  x.mantissa = 0 => y.mantissa > 0
  negative? x and positive? y => true
  negative? y and positive? x => false

```

```

order x < order y => positive? x
order x > order y => negative? x
negative? (x-y)

abs x == if negative? x then -x else normalize x
ceiling x ==
  if negative? x then return (-floor(-x))
  if zero? fractionPart x then x else truncate x + 1
wholePart x == shift2(x.mantissa,x.exponent)
floor x == if negative? x then -ceiling(-x) else truncate x
round x == (half := [sign x,-1]; truncate(x + half))
sign x == if x.mantissa < 0 then -1 else 1
truncate x ==
  if x.exponent >= 0 then return x
  normalize [shift2(x.mantissa,x.exponent),0]
recip(x) == if x=0 then "failed" else 1/x
differentiate x == 0

- x == normalize negate x
negate x == [-x.mantissa,x.exponent]
x + y == normalize plus(x,y)
x - y == normalize plus(x,negate y)
sub(x,y) == plus(x,negate y)
plus(x,y) ==
  mx := x.mantissa; my := y.mantissa
  mx = 0 => y
  my = 0 => x
  ex := x.exponent; ey := y.exponent
  ex = ey => [mx+my,ex]
  de := ex + LENGTH mx - ey - LENGTH my
  de > bits()+1 => x
  de < -(bits()+1) => y
  if ex < ey then (mx,my,ex,ey) := (my,mx,ey,ex)
  mw := my + shift2(mx,ex-ey)
  [mw,ey]

x:% * y:% == normalize times (x,y)
x:I * y:% ==
  if LENGTH x > bits() then normalize [x,0] * y
  else normalize [x * y.mantissa,y.exponent]
x:% / y:% == normalize dvide(x,y)
x:% / y:I ==
  if LENGTH y > bits() then x / normalize [y,0] else x / [y,0]
inv x == 1 / x

times(x:%,y:%) == [x.mantissa * y.mantissa, x.exponent + y.exponent]
itimes(n:I,y:%) == [n * y.mantissa,y.exponent]

dvide(x,y) ==
  ew := LENGTH y.mantissa - LENGTH x.mantissa + bits() + 1

```

```

mw := shift2(x.mantissa,ew) quo y.mantissa
ew := x.exponent - y.exponent - ew
[mw,ew]

square(x,n) ==
  ma := x.mantissa; ex := x.exponent
  for k in 1..n repeat
    ma := ma * ma; ex := ex + ex
    l:I := bits()::I - LENGTH ma
    ma := shift2(ma,l); ex := ex - l
  [ma,ex]

power(x,n) ==
  y:% := 1; z:% := x
  repeat
    if odd? n then y := chop( times(y,z), bits() )
    if (n := n quo 2) = 0 then return y
    z := chop( times(z,z), bits() )

x:% ** y:% ==
  x = 0 =>
    y = 0 => error "0**0 is undefined"
    y < 0 => error "division by 0"
    y > 0 => 0
  y = 0 => 1
  y = 1 => x
  x = 1 => 1
  p := abs order y + 5
  inc p; r := exp(y*log(x)); dec p
  normalize r

x:% ** r:RN ==
  x = 0 =>
    r = 0 => error "0**0 is undefined"
    r < 0 => error "division by 0"
    r > 0 => 0
  r = 0 => 1
  r = 1 => x
  x = 1 => 1
  n := numer r
  d := denom r
  negative? x =>
    odd? d =>
      odd? n => return -((-x)**r)
      return ((-x)**r)
      error "negative root"
  if d = 2 then
    inc LENGTH n; y := sqrt(x); y := y**n; dec LENGTH n
    return normalize y
  y := [n,0]/[d,0]

```

```

x ** y

x:% ** n:I ==
  x = 0 =>
    n = 0 => error "0**0 is undefined"
    n < 0 => error "division by 0"
    n > 0 => 0
  n = 0 => 1
  n = 1 => x
  x = 1 => 1
  p := bits()
  bits(p + LENGTH n + 2)
  y := power(x,abs n)
  if n < 0 then y := dvide(1,y)
  bits p
  normalize y

-- Utility routines for conversion to decimal
ceillength10: I -> I
chop10: (% ,I) -> %
convert10:(% ,I) -> %
floorlength10: I -> I
length10: I -> I
normalize10: (% ,I) -> %
quotient10: (% ,% ,I) -> %
power10: (% ,I ,I) -> %
times10: (% ,% ,I) -> %

convert10(x,d) ==
  m := x.mantissa; e := x.exponent
  --!! deal with bits here
  b := bits(); (q,r) := divide(abs e, b)
  b := 2**b::N; r := 2**r::N
  -- compute 2**e = b**q * r
  h := power10([b,0],q,d+5)
  h := chop10([r*h.mantissa,h.exponent],d+5)
  if e < 0 then h := quotient10([m,0],h,d)
  else times10([m,0],h,d)

ceillength10 n == 146 * LENGTH n quo 485 + 1
floorlength10 n == 643 * LENGTH n quo 2136
-- length10 n == DECIMAL_-LENGTH(n)$Lisp
length10 n ==
  ln := LENGTH(n:=abs n)
  upper := 76573 * ln quo 254370
  lower := 21306 * (ln-1) quo 70777
  upper = lower => upper + 1
  n := n quo (10**lower::N)
  while n >= 10 repeat
    n:= n quo 10

```

```

        lower := lower + 1
    lower + 1

chop10(x,p) ==
    e : I := floorLength10 x.mantissa - p
    if e > 0 then x := [x.mantissa quo 10**e::N,x.exponent+e]
    x
normalize10(x,p) ==
    ma := x.mantissa
    ex := x.exponent
    e : I := length10 ma - p
    if e > 0 then
        ma := ma quo 10**(e-1)::N
        ex := ex + e
        (ma,r) := divide(ma, 10)
        if r > 4 then
            ma := ma + 1
            if ma = 10**p::N then (ma := 1; ex := ex + p)
    [ma,ex]
times10(x,y,p) == normalize10(times(x,y),p)
quotient10(x,y,p) ==
    ew := floorLength10 y.mantissa - ceilLength10 x.mantissa + p + 2
    if ew < 0 then ew := 0
    mw := (x.mantissa * 10**ew::N) quo y.mantissa
    ew := x.exponent - y.exponent - ew
    normalize10([mw,ew],p)
power10(x,n,d) ==
    x = 0 => 0
    n = 0 => 1
    n = 1 => x
    x = 1 => 1
    p:I := d + LENGTH n + 1
    e:I := n
    y:% := 1
    z:% := x
    repeat
        if odd? e then y := chop10(times(y,z),p)
        if (e := e quo 2) = 0 then return y
        z := chop10(times(z,z),p)

-----
-- Output routines for Floats --
-----

zero ==> char("0")
separator ==> space()$Character

SPACING : Reference(N) := ref 10
OUTMODE : Reference(S) := ref "general"
OUTPREC : Reference(I) := ref(-1)

```

```

fixed : % -> S
floating : % -> S
general : % -> S

padFromLeft(s:S):S ==
  zero? SPACING() => s
  n:I := #s - 1
  t := new( (n + 1 + n quo SPACING()) :: N , separator )
  for i in 0..n for j in minIndex t .. repeat
    t.j := s.(i + minIndex s)
    if (i+1) rem SPACING() = 0 then j := j+1
  t

padFromRight(s:S):S ==
  SPACING() = 0 => s
  n:I := #s - 1
  t := new( (n + 1 + n quo SPACING()) :: N , separator )
  for i in n..0 by -1 for j in maxIndex t .. by -1 repeat
    t.j := s.(i + minIndex s)
    if (n-i+1) rem SPACING() = 0 then j := j-1
  t

fixed f ==
  d := if OUTPREC() = -1 then digits::I else OUTPREC()
  dpos:N:= if (d > 0) then d::N else 1::N
  zero? f =>
    OUTPREC() = -1 => "0.0"
    concat("0",concat(".",padFromLeft new(dpos,zero)))
  zero? exponent f =>
    concat(padFromRight convert(mantissa f)@S,
      concat(".",padFromLeft new(dpos,zero)))
  negative? f => concat("-", fixed abs f)
  bl := LENGTH(f.mantissa) + f.exponent
  dd :=
    OUTPREC() = -1 => d
    bl > 0 => (146*bl) quo 485 + 1 + d
    d
  g := convert10(abs f,dd)
  m := g.mantissa
  e := g.exponent
  if OUTPREC() ^= -1 then
    -- round g to OUTPREC digits after the decimal point
    l := length10 m
    if -e > OUTPREC() and -e < 2*digits::I then
      g := normalize10(g,l+e+OUTPREC())
      m := g.mantissa; e := g.exponent
  s := convert(m)@S; n := #s; o := e+n
  p := if OUTPREC() = -1 then n::I else OUTPREC()
  t:S
  if e >= 0 then
    s := concat(s, new(e::N, zero))

```



```

    t := ""
  else if o <= 0 then
    t := concat(new((-o)::N,zero), s)
    s := "0"
  else
    t := s(o + minIndex s .. n + minIndex s - 1)
    s := s(minIndex s .. o + minIndex s - 1)
  n := #t
  if OUTPREC() = -1 then
    t := rightTrim(t,zero)
    if t = "" then t := "0"
  else if n > p then t := t(minIndex t .. p + minIndex t - 1)
    else t := concat(t, new((p-n)::N,zero))
  concat(padFromRight s, concat(".", padFromLeft t))

floating f ==
  zero? f => "0.0"
  negative? f => concat("-", floating abs f)
  t:S := if zero? SPACING() then "E" else " E "
  zero? exponent f =>
    s := convert(mantissa f)@S
    concat ["0.", padFromLeft s, t, convert(#s)@S]
  -- base conversion to decimal rounded to the requested precision
  d := if OUTPREC() = -1 then digits::I else OUTPREC()
  g := convert10(f,d); m := g.mantissa; e := g.exponent
  -- I'm assuming that length10 m = # s given n > 0
  s := convert(m)@S; n := #s; o := e+n
  s := padFromLeft s
  concat ["0.", s, t, convert(o)@S]

general(f) ==
  zero? f => "0.0"
  negative? f => concat("-", general abs f)
  d := if OUTPREC() = -1 then digits::I else OUTPREC()
  zero? exponent f =>
    d := d + 1
    s := convert(mantissa f)@S
    OUTPREC() ^=- 1 and (e := #s) > d =>
      t:S := if zero? SPACING() then "E" else " E "
      concat ["0.", padFromLeft s, t, convert(e)@S]
    padFromRight concat(s, ".0")
  -- base conversion to decimal rounded to the requested precision
  g := convert10(f,d); m := g.mantissa; e := g.exponent
  -- I'm assuming that length10 m = # s given n > 0
  s := convert(m)@S; n := #s; o := n + e
  -- Note: at least one digit is displayed after the decimal point
  -- and trailing zeroes after the decimal point are dropped
  if o > 0 and o <= max(n,d) then
    -- fixed format: add trailing zeroes before the decimal point
    if o > n then s := concat(s, new((o-n)::N,zero))

```

```

    t := rightTrim(s(o + minIndex s .. n + minIndex s - 1), zero)
    if t = "" then t := "0" else t := padFromLeft t
    s := padFromRight s(minIndex s .. o + minIndex s - 1)
    concat(s, concat(".", t))
else if o <= 0 and o >= -5 then
    -- fixed format: up to 5 leading zeroes after the decimal point
    concat("0.", padFromLeft concat(new((-o)::N, zero), rightTrim(s, zero)))
else
    -- print using E format written 0.mantissa E exponent
    t := padFromLeft rightTrim(s, zero)
    s := if zero? SPACING() then "E" else " E "
    concat ["0.", t, s, convert(e+n)@S]

outputSpacing n == SPACING() := n
outputFixed() == (OUTMODE() := "fixed"; OUTPREC() := -1)
outputFixed n == (OUTMODE() := "fixed"; OUTPREC() := n::I)
outputGeneral() == (OUTMODE() := "general"; OUTPREC() := -1)
outputGeneral n == (OUTMODE() := "general"; OUTPREC() := n::I)
outputFloating() == (OUTMODE() := "floating"; OUTPREC() := -1)
outputFloating n == (OUTMODE() := "floating"; OUTPREC() := n::I)

convert(f):S ==
    b:Integer :=
        OUTPREC() = -1 and not zero? f =>
            bits(length(abs mantissa f)::PositiveInteger)
        0
    s :=
        OUTMODE() = "fixed" => fixed f
        OUTMODE() = "floating" => floating f
        OUTMODE() = "general" => general f
    empty()$String
    if b > 0 then bits(b::PositiveInteger)
    s = empty()$String => error "bad output mode"
    s

coerce(f):OutputForm ==
    f >= 0 => message(convert(f)@S)
    - (coerce(-f)@OutputForm)

convert(f):InputForm ==
    convert [convert("float"::Symbol), convert mantissa f,
            convert exponent f, convert base()]$List(InputForm)

-- Conversion routines
convert(x:%):Float == x pretend Float
convert(x:%):SF == makeSF(x.mantissa, x.exponent)$Lisp
coerce(x:%):SF == convert(x)@SF
convert(sf:SF):% == float(mantissa sf, exponent sf, base())$SF

retract(f:%):RN == rationalApproximation(f, (bits()-1)::N, BASE)

```

```

retractIfCan(f:%):Union(RN, "failed") ==
  rationalApproximation(f,(bits()-1)::N,BASE)

retract(f:%):I ==
  (f = (n := wholePart f)::%) => n
  error "Not an integer"

retractIfCan(f:%):Union(I, "failed") ==
  (f = (n := wholePart f)::%) => n
  "failed"

rationalApproximation(f,d) == rationalApproximation(f,d,10)

rationalApproximation(f,d,b) ==
  t: Integer
  nu := f.mantissa; ex := f.exponent
  if ex >= 0 then return ((nu*BASE**(ex::N))/1)
  de := BASE**((-ex)::N)
  if b < 2 then error "base must be > 1"
  tol := b**d
  s := nu; t := de
  p0,p1,q0,q1 : Integer
  p0 := 0; p1 := 1; q0 := 1; q1 := 0
  repeat
    (q,r) := divide(s, t)
    p2 := q*p1+p0
    q2 := q*q1+q0
    if r = 0 or tol*abs(nu*q2-de*p2) < de*abs(p2) then return (p2/q2)
    (p0,p1) := (p1,p2)
    (q0,q1) := (q1,q2)
    (s,t) := (t,r)

```

— **FLOAT.dotabb** —

```

"FLOAT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLOAT"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"FLOAT" -> "ALIST"

```

domain FC FortranCode

— FortranCode.input —

```
)set break resume
)sys rm -f FortranCode.output
)spool FortranCode.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FortranCode
--R
--R FortranCode is a domain constructor
--R Abbreviation for FortranCode is FC
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FC
--R
--R----- Operations -----
--R ?? : (%,% ) -> Boolean          assign : (Symbol,String) -> %
--R block : List(% ) -> %          call : String -> %
--R coerce : % -> OutputForm       comment : List(String) -> %
--R comment : String -> %         cond : (Switch,%,% ) -> %
--R cond : (Switch,% ) -> %       continue : SingleInteger -> %
--R getCode : % -> SExpression     goto : SingleInteger -> %
--R hash : % -> SingleInteger      latex : % -> String
--R printCode : % -> Void         returns : Expression(Float) -> %
--R returns : () -> %            save : () -> %
--R stop : () -> %               whileLoop : (Switch,% ) -> %
--R ~=? : (%,% ) -> Boolean
--R assign : (Symbol,List(Polynomial(Integer)),Expression(Complex(Float))) -> %
--R assign : (Symbol,List(Polynomial(Integer)),Expression(Float)) -> %
--R assign : (Symbol,List(Polynomial(Integer)),Expression(Integer)) -> %
--R assign : (Symbol,Vector(Expression(Complex(Float)))) -> %
--R assign : (Symbol,Vector(Expression(Float))) -> %
--R assign : (Symbol,Vector(Expression(Integer))) -> %
--R assign : (Symbol,Matrix(Expression(Complex(Float)))) -> %
--R assign : (Symbol,Matrix(Expression(Float))) -> %
--R assign : (Symbol,Matrix(Expression(Integer))) -> %
--R assign : (Symbol,Expression(Complex(Float))) -> %
--R assign : (Symbol,Expression(Float)) -> %
--R assign : (Symbol,Expression(Integer)) -> %
--R assign : (Symbol,List(Polynomial(Integer)),Expression(MachineComplex)) -> %
--R assign : (Symbol,List(Polynomial(Integer)),Expression(MachineFloat)) -> %
--R assign : (Symbol,List(Polynomial(Integer)),Expression(MachineInteger)) -> %
--R assign : (Symbol,Vector(Expression(MachineComplex))) -> %
--R assign : (Symbol,Vector(Expression(MachineFloat))) -> %
```

```

--R assign : (Symbol,Vector(Expression(MachineInteger))) -> %
--R assign : (Symbol,Matrix(Expression(MachineComplex))) -> %
--R assign : (Symbol,Matrix(Expression(MachineFloat))) -> %
--R assign : (Symbol,Matrix(Expression(MachineInteger))) -> %
--R assign : (Symbol,Vector(MachineComplex)) -> %
--R assign : (Symbol,Vector(MachineFloat)) -> %
--R assign : (Symbol,Vector(MachineInteger)) -> %
--R assign : (Symbol,Matrix(MachineComplex)) -> %
--R assign : (Symbol,Matrix(MachineFloat)) -> %
--R assign : (Symbol,Matrix(MachineInteger)) -> %
--R assign : (Symbol,Expression(MachineComplex)) -> %
--R assign : (Symbol,Expression(MachineFloat)) -> %
--R assign : (Symbol,Expression(MachineInteger)) -> %
--R code : % -> Union(nullBranch: null,assignmentBranch: Record(var: Symbol,arrayIndex: List(Pol
--R common : (Symbol,List(Symbol)) -> %
--R forLoop : (SegmentBinding(Polynomial(Integer)),Polynomial(Integer),%) -> %
--R forLoop : (SegmentBinding(Polynomial(Integer)),%) -> %
--R operation : % -> Union(Null: null,Assignment: assignment,Conditional: conditional,Return: re
--R printStatement : List(OutputForm) -> %
--R repeatUntilLoop : (Switch,%) -> %
--R returns : Expression(Complex(Float)) -> %
--R returns : Expression(Integer) -> %
--R returns : Expression(MachineComplex) -> %
--R returns : Expression(MachineInteger) -> %
--R returns : Expression(MachineFloat) -> %
--R setLabelValue : SingleInteger -> SingleInteger
--R
--E 1

)spool
)lisp (bye)

```

— FortranCode.help —

=====

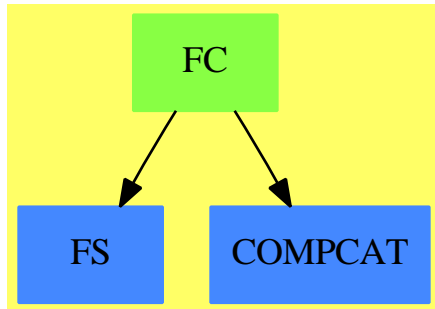
FortranCode examples

=====

This domain builds representations of program code segments for use with the FortranProgram domain.

See Also:

- o)show FortranCode

FortranCode (FC)**See**

- ⇒ “Result” (RESULT) [19](#) on page [2300](#)
- ⇒ “FortranProgram” (FORTRAN) [7](#) on page [955](#)
- ⇒ “ThreeDimensionalMatrix” (M3D) [21](#) on page [2702](#)
- ⇒ “SimpleFortranProgram” (SFORT) [20](#) on page [2405](#)
- ⇒ “Switch” (SWITCH) [20](#) on page [2629](#)
- ⇒ “FortranTemplate” (FTEM) [7](#) on page [967](#)
- ⇒ “FortranExpression” (FEXPR) [7](#) on page [946](#)

Exports:

assign	block	call	code	coerce
comment	common	cond	continue	forLoop
getCode	goto	hash	latex	operation
printCode	printStatement	repeatUntilLoop	returns	save
setLabelValue	stop	whileLoop	?=?	?=?

— domain FC FortranCode —

```

)abbrev domain FC FortranCode
++ Author: Mike Dewar
++ Date Created: April 1991
++ Date Last Updated: 9 January 1995 Added fortran2Lines to getCall, MCD
++ Description:
++ This domain builds representations of program code segments for use with
++ the FortranProgram domain.

```

```

FortranCode(): public == private where
  L ==> List
  PI ==> PositiveInteger
  PIN ==> Polynomial Integer
  SEX ==> SExpression
  O ==> OutputForm
  OP ==> Union(Null:"null",
    Assignment:"assignment",
    Conditional:"conditional",

```

```

    Return:"return",
    Block:"block",
    Comment:"comment",
    Call:"call",
    For:"for",
    While:"while",
    Repeat:"repeat",
    Goto:"goto",
    Continue:"continue",
    ArrayAssignment:"arrayAssignment",
    Save:"save",
    Stop:"stop",
    Common:"common",
    Print:"print")
ARRAYASS ==> Record(var:Symbol, rand:0, ints2Floats?:Boolean)
EXPRESSION ==> Record(ints2Floats?:Boolean,expr:0)
ASS ==> Record(var:Symbol,
    arrayIndex:L PIN,
    rand:EXPRESSION
)
COND ==> Record(switch: Switch(),
    thenClause: $,
    elseClause: $
)
RETURN ==> Record(empty?:Boolean,value:EXPRESSION)
BLOCK ==> List $
COMMENT ==> List String
COMMON ==> Record(name:Symbol,contents:List Symbol)
CALL ==> String
FOR ==> Record(range:SegmentBinding PIN, span:PIN, body:$)
LABEL ==> SingleInteger
LOOP ==> Record(switch:Switch(),body:$)
PRINTLIST ==> List 0
OPREC ==> Union(nullBranch:"null", assignmentBranch:ASS,
    arrayAssignmentBranch:ARRAYASS,
    conditionalBranch:COND, returnBranch:RETURN,
    blockBranch:BLOCK, commentBranch:COMMENT, callBranch:CALL,
    forBranch:FOR, labelBranch:LABEL, loopBranch:LOOP,
    commonBranch:COMMON, printBranch:PRINTLIST)

public == SetCategory with
    coerce: $ -> 0
    ++ coerce(f) returns an object of type OutputForm.
    forLoop: (SegmentBinding PIN,$) -> $
    ++ forLoop(i=1..10,c) creates a representation of a FORTRAN DO loop with
    ++ \spad{i} ranging over the values 1 to 10.
    forLoop: (SegmentBinding PIN,PIN,$) -> $
    ++ forLoop(i=1..10,n,c) creates a representation of a FORTRAN DO loop with
    ++ \spad{i} ranging over the values 1 to 10 by n.
    whileLoop: (Switch,$) -> $

```

```

    ++ whileLoop(s,c) creates a while loop in FORTRAN.
repeatUntilLoop: (Switch,$) -> $
    ++ repeatUntilLoop(s,c) creates a repeat ... until loop in FORTRAN.
goto: SingleInteger -> $
    ++ goto(l) creates a representation of a FORTRAN GOTO statement
continue: SingleInteger -> $
    ++ continue(l) creates a representation of a FORTRAN CONTINUE labelled
    ++ with l
comment: String -> $
    ++ comment(s) creates a representation of the String s as a single FORTRAN
    ++ comment.
comment: List String -> $
    ++ comment(s) creates a representation of the Strings s as a multi-line
    ++ FORTRAN comment.
call: String -> $
    ++ call(s) creates a representation of a FORTRAN CALL statement
returns: () -> $
    ++ returns() creates a representation of a FORTRAN RETURN statement.
returns: Expression MachineFloat -> $
    ++ returns(e) creates a representation of a FORTRAN RETURN statement
    ++ with a returned value.
returns: Expression MachineInteger -> $
    ++ returns(e) creates a representation of a FORTRAN RETURN statement
    ++ with a returned value.
returns: Expression MachineComplex -> $
    ++ returns(e) creates a representation of a FORTRAN RETURN statement
    ++ with a returned value.
returns: Expression Float -> $
    ++ returns(e) creates a representation of a FORTRAN RETURN statement
    ++ with a returned value.
returns: Expression Integer -> $
    ++ returns(e) creates a representation of a FORTRAN RETURN statement
    ++ with a returned value.
returns: Expression Complex Float -> $
    ++ returns(e) creates a representation of a FORTRAN RETURN statement
    ++ with a returned value.
cond: (Switch,$) -> $
    ++ cond(s,e) creates a representation of the FORTRAN expression
    ++ IF (s) THEN e.
cond: (Switch,$,$) -> $
    ++ cond(s,e,f) creates a representation of the FORTRAN expression
    ++ IF (s) THEN e ELSE f.
assign: (Symbol,String) -> $
    ++ assign(x,y) creates a representation of the FORTRAN expression
    ++ x=y.
assign: (Symbol,Expression MachineInteger) -> $
    ++ assign(x,y) creates a representation of the FORTRAN expression
    ++ x=y.
assign: (Symbol,Expression MachineFloat) -> $
    ++ assign(x,y) creates a representation of the FORTRAN expression

```



```

++ x=y.
assign: (Symbol,Expression MachineComplex) -> $
++ assign(x,y) creates a representation of the FORTRAN expression
++ x=y.
assign: (Symbol,Matrix MachineInteger) -> $
++ assign(x,y) creates a representation of the FORTRAN expression
++ x=y.
assign: (Symbol,Matrix MachineFloat) -> $
++ assign(x,y) creates a representation of the FORTRAN expression
++ x=y.
assign: (Symbol,Matrix MachineComplex) -> $
++ assign(x,y) creates a representation of the FORTRAN expression
++ x=y.
assign: (Symbol,Vector MachineInteger) -> $
++ assign(x,y) creates a representation of the FORTRAN expression
++ x=y.
assign: (Symbol,Vector MachineFloat) -> $
++ assign(x,y) creates a representation of the FORTRAN expression
++ x=y.
assign: (Symbol,Vector MachineComplex) -> $
++ assign(x,y) creates a representation of the FORTRAN expression
++ x=y.
assign: (Symbol,Matrix Expression MachineInteger) -> $
++ assign(x,y) creates a representation of the FORTRAN expression
++ x=y.
assign: (Symbol,Matrix Expression MachineFloat) -> $
++ assign(x,y) creates a representation of the FORTRAN expression
++ x=y.
assign: (Symbol,Matrix Expression MachineComplex) -> $
++ assign(x,y) creates a representation of the FORTRAN expression
++ x=y.
assign: (Symbol,Vector Expression MachineInteger) -> $
++ assign(x,y) creates a representation of the FORTRAN expression
++ x=y.
assign: (Symbol,Vector Expression MachineFloat) -> $
++ assign(x,y) creates a representation of the FORTRAN expression
++ x=y.
assign: (Symbol,Vector Expression MachineComplex) -> $
++ assign(x,y) creates a representation of the FORTRAN expression
++ x=y.
assign: (Symbol,L PIN,Expression MachineInteger) -> $
++ assign(x,l,y) creates a representation of the assignment of \spad{y}
++ to the \spad{l}'th element of array \spad{x} (\spad{l} is a list of
++ indices).
assign: (Symbol,L PIN,Expression MachineFloat) -> $
++ assign(x,l,y) creates a representation of the assignment of \spad{y}
++ to the \spad{l}'th element of array \spad{x} (\spad{l} is a list of
++ indices).
assign: (Symbol,L PIN,Expression MachineComplex) -> $
++ assign(x,l,y) creates a representation of the assignment of \spad{y}

```

```

    ++ to the \spad{1}'th element of array \spad{x} (\spad{1} is a list of
    ++ indices).
assign: (Symbol,Expression Integer) -> $
    ++ assign(x,y) creates a representation of the FORTRAN expression
    ++ x=y.
assign: (Symbol,Expression Float) -> $
    ++ assign(x,y) creates a representation of the FORTRAN expression
    ++ x=y.
assign: (Symbol,Expression Complex Float) -> $
    ++ assign(x,y) creates a representation of the FORTRAN expression
    ++ x=y.
assign: (Symbol,Matrix Expression Integer) -> $
    ++ assign(x,y) creates a representation of the FORTRAN expression
    ++ x=y.
assign: (Symbol,Matrix Expression Float) -> $
    ++ assign(x,y) creates a representation of the FORTRAN expression
    ++ x=y.
assign: (Symbol,Matrix Expression Complex Float) -> $
    ++ assign(x,y) creates a representation of the FORTRAN expression
    ++ x=y.
assign: (Symbol,Vector Expression Integer) -> $
    ++ assign(x,y) creates a representation of the FORTRAN expression
    ++ x=y.
assign: (Symbol,Vector Expression Float) -> $
    ++ assign(x,y) creates a representation of the FORTRAN expression
    ++ x=y.
assign: (Symbol,Vector Expression Complex Float) -> $
    ++ assign(x,y) creates a representation of the FORTRAN expression
    ++ x=y.
assign: (Symbol,L PIN,Expression Integer) -> $
    ++ assign(x,l,y) creates a representation of the assignment of \spad{y}
    ++ to the \spad{1}'th element of array \spad{x} (\spad{1} is a list of
    ++ indices).
assign: (Symbol,L PIN,Expression Float) -> $
    ++ assign(x,l,y) creates a representation of the assignment of \spad{y}
    ++ to the \spad{1}'th element of array \spad{x} (\spad{1} is a list of
    ++ indices).
assign: (Symbol,L PIN,Expression Complex Float) -> $
    ++ assign(x,l,y) creates a representation of the assignment of \spad{y}
    ++ to the \spad{1}'th element of array \spad{x} (\spad{1} is a list of
    ++ indices).
block: List($ ) -> $
    ++ block(l) creates a representation of the statements in l as a block.
stop: () -> $
    ++ stop() creates a representation of a STOP statement.
save: () -> $
    ++ save() creates a representation of a SAVE statement.
printStatement: List 0 -> $
    ++ printStatement(l) creates a representation of a PRINT statement.
common: (Symbol,List Symbol) -> $

```

```

    ++ common(name,contents) creates a representation a named common block.
operation: $ -> OP
    ++ operation(f) returns the name of the operation represented by \spad{f}.
code: $ -> OPREC
    ++ code(f) returns the internal representation of the object represented
    ++ by \spad{f}.
printCode: $ -> Void
    ++ printCode(f) prints out \spad{f} in FORTRAN notation.
getCode: $ -> SEX
    ++ getCode(f) returns a Lisp list of strings representing \spad{f}
    ++ in Fortran notation. This is used by the FortranProgram domain.
setLabelValue:SingleInteger -> SingleInteger
    ++ setLabelValue(i) resets the counter which produces labels to i

private == add
import Void
import ASS
import COND
import RETURN
import L PIN
import O
import SEX
import FortranType
import TheSymbolTable

Rep := Record(op: OP, data: OPREC)

-- We need to be able to generate unique labels
labelValue:SingleInteger := 25000::SingleInteger
setLabelValue(u:SingleInteger):SingleInteger == labelValue := u
newLabel():SingleInteger ==
    labelValue := labelValue + 1$SingleInteger
    labelValue

commaSep(l>List String):List(String) ==
    [(l.1),:[:["",u] for u in rest(l)]]

getReturn(rec:RETURN):SEX ==
    returnToken : SEX := convert("RETURN":Symbol::O)$SEX
    elt(rec,empty?)$RETURN =>
        getStatement(returnToken,NIL$Lisp)$Lisp
    rt : EXPRESSION := elt(rec,value)$RETURN
    rv : O := elt(rt,expr)$EXPRESSION
    getStatement([returnToken,convert(rv)$SEX]$Lisp,
        elt(rt,ints2Floats?)$EXPRESSION )$Lisp

getStop():SEX ==
    fortran2Lines(LIST("STOP")$Lisp)$Lisp

getSave():SEX ==

```

```

fortran2Lines(LIST("SAVE")$Lisp)$Lisp

getCommon(u:COMMON):SEX ==
  fortran2Lines(APPEND(LIST("COMMON"," /",string (u.name),"/ ")$Lisp,_
    addCommas(u.contents)$Lisp)$Lisp)$Lisp

getPrint(l:PRINTLIST):SEX ==
  ll : SEX := LIST("PRINT*")$Lisp
  for i in l repeat
    ll := APPEND(ll,CONS(", ",expression2Fortran(i)$Lisp)$Lisp)$Lisp
  fortran2Lines(ll)$Lisp

getBlock(rec:BLOCK):SEX ==
  indentFortLevel(convert(1@Integer)$SEX)$Lisp
  expr : SEX := LIST()$Lisp
  for u in rec repeat
    expr := APPEND(expr,getCode(u))$Lisp
  indentFortLevel(convert(-1@Integer)$SEX)$Lisp
  expr

getBody(f:$):SEX ==
  operation(f) case Block => getCode f
  indentFortLevel(convert(1@Integer)$SEX)$Lisp
  expr := getCode f
  indentFortLevel(convert(-1@Integer)$SEX)$Lisp
  expr

getElseIf(f:$):SEX ==
  rec := code f
  expr :=
    fortFormatElseIf(elt(rec.conditionalBranch,switch)$COND::0)$Lisp
  expr :=
    APPEND(expr,getBody elt(rec.conditionalBranch,thenClause)$COND)$Lisp
  elseBranch := elt(rec.conditionalBranch,elseClause)$COND
  not(operation(elseBranch) case Null) =>
    operation(elseBranch) case Conditional =>
      APPEND(expr,getElseIf elseBranch)$Lisp
  expr := APPEND(expr, getStatement(ELSE::0,NIL$Lisp)$Lisp)$Lisp
  expr := APPEND(expr, getBody elseBranch)$Lisp
  expr

getContinue(label:SingleInteger):SEX ==
  lab : 0 := label::0
  if (width(lab) > 6) then error "Label too big"
  cnt : 0 := "CONTINUE":0
  --sp : 0 := hspace(6-width lab)
  sp : 0 := hspace(_$fortIndent$Lisp -width lab)
  LIST(STRCONC(STRINGIMAGE(lab)$Lisp,sp,cnt)$Lisp)$Lisp

getGoto(label:SingleInteger):SEX ==

```

```

fortran2Lines(
  LIST(STRCONC("GOTO ",STRINGIMAGE(label::0)$Lisp)$Lisp)$Lisp)$Lisp

getRepeat(repRec:LOOP):SEX ==
  sw : Switch := NOT elt(repRec,switch)$LOOP
  lab := newLabel()
  bod := elt(repRec,body)$LOOP
  APPEND(getContinue lab,getBody bod,
    fortFormatIfGoto(sw::0,lab)$Lisp)$Lisp

getWhile(whileRec:LOOP):SEX ==
  sw := NOT elt(whileRec,switch)$LOOP
  lab1 := newLabel()
  lab2 := newLabel()
  bod := elt(whileRec,body)$LOOP
  APPEND(fortFormatLabelledIfGoto(sw::0,lab1,lab2)$Lisp,
    getBody bod, getBody goto(lab1), getContinue lab2)$Lisp

getArrayAssign(rec:ARRAYASS):SEX ==
  getfortarrayexp((rec.var)::0,rec.rand,rec.ints2Floats?)$Lisp

getAssign(rec:ASS):SEX ==
  indices : L PIN := elt(rec,arrayIndex)$ASS
  if indices = []::(L PIN) then
    lhs := elt(rec,var)$ASS::0
  else
    lhs := cons(elt(rec,var)$ASS::PIN,indices)::0
    -- Must get the index brackets correct:
    lhs := (cdr car cdr convert(lhs)$SEX::SEX)::0 -- Yuck!
  elt(elt(rec,rnd)$ASS,ints2Floats?)$EXPRESSION =>
    assignment2Fortran1(lhs,elt(elt(rec,rnd)$ASS,expr)$EXPRESSION)$Lisp
  integerAssignment2Fortran1(lhs,elt(elt(rec,rnd)$ASS,expr)$EXPRESSION)$Lisp

getCond(rec:COND):SEX ==
  expr := APPEND(fortFormatIf(elt(rec,switch)$COND::0)$Lisp,
    getBody elt(rec,thenClause)$COND)$Lisp
  elseBranch := elt(rec,elseClause)$COND
  if not(operation(elseBranch) case Null) then
    operation(elseBranch) case Conditional =>
      expr := APPEND(expr,getElseIf elseBranch)$Lisp
      expr := APPEND(expr,getStateMENT(ELSE::0,NIL$Lisp)$Lisp,
        getBody elseBranch)$Lisp
  APPEND(expr,getStateMENT(ENDIF::0,NIL$Lisp)$Lisp)$Lisp

getComment(rec:COMMENT):SEX ==
  convert([convert(concat("C      ",c)$String)$SEX for c in rec])$SEX

getCall(rec:CALL):SEX ==
  expr := concat("CALL ",rec)$String
  #expr > 1320 => error "Fortran CALL too large"

```

```

fortran2Lines(convert([convert(expr)@SEX ]>@SEX)$Lisp

getFor(rec:FOR):SEX ==
  rng : SegmentBinding PIN := elt(rec,range)$FOR
  increment : PIN := elt(rec,span)$FOR
  lab : SingleInteger := newLabel()
  declare!(variable rng,fortranInteger())
  expr := fortFormatDo(variable rng, (lo segment rng)::0,_
    (hi segment rng)::0,increment::0,lab)$Lisp
  APPEND(expr, getBody elt(rec,body)$FOR, getContinue(lab))$Lisp

getCode(f:$):SEX ==
  opp:OP := operation f
  rec:OPREC:= code f
  opp case Assignment => getAssign(rec.assignmentBranch)
  opp case ArrayAssignment => getArrayAssign(rec.arrayAssignmentBranch)
  opp case Conditional => getCond(rec.conditionalBranch)
  opp case Return => getReturn(rec.returnBranch)
  opp case Block => getBlock(rec.blockBranch)
  opp case Comment => getComment(rec.commentBranch)
  opp case Call => getCall(rec.callBranch)
  opp case For => getFor(rec.forBranch)
  opp case Continue => getContinue(rec.labelBranch)
  opp case Goto => getGoto(rec.labelBranch)
  opp case Repeat => getRepeat(rec.loopBranch)
  opp case While => getWhile(rec.loopBranch)
  opp case Save => getSave()
  opp case Stop => getStop()
  opp case Print => getPrint(rec.printBranch)
  opp case Common => getCommon(rec.commonBranch)
  error "Unsupported program construct."
  convert(0)@SEX

printCode(f:$):Void ==
  displayLines1$Lisp getCode f
  void()$Void

code (f:$):OPREC ==
  elt(f,data)$Rep

operation (f:$):OP ==
  elt(f,op)$Rep

common(name:Symbol,contents:List Symbol):$ ==
  [["common"]$OP,[[name,contents]$COMMON]$OPREC]$Rep

stop():$ ==
  [["stop"]$OP,["null"]$OPREC]$Rep

save():$ ==

```

```

    ["save"]$OP,["null"]$OPREC]$Rep

printStats(l:List 0):$ ==
    ["print"]$OP,[l]$OPREC]$Rep

comment(s:List String):$ ==
    ["comment"]$OP,[s]$OPREC]$Rep

comment(s:String):$ ==
    ["comment"]$OP,[list s]$OPREC]$Rep

forLoop(r:SegmentBinding PIN,body:$):$ ==
    ["for"]$OP,[r,(incr segment r)::PIN,body]$FOR]$OPREC]$Rep

forLoop(r:SegmentBinding PIN,increment:PIN,body:$):$ ==
    ["for"]$OP,[r,increment,body]$FOR]$OPREC]$Rep

goto(l:SingleInteger):$ ==
    ["goto"]$OP,[l]$OPREC]$Rep

continue(l:SingleInteger):$ ==
    ["continue"]$OP,[l]$OPREC]$Rep

whileLoop(sw:Switch,b:$):$ ==
    ["while"]$OP,[sw,b]$LOOP]$OPREC]$Rep

repeatUntilLoop(sw:Switch,b:$):$ ==
    ["repeat"]$OP,[sw,b]$LOOP]$OPREC]$Rep

returns():$ ==
    v := [false,0::0]$EXPRESSION
    ["return"]$OP,[true,v]$RETURN]$OPREC]$Rep

returns(v:Expression MachineInteger):$ ==
    ["return"]$OP,[false,[false,v::0]$EXPRESSION]$RETURN]$OPREC]$Rep

returns(v:Expression MachineFloat):$ ==
    ["return"]$OP,[false,[false,v::0]$EXPRESSION]$RETURN]$OPREC]$Rep

returns(v:Expression MachineComplex):$ ==
    ["return"]$OP,[false,[false,v::0]$EXPRESSION]$RETURN]$OPREC]$Rep

returns(v:Expression Integer):$ ==
    ["return"]$OP,[false,[false,v::0]$EXPRESSION]$RETURN]$OPREC]$Rep

returns(v:Expression Float):$ ==
    ["return"]$OP,[false,[false,v::0]$EXPRESSION]$RETURN]$OPREC]$Rep

returns(v:Expression Complex Float):$ ==
    ["return"]$OP,[false,[false,v::0]$EXPRESSION]$RETURN]$OPREC]$Rep

```

```

block(l:List $):$ ==
  [["block"]$OP,[1]$OPREC]$Rep

cond(sw:Switch,thenC:$):$ ==
  [["conditional"]$OP,
    [[sw,thenC,["null"]$OP,["null"]$OPREC]$Rep]$COND]$OPREC]$Rep

cond(sw:Switch,thenC:$,elseC:$):$ ==
  [["conditional"]$OP,[[sw,thenC,elseC]$COND]$OPREC]$Rep

coerce(f : $):0 ==
  (f.op)::0

assign(v:Symbol,rhs:String):$ ==
  [["assignment"]$OP,[[v,nil()::L PIN,[false,rhs::0]$EXPRESSION]$ASS]$OPREC]$Rep

assign(v:Symbol,rhs:Matrix MachineInteger):$ ==
  [["arrayAssignment"]$OP,[[v,rhs::0,false]$ARRAYASS]$OPREC]$Rep

assign(v:Symbol,rhs:Matrix MachineFloat):$ ==
  [["arrayAssignment"]$OP,[[v,rhs::0,true]$ARRAYASS]$OPREC]$Rep

assign(v:Symbol,rhs:Matrix MachineComplex):$ ==
  [["arrayAssignment"]$OP,[[v,rhs::0,true]$ARRAYASS]$OPREC]$Rep

assign(v:Symbol,rhs:Vector MachineInteger):$ ==
  [["arrayAssignment"]$OP,[[v,rhs::0,false]$ARRAYASS]$OPREC]$Rep

assign(v:Symbol,rhs:Vector MachineFloat):$ ==
  [["arrayAssignment"]$OP,[[v,rhs::0,true]$ARRAYASS]$OPREC]$Rep

assign(v:Symbol,rhs:Vector MachineComplex):$ ==
  [["arrayAssignment"]$OP,[[v,rhs::0,true]$ARRAYASS]$OPREC]$Rep

assign(v:Symbol,rhs:Matrix Expression MachineInteger):$ ==
  [["arrayAssignment"]$OP,[[v,rhs::0,false]$ARRAYASS]$OPREC]$Rep

assign(v:Symbol,rhs:Matrix Expression MachineFloat):$ ==
  [["arrayAssignment"]$OP,[[v,rhs::0,true]$ARRAYASS]$OPREC]$Rep

assign(v:Symbol,rhs:Matrix Expression MachineComplex):$ ==
  [["arrayAssignment"]$OP,[[v,rhs::0,true]$ARRAYASS]$OPREC]$Rep

assign(v:Symbol,rhs:Vector Expression MachineInteger):$ ==
  [["arrayAssignment"]$OP,[[v,rhs::0,false]$ARRAYASS]$OPREC]$Rep

assign(v:Symbol,rhs:Vector Expression MachineFloat):$ ==
  [["arrayAssignment"]$OP,[[v,rhs::0,true]$ARRAYASS]$OPREC]$Rep

```



```

assign(v:Symbol,rhs:Vector Expression MachineComplex):$ ==
  ["arrayAssignment"]$OP,[[v,rhs::0,true]$ARRAYASS]$OPREC]$Rep

assign(v:Symbol,index:L PIN,rhs:Expression MachineInteger):$ ==
  ["assignment"]$OP,[[v,index,[false,rhs::0]$EXPRESSION]$ASS]$OPREC]$Rep

assign(v:Symbol,index:L PIN,rhs:Expression MachineFloat):$ ==
  ["assignment"]$OP,[[v,index,[true,rhs::0]$EXPRESSION]$ASS]$OPREC]$Rep

assign(v:Symbol,index:L PIN,rhs:Expression MachineComplex):$ ==
  ["assignment"]$OP,[[v,index,[true,rhs::0]$EXPRESSION]$ASS]$OPREC]$Rep

assign(v:Symbol,rhs:Expression MachineInteger):$ ==
  ["assignment"]$OP,[[v,nil()::L PIN,[false,rhs::0]$EXPRESSION]$ASS]$OPREC]$Rep

assign(v:Symbol,rhs:Expression MachineFloat):$ ==
  ["assignment"]$OP,[[v,nil()::L PIN,[true,rhs::0]$EXPRESSION]$ASS]$OPREC]$Rep

assign(v:Symbol,rhs:Expression MachineComplex):$ ==
  ["assignment"]$OP,[[v,nil()::L PIN,[true,rhs::0]$EXPRESSION]$ASS]$OPREC]$Rep

assign(v:Symbol,rhs:Matrix Expression Integer):$ ==
  ["arrayAssignment"]$OP,[[v,rhs::0,false]$ARRAYASS]$OPREC]$Rep

assign(v:Symbol,rhs:Matrix Expression Float):$ ==
  ["arrayAssignment"]$OP,[[v,rhs::0,true]$ARRAYASS]$OPREC]$Rep

assign(v:Symbol,rhs:Matrix Expression Complex Float):$ ==
  ["arrayAssignment"]$OP,[[v,rhs::0,true]$ARRAYASS]$OPREC]$Rep

assign(v:Symbol,rhs:Vector Expression Integer):$ ==
  ["arrayAssignment"]$OP,[[v,rhs::0,false]$ARRAYASS]$OPREC]$Rep

assign(v:Symbol,rhs:Vector Expression Float):$ ==
  ["arrayAssignment"]$OP,[[v,rhs::0,true]$ARRAYASS]$OPREC]$Rep

assign(v:Symbol,rhs:Vector Expression Complex Float):$ ==
  ["arrayAssignment"]$OP,[[v,rhs::0,true]$ARRAYASS]$OPREC]$Rep

assign(v:Symbol,index:L PIN,rhs:Expression Integer):$ ==
  ["assignment"]$OP,[[v,index,[false,rhs::0]$EXPRESSION]$ASS]$OPREC]$Rep

assign(v:Symbol,index:L PIN,rhs:Expression Float):$ ==
  ["assignment"]$OP,[[v,index,[true,rhs::0]$EXPRESSION]$ASS]$OPREC]$Rep

assign(v:Symbol,index:L PIN,rhs:Expression Complex Float):$ ==
  ["assignment"]$OP,[[v,index,[true,rhs::0]$EXPRESSION]$ASS]$OPREC]$Rep

assign(v:Symbol,rhs:Expression Integer):$ ==
  ["assignment"]$OP,[[v,nil()::L PIN,[false,rhs::0]$EXPRESSION]$ASS]$OPREC]$Rep

```

```

assign(v:Symbol,rhs:Expression Float):$ ==
  [{"assignment"}$OP,[[v,nil()::L PIN,[true,rhs::0]$EXPRESSION]$ASS]$OPREC]$Rep

assign(v:Symbol,rhs:Expression Complex Float):$ ==
  [{"assignment"}$OP,[[v,nil()::L PIN,[true,rhs::0]$EXPRESSION]$ASS]$OPREC]$Rep

call(s:String):$ ==
  [{"call"}$OP,[s]$OPREC]$Rep

```

— FC.dotabb —

```

"FC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FC"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"FC" -> "COMPCAT"
"FC" -> "FS"

```

domain FEXPR FortranExpression

— FortranExpression.input —

```

)set break resume
)sys rm -f FortranExpression.output
)spool FortranExpression.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FortranExpression
--R
--R FortranExpression(basicSymbols: List(Symbol),subscriptedSymbols: List(Symbol),R: FortranMachineTypeCate
--R Abbreviation for FortranExpression is FEXPR
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FEXPR
--R
--R----- Operations -----
--R ?? : (PositiveInteger,%) -> %      ?? : (Integer,%) -> %
--R ?? : (%,%) -> %                  ?? : (%,R) -> %

```

```

--R ?? : (R,%) -> %
--R ?+? : (%,%) -> %
--R ?-? : (%,%) -> %
--R ?<=? : (%,%) -> Boolean
--R ?>? : (%,%) -> Boolean
--R D : (%,Symbol) -> %
--R 1 : () -> %
--R ?^? : (%,PositiveInteger) -> %
--R acos : % -> %
--R atan : % -> %
--R box : List(%) -> %
--R coerce : % -> Expression(R)
--R coerce : R -> %
--R coerce : % -> OutputForm
--R cosh : % -> %
--R distribute : (%,%) -> %
--R elt : (BasicOperator,%,%) -> %
--R eval : (%,%,%) -> %
--R eval : (%,Kernel(%),%) -> %
--R freeOf? : (%,Symbol) -> Boolean
--R hash : % -> SingleInteger
--R is? : (%,Symbol) -> Boolean
--R kernels : % -> List(Kernel(%))
--R log : % -> %
--R max : (%,%) -> %
--R one? : % -> Boolean
--R paren : % -> %
--R recip : % -> Union(%, "failed")
--R retract : Expression(R) -> %
--R retract : % -> Kernel(%)
--R sin : % -> %
--R sqrt : % -> %
--R tan : % -> %
--R tower : % -> List(Kernel(%))
--R variables : % -> List(Symbol)
--R ?^=? : (%,%) -> Boolean
--R ??? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R D : (%,Symbol,NonNegativeInteger) -> %
--R D : (%,List(Symbol),List(NonNegativeInteger)) -> %
--R ?^? : (%,NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R definingPolynomial : % -> % if $ has RING
--R differentiate : (%,List(Symbol)) -> %
--R differentiate : (%,Symbol,NonNegativeInteger) -> %
--R differentiate : (%,List(Symbol),List(NonNegativeInteger)) -> %
--R elt : (BasicOperator,List(%)) -> %
--R elt : (BasicOperator,%,%,%,%) -> %
--R elt : (BasicOperator,%,%,%) -> %
--R eval : (%,BasicOperator,(% -> %)) -> %

***? : (%,PositiveInteger) -> %
-? : % -> %
?<? : (%,%) -> Boolean
?=? : (%,%) -> Boolean
?>=? : (%,%) -> Boolean
D : (%,List(Symbol)) -> %
0 : () -> %
abs : % -> %
asin : % -> %
belong? : BasicOperator -> Boolean
box : % -> %
coerce : Integer -> %
coerce : Kernel(%) -> %
cos : % -> %
differentiate : (%,Symbol) -> %
distribute : % -> %
elt : (BasicOperator,%) -> %
eval : (%,Equation(%)) -> %
exp : % -> %
freeOf? : (%,%) -> Boolean
height : % -> NonNegativeInteger
kernel : (BasicOperator,%) -> %
latex : % -> String
log10 : % -> %
min : (%,%) -> %
paren : List(%) -> %
pi : () -> %
retract : Symbol -> %
retract : % -> R
sample : () -> %
sinh : % -> %
subst : (%,Equation(%)) -> %
tanh : % -> %
useNagFunctions : () -> Boolean
zero? : % -> Boolean

```

```

--R eval : (% , BasicOperator , (List(%) -> %)) -> %
--R eval : (% , List(BasicOperator) , List((List(%) -> %))) -> %
--R eval : (% , List(BasicOperator) , List((% -> %))) -> %
--R eval : (% , Symbol , (% -> %)) -> %
--R eval : (% , Symbol , (List(%) -> %)) -> %
--R eval : (% , List(Symbol) , List((List(%) -> %))) -> %
--R eval : (% , List(Symbol) , List((% -> %))) -> %
--R eval : (% , List(%) , List(%)) -> %
--R eval : (% , List(Equation(%))) -> %
--R eval : (% , List(Kernel(%)) , List(%)) -> %
--R even? : % -> Boolean if $ has RETRACT(INT)
--R is? : (% , BasicOperator) -> Boolean
--R kernel : (BasicOperator , List(%)) -> %
--R mainKernel : % -> Union(Kernel(%), "failed")
--R map : ((% -> %), Kernel(%)) -> %
--R minPoly : Kernel(%) -> SparseUnivariatePolynomial(%) if $ has RING
--R odd? : % -> Boolean if $ has RETRACT(INT)
--R operator : BasicOperator -> BasicOperator
--R operators : % -> List(BasicOperator)
--R retract : Polynomial(Float) -> % if R has RETRACT(FLOAT)
--R retract : Fraction(Polynomial(Float)) -> % if R has RETRACT(FLOAT)
--R retract : Expression(Float) -> % if R has RETRACT(FLOAT)
--R retract : Polynomial(Integer) -> % if R has RETRACT(INT)
--R retract : Fraction(Polynomial(Integer)) -> % if R has RETRACT(INT)
--R retract : Expression(Integer) -> % if R has RETRACT(INT)
--R retractIfCan : Polynomial(Float) -> Union(%, "failed") if R has RETRACT(FLOAT)
--R retractIfCan : Fraction(Polynomial(Float)) -> Union(%, "failed") if R has RETRACT(FLOAT)
--R retractIfCan : Expression(Float) -> Union(%, "failed") if R has RETRACT(FLOAT)
--R retractIfCan : Polynomial(Integer) -> Union(%, "failed") if R has RETRACT(INT)
--R retractIfCan : Fraction(Polynomial(Integer)) -> Union(%, "failed") if R has RETRACT(INT)
--R retractIfCan : Expression(Integer) -> Union(%, "failed") if R has RETRACT(INT)
--R retractIfCan : Symbol -> Union(%, "failed")
--R retractIfCan : Expression(R) -> Union(%, "failed")
--R retractIfCan : % -> Union(R, "failed")
--R retractIfCan : % -> Union(Kernel(%), "failed")
--R subst : (% , List(Kernel(%)) , List(%)) -> %
--R subst : (% , List(Equation(%))) -> %
--R subtractIfCan : (% , %) -> Union(%, "failed")
--R useNagFunctions : Boolean -> Boolean
--R
--E 1

```

```

)spool
)lisp (bye)

```

```
=====
FortranExpression examples
=====
```

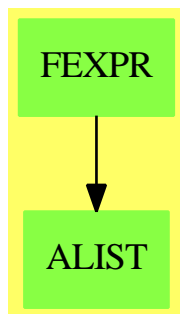
A domain of expressions involving functions which can be translated into standard Fortran-77, with some extra extensions from the NAG Fortran Library.

See Also:

o)show FortranExpression

—————→

FortranExpression (FEXPR)



See

- ⇒ “Result” (RESULT) [19](#) on page [2300](#)
- ⇒ “FortranCode” (FC) [7](#) on page [931](#)
- ⇒ “FortranProgram” (FORTRAN) [7](#) on page [955](#)
- ⇒ “ThreeDimensionalMatrix” (M3D) [21](#) on page [2702](#)
- ⇒ “SimpleFortranProgram” (SFORT) [20](#) on page [2405](#)
- ⇒ “Switch” (SWITCH) [20](#) on page [2629](#)
- ⇒ “FortranTemplate” (FTEM) [7](#) on page [967](#)

Exports:

0	1	abs	acos	asin
atan	belong?	box	characteristic	coerce
cos	cosh	D	definingPolynomial	differentiate
distribute	elt	eval	even?	exp
freeOf?	hash	height	is?	kernel
kernels	latex	log	log10	mainKernel
map	max	min	minPoly	odd?
one?	operator	operators	paren	pi
recip	retract	retractIfCan	sample	sin
sinh	sqrt	subst	subtractIfCan	tan
tanh	tower	useNagFunctions	variables	zero?
?*?	?**?	?+?	-?	?-?
?<?	?<=?	?=?	?>?	?>=?
?^?	?~=?			

— domain FEXPR FortranExpression —

```
)abbrev domain FEXPR FortranExpression
++ Author: Mike Dewar
++ Date Created: December 1993
++ Date Last Updated: 12 July 1994 added RetractableTo(R)
++ Description:
++ A domain of expressions involving functions which can be
++ translated into standard Fortran-77, with some extra extensions from
++ the NAG Fortran Library.
```

```
FortranExpression(basicSymbols,subscriptedSymbols,R):
    Exports==Implementation where
```

```
basicSymbols : List Symbol
subscriptedSymbols : List Symbol
R : FortranMachineTypeCategory
```

```
EXPR ==> Expression
EXF2 ==> ExpressionFunctions2
S ==> Symbol
L ==> List
BO ==> BasicOperator
FRAC ==> Fraction
POLY ==> Polynomial
```

```
Exports ==> Join(ExpressionSpace,Algebra(R),RetractableTo(R),
    PartialDifferentialRing(Symbol)) with
retract : EXPR R -> $
++ retract(e) takes e and transforms it into a
++ FortranExpression checking that it contains no non-Fortran
++ functions, and that it only contains the given basic symbols
++ and subscripted symbols which correspond to scalar and array
++ parameters respectively.
```

```

retractIfCan : EXPR R -> Union($,"failed")
  ++ retractIfCan(e) takes e and tries to transform it into a
  ++ FortranExpression checking that it contains no non-Fortran
  ++ functions, and that it only contains the given basic symbols
  ++ and subscripted symbols which correspond to scalar and array
  ++ parameters respectively.
retract : S -> $
  ++ retract(e) takes e and transforms it into a FortranExpression
  ++ checking that it is one of the given basic symbols
  ++ or subscripted symbols which correspond to scalar and array
  ++ parameters respectively.
retractIfCan : S -> Union($,"failed")
  ++ retractIfCan(e) takes e and tries to transform it into a
  ++ FortranExpression checking that it is one of the given basic symbols
  ++ or subscripted symbols which correspond to scalar and array
  ++ parameters respectively.
coerce : $ -> EXPR R
  ++ coerce(x) is not documented
if (R has RetractableTo(Integer)) then
  retract : EXPR Integer -> $
    ++ retract(e) takes e and transforms it into a
    ++ FortranExpression checking that it contains no non-Fortran
    ++ functions, and that it only contains the given basic symbols
    ++ and subscripted symbols which correspond to scalar and array
    ++ parameters respectively.
  retractIfCan : EXPR Integer -> Union($,"failed")
    ++ retractIfCan(e) takes e and tries to transform it into a
    ++ FortranExpression checking that it contains no non-Fortran
    ++ functions, and that it only contains the given basic symbols
    ++ and subscripted symbols which correspond to scalar and array
    ++ parameters respectively.
  retract : FRAC POLY Integer -> $
    ++ retract(e) takes e and transforms it into a
    ++ FortranExpression checking that it contains no non-Fortran
    ++ functions, and that it only contains the given basic symbols
    ++ and subscripted symbols which correspond to scalar and array
    ++ parameters respectively.
  retractIfCan : FRAC POLY Integer -> Union($,"failed")
    ++ retractIfCan(e) takes e and tries to transform it into a
    ++ FortranExpression checking that it contains no non-Fortran
    ++ functions, and that it only contains the given basic symbols
    ++ and subscripted symbols which correspond to scalar and array
    ++ parameters respectively.
  retract : POLY Integer -> $
    ++ retract(e) takes e and transforms it into a
    ++ FortranExpression checking that it contains no non-Fortran
    ++ functions, and that it only contains the given basic symbols
    ++ and subscripted symbols which correspond to scalar and array
    ++ parameters respectively.
  retractIfCan : POLY Integer -> Union($,"failed")

```

```

    ++ retractIfCan(e) takes e and tries to transform it into a
    ++ FortranExpression checking that it contains no non-Fortran
    ++ functions, and that it only contains the given basic symbols
    ++ and subscripted symbols which correspond to scalar and array
    ++ parameters respectively.
if (R has RetractableTo(Float)) then
  retract : EXPR Float -> $
    ++ retract(e) takes e and transforms it into a
    ++ FortranExpression checking that it contains no non-Fortran
    ++ functions, and that it only contains the given basic symbols
    ++ and subscripted symbols which correspond to scalar and array
    ++ parameters respectively.
  retractIfCan : EXPR Float -> Union($,"failed")
    ++ retractIfCan(e) takes e and tries to transform it into a
    ++ FortranExpression checking that it contains no non-Fortran
    ++ functions, and that it only contains the given basic symbols
    ++ and subscripted symbols which correspond to scalar and array
    ++ parameters respectively.
  retract : FRAC POLY Float -> $
    ++ retract(e) takes e and transforms it into a
    ++ FortranExpression checking that it contains no non-Fortran
    ++ functions, and that it only contains the given basic symbols
    ++ and subscripted symbols which correspond to scalar and array
    ++ parameters respectively.
  retractIfCan : FRAC POLY Float -> Union($,"failed")
    ++ retractIfCan(e) takes e and tries to transform it into a
    ++ FortranExpression checking that it contains no non-Fortran
    ++ functions, and that it only contains the given basic symbols
    ++ and subscripted symbols which correspond to scalar and array
    ++ parameters respectively.
  retract : POLY Float -> $
    ++ retract(e) takes e and transforms it into a
    ++ FortranExpression checking that it contains no non-Fortran
    ++ functions, and that it only contains the given basic symbols
    ++ and subscripted symbols which correspond to scalar and array
    ++ parameters respectively.
  retractIfCan : POLY Float -> Union($,"failed")
    ++ retractIfCan(e) takes e and tries to transform it into a
    ++ FortranExpression checking that it contains no non-Fortran
    ++ functions, and that it only contains the given basic symbols
    ++ and subscripted symbols which correspond to scalar and array
    ++ parameters respectively.
abs    : $ -> $
    ++ abs(x) represents the Fortran intrinsic function ABS
sqrt   : $ -> $
    ++ sqrt(x) represents the Fortran intrinsic function SQRT
exp    : $ -> $
    ++ exp(x) represents the Fortran intrinsic function EXP
log    : $ -> $
    ++ log(x) represents the Fortran intrinsic function LOG

```



```

log10 : $ -> $
  ++ log10(x) represents the Fortran intrinsic function LOG10
sin    : $ -> $
  ++ sin(x) represents the Fortran intrinsic function SIN
cos    : $ -> $
  ++ cos(x) represents the Fortran intrinsic function COS
tan    : $ -> $
  ++ tan(x) represents the Fortran intrinsic function TAN
asin   : $ -> $
  ++ asin(x) represents the Fortran intrinsic function ASIN
acos   : $ -> $
  ++ acos(x) represents the Fortran intrinsic function ACOS
atan   : $ -> $
  ++ atan(x) represents the Fortran intrinsic function ATAN
sinh   : $ -> $
  ++ sinh(x) represents the Fortran intrinsic function SINH
cosh   : $ -> $
  ++ cosh(x) represents the Fortran intrinsic function COSH
tanh   : $ -> $
  ++ tanh(x) represents the Fortran intrinsic function TANH
pi     : () -> $
  ++ pi(x) represents the NAG Library function X01AAF which returns
  ++ an approximation to the value of pi
variables : $ -> L S
  ++ variables(e) return a list of all the variables in \spad{e}.
useNagFunctions : () -> Boolean
  ++ useNagFunctions() indicates whether NAG functions are being used
  ++ for mathematical and machine constants.
useNagFunctions : Boolean -> Boolean
  ++ useNagFunctions(v) sets the flag which controls whether NAG functions
  ++ are being used for mathematical and machine constants. The previous
  ++ value is returned.

Implementation ==> EXPR R add

-- The standard FORTRAN-77 intrinsic functions, plus nthRoot which
-- can be translated into an arithmetic expression:
f77Functions : L S := [abs,sqrt,exp,log,log10,sin,cos,tan,asin,acos,
                      atan,sinh,cosh,tanh,nthRoot,%power]
nagFunctions : L S := [pi, X01AAF]
useNagFunctionsFlag : Boolean := true

-- Local functions to check for "unassigned" symbols etc.

mkEqn(s1:Symbol,s2:Symbol):Equation EXPR(R) ==
  equation(s2::EXPR(R),script(s1,scripts(s2))::EXPR(R))

fixUpSymbols(u:EXPR R):Union(EXPR R,"failed") ==
  -- If its a univariate expression then just fix it up:
  syms    : L S := variables(u)

```

```

--      one? (#basicSymbols) and zero? (#subscriptedSymbols) =>
--      (#basicSymbols = 1) and zero? (#subscriptedSymbols) =>
--      not one? (#syms) => "failed"
--      not (#syms = 1) => "failed"
--      subst(u, equation(first(syms)::EXPR(R), first(basicSymbols)::EXPR(R)))
-- We have one variable but it is subscripted:
--      zero? (#basicSymbols) and one? (#subscriptedSymbols) =>
--      zero? (#basicSymbols) and (#subscriptedSymbols = 1) =>
--      -- Make sure we don't have both X and X_i
--      for s in syms repeat
--      not scripted?(s) => return "failed"
--      not one? (#(syms:=removeDuplicates! [name(s) for s in syms])) => "failed"
--      not ((#(syms:=removeDuplicates! [name(s) for s in syms])) = 1) => "failed"
--      sym : Symbol := first subscriptedSymbols
--      subst(u, [mkEqn(sym, i) for i in variables(u)])
--      "failed"

extraSymbols?(u:EXPR R):Boolean ==
  syms   : L S := [name(v) for v in variables(u)]
  extras : L S := setDifference(syms,
                                setUnion(basicSymbols, subscriptedSymbols))

  not empty? extras

checkSymbols(u:EXPR R):EXPR(R) ==
  syms   : L S := [name(v) for v in variables(u)]
  extras : L S := setDifference(syms,
                                setUnion(basicSymbols, subscriptedSymbols))

  not empty? extras =>
    m := fixUpSymbols(u)
    m case EXPR(R) => m::EXPR(R)
    error("Extra symbols detected:", [string(v) for v in extras]$L(String))
  u

notSymbol?(v:B0):Boolean ==
  s : S := name v
  member?(s, basicSymbols) or
  scripted?(s) and member?(name s, subscriptedSymbols) => false
  true

extraOperators?(u:EXPR R):Boolean ==
  ops   : L S := [name v for v in operators(u) | notSymbol?(v)]
  if useNagFunctionsFlag then
    fortranFunctions : L S := append(f77Functions, nagFunctions)
  else
    fortranFunctions : L S := f77Functions
  extras : L S := setDifference(ops, fortranFunctions)
  not empty? extras

checkOperators(u:EXPR R):Void ==
  ops   : L S := [name v for v in operators(u) | notSymbol?(v)]

```

```

if useNagFunctionsFlag then
  fortranFunctions : L S := append(f77Functions,nagFunctions)
else
  fortranFunctions : L S := f77Functions
extras : L S := setDifference(ops,fortranFunctions)
not empty? extras =>
  error("Non FORTRAN-77 functions detected:",[string(v) for v in extras])
void()

checkForNagOperators(u:EXPR R):$ ==
  useNagFunctionsFlag =>
    import Pi
    import PiCoercions(R)
    piOp : BasicOperator := operator X01AAF
    piSub : Equation EXPR R :=
      equation(pi()$Pi::EXPR(R),kernel(piOp,0::EXPR(R))$EXPR(R))
    subst(u,piSub) pretend $
  u pretend $

-- Conditional retractions:

if R has RetractableTo(Integer) then

  retractIfCan(u:POLY Integer):Union($,"failed") ==
    retractIfCan((u::EXPR Integer)$EXPR(Integer))@Union($,"failed")

  retract(u:POLY Integer):$ ==
    retract((u::EXPR Integer)$EXPR(Integer))@$

  retractIfCan(u:FRAC POLY Integer):Union($,"failed") ==
    retractIfCan((u::EXPR Integer)$EXPR(Integer))@Union($,"failed")

  retract(u:FRAC POLY Integer):$ ==
    retract((u::EXPR Integer)$EXPR(Integer))@$

  int2R(u:Integer):R == u::R

  retractIfCan(u:EXPR Integer):Union($,"failed") ==
    retractIfCan(map(int2R,u)$EXF2(Integer,R))@Union($,"failed")

  retract(u:EXPR Integer):$ ==
    retract(map(int2R,u)$EXF2(Integer,R))@$

if R has RetractableTo(Float) then

  retractIfCan(u:POLY Float):Union($,"failed") ==
    retractIfCan((u::EXPR Float)$EXPR(Float))@Union($,"failed")

  retract(u:POLY Float):$ ==
    retract((u::EXPR Float)$EXPR(Float))@$

```

```

retractIfCan(u:FRAC POLY Float):Union($,"failed") ==
  retractIfCan((u::EXPR Float)$EXPR(Float))@Union($,"failed")

retract(u:FRAC POLY Float):$ ==
  retract((u::EXPR Float)$EXPR(Float))@$

float2R(u:Float):R == (u::R)

retractIfCan(u:EXPR Float):Union($,"failed") ==
  retractIfCan(map(float2R,u)$EXF2(Float,R))@Union($,"failed")

retract(u:EXPR Float):$ ==
  retract(map(float2R,u)$EXF2(Float,R))@$

-- Exported Functions

useNagFunctions():Boolean == useNagFunctionsFlag
useNagFunctions(v:Boolean):Boolean ==
  old := useNagFunctionsFlag
  useNagFunctionsFlag := v
  old

log10(x:$):$ ==
  kernel(operator log10,x)

pi():$ == kernel(operator X01AAF,0)

coerce(u:$):EXPR R == u pretend EXPR(R)

retractIfCan(u:EXPR R):Union($,"failed") ==
  if (extraSymbols? u) then
    m := fixUpSymbols(u)
    m case "failed" => return "failed"
    u := m::EXPR(R)
  extraOperators? u => "failed"
  checkForNagOperators(u)

retract(u:EXPR R):$ ==
  u:=checkSymbols(u)
  checkOperators(u)
  checkForNagOperators(u)

retractIfCan(u:Symbol):Union($,"failed") ==
  not (member?(u,basicSymbols) or
    scripted?(u) and member?(name u,subscriptedSymbols)) => "failed"
  ((u::EXPR(R))$(EXPR R))pretend Rep>::$

retract(u:Symbol):$ ==
  res : Union($,"failed") := retractIfCan(u)

```

```
res case "failed" => error("Illegal Symbol Detected:",u::String)
res::$
```

— FEXPR.dotabb —

```
"FEXPR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FEXPR"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"FEXPR" -> "ALIST"
```

domain FORTRAN FortranProgram

— FortranProgram.input —

```
)set break resume
)sys rm -f FortranProgram.output
)spool FortranProgram.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FortranProgram
--R
--R FortranProgram(name: Symbol,returnType: Union(fst: FortranScalarType,void: void),arguments:
--R Abbreviation for FortranProgram is FORTRAN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FORTRAN
--R
--R----- Operations -----
--R coerce : Expression(Float) -> %      coerce : Expression(Integer) -> %
--R coerce : List(FortranCode) -> %      coerce : FortranCode -> %
--R coerce : % -> OutputForm              outputAsFortran : % -> Void
--R coerce : Equation(Expression(Complex(Float))) -> %
--R coerce : Equation(Expression(Float)) -> %
--R coerce : Equation(Expression(Integer)) -> %
--R coerce : Expression(Complex(Float)) -> %
--R coerce : Equation(Expression(MachineComplex)) -> %
--R coerce : Equation(Expression(MachineFloat)) -> %
--R coerce : Equation(Expression(MachineInteger)) -> %
--R coerce : Expression(MachineComplex) -> %
```

```

--R coerce : Expression(MachineFloat) -> %
--R coerce : Expression(MachineInteger) -> %
--R coerce : Record(localSymbols: SymbolTable,code: List(FortranCode)) -> %
--R
--E 1

)spool
)lisp (bye)

```

— FortranProgram.help —

=====

FortranProgram examples

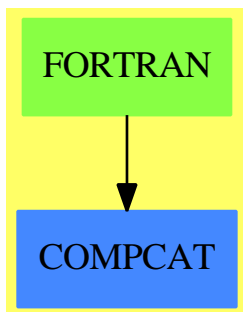
=====

FortranProgram allows the user to build and manipulate simple models of FORTRAN subprograms. These can then be transformed into actual FORTRAN notation.

See Also:

o)show FortranProgram

FortranProgram (FORTRAN)



See

- ⇒ “Result” (RESULT) [19](#) on page [2300](#)
- ⇒ “FortranCode” (FC) [7](#) on page [931](#)
- ⇒ “ThreeDimensionalMatrix” (M3D) [21](#) on page [2702](#)
- ⇒ “SimpleFortranProgram” (SFORT) [20](#) on page [2405](#)
- ⇒ “Switch” (SWITCH) [20](#) on page [2629](#)

⇒ “FortranTemplate” (FTEM) 7 on page 967

⇒ “FortranExpression” (FEXPR) 7 on page 946

Exports:

coerce outputAsFortran

— domain FORTRAN FortranProgram —

```

)abbrev domain FORTRAN FortranProgram
++ Author: Mike Dewar
++ Date Created: October 1992
++ Date Last Updated: 23 January 1995 Added support for intrinsic functions
++ Description:
++ \axiomType{FortranProgram} allows the user to build and manipulate simple
++ models of FORTRAN subprograms. These can then be transformed into
++ actual FORTRAN notation.

FortranProgram(name,returnType,arguments,symbols): Exports == Implement where
  name      : Symbol
  returnType : Union(fst:FortranScalarType,void:"void")
  arguments  : List Symbol
  symbols    : SymbolTable

FC      ==> FortranCode
EXPR    ==> Expression
INT      ==> Integer
CMPX    ==> Complex
MINT     ==> MachineInteger
MFLOAT  ==> MachineFloat
MCMPX   ==> MachineComplex
REP      ==> Record(localSymbols : SymbolTable, code : List FortranCode)

Exports ==> FortranProgramCategory with
  coerce : FortranCode -> $
    ++ coerce(fc) is not documented
  coerce : List FortranCode -> $
    ++ coerce(lfc) is not documented
  coerce : REP -> $
    ++ coerce(r) is not documented
  coerce : EXPR MINT -> $
    ++ coerce(e) is not documented
  coerce : EXPR MFLOAT -> $
    ++ coerce(e) is not documented
  coerce : EXPR MCMPX -> $
    ++ coerce(e) is not documented
  coerce : Equation EXPR MINT -> $
    ++ coerce(eq) is not documented
  coerce : Equation EXPR MFLOAT -> $
    ++ coerce(eq) is not documented
  coerce : Equation EXPR MCMPX -> $

```

```

    ++ coerce(eq) is not documented
coerce : EXPR INT -> $
    ++ coerce(e) is not documented
coerce : EXPR Float -> $
    ++ coerce(e) is not documented
coerce : EXPR CMPX Float -> $
    ++ coerce(e) is not documented
coerce : Equation EXPR INT -> $
    ++ coerce(eq) is not documented
coerce : Equation EXPR Float -> $
    ++ coerce(eq) is not documented
coerce : Equation EXPR CMPX Float -> $
    ++ coerce(eq) is not documented

Implement ==> add

Rep := REP

import SExpression
import TheSymbolTable
import FortranCode

makeRep(b:List FortranCode):$ ==
    construct(empty())$SymbolTable,b)$REP

codeFrom(u:$):List FortranCode ==
    elt(u::Rep,code)$REP

outputAsFortran(p:$):Void ==
    setLabelValue(25000::SingleInteger)$FC
    -- Do this first to catch any extra type declarations:
    tempName := "FPTMP"::Symbol
    newSubProgram(tempName)
    initialiseIntrinsicList()$Lisp
    body : List SExpression := [getCode(l)$FortranCode for l in codeFrom(p)]
    intrinsics : SExpression := getIntrinsicList()$Lisp
    endSubProgram()
    fortFormatHead(returnType::OutputForm, name::OutputForm, _
        arguments::OutputForm)$Lisp
    printTypes(symbols)$SymbolTable
    printTypes((p::Rep).localSymbols)$SymbolTable
    printTypes(tempName)$TheSymbolTable
    fortFormatIntrinsics(intrinsics)$Lisp
    clearTheSymbolTable(tempName)
    for expr in body repeat displayLines1(expr)$Lisp
    dispStatement(END::OutputForm)$Lisp
    void()$Void

mkString(l:List Symbol):String ==
    unparse(convert(l::OutputForm)$InputForm)$InputForm

```



```

checkVariables(user:List Symbol,target:List Symbol):Void ==
  -- We don't worry about whether the user has subscripted the
  -- variables or not.
  setDifference(map(name$Symbol,user),target) ^~= empty()$List(Symbol) =>
    s1 : String := mkString(user)
    s2 : String := mkString(target)
    error ["Incompatible variable lists:", s1, s2]
  void()$Void

coerce(u:EXPR MINT) : $ ==
  checkVariables(variables(u)$EXPR(MINT),arguments)
  l : List(FC) := [assign(name,u)$FC,returns()$FC]
  makeRep l

coerce(u:Equation EXPR MINT) : $ ==
  retractIfCan(lhs u)@Union(Kernel(EXPR MINT),"failed") case "failed" =>
    error "left hand side is not a kernel"
  vList : List Symbol := variables lhs u
  #vList ^~= #arguments =>
    error "Incorrect number of arguments"
  veList : List EXPR MINT := [w::EXPR(MINT) for w in vList]
  aeList : List EXPR MINT := [w::EXPR(MINT) for w in arguments]
  eList : List Equation EXPR MINT :=
    [equation(w,v) for w in veList for v in aeList]
  (subst(rhs u,eList)):$

coerce(u:EXPR MFLOAT) : $ ==
  checkVariables(variables(u)$EXPR(MFLOAT),arguments)
  l : List(FC) := [assign(name,u)$FC,returns()$FC]
  makeRep l

coerce(u:Equation EXPR MFLOAT) : $ ==
  retractIfCan(lhs u)@Union(Kernel(EXPR MFLOAT),"failed") case "failed" =>
    error "left hand side is not a kernel"
  vList : List Symbol := variables lhs u
  #vList ^~= #arguments =>
    error "Incorrect number of arguments"
  veList : List EXPR MFLOAT := [w::EXPR(MFLOAT) for w in vList]
  aeList : List EXPR MFLOAT := [w::EXPR(MFLOAT) for w in arguments]
  eList : List Equation EXPR MFLOAT :=
    [equation(w,v) for w in veList for v in aeList]
  (subst(rhs u,eList)):$

coerce(u:EXPR MCMLX) : $ ==
  checkVariables(variables(u)$EXPR(MCMLX),arguments)
  l : List(FC) := [assign(name,u)$FC,returns()$FC]
  makeRep l

coerce(u:Equation EXPR MCMLX) : $ ==

```

```

    retractIfCan(lhs u)@Union(Kernel EXPR MCMPLEX,"failed") case "failed"=>
        error "left hand side is not a kernel"
    vList : List Symbol := variables lhs u
    #vList ^= #arguments =>
        error "Incorrect number of arguments"
    veList : List EXPR MCMPLEX := [w::EXPR(MCMPLEX) for w in vList]
    aeList : List EXPR MCMPLEX := [w::EXPR(MCMPLEX) for w in arguments]
    eList : List Equation EXPR MCMPLEX :=
        [equation(w,v) for w in veList for v in aeList]
    (subst(rhs u,eList))::$

coerce(u:REP):$ ==
    u@Rep

coerce(u:$):OutputForm ==
    coerce(name)$Symbol

coerce(c:List FortranCode):$ ==
    makeRep c

coerce(c:FortranCode):$ ==
    makeRep [c]

coerce(u:EXPR INT) : $ ==
    checkVariables(variables(u)$EXPR(INT),arguments)
    l : List(FC) := [assign(name,u)$FC,returns()$FC]
    makeRep l

coerce(u:Equation EXPR INT) : $ ==
    retractIfCan(lhs u)@Union(Kernel(EXPR INT),"failed") case "failed" =>
        error "left hand side is not a kernel"
    vList : List Symbol := variables lhs u
    #vList ^= #arguments =>
        error "Incorrect number of arguments"
    veList : List EXPR INT := [w::EXPR(INT) for w in vList]
    aeList : List EXPR INT := [w::EXPR(INT) for w in arguments]
    eList : List Equation EXPR INT :=
        [equation(w,v) for w in veList for v in aeList]
    (subst(rhs u,eList))::$

coerce(u:EXPR Float) : $ ==
    checkVariables(variables(u)$EXPR(Float),arguments)
    l : List(FC) := [assign(name,u)$FC,returns()$FC]
    makeRep l

coerce(u:Equation EXPR Float) : $ ==
    retractIfCan(lhs u)@Union(Kernel(EXPR Float),"failed") case "failed" =>
        error "left hand side is not a kernel"
    vList : List Symbol := variables lhs u

```

```

#vList ^= #arguments =>
  error "Incorrect number of arguments"
veList : List EXPR Float := [w::EXPR(Float) for w in vList]
aeList : List EXPR Float := [w::EXPR(Float) for w in arguments]
eList : List Equation EXPR Float :=
  [equation(w,v) for w in veList for v in aeList]
(subst(rhs u,eList))::$

coerce(u:EXPR Complex Float) : $ ==
  checkVariables(variables(u)$EXPR(Complex Float),arguments)
  l : List(FC) := [assign(name,u)$FC,returns()$FC]
  makeRep l

coerce(u:Equation EXPR CMPX Float) : $ ==
  retractIfCan(lhs u)@Union(Kernel EXPR CMPX Float,"failed") case "failed"=>
    error "left hand side is not a kernel"
  vList : List Symbol := variables lhs u
  #vList ^= #arguments =>
    error "Incorrect number of arguments"
  veList : List EXPR CMPX Float := [w::EXPR(CMPX Float) for w in vList]
  aeList : List EXPR CMPX Float := [w::EXPR(CMPX Float) for w in arguments]
  eList : List Equation EXPR CMPX Float :=
    [equation(w,v) for w in veList for v in aeList]
  (subst(rhs u,eList))::$

```

— FORTRAN.dotabb —

```

"FORTRAN" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FORTRAN"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"FORTRAN" -> "COMPCAT"

```

domain FST FortranScalarType

— FortranScalarType.input —

```

)set break resume
)sys rm -f FortranScalarType.output
)spool FortranScalarType.output
)set message test on
)set message auto off

```

```

)clear all

--S 1 of 1
)show FortranScalarType
--R FortranScalarType is a domain constructor
--R Abbreviation for FortranScalarType is FST
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FST
--R
--R----- Operations -----
--R ?? : (% ,%) -> Boolean          character? : % -> Boolean
--R coerce : % -> SExpression      coerce : % -> Symbol
--R coerce : Symbol -> %          coerce : String -> %
--R coerce : % -> OutputForm      complex? : % -> Boolean
--R double? : % -> Boolean        doubleComplex? : % -> Boolean
--R integer? : % -> Boolean      logical? : % -> Boolean
--R real? : % -> Boolean
--R
--E 1

)spool
)lisp (bye)

```

— FortranScalarType.help —

=====

FortranScalarType examples

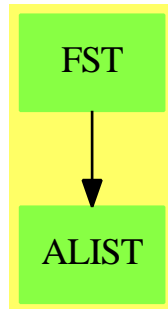
=====

Creates and manipulates objects which correspond to the
basic FORTRAN data types: REAL, INTEGER, COMPLEX, LOGICAL and CHARACTER

See Also:

o)show FortranScalarType

FortranScalarType (FST)



See

- ⇒ “FortranType” (FT) [7](#) on page [971](#)
- ⇒ “SymbolTable” (SYMTAB) [20](#) on page [2648](#)
- ⇒ “TheSymbolTable” (SYMS) [21](#) on page [2696](#)

Exports:

character? coerce complex? double? doubleComplex? integer? logical? real? ==?

— domain FST FortranScalarType —

```

)abbrev domain FST FortranScalarType
++ Author: Mike Dewar
++ Date Created: October 1992
++ Description:
++ Creates and manipulates objects which correspond to the
++ basic FORTRAN data types: REAL, INTEGER, COMPLEX, LOGICAL and CHARACTER

FortranScalarType() : exports == implementation where

exports == CoercibleTo OutputForm with
  coerce : String -> $
    ++ coerce(s) transforms the string s into an element of
    ++ FortranScalarType provided s is one of "real", "double precision",
    ++ "complex", "logical", "integer", "character", "REAL",
    ++ "COMPLEX", "LOGICAL", "INTEGER", "CHARACTER",
    ++ "DOUBLE PRECISION"
  coerce : Symbol -> $
    ++ coerce(s) transforms the symbol s into an element of
    ++ FortranScalarType provided s is one of real, complex,double precision,
    ++ logical, integer, character, REAL, COMPLEX, LOGICAL,
    ++ INTEGER, CHARACTER, DOUBLE PRECISION
  coerce : $ -> Symbol
    ++ coerce(x) returns the symbol associated with x
  coerce : $ -> SExpression
    ++ coerce(x) returns the s-expression associated with x
  
```

```

real? : $ -> Boolean
  ++ real?(t) tests whether t is equivalent to the FORTRAN type REAL.
double? : $ -> Boolean
  ++ double?(t) tests whether t is equivalent to the FORTRAN type
  ++ DOUBLE PRECISION
integer? : $ -> Boolean
  ++ integer?(t) tests whether t is equivalent to the FORTRAN type INTEGER.
complex? : $ -> Boolean
  ++ complex?(t) tests whether t is equivalent to the FORTRAN type COMPLEX.
doubleComplex? : $ -> Boolean
  ++ doubleComplex?(t) tests whether t is equivalent to the (non-standard)
  ++ FORTRAN type DOUBLE COMPLEX.
character? : $ -> Boolean
  ++ character?(t) tests whether t is equivalent to the FORTRAN type
  ++ CHARACTER.
logical? : $ -> Boolean
  ++ logical?(t) tests whether t is equivalent to the FORTRAN type LOGICAL.
"=" : ($,$) -> Boolean
  ++ x=y tests for equality

implementation == add

U == Union(RealThing:"real",
           IntegerThing:"integer",
           ComplexThing:"complex",
           CharacterThing:"character",
           LogicalThing:"logical",
           DoublePrecisionThing:"double precision",
           DoubleComplexThing:"double complex")
Rep := U

doubleSymbol : Symbol := "double precision"::Symbol
upperDoubleSymbol : Symbol := "DOUBLE PRECISION"::Symbol
doubleComplexSymbol : Symbol := "double complex"::Symbol
upperDoubleComplexSymbol : Symbol := "DOUBLE COMPLEX"::Symbol

u = v ==
  u case RealThing and v case RealThing => true
  u case IntegerThing and v case IntegerThing => true
  u case ComplexThing and v case ComplexThing => true
  u case LogicalThing and v case LogicalThing => true
  u case CharacterThing and v case CharacterThing => true
  u case DoublePrecisionThing and v case DoublePrecisionThing => true
  u case DoubleComplexThing and v case DoubleComplexThing => true
  false

coerce(t:$):OutputForm ==
  t case RealThing => coerce(REAL)$Symbol
  t case IntegerThing => coerce(INTEGER)$Symbol
  t case ComplexThing => coerce(COMPLEX)$Symbol

```

```

t case CharacterThing => coerce(CHARACTER)$Symbol
t case DoublePrecisionThing => coerce(upperDoubleSymbol)$Symbol
t case DoubleComplexThing => coerce(upperDoubleComplexSymbol)$Symbol
coerce(LOGICAL)$Symbol

coerce(t:$):SEExpression ==
  t case RealThing => convert(real::Symbol)@SEExpression
  t case IntegerThing => convert(integer::Symbol)@SEExpression
  t case ComplexThing => convert(complex::Symbol)@SEExpression
  t case CharacterThing => convert(character::Symbol)@SEExpression
  t case DoublePrecisionThing => convert(doubleSymbol)@SEExpression
  t case DoubleComplexThing => convert(doubleComplexSymbol)@SEExpression
  convert(logical::Symbol)@SEExpression

coerce(t:$):Symbol ==
  t case RealThing => real::Symbol
  t case IntegerThing => integer::Symbol
  t case ComplexThing => complex::Symbol
  t case CharacterThing => character::Symbol
  t case DoublePrecisionThing => doubleSymbol
  t case DoubleComplexThing => doubleComplexSymbol
  logical::Symbol

coerce(s:Symbol):$ ==
  s = real => ["real"]$Rep
  s = REAL => ["real"]$Rep
  s = integer => ["integer"]$Rep
  s = INTEGER => ["integer"]$Rep
  s = complex => ["complex"]$Rep
  s = COMPLEX => ["complex"]$Rep
  s = character => ["character"]$Rep
  s = CHARACTER => ["character"]$Rep
  s = logical => ["logical"]$Rep
  s = LOGICAL => ["logical"]$Rep
  s = doubleSymbol => ["double precision"]$Rep
  s = upperDoubleSymbol => ["double precision"]$Rep
  s = doubleComplexSymbol => ["double complex"]$Rep
  s = upperDoubleComplexSymbol => ["double complex"]$Rep

coerce(s:String):$ ==
  s = "real" => ["real"]$Rep
  s = "integer" => ["integer"]$Rep
  s = "complex" => ["complex"]$Rep
  s = "character" => ["character"]$Rep
  s = "logical" => ["logical"]$Rep
  s = "double precision" => ["double precision"]$Rep
  s = "double complex" => ["double complex"]$Rep
  s = "REAL" => ["real"]$Rep
  s = "INTEGER" => ["integer"]$Rep
  s = "COMPLEX" => ["complex"]$Rep

```

```

s = "CHARACTER" => ["character"]$Rep
s = "LOGICAL" => ["logical"]$Rep
s = "DOUBLE PRECISION" => ["double precision"]$Rep
s = "DOUBLE COMPLEX" => ["double complex"]$Rep
error concat([s," is invalid as a Fortran Type"])$String

real?(t:$):Boolean == t case RealThing

double?(t:$):Boolean == t case DoublePrecisionThing

logical?(t:$):Boolean == t case LogicalThing

integer?(t:$):Boolean == t case IntegerThing

character?(t:$):Boolean == t case CharacterThing

complex?(t:$):Boolean == t case ComplexThing

doubleComplex?(t:$):Boolean == t case DoubleComplexThing

```

— FST.dotabb —

```

"FST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FST"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"FST" -> "ALIST"

```

domain FTEM FortranTemplate

— FortranTemplate.input —

```

)set break resume
)sys rm -f FortranTemplate.output
)spool FortranTemplate.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FortranTemplate
--R FortranTemplate is a domain constructor

```



```

--R Abbreviation for FortranTemplate is FTEM
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FTEM
--R
--R----- Operations -----
--R ?=? : (%,%) -> Boolean          close! : % -> %
--R coerce : % -> OutputForm        flush : % -> Void
--R fortranCarriageReturn : () -> Void  fortranLiteral : String -> Void
--R hash : % -> SingleInteger        iomode : % -> String
--R latex : % -> String              name : % -> FileName
--R open : (FileName,String) -> %     open : FileName -> %
--R read! : % -> String              reopen! : (%,String) -> %
--R write! : (%,String) -> String     ?~=? : (%,%) -> Boolean
--R fortranLiteralline : String -> Void
--R processTemplate : FileName -> FileName
--R processTemplate : (FileName,FileName) -> FileName
--R
--E 1

)spool
)lisp (bye)

```

— FortranTemplate.help —

```

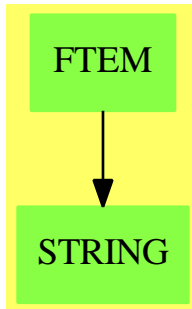
=====
FortranTemplate examples
=====

```

Code to manipulate Fortran templates

See Also:

o)show FortranTemplate

FortranTemplate (FTEM)**See**

- ⇒ “Result” (RESULT) [19](#) on page [2300](#)
- ⇒ “FortranCode” (FC) [7](#) on page [931](#)
- ⇒ “FortranProgram” (FORTRAN) [7](#) on page [955](#)
- ⇒ “ThreeDimensionalMatrix” (M3D) [21](#) on page [2702](#)
- ⇒ “SimpleFortranProgram” (SFORT) [20](#) on page [2405](#)
- ⇒ “Switch” (SWITCH) [20](#) on page [2629](#)
- ⇒ “FortranExpression” (FEXPR) [7](#) on page [946](#)

Exports:

close!	coerce	fortranCarriageReturn	fortranLiteral	fortranLiteralLine
hash	iomode	latex	name	open
processTemplate	read!	reopen!	write!	==?
?~=?				

— domain FTEM FortranTemplate —

```

)abbrev domain FTEM FortranTemplate
++ Author: Mike Dewar
++ Date Created:  October 1992
++ Description:
++ Code to manipulate Fortran templates

```

```

FortranTemplate() : specification == implementation where

```

```

specification == FileCategory(FileName, String) with

processTemplate : (FileName, FileName) -> FileName
  ++ processTemplate(tp,fn) processes the template tp, writing the
  ++ result out to fn.
processTemplate : (FileName) -> FileName
  ++ processTemplate(tp) processes the template tp, writing the
  ++ result to the current FORTRAN output stream.
fortranLiteralLine : String -> Void
  ++ fortranLiteralLine(s) writes s to the current Fortran output stream,

```

```

    ++ followed by a carriage return
fortranLiteral : String -> Void
    ++ fortranLiteral(s) writes s to the current Fortran output stream
fortranCarriageReturn : () -> Void
    ++ fortranCarriageReturn() produces a carriage return on the current
    ++ Fortran output stream

implementation == TextFile add

import TemplateUtilities
import FortranOutputStackPackage

Rep := TextFile

fortranLiteralLine(s:String):Void ==
    PRINTEXP(s, _$fortranOutputStream$Lisp)$Lisp
    TERPRI(_$fortranOutputStream$Lisp)$Lisp

fortranLiteral(s:String):Void ==
    PRINTEXP(s, _$fortranOutputStream$Lisp)$Lisp

fortranCarriageReturn():Void ==
    TERPRI(_$fortranOutputStream$Lisp)$Lisp

writePassiveLine!(line:String):Void ==
-- We might want to be a bit clever here and look for new SubPrograms etc.
    fortranLiteralLine line

processTemplate(tp:FileName, fn:FileName):FileName ==
    pushFortranOutputStack(fn)
    processTemplate(tp)
    popFortranOutputStack()
    fn

getLine(fp:TextFile):String ==
    line : String := stripCommentsAndBlanks readLine!(fp)
    while not empty?(line) and elt(line,maxIndex line) = char "__" repeat
        setelt(line,maxIndex line,char " ")
        line := concat(line, stripCommentsAndBlanks readLine!(fp))$String
    line

processTemplate(tp:FileName):FileName ==
    fp : TextFile := open(tp,"input")
    active : Boolean := true
    line : String
    endInput : Boolean := false
    while not (endInput or endOfFile? fp) repeat
        if active then
            line := getLine fp
            line = "endInput" => endInput := true

```

```

    if line = "beginVerbatim" then
        active := false
    else
        not empty? line => interpretString line
    else
        line := readLine!(fp)
        if line = "endVerbatim" then
            active := true
        else
            writePassiveLine! line
    close!(fp)
    if not active then
        error concat(["Missing 'endVerbatim' line in ", tp::String])$String
    string(_$fortranOutputFile$Lisp)::FileName

```

— FTEM.dotabb —

```

"FTEM" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FTEM"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"FTEM" -> "STRING"

```

domain FT FortranType

— FortranType.input —

```

)set break resume
)sys rm -f FortranType.output
)spool FortranType.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FortranType
--R
--R FortranType is a domain constructor
--R Abbreviation for FortranType is FT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FT
--R

```

```

--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : FortranScalarType -> %
--R coerce : % -> OutputForm          external? : % -> Boolean
--R fortranCharacter : () -> %        fortranComplex : () -> %
--R fortranDouble : () -> %           fortranDoubleComplex : () -> %
--R fortranInteger : () -> %          fortranLogical : () -> %
--R fortranReal : () -> %             hash : % -> SingleInteger
--R latex : % -> String               ?~=? : (%,% ) -> Boolean
--R construct : (Union(fst: FortranScalarType,void: void),List(Polynomial(Integer)),Boolean) -> %
--R construct : (Union(fst: FortranScalarType,void: void),List(Symbol),Boolean) -> %
--R dimensionsOf : % -> List(Polynomial(Integer))
--R scalarTypeOf : % -> Union(fst: FortranScalarType,void: void)
--R
--E 1

)spool
)lisp (bye)

```

— FortranType.help —

=====

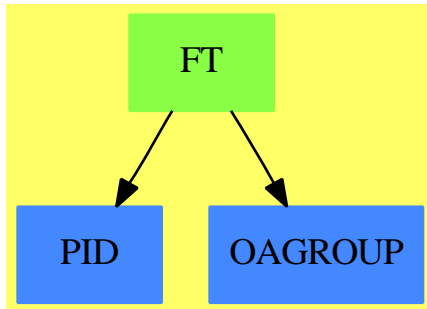
FortranType examples

=====

Creates and manipulates objects which correspond to FORTRAN data types,
including array dimensions.

See Also:

o)show FortranType

FortranType (FT)**See**

- ⇒ “FortranScalarType” (FST) [7](#) on page [962](#)
- ⇒ “SymbolTable” (SYMTAB) [20](#) on page [2648](#)
- ⇒ “TheSymbolTable” (SYMS) [21](#) on page [2696](#)

Exports:

coerce	construct	dimensionsOf	external?
fortranCharacter	fortranComplex	fortranDouble	fortranDoubleComplex
fortranInteger	fortranLogical	fortranReal	hash
latex	scalarTypeOf	?=?	?~=?

— domain FT FortranType —

```

)abbrev domain FT FortranType
++ Author: Mike Dewar
++ Date Created:  October 1992
++ Description:
++ Creates and manipulates objects which correspond to FORTRAN
++ data types, including array dimensions.

```

```

FortranType() : exports == implementation where

```

```

FST    ==> FortranScalarType
FSTU   ==> Union(fst:FST,void:"void")

```

```

exports == SetCategory with
  coerce : $ -> OutputForm
    ++ coerce(x) provides a printable form for x
  coerce : FST -> $
    ++ coerce(t) creates an element from a scalar type
  scalarTypeOf : $ -> FSTU
    ++ scalarTypeOf(t) returns the FORTRAN data type of t
  dimensionsOf : $ -> List Polynomial Integer
    ++ dimensionsOf(t) returns the dimensions of t
  external? : $ -> Boolean
    ++ external?(u) returns true if u is declared to be EXTERNAL

```

```

construct : (FSTU,List Symbol,Boolean) -> $
  ++ construct(type,dims) creates an element of FortranType
construct : (FSTU,List Polynomial Integer,Boolean) -> $
  ++ construct(type,dims) creates an element of FortranType
fortranReal : () -> $
  ++ fortranReal() returns REAL, an element of FortranType
fortranDouble : () -> $
  ++ fortranDouble() returns DOUBLE PRECISION, an element of FortranType
fortranInteger : () -> $
  ++ fortranInteger() returns INTEGER, an element of FortranType
fortranLogical : () -> $
  ++ fortranLogical() returns LOGICAL, an element of FortranType
fortranComplex : () -> $
  ++ fortranComplex() returns COMPLEX, an element of FortranType
fortranDoubleComplex : () -> $
  ++ fortranDoubleComplex() returns DOUBLE COMPLEX, an element of
  ++ FortranType
fortranCharacter : () -> $
  ++ fortranCharacter() returns CHARACTER, an element of FortranType

implementation == add

Dims == List Polynomial Integer
Rep := Record(type : FSTU, dimensions : Dims, external : Boolean)

coerce(a:$):OutputForm ==
  t : OutputForm
  if external?(a) then
    if scalarTypeOf(a) case void then
      t := "EXTERNAL":OutputForm
    else
      t := blankSeparate(["EXTERNAL":OutputForm,
                          coerce(scalarTypeOf a)$FSTU])$OutputForm
  else
    t := coerce(scalarTypeOf a)$FSTU
  empty? dimensionsOf(a) => t
  sub(t,
    paren([u:OutputForm for u in dimensionsOf(a)])$OutputForm)$OutputForm

scalarTypeOf(u:$):FSTU ==
  u.type

dimensionsOf(u:$):Dims ==
  u.dimensions

external?(u:$):Boolean ==
  u.external

construct(t:FSTU, d:List Symbol, e:Boolean):$ ==
  e and not empty? d => error "EXTERNAL objects cannot have dimensions"

```

```

    not(e) and t case void => error "VOID objects must be EXTERNAL"
    construct(t,[1::Polynomial(Integer) for l in d],e)$Rep

construct(t:FSTU, d:List Polynomial Integer, e:Boolean):$ ==
  e and not empty? d => error "EXTERNAL objects cannot have dimensions"
  not(e) and t case void => error "VOID objects must be EXTERNAL"
  construct(t,d,e)$Rep

coerce(u:FST):$ ==
  construct([u]$FSTU,[]@List Polynomial Integer,false)

fortranReal():$ == ("real":FST):$

fortranDouble():$ == ("double precision":FST):$

fortranInteger():$ == ("integer":FST):$

fortranComplex():$ == ("complex":FST):$

fortranDoubleComplex():$ == ("double complex":FST):$

fortranCharacter():$ == ("character":FST):$

fortranLogical():$ == ("logical":FST):$

```

— FT.dotabb —

```

"FT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FT"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"FT" -> "PID"
"FT" -> "OAGROUP"

```

domain FCOMP FourierComponent

— FourierComponent.input —

```

)set break resume
)sys rm -f FourierComponent.output
)spool FourierComponent.output

```



```

)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FourierComponent
--R
--R FourierComponent(E: OrderedSet) is a domain constructor
--R Abbreviation for FourierComponent is FCOMP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FCOMP
--R
--R----- Operations -----
--R ?<? : (%,% ) -> Boolean          ?<=? : (%,% ) -> Boolean
--R ?=? : (%,% ) -> Boolean          ?>? : (%,% ) -> Boolean
--R ?>=? : (%,% ) -> Boolean          argument : % -> E
--R coerce : % -> OutputForm          cos : E -> %
--R hash : % -> SingleInteger          latex : % -> String
--R max : (%,% ) -> %                  min : (%,% ) -> %
--R sin : E -> %                      sin? : % -> Boolean
--R ?~=? : (%,% ) -> Boolean
--R
--E 1

)spool
)lisp (bye)

```

— FourierComponent.help —

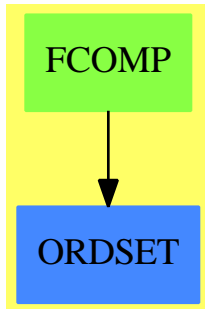
```

=====
FourierComponent examples
=====

```

This domain creates kernels for use in Fourier series

See Also:
o)show FourierComponent

FourierComponent (FCOMP)

See

⇒ “FourierSeries” (FSERIES) [7](#) on page [978](#)

Exports:

argument	coerce	cos	hash	latex
max	min	sin	sin?	?~=?
?<?	?<=?	?=?	?>?	?>=?

— domain **FCOMP** FourierComponent —

```

)abbrev domain FCOMP FourierComponent
++ Author: James Davenport
++ Date Created: 17 April 1992
++ Date Last Updated: 12 June 1992
++ Description:
++ This domain creates kernels for use in Fourier series
  
```

```

FourierComponent(E:OrderedSet):
  OrderedSet with
    sin: E -> $
    ++ sin(x) makes a sin kernel for use in Fourier series
    cos: E -> $
    ++ cos(x) makes a cos kernel for use in Fourier series
    sin?: $ -> Boolean
    ++ sin?(x) returns true if term is a sin, otherwise false
    argument: $ -> E
    ++ argument(x) returns the argument of a given sin/cos expressions
  ==
add
--representations
Rep:=Record(SinIfTrue:Boolean, arg:E)
e:E
x,y:$
sin e == [true,e]
cos e == [false,e]
sin? x == x.SinIfTrue
  
```

```

argument x == x.arg
coerce(x):OutputForm ==
  hconcat((if x.SinIfTrue then "sin" else "cos")::OutputForm,
    bracket((x.arg)::OutputForm))
x<y ==
  x.arg < y.arg => true
  y.arg < x.arg => false
  x.SinIfTrue => false
  y.SinIfTrue

```

— FCOMP.dotabb —

```

"FCOMP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FCOMP"]
"ORDSET" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ORDSET"]
"FCOMP" -> "ORDSET"

```

domain FSERIES FourierSeries

— FourierSeries.input —

```

)set break resume
)sys rm -f FourierSeries.output
)spool FourierSeries.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FourierSeries
--R
--R FourierSeries(R: Join(CommutativeRing,Algebra(Fraction(Integer))),E: Join(OrderedSet,Abelian
--R Abbreviation for FourierSeries is FSERIES
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FSERIES
--R
--R----- Operations -----
--R ?? : (R,%) -> %          ?? : (%,R) -> %
--R ?? : (%,%) -> %          ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %      ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %          ?-? : (%,%) -> %

```

```

--R -? : % -> %
--R 1 : () -> %
--R ?? : (% , PositiveInteger) -> %
--R coerce : R -> %
--R coerce : % -> OutputForm
--R latex : % -> String
--R makeSin : (E,R) -> %
--R recip : % -> Union(%,"failed")
--R zero? : % -> Boolean
--R ?? : (NonNegativeInteger,% ) -> %
--R ***? : (% , NonNegativeInteger) -> %
--R ?? : (% , NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R subtractIfCan : (% ,%) -> Union(%,"failed")
--R
--E 1

```

```

)spool
)lisp (bye)

```

— FourierSeries.help —

=====

FourierSeries examples

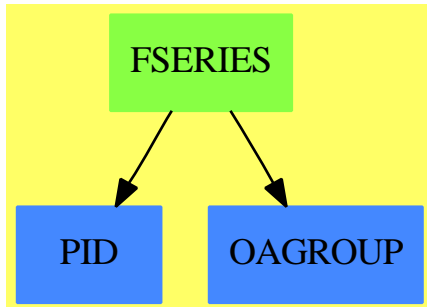
=====

This domain converts terms into Fourier series

See Also:

o)show FourierSeries

FourierSeries (FSERIES)



See

⇒ “FourierComponent” (FCOMP) 7 on page 975

Exports:

0	1	characteristic	coerce	hash
latex	makeCos	makeSin	one?	recip
sample	subtractIfCan	zero?	?~=?	?*?
?**?	?^?	?+?	?-?	-?
?=?				

— domain **FSERIES** *FourierSeries* —

```

)abbrev domain FSERIES FourierSeries
++ Author: James Davenport
++ Date Created: 17 April 1992
++ Description:
++ This domain converts terms into Fourier series

FourierSeries(R:Join(CommutativeRing,Algebra(Fraction Integer)),
  E:Join(OrderedSet,AbelianGroup)):
  Algebra(R) with
    if E has canonical and R has canonical then canonical
    coerce: R -> $
      ++ coerce(r) converts coefficients into Fourier Series
    coerce: FourierComponent(E) -> $
      ++ coerce(c) converts sin/cos terms into Fourier Series
    makeSin: (E,R) -> $
      ++ makeSin(e,r) makes a sin expression with given
      ++ argument and coefficient
    makeCos: (E,R) -> $
      ++ makeCos(e,r) makes a sin expression with given
      ++ argument and coefficient
    == FreeModule(R,FourierComponent(E))
  add
    --representations
    Term := Record(k:FourierComponent(E),c:R)

```

```

Rep  := List Term
multiply : (Term,Term) -> $
w,x1,x2:$
t1,t2:Term
n:NonNegativeInteger
z:Integer
e:FourierComponent(E)
a:E
r:R
1 == [[cos 0,1]]
coerce e ==
  sin? e and zero? argument e => 0
  if argument e < 0 then
    not sin? e => e:=cos(- argument e)
    return [[sin(- argument e),-1]]
  [[e,1]]
multiply(t1,t2) ==
  r:=(t1.c*t2.c)*(1/2)
  s1:=argument t1.k
  s2:=argument t2.k
  sum:=s1+s2
  diff:=s1-s2
  sin? t1.k =>
    sin? t2.k =>
      makeCos(diff,r) + makeCos(sum,-r)
      makeSin(sum,r) + makeSin(diff,r)
  sin? t2.k =>
    makeSin(sum,r) + makeSin(diff,r)
    makeCos(diff,r) + makeCos(sum,r)
x1*x2 ==
  null x1 => 0
  null x2 => 0
  +/[+/[multiply(t1,t2) for t2 in x2] for t1 in x1]
makeCos(a,r) ==
  a<0 => [[cos(-a),r]]
  [[cos a,r]]
makeSin(a,r) ==
  zero? a => []
  a<0 => [[sin(-a),-r]]
  [[sin a,r]]

```

— FSERIES.dotabb —

"FSERIES" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FSERIES"]
 "PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
 "OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]

```
"FSERIES" -> "PID"
"FSERIES" -> "OAGROUP"
```

domain FRAC Fraction

— Fraction.input —

```
)set break resume
)sys rm -f Fraction.output
)spool Fraction.output
)set message test on
)set message auto off
)clear all
--S 1 of 12
a := 11/12
--R
--R
--R      11
--R  (1)  --
--R      12
--R
--R                                          Type: Fraction(Integer)
--E 1

--S 2 of 12
b := 23/24
--R
--R
--R      23
--R  (2)  --
--R      24
--R
--R                                          Type: Fraction(Integer)
--E 2

--S 3 of 12
3 - a*b**2 + a + b/a
--R
--R
--R      313271
--R  (3)  -----
--R      76032
--R
--R                                          Type: Fraction(Integer)
--E 3

--S 4 of 12
```

```

numer(a)
--R
--R
--R (4) 11
--R
--R                                          Type: PositiveInteger
--E 4

--S 5 of 12
denom(b)
--R
--R
--R (5) 24
--R
--R                                          Type: PositiveInteger
--E 5

--S 6 of 12
r := (x**2 + 2*x + 1)/(x**2 - 2*x + 1)
--R
--R
--R          2
--R      x  + 2x + 1
--R (6)  -----
--R          2
--R      x  - 2x + 1
--R
--R                                          Type: Fraction(Polynomial(Integer))
--E 6

--S 7 of 12
factor(r)
--R
--R
--R          2
--R      x  + 2x + 1
--R (7)  -----
--R          2
--R      x  - 2x + 1
--R
--R                                          Type: Factored(Fraction(Polynomial(Integer)))
--E 7

--S 8 of 12
map(factor,r)
--R
--R
--R          2
--R      (x + 1)
--R (8)  -----
--R          2
--R      (x - 1)
--R
--R                                          Type: Fraction(Factored(Polynomial(Integer)))
--E 8

```



```

--S 9 of 12
continuedFraction(7/12)
--R
--R
--R      1 |      1 |      1 |      1 |
--R (9)  +---+ + +---+ + +---+ + +---+
--R      | 1      | 1      | 2      | 2
--R
--R                                          Type: ContinuedFraction(Integer)
--E 9

--S 10 of 12
partialFraction(7,12)
--R
--R
--R      3 1
--R (10) 1 - -- + -
--R      2 3
--R      2
--R
--R                                          Type: PartialFraction(Integer)
--E 10

--S 11 of 12
g := 2/3 + 4/5*%i
--R
--R
--R      2 4
--R (11) - + - %i
--R      3 5
--R
--R                                          Type: Complex(Fraction(Integer))
--E 11

--S 12 of 12
g :: FRAC COMPLEX INT
--R
--R
--R      10 + 12%i
--R (12) -----
--R      15
--R
--R                                          Type: Fraction(Complex(Integer))
--E 12

)spool
)lisp (bye)

```

```
=====
Fraction examples
=====
```

The Fraction domain implements quotients. The elements must belong to a domain of category IntegralDomain: multiplication must be commutative and the product of two non-zero elements must not be zero. This allows you to make fractions of most things you would think of, but don't expect to create a fraction of two matrices! The abbreviation for Fraction is FRAC.

Use / to create a fraction.

```
a := 11/12
  11
  --
  12
                                Type: Fraction Integer
```

```
b := 23/24
  23
  --
  24
                                Type: Fraction Integer
```

The standard arithmetic operations are available.

```
3 - a*b**2 + a + b/a
313271
-----
76032
                                Type: Fraction Integer
```

Extract the numerator and denominator by using numer and denom, respectively.

```
numer(a)
  11
                                Type: PositiveInteger
```

```
denom(b)
  24
                                Type: PositiveInteger
```

Operations like max, min, negative?, positive? and zero? are all available if they are provided for the numerators and denominators.

Don't expect a useful answer from factor, gcd or lcm if you apply them to fractions.

```

r := (x**2 + 2*x + 1)/(x**2 - 2*x + 1)
      2
      x  + 2x + 1
      -----
      2
      x  - 2x + 1
Type: Fraction Polynomial Integer

```

Since all non-zero fractions are invertible, these operations have trivial definitions.

```

factor(r)
      2
      x  + 2x + 1
      -----
      2
      x  - 2x + 1
Type: Factored Fraction Polynomial Integer

```

Use `map` to apply `factor` to the numerator and denominator, which is probably what you mean.

```

map(factor,r)
      2
      (x + 1)
      -----
      2
      (x - 1)
Type: Fraction Factored Polynomial Integer

```

Other forms of fractions are available. Use `continuedFraction` to create a continued fraction.

```

continuedFraction(7/12)
      1 |      1 |      1 |      1 |
      +---+ + +---+ + +---+ + +---+
      | 1      | 1      | 2      | 2
Type: ContinuedFraction Integer

```

Use `partialFraction` to create a partial fraction.

```

partialFraction(7,12)
      3  1
      1 - -- + -
      2  3
      2
Type: PartialFraction Integer

```

Use `conversion` to create alternative views of fractions with objects

moved in and out of the numerator and denominator.

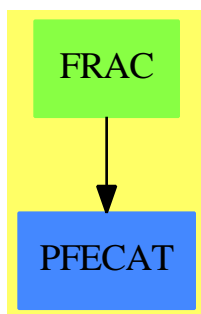
```
g := 2/3 + 4/5*i
      2   4
      - + - %i
      3   5
                                     Type: Complex Fraction Integer
```

```
g :: FRAC COMPLEX INT
      10 + 12%i
      -----
      15
                                     Type: Fraction Complex Integer
```

See Also:

- o)help ContinuedFraction
- o)help PartialFraction
- o)help Integer
- o)show Fraction

Fraction (FRAC)



See

⇒ “Localize” (LO) [13](#) on page [1522](#)

⇒ “LocalAlgebra” (LA) [13](#) on page [1520](#)

Exports:

0	1	abs
associates?	characteristic	charthRoot
ceiling	coerce	conditionP
convert	D	denom
denominator	differentiate	divide
euclideanSize	eval	expressIdealMember
exquo	extendedEuclidean	factor
factorPolynomial	factorSquareFreePolynomial	floor
fractionPart	gcd	gcdPolynomial
hash	init	inv
latex	lcm	map
max	min	multiEuclidean
negative?	nextItem	numer
numerator	OMwrite	one?
patternMatch	positive?	prime?
principalIdeal	random	recip
reducedSystem	retract	retractIfCan
sample	sign	sizeLess?
solveLinearPolynomialEquation	squareFree	squareFreePart
squareFreePolynomial	subtractIfCan	unit?
unitCanonical	unitNormal	wholePart
zero?	?*?	?**?
?+?	?-?	-?
?/?	?=?	?^?
?~=?	?<?	?<=?
?>?	?>=?	?..?
?quo?	?rem?	

— domain FRAC Fraction —

```

)abbrev domain FRAC Fraction
++ Author: Mark Botch
++ Date Last Updated: 12 February 1992
++ Basic Functions: Field, numer, denom
++ Description:
++ Fraction takes an IntegralDomain S and produces
++ the domain of Fractions with numerators and denominators from S.
++ If S is also a GcdDomain, then gcd's between numerator and
++ denominator will be cancelled during all operations.

Fraction(S: IntegralDomain): QuotientFieldCategory S with
  if S has IntegerNumberSystem and S has OpenMath then OpenMath
  if S has canonical and S has GcdDomain and S has canonicalUnitNormal
  then canonical
  ++ \spad{canonical} means that equal elements are in fact identical.
== LocalAlgebra(S, S, S) add
Rep:= Record(num:S, den:S)

```

```

coerce(d:S):% == [d,1]
zero?(x:%) == zero? x.num

if S has GcdDomain and S has canonicalUnitNormal then
  retract(x:%):S ==
--    one?(x.den) => x.num
    ((x.den) = 1) => x.num
    error "Denominator not equal to 1"

  retractIfCan(x:%):Union(S, "failed") ==
--    one?(x.den) => x.num
    ((x.den) = 1) => x.num
    "failed"
else
  retract(x:%):S ==
    (a:= x.num exquo x.den) case "failed" =>
      error "Denominator not equal to 1"
    a
  retractIfCan(x:%):Union(S,"failed") == x.num exquo x.den

if S has EuclideanDomain then
  wholePart x ==
--    one?(x.den) => x.num
    ((x.den) = 1) => x.num
    x.num quo x.den

if S has IntegerNumberSystem then

  floor x ==
--    one?(x.den) => x.num
    ((x.den) = 1) => x.num
    x < 0 => -ceiling(-x)
    wholePart x

  ceiling x ==
--    one?(x.den) => x.num
    ((x.den) = 1) => x.num
    x < 0 => -floor(-x)
    1 + wholePart x

if S has OpenMath then
-- TODO: somewhere this file does something which redefines the division
-- operator. Doh!

writeOMFrac(dev: OpenMathDevice, x: %): Void ==
  OMputApp(dev)
  OMputSymbol(dev, "nums1", "rational")
  OMwrite(dev, x.num, false)
  OMwrite(dev, x.den, false)

```

```

OMputEndApp(dev)

OMwrite(x: %): String ==
  s: String := ""
  sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
  dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML)
  OMputObject(dev)
  writeOMFrac(dev, x)
  OMputEndObject(dev)
  OMclose(dev)
  s := OM_-STRINGPTRTOSTRING(sp)$Lisp pretend String
  s

OMwrite(x: %, wholeObj: Boolean): String ==
  s: String := ""
  sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
  dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML)
  if wholeObj then
    OMputObject(dev)
  writeOMFrac(dev, x)
  if wholeObj then
    OMputEndObject(dev)
  OMclose(dev)
  s := OM_-STRINGPTRTOSTRING(sp)$Lisp pretend String
  s

OMwrite(dev: OpenMathDevice, x: %): Void ==
  OMputObject(dev)
  writeOMFrac(dev, x)
  OMputEndObject(dev)

OMwrite(dev: OpenMathDevice, x: %, wholeObj: Boolean): Void ==
  if wholeObj then
    OMputObject(dev)
  writeOMFrac(dev, x)
  if wholeObj then
    OMputEndObject(dev)

if S has GcdDomain then
  cancelGcd: % -> S
  normalize: % -> %

  normalize x ==
    zero?(x.num) => 0
  -- one?(x.den) => x
    ((x.den) = 1) => x
    uca := unitNormal(x.den)
    zero?(x.den := uca.canonical) => error "division by zero"
    x.num := x.num * uca.associate
  x

```

```

recip x ==
  zero?(x.num) => "failed"
  normalize [x.den, x.num]

cancelGcd x ==
--   one?(x.den) => x.den
  ((x.den) = 1) => x.den
  d := gcd(x.num, x.den)
  xn := x.num exquo d
  xn case "failed" =>
    error "gcd not gcd in QF cancelGcd (numerator)"
  xd := x.den exquo d
  xd case "failed" =>
    error "gcd not gcd in QF cancelGcd (denominator)"
  x.num := xn :: S
  x.den := xd :: S
  d

nn:S / dd:S ==
  zero? dd => error "division by zero"
  cancelGcd(z := [nn, dd])
  normalize z

x + y ==
  zero? y => x
  zero? x => y
  z := [x.den, y.den]
  d := cancelGcd z
  g := [z.den * x.num + z.num * y.num, d]
  cancelGcd g
  g.den := g.den * z.num * z.den
  normalize g

-- We can not rely on the defaulting mechanism
-- to supply a definition for -, even though this
-- definition would do, for the following reasons:
-- 1) The user could have defined a subtraction
--    in Localize, which would not work for
--    QuotientField;
-- 2) even if he doesn't, the system currently
--    places a default definition in Localize,
--    which uses Localize's +, which does not
--    cancel gcDs
x - y ==
  zero? y => x
  z := [x.den, y.den]
  d := cancelGcd z
  g := [z.den * x.num - z.num * y.num, d]
  cancelGcd g

```



```

    g.den := g.den * z.num * z.den
    normalize g

x:% * y:% ==
    zero? x or zero? y => 0
--    one? x => y
    (x = 1) => y
--    one? y => x
    (y = 1) => x
    (x, y) := ([x.num, y.den], [y.num, x.den])
    cancelGcd x; cancelGcd y;
    normalize [x.num * y.num, x.den * y.den]

n:Integer * x:% ==
    y := [n::S, x.den]
    cancelGcd y
    normalize [x.num * y.num, y.den]

nn:S * x:% ==
    y := [nn, x.den]
    cancelGcd y
    normalize [x.num * y.num, y.den]

differentiate(x:%, deriv:S -> S) ==
    y := [deriv(x.den), x.den]
    d := cancelGcd(y)
    y.num := deriv(x.num) * y.den - x.num * y.num
    (d, y.den) := (y.den, d)
    cancelGcd y
    y.den := y.den * d * d
    normalize y

if S has canonicalUnitNormal then
    x = y == (x.num = y.num) and (x.den = y.den)
--x / dd == (cancelGcd (z:=[x.num,dd*x.den])); normalize z

--    one? x == one? (x.num) and one? (x.den)
    one? x == ((x.num) = 1) and ((x.den) = 1)
        -- again assuming canonical nature of representation

else
    nn:S/dd:S == if zero? dd then error "division by zero" else [nn,dd]

    recip x ==
        zero?(x.num) => "failed"
        [x.den, x.num]

if (S has RetractableTo Fraction Integer) then
    retract(x:%):Fraction(Integer) == retract(retract(x)@S)

```

```

    retractIfCan(x: %): Union(Fraction Integer, "failed") ==
      (u := retractIfCan(x) @ Union(S, "failed")) case "failed" => "failed"
      retractIfCan(u::S)

  else if (S has RetractableTo Integer) then
    retract(x: %): Fraction(Integer) ==
      retract(number x) / retract(denom x)

    retractIfCan(x: %): Union(Fraction Integer, "failed") ==
      (n := retractIfCan numer x) case "failed" => "failed"
      (d := retractIfCan denom x) case "failed" => "failed"
      (n::Integer) / (d::Integer)

  QFP ==> SparseUnivariatePolynomial %
  DP ==> SparseUnivariatePolynomial S
  import UnivariatePolynomialCategoryFunctions2(%, QFP, S, DP)
  import UnivariatePolynomialCategoryFunctions2(S, DP, %, QFP)

  if S has GcdDomain then
    gcdPolynomial(pp, qq) ==
      zero? pp => qq
      zero? qq => pp
      zero? degree pp or zero? degree qq => 1
      denpp := "lcm" / [denom u for u in coefficients pp]
      ppD: DP := map(x+>retract(x*denpp), pp)
      denqq := "lcm" / [denom u for u in coefficients qq]
      qqD: DP := map(x+>retract(x*denqq), qq)
      g := gcdPolynomial(ppD, qqD)
      zero? degree g => 1
--      one? (lc := leadingCoefficient g) => map(#1::%, g)
      ((lc := leadingCoefficient g) = 1) => map(x+>x::%, g)
      map(x+>x/lc, g)

  if (S has PolynomialFactorizationExplicit) then
    -- we'll let the solveLinearPolynomialEquations operator
    -- default from Field
    pp, qq: QFP
    lpp: List QFP
    import Factored SparseUnivariatePolynomial %
    if S has CharacteristicNonZero then
      if S has canonicalUnitNormal and S has GcdDomain then
        charthRoot x ==
          n := charthRoot x.num
          n case "failed" => "failed"
          d := charthRoot x.den
          d case "failed" => "failed"
          n/d
      else
        charthRoot x ==
          -- to find x = p-th root of n/d

```

```

-- observe that xd is p-th root of n*d**(p-1)
ans:=charthRoot(x.num *
  (x.den)**(characteristic()$%-1)::NonNegativeInteger)
ans case "failed" => "failed"
ans / x.den
clear: List % -> List S
clear l ==
  d:="lcm"/[x.den for x in l]
  [ x.num * (d exquo x.den)::S for x in l]
mat: Matrix %
conditionP mat ==
  matD: Matrix S
  matD:= matrix [ clear l for l in listOfLists mat ]
  ansD := conditionP matD
  ansD case "failed" => "failed"
  ansDD:=ansD :: Vector(S)
  [ ansDD(i)::% for i in 1..#ansDD]$Vector(%)

factorPolynomial(pp) ==
  zero? pp => 0
  denpp:="lcm"/[denom u for u in coefficients pp]
  ppD:DP:=map(x+>retract(x*denpp),pp)
  ff:=factorPolynomial ppD
  den1:%=denpp::%
  lfact>List Record(flg:Union("nil", "sqfr", "irred", "prime"),
    fctr:QFP, xpnt:Integer)
  lfact:= [[w.flg,
    if leadingCoefficient w.fctr =1 then
      map(x+>x::%,w.fctr)
    else (lc:=(leadingCoefficient w.fctr)::%;
      den1:=den1/lc**w.xpnt;
      map(x+>x::%/lc,w.fctr)),
    w.xpnt] for w in factorList ff]
  makeFR(map(x+>x::%/den1,unit(ff)),lfact)
factorSquareFreePolynomial(pp) ==
  zero? pp => 0
  degree pp = 0 => makeFR(pp,empty())
  lcpp:=leadingCoefficient pp
  pp:=pp/lcpp
  denpp:="lcm"/[denom u for u in coefficients pp]
  ppD:DP:=map(x+>retract(x*denpp),pp)
  ff:=factorSquareFreePolynomial ppD
  den1:%=denpp::%/lcpp
  lfact>List Record(flg:Union("nil", "sqfr", "irred", "prime"),
    fctr:QFP, xpnt:Integer)
  lfact:= [[w.flg,
    if leadingCoefficient w.fctr =1 then
      map(x+>x::%,w.fctr)
    else (lc:=(leadingCoefficient w.fctr)::%;
      den1:=den1/lc**w.xpnt;

```

```

        map(x+>x::%/lc,w.fctr)),
      w.xpnt] for w in factorList ff]
makeFR(map(x+>x::%/den1,unit(ff)),lfact)

```

— FRAC.dotabb —

```

"FRAC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FRAC"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FRAC" -> "PFECAT"

```

domain FRIDEAL FractionalIdeal

— FractionalIdeal.input —

```

)set break resume
)sys rm -f FractionalIdeal.output
)spool FractionalIdeal.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FractionalIdeal
--R
--R FractionalIdeal(R: EuclideanDomain,F: QuotientFieldCategory(R),UP: UnivariatePolynomialCategory(F),A: J
--R Abbreviation for FractionalIdeal is FRIDEAL
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FRIDEAL
--R
--R----- Operations -----
--R ?? : (%,% ) -> %
--R ??? : (% ,PositiveInteger) -> %
--R ?? : (%,% ) -> Boolean
--R ?? : (% ,Integer) -> %
--R basis : % -> Vector(A)
--R commutator : (%,% ) -> %
--R denom : % -> R
--R ideal : Vector(A) -> %
--R latex : % -> String
--R norm : % -> F
--R ??? : (% ,Integer) -> %
--R ?/? : (%,% ) -> %
--R 1 : () -> %
--R ?? : (% ,PositiveInteger) -> %
--R coerce : % -> OutputForm
--R conjugate : (%,% ) -> %
--R hash : % -> SingleInteger
--R inv : % -> %
--R minimize : % -> %
--R numer : % -> Vector(A)

```

```

--R one? : % -> Boolean                      recip : % -> Union(%, "failed")
--R sample : () -> %                          ?~=? : (%,%) -> Boolean
--R ?**? : (%,NonNegativeInteger) -> %
--R ?^? : (%,NonNegativeInteger) -> %
--R randomLC : (NonNegativeInteger,Vector(A)) -> A
--R
--E 1

)spool
)lisp (bye)

```

— FractionalIdeal.help —

```

=====
FractionalIdeal examples
=====

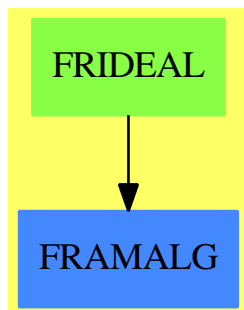
```

Fractional ideals in a framed algebra.

See Also:

o)show FractionalIdeal

FractionalIdeal (FRIDEAL)



See

- ⇒ “FramedModule” (FRMOD) [7](#) on page [1000](#)
- ⇒ “HyperellipticFiniteDivisor” (HELLFDIV) [9](#) on page [1183](#)
- ⇒ “FiniteDivisor” (FDIV) [7](#) on page [813](#)

Exports:

1	basis	coerce	commutator	conjugate
denom	hash	ideal	inv	latex
minimize	norm	numer	one?	randomLC
recip	sample	?~=?	?**?	?^?
?*?	?**?	?/?	?=?	?^?

— domain FRIDEAL FractionalIdeal —

```

)abbrev domain FRIDEAL FractionalIdeal
++ Author: Manuel Bronstein
++ Date Created: 27 Jan 1989
++ Date Last Updated: 30 July 1993
++ Description:
++ Fractional ideals in a framed algebra.

FractionalIdeal(R, F, UP, A): Exports == Implementation where
  R : EuclideanDomain
  F : QuotientFieldCategory R
  UP: UnivariatePolynomialCategory F
  A : Join(FramedAlgebra(F, UP), RetractableTo F)

VF ==> Vector F
VA ==> Vector A
UPA ==> SparseUnivariatePolynomial A
QF ==> Fraction UP

Exports ==> Group with
  ideal   : VA -> %
    ++ ideal([f1,...,fn]) returns the ideal \spad{(f1,...,fn)}.
  basis   : % -> VA
    ++ basis((f1,...,fn)) returns the vector \spad{[f1,...,fn]}.
  norm    : % -> F
    ++ norm(I) returns the norm of the ideal I.
  numer   : % -> VA
    ++ numer(1/d * (f1,...,fn)) = the vector \spad{[f1,...,fn]}.
  denom   : % -> R
    ++ denom(1/d * (f1,...,fn)) returns d.
  minimize: % -> %
    ++ minimize(I) returns a reduced set of generators for \spad{I}.
  randomLC: (NonNegativeInteger, VA) -> A
    ++ randomLC(n,x) should be local but conditional.

Implementation ==> add
  import CommonDenominator(R, F, VF)
  import MatrixCommonDenominator(UP, QF)
  import InnerCommonDenominator(R, F, List R, List F)
  import MatrixCategoryFunctions2(F, Vector F, Vector F, Matrix F,
    UP, Vector UP, Vector UP, Matrix UP)
  import MatrixCategoryFunctions2(UP, Vector UP, Vector UP,

```

```

Matrix UP, F, Vector F, Vector F, Matrix F)
import MatrixCategoryFunctions2(UP, Vector UP, Vector UP,
Matrix UP, QF, Vector QF, Vector QF, Matrix QF)

Rep := Record(num:VA, den:R)

poly      : % -> UPA
invrep     : Matrix F -> A
upmat      : (A, NonNegativeInteger) -> Matrix UP
summat     : % -> Matrix UP
num2O      : VA -> OutputForm
agcd       : List A -> R
vgcd       : VF -> R
mkIdeal    : (VA, R) -> %
intIdeal   : (List A, R) -> %
ret?       : VA -> Boolean
tryRange   : (NonNegativeInteger, VA, R, %) -> Union(%, "failed")

1          == [[1]$VA, 1]
numer i    == i.num
denom i     == i.den
mkIdeal(v, d) == [v, d]
invrep m    == represents(transpose(m) * coordinates(1$A))
upmat(x, i) == map(s +-> monomial(s, i)$UP, regularRepresentation x)
ret? v      == any?(s+>retractIfCan(s)@Union(F,"failed") case F, v)
x = y      == denom(x) = denom(y) and numer(x) = numer(y)
agcd l      == reduce("gcd", [vgcd coordinates a for a in l]$List(R), 0)

norm i ==
  ("gcd"/[retract(u)@R for u in coefficients determinant summat i])
    / denom(i) ** rank()$A

tryRange(range, nm, nrm, i) ==
  for j in 0..10 repeat
    a := randomLC(10 * range, nm)
    unit? gcd((retract(norm a)@R exquo nrm)::R, nrm) =>
      return intIdeal([nrm::F::A, a], denom i)
  "failed"

summat i ==
  m := minIndex(v := numer i)
  reduce("+",
    [upmat(qelt(v, j + m), j) for j in 0..#v-1]$List(Matrix UP))

inv i ==
  m := inverse(map(s+>s::QF, summat i))::Matrix(QF)
  cd := splitDenominator(denom(i)::F::UP::QF * m)
  cd2 := splitDenominator coefficients(cd.den)
  invd:= cd2.den / reduce("gcd", cd2.num)
  d := reduce("max", [degree p for p in parts(cd.num)])

```

```

ideal
  [invd * invrep map(s+>coefficient(s, j), cd.num) for j in 0..d]$VA

ideal v ==
  d := reduce("lcm", [commonDenominator coordinates qelt(v, i)
    for i in minIndex v .. maxIndex v]$List(R))
  intIdeal([d::F * qelt(v, i) for i in minIndex v .. maxIndex v], d)

intIdeal(l, d) ==
  lr := empty()$List(R)
  nr := empty()$List(A)
  for x in removeDuplicates l repeat
    if (u := retractIfCan(x)@Union(F, "failed")) case F
      then lr := concat(retract(u::F)@R, lr)
      else nr := concat(x, nr)
  r := reduce("gcd", lr, 0)
  g := agcd nr
  a := (r quo (b := gcd(gcd(d, r), g))):F::A
  d := d quo b
  r ^= 0 and ((g exquo r) case R) => mkIdeal([a], d)
  invb := inv(b::F)
  va:VA := [invb * m for m in nr]
  zero? a => mkIdeal(va, d)
  mkIdeal(concat(a, va), d)

vgcd v ==
  reduce("gcd",
    [retract(v.i)@R for i in minIndex v .. maxIndex v]$List(R))

poly i ==
  m := minIndex(v := numer i)
  +/[monomial(qelt(v, i + m), i) for i in 0..#v-1]

i1 * i2 ==
  intIdeal(coefficients(poly i1 * poly i2), denom i1 * denom i2)

i:$ ** m:Integer ==
  m < 0 => inv(i) ** (-m)
  n := m::NonNegativeInteger
  v := numer i
  intIdeal([qelt(v, j) ** n for j in minIndex v .. maxIndex v],
    denom(i) ** n)

num20 v ==
  paren [qelt(v, i)::OutputForm
    for i in minIndex v .. maxIndex v]$List(OutputForm)

basis i ==
  v := numer i
  d := inv(denom(i)::F)

```



```

[d * qelt(v, j) for j in minIndex v .. maxIndex v]

coerce(i:$):OutputForm ==
  nm := num20 numer i
--   one? denom i => nm
  (denom i = 1) => nm
  (1::Integer::OutputForm) / (denom(i)::OutputForm) * nm

if F has Finite then
  randomLC(m, v) ==
    +/[random()$F * qelt(v, j) for j in minIndex v .. maxIndex v]
else
  randomLC(m, v) ==
    +/[(random()$Integer rem m::Integer) * qelt(v, j)
      for j in minIndex v .. maxIndex v]

minimize i ==
  n := (#(nm := numer i))
--   one?(n) or (n < 3 and ret? nm) => i
  (n = 1) or (n < 3 and ret? nm) => i
  nrm := retract(norm mkIdeal(nm, 1))@R
  for range in 1..5 repeat
    (u := tryRange(range, nm, nrm, i)) case $ => return(u::$)
  i

```

— FRIDEAL.dotabb —

```

"FRIDEAL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FRIDEAL"]
"FRAMALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRAMALG"]
"FRIDEAL" -> "FRAMALG"

```

domain FRMOD FramedModule

— FramedModule.input —

```

)set break resume
)sys rm -f FramedModule.output
)spool FramedModule.output
)set message test on
)set message auto off

```

```
)clear all
```

```
--S 1 of 1
```

```
)show FramedModule
```

```
--R
```

```
--R FramedModule(R: EuclideanDomain,F: QuotientFieldCategory(R),UP: UnivariatePolynomialCategory(F),A: FramedModule(R,F,UP,A))
```

```
--R Abbreviation for FramedModule is FRMOD
```

```
--R This constructor is not exposed in this frame.
```

```
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FRMOD
```

```
--R
```

```
--R----- Operations -----
```

```
--R ?? : (%,% ) -> %                                ??? : (% ,PositiveInteger) -> %
```

```
--R ?=? : (%,% ) -> Boolean                            1 : () -> %
```

```
--R ?^? : (% ,PositiveInteger) -> %                    basis : % -> Vector(A)
```

```
--R coerce : % -> OutputForm                            hash : % -> SingleInteger
```

```
--R latex : % -> String                                module : Vector(A) -> %
```

```
--R norm : % -> F                                        one? : % -> Boolean
```

```
--R recip : % -> Union(%,"failed")                    sample : () -> %
```

```
--R ?^=? : (%,% ) -> Boolean
```

```
--R ??? : (% ,NonNegativeInteger) -> %
```

```
--R ?^? : (% ,NonNegativeInteger) -> %
```

```
--R module : FractionalIdeal(R,F,UP,A) -> % if A has RETRACT(F)
```

```
--R
```

```
--E 1
```

```
)spool
```

```
)lisp (bye)
```

— FramedModule.help —

```
=====
```

```
FramedModule examples
```

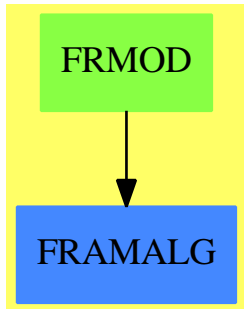
```
=====
```

Module representation of fractional ideals.

See Also:

o)show FramedModule

FramedModule (FRMOD)



See

- ⇒ “FractionalIdeal” (FRIDEAL) [7](#) on page [994](#)
- ⇒ “HyperellipticFiniteDivisor” (HELLFDIV) [9](#) on page [1183](#)
- ⇒ “FiniteDivisor” (FDIV) [7](#) on page [813](#)

Exports:

1	basis	coerce	hash	latex
module	norm	one?	recip	sample
?~=?	?**?	?^?	?*?	?**?
?=?				

— domain FRMOD FramedModule —

```

)abbrev domain FRMOD FramedModule
++ Author: Manuel Bronstein
++ Date Created: 27 Jan 1989
++ Date Last Updated: 24 Jul 1990
++ Description:
++ Module representation of fractional ideals.

FramedModule(R, F, UP, A, ibasis): Exports == Implementation where
  R      : EuclideanDomain
  F      : QuotientFieldCategory R
  UP     : UnivariatePolynomialCategory F
  A      : FramedAlgebra(F, UP)
  ibasis: Vector A

  VR ==> Vector R
  VF ==> Vector F
  VA ==> Vector A
  M  ==> Matrix F

Exports ==> Monoid with
  basis : % -> VA
    ++ basis((f1,...,fn)) = the vector \spad{[f1,...,fn]}.
  norm  : % -> F
  
```

```

    ++ norm(f) returns the norm of the module f.
module: VA -> %
    ++ module([f1,...,fn]) = the module generated by \spad{(f1,...,fn)}
    ++ over R.
if A has RetractableTo F then
    module: FractionalIdeal(R, F, UP, A) -> %
        ++ module(I) returns I viewed has a module over R.

Implementation ==> add
import MatrixCommonDenominator(R, F)
import ModularHermitianRowReduction(R)

Rep := VA

iflag?:Reference(Boolean) := ref true
wflag?:Reference(Boolean) := ref true
imat := new(#ibasis, #ibasis, 0)$M
wmat := new(#ibasis, #ibasis, 0)$M

rowdiv      : (VR, R)  -> VF
vectProd    : (VA, VA) -> VA
wmatrix     : VA -> M
W2A        : VF -> A
intmat      : () -> M
invintmat   : () -> M
getintmat   : () -> Boolean
getinvintmat: () -> Boolean

1              == ibasis
module(v:VA)   == v
basis m        == m pretend VA
rowdiv(r, f)   == [r.i / f for i in minIndex r..maxIndex r]
coerce(m:%):OutputForm == coerce(basis m)$VA
W2A v         == represents(v * intmat())
wmatrix v     == coordinates(v) * invintmat()

getinvintmat() ==
    m := inverse(intmat()):M
    for i in minRowIndex m .. maxRowIndex m repeat
        for j in minColIndex m .. maxColIndex m repeat
            imat(i, j) := qelt(m, i, j)
    false

getintmat() ==
    m := coordinates ibasis
    for i in minRowIndex m .. maxRowIndex m repeat
        for j in minColIndex m .. maxColIndex m repeat
            wmat(i, j) := qelt(m, i, j)
    false

```

```

invintmat() ==
  if iflag?() then iflag?() := getinvintmat()
  imat

intmat() ==
  if wflag?() then wflag?() := getintmat()
  wmat

vectProd(v1, v2) ==
  k := minIndex(v := new(#v1 * #v2, 0)$VA)
  for i in minIndex v1 .. maxIndex v1 repeat
    for j in minIndex v2 .. maxIndex v2 repeat
      qsetelt_!(v, k, qelt(v1, i) * qelt(v2, j))
      k := k + 1
  v pretend VA

norm m ==
  #(basis m) ^= #ibasis => error "Module not of rank n"
  determinant(coordinates(basis m) * invintmat())

m1 * m2 ==
  m := rowEch((cd := splitDenominator wmatrix(
    vectProd(basis m1, basis m2))).num)
  module [u for i in minRowIndex m .. maxRowIndex m |
    (u := W2A rowdiv(row(m, i), cd.den)) ^= 0]$VA

if A has RetractableTo F then
  module(i:FractionalIdeal(R, F, UP, A)) ==
    module(basis i) * module(ibasis)

```

— FRMOD.dotabb —

```

"FRMOD" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FRMOD"]
"FRAMALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRAMALG"]
"FRMOD" -> "FRAMALG"

```

domain FAGROUP FreeAbelianGroup

— FreeAbelianGroup.input —

```
)set break resume
```

```

)sys rm -f FreeAbelianGroup.output
)spool FreeAbelianGroup.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FreeAbelianGroup
--R
--R FreeAbelianGroup(S: SetCategory) is a domain constructor
--R Abbreviation for FreeAbelianGroup is FAGROUP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FAGROUP
--R
--R----- Operations -----
--R ?? : (Integer,S) -> %           ?? : (%,Integer) -> %
--R ?? : (Integer,%) -> %           ?? : (PositiveInteger,%) -> %
--R ?+? : (S,%) -> %               ?+? : (%,%) -> %
--R ?-? : (%,) -> %               -? : % -> %
--R ?? : (%,) -> Boolean           0 : () -> %
--R coefficient : (S,%) -> Integer  coerce : S -> %
--R coerce : % -> OutputForm        hash : % -> SingleInteger
--R latex : % -> String              mapGen : ((S -> S),%) -> %
--R nthCoef : (%,Integer) -> Integer nthFactor : (%,Integer) -> S
--R retract : % -> S                sample : () -> %
--R size : % -> NonNegativeInteger  zero? : % -> Boolean
--R ?~=? : (%,) -> Boolean
--R ?? : (NonNegativeInteger,%) -> %
--R ?<? : (%,) -> Boolean if S has ORDSET
--R ?<=? : (%,) -> Boolean if S has ORDSET
--R ?>? : (%,) -> Boolean if S has ORDSET
--R ?>=? : (%,) -> Boolean if S has ORDSET
--R highCommonTerms : (%,) -> % if Integer has OAMON
--R mapCoef : ((Integer -> Integer),%) -> %
--R max : (%,) -> % if S has ORDSET
--R min : (%,) -> % if S has ORDSET
--R retractIfCan : % -> Union(S,"failed")
--R subtractIfCan : (%,) -> Union(%,"failed")
--R terms : % -> List(Record(gen: S,exp: Integer))
--R
--E 1

)spool
)lisp (bye)

```

```
=====
FreeAbelianGroup examples
=====
```

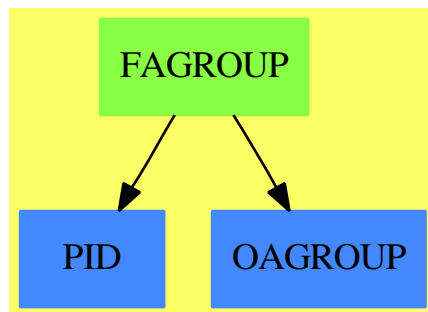
Free abelian group on any set of generators

The free abelian group on a set S is the monoid of finite sums of the form $\text{reduce}(+, [n_i * s_i])$ where the s_i 's are in S , and the n_i 's are integers. The operation is commutative.

See Also:

```
o )show FreeAbelianGroup
```

FreeAbelianGroup (FAGROUP)



See

⇒ “ListMonoidOps” (LMOPS) [13](#) on page [1509](#)

⇒ “FreeMonoid” (FMONOID) [7](#) on page [1021](#)

⇒ “FreeGroup” (FGROUP) [7](#) on page [1010](#)

⇒ “InnerFreeAbelianMonoid” (IFAMON) [10](#) on page [1286](#)

⇒ “FreeAbelianMonoid” (FAMONOID) [7](#) on page [1007](#)

Exports:

0	coefficient	coerce	hash	highCommonTerms
latex	mapCoef	mapGen	max	min
nthCoef	nthFactor	retract	retractIfCan	sample
size	subtractIfCan	terms	zero?	?~=?
?*?	?<?	?<=?	?>?	?>=?
?+?	?-?	-?	?=?	

— domain **FAGROUP** **FreeAbelianGroup** —

```
)abbrev domain FAGROUP FreeAbelianGroup
```

```
++ Author: Manuel Bronstein
```

```

++ Date Created: November 1989
++ Date Last Updated: 6 June 1991
++ Description:
++ Free abelian group on any set of generators
++ The free abelian group on a set S is the monoid of finite sums of
++ the form \spad{reduce(+,[ni * si])} where the si's are in S, and the ni's
++ are integers. The operation is commutative.

```

```

FreeAbelianGroup(S:SetCategory): Exports == Implementation where
  Exports ==> Join(AbelianGroup, Module Integer,
    FreeAbelianMonoidCategory(S, Integer)) with
    if S has OrderedSet then OrderedSet

```

```

Implementation ==> InnerFreeAbelianMonoid(S, Integer, 1) add
  - f == mapCoef("-", f)

```

```

if S has OrderedSet then
  inmax: List Record(gen: S, exp: Integer) -> Record(gen: S, exp: Integer)

```

```

  inmax l ==
    mx := first l
    for t in rest l repeat
      if mx.gen < t.gen then mx := t
    mx

```

```

-- lexicographic order
a < b ==
  zero? a =>
    zero? b => false
    0 < (inmax terms b).exp
  ta := inmax terms a
  zero? b => ta.exp < 0
  tb := inmax terms b
  ta.gen < tb.gen => 0 < tb.exp
  tb.gen < ta.gen => ta.exp < 0
  ta.exp < tb.exp => true
  tb.exp < ta.exp => false
  lc := ta.exp * ta.gen
  (a - lc) < (b - lc)

```

— FAGROUP.dotabb —

```

"FAGROUP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FAGROUP"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"FAGROUP" -> "PID"

```



```
"FAGROUP" -> "OAGROUP"
```

domain FAMONOID FreeAbelianMonoid

— FreeAbelianMonoid.input —

```
)set break resume
)sys rm -f FreeAbelianMonoid.output
)spool FreeAbelianMonoid.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FreeAbelianMonoid
--R
--R FreeAbelianMonoid(S: SetCategory) is a domain constructor
--R Abbreviation for FreeAbelianMonoid is FAMONOID
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FAMONOID
--R
--R----- Operations -----
--R ?? : (NonNegativeInteger,S) -> %      ?? : (PositiveInteger,%) -> %
--R ?+? : (S,%) -> %                      ?+? : (%,%) -> %
--R ?=? : (%,%) -> Boolean                0 : () -> %
--R coerce : S -> %                      coerce : % -> OutputForm
--R hash : % -> SingleInteger             latex : % -> String
--R mapGen : ((S -> S),%) -> %            nthFactor : (%,Integer) -> S
--R retract : % -> S                     sample : () -> %
--R size : % -> NonNegativeInteger        zero? : % -> Boolean
--R ?^=? : (%,%) -> Boolean
--R ?? : (NonNegativeInteger,%) -> %
--R coefficient : (S,%) -> NonNegativeInteger
--R highCommonTerms : (%,%) -> % if NonNegativeInteger has OAMON
--R mapCoef : ((NonNegativeInteger -> NonNegativeInteger),%) -> %
--R nthCoef : (%,Integer) -> NonNegativeInteger
--R retractIfCan : % -> Union(S,"failed")
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R terms : % -> List(Record(gen: S,exp: NonNegativeInteger))
--R
--E 1

)spool
)lisp (bye)
```

— **FreeAbelianMonoid.help** —

=====

FreeAbelianMonoid examples

=====

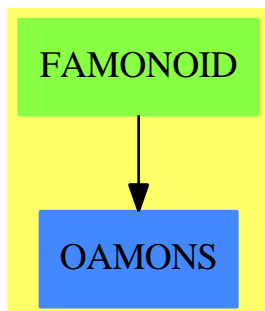
Free abelian monoid on any set of generators

The free abelian monoid on a set S is the monoid of finite sums of the form `reduce(+,[ni * si])` where the si 's are in S , and the ni 's are non-negative integers. The operation is commutative.

See Also:

o `)show FreeAbelianMonoid`

FreeAbelianMonoid (FAMONOID)



See

- ⇒ “ListMonoidOps” (LMOPS) [13](#) on page [1509](#)
- ⇒ “FreeMonoid” (FMONOID) [7](#) on page [1021](#)
- ⇒ “FreeGroup” (FGROUP) [7](#) on page [1010](#)
- ⇒ “InnerFreeAbelianMonoid” (IFAMON) [10](#) on page [1286](#)
- ⇒ “FreeAbelianGroup” (FAGROUP) [7](#) on page [1004](#)

Exports:

<code>0</code>	<code>coefficient</code>	<code>coerce</code>	<code>hash</code>	<code>highCommonTerms</code>
<code>latex</code>	<code>mapCoef</code>	<code>mapGen</code>	<code>nthCoef</code>	<code>nthFactor</code>
<code>retract</code>	<code>retractIfCan</code>	<code>sample</code>	<code>size</code>	<code>subtractIfCan</code>
<code>terms</code>	<code>zero?</code>	<code>?~=?</code>	<code>?*?</code>	<code>?+?</code>
<code>?=?</code>				

```

)abbrev domain FAMONOID FreeAbelianMonoid
++ Author: Manuel Bronstein
++ Date Created: November 1989
++ Date Last Updated: 6 June 1991
++ Description:
++ Free abelian monoid on any set of generators
++ The free abelian monoid on a set S is the monoid of finite sums of
++ the form \spad{reduce(+,[ni * si])} where the si's are in S, and the ni's
++ are non-negative integers. The operation is commutative.

FreeAbelianMonoid(S: SetCategory):
  FreeAbelianMonoidCategory(S, NonNegativeInteger)
    == InnerFreeAbelianMonoid(S, NonNegativeInteger, 1)

```

— FAMONOID.dotabb —

```

"FAMONOID" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FAMONOID"]
"OAMONS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAMONS"]
"FAMONOID" -> "OAMONS"

```

domain FGROUP FreeGroup

— FreeGroup.input —

```

)set break resume
)sys rm -f FreeGroup.output
)spool FreeGroup.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FreeGroup
--R
--R FreeGroup(S: SetCategory) is a domain constructor
--R Abbreviation for FreeGroup is FGROU
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FGROU
--R
--R----- Operations -----

```

```

--R ?? : (S,% ) -> %
--R ?? : (%,% ) -> %
--R ??? : (%,Integer) -> %
--R ?/? : (%,% ) -> %
--R 1 : () -> %
--R ?? : (%,PositiveInteger) -> %
--R coerce : % -> OutputForm
--R conjugate : (%,% ) -> %
--R inv : % -> %
--R mapGen : ((S -> S),%) -> %
--R nthFactor : (%,Integer) -> S
--R recip : % -> Union(%,"failed")
--R sample : () -> %
--R ?=? : (%,% ) -> Boolean
--R ??? : (%,NonNegativeInteger) -> %
--R ?? : (%,NonNegativeInteger) -> %
--R factors : % -> List(Record(gen: S,exp: Integer))
--R mapExpon : ((Integer -> Integer),%) -> %
--R retractIfCan : % -> Union(S,"failed")
--R
--E 1

```

```

)spool
)lisp (bye)

```

— FreeGroup.help —

FreeGroup examples

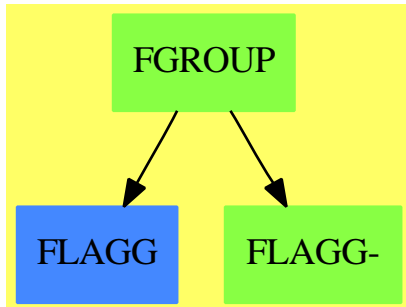
Free group on any set of generators

The free group on a set S is the group of finite products of the form $\text{reduce}(*,[s_i ** n_i])$ where the s_i 's are in S , and the n_i 's are integers. The multiplication is not commutative.

See Also:

o)show FreeGroup

FreeGroup (FGROUP)



See

- ⇒ “ListMonoidOps” (LMOPS) [13](#) on page [1509](#)
- ⇒ “FreeMonoid” (FMONOID) [7](#) on page [1021](#)
- ⇒ “InnerFreeAbelianMonoid” (IFAMON) [10](#) on page [1286](#)
- ⇒ “FreeAbelianMonoid” (FAMONOID) [7](#) on page [1007](#)
- ⇒ “FreeAbelianGroup” (FAGROUP) [7](#) on page [1004](#)

Exports:

1	coerce	commutator	conjugate	factors
hash	inv	latex	mapExpon	mapGen
nthExpon	nthFactor	one?	recip	retract
retractIfCan	sample	size	?~=?	?**?
?^?	?*?	?/?	?=?	

— domain FGROUP FreeGroup —

```

)abbrev domain FGROUP FreeGroup
++ Author: Stephen M. Watt
++ Date Last Updated: 6 June 1991
++ Description:
++ Free group on any set of generators
++ The free group on a set S is the group of finite products of
++ the form \spad{reduce(*,[si ** ni])} where the si's are in S, and the ni's
++ are integers. The multiplication is not commutative.

```

```

FreeGroup(S: SetCategory): Join(Group, RetractableTo S) with
  "*":      (S, $) -> $
    ++ s * x returns the product of x by s on the left.
  "*":      ($, S) -> $
    ++ x * s returns the product of x by s on the right.
  "**":      (S, Integer) -> $
    ++ s ** n returns the product of s by itself n times.
  size      : $ -> NonNegativeInteger
    ++ size(x) returns the number of monomials in x.
  nthExpon   : ($, Integer) -> Integer

```

```

    ++ nthExpon(x, n) returns the exponent of the nth monomial of x.
nthFactor    : ($, Integer) -> S
    ++ nthFactor(x, n) returns the factor of the nth monomial of x.
mapExpon     : (Integer -> Integer, $) -> $
    ++ mapExpon(f, a1e1 ... anen) returns
    ++ \spad{a1f(e1) ... anf(en)}.
mapGen       : (S -> S, $) -> $
    ++ mapGen(f, a1e1 ... anen) returns
    ++ \spad{f(a1)e1 ... f(an)en}.
factors      : $ -> List Record(gen: S, exp: Integer)
    ++ factors(a1e1, ..., anen) returns \spad{[[a1, e1], ..., [an, en]]}.
== ListMonoidOps(S, Integer, 1) add
Rep := ListMonoidOps(S, Integer, 1)

1                == makeUnit()
one? f           == empty? listOfMonoms f
s:S ** n:Integer == makeTerm(s, n)
f:$ * s:S        == rightMult(f, s)
s:S * f:$        == leftMult(s, f)
inv f            == reverse_! mapExpon("-", f)
factors f        == copy listOfMonoms f
mapExpon(f, x)   == mapExpon(f, x)$Rep
mapGen(f, x)     == mapGen(f, x)$Rep
coerce(f:$):OutputForm == outputForm(f, "*", "**", 1)

f:$ * g:$ ==
  one? f => g
  one? g => f
  r := reverse listOfMonoms f
  q := copy listOfMonoms g
  while not empty? r and not empty? q and r.first.gen = q.first.gen
    and r.first.exp = -q.first.exp repeat
    r := rest r
    q := rest q
  empty? r => makeMulti q
  empty? q => makeMulti reverse_! r
  r.first.gen = q.first.gen =>
    setlast_!(h := reverse_! r,
              [q.first.gen, q.first.exp + r.first.exp])
  makeMulti concat_!(h, rest q)
  makeMulti concat_!(reverse_! r, q)

```

— FGROUPE.dotabb —

"FGROUPE" [[color="#88FF44",href="bookvol10.3.pdf#nameddest=FGROUPE"](bookvol10.3.pdf#nameddest=FGROUPE)]

"FLAGG" [[color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"](bookvol10.2.pdf#nameddest=FLAGG)]

```
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"FGROUP" -> "FLAGG"
"FGROUP" -> "FLAGG-"
```

domain FM FreeModule

— FreeModule.input —

```
)set break resume
)sys rm -f FreeModule.output
)spool FreeModule.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FreeModule
--R FreeModule(R: Ring,S: OrderedSet) is a domain constructor
--R Abbreviation for FreeModule is FM
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FM
--R
--R----- Operations -----
--R ???: (R,R) -> %           ???: (R,%) -> %
--R ???: (Integer,%) -> %     ???: (PositiveInteger,%) -> %
--R ?+?: (%,%) -> %          ?-?: (%,%) -> %
--R -?: % -> %                ?=: (%,%) -> Boolean
--R 0 : () -> %               coerce : % -> OutputForm
--R hash : % -> SingleInteger latex : % -> String
--R leadingCoefficient : % -> R leadingSupport : % -> S
--R map : ((R -> R),%) -> %    monomial : (R,S) -> %
--R reductum : % -> %          sample : () -> %
--R zero? : % -> Boolean       ?~=? : (%,%) -> Boolean
--R ???: (NonNegativeInteger,%) -> %
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)
```

— FreeModule.help —

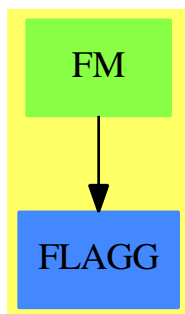
```
=====
FreeModule examples
=====
```

A bi-module is a free module over a ring with generators indexed by an ordered set. Each element can be expressed as a finite linear combination of generators. Only non-zero terms are stored.

See Also:

o)show FreeModule

FreeModule (FM)



See

⇒ “PolynomialRing” (PR) [17](#) on page [2092](#)

⇒ “SparseUnivariatePolynomial” (SUP) [20](#) on page [2466](#)

⇒ “UnivariatePolynomial” (UP) [22](#) on page [2827](#)

Exports:

0	coerce	hash	latex	leadingCoefficient
leadingSupport	map	monomial	reductum	sample
subtractIfCan	zero?	?~=?	?*?	?+?
?-?	-?	?=?		

— domain FM FreeModule —

```
)abbrev domain FM FreeModule
++ Author: Dave Barton, James Davenport, Barry Trager
++ Description:
++ A bi-module is a free module
++ over a ring with generators indexed by an ordered set.
++ Each element can be expressed as a finite linear combination of
++ generators. Only non-zero terms are stored.
```



```

FreeModule(R:Ring,S:OrderedSet):
  Join(BiModule(R,R),IndexedDirectProductCategory(R,S)) with
  if R has CommutativeRing then Module(R)
== IndexedDirectProductAbelianGroup(R,S) add
--representations
  Term:= Record(k:S,c:R)
  Rep:= List Term
--declarations
  x,y: %
  r: R
  n: Integer
  f: R -> R
  s: S
--define
  if R has EntireRing then
    r * x ==
      zero? r => 0
--      one? r => x
      (r = 1) => x
      --map(r##1,x)
      [[u.k,r*u.c] for u in x ]
  else
    r * x ==
      zero? r => 0
--      one? r => x
      (r = 1) => x
      --map(r##1,x)
      [[u.k,a] for u in x | (a:=r*u.c) ^= 0$R]
  if R has EntireRing then
    x * r ==
      zero? r => 0
--      one? r => x
      (r = 1) => x
      --map(r##1,x)
      [[u.k,u.c*r] for u in x ]
  else
    x * r ==
      zero? r => 0
--      one? r => x
      (r = 1) => x
      --map(r##1,x)
      [[u.k,a] for u in x | (a:=u.c*r) ^= 0$R]

coerce(x) : OutputForm ==
  null x => (0$R) :: OutputForm
  le : List OutputForm := nil
  for rec in reverse x repeat
    rec.c = 1 => le := cons(rec.k :: OutputForm, le)
    le := cons(rec.c :: OutputForm * rec.k :: OutputForm, le)

```

```
reduce("+",le)
```

— FM.dotabb —

```
"FM" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FM"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FM" -> "FLAGG"
```

domain FM1 FreeModule1

— FreeModule1.input —

```
)set break resume
)sys rm -f FreeModule1.output
)spool FreeModule1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FreeModule1
--R
--R FreeModule1(R: Ring,S: OrderedSet) is a domain constructor
--R Abbreviation for FreeModule1 is FM1
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FM1
--R
--R----- Operations -----
--R ?? : (S,R) -> %           ?? : (R,S) -> %
--R ?? : (% ,R) -> %         ?? : (R,% ) -> %
--R ?? : (Integer,% ) -> %   ?? : (PositiveInteger,% ) -> %
--R ?? : (% ,% ) -> %       ?-? : (% ,% ) -> %
--R -? : % -> %             ?=? : (% ,% ) -> Boolean
--R 0 : () -> %             coefficient : (% ,S) -> R
--R coefficients : % -> List(R)  coerce : S -> %
--R coerce : % -> OutputForm    hash : % -> SingleInteger
--R latex : % -> String        leadingCoefficient : % -> R
--R leadingMonomial : % -> S    map : ((R -> R),%) -> %
--R monom : (S,R) -> %         monomial? : % -> Boolean
--R monomials : % -> List(% )  reductum : % -> %
```

```

--R retract : % -> S                                sample : () -> %
--R zero? : % -> Boolean                             ?~=? : (%,% ) -> Boolean
--R ??? : (NonNegativeInteger,% ) -> %
--R leadingTerm : % -> Record(k: S,c: R)
--R listOfTerms : % -> List(Record(k: S,c: R))
--R numberOfMonomials : % -> NonNegativeInteger
--R retractIfCan : % -> Union(S,"failed")
--R subtractIfCan : (%,% ) -> Union(%,"failed")
--R
--E 1

)spool
)lisp (bye)

```

— FreeModule1.help —

=====

FreeModule1 examples

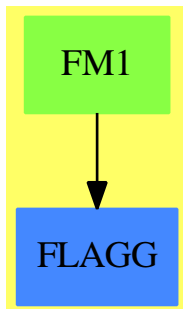
=====

This domain implements linear combinations of elements from the domain S with coefficients in the domain R where S is an ordered set and R is a ring (which may be non-commutative). This domain is used by domains of non-commutative algebra such as: XDistributedPolynomial, XRecursivePolynomial.

See Also:

- o)show FreeModule1

FreeModule1 (FM1)



Exports:

0	coefficient	coefficients	coerce	hash
latex	leadingCoefficient	leadingMonomial	leadingTerm	listOfTerms
map	monom	monomial?	monomials	numberOfMonomials
reductum	retract	retractIfCan	sample	subtractIfCan
zero?	?^=?	?*?	?+?	?-?
-?	?=?			

— domain FM1 FreeModule1 —

```

)abbrev domain FM1 FreeModule1
++ Author: Michel Petitot petitot@lifl.fr
++ Date Created: 91
++ Date Last Updated: 7 Juillet 92
++ Fix History: compilation v 2.1 le 13 dec 98
++ Description:
++ This domain implements linear combinations
++ of elements from the domain \spad{S} with coefficients
++ in the domain \spad{R} where \spad{S} is an ordered set
++ and \spad{R} is a ring (which may be non-commutative).
++ This domain is used by domains of non-commutative algebra such as:
++ XDistributedPolynomial, XRecursivePolynomial.

```

```

FreeModule1(R:Ring,S:OrderedSet): FMcat == FMdef where
  EX ==> OutputForm
  TERM ==> Record(k:S,c:R)

```

```

FMcat == FreeModuleCat(R,S) with
  "*" : (S,R) -> %
  ++ \spad{s*r} returns the product \spad{r*s}
  ++ used by \spadtype{XRecursivePolynomial}

```

```

FMdef == FreeModule(R,S) add
  -- representation
  Rep := List TERM

```

```

-- declarations
  lt: List TERM
  x : %
  r : R
  s : S

```

```

-- define
  numberOfMonomials p ==
    # (p::Rep)

```

```

  listOfTerms(x) == x::List TERM

```

```

  leadingTerm x == x.first
  leadingMonomial x == x.first.k

```

```

coefficients x == [t.c for t in x]
monomials x == [ monom (t.k, t.c) for t in x]

retractIfCan x ==
  numberOfMonomials(x) ^= 1 => "failed"
  x.first.c = 1 => x.first.k
  "failed"

coerce(s:S):% == [[s,1$R]]
retract x ==
  (rr := retractIfCan x) case "failed" => error "FM1.retract impossible"
  rr :: S

if R has noZeroDivisors then
  r * x ==
    r = 0 => 0
    [[u.k,r * u.c]$TERM for u in x]
  x * r ==
    r = 0 => 0
    [[u.k,u.c * r]$TERM for u in x]
else
  r * x ==
    r = 0 => 0
    [[u.k,a] for u in x | not (a:=r*u.c)= 0$R]
  x * r ==
    r = 0 => 0
    [[u.k,a] for u in x | not (a:=u.c*r)= 0$R]

r * s ==
  r = 0 => 0
  [[s,r]$TERM]

s * r ==
  r = 0 => 0
  [[s,r]$TERM]

monom(b,r):% == [[b,r]$TERM]

outTerm(r:R, s:S):EX ==
  r=1 => s::EX
  r::EX * s::EX

coerce(a:%):EX ==
  empty? a => (0$R)::EX
  reduce( _, reverse_! [outTerm(t.c, t.k) for t in a])$List(EX)

coefficient(x,s) ==
  null x => 0$R
  x.first.k > s => coefficient(rest x,s)
  x.first.k = s => x.first.c

```

O\$R

— FM1.dotabb —

```
"FM1" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FM1"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FM1" -> "FLAGG"
```

domain FMONOID FreeMonoid

— FreeMonoid.input —

```
)set break resume
)sys rm -f FreeMonoid.output
)spool FreeMonoid.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FreeMonoid
--R
--R FreeMonoid(S: SetCategory) is a domain constructor
--R Abbreviation for FreeMonoid is FMONOID
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FMONOID
--R
--R----- Operations -----
--R ?? : (% , S) -> %
--R ?? : (% , %) -> %
--R ?? : (% , %) -> Boolean
--R ?? : (% , PositiveInteger) -> %
--R coerce : % -> OutputForm
--R hclf : (% , %) -> %
--R latex : % -> String
--R nthFactor : (% , Integer) -> S
--R recip : % -> Union(% , "failed")
--R sample : () -> %
--R ~=? : (% , %) -> Boolean
--R ??? : (S , NonNegativeInteger) -> %
--R ?? : (S , %) -> %
--R ??? : (% , PositiveInteger) -> %
--R 1 : () -> %
--R coerce : S -> %
--R hash : % -> SingleInteger
--R hcrf : (% , %) -> %
--R mapGen : ((S -> S) , %) -> %
--R one? : % -> Boolean
--R retract : % -> S
--R size : % -> NonNegativeInteger
```

```

--R ??? : (% , NonNegativeInteger) -> %
--R ?<? : (% , %) -> Boolean if S has ORDSET
--R ?<=? : (% , %) -> Boolean if S has ORDSET
--R ?>? : (% , %) -> Boolean if S has ORDSET
--R ?>=? : (% , %) -> Boolean if S has ORDSET
--R ?^? : (% , NonNegativeInteger) -> %
--R divide : (% , %) -> Union(Record(lm: % , rm: %), "failed")
--R factors : % -> List(Record(gen: S , exp: NonNegativeInteger))
--R lquo : (% , %) -> Union(% , "failed")
--R mapExpon : ((NonNegativeInteger -> NonNegativeInteger), %) -> %
--R max : (% , %) -> % if S has ORDSET
--R min : (% , %) -> % if S has ORDSET
--R nthExpon : (% , Integer) -> NonNegativeInteger
--R overlap : (% , %) -> Record(lm: % , mm: % , rm: %)
--R retractIfCan : % -> Union(S , "failed")
--R rquo : (% , %) -> Union(% , "failed")
--R
--E 1

)spool
)lisp (bye)

```

— FreeMonoid.help —

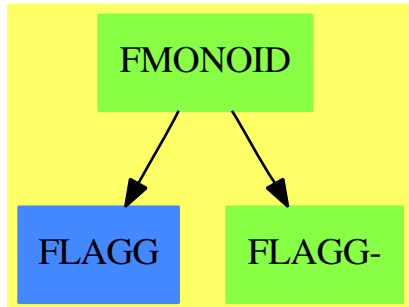
```

=====
FreeMonoid examples
=====

```

Free monoid on any set of generators. The free monoid on a set S is the monoid of finite products of the form $\text{reduce}(*, [s_i ** n_i])$ where the s_i 's are in S , and the n_i 's are nonnegative integers. The multiplication is not commutative.

See Also:
o)show FreeMonoid

FreeMonoid (FMONOID)**See**

- ⇒ “ListMonoidOps” (LMOPS) [13](#) on page [1509](#)
- ⇒ “FreeGroup” (FGROUP) [7](#) on page [1010](#)
- ⇒ “InnerFreeAbelianMonoid” (IFAMON) [10](#) on page [1286](#)
- ⇒ “FreeAbelianMonoid” (FAMONOID) [7](#) on page [1007](#)
- ⇒ “FreeAbelianGroup” (FAGROUP) [7](#) on page [1004](#)

Exports:

1	coerce	divide	factors	hash
hclf	hcrf	latex	lquo	mapExpon
mapGen	max	min	nthExpon	nthFactor
one?	overlap	recip	rquo	retract
retractIfCan	sample	size	?~=?	?**?
?<?	?<=?	?>?	?>=?	?^?
?*?	?=?			

— domain FMONOID FreeMonoid —

```

)abbrev domain FMONOID FreeMonoid
++ Author: Stephen M. Watt
++ Date Last Updated: 6 June 1991
++ Description:
++ Free monoid on any set of generators
++ The free monoid on a set S is the monoid of finite products of
++ the form \spad{reduce(*,[si ** ni])} where the si's are in S, and the ni's
++ are nonnegative integers. The multiplication is not commutative.

```

```

FreeMonoid(S: SetCategory): FMcategory == FMdefinition where
  NNI ==> NonNegativeInteger
  REC ==> Record(gen: S, exp: NonNegativeInteger)
  Ex ==> OutputForm

```

```

FMcategory ==> Join(Monoid, RetractableTo S) with
  "(": (S, $) -> $
  ++ s * x returns the product of x by s on the left.

```



```

"*":      ($, S) -> $
  ++ x * s returns the product of x by s on the right.
"***":    (S, NonNegativeInteger) -> $
  ++ s ** n returns the product of s by itself n times.
hclf:     ($, $) -> $
  ++ hclf(x, y) returns the highest common left factor of x and y,
  ++ i.e. the largest d such that \spad{x = d a} and \spad{y = d b}.
hcrf:     ($, $) -> $
  ++ hcrf(x, y) returns the highest common right factor of x and y,
  ++ i.e. the largest d such that \spad{x = a d} and \spad{y = b d}.
lquo:     ($, $) -> Union($, "failed")
  ++ lquo(x, y) returns the exact left quotient of x by y i.e.
  ++ q such that \spad{x = y * q},
  ++ "failed" if x is not of the form \spad{y * q}.
rquo:     ($, $) -> Union($, "failed")
  ++ rquo(x, y) returns the exact right quotient of x by y i.e.
  ++ q such that \spad{x = q * y},
  ++ "failed" if x is not of the form \spad{q * y}.
divide:   ($, $) -> Union(Record(lm: $, rm: $), "failed")
  ++ divide(x, y) returns the left and right exact quotients of
  ++ x by y, i.e. \spad{[l, r]} such that \spad{x = l * y * r},
  ++ "failed" if x is not of the form \spad{l * y * r}.
overlap:  ($, $) -> Record(lm: $, mm: $, rm: $)
  ++ overlap(x, y) returns \spad{[l, m, r]} such that
  ++ \spad{x = l * m}, \spad{y = m * r} and l and r have no overlap,
  ++ i.e. \spad{overlap(l, r) = [l, 1, r]}.
size      :      $ -> NNI
  ++ size(x) returns the number of monomials in x.
factors    :      $ -> List Record(gen: S, exp: NonNegativeInteger)
  ++ factors(a1\^e1,...,an\^en) returns \spad{[[a1, e1],...,[an, en]]}.
nthExpon   : ($, Integer) -> NonNegativeInteger
  ++ nthExpon(x, n) returns the exponent of the nth monomial of x.
nthFactor  : ($, Integer) -> S
  ++ nthFactor(x, n) returns the factor of the nth monomial of x.
mapExpon   : (NNI -> NNI, $) -> $
  ++ mapExpon(f, a1\^e1 ... an\^en) returns \spad{a1\^f(e1) ... an\^f(en)}.
mapGen     : (S -> S, $) -> $
  ++ mapGen(f, a1\^e1 ... an\^en) returns \spad{f(a1)\^e1 ... f(an)\^en}.
if S has OrderedSet then OrderedSet

FMdefinition ==> ListMonoidOps(S, NonNegativeInteger, 1) add
Rep := ListMonoidOps(S, NonNegativeInteger, 1)

1          == makeUnit()
one? f     == empty? listOfMonoms f
coerce(f:$): Ex == outputForm(f, "*", "***", 1)
hcrf(f, g) == reverse_! hclf(reverse f, reverse g)
f:$ * s:$  == rightMult(f, s)
s:$ * f:$  == leftMult(s, f)
factors f  == copy listOfMonoms f

```

```

mapExpon(f, x) == mapExpon(f, x)$Rep
mapGen(f, x)   == mapGen(f, x)$Rep
s:S ** n:NonNegativeInteger == makeTerm(s, n)

f:$ * g:$ ==
--   one? f => g
   (f = 1) => g
--   one? g => f
   (g = 1) => f
   lg := listOfMonoms g
   ls := last(lf := listOfMonoms f)
   ls.gen = lg.first.gen =>
       setlast_!(h := copy lf, [lg.first.gen, lg.first.exp+ls.exp])
       makeMulti concat(h, rest lg)
   makeMulti concat(lf, lg)

overlap(la, ar) ==
--   one? la or one? ar => [la, 1, ar]
   (la = 1) or (ar = 1) => [la, 1, ar]
   lla := la0 := listOfMonoms la
   lar := listOfMonoms ar
   l:List(REC) := empty()
   while not empty? lla repeat
       if lla.first.gen = lar.first.gen then
           if lla.first.exp < lar.first.exp and empty? rest lla then
               return [makeMulti 1,
                       makeTerm(lla.first.gen, lla.first.exp),
                       makeMulti concat([lar.first.gen,
                                           (lar.first.exp - lla.first.exp)::NNI],
                                           rest lar)]

           if lla.first.exp >= lar.first.exp then
               if (ru:= lquo(makeMulti rest lar,
                             makeMulti rest lla)) case $ then
                   if lla.first.exp > lar.first.exp then
                       l := concat_!(l, [lla.first.gen,
                                           (lla.first.exp - lar.first.exp)::NNI])
                       m := concat([lla.first.gen, lar.first.exp],
                                   rest lla)
                   else m := lla
               return [makeMulti l, makeMulti m, ru::$]
           l := concat_!(l, lla.first)
           lla := rest lla
   [makeMulti la0, 1, makeMulti lar]

divide(lar, a) ==
--   one? a => [lar, 1]
   (a = 1) => [lar, 1]
   Na : Integer := #(la := listOfMonoms a)
   Nlar : Integer := #(llar := listOfMonoms lar)
   l:List(REC) := empty()

```

```

while Na <= Nlar repeat
  if llar.first.gen = la.first.gen and
    llar.first.exp >= la.first.exp then
    -- Can match a portion of this lar factor.
    -- Now match tail.
    (q:=lquo(makeMulti rest llar,makeMulti rest la))case $ =>
      if llar.first.exp > la.first.exp then
        l := concat_!(l, [la.first.gen,
                          (llar.first.exp - la.first.exp)::NNI])
        return [makeMulti l, q::$]
      l := concat_!(l, first llar)
      llar := rest llar
      Nlar := Nlar - 1
"failed"

hclf(f, g) ==
  h:List(REC) := empty()
  for f0 in listOfMonoms f for g0 in listOfMonoms g repeat
    f0.gen ^= g0.gen => return makeMulti h
    h := concat_!(h, [f0.gen, min(f0.exp, g0.exp)])
    f0.exp ^= g0.exp => return makeMulti h
  makeMulti h

lquo(aq, a) ==
  size a > #(laq := copy listOfMonoms aq) => "failed"
  for a0 in listOfMonoms a repeat
    a0.gen ^= laq.first.gen or a0.exp > laq.first.exp =>
      return "failed"
    if a0.exp = laq.first.exp then laq := rest laq
    else setfirst_!(laq, [laq.first.gen,
                          (laq.first.exp - a0.exp)::NNI])
  makeMulti laq

rquo(qa, a) ==
  (u := lquo(reverse qa, reverse a)) case "failed" => "failed"
  reverse_!(u::$)

if S has OrderedSet then
  a < b ==
    la := listOfMonoms a
    lb := listOfMonoms b
    na: Integer := #la
    nb: Integer := #lb
    while na > 0 and nb > 0 repeat
      la.first.gen > lb.first.gen => return false
      la.first.gen < lb.first.gen => return true
      if la.first.exp = lb.first.exp then
        la:=rest la
        lb:=rest lb
        na:=na - 1

```

```

      nb:=nb - 1
    else if la.first.exp > lb.first.exp then
      la:=concat([la.first.gen,
                  (la.first.exp - lb.first.exp)::NNI], rest lb)
      lb:=rest lb
      nb:=nb - 1
    else
      lb:=concat([lb.first.gen,
                  (lb.first.exp-la.first.exp)::NNI], rest la)
      la:=rest la
      na:=na-1
    empty? la and not empty? lb

```

— FMONOID.dotabb —

```

"FMONOID" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FMONOID"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"FMONOID" -> "FLAGG-"
"FMONOID" -> "FLAGG"

```

domain FNLA FreeNilpotentLie

— FreeNilpotentLie.input —

```

)set break resume
)sys rm -f FreeNilpotentLie.output
)spool FreeNilpotentLie.output
)set message test on
)set message auto off
)clear all

```

```
--S 1 of 1
```

```
)show FreeNilpotentLie
```

```
--R FreeNilpotentLie(n: NonNegativeInteger,class: NonNegativeInteger,R: CommutativeRing) is a domain const
```

```
--R Abbreviation for FreeNilpotentLie is FNLA
```

```
--R This constructor is exposed in this frame.
```

```
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FNLA
```

```
--R
```

```
--R----- Operations -----
```

```

--R ?? : (R,%) -> %
--R ?? : (%,%) -> %
--R ?? : (PositiveInteger,%) -> %
--R +? : (%,%) -> %
--R -? : % -> %
--R 0 : () -> %
--R associator : (%,%,%) -> %
--R commutator : (%,%) -> %
--R hash : % -> SingleInteger
--R sample : () -> %
--R zero? : % -> Boolean
--R ?? : (NonNegativeInteger,%) -> %
--R dimension : () -> NonNegativeInteger
--R generator : NonNegativeInteger -> %
--R leftPower : (%,PositiveInteger) -> %
--R plenaryPower : (%,PositiveInteger) -> %
--R rightPower : (%,PositiveInteger) -> %
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)

```

— FreeNilpotentLie.help —

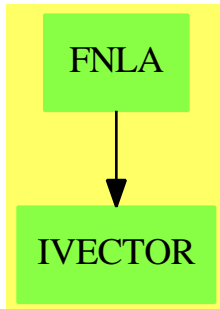
```

=====
FreeNilpotentLie examples
=====

```

Generate the Free Lie Algebra over a ring R with identity;
A P. Hall basis is generated by a package call to HallBasis.

See Also:
o)show FreeNilpotentLie

FreeNilpotentLie (FNLA)**See**⇒ “OrdSetInts” (OSI) [16](#) on page [1864](#)⇒ “Commutator” (COMM) [4](#) on page [411](#)**Exports:**

0	antiCommutator	associator	coerce	commutator
deepExpand	dimension	generator	hash	latex
leftPower	plenaryPower	rightPower	sample	shallowExpand
subtractIfCan	zero?	?~=?	?*?	?**?
?+?	?-?	-?	?=?	

— domain FNLA FreeNilpotentLie —

```

)abbrev domain FNLA FreeNilpotentLie
++ Author: Larry Lambe
++ Date Created: July 1988
++ Date Last Updated: March 13 1991
++ Description:
++ Generate the Free Lie Algebra over a ring R with identity;
++ A P. Hall basis is generated by a package call to HallBasis.

```

```

FreeNilpotentLie(n:NNI,class:NNI,R: CommutativeRing): Export == Implement where

```

```

  B ==> Boolean
  Com ==> Commutator
  HB ==> HallBasis
  I ==> Integer
  NNI ==> NonNegativeInteger
  O ==> OutputForm
  OSI ==> OrdSetInts
  FM ==> FreeModule(R,OSI)
  VI ==> Vector Integer
  VLI ==> Vector List Integer
  lC ==> leadingCoefficient
  lS ==> leadingSupport

```

```

Export ==> NonAssociativeAlgebra(R) with

```

```

dimension : () -> NNI
++ dimension() is the rank of this Lie algebra
deepExpand : % -> 0
++ deepExpand(x) is not documented
shallowExpand : % -> 0
++ shallowExpand(x) is not documented
generator : NNI -> %
++ generator(i) is the ith Hall Basis element

Implement ==> FM add
Rep := FM
f,g : %

coms:VLI
coms := generate(n,class)$HB

dimension == #coms

have : (I,I) -> %
-- have(left,right) is a lookup function for basic commutators
-- already generated; if the nth basic commutator is
-- [left,wt,right], then have(left,right) = n
have(i,j) ==
wt:I := coms(i).2 + coms(j).2
wt > class => 0
lo:I := 1
hi:I := dimension
while hi-lo > 1 repeat
mid:I := (hi+lo) quo 2
if coms(mid).2 < wt then lo := mid else hi := mid
while coms(hi).1 < i repeat hi := hi + 1
while coms(hi).3 < j repeat hi := hi + 1
monomial(1,hi::OSI)$FM

generator(i) ==
i > dimension => 0$Rep
monomial(1,i::OSI)$FM

putIn : I -> %
putIn(i) ==
monomial(1$R,i::OSI)$FM

brkt : (I,%) -> %
brkt(k,f) ==
f = 0 => 0
dg:I := value 1S f
reductum(f) = 0 =>
k = dg => 0
k > dg => -1C(f)*brkt(dg, putIn(k))
inHallBasis?(n,k,dg,coms(dg).1) => 1C(f)*have(k, dg)

```

```

    lC(f)*( brkt(coms(dg).1, _
      brkt(k,putIn coms(dg).3)) - brkt(coms(dg).3, _
      brkt(k,putIn coms(dg).1) ))
    brkt(k,monomial(lC f,lS f)$FM)+brkt(k,reductum f)

f*g ==
  reductum(f) = 0 =>
    lC(f)*brkt(value(lS f),g)
  monomial(lC f,lS f)$FM*g + reductum(f)*g

Fac : I -> Com
  -- an auxilliary function used for output of Free Lie algebra
  -- elements (see expand)
Fac(m) ==
  coms(m).1 = 0 => mkcomm(m)$Com
  mkcomm(Fac coms(m).1, Fac coms(m).3)

shallowE : (R,OSI) -> 0
shallowE(r,s) ==
  k := value s
  r = 1 =>
    k <= n => s::0
    mkcomm(mkcomm(coms(k).1)$Com,mkcomm(coms(k).3)$Com)$Com::0
  k <= n => r::0 * s::0
  r::0 * mkcomm(mkcomm(coms(k).1)$Com,mkcomm(coms(k).3)$Com)$Com::0

shallowExpand(f) ==
  f = 0 => 0::0
  reductum(f) = 0 => shallowE(lC f,lS f)
  shallowE(lC f,lS f) + shallowExpand(reductum f)

deepExpand(f) ==
  f = 0 => 0::0
  reductum(f) = 0 =>
    lC(f)=1 => Fac(value(lS f))::0
    lC(f)::0 * Fac(value(lS f))::0
  lC(f)=1 => Fac(value(lS f))::0 + deepExpand(reductum f)
  lC(f)::0 * Fac(value(lS f))::0 + deepExpand(reductum f)

```

— FNLA.dotabb —

"FNLA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FNLA"]
 "IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
 "FNLA" -> "IVECTOR"

domain FPARFRAC FullPartialFractionExpansion

— FullPartialFractionExpansion.input —

```

)set break resume
)sys rm -f FullPartialFractionExpansion.output
)spool FullPartialFractionExpansion.output
)set message test on
)set message auto off
)clear all
--S 1 of 16
Fx := FRAC UP(x, FRAC INT)
--R
--R
--R (1) Fraction(UnivariatePolynomial(x,Fraction(Integer)))
--R
--R                                          Type: Domain
--E 1

--S 2 of 16
f : Fx := 36 / (x**5-2*x**4-2*x**3+4*x**2+x-2)
--R
--R
--R
--R          36
--R (2)  -----
--R      5      4      3      2
--R      x  - 2x  - 2x  + 4x  + x - 2
--R
--R                                          Type: Fraction(UnivariatePolynomial(x,Fraction(Integer)))
--E 2

--S 3 of 16
g := fullPartialFraction f
--R
--R
--R
--R          4      4      --+      - 3%A - 6
--R (3)  ----- - ----- + > -----
--R          4      4      --+      2
--R          x  - 2  x  + 1  --+      (x - %A)
--R
--R          2
--R          %A  - 1= 0
--R
--R                                          Type: FullPartialFractionExpansion(Fraction(Integer),UnivariatePolynomial(x,Fraction(Integer)))
--E 3

--S 4 of 16
g :: Fx
--R
--R
--R
--R          36
--R (4)  -----
--R      5      4      3      2

```

```

--R      x  - 2x  - 2x  + 4x  + x - 2
--R      Type: Fraction(UnivariatePolynomial(x,Fraction(Integer)))
--E 4

--S 5 of 16
g5 := D(g, 5)
--R
--R
--R      480      480      --+      2160%A + 4320
--R      (5)  - ---- + ---- + > ----
--R            6      6      --+      7
--R      (x - 2)  (x + 1)  2      (x - %A)
--R      %A - 1 = 0
--RType: FullPartialFractionExpansion(Fraction(Integer),UnivariatePolynomial(x,Fraction(Integer)))
--E 5

--S 6 of 16
f5 := D(f, 5)
--R
--R
--R      (6)
--R      10      9      8      7      6
--R      - 544320x  + 4354560x  - 14696640x  + 28615680x  - 40085280x
--R      +
--R      5      4      3      2
--R      46656000x  - 39411360x  + 18247680x  - 5870880x  + 3317760x + 246240
--R      /
--R      20      19      18      17      16      15      14      13
--R      x  - 12x  + 53x  - 76x  - 159x  + 676x  - 391x  - 1596x
--R      +
--R      12      11      10      9      8      7      6      5
--R      2527x  + 1148x  - 4977x  + 1372x  + 4907x  - 3444x  - 2381x  + 2924x
--R      +
--R      4      3      2
--R      276x  - 1184x  + 208x  + 192x - 64
--R      Type: Fraction(UnivariatePolynomial(x,Fraction(Integer)))
--E 6

--S 7 of 16
g5:=Fx - f5
--R
--R
--R      (7)  0
--R      Type: Fraction(UnivariatePolynomial(x,Fraction(Integer)))
--E 7

--S 8 of 16
f : Fx := (x**5 * (x-1)) / ((x**2 + x + 1)**2 * (x-2)**3)
--R
--R

```

```

--R          6      5
--R          x  - x
--R (8)  -----
--R          7      6      5      3      2
--R          x  - 4x  + 3x  + 9x  - 6x  - 4x  - 8
--R                                     Type: Fraction(UnivariatePolynomial(x,Fraction(Integer)))
--E 8

```

```

--S 9 of 16

```

```

g := fullPartialFraction f

```

```

--R
--R
--R (9)
--R          1952      464      32      179      135
--R          ----      ---      --      - ---- %A + ----
--R          2401      343      49      2401      2401
--R          ----- + ----- + ----- + ----- > -----
--R          x  - 2      2      3      2      x  - %A
--R          (x  - 2)      (x  - 2)      2
--R                                     %A  + %A  + 1= 0
--R +
--R          37      20
--R          ---- %A + ----
--R          1029      1029
--R          > -----
--R          2
--R          (x  - %A)
--R          2
--R          %A  + %A  + 1= 0
--RType: FullPartialFractionExpansion(Fraction(Integer),UnivariatePolynomial(x,Fraction(Integer)))
--E 9

```

```

--S 10 of 16

```

```

g :: Fx - f

```

```

--R
--R
--R (10)  0
--R                                     Type: Fraction(UnivariatePolynomial(x,Fraction(Integer)))
--E 10

```

```

--S 11 of 16

```

```

f : Fx := (2*x**7-7*x**5+26*x**3+8*x) / (x**8-5*x**6+6*x**4+4*x**2-8)

```

```

--R
--R
--R          7      5      3
--R          2x  - 7x  + 26x  + 8x
--R (11)  -----
--R          8      6      4      2
--R          x  - 5x  + 6x  + 4x  - 8
--R                                     Type: Fraction(UnivariatePolynomial(x,Fraction(Integer)))
--E 11

```

```

--S 12 of 16
g := fullPartialFraction f
--R
--R
--R
--R      1
--R      -
--R      2
--R      1
--R      -
--R      2
--R      (12)  > ----- + > ----- + > -----
--R      2      2      2      3      2
--R      x - %A      (x - %A)      x - %A
--R      %A - 2 = 0      %A - 2 = 0      %A + 1 = 0
--RType: FullPartialFractionExpansion(Fraction(Integer),UnivariatePolynomial(x,Fraction(Integer)))
--E 12

--S 13 of 16
g :: Fx - f
--R
--R
--R      (13)  0
--R
--R      Type: Fraction(UnivariatePolynomial(x,Fraction(Integer)))
--E 13

--S 14 of 16
f:Fx := x**3 / (x**21 + 2*x**20 + 4*x**19 + 7*x**18 + 10*x**17 + 17*x**16 + 22*x**15 + 30*x**14 + 36*x**13 + 40*x**12 + 36*x**11 + 22*x**10 + 10*x**9 + 7*x**8 + 4*x**7 + 2*x**6 + 3*x**5 + 2*x**4 + 3*x**3 + 2*x**2 + 3*x + 2)
--R
--R
--R      (14)
--R      3
--R      x
--R      /
--R      21      20      19      18      17      16      15      14      13      12
--R      x  + 2x  + 4x  + 7x  + 10x  + 17x  + 22x  + 30x  + 36x  + 40x
--R      +
--R      11      10      9      8      7      6      5      4      3      2
--R      47x  + 46x  + 49x  + 43x  + 38x  + 32x  + 23x  + 19x  + 10x  + 7x  + 2x
--R      +
--R      1
--R
--R      Type: Fraction(UnivariatePolynomial(x,Fraction(Integer)))
--E 14

--S 15 of 16
g := fullPartialFraction f
--R
--R
--R      (15)
--R      1
--R      - %A
--R      2
--R      1
--R      - %A - --
--R      9      27
--R      > ----- + > -----

```

```

--R      +-+      x - %A      +-+      x - %A
--R      2      2
--R      %A  + 1= 0      %A  + %A + 1= 0
--R  +
--R      1      1
--R      -- %A - --
--R      +-+      27      27
--R      >      -----
--R      +-+      2
--R      2      (x - %A)
--R      %A  + %A + 1= 0
--R  +
--R  SIGMA
--R      5      2
--R      %A  + %A  + 1= 0
--R  ,
--R      96556567040  4  420961732891  3  59101056149  2
--R      - ----- %A  + ----- %A  - ----- %A
--R      912390759099  912390759099  912390759099
--R  +
--R      373545875923  529673492498
--R      - ----- %A  + -----
--R      912390759099  912390759099
--R  /
--R      x - %A
--R  +
--R  SIGMA
--R      5      2
--R      %A  + %A  + 1= 0
--R  ,
--R      5580868  4  2024443  3  4321919  2  84614  5070620
--R      - ----- %A  - ----- %A  + ----- %A  - ----- %A  - -----
--R      94070601  94070601  94070601  1542141  94070601
--R      -----
--R      2
--R      (x - %A)
--R  +
--R  SIGMA
--R      5      2
--R      %A  + %A  + 1= 0
--R  ,
--R      1610957  4  2763014  3  2016775  2  266953  4529359
--R      ----- %A  + ----- %A  - ----- %A  + ----- %A  + -----
--R      94070601  94070601  94070601  94070601  94070601
--R      -----
--R      3
--R      (x - %A)
--RType: FullPartialFractionExpansion(Fraction(Integer),UnivariatePolynomial(x,Fraction(Integer))
--E 15

```

```

--S 16 of 16
g :: Fx - f
--R
--R
--R (16)  0
--R
--R                                     Type: Fraction(UnivariatePolynomial(x,Fraction(Integer)))
--E 16
)spool
)lisp (bye)

```

— FullPartialFractionExpansion.help —

```

=====
FullPartialFractionExpansion expansion
=====

```

The domain FullPartialFractionExpansion implements factor-free conversion of quotients to full partial fractions.

Our examples will all involve quotients of univariate polynomials with rational number coefficients.

```

Fx := FRAC UP(x, FRAC INT)
      Fraction UnivariatePolynomial(x,Fraction Integer)
      Type: Domain

```

Here is a simple-looking rational function.

```

f : Fx := 36 / (x**5-2*x**4-2*x**3+4*x**2+x-2)
          36
-----
      5      4      3      2
      x  - 2x  - 2x  + 4x  + x - 2
          Type: Fraction UnivariatePolynomial(x,Fraction Integer)

```

We use fullPartialFraction to convert it to an object of type FullPartialFractionExpansion.

```

g := fullPartialFraction f
      4      4      --+      - 3%A - 6
      ---- - ---- + > -----
      x - 2   x + 1   --+      2
                      2      (x - %A)
                      %A  - 1= 0
Type: FullPartialFractionExpansion(Fraction Integer,
                                   UnivariatePolynomial(x,Fraction Integer))

```

Use a coercion to change it back into a quotient.

```
g :: Fx
      36
-----
      5      4      3      2
x  - 2x  - 2x  + 4x  + x  - 2
Type: Fraction UnivariatePolynomial(x,Fraction Integer)
```

Full partial fractions differentiate faster than rational functions.

```
g5 := D(g, 5)
      480      480      --+      2160%A + 4320
- ---- + ---- + > ----
      6      6      --+      7
(x - 2) (x + 1) 2 (x - %A)
      %A - 1 = 0
Type: FullPartialFractionExpansion(Fraction Integer,
UnivariatePolynomial(x,Fraction Integer))
```

```
f5 := D(f, 5)
      10      9      8      7      6
- 544320x  + 4354560x  - 14696640x  + 28615680x  - 40085280x
+
      5      4      3      2
46656000x  - 39411360x  + 18247680x  - 5870880x  + 3317760x + 246240
/
      20      19      18      17      16      15      14      13
x  - 12x  + 53x  - 76x  - 159x  + 676x  - 391x  - 1596x
+
      12      11      10      9      8      7      6      5
2527x  + 1148x  - 4977x  + 1372x  + 4907x  - 3444x  - 2381x  + 2924x
+
      4      3      2
276x  - 1184x  + 208x  + 192x - 64
Type: Fraction UnivariatePolynomial(x,Fraction Integer)
```

We can check that the two forms represent the same function.

```
g5::Fx - f5
0
Type: Fraction UnivariatePolynomial(x,Fraction Integer)
```

Here are some examples that are more complicated.

```
f : Fx := (x**5 * (x-1)) / ((x**2 + x + 1)**2 * (x-2)**3)
      6      5
x  - x
-----
      7      6      5      3      2
```

```

x  - 4x  + 3x  + 9x  - 6x  - 4x - 8
Type: Fraction UnivariatePolynomial(x,Fraction Integer)

g := fullPartialFraction f
      1952      464      32      179      135
      ----      ---      --      - ---- %A + ----
      2401      343      49      2401      2401
      ----- + ----- + ----- + -----
      x - 2      (x - 2)  (x - 2)  (x - 2)  (x - 2)
      %A  + %A + 1= 0
+
      37      20
      ---- %A + ----
      1029      1029
      -----
      (x - %A)
      %A  + %A + 1= 0
Type: FullPartialFractionExpansion(Fraction Integer,
UnivariatePolynomial(x,Fraction Integer))

g :: Fx - f
0
Type: Fraction UnivariatePolynomial(x,Fraction Integer)

f : Fx := (2*x**7-7*x**5+26*x**3+8*x) / (x**8-5*x**6+6*x**4+4*x**2-8)
      7      5      3
      2x  - 7x  + 26x  + 8x
      -----
      8      6      4      2
      x  - 5x  + 6x  + 4x  - 8
Type: Fraction UnivariatePolynomial(x,Fraction Integer)

g := fullPartialFraction f
      1      1
      -      -
      2      2
      > ----- + > ----- + > -----
      x - %A      (x - %A)  x - %A
      2      2      2
      %A  - 2= 0      %A  - 2= 0      %A  + 1= 0
Type: FullPartialFractionExpansion(Fraction Integer,
UnivariatePolynomial(x,Fraction Integer))

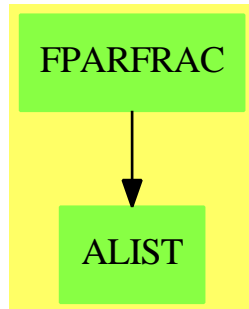
g :: Fx - f
0
Type: Fraction UnivariatePolynomial(x,Fraction Integer)

f:Fx := x**3 / (x**21 + 2*x**20 + 4*x**19 + 7*x**18 + 10*x**17 + 17*x**16 + 22*x**15 + 30*x**14 + 36*x**13 + 35*x**12 + 28*x**11 + 21*x**10 + 14*x**9 + 9*x**8 + 6*x**7 + 4*x**6 + 3*x**5 + 2*x**4 + x**3 + x**2 + x + 1)

```


[illegible]

FullPartialFractionExpansion (FPARFRAC)

**Exports:**

coerce	construct	convert	D	differentiate
hash	latex	polyPart	fracPart	fullPartialFraction
?~=?	?+?	?=?		

— domain **FPARFRAC FullPartialFractionExpansion** —

```

)abbrev domain FPARFRAC FullPartialFractionExpansion
++ Author: Manuel Bronstein
++ Date Created: 9 December 1992
++ Date Last Updated: 6 October 1993
++ References: M.Bronstein & B.Salvy,
++             Full Partial Fraction Decomposition of Rational Functions,
++             in Proceedings of ISSAC'93, Kiev, ACM Press.
++ Description:
++ Full partial fraction expansion of rational functions

FullPartialFractionExpansion(F, UP): Exports == Implementation where
  F  : Join(Field, CharacteristicZero)
  UP : UnivariatePolynomialCategory F

  N  ==> NonNegativeInteger
  Q  ==> Fraction Integer
  O  ==> OutputForm
  RF ==> Fraction UP
  SUP ==> SparseUnivariatePolynomial RF
  REC ==> Record(exponent: N, center: UP, num: UP)
  ODV ==> OrderlyDifferentialVariable Symbol
  ODP ==> OrderlyDifferentialPolynomial UP
  ODF ==> Fraction ODP
  PPF ==> Record(polyPart: UP, fracPart: List REC)

Exports ==> Join(SetCategory, ConvertibleTo RF) with
  "+": (UP, $) -> $
  ++ p + x returns the sum of p and x

```

```

fullPartialFraction: RF -> $
  ++ fullPartialFraction(f) returns \spad{[p, [[j, Dj, Hj]...]]} such that
  ++ \spad{f = p(x) + sum_{[j,Dj,Hj] in l} sum_{Dj(a)=0} Hj(a)/(x - a)\^j}.
polyPart:          $ -> UP
  ++ polyPart(f) returns the polynomial part of f.
fracPart:          $ -> List REC
  ++ fracPart(f) returns the list of summands of the fractional part of f.
construct:          List REC -> $
  ++ construct(l) is the inverse of fracPart.
differentiate:      $ -> $
  ++ differentiate(f) returns the derivative of f.
D:                  $ -> $
  ++ D(f) returns the derivative of f.
differentiate:      ($, N) -> $
  ++ differentiate(f, n) returns the n-th derivative of f.
D: ($, NonNegativeInteger) -> $
  ++ D(f, n) returns the n-th derivative of f.

Implementation ==> add
Rep := FPF

fullParFrac: (UP, UP, UP, N) -> List REC
outputexp : (0, N) -> 0
output    : (N, UP, UP) -> 0
REC2RF    : (UP, UP, N) -> RF
UP2SUP    : UP -> SUP
diffrec   : REC -> REC
FP20      : List REC -> 0

-- create a differential variable
u := new()$Symbol
u0 := makeVariable(u, 0)$ODV
alpha := u::0
x := monomial(1, 1)$UP
xx := x::0
zr := (0$N)::0

construct l == [0, 1]
D r == differentiate r
D(r, n) == differentiate(r,n)
polyPart f == f.polyPart
fracPart f == f.fracPart
p:UP + f:$ == [p + polyPart f, fracPart f]

differentiate f ==
  differentiate(polyPart f) + construct [diffrec rec for rec in fracPart f]

differentiate(r, n) ==
  for i in 1..n repeat r := differentiate r
  r

```

```

-- diffrec(sum_{rec.center(a) = 0} rec.num(a) / (x - a)^e) =
--      sum_{rec.center(a) = 0} -e rec.num(a) / (x - a)^{e+1}
--      where e = rec.exponent
diffrec rec ==
  e := rec.exponent
  [e + 1, rec.center, - e * rec.num]

convert(f:$):RF ==
  ans := polyPart(f)::RF
  for rec in fracPart f repeat
    ans := ans + REC2RF(rec.center, rec.num, rec.exponent)
  ans

UP2SUP p == map((z1:F):RF --> z1::UP::RF, p)_
  $UnivariatePolynomialCategoryFunctions2(F, UP, RF, SUP)

-- returns Trace_k^k(a) (h(a) / (x - a)^n) where d(a) = 0
REC2RF(d, h, n) ==
--      one?(m := degree d) =>
--      ((m := degree d) = 1) =>
--      a := - (leadingCoefficient reductum d) / (leadingCoefficient d)
--      h(a)::UP / (x - a::UP)**n
dd := UP2SUP d
hh := UP2SUP h
aa := monomial(1, 1)$SUP
p := (x::RF::SUP - aa)**n rem dd
rec := extendedEuclidean(p, dd, hh)::Record(coef1:SUP, coef2:SUP)
t := rec.coef1 -- we want Trace_k^k(a)(t) now
ans := coefficient(t, 0)
for i in 1..degree(d)-1 repeat
  t := (t * aa) rem dd
  ans := ans + coefficient(t, i)
ans

fullPartialFraction f ==
  qr := divide( Numer f, d := Denom f)
  qr.quotient + construct concat
    [fullParFrac(qr.remainder, d, rec.factor, rec.exponent::N)
     for rec in factors squareFree Denom f]

fullParFrac(a, d, q, n) ==
  ans:List REC := empty()
  em := e := d quo (q ** n)
  rec := extendedEuclidean(e, q, 1)::Record(coef1:UP,coef2:UP)
  bm := b := rec.coef1 -- b = inverse of e modulo q
  lvar:List(ODV) := [u0]
  um := 1::ODP
  un := (u1 := u0::ODP)**n
  lval:List(UP) := [q1 := q := differentiate(q0 := q)]

```

```

h:ODF := a::ODP / (e * un)
rec := extendedEuclidean(q1, q0, 1)::Record(coef1:UP,coef2:UP)
c := rec.coef1 -- c = inverse of q' modulo q
cm := 1::UP
cn := (c ** n) rem q0
for m in 1..n repeat
  p := retract(em * un * um * h)@ODP
  pp := retract(eval(p, lvar, lval))@UP
  h := inv(m::Q) * differentiate h
  q := differentiate q
  lvar := concat(makeVariable(u, m), lvar)
  lval := concat(inv((m+1)::F) * q, lval)
  qq := q0 quo gcd(pp, q0) -- new center
  if (degree(qq) > 0) then
    ans := concat([(n + 1 - m)::N, qq, (pp * bm * cn * cm) rem qq], ans)
  cm := (c * cm) rem q0 -- cm = c**m modulo q now
  um := u1 * um -- um = u**m now
  em := e * em -- em = e**{m+1} now
  bm := (b * bm) rem q0 -- bm = b**{m+1} modulo q now
ans

coerce(f:$):0 ==
  ans := FP20(l := fracPart f)
  zero?(p := polyPart f) =>
    empty? l => (0$N)::0
  ans
p::0 + ans

FP20 l ==
  empty? l => empty()
  rec := first l
  ans := output(rec.exponent, rec.center, rec.num)
  for rec in rest l repeat
    ans := ans + output(rec.exponent, rec.center, rec.num)
  ans

output(n, d, h) ==
--   one? degree d =>
  (degree d) = 1 =>
    a := - leadingCoefficient(reductum d) / leadingCoefficient(d)
    h(a)::0 / outputexp((x - a::UP)::0, n)
    sum(outputForm(makeSUP h, alpha) / outputexp(xx - alpha, n),
      outputForm(makeSUP d, alpha) = zr)

outputexp(f, n) ==
--   one? n => f
  (n = 1) => f
  f ** (n::0)

```

— FPARFRAC.dotabb —

```
"FPARFRAC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FPARFRAC"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"FPARFRAC" -> "ALIST"
```

domain FUNCTION FunctionCalled

— FunctionCalled.input —

```
)set break resume
)sys rm -f FunctionCalled.output
)spool FunctionCalled.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FunctionCalled
--R
--R FunctionCalled(f: Symbol) is a domain constructor
--R Abbreviation for FunctionCalled is FUNCTION
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FUNCTION
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger        latex : % -> String
--R name : % -> Symbol               ?~=? : (%,% ) -> Boolean
--R
--E 1

)spool
)lisp (bye)
```

— FunctionCalled.help —

=====

FunctionCalled examples

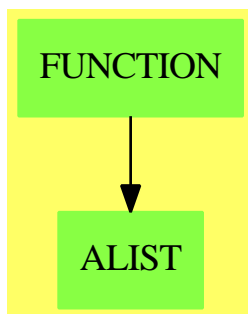
=====

This domain implements named functions

See Also:

o)show FunctionCalled

FunctionCalled (FUNCTION)



Exports:

coerce hash latex name ?=? ?~=?

— domain FUNCTION FunctionCalled —

```

)abbrev domain FUNCTION FunctionCalled
++ Author: Mark Botch
++ Description:
++ This domain implements named functions
  
```

```

FunctionCalled(f:Symbol): SetCategory with
  name: % -> Symbol
  ++ name(x) returns the symbol
== add
  name r == f
  coerce(r:%):OutputForm == f::OutputForm
  x = y == true
  latex(x:%):String == latex f
  
```

— FUNCTION.dotabb —

```
"FUNCTION" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FUNCTION"]  
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]  
"FUNCTION" -> "ALIST"
```

—————

Chapter 8

Chapter G

domain GDMP GeneralDistributedMultivariatePolynomial

— GeneralDistributedMultivariatePolynomial.input —

```
)set break resume
)sys rm -f GeneralDistributedMultivariatePolynomial.output
)spool GeneralDistributedMultivariatePolynomial.output
)set message test on
)set message auto off
)clear all
--S 1 of 10
(d1,d2,d3) : DMP([z,y,x],FRAC INT)
--R
--R
--R                                          Type: Void
--E 1

--S 2 of 10
d1 := -4*z + 4*y**2*x + 16*x**2 + 1
--R
--R
--R          2      2
--R  (2)  - 4z + 4y x + 16x  + 1
--R          Type: DistributedMultivariatePolynomial([z,y,x],Fraction(Integer))
--E 2

--S 3 of 10
d2 := 2*z*y**2 + 4*x + 1
--R
--R
```

```

--R      2
--R (3) 2z y + 4x + 1
--R      Type: DistributedMultivariatePolynomial([z,y,x],Fraction(Integer))
--E 3

```

```

--S 4 of 10
d3 := 2*z*x**2 - 2*y**2 - x
--R
--R
--R      2      2
--R (4) 2z x - 2y - x
--R      Type: DistributedMultivariatePolynomial([z,y,x],Fraction(Integer))
--E 4

```

```

--S 5 of 10
groebner [d1,d2,d3]
--R
--R
--R (5)
--R      1568 6 1264 5 6 4 182 3 2047 2 103 2857
--R [z - ---- x - ---- x + --- x + --- x - ---- x - ---- x - ----,
--R      2745 305 305 549 610 2745 10980
--R      2 112 6 84 5 1264 4 13 3 84 2 1772 2
--R y + ---- x - ---- x - ---- x - ---- x + ---- x + ---- x + ----,
--R      2745 305 305 549 305 2745 2745
--R      7 29 6 17 4 11 3 1 2 15 1
--R x + -- x - -- x - -- x + -- x + -- x + -]
--R      4 16 8 32 16 4
--R      Type: List(DistributedMultivariatePolynomial([z,y,x],Fraction(Integer)))
--E 5

```

```

--S 6 of 10
(n1,n2,n3) : HDMP([z,y,x],FRAC INT)
--R
--R
--R                                          Type: Void
--E 6

```

```

--S 7 of 10
n1 := d1
--R
--R
--R      2      2
--R (7) 4y x + 16x - 4z + 1
--R      Type: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction(Integer))
--E 7

```

```

--S 8 of 10
n2 := d2
--R
--R

```

```

--R      2
--R (8) 2z y + 4x + 1
--RType: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction(Integer))
--E 8

--S 9 of 10
n3 := d3
--R
--R
--R      2      2
--R (9) 2z x - 2y - x
--RType: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction(Integer))
--E 9

--S 10 of 10
groebner [n1,n2,n3]
--R
--R
--R (10)
--R      4      3      3      2      1      1      4      29      3      1      2      7      9      1
--R [y + 2x - - x + - z - -, x + - x - - y - - z x - - x - -,
--R      2      2      8      4      8      4      16      4
--R      2      1      2      2      1      2      2      1
--R z y + 2x + -, y x + 4x - z + -, z x - y - - x,
--R      2      2      4      2
--R z - 4y + 2x - - z - - x]
--R      4      2
--RType: List(HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction(Integer)))
--E 10
)spool
)lisp (bye)

```

— GeneralDistributedMultivariatePolynomial.help —

```

=====
MultivariatePolynomial
DistributedMultivariatePolynomial
HomogeneousDistributedMultivariatePolynomial
GeneralDistributedMultivariatePolynomial
=====

```

DistributedMultivariatePolynomial which is abbreviated as DMP and HomogeneousDistributedMultivariatePolynomial, which is abbreviated as HDMP, are very similar to MultivariatePolynomial except that they are represented and displayed in a non-recursive manner.

```
(d1,d2,d3) : DMP([z,y,x],FRAC INT)
              Type: Void
```

The constructor DMP orders its monomials lexicographically while HDMP orders them by total order refined by reverse lexicographic order.

```
d1 := -4*z + 4*y**2*x + 16*x**2 + 1
      2      2
      - 4z + 4y x + 16x + 1
      Type: DistributedMultivariatePolynomial([z,y,x],Fraction Integer)

d2 := 2*z*y**2 + 4*x + 1
      2
      2z y + 4x + 1
      Type: DistributedMultivariatePolynomial([z,y,x],Fraction Integer)

d3 := 2*z*x**2 - 2*y**2 - x
      2      2
      2z x - 2y - x
      Type: DistributedMultivariatePolynomial([z,y,x],Fraction Integer)
```

These constructors are mostly used in Groebner basis calculations.

```
groebner [d1,d2,d3]
      1568 6 1264 5 6 4 182 3 2047 2 103 2857
      [z - ---- x - ---- x + --- x + --- x - ---- x - ---- x - ----,
      2745 305 305 549 610 2745 10980
      2 112 6 84 5 1264 4 13 3 84 2 1772 2
      y + ---- x - --- x - ---- x - --- x + --- x + ---- x + ----,
      2745 305 305 549 305 2745 2745
      7 29 6 17 4 11 3 1 2 15 1
      x + -- x - -- x - -- x + -- x + -- x + -]
      4 16 8 32 16 4
      Type: List DistributedMultivariatePolynomial([z,y,x],Fraction Integer)
```

```
(n1,n2,n3) : HDMP([z,y,x],FRAC INT)
              Type: Void
```

```
n1 := d1
      2      2
      4y x + 16x - 4z + 1
      Type: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction Integer)

n2 := d2
      2
      2z y + 4x + 1
      Type: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction Integer)

n3 := d3
```

```

      2      2
2z x  - 2y  - x
Type: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction Integer)

```

Note that we get a different Groebner basis when we use the HDMP polynomials, as expected.

```

groebner [n1,n2,n3]
      4      3      3      2      1      1      4      29      3      1      2      7      9      1
[y  + 2x  - - x  + - z - -, x  + -- x  - - y  - - z x - -- x - -,
      2      2      8      4      8      4      16      4
      2      1      2      2      1      2      2      1
z y  + 2x + -, y x + 4x  - z + -, z x  - y  - - x,
      2      4      2
      2      2      2      1      3
z  - 4y  + 2x  - - z - - x]
      4      2
Type: List HomogeneousDistributedMultivariatePolynomial([z,y,x],
Fraction Integer)

```

GeneralDistributedMultivariatePolynomial is somewhat more flexible in the sense that as well as accepting a list of variables to specify the variable ordering, it also takes a predicate on exponent vectors to specify the term ordering. With this polynomial type the user can experiment with the effect of using completely arbitrary term orderings. This flexibility is mostly important for algorithms such as Groebner basis calculations which can be very sensitive to term ordering.

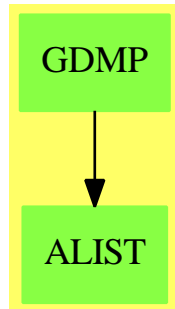
See Also:

```

o )help Polynomial
o )help UnivariatePolynomial
o )help MultivariatePolynomial
o )help HomogeneousDistributedMultivariatePolynomial
o )help DistributedMultivariatePolynomial
o )show GeneralDistributedMultivariatePolynomial

```

GeneralDistributedMultivariatePolynomial (GDMP)



See

⇒ “DistributedMultivariatePolynomial” (DMP) [5](#) on page [571](#)

⇒ “HomogeneousDistributedMultivariatePolynomial” (HDMP) [9](#) on page [1179](#)

Exports:

0	1	associates?
binomThmExpt	characteristic	charthRoot
coefficient	coefficients	coerce
conditionP	content	D
degree	differentiate	discriminant
eval	exquo	factor
factorPolynomial	factorSquareFreePolynomial	gcd
gcdPolynomial	ground	ground?
hash	isExpt	isPlus
isTimes	latex	lcm
leadingCoefficient	leadingMonomial	mainVariable
map	mapExponents	max
min	minimumDegree	monicDivide
monomial	monomial?	monomials
multivariate	numberOfMonomials	one?
patternMatch	pomopo!	prime?
primitiveMonomials	primitivePart	recip
reducedSystem	reductum	reorder
resultant	retract	retractIfCan
sample	solveLinearPolynomialEquation	squareFree
squareFreePart	squareFreePolynomial	subtractIfCan
totalDegree	unit?	unitCanonical
unitNormal	univariate	variables
zero?	?*?	?**?
?+?	?-?	-?
?=?	?~=?	?<?
?<=?	?>?	?>=?
?^?		

— domain GDMP GeneralDistributedMultivariatePolynomial —

```

)abbrev domain GDMP GeneralDistributedMultivariatePolynomial
++ Author: Barry Trager
++ Description:
++ This type supports distributed multivariate polynomials
++ whose variables are from a user specified list of symbols.
++ The coefficient ring may be non commutative,
++ but the variables are assumed to commute.
++ The term ordering is specified by its third parameter.
++ Suggested types which define term orderings include:
++ \spadtype{DirectProduct}, \spadtype{HomogeneousDirectProduct},
++ \spadtype{SplitHomogeneousDirectProduct} and finally
++ \spadtype{OrderedDirectProduct} which accepts an arbitrary user
++ function to define a term ordering.

GeneralDistributedMultivariatePolynomial(vl,R,E): public == private where
  vl: List Symbol
  R: Ring
  E: DirectProductCategory(#vl,NonNegativeInteger)
  OV ==> OrderedVariableList(vl)
  SUP ==> SparseUnivariatePolynomial
  NNI ==> NonNegativeInteger

  public == PolynomialCategory(R,E,OV) with
    reorder: (% ,List Integer) -> %
      ++ reorder(p, perm) applies the permutation perm to the variables
      ++ in a polynomial and returns the new correctly ordered polynomial

  private == PolynomialRing(R,E) add
    --representations
    Term := Record(k:E,c:R)
    Rep := List Term
    n := #vl
    Vec ==> Vector(NonNegativeInteger)
    zero?(p : %): Boolean == null(p : Rep)

    totalDegree p ==
      zero? p => 0
      "max"/[reduce("+", (t.k)::(Vector NNI), 0) for t in p]

    monomial(p:%, v: OV,e: NonNegativeInteger):% ==
      locv := lookup v
      p*monomial(1,
        directProduct [if z=locv then e else 0 for z in 1..n]$Vec)

    coerce(v: OV):% == monomial(1,v,1)

    listCoef(p : %): List R ==

```



```

    rec : Term
    [rec.c for rec in (p:Rep)]

mainVariable(p: %) ==
  zero?(p) => "failed"
  for v in vl repeat
    vv := variable(v)::OV
    if degree(p,vv)>0 then return vv
  "failed"

ground?(p) == mainVariable(p) case "failed"

retract(p : %): R ==
  not ground? p => error "not a constant"
  leadingCoefficient p

retractIfCan(p : %): Union(R,"failed") ==
  ground?(p) => leadingCoefficient p
  "failed"

degree(p: %,v: OV) == degree(univariate(p,v))
minimumDegree(p: %,v: OV) == minimumDegree(univariate(p,v))
differentiate(p: %,v: OV) ==
  multivariate(differentiate(univariate(p,v)),v)

degree(p: %,lv: List OV) == [degree(p,v) for v in lv]
minimumDegree(p: %,lv: List OV) == [minimumDegree(p,v) for v in lv]

numberOfMonomials(p:%) ==
  l : Rep := p : Rep
  null(l) => 1
  #l

monomial?(p : %): Boolean ==
  l : Rep := p : Rep
  null(l) or null rest(l)

if R has OrderedRing then
  maxNorm(p : %): R ==
    l : List R := nil
    r,m : R
    m := 0
    for r in listCoef(p) repeat
      if r > m then m := r
      else if (-r) > m then m := -r
    m

--trailingCoef(p : %) ==
-- l : Rep := p : Rep
-- null l => 0

```

```

-- r : Term := last l
-- r.c

--leadingPrimitiveMonomial(p : %) ==
--  ground?(p) => 1$%
--  r : Term := first(p:Rep)
--  r := [r.k,1$R]$Term      -- new cell
--  list(r)$Rep :: %

-- The following 2 defs are inherited from PolynomialRing

--leadingMonomial(p : %) ==
--  ground?(p) => p
--  r : Term := first(p:Rep)
--  r := [r.k,r.c]$Term      -- new cell
--  list(r)$Rep :: %

--reductum(p : %): % ==
--  ground? p => 0$%
--  (rest(p:Rep)): %

if R has Field then
  (p : %) / (r : R) == inv(r) * p

variables(p : %) ==
  maxdeg:Vector(NonNegativeInteger) := new(n,0)
  while not zero?(p) repeat
    tdeg := degree p
    p := reductum p
    for i in 1..n repeat
      maxdeg.i := max(maxdeg.i, tdeg.i)
  [index(i:PositiveInteger) for i in 1..n | maxdeg.i^=0]

reorder(p : %,perm: List Integer): % ==
  #perm ^= n => error "must be a complete permutation of all vars"
  q := [[directProduct [term.k.j for j in perm]$Vec,term.c]$Term
        for term in p]
  sort((z1,z2) +-> z1.k > z2.k,q)

--coerce(dp:DistributedMultivariatePolynomial(v1,R)): % ==
--  q:=dp:List(Term)
--  sort(#1.k > #2.k,q): %

univariate(p : %,v: OV):SUP(%) ==
  zero?(p) => 0
  exp := degree p
  locv := lookup v
  deg:NonNegativeInteger := 0
  nexp := directProduct [if i=locv then (deg :=exp.i;0) else exp.i
                        for i in 1..n]$Vec

```

```

    monomial(monomial(leadingCoefficient p,nexp),deg)+
        univariate(reductum p,v)

eval(p: %,v: OV,val:%):% == univariate(p,v)(val)

eval(p: %,v: OV,val:R):% == eval(p,v,val::%)$%

eval(p: %,lv: List OV,lval: List R):% ==
    lv = [] => p
    eval(eval(p,first lv,(first lval)::%)$%, rest lv, rest lval)$%

-- assume Lvar are sorted correctly
evalSortedVarlist(p: %,Lvar: List OV,Lpval: List %):% ==
    v := mainVariable p
    v case "failed" => p
    pv := v:: OV
    Lvar=[] or Lpval=[] => p
    mvar := Lvar.first
    mvar > pv => evalSortedVarlist(p,Lvar.rest,Lpval.rest)
    pval := Lpval.first
    pts:SUP(%) := map(x+>evalSortedVarlist(x,Lvar,Lpval),univariate(p,pv))
    mvar=pv => pts(pval)
    multivariate(pts,pv)

eval(p:%,Lvar:List OV,Lpval:List %):% ==
    nlvar:List OV := sort((x,y) +-> x > y,Lvar)
    nlpval :=
        Lvar = nlvar => Lpval
        nlpval := [Lpval.position(mvar,Lvar) for mvar in nlvar]
    evalSortedVarlist(p,nlvar,nlpval)

multivariate(p1:SUP(%),v: OV):% ==
    0=p1 => 0
    degree p1 = 0 => leadingCoefficient p1
    leadingCoefficient(p1)*(v::%)**degree(p1) +
        multivariate(reductum p1,v)

univariate(p: %):SUP(R) ==
    (v := mainVariable p) case "failed" =>
        monomial(leadingCoefficient p,0)
    q := univariate(p,v:: OV)
    ans:SUP(R) := 0
    while q ^!= 0 repeat
        ans := ans + monomial(ground leadingCoefficient q,degree q)
        q := reductum q
    ans

multivariate(p:SUP(R),v: OV):% ==
    0=p => 0
    (leadingCoefficient p)*monomial(1,v,degree p) +

```

```

multivariate(reductum p,v)

if R has GcdDomain then
  content(p: %):R ==
    zero?(p) => 0
    "gcd"/[t.c for t in p]

if R has EuclideanDomain and not(R has FloatingPointSystem) then
  gcd(p: %,q:%):% ==
    gcd(p,q)$PolynomialGcdPackage(E,OV,R,%)

else gcd(p: %,q:%):% ==
  r : R
  (pv := mainVariable(p)) case "failed" =>
    (r := leadingCoefficient p) = 0$R => q
    gcd(r,content q)::%
  (qv := mainVariable(q)) case "failed" =>
    (r := leadingCoefficient q) = 0$R => p
    gcd(r,content p)::%
  pv<qv => gcd(p,content univariate(q,qv))
  qv<pv => gcd(q,content univariate(p,pv))
  multivariate(gcd(univariate(p,pv),univariate(q,qv)),pv)

coerce(p: %) : OutputForm ==
  zero?(p) => (0$R) :: OutputForm
  l,lt : List OutputForm
  lt := nil
  vl1 := [v::OutputForm for v in vl]
  for t in reverse p repeat
    l := nil
    for i in 1..#vl1 repeat
      t.k.i = 0 => l
      t.k.i = 1 => l := cons(vl1.i,l)
    l := cons(vl1.i ** t.k.i ::OutputForm,l)
  l := reverse l
  if (t.c ^= 1) or (null l) then l := cons(t.c :: OutputForm,l)
  1 = #l => lt := cons(first l,lt)
  lt := cons(reduce("*",l),lt)
  1 = #lt => first lt
  reduce("+",lt)

```

— GDMP.dotabb —

"GDMP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=GDMP"]

"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
 "GDMP" -> "ALIST"

domain GMPOL GeneralModulePolynomial

— GeneralModulePolynomial.input —

```
)set break resume
)sys rm -f GeneralModulePolynomial.output
)spool GeneralModulePolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GeneralModulePolynomial
--R
--R GeneralModulePolynomial(vl: List(Symbol),R: CommutativeRing,IS: OrderedSet,E: DirectProductC
--R Abbreviation for GeneralModulePolynomial is GMPOL
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for GMPOL
--R
--R----- Operations -----
--R ?? : (R,%) -> %                ?? : (%,R) -> %
--R ?? : (%,P) -> %                ?? : (P,%) -> %
--R ?? : (Integer,%) -> %          ?? : (PositiveInteger,%) -> %
--R ?? : (%,%) -> %                ?-? : (%,%) -> %
--R -? : % -> %                    ?? : (%,%) -> Boolean
--R 0 : () -> %                    build : (R,IS,E) -> %
--R coerce : % -> OutputForm       hash : % -> SingleInteger
--R latex : % -> String             leadingCoefficient : % -> R
--R leadingExponent : % -> E         leadingIndex : % -> IS
--R multMonom : (R,E,%) -> %        reductum : % -> %
--R sample : () -> %                unitVector : IS -> %
--R zero? : % -> Boolean             ?~? : (%,%) -> Boolean
--R ?? : (NonNegativeInteger,%) -> %
--R leadingMonomial : % -> ModuleMonomial(IS,E,ff)
--R monomial : (R,ModuleMonomial(IS,E,ff)) -> %
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)
```

— GeneralModulePolynomial.help —

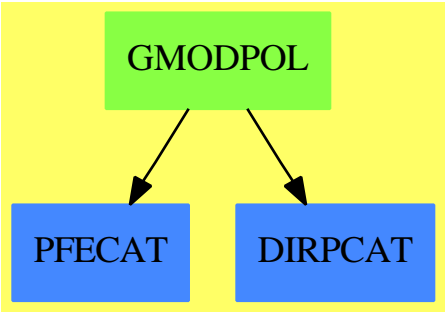
GeneralModulePolynomial examples

This package is undocumented

See Also:

o)show GeneralModulePolynomial

GeneralModulePolynomial (GMODPOL)



See

⇒ “ModuleMonomial” (MODMONOM) [14](#) on page [1644](#)

Exports:

0	build	coerce	hash	latex
leadingCoefficient	leadingExponent	leadingIndex	leadingMonomial	monomial
multMonom	reductum	sample	subtractIfCan	unitVector
zero?	?~=?	?*?	?+?	?-?
-?	?=?			

— domain GMODPOL GeneralModulePolynomial —

```
)abbrev domain GMODPOL GeneralModulePolynomial
++ Author: Mark Botch
++ Description:
++ This package is undocumented
```

GeneralModulePolynomial(v1, R, IS, E, ff, P): public == private where

```

v1: List(Symbol)
R: CommutativeRing
IS: OrderedSet
NNI ==> NonNegativeInteger
E: DirectProductCategory(#v1, NNI)
MM ==> Record(index:IS, exponent:E)
ff: (MM, MM) -> Boolean
OV ==> OrderedVariableList(v1)
P: PolynomialCategory(R, E, OV)
ModMonom ==> ModuleMonomial(IS, E, ff)

public == Join(Module(P), Module(R)) with
  leadingCoefficient: $ -> R
    ++ leadingCoefficient(x) is not documented
  leadingMonomial: $ -> ModMonom
    ++ leadingMonomial(x) is not documented
  leadingExponent: $ -> E
    ++ leadingExponent(x) is not documented
  leadingIndex: $ -> IS
    ++ leadingIndex(x) is not documented
  reductum: $ -> $
    ++ reductum(x) is not documented
  monomial: (R, ModMonom) -> $
    ++ monomial(r,x) is not documented
  unitVector: IS -> $
    ++ unitVector(x) is not documented
  build: (R, IS, E) -> $
    ++ build(r,i,e) is not documented
  multMonom: (R, E, $) -> $
    ++ multMonom(r,e,x) is not documented
  "·": (P,$) -> $
    ++ p*x is not documented

private == FreeModule(R, ModMonom) add
  Rep:= FreeModule(R, ModMonom)
  leadingMonomial(p:$):ModMonom == leadingSupport(p)$Rep
  leadingExponent(p:$):E == exponent(leadingMonomial p)
  leadingIndex(p:$):IS == index(leadingMonomial p)
  unitVector(i:IS):$ == monomial(1,[i, 0$E]$ModMonom)

-----

  build(c:R, i:IS, e:E):$ == monomial(c, construct(i, e))

-----

---- WARNING: assumes  $c \neq 0$ 

```

```

multMonom(c:R, e:E, mp:$):$ ==
  zero? mp => mp
  monomial(c * leadingCoefficient mp, [leadingIndex mp,
    e + leadingExponent mp]) + multMonom(c, e, reductum mp)

```

```

((p:P) * (mp:$)): $ ==
  zero? p => 0
  multMonom(leadingCoefficient p, degree p, mp) +
    reductum(p) * mp

```

— GMODPOL.dotabb —

```

"GMODPOL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=GMODPOL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"GMODPOL" -> "PFECAT"
"GMODPOL" -> "DIRPCAT"

```

domain GCNAALG GenericNonAssociativeAlgebra

— GenericNonAssociativeAlgebra.input —

```

)set break resume
)sys rm -f GenericNonAssociativeAlgebra.output
)spool GenericNonAssociativeAlgebra.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GenericNonAssociativeAlgebra
--R
--R GenericNonAssociativeAlgebra(R: CommutativeRing,n: PositiveInteger,ls: List(Symbol),gamma: Vector(Matrix))
--R Abbreviation for GenericNonAssociativeAlgebra is GCNAALG
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for GCNAALG

```



```

--R
--R----- Operations -----
--R ???: (% , %) -> %
--R ??? : (PositiveInteger, %) -> %
--R ?+? : (% , %) -> %
--R -? : % -> %
--R 0 : () -> %
--R antiAssociative? : () -> Boolean
--R antiCommutator : (% , %) -> %
--R associator : (% , % , %) -> %
--R coerce : % -> OutputForm
--R commutator : (% , %) -> %
--R generic : Vector(%) -> %
--R generic : Symbol -> %
--R hash : % -> SingleInteger
--R jordanAdmissible? : () -> Boolean
--R latex : % -> String
--R lieAdmissible? : () -> Boolean
--R powerAssociative? : () -> Boolean
--R rightAlternative? : () -> Boolean
--R someBasis : () -> Vector(%)
--R ?~=? : (% , %) -> Boolean
--R ??? : (SquareMatrix(n, Fraction(Polynomial(R))), %) -> %
--R ??? : (Fraction(Polynomial(R)), %) -> %
--R ??? : (% , Fraction(Polynomial(R))) -> %
--R ??? : (NonNegativeInteger, %) -> %
--R apply : (Matrix(Fraction(Polynomial(R))), %) -> %
--R associatorDependence : () -> List(Vector(Fraction(Polynomial(R)))) if Fraction(Polynomial(R))
--R coerce : Vector(Fraction(Polynomial(R))) -> %
--R conditionsForIdempotents : () -> List(Polynomial(R)) if R has INTDOM
--R conditionsForIdempotents : Vector(%) -> List(Polynomial(R)) if R has INTDOM
--R conditionsForIdempotents : () -> List(Polynomial(Fraction(Polynomial(R))))
--R conditionsForIdempotents : Vector(%) -> List(Polynomial(Fraction(Polynomial(R))))
--R convert : Vector(Fraction(Polynomial(R))) -> %
--R convert : % -> Vector(Fraction(Polynomial(R)))
--R coordinates : Vector(%) -> Matrix(Fraction(Polynomial(R)))
--R coordinates : % -> Vector(Fraction(Polynomial(R)))
--R coordinates : (Vector(%) , Vector(%)) -> Matrix(Fraction(Polynomial(R)))
--R coordinates : (% , Vector(%)) -> Vector(Fraction(Polynomial(R)))
--R ?.? : (% , Integer) -> Fraction(Polynomial(R))
--R generic : (Vector(Symbol) , Vector(%)) -> %
--R generic : (Symbol , Vector(%)) -> %
--R genericLeftDiscriminant : () -> Fraction(Polynomial(R)) if R has INTDOM
--R genericLeftMinimalPolynomial : % -> SparseUnivariatePolynomial(Fraction(Polynomial(R))) if R
--R genericLeftNorm : % -> Fraction(Polynomial(R)) if R has INTDOM
--R genericLeftTrace : % -> Fraction(Polynomial(R)) if R has INTDOM
--R genericLeftTraceForm : (% , %) -> Fraction(Polynomial(R)) if R has INTDOM
--R genericRightDiscriminant : () -> Fraction(Polynomial(R)) if R has INTDOM
--R genericRightMinimalPolynomial : % -> SparseUnivariatePolynomial(Fraction(Polynomial(R))) if
--R genericRightNorm : % -> Fraction(Polynomial(R)) if R has INTDOM

```

```

--R genericRightTrace : % -> Fraction(Polynomial(R)) if R has INTDOM
--R genericRightTraceForm : (%,%) -> Fraction(Polynomial(R)) if R has INTDOM
--R leftCharacteristicPolynomial : % -> SparseUnivariatePolynomial(Fraction(Polynomial(R)))
--R leftDiscriminant : () -> Fraction(Polynomial(R))
--R leftDiscriminant : Vector(%) -> Fraction(Polynomial(R))
--R leftMinimalPolynomial : % -> SparseUnivariatePolynomial(Fraction(Polynomial(R))) if Fraction(Polynomial(R)) has INTDOM
--R leftNorm : % -> Fraction(Polynomial(R))
--R leftPower : (%,PositiveInteger) -> %
--R leftRankPolynomial : () -> SparseUnivariatePolynomial(Fraction(Polynomial(R))) if R has INTDOM
--R leftRankPolynomial : () -> SparseUnivariatePolynomial(Polynomial(Fraction(Polynomial(R)))) if Fraction(Polynomial(R)) has INTDOM
--R leftRecip : % -> Union(%, "failed") if Fraction(Polynomial(R)) has INTDOM
--R leftRegularRepresentation : % -> Matrix(Fraction(Polynomial(R)))
--R leftRegularRepresentation : (%,Vector(%%)) -> Matrix(Fraction(Polynomial(R)))
--R leftTrace : % -> Fraction(Polynomial(R))
--R leftTraceMatrix : () -> Matrix(Fraction(Polynomial(R)))
--R leftTraceMatrix : Vector(%) -> Matrix(Fraction(Polynomial(R)))
--R leftUnit : () -> Union(%, "failed") if Fraction(Polynomial(R)) has INTDOM
--R leftUnits : () -> Union(Record(particular: %,basis: List(%)), "failed")
--R noncommutativeJordanAlgebra? : () -> Boolean
--R plenaryPower : (%,PositiveInteger) -> %
--R recip : % -> Union(%, "failed") if Fraction(Polynomial(R)) has INTDOM
--R represents : Vector(Fraction(Polynomial(R))) -> %
--R represents : (Vector(Fraction(Polynomial(R))),Vector(%%)) -> %
--R rightCharacteristicPolynomial : % -> SparseUnivariatePolynomial(Fraction(Polynomial(R)))
--R rightDiscriminant : () -> Fraction(Polynomial(R))
--R rightDiscriminant : Vector(%) -> Fraction(Polynomial(R))
--R rightMinimalPolynomial : % -> SparseUnivariatePolynomial(Fraction(Polynomial(R))) if Fraction(Polynomial(R)) has INTDOM
--R rightNorm : % -> Fraction(Polynomial(R))
--R rightPower : (%,PositiveInteger) -> %
--R rightRankPolynomial : () -> SparseUnivariatePolynomial(Fraction(Polynomial(R))) if R has INTDOM
--R rightRankPolynomial : () -> SparseUnivariatePolynomial(Polynomial(Fraction(Polynomial(R)))) if Fraction(Polynomial(R)) has INTDOM
--R rightRecip : % -> Union(%, "failed") if Fraction(Polynomial(R)) has INTDOM
--R rightRegularRepresentation : % -> Matrix(Fraction(Polynomial(R)))
--R rightRegularRepresentation : (%,Vector(%%)) -> Matrix(Fraction(Polynomial(R)))
--R rightTrace : % -> Fraction(Polynomial(R))
--R rightTraceMatrix : () -> Matrix(Fraction(Polynomial(R)))
--R rightTraceMatrix : Vector(%) -> Matrix(Fraction(Polynomial(R)))
--R rightUnit : () -> Union(%, "failed") if Fraction(Polynomial(R)) has INTDOM
--R rightUnits : () -> Union(Record(particular: %,basis: List(%)), "failed")
--R structuralConstants : () -> Vector(Matrix(Fraction(Polynomial(R))))
--R structuralConstants : Vector(%) -> Vector(Matrix(Fraction(Polynomial(R))))
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R unit : () -> Union(%, "failed") if Fraction(Polynomial(R)) has INTDOM
--R
--E 1

)spool
)lisp (bye)

```

— GenericNonAssociativeAlgebra.help —

```
=====
GenericNonAssociativeAlgebra examples
=====
```

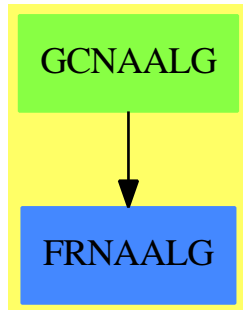
AlgebraGenericElementPackage allows you to create generic elements of an algebra, i.e. the scalars are extended to include symbolic coefficients.

See Also:

o)show GenericNonAssociativeAlgebra

—————

GenericNonAssociativeAlgebra (GCNAALG)



Exports:

0	alternative?
antiAssociative?	antiCommutative?
antiCommutator	apply
associative?	associator
associatorDependence	basis
coerce	commutative?
commutator	conditionsForIdempotents
convert	convert
coordinates	coordinates
coordinates	coordinates
flexible?	generic
genericLeftDiscriminant	genericLeftMinimalPolynomial
genericLeftNorm	genericLeftTrace
genericLeftTraceForm	genericRightDiscriminant
genericRightMinimalPolynomial	genericRightNorm
genericRightTrace	genericRightTraceForm
hash	jacobiIdentity?
jordanAdmissible?	jordanAlgebra?
latex	leftAlternative?
leftCharacteristicPolynomial	leftDiscriminant
leftDiscriminant	leftMinimalPolynomial
leftNorm	leftPower
leftRankPolynomial	leftRankPolynomial
leftRecip	leftRegularRepresentation
leftRegularRepresentation	leftTrace
leftTraceMatrix	leftTraceMatrix
leftUnit	leftUnits
lieAdmissible?	lieAlgebra?
noncommutativeJordanAlgebra?	plenaryPower
powerAssociative?	rank
recip	represents
rightAlternative?	rightCharacteristicPolynomial
rightDiscriminant	rightDiscriminant
rightMinimalPolynomial	rightNorm
rightPower	rightRankPolynomial
rightRankPolynomial	rightRecip
rightRegularRepresentation	rightRegularRepresentation
rightTrace	rightTraceMatrix
rightTraceMatrix	rightUnit
rightUnits	sample
someBasis	structuralConstants
structuralConstants	subtractIfCan
unit	zero?
?*?	?**?
?+?	?-?
-?	?=?
?.?	?~=?

— domain GCNAALG GenericNonAssociativeAlgebra —

```

)abbrev domain GCNAALG GenericNonAssociativeAlgebra
++ Authors: J. Grabmeier, R. Wisbauer
++ Date Created: 26 June 1991
++ Date Last Updated: 26 June 1991
++ Reference:
++   A. Woerz-Busekros: Algebra in Genetics
++   Lectures Notes in Biomathematics 36,
++   Springer-Verlag, Heidelberg, 1980
++ Description:
++ AlgebraGenericElementPackage allows you to create generic elements
++ of an algebra, i.e. the scalars are extended to include symbolic
++ coefficients

GenericNonAssociativeAlgebra(R : CommutativeRing, n : PositiveInteger,
  ls : List Symbol, gamma: Vector Matrix R ): public == private where

NNI ==> NonNegativeInteger
V   ==> Vector
PR  ==> Polynomial R
FPR ==> Fraction Polynomial R
SUP ==> SparseUnivariatePolynomial
S   ==> Symbol

public ==> Join(FramedNonAssociativeAlgebra(FPR), _
  LeftModule(SquareMatrix(n,FPR)) ) with

coerce : Vector FPR -> %
  ++ coerce(v) assumes that it is called with a vector
  ++ of length equal to the dimension of the algebra, then
  ++ a linear combination with the basis element is formed
leftUnits:() -> Union(Record(particular: %, basis: List %), "failed")
  ++ leftUnits() returns the affine space of all left units of the
  ++ algebra, or \spad{"failed"} if there is none
rightUnits:() -> Union(Record(particular: %, basis: List %), "failed")
  ++ rightUnits() returns the affine space of all right units of the
  ++ algebra, or \spad{"failed"} if there is none
generic : () -> %
  ++ generic() returns a generic element, i.e. the linear combination
  ++ of the fixed basis with the symbolic coefficients
  ++ \spad{%x1,%x2,...}
generic : Symbol -> %
  ++ generic(s) returns a generic element, i.e. the linear combination
  ++ of the fixed basis with the symbolic coefficients
  ++ \spad{s1,s2,...}
generic : Vector Symbol -> %
  ++ generic(vs) returns a generic element, i.e. the linear combination
  ++ of the fixed basis with the symbolic coefficients

```

```

++ \spad{vs};
++ error, if the vector of symbols is too short
generic : Vector % -> %
++ generic(ve) returns a generic element, i.e. the linear combination
++ of \spad{ve} basis with the symbolic coefficients
++ \spad{%x1,%x2,...}
generic : (Symbol, Vector %) -> %
++ generic(s,v) returns a generic element, i.e. the linear combination
++ of v with the symbolic coefficients
++ \spad{s1,s2,...}
generic : (Vector Symbol, Vector %) -> %
++ generic(vs,ve) returns a generic element, i.e. the linear combination
++ of \spad{ve} with the symbolic coefficients \spad{vs}
++ error, if the vector of symbols is shorter than the vector of
++ elements
if R has IntegralDomain then
leftRankPolynomial : () -> SparseUnivariatePolynomial FPR
++ leftRankPolynomial() returns the left minimal polynomial
++ of the generic element
genericLeftMinimalPolynomial : % -> SparseUnivariatePolynomial FPR
++ genericLeftMinimalPolynomial(a) substitutes the coefficients
++ of {em a} for the generic coefficients in
++ \spad{leftRankPolynomial()}
genericLeftTrace : % -> FPR
++ genericLeftTrace(a) substitutes the coefficients
++ of \spad{a} for the generic coefficients into the
++ coefficient of the second highest term in
++ \spadfun{leftRankPolynomial} and changes the sign.
++ This is a linear form
genericLeftNorm : % -> FPR
++ genericLeftNorm(a) substitutes the coefficients
++ of \spad{a} for the generic coefficients into the
++ coefficient of the constant term in \spadfun{leftRankPolynomial}
++ and changes the sign if the degree of this polynomial is odd.
++ This is a form of degree k
rightRankPolynomial : () -> SparseUnivariatePolynomial FPR
++ rightRankPolynomial() returns the right minimal polynomial
++ of the generic element
genericRightMinimalPolynomial : % -> SparseUnivariatePolynomial FPR
++ genericRightMinimalPolynomial(a) substitutes the coefficients
++ of \spad{a} for the generic coefficients in
++ \spadfun{rightRankPolynomial}
genericRightTrace : % -> FPR
++ genericRightTrace(a) substitutes the coefficients
++ of \spad{a} for the generic coefficients into the
++ coefficient of the second highest term in
++ \spadfun{rightRankPolynomial} and changes the sign
genericRightNorm : % -> FPR
++ genericRightNorm(a) substitutes the coefficients
++ of \spad{a} for the generic coefficients into the

```

```

    ++ coefficient of the constant term in \spadfun{rightRankPolynomial}
    ++ and changes the sign if the degree of this polynomial is odd
genericLeftTraceForm : (%,%) -> FPR
    ++ genericLeftTraceForm (a,b) is defined to be
    ++ \spad{genericLeftTrace (a*b)}, this defines
    ++ a symmetric bilinear form on the algebra
genericLeftDiscriminant: () -> FPR
    ++ genericLeftDiscriminant() is the determinant of the
    ++ generic left trace forms of all products of basis element,
    ++ if the generic left trace form is associative, an algebra
    ++ is separable if the generic left discriminant is invertible,
    ++ if it is non-zero, there is some ring extension which
    ++ makes the algebra separable
genericRightTraceForm : (%,%) -> FPR
    ++ genericRightTraceForm (a,b) is defined to be
    ++ \spadfun{genericRightTrace (a*b)}, this defines
    ++ a symmetric bilinear form on the algebra
genericRightDiscriminant: () -> FPR
    ++ genericRightDiscriminant() is the determinant of the
    ++ generic left trace forms of all products of basis element,
    ++ if the generic left trace form is associative, an algebra
    ++ is separable if the generic left discriminant is invertible,
    ++ if it is non-zero, there is some ring extension which
    ++ makes the algebra separable
conditionsForIdempotents: Vector % -> List Polynomial R
    ++ conditionsForIdempotents([v1,...,vn]) determines a complete list
    ++ of polynomial equations for the coefficients of idempotents
    ++ with respect to the \spad{R}-module basis \spad{v1},...,\spad{vn}
conditionsForIdempotents: () -> List Polynomial R
    ++ conditionsForIdempotents() determines a complete list
    ++ of polynomial equations for the coefficients of idempotents
    ++ with respect to the fixed \spad{R}-module basis

private ==> AlgebraGivenByStructuralConstants(FPR,n,ls,_
    coerce(gamma)$CoerceVectorMatrixPackage(R) ) add

listOfNumbers : List String := [STRINGIMAGE(q)$Lisp for q in 1..n]
symbolsForCoef : V Symbol :=
    [concat("%", concat("x", i))::Symbol for i in listOfNumbers]
genericElement : % :=
    v : Vector PR :=
        [monomial(1$PR, [symbolsForCoef.i],[1]) for i in 1..n]
    convert map(coerce,v)$VectorFunctions2(PR,FPR)

eval : (FPR, %) -> FPR
eval(rf,a) ==
    -- for the moment we only substitute the numerators
    -- of the coefficients
coefUfa : List PR :=
    map(numer, entries coordinates a)$ListFunctions2(FPR,PR)

```

```

ls : List PR := [monomial(1$PR, [s],[1]) for s in entries symbolsForCoef]
lEq : List Equation PR := []
for i in 1..maxIndex ls repeat
  lEq := cons(equation(ls.i,coefOfa.i)$Equation(PR) , lEq)
top : PR := eval(numer(rf),lEq)$PR
bot : PR := eval(denom(rf),lEq)$PR
top/bot

```

```

if R has IntegralDomain then

```

```

  genericLeftTraceForm(a,b) == genericLeftTrace(a*b)
  genericLeftDiscriminant() ==
    listBasis : List % := entries basis()$%
    m : Matrix FPR := matrix
      [[genericLeftTraceForm(a,b) for a in listBasis] for b in listBasis]
    determinant m

  genericRightTraceForm(a,b) == genericRightTrace(a*b)
  genericRightDiscriminant() ==
    listBasis : List % := entries basis()$%
    m : Matrix FPR := matrix
      [[genericRightTraceForm(a,b) for a in listBasis] for b in listBasis]
    determinant m

```

```

leftRankPoly : SparseUnivariatePolynomial FPR := 0
initLeft? : Boolean :=true

```

```

initializeLeft: () -> Void
initializeLeft() ==
  -- reset initialize flag
  initLeft?:=false
  leftRankPoly := leftMinimalPolynomial genericElement
  void()$Void

```

```

rightRankPoly : SparseUnivariatePolynomial FPR := 0
initRight? : Boolean :=true

```

```

initializeRight: () -> Void
initializeRight() ==
  -- reset initialize flag
  initRight?:=false
  rightRankPoly := rightMinimalPolynomial genericElement
  void()$Void

```

```

leftRankPolynomial() ==
  if initLeft? then initializeLeft()
  leftRankPoly

```



```

rightRankPolynomial() ==
  if initRight? then initializeRight()
  rightRankPoly

genericLeftMinimalPolynomial a ==
  if initLeft? then initializeLeft()
  map(x+>eval(x,a),leftRankPoly)$SUP(FPR)

genericRightMinimalPolynomial a ==
  if initRight? then initializeRight()
  map(x+>eval(x,a),rightRankPoly)$SUP(FPR)

genericLeftTrace a ==
  if initLeft? then initializeLeft()
  d1 : NNI := (degree leftRankPoly - 1) :: NNI
  rf : FPR := coefficient(leftRankPoly, d1)
  rf := eval(rf,a)
  - rf

genericRightTrace a ==
  if initRight? then initializeRight()
  d1 : NNI := (degree rightRankPoly - 1) :: NNI
  rf : FPR := coefficient(rightRankPoly, d1)
  rf := eval(rf,a)
  - rf

genericLeftNorm a ==
  if initLeft? then initializeLeft()
  rf : FPR := coefficient(leftRankPoly, 1)
  if odd? degree leftRankPoly then rf := - rf
  rf

genericRightNorm a ==
  if initRight? then initializeRight()
  rf : FPR := coefficient(rightRankPoly, 1)
  if odd? degree rightRankPoly then rf := - rf
  rf

conditionsForIdempotents(b: V %) : List Polynomial R ==
  x : % := generic(b)
  map(numer,entries coordinates(x*x-x,b))$ListFunctions2(FPR,PR)

conditionsForIdempotents(): List Polynomial R ==
  x : % := genericElement
  map(numer,entries coordinates(x*x-x))$ListFunctions2(FPR,PR)

generic() == genericElement

generic(vs:V S, ve: V %): % ==

```

```

maxIndex v > maxIndex ve =>
  error "generic: too little symbols"
v : Vector PR :=
  [monomial(1$PR, [vs.i],[1]) for i in 1..maxIndex ve]
represents(map(coerce,v)$VectorFunctions2(PR,FPR),ve)

generic(s: S, ve: V %): % ==
  lON : List String := [STRINGIMAGE(q)$Lisp for q in 1..maxIndex ve]
  sFC : Vector Symbol :=
    [concat(s pretend String, i)::Symbol for i in lON]
  generic(sFC, ve)

generic(ve : V %) ==
  lON : List String := [STRINGIMAGE(q)$Lisp for q in 1..maxIndex ve]
  sFC : Vector Symbol :=
    [concat("%", concat("x", i))::Symbol for i in lON]
  v : Vector PR :=
    [monomial(1$PR, [sFC.i],[1]) for i in 1..maxIndex ve]
  represents(map(coerce,v)$VectorFunctions2(PR,FPR),ve)

generic(vs:V S): % == generic(vs, basis()$%)

generic(s: S): % == generic(s, basis()$%)

-- variations on eval
--coefOfa : List FPR := entries coordinates a
--ls : List Symbol := entries symbolsForCoef
-- a very dangerous sequential implementation for the moment,
-- because the compiler doesn't manage the parallel code
-- also doesn't run:
-- not known that (Fraction (Polynomial R)) has (has (Polynomial R)
-- (Evalable (Fraction (Polynomial R))))
--res : FPR := rf
--for eq in lEq repeat res := eval(res,eq)$FPR
--res
--rf
--eval(rf, le)$FPR
--eval(rf, entries symbolsForCoef, coefOfa)$FPR
--eval(rf, ls, coefOfa)$FPR
--le : List Equation PR := [equation(lh,rh) for lh in ls for rh in coefOfa]

```

— GCNAALG.dotabb —

"GCNAALG" [color="#88FF44",href="bookvol10.3.pdf#nameddest=GCNAALG"]
 "FRNAALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRNAALG"]
 "GCNAALG" -> "FRNAALG"

domain GPOLSET GeneralPolynomialSet

— GeneralPolynomialSet.input —

```

)set break resume
)sys rm -f GeneralPolynomialSet.output
)spool GeneralPolynomialSet.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GeneralPolynomialSet
--R
--R GeneralPolynomialSet(R: Ring,E: OrderedAbelianMonoidSup,VarSet: OrderedSet,P: RecursivePolynomialSet)
--R Abbreviation for GeneralPolynomialSet is GPOLSET
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for GPOLSET
--R
--R----- Operations -----
--R ?? : (%,%) -> Boolean
--R coerce : % -> List(P)
--R collect : (%,VarSet) -> %
--R collectUnder : (%,VarSet) -> %
--R collectUpper : (%,VarSet) -> %
--R construct : List(P) -> %
--R convert : List(P) -> %
--R copy : % -> %
--R empty : () -> %
--R empty? : % -> Boolean
--R eq? : (%,%) -> Boolean
--R hash : % -> SingleInteger
--R latex : % -> String
--R mainVariables : % -> List(VarSet)
--R map : ((P -> P),%) -> %
--R mvar : % -> VarSet
--R retract : List(P) -> %
--R sample : () -> %
--R trivialIdeal? : % -> Boolean
--R variables : % -> List(VarSet)
--R ~=? : (%,%) -> Boolean
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R any? : ((P -> Boolean),%) -> Boolean if $ has finiteAggregate
--R convert : % -> InputForm if P has KONVERT(INFORM)
--R count : ((P -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R count : (P,%) -> NonNegativeInteger if $ has finiteAggregate and P has SETCAT
--R eval : (%,List(Equation(P))) -> % if P has EVALAB(P) and P has SETCAT
--R eval : (%,Equation(P)) -> % if P has EVALAB(P) and P has SETCAT
--R eval : (%,P,P) -> % if P has EVALAB(P) and P has SETCAT
--R eval : (%,List(P),List(P)) -> % if P has EVALAB(P) and P has SETCAT
--R every? : ((P -> Boolean),%) -> Boolean if $ has finiteAggregate
--R find : ((P -> Boolean),%) -> Union(P,"failed")
--R headRemainder : (P,%) -> Record(num: P,den: R) if R has INTDOM

```

```

--R less? : (%,NonNegativeInteger) -> Boolean
--R mainVariable? : (VarSet,%) -> Boolean
--R map! : ((P -> P),%) -> % if $ has shallowlyMutable
--R member? : (P,%) -> Boolean if $ has finiteAggregate and P has SETCAT
--R members : % -> List(P) if $ has finiteAggregate
--R more? : (%,NonNegativeInteger) -> Boolean
--R parts : % -> List(P) if $ has finiteAggregate
--R reduce : (((P,P) -> P),%) -> P if $ has finiteAggregate
--R reduce : (((P,P) -> P),%,P) -> P if $ has finiteAggregate
--R reduce : (((P,P) -> P),%,P,P) -> P if $ has finiteAggregate and P has SETCAT
--R remainder : (P,%) -> Record(rnum: R,polnum: P,den: R) if R has INTDOM
--R remove : ((P -> Boolean),%) -> % if $ has finiteAggregate
--R remove : (P,%) -> % if $ has finiteAggregate and P has SETCAT
--R removeDuplicates : % -> % if $ has finiteAggregate and P has SETCAT
--R retractIfCan : List(P) -> Union(%, "failed")
--R rewriteIdealWithHeadRemainder : (List(P),%) -> List(P) if R has INTDOM
--R rewriteIdealWithRemainder : (List(P),%) -> List(P) if R has INTDOM
--R roughBase? : % -> Boolean if R has INTDOM
--R roughEqualIdeals? : (%,%) -> Boolean if R has INTDOM
--R roughSubIdeal? : (%,%) -> Boolean if R has INTDOM
--R roughUnitIdeal? : % -> Boolean if R has INTDOM
--R select : ((P -> Boolean),%) -> % if $ has finiteAggregate
--R size? : (%,NonNegativeInteger) -> Boolean
--R sort : (%,VarSet) -> Record(under: %,floor: %,upper: %)
--R triangular? : % -> Boolean if R has INTDOM
--R
--E 1

)spool
)lisp (bye)

```

— GeneralPolynomialSet.help —

```

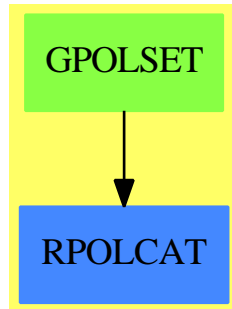
=====
GeneralPolynomialSet examples
=====

```

A domain for polynomial sets.

See Also:

o)show GeneralPolynomialSet

GeneralPolynomialSet (GPOLSET)**Exports:**

any?	coerce
collect	collectUnder
collectUpper	construct
convert	copy
count	empty
empty?	eq?
eval	every?
find	hash
headRemainder	latex
less?	mainVariables
mainVariable?	map
map!	member?
members	more?
mvar	parts
reduce	remainder
remove	removeDuplicates
retract	retractIfCan
rewriteIdealWithHeadRemainder	rewriteIdealWithRemainder
roughBase?	roughEqualIdeals?
roughSubIdeal?	roughUnitIdeal?
sample	select
size?	sort
triangular?	trivialIdeal?
variables	#?
?=?	?~=?

— domain GPOLSET GeneralPolynomialSet —

```

)abbrev domain GPOLSET GeneralPolynomialSet
++ Author: Marc Moreno Maza
++ Date Created: 04/26/1994
++ Date Last Updated: 12/15/1998
  
```

```

++ Description:
++ A domain for polynomial sets.

GeneralPolynomialSet(R,E,VarSet,P) : Exports == Implementation where

R:Ring
VarSet:OrderedSet
E:OrderedAbelianMonoidSup
P:RecursivePolynomialCategory(R,E,VarSet)
LP ==> List P
PtoP ==> P -> P

Exports == PolynomialSetCategory(R,E,VarSet,P) with

  convert : LP -> $
    ++ \axiom{convert(lp)} returns the polynomial set whose members
    ++ are the polynomials of \axiom{lp}.

  finiteAggregate
  shallowlyMutable

Implementation == add

Rep := List P

construct lp ==
  (removeDuplicates(lp)$List(P))::$

copy ps ==
  construct(copy(members(ps))$LP)$

empty() ==
  []

parts ps ==
  ps pretend LP

map (f : PtoP, ps : $) : $ ==
  construct(map(f,members(ps))$LP)$

map! (f : PtoP, ps : $) : $ ==
  construct(map!(f,members(ps))$LP)$

member? (p,ps) ==
  member?(p,members(ps))$LP

ps1 = ps2 ==
  {p for p in parts(ps1)} =$(Set P) {p for p in parts(ps2)}

coerce(ps:$) : OutputForm ==

```

```

lp : List(P) := sort(infRittWu?,members(ps))$(List P)
brace([p::OutputForm for p in lp]$List(OutputForm))$OutputForm

mvar ps ==
  empty? ps => error"Error from GPOLSET in mvar : #1 is empty"
  lv : List VarSet := variables(ps)
  empty? lv =>
    error "Error from GPOLSET in mvar : every polynomial in #1 is constant"
  reduce(max,lv)$(List VarSet)

retractIfCan(lp) ==
  (construct(lp))::Union($,"failed")

coerce(ps:$) : (List P) ==
  ps pretend (List P)

convert(lp:LP) : $ ==
  construct lp

```

— GPOLSET.dotabb —

```

"GPOLSET" [color="#88FF44",href="bookvol10.3.pdf#nameddest=GPOLSET"]
"RPOLCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RPOLCAT"]
"GPOLSET" -> "RPOLCAT"

```

domain GSTBL GeneralSparseTable

— GeneralSparseTable.input —

```

)set break resume
)sys rm -f GeneralSparseTable.output
)spool GeneralSparseTable.output
)set message test on
)set message auto off
)set break resume
)clear all
--S 1 of 7
patrons: GeneralSparseTable(String, Integer, KeyedAccessFile(Integer), 0) := table() ;
--E 1

```

```

--S 2 of 7
patrons."Smith" := 10500
--E 2

--S 3 of 7
patrons."Jones" := 22000
--E 3

--S 4 of 7
patrons."Jones"
--E 4

--S 5 of 7
patrons."Stingy"
--E 5

--S 6 of 7
reduce(+, entries patrons)
--E 6

--S 7 of 7
)system rm -r kaf*.sdata
--E 7
)spool
)lisp (bye)

```

— GeneralSparseTable.help —

```

=====
GeneralSparseTable
=====

```

Sometimes when working with tables there is a natural value to use as the entry in all but a few cases. The GeneralSparseTable constructor can be used to provide any table type with a default value for entries.

Suppose we launched a fund-raising campaign to raise fifty thousand dollars. To record the contributions, we want a table with strings as keys (for the names) and integer entries (for the amount). In a data base of cash contributions, unless someone has been explicitly entered, it is reasonable to assume they have made a zero dollar contribution.

This creates a keyed access file with default entry 0.

```
patrons: GeneralSparseTable(String, Integer, KeyedAccessFile(Integer), 0) := table() ;
```


Now patrons can be used just as any other table. Here we record two gifts.

```
patrons."Smith" := 10500
```

```
patrons."Jones" := 22000
```

Now let us look up the size of the contributions from Jones and Stingy.

```
patrons."Jones"
```

```
patrons."Stingy"
```

Have we met our seventy thousand dollar goal?

```
reduce(+, entries patrons)
```

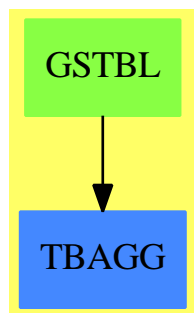
So the project is cancelled and we can delete the data base:

```
)system rm -r kaf*.sdata
```

See Also:

- o)show GeneralSparseTable

GeneralSparseTable (GSTBL)



See

- ⇒ "HashTable" (HASHTBL) [9](#) on page [1120](#)
- ⇒ "InnerTable" (INTABL) [10](#) on page [1335](#)
- ⇒ "Table" (TABLE) [21](#) on page [2663](#)
- ⇒ "EqTable" (EQTBL) [6](#) on page [698](#)
- ⇒ "StringTable" (STRTBL) [20](#) on page [2610](#)

⇒ “SparseTable” (STBL) [20](#) on page [2450](#)

Exports:

any?	bag	coerce	construct	convert
copy	count	dictionary	elt	empty
empty?	entries	entry?	eq?	eval
every?	extract!	fill!	find	first
hash	index?	indices	insert!	inspect
key?	keys	latex	less?	map
map!	maxIndex	member?	members	minIndex
more?	parts	qelt	qsetelt!	reduce
remove	remove!	removeDuplicates	sample	search
select	select!	setelt	size?	swap!
table	#?	?=?	?~=?	?..?

— domain GSTBL GeneralSparseTable —

```

)abbrev domain GSTBL GeneralSparseTable
++ Author: Stephen M. Watt
++ Date Created: 1986
++ Date Last Updated: June 21, 1991
++ Description:
++ A sparse table has a default entry, which is returned if no other
++ value has been explicitly stored for a key.

GeneralSparseTable(Key, Entry, Tbl, dent): TableAggregate(Key, Entry) == Impl
  where
    Key, Entry: SetCategory
    Tbl: TableAggregate(Key, Entry)
    dent: Entry

    Impl ==> Tbl add
      Rep := Tbl

      elt(t:%, k:Key) ==
        (u := search(k, t)$Rep) case "failed" => dent
        u::Entry

      setelt(t:%, k:Key, e:Entry) ==
        e = dent => (remove_!(k, t); e)
        setelt(t, k, e)$Rep

      search(k:Key, t:%) ==
        (u := search(k, t)$Rep) case "failed" => dent
        u

```

— GSTBL.dotabb —

```
"GSTBL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=GSTBL"]
"TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]
"GSTBL" -> "TBAGG"
```

domain GTSET GeneralTriangularSet

— GeneralTriangularSet.input —

```
)set break resume
)sys rm -f GeneralTriangularSet.output
)spool GeneralTriangularSet.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GeneralTriangularSet
--R
--R GeneralTriangularSet(R: IntegralDomain,E: OrderedAbelianMonoidSup,V: OrderedSet,P: Recursive
--R Abbreviation for GeneralTriangularSet is GTSET
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for GTSET
--R
--R----- Operations -----
--R ?=? : (%,%) -> Boolean          algebraic? : (V,%) -> Boolean
--R algebraicVariables : % -> List(V)  coerce : % -> List(P)
--R coerce : % -> OutputForm          collect : (%,V) -> %
--R collectQuasiMonic : % -> %        collectUnder : (%,V) -> %
--R collectUpper : (%,V) -> %        construct : List(P) -> %
--R copy : % -> %                    degree : % -> NonNegativeInteger
--R empty : () -> %                  empty? : % -> Boolean
--R eq? : (%,%) -> Boolean           extend : (%,P) -> %
--R first : % -> Union(P,"failed")    hash : % -> SingleInteger
--R headReduce : (P,%) -> P           headReduced? : % -> Boolean
--R headReduced? : (P,%) -> Boolean  infRittWu? : (%,%) -> Boolean
--R initiallyReduce : (P,%) -> P      initiallyReduced? : % -> Boolean
--R initials : % -> List(P)          last : % -> Union(P,"failed")
--R latex : % -> String              mainVariable? : (V,%) -> Boolean
--R mainVariables : % -> List(V)      map : ((P -> P),%) -> %
--R mvar : % -> V                    normalized? : % -> Boolean
--R normalized? : (P,%) -> Boolean    reduceByQuasiMonic : (P,%) -> P
```

```

--R removeZero : (P,%) -> P                rest : % -> Union(%, "failed")
--R retract : List(P) -> %                  sample : () -> %
--R stronglyReduce : (P,%) -> P              stronglyReduced? : % -> Boolean
--R trivialIdeal? : % -> Boolean             variables : % -> List(V)
--R zeroSetSplit : List(P) -> List(%)       ~=?: (% , %) -> Boolean
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R any? : ((P -> Boolean), %) -> Boolean if $ has finiteAggregate
--R autoReduced? : (% , ((P, List(P)) -> Boolean)) -> Boolean
--R basicSet : (List(P), (P -> Boolean), ((P, P) -> Boolean)) -> Union(Record(bas: %, top: List(P)), "failed")
--R basicSet : (List(P), ((P, P) -> Boolean)) -> Union(Record(bas: %, top: List(P)), "failed")
--R coHeight : % -> NonNegativeInteger if V has FINITE
--R convert : % -> InputForm if P has KONVERT(INFORM)
--R count : ((P -> Boolean), %) -> NonNegativeInteger if $ has finiteAggregate
--R count : (P, %) -> NonNegativeInteger if $ has finiteAggregate and P has SETCAT
--R eval : (% , List(Equation(P))) -> % if P has EVALAB(P) and P has SETCAT
--R eval : (% , Equation(P)) -> % if P has EVALAB(P) and P has SETCAT
--R eval : (% , P, P) -> % if P has EVALAB(P) and P has SETCAT
--R eval : (% , List(P), List(P)) -> % if P has EVALAB(P) and P has SETCAT
--R every? : ((P -> Boolean), %) -> Boolean if $ has finiteAggregate
--R extendIfCan : (% , P) -> Union(%, "failed")
--R find : ((P -> Boolean), %) -> Union(P, "failed")
--R headRemainder : (P, %) -> Record(num: P, den: R) if R has INTDOM
--R initiallyReduced? : (P, %) -> Boolean
--R less? : (% , NonNegativeInteger) -> Boolean
--R map! : ((P -> P), %) -> % if $ has shallowlyMutable
--R member? : (P, %) -> Boolean if $ has finiteAggregate and P has SETCAT
--R members : % -> List(P) if $ has finiteAggregate
--R more? : (% , NonNegativeInteger) -> Boolean
--R parts : % -> List(P) if $ has finiteAggregate
--R quasiComponent : % -> Record(close: List(P), open: List(P))
--R reduce : (P, %, ((P, P) -> P), ((P, P) -> Boolean)) -> P
--R reduce : (((P, P) -> P), %) -> P if $ has finiteAggregate
--R reduce : (((P, P) -> P), %, P) -> P if $ has finiteAggregate
--R reduce : (((P, P) -> P), %, P, P) -> P if $ has finiteAggregate and P has SETCAT
--R reduced? : (P, %, ((P, P) -> Boolean)) -> Boolean
--R remainder : (P, %) -> Record(rnum: R, polnum: P, den: R) if R has INTDOM
--R remove : ((P -> Boolean), %) -> % if $ has finiteAggregate
--R remove : (P, %) -> % if $ has finiteAggregate and P has SETCAT
--R removeDuplicates : % -> % if $ has finiteAggregate and P has SETCAT
--R retractIfCan : List(P) -> Union(%, "failed")
--R rewriteIdealWithHeadRemainder : (List(P), %) -> List(P) if R has INTDOM
--R rewriteIdealWithRemainder : (List(P), %) -> List(P) if R has INTDOM
--R rewriteSetWithReduction : (List(P), %, ((P, P) -> P), ((P, P) -> Boolean)) -> List(P)
--R roughBase? : % -> Boolean if R has INTDOM
--R roughEqualIdeals? : (% , %) -> Boolean if R has INTDOM
--R roughSubIdeal? : (% , %) -> Boolean if R has INTDOM
--R roughUnitIdeal? : % -> Boolean if R has INTDOM
--R select : (% , V) -> Union(P, "failed")
--R select : ((P -> Boolean), %) -> % if $ has finiteAggregate
--R size? : (% , NonNegativeInteger) -> Boolean

```

```

--R sort : (% ,V) -> Record(under: %,floor: %,upper: %)
--R stronglyReduced? : (P,%) -> Boolean
--R triangular? : % -> Boolean if R has INTDOM
--R zeroSetSplitIntoTriangularSystems : List(P) -> List(Record(close: %,open: List(P)))
--R
--E 1

)spool
)lisp (bye)

```

— GeneralTriangularSet.help —

=====

GeneralTriangularSet examples

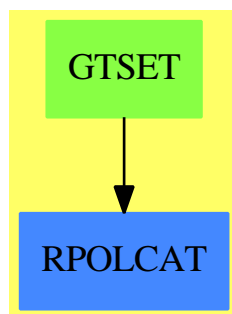
=====

A domain constructor of the category TriangularSetCategory. The only requirement for a list of polynomials to be a member of such a domain is the following: no polynomial is constant and two distinct polynomials have distinct main variables. Such a triangular set may not be auto-reduced or consistent. Triangular sets are stored as sorted lists w.r.t. the main variables of their members but they are displayed in reverse order.

See Also:

o)show GeneralTriangularSet

GeneralTriangularSet (GTSET)



See

⇒ “WuWenTsunTriangularSet” (WUTSET) [24](#) on page [2958](#)

Exports:

algebraic?	algebraicVariables
any?	autoReduced?
basicSet	coerce
collect	collectQuasiMonic
collectUnder	collectUpper
coHeight	construct
convert	copy
count	degree
empty	empty?
eq?	eval
every?	extend
extendIfCan	find
first	hash
headReduce	headReduced?
headReduced?	headRemainder
infRittWu?	initiallyReduce
initiallyReduced?	initials
last	latex
less?	mainVariable?
mainVariables	map
map!	member?
members	more?
mvar	normalized?
normalized?	parts
quasiComponent	reduce
reduceByQuasiMonic	reduced?
remainder	remove
removeDuplicates	removeZero
rest	retract
retractIfCan	rewriteIdealWithHeadRemainder
rewriteIdealWithRemainder	rewriteSetWithReduction
roughBase?	roughEqualIdeals?
roughSubIdeal?	roughUnitIdeal?
sample	select
size?	sort
stronglyReduce	stronglyReduced?
triangular?	trivialIdeal?
variables	zeroSetSplit
zeroSetSplitIntoTriangularSystems	#?
?=?	?~=?

— domain GTSET GeneralTriangularSet —

```
)abbrev domain GTSET GeneralTriangularSet
++ Author: Marc Moreno Maza (marc@nag.co.uk)
```

```

++ Date Created: 10/06/1995
++ Date Last Updated: 06/12/1996
++ References :
++ [1] P. AUBRY, D. LAZARD and M. MORENO MAZA "On the Theories
++      of Triangular Sets" Journal of Symbol. Comp. (to appear)
++ Description:
++ A domain constructor of the category \axiomType{TriangularSetCategory}.
++ The only requirement for a list of polynomials to be a member of such
++ a domain is the following: no polynomial is constant and two distinct
++ polynomials have distinct main variables. Such a triangular set may
++ not be auto-reduced or consistent. Triangular sets are stored
++ as sorted lists w.r.t. the main variables of their members but they
++ are displayed in reverse order.

```

```

GeneralTriangularSet(R,E,V,P) : Exports == Implementation where

```

```

R : IntegralDomain
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
LP ==> List P
PtoP ==> P -> P

```

```

Exports == TriangularSetCategory(R,E,V,P)

```

```

Implementation == add

```

```

Rep ==> LP

rep(s:$):Rep == s pretend Rep
per(l:Rep):$ == l pretend $

copy ts ==
  per(copy(rep(ts))$LP)
empty() ==
  per([])
empty?(ts:$) ==
  empty?(rep(ts))
parts ts ==
  rep(ts)
members ts ==
  rep(ts)
map (f : PtoP, ts : $) : $ ==
  construct(map(f,rep(ts))$LP)$
map! (f : PtoP, ts : $) : $ ==
  construct(map!(f,rep(ts))$LP)$
member? (p,ts) ==

```

```

member?(p,rep(ts))$LP

unitIdealIfCan() ==
  "failed"::Union($,"failed")
roughUnitIdeal? ts ==
  false

-- the following assume that rep(ts) is decreasingly sorted
-- w.r.t. the main variavles of the polynomials in rep(ts)
coerce(ts:$) : OutputForm ==
  lp : List(P) := reverse(rep(ts))
  brace([p::OutputForm for p in lp]$List(OutputForm))$OutputForm
mvar ts ==
  empty? ts => error"failed in mvar : $ -> V from GTSET"
  mvar(first(rep(ts)))$P
first ts ==
  empty? ts => "failed"::Union(P,"failed")
  first(rep(ts))::Union(P,"failed")
last ts ==
  empty? ts => "failed"::Union(P,"failed")
  last(rep(ts))::Union(P,"failed")
rest ts ==
  empty? ts => "failed"::Union($,"failed")
  per(rest(rep(ts)))::Union($,"failed")
coerce(ts:$) : (List P) ==
  rep(ts)
collectUpper (ts,v) ==
  empty? ts => ts
  lp := rep(ts)
  newlp : Rep := []
  while (not empty? lp) and (mvar(first(lp)) > v) repeat
    newlp := cons(first(lp),newlp)
    lp := rest lp
  per(reverse(newlp))
collectUnder (ts,v) ==
  empty? ts => ts
  lp := rep(ts)
  while (not empty? lp) and (mvar(first(lp)) >= v) repeat
    lp := rest lp
  per(lp)

-- for another domain of TSETCAT build on this domain GTSET
-- the following operations must be redefined
extendIfCan(ts:$,p:P) ==
  ground? p => "failed"::Union($,"failed")
  empty? ts => (per([unitCanonical(p)]$LP))::Union($,"failed")
  not (mvar(ts) < mvar(p)) => "failed"::Union($,"failed")
  (per(cons(p,rep(ts))))::Union($,"failed")

```

— GTSET.dotabb —

```
"GTSET" [color="#88FF44",href="bookvol10.3.pdf#nameddest=GTSET"]
"RPOLCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RPOLCAT"]
"GTSET" -> "RPOLCAT"
```

domain GSERIES GeneralUnivariatePowerSeries

— GeneralUnivariatePowerSeries.input —

```
)set break resume
)sys rm -f GeneralUnivariatePowerSeries.output
)spool GeneralUnivariatePowerSeries.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GeneralUnivariatePowerSeries
--R
--R GeneralUnivariatePowerSeries(Coef: Ring,var: Symbol,cen: Coef) is a domain constructor
--R Abbreviation for GeneralUnivariatePowerSeries is GSERIES
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for GSERIES
--R
--R----- Operations -----
--R ?? : (Coef,%) -> %           ?? : (%,Coef) -> %
--R ?? : (%,%) -> %             ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R ?? : (%,%) -> %             ?-? : (%,%) -> %
--R -? : % -> %                 ?? : (%,%) -> Boolean
--R 1 : () -> %                 0 : () -> %
--R ^? : (%,PositiveInteger) -> %   center : % -> Coef
--R coerce : Variable(var) -> %     coerce : Integer -> %
--R coerce : % -> OutputForm        complete : % -> %
--R degree : % -> Fraction(Integer) hash : % -> SingleInteger
--R latex : % -> String             leadingCoefficient : % -> Coef
--R leadingMonomial : % -> %        map : ((Coef -> Coef),%) -> %
--R monomial? : % -> Boolean        one? : % -> Boolean
--R order : % -> Fraction(Integer) pole? : % -> Boolean
--R recip : % -> Union(%, "failed") reductum : % -> %
```

```

--R sample : () -> %                                variable : % -> Symbol
--R zero? : % -> Boolean                             ?~=? : (%,% ) -> Boolean
--R ?? : (% , Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (Fraction(Integer), %) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (NonNegativeInteger, %) -> %
--R ??? : (% , Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (% , %) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (% , Integer) -> % if Coef has FIELD
--R ??? : (% , NonNegativeInteger) -> %
--R ?/? : (% , %) -> % if Coef has FIELD
--R ?/? : (% , Coef) -> % if Coef has FIELD
--R D : % -> % if Coef has *: (Fraction(Integer), Coef) -> Coef
--R D : (% , NonNegativeInteger) -> % if Coef has *: (Fraction(Integer), Coef) -> Coef
--R D : (% , Symbol) -> % if Coef has *: (Fraction(Integer), Coef) -> Coef and Coef has PDRING(SYMBOL)
--R D : (% , List(Symbol)) -> % if Coef has *: (Fraction(Integer), Coef) -> Coef and Coef has PDRING(SYMBOL)
--R D : (% , Symbol, NonNegativeInteger) -> % if Coef has *: (Fraction(Integer), Coef) -> Coef and Coef has PDRING(SYMBOL)
--R D : (% , List(Symbol), List(NonNegativeInteger)) -> % if Coef has *: (Fraction(Integer), Coef) -> Coef and Coef has PDRING(SYMBOL)
--R ?? : (% , Integer) -> % if Coef has FIELD
--R ?? : (% , NonNegativeInteger) -> %
--R acos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acoth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R approximate : (% , Fraction(Integer)) -> Coef if Coef has **: (Coef, Fraction(Integer)) -> Coef and Coef has PDRING(SYMBOL)
--R asec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R associates? : (% , %) -> Boolean if Coef has INTDOM
--R atan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R atanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(% , "failed") if Coef has CHARNZ
--R coefficient : (% , Fraction(Integer)) -> Coef
--R coerce : % -> % if Coef has INTDOM
--R coerce : Fraction(Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R coerce : UnivariatePuisseuxSeries(Coef, var, cen) -> %
--R coerce : Coef -> % if Coef has COMRING
--R cos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R coth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R differentiate : (% , Variable(var)) -> %
--R differentiate : % -> % if Coef has *: (Fraction(Integer), Coef) -> Coef
--R differentiate : (% , NonNegativeInteger) -> % if Coef has *: (Fraction(Integer), Coef) -> Coef
--R differentiate : (% , Symbol) -> % if Coef has *: (Fraction(Integer), Coef) -> Coef and Coef has PDRING(SYMBOL)

```

```

--R differentiate : (% , List(Symbol)) -> % if Coef has *: (Fraction(Integer), Coef) -> Coef and Co
--R differentiate : (% , Symbol, NonNegativeInteger) -> % if Coef has *: (Fraction(Integer), Coef) -
--R differentiate : (% , List(Symbol), List(NonNegativeInteger)) -> % if Coef has *: (Fraction(Inte
--R divide : (% , %) -> Record(quotient: %, remainder: %) if Coef has FIELD
--R ?.? : (% , %) -> % if Fraction(Integer) has SGROUP
--R ?.? : (% , Fraction(Integer)) -> Coef
--R euclideanSize : % -> NonNegativeInteger if Coef has FIELD
--R eval : (% , Coef) -> Stream(Coef) if Coef has **: (Coef, Fraction(Integer)) -> Coef
--R exp : % -> % if Coef has ALGEBRA(FRAC(INT))
--R expressIdealMember : (List(%), %) -> Union(List(%), "failed") if Coef has FIELD
--R exquo : (% , %) -> Union(% , "failed") if Coef has INTDOM
--R extend : (% , Fraction(Integer)) -> %
--R extendedEuclidean : (% , %) -> Record(coef1: %, coef2: %, generator: %) if Coef has FIELD
--R extendedEuclidean : (% , %, %) -> Union(Record(coef1: %, coef2: %), "failed") if Coef has FIELD
--R factor : % -> Factored(%) if Coef has FIELD
--R gcd : (% , %) -> % if Coef has FIELD
--R gcd : List(%) -> % if Coef has FIELD
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUniva
--R integrate : (% , Variable(var)) -> % if Coef has ALGEBRA(FRAC(INT))
--R integrate : (% , Symbol) -> % if Coef has integrate: (Coef, Symbol) -> Coef and Coef has variab
--R integrate : % -> % if Coef has ALGEBRA(FRAC(INT))
--R inv : % -> % if Coef has FIELD
--R lcm : (% , %) -> % if Coef has FIELD
--R lcm : List(%) -> % if Coef has FIELD
--R log : % -> % if Coef has ALGEBRA(FRAC(INT))
--R monomial : (% , List(SingletonAsOrderedSet), List(Fraction(Integer))) -> %
--R monomial : (% , SingletonAsOrderedSet, Fraction(Integer)) -> %
--R monomial : (Coef, Fraction(Integer)) -> %
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed") if Coef has FIELD
--R multiplyExponents : (% , Fraction(Integer)) -> %
--R multiplyExponents : (% , PositiveInteger) -> %
--R nthRoot : (% , Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R order : (% , Fraction(Integer)) -> Fraction(Integer)
--R pi : () -> % if Coef has ALGEBRA(FRAC(INT))
--R prime? : % -> Boolean if Coef has FIELD
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %) if Coef has FIELD
--R ?quo? : (% , %) -> % if Coef has FIELD
--R ?rem? : (% , %) -> % if Coef has FIELD
--R sec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R series : (NonNegativeInteger, Stream(Record(k: Fraction(Integer), c: Coef))) -> %
--R sin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sizeLess? : (% , %) -> Boolean if Coef has FIELD
--R sqrt : % -> % if Coef has ALGEBRA(FRAC(INT))
--R squareFree : % -> Factored(%) if Coef has FIELD
--R squareFreePart : % -> % if Coef has FIELD
--R subtractIfCan : (% , %) -> Union(% , "failed")
--R tan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R tanh : % -> % if Coef has ALGEBRA(FRAC(INT))

```

```

--R terms : % -> Stream(Record(k: Fraction(Integer),c: Coef))
--R truncate : (% ,Fraction(Integer),Fraction(Integer)) -> %
--R truncate : (% ,Fraction(Integer)) -> %
--R unit? : % -> Boolean if Coef has INTDOM
--R unitCanonical : % -> % if Coef has INTDOM
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %) if Coef has INTDOM
--R variables : % -> List(SingletonAsOrderedSet)
--R
--E 1

)spool
)lisp (bye)

```

— GeneralUnivariatePowerSeries.help —

=====

GeneralUnivariatePowerSeries examples

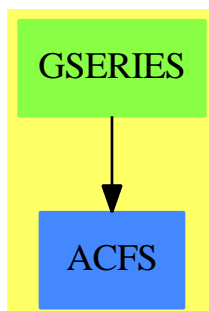
=====

This is a category of univariate Puiseux series constructed from univariate Laurent series. A Puiseux series is represented by a pair $[r, f(x)]$, where r is a positive rational number and $f(x)$ is a Laurent series. This pair represents the Puiseux series $f(x^r)$.

See Also:

o)show GeneralUnivariatePowerSeries

GeneralUnivariatePowerSeries (GSERIES)



Exports:

0	1	acos	acosh
acot	acoth	acsc	acsch
approximate	asec	asech	asin
asinh	associates?	atan	atanh
center	characteristic	charthRoot	coefficient
coerce	complete	cos	cosh
cot	coth	csc	csch
D	degree	differentiate	divide
euclideanSize	eval	exp	expressIdealMember
exquo	extend	extendedEuclidean	factor
gcd	gcdPolynomial	hash	integrate
inv	latex	lcm	leadingCoefficient
leadingMonomial	log	map	monomial
monomial?	multiEuclidean	multiplyExponents	nthRoot
one?	order	pi	pole?
prime?	principalIdeal	recip	reductum
sample	sec	sech	series
sin	sinh	sizeLess?	sqrt
squareFree	squareFreePart	subtractIfCan	tan
tanh	terms	truncate	unit?
unitCanonical	unitNormal	variable	variables
zero?	?+?	?-?	-?
?=?	?^?	?~=?	?*?
?**?	?/?	?..?	
?quo?	?rem?		

— domain GSERIES GeneralUnivariatePowerSeries —

```

)abbrev domain GSERIES GeneralUnivariatePowerSeries
++ Author: Clifton J. Williamson
++ Date Created: 22 September 1993
++ Date Last Updated: 23 September 1993
++ Description:
++ This is a category of univariate Puiseux series constructed
++ from univariate Laurent series. A Puiseux series is represented
++ by a pair \spad{[r,f(x)]}, where r is a positive rational number and
++ \spad{f(x)} is a Laurent series. This pair represents the Puiseux
++ series \spad{f(x^r)}.

GeneralUnivariatePowerSeries(Coef,var,cen): Exports == Implementation where
  Coef : Ring
  var : Symbol
  cen : Coef
  I ==> Integer
  UTS ==> UnivariateTaylorSeries
  ULS ==> UnivariateLaurentSeries
  UPXS ==> UnivariatePuisseuxSeries

```

```

EFULS ==> ElementaryFunctionsUnivariateLaurentSeries
EFUPXS ==> ElementaryFunctionsUnivariatePuisseuxSeries
FS2UPS ==> FunctionSpaceToUnivariatePowerSeries

Exports ==> UnivariatePuisseuxSeriesCategory Coef with
coerce: Variable(var) -> %
  ++ coerce(var) converts the series variable \spad{var} into a
  ++ Puisseux series.
coerce: UPXS(Coef,var,cen) -> %
  ++ coerce(f) converts a Puisseux series to a general power series.
differentiate: (% ,Variable(var)) -> %
  ++ \spad{differentiate(f(x),x)} returns the derivative of
  ++ \spad{f(x)} with respect to \spad{x}.
if Coef has Algebra Fraction Integer then
  integrate: (% ,Variable(var)) -> %
    ++ \spad{integrate(f(x))} returns an anti-derivative of the power
    ++ series \spad{f(x)} with constant coefficient 0.
    ++ We may integrate a series when we can divide coefficients
    ++ by integers.

Implementation ==> UnivariatePuisseuxSeries(Coef,var,cen) add

coerce(upxs:UPXS(Coef,var,cen)) == upxs pretend %

puisseux: % -> UPXS(Coef,var,cen)
puisseux f == f pretend UPXS(Coef,var,cen)

if Coef has Algebra Fraction Integer then

  differentiate f ==
    str1 : String := "'differentiate' unavailable on this domain; "
    str2 : String := "use 'approximate' first"
    error concat(str1,str2)

  differentiate(f:% ,v:Variable(var)) == differentiate f

if Coef has PartialDifferentialRing(Symbol) then
  differentiate(f:% ,s:Symbol) ==
    (s = variable(f)) =>
      str1 : String := "'differentiate' unavailable on this domain; "
      str2 : String := "use 'approximate' first"
      error concat(str1,str2)
    dcds := differentiate(center f,s)
    deriv := differentiate(puisseux f) :: %
    map(x+>differentiate(x,s),f) - dcds * deriv

  integrate f ==
    str1 : String := "'integrate' unavailable on this domain; "
    str2 : String := "use 'approximate' first"
    error concat(str1,str2)

```

```

integrate(f:%,v:Variable(var)) == integrate f

if Coef has integrate: (Coef,Symbol) -> Coef and _
  Coef has variables: Coef -> List Symbol then

integrate(f:%,s:Symbol) ==
  (s = variable(f)) =>
    str1 : String := "'integrate' unavailable on this domain; "
    str2 : String := "use 'approximate' first"
    error concat(str1,str2)
  not entry?(s,variables center f) => map(x+>integrate(x,s),f)
  error "integrate: center is a function of variable of integration"

if Coef has TranscendentalFunctionCategory and _
  Coef has PrimitiveFunctionCategory and _
  Coef has AlgebraicallyClosedFunctionSpace Integer then

integrateWithOneAnswer: (Coef,Symbol) -> Coef
integrateWithOneAnswer(f,s) ==
  res := integrate(f,s)$FunctionSpaceIntegration(Integer,Coef)
  res case Coef => res :: Coef
  first(res :: List Coef)

integrate(f:%,s:Symbol) ==
  (s = variable(f)) =>
    str1 : String := "'integrate' unavailable on this domain; "
    str2 : String := "use 'approximate' first"
    error concat(str1,str2)
  not entry?(s,variables center f) =>
    map(x+>integrateWithOneAnswer(x,s),f)
  error "integrate: center is a function of variable of integration"

```

— GSERIES.dotabb —

```

"GSERIES" [color="#88FF44",href="bookvol10.3.pdf#nameddest=GSERIES"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"GSERIES" -> "ACFS"

```

domain GRIMAGE GraphImage

— GraphImage.input —

```
)set break resume
)sys rm -f GraphImage.output
)spool GraphImage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GraphImage
--R
--R GraphImage is a domain constructor
--R Abbreviation for GraphImage is GRIMAGE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for GRIMAGE
--R
--R----- Operations -----
--R ==? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R graphImage : () -> %             hash : % -> SingleInteger
--R key : % -> Integer               latex : % -> String
--R makeGraphImage : % -> %          units : % -> List(Float)
--R ~=? : (%,% ) -> Boolean
--R appendPoint : (% ,Point(DoubleFloat)) -> Void
--R coerce : List(List(Point(DoubleFloat))) -> %
--R component : (% ,Point(DoubleFloat),Palette,Palette,PositiveInteger) -> Void
--R component : (% ,Point(DoubleFloat)) -> Void
--R component : (% ,List(Point(DoubleFloat)),Palette,Palette,PositiveInteger) -> Void
--R figureUnits : List(List(Point(DoubleFloat))) -> List(DoubleFloat)
--R makeGraphImage : (List(List(Point(DoubleFloat))),List(Palette),List(Palette),List(PositiveInteger),List
--R makeGraphImage : (List(List(Point(DoubleFloat))),List(Palette),List(Palette),List(PositiveInteger)) -> %
--R makeGraphImage : List(List(Point(DoubleFloat))) -> %
--R point : (% ,Point(DoubleFloat),Palette) -> Void
--R pointLists : % -> List(List(Point(DoubleFloat)))
--R putColorInfo : (List(List(Point(DoubleFloat))),List(Palette)) -> List(List(Point(DoubleFloat)))
--R ranges : (% ,List(Segment(Float))) -> List(Segment(Float))
--R ranges : % -> List(Segment(Float))
--R units : (% ,List(Float)) -> List(Float)
--R
--E 1

)spool
)lisp (bye)
```


— GraphImage.help —

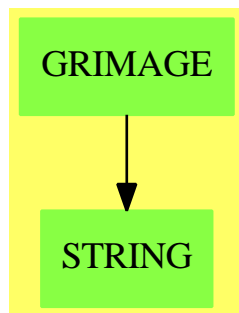
```
=====
GraphImage examples
=====
```

```
TwoDimensionalGraph creates virtual two dimensional graphs
(to be displayed on TwoDimensionalViewports).
```

```
See Also:
```

```
o )show GraphImage
```

GraphImage (GRIMAGE)

**Exports:**

appendPoint	coerce	component	figureUnits	graphImage
hash	key	latex	makeGraphImage	point
pointLists	putColorInfo	ranges	units	?~=?
?=?				

— domain GRIMAGE GraphImage —

```
)abbrev domain GRIMAGE GraphImage
++ Author: Jim Wen
++ Date Created: 27 April 1989
++ Date Last Updated: 1995 September 20, Mike Richardson (MGR)
++ Description:
++ TwoDimensionalGraph creates virtual two dimensional graphs
++ (to be displayed on TwoDimensionalViewports).
```

```
GraphImage (): Exports == Implementation where
```

```

VIEW    ==> VIEWPORTSERVER$Lisp
sendI   ==> SOCK_-SEND_-INT
sendSF  ==> SOCK_-SEND_-FLOAT
sendSTR ==> SOCK_-SEND_-STRING
getI    ==> SOCK_-GET_-INT
getSF   ==> SOCK_-GET_-FLOAT

typeGRAPH ==> 2
typeVIEW2D ==> 3

makeGRAPH ==> (-1)$SingleInteger
makeVIEW2D ==> (-1)$SingleInteger

I    ==> Integer
PI   ==> PositiveInteger
NNI  ==> NonNegativeInteger
SF   ==> DoubleFloat
F    ==> Float
L    ==> List
P    ==> Point(SF)
V    ==> Vector
SEG  ==> Segment
RANGESF ==> L SEG SF
RANGEF  ==> L SEG F
UNITSF  ==> L SF
UNITF   ==> L F
PAL     ==> Palette
E       ==> OutputForm
DROP    ==> DrawOption
PP      ==> PointPackage(SF)
COORDSYS ==> CoordinateSystems(SF)

Exports ==> SetCategory with
graphImage : () -> $
++ graphImage() returns an empty graph with 0 point lists
++ of the domain \spadtype{GraphImage}. A graph image contains
++ the graph data component of a two dimensional viewport.
makeGraphImage : $ -> $
++ makeGraphImage(gi) takes the given graph, \spad{gi} of the
++ domain \spadtype{GraphImage}, and sends it's data to the
++ viewport manager where it waits to be included in a two-dimensional
++ viewport window. \spad{gi} cannot be an empty graph, and it's
++ elements must have been created using the \spadfun{point} or
++ \spadfun{component} functions, not by a previous
++ \spadfun{makeGraphImage}.
makeGraphImage : (L L P) -> $
++ makeGraphImage(llp) returns a graph of the domain
++ \spadtype{GraphImage} which is composed of the points and
++ lines from the list of lists of points, \spad{llp}, with
++ default point size and default point and line colours. The graph

```

```

++ data is then sent to the viewport manager where it waits to be
++ included in a two-dimensional viewport window.
makeGraphImage : (L L P,L PAL,L PAL,L PI) -> $
++ makeGraphImage(llp,lpal1,lpal2,lp) returns a graph of the
++ domain \spadtype{GraphImage} which is composed of the points
++ and lines from the list of lists of points, \spad{llp}, whose
++ point colors are indicated by the list of palette colors,
++ \spad{lpal1}, and whose lines are colored according to the list
++ of palette colors, \spad{lpal2}. The parameter lp is a list of
++ integers which denote the size of the data points. The graph
++ data is then sent to the viewport manager where it waits to be
++ included in a two-dimensional viewport window.
makeGraphImage : (L L P,L PAL,L PAL,L PI,L DROP) -> $
++ makeGraphImage(llp,lpal1,lpal2,lp,lopt) returns a graph of
++ the domain \spadtype{GraphImage} which is composed of the
++ points and lines from the list of lists of points, \spad{llp},
++ whose point colors are indicated by the list of palette colors,
++ \spad{lpal1}, and whose lines are colored according to the list
++ of palette colors, \spad{lpal2}. The parameter lp is a list of
++ integers which denote the size of the data points, and \spad{lopt}
++ is the list of draw command options. The graph data is then sent
++ to the viewport manager where it waits to be included in a
++ two-dimensional viewport window.
pointLists : $ -> L L P
++ pointLists(gi) returns the list of lists of points which compose
++ the given graph, \spad{gi}, of the domain \spadtype{GraphImage}.
key : $ -> I
++ key(gi) returns the process ID of the given graph, \spad{gi},
++ of the domain \spadtype{GraphImage}.
ranges : $ -> RANGEF
++ ranges(gi) returns the list of ranges of the point components from
++ the indicated graph, \spad{gi}, of the domain \spadtype{GraphImage}.
ranges : ($,RANGEF) -> RANGEF
++ ranges(gi,lr) modifies the list of ranges for the given graph,
++ \spad{gi} of the domain \spadtype{GraphImage}, to be that of the
++ list of range segments, \spad{lr}, and returns the new range list
++ for \spad{gi}.
units : $ -> UNITF
++ units(gi) returns the list of unit increments for the x and y
++ axes of the indicated graph, \spad{gi}, of the domain
++ \spadtype{GraphImage}.
units : ($,UNITF) -> UNITF
++ units(gi,lu) modifies the list of unit increments for the x and y
++ axes of the given graph, \spad{gi} of the domain
++ \spadtype{GraphImage}, to be that of the list of unit increments,
++ \spad{lu}, and returns the new list of units for \spad{gi}.
component : ($,L P,PAL,PAL,PI) -> Void
++ component(gi,lp,pal1,pal2,p) sets the components of the
++ graph, \spad{gi} of the domain \spadtype{GraphImage}, to the
++ values given. The point list for \spad{gi} is set to the list

```

```

++ \spad{lp}, the color of the points in \spad{lp} is set to
++ the palette color \spad{pal1}, the color of the lines which
++ connect the points \spad{lp} is set to the palette color
++ \spad{pal2}, and the size of the points in \spad{lp} is given
++ by the integer p.
component      : ($,P)                                -> Void
++ component(gi,pt) modifies the graph \spad{gi} of the domain
++ \spadtype{GraphImage} to contain one point component, \spad{pt}
++ whose point color, line color and point size are determined by
++ the default functions \spadfun{pointColorDefault},
++ \spadfun{lineColorDefault}, and \spadfun{pointSizeDefault}.
component      : ($,P,PAL,PAL,PI)                    -> Void
++ component(gi,pt,pal1,pal2,ps) modifies the graph \spad{gi} of
++ the domain \spadtype{GraphImage} to contain one point component,
++ \spad{pt} whose point color is set to the palette color \spad{pal1},
++ line color is set to the palette color \spad{pal2}, and point
++ size is set to the positive integer \spad{ps}.
appendPoint    : ($,P)                                -> Void
++ appendPoint(gi,pt) appends the point \spad{pt} to the end
++ of the list of points component for the graph, \spad{gi}, which is
++ of the domain \spadtype{GraphImage}.
point          : ($,P,PAL)                            -> Void
++ point(gi,pt,pal) modifies the graph \spad{gi} of the domain
++ \spadtype{GraphImage} to contain one point component, \spad{pt}
++ whose point color is set to be the palette color \spad{pal}, and
++ whose line color and point size are determined by the default
++ functions \spadfun{lineColorDefault} and \spadfun{pointSizeDefault}.
coerce         : L L P                                -> $
++ coerce(llp)
++ component(gi,pt) creates and returns a graph of the domain
++ \spadtype{GraphImage} which is composed of the list of list
++ of points given by \spad{llp}, and whose point colors, line colors
++ and point sizes are determined by the default functions
++ \spadfun{pointColorDefault}, \spadfun{lineColorDefault}, and
++ \spadfun{pointSizeDefault}. The graph data is then sent to the
++ viewport manager where it waits to be included in a two-dimensional
++ viewport window.
coerce         : $                                    -> E
++ coerce(gi) returns the indicated graph, \spad{gi}, of domain
++ \spadtype{GraphImage} as output of the domain \spadtype{OutputForm}.
putColorInfo   : (L L P,L PAL)                        -> L L P
++ putColorInfo(llp,lpal) takes a list of list of points, \spad{llp},
++ and returns the points with their hue and shade components
++ set according to the list of palette colors, \spad{lpal}.
figureUnits : L L P                                -> UNITSF

Implementation ==> add
import Color()
import Palette()
import ViewDefaultsPackage()

```

```

import PlotTools()
import DrawOptionFunctions0
import P
import PP
import COORDSYS

Rep := Record(key: I, rangesField: RANGESF, unitsField: UNITSF, _
  llPoints: L L P, pointColors: L PAL, lineColors: L PAL, pointSizes: L PI, _
  optionsField: L DROP)

--%Internal Functions

graph      : RANGEF                                -> $
scaleStep  : SF                                    -> SF
makeGraph  : $                                     -> $

numberCheck(nums:Point SF):Void ==
  for i in minIndex(nums)..maxIndex(nums) repeat
    COMPLEXP(nums.(i::PositiveInteger))$Lisp =>
      error "An unexpected complex number was encountered in the calculations."

doOptions(g:Rep):Void ==
  lr : RANGEF := ranges(g.optionsField,ranges g)
  if (#lr > 1$I) then
    g.rangesField := [segment(convert(lo(lr.1))@SF,convert(hi(lr.1))@SF)$(Segment(SF)),
                      segment(convert(lo(lr.2))@SF,convert(hi(lr.2))@SF)$(Segment(SF))]
  else
    g.rangesField := []
  lu : UNITF := units(g.optionsField,units g)
  if (#lu > 1$I) then
    g.unitsField := [convert(lu.1)@SF,convert(lu.2)@SF]
  else
    g.unitsField := []
  -- etc - graphimage specific stuff...

putColorInfo(llp,listOfPalettes) ==
  llp2 : L L P := []
  for lp in llp for pal in listOfPalettes repeat
    lp2 : L P := []
    daHue   := (hue(hue pal))::SF
    daShade := (shade pal)::SF
    for p in lp repeat
      if (d := dimension p) < 3 then
        p := extend(p,[daHue,daShade])
      else
        p.3 := daHue
        d < 4 => p := extend(p,[daShade])
        p.4 := daShade

```

```

    lp2 := cons(p,lp2)
    llp2 := cons(reverse_! lp2,llp2)
    reverse_! llp2

graph demRanges ==
null demRanges => [ 0, [], [], [], [], [], [], [] ]
demRangesSF : RANGESF := _
[ segment(convert(lo demRanges.1)@SF,convert(hi demRanges.1)@SF)$(Segment(SF)), _
  segment(convert(lo demRanges.1)@SF,convert(hi demRanges.1)@SF)$(Segment(SF)) ]
[ 0, demRangesSF, [], [], [], [], [], [] ]

scaleStep(range) == -- MGR

adjust:NNI
tryStep:SF
scaleDown:SF
numerals:String
adjust := 0
while range < 100.0::SF repeat
  adjust := adjust + 1
  range := range * 10.0::SF -- might as well take big steps
tryStep := range/10.0::SF
numerals := string(((retract(ceiling(tryStep)$SF)$SF)@I))$String
scaleDown := (10@I **$I (((#(numerals)@I) - 1$I) pretend PI))::SF
scaleDown*ceiling(tryStep/scaleDown - 0.5::SF)/((10 **$I adjust)::SF)

figureUnits(listOfListsOfPoints) ==
-- figure out the min/max and divide by 10 for unit markers
xMin := xMax := xCoord first first listOfListsOfPoints
yMin := yMax := yCoord first first listOfListsOfPoints
if xMin ~= xMin then xMin:=max()
if xMax ~= xMax then xMax:=min()
if yMin ~= yMin then yMin:=max()
if yMax ~= yMax then yMax:=min()
for pL in listOfListsOfPoints repeat
  for p in pL repeat
    if ((px := (xCoord p)) < xMin) then
      xMin := px
    if px > xMax then
      xMax := px
    if ((py := (yCoord p)) < yMin) then
      yMin := py
    if py > yMax then
      yMax := py
if xMin = xMax then
  xMin := xMin - convert(0.5)$Float
  xMax := xMax + convert(0.5)$Float
if yMin = yMax then
  yMin := yMin - convert(0.5)$Float
  yMax := yMax + convert(0.5)$Float

```

```

[scaleStep(xMax-xMin),scaleStep(yMax-yMin)]

plotLists(graf:Rep,listOfListsOfPoints:L L P,listOfPointColors:L PAL,listOfLineColors:L PAL,
givenLen := #listOfListsOfPoints
-- take out point lists that are actually empty
listOfListsOfPoints := [ l for l in listOfListsOfPoints | ^null l ]
if (null listOfListsOfPoints) then
  error "GraphImage was given a list that contained no valid point lists"
if ((len := #listOfListsOfPoints) ^= givenLen) then
  sayBrightly([" Warning: Ignoring pointless point list":E]$List(E))$Lisp
graf.llPoints := listOfListsOfPoints
-- do point colors
if ((givenLen := #listOfPointColors) > len) then
  -- pad or discard elements if given list has length different from the point list
  graf.pointColors := concat(listOfPointColors,
    new((len - givenLen)::NonNegativeInteger + 1, pointColorDefault()))
else graf.pointColors := first(listOfPointColors, len)
-- do line colors
if ((givenLen := #listOfLineColors) > len) then
  graf.lineColors := concat(listOfLineColors,
    new((len - givenLen)::NonNegativeInteger + 1, lineColorDefault()))
else graf.lineColors := first(listOfLineColors, len)
-- do point sizes
if ((givenLen := #listOfPointSizes) > len) then
  graf.pointSizes := concat(listOfPointSizes,
    new((len - givenLen)::NonNegativeInteger + 1, pointSizeDefault()))
else graf.pointSizes := first(listOfPointSizes, len)
graf

makeGraph graf ==
doOptions(graf)
(s := #(graf.llPoints)) = 0 =>
  error "You are trying to make a graph with no points"
key graf ^= 0 =>
  error "You are trying to draw over an existing graph"
transform := coord(graf.optionsField,cartesian$COORDSYS)$DrawOptionFunctions0
graf.llPoints:= putColorInfo(graf.llPoints,graf.pointColors)
if null(ranges graf) then -- figure out best ranges for points
  graf.rangesField := calcRanges(graf.llPoints) --::V SEG SF
if null(units graf) then -- figure out best ranges for points
  graf.unitsField := figureUnits(graf.llPoints) --::V SEG SF
sayBrightly([" Graph data being transmitted to the viewport manager...":E]$List(E))$Lisp
sendI(VIEW,typeGRAPH)$Lisp
sendI(VIEW,makeGRAPH)$Lisp
tonto := (graf.rangesField)::RANGESF
sendSF(VIEW,lo(first tonto))$Lisp
sendSF(VIEW,hi(first tonto))$Lisp
sendSF(VIEW,lo(second tonto))$Lisp
sendSF(VIEW,hi(second tonto))$Lisp
sendSF(VIEW,first (graf.unitsField))$Lisp

```

```

sendSF(VIEW,second (graf.unitsField))$Lisp
sendI(VIEW,s)$Lisp      -- how many lists of points are being sent
for aList in graf.llPoints for pColor in graf.pointColors for lColor in graf.lineColors for s in graf
  sendI(VIEW,#aList)$Lisp -- how many points in this list
  for p in aList repeat
    aPoint := transform p
    sendSF(VIEW,xCoord aPoint)$Lisp
    sendSF(VIEW,yCoord aPoint)$Lisp
    sendSF(VIEW,hue(p)$PP)$Lisp -- ?use aPoint as well...?
    sendSF(VIEW,shade(p)$PP)$Lisp
    hueShade := hue hue pColor + shade pColor * numberOfHues()
    sendI(VIEW,hueShade)$Lisp
    hueShade := (hue hue lColor -1)*5 + shade lColor
    sendI(VIEW,hueShade)$Lisp
    sendI(VIEW,s)$Lisp
graf.key := getI(VIEW)$Lisp
graf

--%Exported Functions
makeGraphImage(graf:$) == makeGraph graf
key graf                == graf.key
pointLists graf         == graf.llPoints
ranges graf             ==
  null graf.rangesField => []
  [segment(convert(lo graf.rangesField.1)@F,convert(hi graf.rangesField.1)@F), _
   segment(convert(lo graf.rangesField.2)@F,convert(hi graf.rangesField.2)@F)]
ranges(graf,rangesList) ==
  graf.rangesField :=
    [segment(convert(lo rangesList.1)@SF,convert(hi rangesList.1)@SF), _
     segment(convert(lo rangesList.2)@SF,convert(hi rangesList.2)@SF)]
  rangesList
units graf              ==
  null(graf.unitsField) => []
  [convert(graf.unitsField.1)@F,convert(graf.unitsField.2)@F]
units (graf,unitsToBe) ==
  graf.unitsField := [convert(unitsToBe.1)@SF,convert(unitsToBe.2)@SF]
  unitsToBe
graphImage              == graph []

makeGraphImage(llp) ==
  makeGraphImage(llp,
    [pointColorDefault() for i in 1..(1:=#llp)],
    [lineColorDefault() for i in 1..1],
    [pointSizeDefault() for i in 1..1])

makeGraphImage(llp,lpc,llc,lps) ==
  makeGraphImage(llp,lpc,llc,lps,[])

makeGraphImage(llp,lpc,llc,lps,opts) ==

```



```

graf := graph(ranges(opts, []))
graf.optionsField := opts
graf := plotLists(graf, llp, lpc, llc, lps)
transform := coord(graf.optionsField, cartesian$COORDSYS)$DrawOptionFunctions0
for aList in graf.llPoints repeat
  for p in aList repeat
    aPoint := transform p
    numberCheck aPoint
makeGraph graf

component (graf:$, ListOfPoints:L P, PointColor:PAL, LineColor:PAL, PointSize:PI) ==
  graf.llPoints := append(graf.llPoints, [ListOfPoints])
  graf.pointColors := append(graf.pointColors, [PointColor])
  graf.lineColors := append(graf.lineColors, [LineColor])
  graf.pointSizes := append(graf.pointSizes, [PointSize])

component (graf,aPoint) ==
  component(graf,aPoint,pointColorDefault(),lineColorDefault(),pointSizeDefault())

component (graf:$,aPoint:P,PointColor:PAL,LineColor:PAL,PointSize:PI) ==
  component (graf,[aPoint],PointColor,LineColor,PointSize)

appendPoint (graf,aPoint) ==
  num : I := #(graf.llPoints) - 1
  num < 0 => error "No point lists to append to!"
  (graf.llPoints.num) := append((graf.llPoints.num), [aPoint])

point (graf,aPoint,PointColor) ==
  component(graf,aPoint,PointColor,lineColorDefault(),pointSizeDefault())

coerce (llp : L L P) : $ ==
  makeGraphImage(llp,
    [pointColorDefault() for i in 1..(1:=#llp)],
    [lineColorDefault() for i in 1..1],
    [pointSizeDefault() for i in 1..1])

coerce (graf : $) : E ==
  hconcat( ["Graph with " :: E, (p := # pointLists graf) :: E,
    (p=1 => " point list"; " point lists") :: E])

```

— GRIMAGE.dotabb —

```

"GRIMAGE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=GRIMAGE"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"GRIMAGE" -> "STRING"

```

domain GOPT GuessOption

— GuessOption.input —

```
)set break resume
)sys rm -f GuessOption.output
)spool GuessOption.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GuessOption
--R
--R GuessOption is a domain constructor
--R Abbreviation for GuessOption is GOPT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for GOPT
--R
--R----- Operations -----
--R ==? : (%,% ) -> Boolean          allDegrees : Boolean -> %
--R checkExtraValues : Boolean -> %   coerce : % -> OutputForm
--R debug : Boolean -> %               displayKind : Symbol -> %
--R functionName : Symbol -> %         functionNames : List(Symbol) -> %
--R hash : % -> SingleInteger          indexName : Symbol -> %
--R latex : % -> String                one : Boolean -> %
--R safety : NonNegativeInteger -> %   variableName : Symbol -> %
--R ~=? : (%,% ) -> Boolean
--R Somos : Union(PositiveInteger,Boolean) -> %
--R check : Union(skip,MonteCarlo,deterministic) -> %
--R homogeneous : Union(PositiveInteger,Boolean) -> %
--R maxDegree : Union(NonNegativeInteger,arbitrary) -> %
--R maxDerivative : Union(NonNegativeInteger,arbitrary) -> %
--R maxLevel : Union(NonNegativeInteger,arbitrary) -> %
--R maxMixedDegree : NonNegativeInteger -> %
--R maxPower : Union(PositiveInteger,arbitrary) -> %
--R maxShift : Union(NonNegativeInteger,arbitrary) -> %
--R maxSubst : Union(PositiveInteger,arbitrary) -> %
--R option : (List(%),Symbol) -> Union(Any,"failed")
--R
--E 1

)spool
)lisp (bye)
```

— **GuessOption.help** —

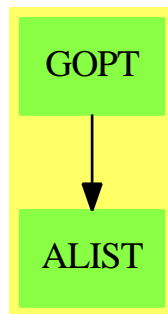
```
=====
GuessOption examples
=====
```

GuessOption is a domain whose elements are various options used by Guess.

See Also:

o)show GuessOption

GuessOption (GOPT)



Exports:

?=?	?~=?	Somos	allDegrees	check
checkExtraValues	coerce	debug	displayKind	functionName
functionNames	hash	homogeneous	indexName	latex
maxDegree	maxDerivative	maxLevel	maxMixedDegree	maxPower
maxShift	maxSubst	one	option	safety variableName

— **domain GOPT GuessOption** —

```
)abbrev domain GOPT GuessOption
++ Author: Martin Rubey
++ Description:
++ GuessOption is a domain whose elements are various options used
++ by Guess.
GuessOption(): Exports == Implementation where

Exports == SetCategory with
```

```

maxDerivative: Union(NonNegativeInteger, "arbitrary") -> %
++ maxDerivative(d) specifies the maximum derivative in an algebraic
++ differential equation. This option is expressed in the form
++ \spad{maxDerivative == d}.

maxShift: Union(NonNegativeInteger, "arbitrary") -> %
++ maxShift(d) specifies the maximum shift in a recurrence
++ equation. This option is expressed in the form \spad{maxShift == d}.

maxSubst: Union(PositiveInteger, "arbitrary") -> %
++ maxSubst(d) specifies the maximum degree of the monomial substituted
++ into the function we are looking for. That is, if \spad{maxSubst ==
++ d}, we look for polynomials such that  $\$p(f(x), f(x^2), \dots,$ 
++  $f(x^d))=0\$$ . equation. This option is expressed in the form
++ \spad{maxSubst == d}.

maxPower: Union(PositiveInteger, "arbitrary") -> %
++ maxPower(d) specifies the maximum degree in an algebraic differential
++ equation. For example, the degree of  $(f'')^3 f'$  is 4. maxPower(-1)
++ specifies that the maximum exponent can be arbitrary. This option is
++ expressed in the form \spad{maxPower == d}.

homogeneous: Union(PositiveInteger, Boolean) -> %
++ homogeneous(d) specifies whether we allow only homogeneous algebraic
++ differential equations. This option is expressed in the form
++ \spad{homogeneous == d}. If true, then maxPower must be
++ set, too, and ADEs with constant total degree are allowed.
++ If a PositiveInteger is given, only ADE's with this total degree are
++ allowed.

Somos: Union(PositiveInteger, Boolean) -> %
++ Somos(d) specifies whether we want that the total degree of the
++ differential operators is constant, and equal to d, or maxDerivative
++ if true. If true, maxDerivative must be set, too.

maxLevel: Union(NonNegativeInteger, "arbitrary") -> %
++ maxLevel(d) specifies the maximum number of recursion levels operators
++ guessProduct and guessSum will be applied. This option is expressed in
++ the form \spad{maxLevel == d}.

maxDegree: Union(NonNegativeInteger, "arbitrary") -> %
++ maxDegree(d) specifies the maximum degree of the coefficient
++ polynomials in an algebraic differential equation or a recursion with
++ polynomial coefficients. For rational functions with an exponential
++ term, \spad{maxDegree} bounds the degree of the denominator
++ polynomial.
++ This option is expressed in the form \spad{maxDegree == d}.

maxMixedDegree: NonNegativeInteger -> %

```

```

++ maxMixedDegree(d) specifies the maximum q-degree of the coefficient
++ polynomials in a recurrence with polynomial coefficients, in the case
++ of mixed shifts. Although slightly inconsistent, maxMixedDegree(0)
++ specifies that no mixed shifts are allowed. This option is expressed
++ in the form \spad{maxMixedDegree == d}.

allDegrees: Boolean -> %
++ allDegrees(d) specifies whether all possibilities of the degree vector
++ - taking into account maxDegree - should be tried. This is mainly
++ interesting for rational interpolation. This option is expressed in
++ the form \spad{allDegrees == d}.

safety: NonNegativeInteger -> %
++ safety(d) specifies the number of values reserved for testing any
++ solutions found. This option is expressed in the form \spad{safety ==
++ d}.

check: Union("skip", "MonteCarlo", "deterministic") -> %
++ check(d) specifies how we want to check the solution. If
++ the value is "skip", we return the solutions found by the
++ interpolation routine without checking. If the value is
++ "MonteCarlo", we use a probabilistic check. This option is
++ expressed in the form \spad{check == d}

checkExtraValues: Boolean -> %
++ checkExtraValues(d) specifies whether we want to check the
++ solution beyond the order given by the degree bounds. This
++ option is expressed in the form \spad{checkExtraValues == d}

one: Boolean -> %
++ one(d) specifies whether we are happy with one solution. This option
++ is expressed in the form \spad{one == d}.

debug: Boolean -> %
++ debug(d) specifies whether we want additional output on the
++ progress. This option is expressed in the form \spad{debug == d}.

functionName: Symbol -> %
++ functionName(d) specifies the name of the function given by the
++ algebraic differential equation or recurrence. This option is
++ expressed in the form \spad{functionName == d}.

functionNames: List(Symbol) -> %
++ functionNames(d) specifies the names for the function in
++ algebraic dependence. This option is
++ expressed in the form \spad{functionNames == d}.

variableName: Symbol -> %
++ variableName(d) specifies the variable used in by the algebraic
++ differential equation. This option is expressed in the form

```

```

++ \spad{variableName == d}.

indexName: Symbol -> %
++ indexName(d) specifies the index variable used for the formulas. This
++ option is expressed in the form \spad{indexName == d}.

displayKind: Symbol -> %
++ displayKind(d) specifies kind of the result: generating function,
++ recurrence or equation. This option should not be set by the
++ user, but rather by the HP-specification.

option : (List %, Symbol) -> Union(Any, "failed")
++ option(l, option) returns which options are given.

Implementation ==> add
import AnyFunctions1(Boolean)
import AnyFunctions1(Symbol)
import AnyFunctions1(NonNegativeInteger)
import AnyFunctions1(Union(NonNegativeInteger, "arbitrary"))
import AnyFunctions1(Union(PositiveInteger, "arbitrary"))
import AnyFunctions1(Union(PositiveInteger, Boolean))
import AnyFunctions1(Union("skip", "MonteCarlo", "deterministic"))

Rep := Record(keyword: Symbol, value: Any)

maxLevel d      == ['maxLevel,      d::Any]
maxDerivative d == ['maxDerivative, d::Any]
maxShift d      == maxDerivative d
maxSubst d      ==
    if d case PositiveInteger
    then maxDerivative((d::Integer-1)::NonNegativeInteger)
    else maxDerivative d
maxDegree d      == ['maxDegree,      d::Any]
maxMixedDegree d == ['maxMixedDegree, d::Any]
allDegrees d      == ['allDegrees,      d::Any]
maxPower d        == ['maxPower,      d::Any]
safety d          == ['safety,      d::Any]
homogeneous d     == ['homogeneous,    d::Any]
Somos d           == ['Somos,      d::Any]
debug d           == ['debug,      d::Any]
check d           == ['check,      d::Any]
checkExtraValues d == ['checkExtraValues, d::Any]
one d             == ['one,      d::Any]
functionName d    == ['functionName,    d::Any]
functionNames d ==
    ['functionNames, coerce(d)$AnyFunctions1(List(Symbol))]
variableName d    == ['variableName,    d::Any]
indexName d       == ['indexName,      d::Any]
displayKind d     == ['displayKind,     d::Any]

```

```
coerce(x: %): OutputForm == x.keyword::OutputForm = x.value::OutputForm
x: % = y: % == x.keyword = y.keyword and x.value = y.value
```

```
option(l, s) ==
  for x in l repeat
    x.keyword = s => return(x.value)
  "failed"
```

— GOPT.dotabb —

```
"GOPT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=GOPT"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"GOPT" -> "ALIST"
```

domain GOPT0 GuessOptionFunctions0

— GuessOptionFunctions0.input —

```
)set break resume
)sys rm -f GuessOptionFunctions0.output
)spool GuessOptionFunctions0.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GuessOptionFunctions0
--R
--R GuessOptionFunctions0 is a domain constructor
--R Abbreviation for GuessOptionFunctions0 is GOPT0
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for GOPT0
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger       latex : % -> String
--R one : List(GuessOption) -> Boolean  ?~=? : (%,% ) -> Boolean
--R Somos : List(GuessOption) -> Union(PositiveInteger,Boolean)
--R allDegrees : List(GuessOption) -> Boolean
--R check : List(GuessOption) -> Union(skip, MonteCarlo, deterministic)
```

```

--R checkExtraValues : List(GuessOption) -> Boolean
--R checkOptions : List(GuessOption) -> Void
--R debug : List(GuessOption) -> Boolean
--R displayAsGF : List(GuessOption) -> Boolean
--R functionName : List(GuessOption) -> Symbol
--R homogeneous : List(GuessOption) -> Union(PositiveInteger,Boolean)
--R indexName : List(GuessOption) -> Symbol
--R maxDegree : List(GuessOption) -> Union(NonNegativeInteger,arbitrary)
--R maxDerivative : List(GuessOption) -> Union(NonNegativeInteger,arbitrary)
--R maxLevel : List(GuessOption) -> Union(NonNegativeInteger,arbitrary)
--R maxMixedDegree : List(GuessOption) -> NonNegativeInteger
--R maxPower : List(GuessOption) -> Union(PositiveInteger,arbitrary)
--R maxShift : List(GuessOption) -> Union(NonNegativeInteger,arbitrary)
--R maxSubst : List(GuessOption) -> Union(PositiveInteger,arbitrary)
--R safety : List(GuessOption) -> NonNegativeInteger
--R variableName : List(GuessOption) -> Symbol
--R
--E 1

)spool
)lisp (bye)

```

— GuessOptionFunctions0.help —

```

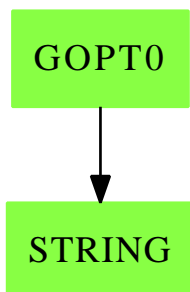
=====
GuessOptionFunctions0 examples
=====

```

GuessOptionFunctions0 provides operations that extract the values of options for Guess.

See Also:

o)show GuessOptionFunctions0

GuessOptionFunctions0 (GOPT0)**Exports:**

?=?	?~=?	MonteCarlo	Somos	allDegrees
check	checkOptions	coerce	debug	displayAsGF
functionName	hash	homogeneous	indexName	latex
maxDegree	maxDerivative	maxLevel	maxMixedDegree	maxPower
maxShift	maxSubst	one	safety	variableName

— domain GOPT0 GuessOptionFunctions0 —

```

)abbrev domain GOPT0 GuessOptionFunctions0
++ Author: Martin Rubey
++ Description:
++ GuessOptionFunctions0 provides operations that extract the
++ values of options for Guess.
GuessOptionFunctions0(): Exports == Implementation where

  LGOPT ==> List GuessOption

Exports == SetCategory with

  maxDerivative: LGOPT -> Union(NonNegativeInteger, "arbitrary")
    ++ maxDerivative returns the specified maxDerivative.

  maxShift: LGOPT -> Union(NonNegativeInteger, "arbitrary")
    ++ maxShift returns the specified maxShift.

  maxSubst: LGOPT -> Union(PositiveInteger, "arbitrary")
    ++ maxSubst returns the specified maxSubst.

  maxPower: LGOPT -> Union(PositiveInteger, "arbitrary")
    ++ maxPower returns the specified maxPower.

  homogeneous: LGOPT -> Union(PositiveInteger, Boolean)
    ++ homogeneous returns whether we allow only homogeneous algebraic
    ++ differential equations, default being false

```

```

Somos: LGOPT -> Union(PositiveInteger, Boolean)
  ++ Somos returns whether we allow only Somos-like operators, default
  ++ being false

maxLevel: LGOPT -> Union(NonNegativeInteger, "arbitrary")
  ++ maxLevel returns the specified maxLevel.

maxDegree: LGOPT -> Union(NonNegativeInteger, "arbitrary")
  ++ maxDegree returns the specified maxDegree.

maxMixedDegree: LGOPT -> NonNegativeInteger
  ++ maxMixedDegree returns the specified maxMixedDegree.

allDegrees: LGOPT -> Boolean
  ++ allDegrees returns whether all possibilities of the degree vector
  ++ should be tried, the default being false.

safety: LGOPT -> NonNegativeInteger
  ++ safety returns the specified safety or 1 as default.

check: LGOPT -> Union("skip", "MonteCarlo", "deterministic")
  ++ check(d) specifies how we want to check the solution. If
  ++ the value is "skip", we return the solutions found by the
  ++ interpolation routine without checking. If the value is
  ++ "MonteCarlo", we use a probabilistic check. The default is
  ++ "deterministic".

checkExtraValues: LGOPT -> Boolean
  ++ checkExtraValues(d) specifies whether we want to check the
  ++ solution beyond the order given by the degree bounds. The
  ++ default is true.

one: LGOPT -> Boolean
  ++ one returns whether we need only one solution, default being true.

functionName: LGOPT -> Symbol
  ++ functionName returns the name of the function given by the algebraic
  ++ differential equation, default being f

variableName: LGOPT -> Symbol
  ++ variableName returns the name of the variable used in by the
  ++ algebraic differential equation, default being x

indexName: LGOPT -> Symbol
  ++ indexName returns the name of the index variable used for the
  ++ formulas, default being n

displayAsGF: LGOPT -> Boolean
  ++ displayAsGF specifies whether the result is a generating function

```

```

++ or a recurrence. This option should not be set by the user, but rather
++ by the HP-specification, therefore, there is no default.

debug: LGOPT -> Boolean
++ debug returns whether we want additional output on the progress,
++ default being false

checkOptions: LGOPT -> Void
++ checkOptions checks whether the given options are consistent, and
++ yields an error otherwise

Implementation == add

maxLevel l ==
  if (opt := option(l, 'maxLevel)) case "failed" then
    "arbitrary"
  else
    retract(opt :: Any)$AnyFunctions1(Union(NonNegativeInteger, "arbitrary"))

maxDerivative l ==
  if (opt := option(l, 'maxDerivative)) case "failed" then
    "arbitrary"
  else
    retract(opt :: Any)$AnyFunctions1(Union(NonNegativeInteger, "arbitrary"))

maxShift l == maxDerivative l

maxSubst l ==
  d := maxDerivative l
  if d case NonNegativeInteger
  then (d+1)::PositiveInteger
  else d

maxDegree l ==
  if (opt := option(l, 'maxDegree)) case "failed" then
    "arbitrary"
  else
    retract(opt :: Any)$AnyFunctions1(Union(NonNegativeInteger, "arbitrary"))

maxMixedDegree l ==
  if (opt := option(l, 'maxMixedDegree)) case "failed" then
    0
  else
    retract(opt :: Any)$AnyFunctions1(NonNegativeInteger)

allDegrees l ==
  if (opt := option(l, 'allDegrees)) case "failed" then
    false
  else
    retract(opt :: Any)$AnyFunctions1(Boolean)

```

```

maxPower l ==
  if (opt := option(l, 'maxPower)) case "failed" then
    "arbitrary"
  else
    retract(opt :: Any)$AnyFunctions1(Union(PositiveInteger, "arbitrary"))

safety l ==
  if (opt := option(l, 'safety)) case "failed" then
    1$NonNegativeInteger
  else
    retract(opt :: Any)$AnyFunctions1(NonNegativeInteger)

check l ==
  if (opt := option(l, 'check)) case "failed" then
    "deterministic"
  else
    retract(opt :: Any)$AnyFunctions1(
      Union("skip", "MonteCarlo", "deterministic"))

checkExtraValues l ==
  if (opt := option(l, 'checkExtraValues)) case "failed" then
    true
  else
    retract(opt :: Any)$AnyFunctions1(Boolean)

one l ==
  if (opt := option(l, 'one)) case "failed" then
    true
  else
    retract(opt :: Any)$AnyFunctions1(Boolean)

debug l ==
  if (opt := option(l, 'debug)) case "failed" then
    false
  else
    retract(opt :: Any)$AnyFunctions1(Boolean)

homogeneous l ==
  if (opt := option(l, 'homogeneous)) case "failed" then
    false
  else
    retract(opt :: Any)$AnyFunctions1(Union(PositiveInteger, Boolean))

Somos l ==
  if (opt := option(l, 'Somos)) case "failed" then
    false
  else
    retract(opt :: Any)$AnyFunctions1(Union(PositiveInteger, Boolean))

```

```

variableName l ==
  if (opt := option(l, 'variableName)) case "failed" then
    'x
  else
    retract(opt :: Any)$AnyFunctions1(Symbol)

functionName l ==
  if (opt := option(l, 'functionName)) case "failed" then
    'f
  else
    retract(opt :: Any)$AnyFunctions1(Symbol)

indexName l ==
  if (opt := option(l, 'indexName)) case "failed" then
    'n
  else
    retract(opt :: Any)$AnyFunctions1(Symbol)

displayAsGF l ==
  if (opt := option(l, 'displayAsGF)) case "failed" then
    error "GuessOption: displayAsGF not set"
  else
    retract(opt :: Any)$AnyFunctions1(Boolean)

NNI ==> NonNegativeInteger
PI ==> PositiveInteger

checkOptions l ==
  maxD := maxDerivative l
  maxP := maxPower l
  homo := homogeneous l
  Somo := Somos l

  if Somo case PI then
    if one? Somo then
      error "Guess: Somos must be Boolean or at least two"

    if maxP case PI and one? maxP then
      error "Guess: Somos requires that maxPower is at least two"

    if maxD case NNI and maxD > Somo then
      err:String:=concat [
        "Guess: if Somos is an integer, it should be larger than ",
        "maxDerivative/maxShift or at least as big as maxSubst" ]
      error err
    else
      if Somo then
        if maxP case PI and one? maxP then
          error "Guess: Somos requires that maxPower is at least two"

```

```

    if not (maxD case NNI) or zero? maxD or one? maxD then
      err:String:= concat [_
        "Guess: Somos==true requires that maxDerivative/maxShift",_
        " is an integer, at least two, or maxSubst is an ",_
        "integer, at least three" ]
      error err

    if not (maxP case PI) and homo case Boolean and not homo then
      err:String:= concat [_
        "Guess: Somos requires that maxPower is set or ", _
        "homogeneous is not false" ]
      error err

  if homo case PI then
    if maxP case PI and maxP ~= homo then
      err:String:= _
      "Guess: only one of homogeneous and maxPower may be an integer"
      error err

    if maxD case NNI and zero? maxD then
      err:String:= concat [_
        "Guess: homogeneous requires that maxShift/maxDerivative ",_
        "is at least one or maxSubst is at least two" ]
      error err
  else
    if homo then
      if not maxP case PI then
        err:String:= concat [_
          "Guess: homogeneous==true requires that maxPower is ", _
          "an integer" ]
        error err

      if maxD case NNI and zero? maxD then
        err:String:= concat [_
          "Guess: homogeneous requires that maxShift/maxDerivative",_
          " is at least one or maxSubst is at least two" ]
        error err

```

— GOPT0.dotabb —

"GOPT0" [color="#88FF44",href="bookvol10.3.pdf#nameddest=GOPT0"]
 "STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
 "GOPT0" -> "STRING"

Chapter 9

Chapter H

domain HASHTBL HashTable

— HashTable.input —

```
)set break resume
)sys rm -f HashTable.output
)spool HashTable.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show HashTable
--R
--R HashTable(Key: SetCategory,Entry: SetCategory,hashfn: String) is a domain constructor
--R Abbreviation for HashTable is HASHTBL
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for HASHTBL
--R
--R----- Operations -----
--R copy : % -> % dictionary : () -> %
--R elt : (% ,Key,Entry) -> Entry ?.? : (% ,Key) -> Entry
--R empty : () -> % empty? : % -> Boolean
--R entries : % -> List(Entry) eq? : (% ,%) -> Boolean
--R index? : (Key,% ) -> Boolean indices : % -> List(Key)
--R key? : (Key,% ) -> Boolean keys : % -> List(Key)
--R map : ((Entry -> Entry),%) -> % qelt : (% ,Key) -> Entry
--R sample : () -> % setelt : (% ,Key,Entry) -> Entry
--R table : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?? : (% ,%) -> Boolean if Record(key: Key,entry: Entry) has SETCAT or Entry has SETCAT
```



```

--R any? : ((Entry -> Boolean),%) -> Boolean if $ has finiteAggregate
--R any? : ((Record(key: Key,entry: Entry) -> Boolean),%) -> Boolean if $ has finiteAggregate
--R bag : List(Record(key: Key,entry: Entry)) -> %
--R coerce : % -> OutputForm if Record(key: Key,entry: Entry) has SETCAT or Entry has SETCAT
--R construct : List(Record(key: Key,entry: Entry)) -> %
--R convert : % -> InputForm if Record(key: Key,entry: Entry) has KONVERT(INFORM)
--R count : ((Entry -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R count : (Entry,%) -> NonNegativeInteger if $ has finiteAggregate and Entry has SETCAT
--R count : (Record(key: Key,entry: Entry),%) -> NonNegativeInteger if $ has finiteAggregate and
--R count : ((Record(key: Key,entry: Entry) -> Boolean),%) -> NonNegativeInteger if $ has finite
--R dictionary : List(Record(key: Key,entry: Entry)) -> %
--R entry? : (Entry,%) -> Boolean if $ has finiteAggregate and Entry has SETCAT
--R eval : (%,List(Equation(Entry))) -> % if Entry has EVALAB(Entry) and Entry has SETCAT
--R eval : (%,Equation(Entry)) -> % if Entry has EVALAB(Entry) and Entry has SETCAT
--R eval : (%,Entry,Entry) -> % if Entry has EVALAB(Entry) and Entry has SETCAT
--R eval : (%,List(Entry),List(Entry)) -> % if Entry has EVALAB(Entry) and Entry has SETCAT
--R eval : (%,List(Record(key: Key,entry: Entry)),List(Record(key: Key,entry: Entry))) -> % if R
--R eval : (%,Record(key: Key,entry: Entry),Record(key: Key,entry: Entry)) -> % if Record(key: K
--R eval : (%,Equation(Record(key: Key,entry: Entry))) -> % if Record(key: Key,entry: Entry) has
--R eval : (%,List(Equation(Record(key: Key,entry: Entry)))) -> % if Record(key: Key,entry: Entr
--R every? : ((Entry -> Boolean),%) -> Boolean if $ has finiteAggregate
--R every? : ((Record(key: Key,entry: Entry) -> Boolean),%) -> Boolean if $ has finiteAggregate
--R extract! : % -> Record(key: Key,entry: Entry)
--R fill! : (%,Entry) -> % if $ has shallowlyMutable
--R find : ((Record(key: Key,entry: Entry) -> Boolean),%) -> Union(Record(key: Key,entry: Entry))
--R first : % -> Entry if Key has ORDSET
--R hash : % -> SingleInteger if Record(key: Key,entry: Entry) has SETCAT or Entry has SETCAT
--R insert! : (Record(key: Key,entry: Entry),%) -> %
--R inspect : % -> Record(key: Key,entry: Entry)
--R latex : % -> String if Record(key: Key,entry: Entry) has SETCAT or Entry has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R map : (((Entry,Entry) -> Entry),%,%) -> %
--R map : ((Record(key: Key,entry: Entry) -> Record(key: Key,entry: Entry)),%) -> %
--R map! : ((Entry -> Entry),%) -> % if $ has shallowlyMutable
--R map! : ((Record(key: Key,entry: Entry) -> Record(key: Key,entry: Entry)),%) -> % if $ has sh
--R maxIndex : % -> Key if Key has ORDSET
--R member? : (Entry,%) -> Boolean if $ has finiteAggregate and Entry has SETCAT
--R member? : (Record(key: Key,entry: Entry),%) -> Boolean if $ has finiteAggregate and Record(k
--R members : % -> List(Entry) if $ has finiteAggregate
--R members : % -> List(Record(key: Key,entry: Entry)) if $ has finiteAggregate
--R minIndex : % -> Key if Key has ORDSET
--R more? : (%,NonNegativeInteger) -> Boolean
--R parts : % -> List(Entry) if $ has finiteAggregate
--R parts : % -> List(Record(key: Key,entry: Entry)) if $ has finiteAggregate
--R qsetelt! : (%,Key,Entry) -> Entry if $ has shallowlyMutable
--R reduce : (((Record(key: Key,entry: Entry),Record(key: Key,entry: Entry)) -> Record(key: Key,
--R reduce : (((Record(key: Key,entry: Entry),Record(key: Key,entry: Entry)) -> Record(key: Key,
--R reduce : (((Record(key: Key,entry: Entry),Record(key: Key,entry: Entry)) -> Record(key: Key,
--R remove : ((Record(key: Key,entry: Entry) -> Boolean),%) -> % if $ has finiteAggregate
--R remove : (Record(key: Key,entry: Entry),%) -> % if $ has finiteAggregate and Record(key: Key

```

```

--R remove! : (Key,%) -> Union(Entry,"failed")
--R remove! : ((Record(key: Key,entry: Entry) -> Boolean),%) -> % if $ has finiteAggregate
--R remove! : (Record(key: Key,entry: Entry),%) -> % if $ has finiteAggregate
--R removeDuplicates : % -> % if $ has finiteAggregate and Record(key: Key,entry: Entry) has SETCAT
--R search : (Key,%) -> Union(Entry,"failed")
--R select : ((Record(key: Key,entry: Entry) -> Boolean),%) -> % if $ has finiteAggregate
--R select! : ((Record(key: Key,entry: Entry) -> Boolean),%) -> % if $ has finiteAggregate
--R size? : (%,NonNegativeInteger) -> Boolean
--R swap! : (%,Key,Key) -> Void if $ has shallowlyMutable
--R table : List(Record(key: Key,entry: Entry)) -> %
--R ~=? : (%,%) -> Boolean if Record(key: Key,entry: Entry) has SETCAT or Entry has SETCAT
--R
--E 1

```

```

)spool
)lisp (bye)

```

— HashTable.help —

=====

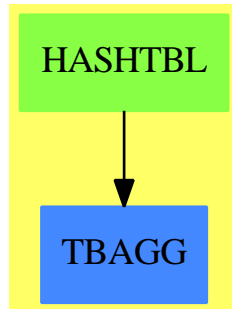
HashTable examples

=====

This domain provides access to the underlying Lisp hash tables.
 By varying the hashfn parameter, tables suited for different
 purposes can be obtained.

See Also:
 o)show HashTable

HashTable (HASHTBL)



See

- ⇒ “InnerTable” (INTABL) 10 on page 1335
- ⇒ “Table” (TABLE) 21 on page 2663
- ⇒ “EqTable” (EQTBL) 6 on page 698
- ⇒ “StringTable” (STRTBL) 20 on page 2610
- ⇒ “GeneralSparseTable” (GSTBL) 8 on page 1078
- ⇒ “SparseTable” (STBL) 20 on page 2450

Exports:

any?	bag	coerce	construct	convert
copy	count	dictionary	entry?	elt
empty	empty?	entries	eq?	eval
every?	extract!	fill!	find	first
hash	index?	indices	insert!	inspect
key?	keys	latex	less?	map
map!	maxIndex	member?	members	minIndex
more?	parts	qelt	qsetelt!	reduce
remove	remove!	removeDuplicates	sample	search
select	select!	setelt	size?	swap!
table	#?	?=?	?~=?	?..?

— domain HASHTBL HashTable —

```

)abbrev domain HASHTBL HashTable
++ Author: Stephen M. Watt
++ Date Created: 1985
++ Date Last Updated: June 21, 1991
++ Description:
++ This domain provides access to the underlying Lisp hash tables.
++ By varying the hashfn parameter, tables suited for different
++ purposes can be obtained.
  
```

```

HashTable(Key, Entry, hashfn): Exports == Implementation where
  Key, Entry: SetCategory
  hashfn: String -- Union("EQ", "UEQUAL", "CVEC", "ID")
  
```

```

Exports ==> TableAggregate(Key, Entry) with
    finiteAggregate

Implementation ==> add
    Pair ==> Record(key: Key, entry: Entry)
    Ex    ==> OutputForm
    failMsg := GENSYM()$Lisp

    t1 = t2          == EQ(t1, t2)$Lisp
    keys t           == HKEYS(t)$Lisp
    # t              == HASH_-TABLE_-COUNT(t)$Lisp
    setelt(t, k, e)   == HPUT(t,k,e)$Lisp
    remove_!(k:Key, t:%) ==
        r := HGET(t,k,failMsg)$Lisp
        not EQ(r,failMsg)$Lisp =>
            HREM(t, k)$Lisp
            r pretend Entry
            "failed"

    empty() ==
        MAKE_-HASHTABLE(INTERN(hashfn)$Lisp,
            INTERN("STRONG")$Lisp)$Lisp

    search(k:Key, t:%) ==
        r := HGET(t, k, failMsg)$Lisp
        not EQ(r, failMsg)$Lisp => r pretend Entry
        "failed"

    -----

    — HASHTBL.dotabb —

    "HASHTBL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=HASHTBL"]
    "TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]
    "HASHTBL" -> "TBAGG"

    -----

```

domain HEAP Heap

```

    — Heap.input —

)set break resume

```

```

)sys rm -f Heap.output
)spool Heap.output
)set message test on
)set message auto off
)clear all

```

```

--S 1 of 42
a:=Heap INT:= heap [1,2,3,4,5]
--R
--R
--R (1) [5,4,2,1,3]
--R
--E 1

```

Type: Heap(Integer)

```

--S 2 of 42
bag([1,2,3,4,5])$Heap(INT)
--R
--R
--R (2) [5,4,3,1,2]
--R
--E 2

```

Type: Heap(Integer)

```

--S 3 of 42
c:=copy a
--R
--R
--R (3) [5,4,2,1,3]
--R
--E 3

```

Type: Heap(Integer)

```

--S 4 of 42
empty? a
--R
--R
--R (4) false
--R
--E 4

```

Type: Boolean

```

--S 5 of 42
b:=empty()$(Heap INT)
--R
--R
--R (5) []
--R
--E 5

```

Type: Heap(Integer)

```

--S 6 of 42
empty? b
--R
--R

```

[illegible]

```
--E 11

--S 12 of 42
(a=c)@Boolean
--R
--R
--R (13) false
--R
--R Type: Boolean
--E 12

--S 13 of 42
(a~=c)
--R
--R
--R (14) true
--R
--R Type: Boolean
--E 13

--S 14 of 42
a
--R
--R
--R (15) [4,3,2,1]
--R
--R Type: Heap(Integer)
--E 14

--S 15 of 42
inspect a
--R
--R
--R (16) 4
--R
--R Type: PositiveInteger
--E 15

--S 16 of 42
insert!(9,a)
--R
--R
--R (17) [9,4,2,1,3]
--R
--R Type: Heap(Integer)
--E 16

--S 17 of 42
map(x+>x+10,a)
--R
--R
--R (18) [19,14,12,11,13]
--R
--R Type: Heap(Integer)
--E 17
```

--S 18 of 42

a

--R

--R

--R (19) [9,4,2,1,3]

--R

Type: Heap(Integer)

--E 18

--S 19 of 42

map!(x+>x+10,a)

--R

--R

--R (20) [19,14,12,11,13]

--R

Type: Heap(Integer)

--E 19

--S 20 of 42

a

--R

--R

--R (21) [19,14,12,11,13]

--R

Type: Heap(Integer)

--E 20

--S 21 of 42

max a

--R

--R

--R (22) 19

--R

Type: PositiveInteger

--E 21

--S 22 of 42

merge(a,c)

--R

--R

--R (23) [19,14,12,11,13,5,4,2,1,3]

--R

Type: Heap(Integer)

--E 22

--S 23 of 42

a

--R

--R

--R (24) [19,14,12,11,13]

--R

Type: Heap(Integer)

--E 23

--S 24 of 42

merge!(a,c)


```

--R
--R
--R (25) [19,14,12,11,13,5,4,2,1,3]
--R
--R                                         Type: Heap(Integer)
--E 24

--S 25 of 42
a
--R
--R
--R (26) [19,14,12,11,13,5,4,2,1,3]
--R
--R                                         Type: Heap(Integer)
--E 25

--S 26 of 42
c
--R
--R
--R (27) [5,4,2,1,3]
--R
--R                                         Type: Heap(Integer)
--E 26

--S 27 of 42
sample()$Heap(INT)
--R
--R
--R (28) []
--R
--R                                         Type: Heap(Integer)
--E 27

--S 28 of 42
#a
--R
--R
--R (29) 10
--R
--R                                         Type: PositiveInteger
--E 28

--S 29 of 42
any?(x+>(x=14),a)
--R
--R
--R (30) true
--R
--R                                         Type: Boolean
--E 29

--S 30 of 42
every?(x+>(x=11),a)
--R
--R

```



```

--E 36

--S 37 of 42
latex a
--R
--R
--R (38) "\mbox{\bf Unimplemented}"
--R
--R                                         Type: String
--E 37

--S 38 of 42
hash a
--R
--R
--R (39) 36647017
--R
--R                                         Type: SingleInteger
--E 38

--S 39 of 42
count(14,a)
--R
--R
--R (40) 1
--R
--R                                         Type: PositiveInteger
--E 39

--S 40 of 42
count(x+-(x>13),a)
--R
--R
--R (41) 2
--R
--R                                         Type: PositiveInteger
--E 40

--S 41 of 42
coerce a
--R
--R
--R (42) [19,14,12,11,13,5,4,2,1,3]
--R
--R                                         Type: OutputForm
--E 41

--S 42 of 42
)show Heap
--R
--R Heap(S: OrderedSet) is a domain constructor
--R Abbreviation for Heap is HEAP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for HEAP
--R

```

```

--R----- Operations -----
--R bag : List(S) -> %                copy : % -> %
--R empty : () -> %                  empty? : % -> Boolean
--R eq? : (%,% ) -> Boolean          extract! : % -> S
--R heap : List(S) -> %              insert! : (S,% ) -> %
--R inspect : % -> S                 map : ((S -> S),%) -> %
--R max : % -> S                     merge : (%,% ) -> %
--R merge! : (%,% ) -> %             sample : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ==? : (%,% ) -> Boolean if S has SETCAT
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if S has SETCAT
--R count : (S,% ) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R eval : (% ,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (% ,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (% ,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (% ,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R hash : % -> SingleInteger if S has SETCAT
--R latex : % -> String if S has SETCAT
--R less? : (% ,NonNegativeInteger) -> Boolean
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R member? : (S,% ) -> Boolean if $ has finiteAggregate and S has SETCAT
--R members : % -> List(S) if $ has finiteAggregate
--R more? : (% ,NonNegativeInteger) -> Boolean
--R parts : % -> List(S) if $ has finiteAggregate
--R size? : (% ,NonNegativeInteger) -> Boolean
--R ~=? : (%,% ) -> Boolean if S has SETCAT
--R
--E 42

)spool
)lisp (bye)

```

— Heap.help —

Heap examples

The domain Heap(S) implements a priority queue of objects of type S such that the operation extract! removes and returns the maximum element. The implementation represents heaps as flexible arrays. The representation and algorithms give complexity of $O(\log(n))$ for insertion and extractions, and $O(n)$ for construction.

Create a heap of five elements:

```
a:Heap INT:= heap [1,2,3,4,5]
               [5,4,2,1,3]
```

Use bag to convert a Bag into a Heap:

```
bag([1,2,3,4,5])$Heap(INT)
    [5,4,3,1,2]
```

The operation copy can be used to copy a Heap:

```
c:=copy a
    [5,4,2,1,3]
```

Use empty? to check if the heap is empty:

```
empty? a
    false
```

Use empty to create a new, empty heap:

```
b:=empty()$(Heap INT)
    []
```

and we can see that the newly created heap is empty:

```
empty? b
    true
```

The eq? function compares the reference of one heap to another:

```
eq?(a,c)
    false
```

The extract! function removes largest element of the heap:

```
extract! a
    5
```

Now extract! elements repeatedly until none are left, collecting the elements in a list.

```
[extract!(h) while not empty?(h)]
    [9,7,3,2,- 4,- 7]
                                Type: List Integer
```

Another way to produce the same result is by defining a heapsort function.

```
heapsort(x) == (empty? x => []; cons(extract!(x),heapsort x))
```

Type: Void

Create another sample heap.

```
h1 := heap [17,-4,9,-11,2,7,-7]
      [17,2,9,- 11,- 4,7,- 7]
      Type: Heap Integer
```

Apply heapsort to present elements in order.

```
heapsort h1
      [17,9,7,2,- 4,- 7,- 11]
      Type: List Integer
```

Heaps can be compared with =

```
(a=c)@Boolean
      false
```

and ~=

```
(a~=c)
      true
```

The inspect function shows the largest element in the heap:

```
inspect a
      4
```

The insert! function adds an element to the heap:

```
insert!(9,a)
      [9,4,2,1,3]
```

The map function applies a function to every element of the heap and returns a new heap:

```
map(x+>x+10,a)
      [19,14,12,11,13]
```

The original heap is unchanged:

```
a
      [9,4,2,1,3]
```

The map! function applies a function to every element of the heap and returns the original heap with modifications:

```
map!(x+>x+10,a)
      [19,14,12,11,13]
```

The original heap has been modified:

```
a
  [19,14,12,11,13]
```

The max function returns the largest element in the heap:

```
max a
  19
```

The merge function takes two heaps and creates a new heap with all of the elements:

```
merge(a,c)
  [19,14,12,11,13,5,4,2,1,3]
```

Notice that the original heap is unchanged:

```
a
  [19,14,12,11,13]
```

The merge! function takes two heaps and modifies the first heap argument to contain all of the elements:

```
merge!(a,c)
  [19,14,12,11,13,5,4,2,1,3]
```

Notice that the first argument was modified:

```
a
  [19,14,12,11,13,5,4,2,1,3]
```

but the second argument was not:

```
c
  [5,4,2,1,3]
```

A new, empty heap can be created with sample:

```
sample()$Heap(INT)
  []
```

The # function gives the size of the heap:

```
#a
  10
```

The any? function tests each element against a predicate function and returns true if any pass:

```
any?(x+-(x=14),a)
true
```

The `every?` function tests each element against a predicate function and returns true if they all pass:

```
every?(x+-(x=11),a)
false
```

The `parts` function returns a list of the elements in the heap:

```
parts a
[19,14,12,11,13,5,4,2,1,3]
```

The `size?` predicate compares the size of the heap to a value:

```
size?(a,9)
false
```

The `more?` predicate asks if the heap size is larger than a value:

```
more?(a,9)
true
```

The `less?` predicate asks if the heap size is smaller than a value:

```
less?(a,9)
false
```

The `members` function returns a list of the elements of the heap:

```
members a
[19,14,12,11,13,5,4,2,1,3]
```

The `member?` predicate asks if an element is in the heap:

```
member?(14,a)
true
```

The `count` function has two forms, one of which counts the number of copies of an element in the heap:

```
count(14,a)
1
```

The second form of the `count` function accepts a predicate to test against each member of the heap and counts the number of true results:

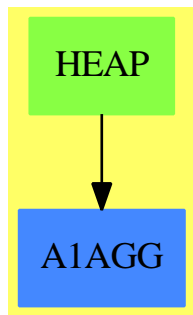
```
count(x+-(x>13),a)
```


2

See Also:

- o `)show Stack`
- o `)show ArrayStack`
- o `)show Queue`
- o `)show Dequeue`
- o `)show Heap`
- o `)show BagAggregate`

Heap (HEAP)



See

- ⇒ “Stack” (STACK) [20](#) on page [2563](#)
- ⇒ “ArrayStack” (ASTACK) [2](#) on page [65](#)
- ⇒ “Queue” (QUEUE) [18](#) on page [2184](#)
- ⇒ “Dequeue” (DEQUEUE) [5](#) on page [512](#)

Exports:

<code>any?</code>	<code>bag</code>	<code>coerce</code>	<code>copy</code>	<code>count</code>
<code>empty</code>	<code>empty?</code>	<code>eq?</code>	<code>eval</code>	<code>every?</code>
<code>extract!</code>	<code>hash</code>	<code>heap</code>	<code>insert!</code>	<code>inspect</code>
<code>latex</code>	<code>less?</code>	<code>map</code>	<code>map!</code>	<code>max</code>
<code>member?</code>	<code>members</code>	<code>merge</code>	<code>merge!</code>	<code>more?</code>
<code>parts</code>	<code>sample</code>	<code>size?</code>	<code>#?</code>	<code>?=?</code>
<code>?~=?</code>				

— domain **HEAP** **Heap** —

```

)abbrev domain HEAP Heap
++ Author: Michael Monagan and Stephen Watt
++ Date Created: June 86 and July 87
  
```

```

++ Date Last Updated:Feb 92
++ Description:
++ Heap implemented in a flexible array to allow for insertions
++ Complexity:  $O(\log n)$  insertion, extraction and  $O(n)$  construction
--% Dequeue and Heap data types

```

```

Heap(S:OrderedSet): Exports == Implementation where
  Exports == PriorityQueueAggregate S with
    heap : List S -> %
      ++ heap(ls) creates a heap of elements consisting of the
      ++ elements of ls.
      ++
      ++E i:Heap INT := heap [1,6,3,7,5,2,4]

```

```

-- Inherited Signatures repeated for examples documentation

```

```

bag : List S -> %
  ++
  ++X bag([1,2,3,4,5])$Heap(INT)
copy : % -> %
  ++
  ++X a:Heap INT:= heap [1,2,3,4,5]
  ++X copy a
empty? : % -> Boolean
  ++
  ++X a:Heap INT:= heap [1,2,3,4,5]
  ++X empty? a
empty : () -> %
  ++
  ++X b:=empty()$(Heap INT)
eq? : (%,%) -> Boolean
  ++
  ++X a:Heap INT:= heap [1,2,3,4,5]
  ++X b:=copy a
  ++X eq?(a,b)
extract_! : % -> S
  ++
  ++X a:Heap INT:= heap [1,2,3,4,5]
  ++X extract! a
  ++X a
insert_! : (S,%) -> %
  ++
  ++X a:Heap INT:= heap [1,2,3,4,5]
  ++X insert!(8,a)
  ++X a
inspect : % -> S
  ++
  ++X a:Heap INT:= heap [1,2,3,4,5]
  ++X inspect a
map : ((S -> S),%) -> %

```

```

++
++X a:Heap INT:= heap [1,2,3,4,5]
++X map(x+>x+10,a)
++X a
max : % -> S
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X max a
merge : (%,%) -> %
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X b:Heap INT:= heap [6,7,8,9,10]
++X merge(a,b)
merge! : (%,%) -> %
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X b:Heap INT:= heap [6,7,8,9,10]
++X merge!(a,b)
++X a
++X b
sample : () -> %
++
++X sample()$Heap(INT)
less? : (%,NonNegativeInteger) -> Boolean
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X less?(a,9)
more? : (%,NonNegativeInteger) -> Boolean
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X more?(a,9)
size? : (%,NonNegativeInteger) -> Boolean
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X size?(a,5)
if $ has shallowlyMutable then
map! : ((S -> S),%) -> %
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X map!(x+>x+10,a)
++X a
if S has SetCategory then
latex : % -> String
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X latex a
hash : % -> SingleInteger
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X hash a

```

```

coerce : % -> OutputForm
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X coerce a
"=": (%,% ) -> Boolean
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X b:Heap INT:= heap [1,2,3,4,5]
++X (a=b)@Boolean
"~=" : (%,% ) -> Boolean
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X b:=copy a
++X (a~b)
if % has finiteAggregate then
every? : ((S -> Boolean),%) -> Boolean
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X every?(x+->(x=4),a)
any? : ((S -> Boolean),%) -> Boolean
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X any?(x+->(x=4),a)
count : ((S -> Boolean),%) -> NonNegativeInteger
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X count(x+->(x>2),a)
_# : % -> NonNegativeInteger
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X #a
parts : % -> List S
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X parts a
members : % -> List S
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X members a
if % has finiteAggregate and S has SetCategory then
member? : (S,% ) -> Boolean
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X member?(3,a)
count : (S,% ) -> NonNegativeInteger
++
++X a:Heap INT:= heap [1,2,3,4,5]
++X count(4,a)

Implementation == IndexedFlexibleArray(S,0) add

```

```

Rep := IndexedFlexibleArray( S,0)
empty() == empty()$Rep
heap l ==
  n := #l
  h := empty()
  n = 0 => h
  for x in l repeat insert_!(x,h)
  h
siftUp: (%,Integer,Integer) -> Void
siftUp(r,i,n) ==
  -- assertion 0 <= i < n
  t := r.i
  while (j := 2*i+1) < n repeat
    if (k := j+1) < n and r.j < r.k then j := k
    if t < r.j then (r.i := r.j; r.j := t; i := j) else leave

extract_! r ==
  -- extract the maximum from the heap O(log n)
  n := #r :: Integer
  n = 0 => error "empty heap"
  t := r(0)
  r(0) := r(n-1)
  delete_!(r,n-1)
  n = 1 => t
  siftUp(r,0,n-1)
  t

insert_!(x,r) ==
  -- Williams' insertion algorithm O(log n)
  j := (#r) :: Integer
  r:=concat_!(r,concat(x,empty()$Rep))
  while j > 0 repeat
    i := (j-1) quo 2
    if r(i) >= x then leave
    r(j) := r(i)
    j := i
  r(j):=x
  r

max r == if #r = 0 then error "empty heap" else r.0
inspect r == max r

makeHeap(r:%):% ==
  -- Floyd's heap construction algorithm O(n)
  n := #r
  for k in n quo 2 -1 .. 0 by -1 repeat siftUp(r,k,n)
  r
bag l == makeHeap construct(l)$Rep
merge(a,b) == makeHeap concat(a,b)
merge_!(a,b) == makeHeap concat_!(a,b)

```

— HEAP.dotabb —

```
"HEAP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=HEAP"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"HEAP" -> "A1AGG"
```

domain HEXADEC HexadecimalExpansion

— HexadecimalExpansion.input —

```
)set break resume
)sys rm -f HexadecimalExpansion.output
)spool HexadecimalExpansion.output
)set message test on
)set message auto off
)clear all
--S 1 of 7
r := hex(22/7)
--R
--R
--R
--R (1) 3.249
--R
--R                                          Type: HexadecimalExpansion
--E 1

--S 2 of 7
r + hex(6/7)
--R
--R
--R (2) 4
--R
--R                                          Type: HexadecimalExpansion
--E 2

--S 3 of 7
[hex(1/i) for i in 350..354]
--R
--R
--R (3)
--R -----
```

```

--R [0.00BB3EE721A54D88, 0.00BAB6561, 0.00BA2E8, 0.00B9A7862A0FF465879D5F,
--R -----
--R 0.00B92143FA36F5E02E4850FE8DBD78]
--R                                         Type: List(HexadecimalExpansion)
--E 3

--S 4 of 7
hex(1/1007)
--R
--R
--R (4)
--R 0.
--R OVERBAR
--R 0041149783F0BF2C7D13933192AF6980619EE345E91EC2BB9D5CCA5C071E40926E54E8D
--R DAE24196C0B2F8A0AAD60DBA57F5D4C8536262210C74F1
--R                                         Type: HexadecimalExpansion
--E 4

--S 5 of 7
p := hex(1/4)*x**2 + hex(2/3)*x + hex(4/9)
--R
--R
--R (5)  $0.4x^2 + 0.Ax + 0.71C$ 
--R                                         Type: Polynomial(HexadecimalExpansion)
--E 5

--S 6 of 7
q := D(p, x)
--R
--R
--R (6)  $0.8x + 0.A$ 
--R                                         Type: Polynomial(HexadecimalExpansion)
--E 6

--S 7 of 7
g := gcd(p, q)
--R
--R
--R (7)  $x + 1.5$ 
--R                                         Type: Polynomial(HexadecimalExpansion)
--E 7
)spool
)lisp (bye)

```

— HexadecimalExpansion.help —

```
=====
HexadecimalExpansion
=====
```

All rationals have repeating hexadecimal expansions. The operation `hex` returns these expansions of type `HexadecimalExpansion`. Operations to access the individual numerals of a hexadecimal expansion can be obtained by converting the value to `RadixExpansion(16)`. More examples of expansions are available in the `DecimalExpansion`, `BinaryExpansion`, and `RadixExpansion`.

This is a hexadecimal expansion of a rational number.

```
r := hex(22/7)
    ---
    3.249
                                Type: HexadecimalExpansion
```

Arithmetic is exact.

```
r + hex(6/7)
    4
                                Type: HexadecimalExpansion
```

The period of the expansion can be short or long ...

```
[hex(1/i) for i in 350..354]
    -----
    [0.00BB3EE721A54D88, 0.00BAB6561, 0.00BA2E8, 0.00B9A7862A0FF465879D5F,
    0.00B92143FA36F5E02E4850FE8DBD78]
                                Type: List HexadecimalExpansion
```

or very long!

```
hex(1/1007)
    -----
    0.0041149783F0BF2C7D13933192AF6980619EE345E91EC2BB9D5CCA5C071E40926E54E8D
    -----
    DAE24196C0B2F8A0AAD60DBA57F5D4C8536262210C74F1
                                Type: HexadecimalExpansion
```

These numbers are bona fide algebraic objects.

```
p := hex(1/4)*x**2 + hex(2/3)*x + hex(4/9)
    2      -      ---
    0.4x  + 0.Ax + 0.71C
```


Type: Polynomial HexadecimalExpansion

$q := D(p, x)$

$0.8x + 0.\overline{A}$

Type: Polynomial HexadecimalExpansion

$g := \gcd(p, q)$

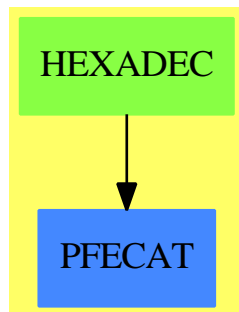
$x + 1.\overline{5}$

Type: Polynomial HexadecimalExpansion

See Also:

- o)help RadixExpansion
- o)help BinaryExpansion
- o)help DecimalExpansion
- o)show HexadecimalExpansion

HexadecimalExpansion (HEXADEC)



See

- ⇒ “RadixExpansion” (RADIX) [19](#) on page [2205](#)
- ⇒ “BinaryExpansion” (BINARY) [3](#) on page [292](#)
- ⇒ “DecimalExpansion” (DECIMAL) [5](#) on page [465](#)

Exports:

0	1
abs	associates?
ceiling	characteristic
charthRoot	coerce
conditionP	convert
D	denom
denominator	differentiate
divide	euclideanSize
eval	expressIdealMember
exquo	extendedEuclidean
factor	factorPolynomial
factorSquareFreePolynomial	floor
fractionPart	gcd
gcdPolynomial	hash
hex	init
inv	latex
lcm	map
max	min
multiEuclidean	negative?
nextItem	numer
numerator	one?
patternMatch	positive?
prime?	principalIdeal
random	recip
reducedSystem	retract
retractIfCan	sample
sign	sizeLess?
solveLinearPolynomialEquation	squareFree
squareFreePart	squareFreePolynomial
subtractIfCan	unit?
unitCanonical	unitNormal
wholePart	zero?
?*?	?**?
?+?	?-?
~?	?/?
?=?	?^?
?~=?	?<?
?<=?	?>?
?>=?	?..?
?quo?	?rem?

— domain HEXADEC HexadecimalExpansion —

```
)abbrev domain HEXADEC HexadecimalExpansion
++ Author: Clifton J. Williamson
```

```

++ Date Created: April 26, 1990
++ Date Last Updated: May 15, 1991
++ Description:
++ This domain allows rational numbers to be presented as repeating
++ hexadecimal expansions.

HexadecimalExpansion(): Exports == Implementation where
Exports ==> QuotientFieldCategory(Integer) with
  coerce: % -> Fraction Integer
    ++ coerce(h) converts a hexadecimal expansion to a rational number.
  coerce: % -> RadixExpansion(16)
    ++ coerce(h) converts a hexadecimal expansion to a radix expansion
    ++ with base 16.
  fractionPart: % -> Fraction Integer
    ++ fractionPart(h) returns the fractional part of a hexadecimal expansion.
  hex: Fraction Integer -> %
    ++ hex(r) converts a rational number to a hexadecimal expansion.

Implementation ==> RadixExpansion(16) add
  hex r == r :: %
  coerce(x:%): RadixExpansion(16) == x pretend RadixExpansion(16)

```

— HEXADEC.dotabb —

```

"HEXADEC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=HEXADEC"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"HEXADEC" -> "PFECAT"

```

package HTMLFORM HTMLFormat

Here I have put some information about 'how to use' and 'the benefits of' this HTML formatter. Also some information for programmers if they want to extend this package.

If you want information about creating output formatters in general then, rather than duplicating content here I refer you to `mathml.spad.pamphlet` containing the `MathMLFormat` domain by Arthur C. Ralfs. This contains useful information for writers of output formatters.

Overview

This package allows users to cut and paste output from the Axiom command line to a HTML page. This output is enabled by typing:

```
)set output html on
```

After this the command line will output html (in addition to other formats that are enabled) and this html code can then be copied and pasted into a HTML document.

The HTML produced is well formed XML, that is, all tags have equivalent closing tags.

Why output to HTML?

In some ways HTMLFormat is a compromise between the standard text output and specialised formats like MathMLFormat. The potential quality is never going to be as good as output to a specialised maths renderer but on the other hand it is a lot better than the clunky fixed width font text output. The quality is not the only issue though, the direct output in any format is unlikely to be exactly what the user wants, so possibly more important than quality is the ability to edit the output.

HTMLFormat has advantages that the other output formats don't, for instance,

- It works with any browser without the need for plugins (as far as I know most computers should have the required fonts)
- Users can easily annotate and add comments using colour, bold, underline and so on.
- Annotations can easily be done with whatever html editor or text editor you are familiar with.
- Edits to the output will cause the width of columns and so on to be automatically adjusted, no need to try to insert spaces to get the superscripts to line up again!
- It is very easy to customise output so, for instance, we can fit a lot of information in a compact space on the page.

Using the formatter

We can cause the command line interpreter to output in html by typing the following:

```
)set output html on
```

After this the command line will output html (in addition to other formats that are enabled) and this html code can then be copied and pasted into an existing HTML document.

If you do not already have an html page to copy the output to then you can create one with a text editor and entering the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
```

```

<head>
  <title>Enter Your Title Here</title>
</head>
<body>
  Copy and paste the output from command line here.
</body>
</html>

```

Or using any program that will export to html such as OpenOffice.org writer.

Form of the output

HTMLFormat does not try to interpret syntax, for instance in an example like:
 (1) -> integral(x^x,x)
 it just takes what OutputForm provides and does not try to replace
 %A with the bound variable x.

Matrix Formatting

A big requirement for me is to fit big matrices on ordinary web pages.

At the moment the default output for a matrix is a grid, however it easy to modify this for a single matrix, or a whole page or whole site by using css (cascading style sheets). For instance we can get a more conventional looking matrix by adding the following style to the top of the page after the `<head>` tag:

```

<style type="text/css">
#matl {border-left-style:solid}
#matr {border-right-style:solid}
#matlt {border-left-style:solid;border-top-style:solid}
#matrt {border-right-style:solid;border-top-style:solid}
#matlb {border-left-style:solid;border-bottom-style:solid}
#matrb {border-right-style:solid;border-bottom-style:solid}
</style>

```

There are many other possibilities, for instance we can generate a matrix with bars either side to indicate a determinant. All we have to do is change the css for the site, page or individual element.

Programmers Guide

This package converts from OutputForm, which is a hierarchical tree structure, to html which uses tags arranged in a hierarchical tree structure. So the package converts from one tree (graph) structure to another.

This conversion is done in two stages using an intermediate Tree String structure. This Tree String structure represents HTML where:

- leafs represents unstructured text
- string in leafs contains the text
- non-leafs represents xml elements
- string in non-leafs represents xml attributes

This is created by traversing `OutputForm` while building up the Tree String structure.

The second stage is to convert the Tree Structure to text. All text output is done using:

```
sayTeX$Lisp
```

I have not produced and output to String as I don't know a way to append to a long string efficiently and I don't know how to insert carriage- returns into a String.

Future Developments

There would be some benefits in creating a `XMLFormat` category which would contain common elements for all xml formatted outputs such as `HTMLFormat`, `MathMLFormat`, `SVGFormat` and `X3DFormat`. However programming effort might be better spent creating a version of `OutputForm` which has better syntax information.

— HTMLFormat.input —

```
)set break resume
)sys rm -f HTMLFormat.output
)spool HTMLFormat.output
)set message test on
)set message auto off
)clear all

--S 1 of 9
)show HTMLFormat
--R HTMLFormat is a domain constructor
--R Abbreviation for HTMLFormat is HTMLFORM
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for HTMLFORM
--R
--R----- Operations -----
--R ?? : (%,% ) -> Boolean          coerce : OutputForm -> String
--R coerce : % -> OutputForm        coerceL : OutputForm -> String
--R coerceS : OutputForm -> String  display : String -> Void
--R exprex : OutputForm -> String   hash : % -> SingleInteger
```

```
--R latex : % -> String
--R
--E 1

--S 2 of 9
coerce("3+4"::OutputForm)$HTMLFORM
--R
--R "3+4"
--R
--R (1) " "
--R
--R
--E 2
Type: String

--S 3 of 9
coerce("sqrt(3+4)"::OutputForm)$HTMLFORM
--R
--R "sqrt(3+4)"
--R
--R (2) " "
--R
--R
--E 3
Type: String

--S 4 of 9
coerce(sqrt(3+4)::OutputForm)$HTMLFORM
--R
--R &radic;7
--R
--R (3) " "
--R
--R
--E 4
Type: String

--S 5 of 9
coerce(sqrt(3+x)::OutputForm)$HTMLFORM
--R
--R <table border='0' id='root'>
--R <tr id='root'>
--R <td id='root'>
--R &radic;
--R </td>
--R <td id='root' style='border-top-style:solid'>
--R x+3
--R </td>
--R </tr>
--R </table>
--R
--R (4) " "
--R
--R
--E 5
Type: String

--S 6 of 9
```

```
coerceS(sqrt(3+x)::OutputForm)$HTMLFORM
```

```
--R
--R<table border='0' id='root'>
--R<tr id='root'>
--R<td id='root'>
--R&radic;
--R</td>
--R<td id='root' style='border-top-style:solid'>
--Rx+3
--R</td>
--R</tr>
--R</table>
--R
--R (5) " "
```

Type: String

```
--S 7 of 9
```

```
coerceL(sqrt(3+x)::OutputForm)$HTMLFORM
```

```
--R
--R<table border='0' id='root'>
--R<tr id='root'>
--R<td id='root'>
--R&radic;
--R</td>
--R<td id='root' style='border-top-style:solid'>
--Rx+3
--R</td>
--R</tr>
--R</table>
--R
--R (6) " "
```

Type: String

```
--S 8 of 9
```

```
expres(sqrt(3+x)::OutputForm)$HTMLFORM
```

```
--R
--R
--R (7) "{ROOT}{+}{x}{3}"
--R
--E 8
```

Type: String

```
--S 9 of 9
```

```
display(coerce(sqrt(3+x)::OutputForm)$HTMLFORM)$HTMLFORM
```

```
--R
--R<table border='0' id='root'>
--R<tr id='root'>
--R<td id='root'>
--R&radic;
```



```

--R</td>
--R<td id='root' style='border-top-style:solid'>
--Rx+3
--R</td>
--R</tr>
--R</table>
--R
--R
--E 9
)spool
)lisp (bye)

```

Type: Void

— HTMLFormat.help —

=====

HTMLFormat examples

=====

HtmlFormat provides a coercion from OutputForm to html.

```
coerce("3+4"::OutputForm)$HTMLFORM
"3+4"
```

```
coerce("sqrt(3+4)"::OutputForm)$HTMLFORM
"sqrt(3+4)"
```

```
coerce(sqrt(3+4)::OutputForm)$HTMLFORM
&radic;7
```

```
coerce(sqrt(3+x)::OutputForm)$HTMLFORM
<table border='0' id='root'>
  <tr id='root'>
    <td id='root'>
      &radic;
    </td>
    <td id='root' style='border-top-style:solid'>
      x+3
    </td>
  </tr>
</table>

```

```
coerceS(sqrt(3+x)::OutputForm)$HTMLFORM
<table border='0' id='root'>
  <tr id='root'>
    <td id='root'>
      &radic;
    </td>
  </tr>
</table>

```

```

<td id='root' style='border-top-style:solid'>
x+3
</td>
</tr>
</table>

coerceL(sqrt(3+x)::OutputForm)$HTMLFORM
<table border='0' id='root'>
<tr id='root'>
<td id='root'>
&radic;
</td>
<td id='root' style='border-top-style:solid'>
x+3
</td>
</tr>
</table>

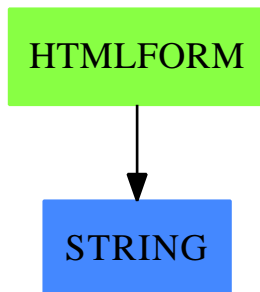
expres(sqrt(3+x)::OutputForm)$HTMLFORM
"{{ROOT}}{{+}}{{x}}{{3}}}"

display(coerce(sqrt(3+x)::OutputForm)$HTMLFORM)$HTMLFORM
<table border='0' id='root'>
<tr id='root'>
<td id='root'>
&radic;
</td>
<td id='root' style='border-top-style:solid'>
x+3
</td>
</tr>
</table>

See Also:
o )show HTMLFormat

```

HTMLFormat (HTMLFORM)



See

← “SetCategory” (SETCAT) ?? on page ??

Exports:

?=? ?=? coerce coerceL coerceS
 display exprex hash latex

— domain HTMLFORM HTMLFormat —

```

)abbrev domain HTMLFORM HTMLFormat
++ Author: Martin J Baker, Arthur C. Ralfs, Robert S. Sutor
++ Date: January 2010
++ Description:
++ HtmlFormat provides a coercion from OutputForm to html.
HTMLFormat(): public == private where
  E      ==> OutputForm
  I      ==> Integer
  L      ==> List
  S      ==> String

public == SetCategory with
  coerce:   E -> S
    ++ coerce(o) changes o in the standard output format to html format.
    ++
    ++X coerce(sqrt(3+x)::OutputForm)$HTMLFORM
  coerceS:  E -> S
    ++ coerceS(o) changes o in the standard output format to html
    ++ format and displays formatted result.
    ++
    ++X coerceS(sqrt(3+x)::OutputForm)$HTMLFORM
  coerceL:  E -> S
    ++ coerceL(o) changes o in the standard output format to html
    ++ format and displays result as one long string.
    ++
    ++X coerceL(sqrt(3+x)::OutputForm)$HTMLFORM
  exprex:   E -> S
  
```

```

    ++ exprex(o) coverts \spadtype{OutputForm} to \spadtype{String}
    ++
    ++X exprex(sqrt(3+x)::OutputForm)$HTMLFORM
display: S -> Void
    ++ display(o) prints the string returned by coerce.
    ++
    ++X display(coerce(sqrt(3+x)::OutputForm)$HTMLFORM)$HTMLFORM

private == add
import OutputForm
import Character
import Integer
import List OutputForm
import List String

expr: E
prec,opPrec: I
str: S
blank      : S := " \ "

maxPrec    : I := 1000000
minPrec    : I := 0

unaryOps   : L S := ["-"]$(L S)
unaryPrecs : L I := [700]$(L I)

-- the precedence of / in the following is relatively low because
-- the bar obviates the need for parentheses.
binaryOps  : L S := ["+>", "|", "^", "/", "<", ">", "=", "OVER"]$(L S)
binaryPrecs : L I := [0,0,900,700,400,400,400,700]$(L I)
naryOps    : L S := ["-", "+", "*", blank, ",", ";", " ", "ROW", "",
    " \cr", "&", "\", "\/"]$(L S)
naryPrecs  : L I := [700,700,800,800,110,110,0,0,0,0,600,600]$(L I)
naryNGOps  : L S := ["ROW", "&"]$(L S)
plexOps    : L S := ["SIGMA", "SIGMA2", "PI", "PI2", "INTSIGN", _
    "INDEFINTEGRAL"]$(L S)
plexPrecs  : L I := [700,800,700,800,700,700]$(L I)
specialOps : L S := ["MATRIX", "BRACKET", "BRACE", "CONCATB", "VCONCAT", _
    "AGGLST", "CONCAT", "OVERBAR", "ROOT", "SUB", "TAG", _
    "SUPERSUB", "ZAG", "AGGSET", "SC", "PAREN", _
    "SEGMENT", "QUOTE", "theMap", "SLASH"]

-- the next two lists provide translations for some strings for
-- which HTML has some special character codes.
specialStrings : L S :=
["cos", "cot", "csc", "log", "sec", "sin", "tan", _
    "cosh", "coth", "csch", "sech", "sinh", "tanh", _
    "acos", "asin", "atan", "erf", "...", "$", "infinity", "Gamma", _
    "%pi", "%e", "%i"]
specialStringsInHTML : L S :=

```

```

["cos","cot","csc","log","sec","sin","tan", _
 "cosh","coth","csch","sech","sinh","tanh", _
 "arccos","arcsin","arctan","erf",&#x2026;","$",&#x221E;","_
 &#x0413;",&#x003C0;",&#x02147;",&#x02148;"]

debug := false

atomize:E -> L E

formatBinary:(S,L E, I) -> Tree S

formatFunction:(Tree S,L E, I) -> Tree S

formatMatrix:L E -> Tree S

formatNary:(S,L E, I) -> Tree S

formatNaryNoGroup:(S,L E, I) -> Tree S

formatNullary:S -> Tree S

formatPlex:(S,L E, I) -> Tree S

formatSpecial:(S,L E, I) -> Tree S

formatUnary:(S, E, I) -> Tree S

formatHtml:(E,I) -> Tree S

precondition:E -> E
-- this function is applied to the OutputForm expression before
-- doing anything else.

outputTree:Tree S -> Void
-- This function traverses the tree and linierises it into a string.
-- To get the formatting we use a nested set of tables. It also checks
-- for +- and removes the +. it may also need to remove the outer
-- set of brackets.

stringify:E -> S

coerce(expr : E): S ==
  outputTree formatHtml(precondition expr, minPrec)
  " "

coerceS(expr : E): S ==
  outputTree formatHtml(precondition expr, minPrec)
  " "

coerceL(expr : E): S ==

```

```

outputTree formatHtml(precondition expr, minPrec)
" "

display(html : S): Void ==
  sayTeX$Lisp html
  void()$Void

newNode(tag:S,node: Tree S): (Tree S) ==
  t := tree(S,[node])
  setvalue!(t,tag)
  t

newNodes(tag:S,nodes: L Tree S): (Tree S) ==
  t := tree(S,nodes)
  setvalue!(t,tag)
  t

-- returns true if this can be represented without a table
notTable?(node: Tree S): Boolean ==
  empty?(node) => true
  leaf?(node) => true
  prefix?("table",value(node))$String => false
  c := children(node)
  for a in c repeat
    if not notTable?(a) then return false
  true

-- this returns a string representation of OutputForm arguments
-- it is used when debug is true to trace the calling of functions
-- in this package
argsToString(args : L E): S ==
  sop : S := exprex first args
  args := rest args
  s : S := concat ["{",sop]
  for a in args repeat
    s1 : S := exprex a
    s := concat [s,s1]
  s := concat [s,"}"]

exprex(expr : E): S ==
  -- This breaks down an expression into atoms and returns it as
  -- a string. It's for developmental purposes to help understand
  -- the expressions.
  a : E
  expr := precondition expr
  (ATOM(expr)$Lisp@Boolean) or (stringify expr = "NOTHING") =>
    concat ["{",stringify expr,"}"]
  le : L E := (expr pretend L E)
  op := first le
  sop : S := exprex op

```

```

args : L E := rest le
nargs : I := #args
s : S := concat ["{",sop]
if nargs > 0 then
  for a in args repeat
    s1 : S := exprex a
    s := concat [s,s1]
s := concat [s,"}"]

atomize(expr : E): L E ==
-- This breaks down an expression into a flat list of atomic
-- expressions.
-- expr should be preconditioned.
le : L E := nil()
a : E
letmp : L E
(ATOM(expr)$Lisp@Boolean) or (stringify expr = "NOTHING") =>
  le := append(le,list(expr))
letmp := expr pretend L E
for a in letmp repeat
  le := append(le,atomize a)
le

-- output html test using tables and
-- remove unnecessary '+' at end of first string
-- when second string starts with '-'
outputTree(t: Tree S): Void ==
  endWithPlus:Boolean := false -- if the last string ends with '+'
  -- and the next string starts with '-' then the '+' needs to be
  -- removed
  if empty?(t) then
    --if debug then sayTeX$Lisp "outputTree empty"
    return void()$Void
  if leaf?(t) then
    --if debug then sayTeX$Lisp concat("outputTree leaf:",value(t))
    sayTeX$Lisp value(t)
    return void()$Void
  tagName := copy value(t)
  tagPos := position(char(" "),tagName,1)$String
  if tagPos > 1 then
    tagName := split(tagName,char(" ")).1
    --sayTeX$Lisp "outputTree: tagPos="string(tagPos)" "tagName
  if value(t) ~= "" then sayTeX$Lisp concat ["<",value(t),">"]
  c := children(t)
  enableGrid:Boolean := (#c > 1) and not notTable?(t)
  if enableGrid then
    if tagName = "table" then enableGrid := false
    if tagName = "tr" then enableGrid := false
  b:List Boolean := [leaf?(c1) for c1 in c]
  -- if all children are strings then no need to wrap in table

```

```

allString: Boolean := true
for c1 in c repeat if not leaf?(c1) then allString := false
if allString then
  s:String := ""
  for c1 in c repeat s := concat(s,value(c1))
  sayTeX$Lisp s
  if value(t) ~= "" then sayTeX$Lisp concat ["</",tagName,">"]
  return void()$Void
if enableGrid then
  sayTeX$Lisp "<table border='0'>"
  sayTeX$Lisp "<tr>"
  for c1 in c repeat
    if enableGrid then sayTeX$Lisp "<td>"
    outputTree(c1)
    if enableGrid then sayTeX$Lisp "</td>"
  if enableGrid then
    sayTeX$Lisp "</tr>"
    sayTeX$Lisp "</table>"
  if value(t) ~= "" then sayTeX$Lisp concat ["</",tagName,">"]
  void()$Void

stringify expr == (mathObject2String$Lisp expr)@S

precondition expr ==
  outputTran$Lisp expr

-- I dont know what SC is so put it in a table for now
formatSC(args : L E, prec : I) : Tree S ==
  if debug then sayTeX$Lisp "formatSC: "concat [" args=",_
    argsToString(args)," prec=",string(prec)$S]
  null args => tree("")
  cells:L Tree S := [_
    newNode("td id='sc' style='border-bottom-style:solid',_
      formatHtml(a,prec)) for a in args]
  row:Tree S := newNodes("tr id='sc'",cells)
  newNode("table border='0' id='sc'",row)

-- to build an overbar we put it in a single column,
-- single row table and set the top border to solid
buildOverbar(content : Tree S) : Tree S ==
  if debug then sayTeX$Lisp "buildOverbar"
  cell:Tree S := _
    newNode("td id='overbar' style='border-top-style:solid',content)
  row:Tree S := newNode("tr id='overbar'",cell)
  newNode("table border='0' id='overbar'",row)

-- to build an square root we put it in a double column,
-- single row table and set the top border of the second column to
-- solid
buildRoot(content : Tree S) : Tree S ==

```



```

if debug then sayTeX$Lisp "buildRoot"
if leaf?(content) then
  -- root of a single term so no need for overbar
  return newNodes("",[tree("&radic;"),content])
cell1:Tree S := newNode("td id='root'",tree("&radic;"))
cell2:Tree S := _
  newNode("td id='root' style='border-top-style:solid'",content)
row:Tree S := newNodes("tr id='root'",[cell1,cell2])
newNode("table border='0' id='root'",row)

-- to build an 'n'th root we put it in a double column,
-- single row table and set the top border of the second column to
-- solid
buildNRoot(content : Tree S,nth: Tree S) : Tree S ==
  if debug then sayTeX$Lisp "buildNRoot"
  power:Tree S := newNode("sup",nth)
  if leaf?(content) then
    -- root of a single term so no need for overbar
    return newNodes("",[power,tree("&radic;"),content])
  cell1:Tree S := newNodes("td id='nroot'",[power,tree("&radic;")])
  cell2:Tree S := _
    newNode("td id='nroot' style='border-top-style:solid'",content)
  row:Tree S := newNodes("tr id='nroot'",[cell1,cell2])
  newNode("table border='0' id='nroot'",row)

-- formatSpecial handles "theMap","AGGLST","AGGSET","TAG","SLASH",
-- "VCONCAT", "CONCATB","CONCAT","QUOTE","BRACKET","BRACE","PAREN",
-- "OVERBAR","ROOT", "SEGMENT","SC","MATRIX","ZAG"
-- note "SUB" and "SUPERSUB" are handled directly by formatHtml
formatSpecial(op : S, args : L E, prec : I) : Tree S ==
  if debug then sayTeX$Lisp _
    "formatSpecial: " concat ["op=",op," args=",argsToString(args),_
      " prec=",string(prec)$S]
  arg : E
  prescript : Boolean := false
  op = "theMap" => tree("theMap(...)")
  op = "AGGLST" =>
    formatNary(", ",args,prec)
  op = "AGGSET" =>
    formatNary(";",args,prec)
  op = "TAG" =>
    newNodes("",[formatHtml(first args,prec),tree("&#x02192;"),_
      formatHtml(second args,prec)])
    --RightArrow
  op = "SLASH" =>
    newNodes("",[formatHtml(first args, prec),tree("/"),_
      formatHtml(second args,prec)])
  op = "VCONCAT" =>
    newNodes("table",[newNode("td",formatHtml(u, minPrec))_
      for u in args]::L Tree S)

```

```

op = "CONCATB" =>
  formatNary(" ",args,prec)
op = "CONCAT" =>
  formatNary("",args,minPrec)
op = "QUOTE" =>
  newNodes("",[tree("'"),formatHtml(first args, minPrec)])
op = "BRACKET" =>
  newNodes("",[tree("[",formatHtml(first args, minPrec),tree(")"))])
op = "BRACE" =>
  newNodes("",[tree("{",formatHtml(first args, minPrec),tree("}"))])
op = "PAREN" =>
  newNodes("",[tree("(",formatHtml(first args, minPrec),tree(")"))])
op = "OVERBAR" =>
  null args => tree("")
  buildOverbar(formatHtml(first args,minPrec))
op = "ROOT" and #args < 1 => tree("")
op = "ROOT" and #args = 1 => _
  buildRoot(formatHtml(first args, minPrec))
op = "ROOT" and #args > 1 => _
  buildNRoot(formatHtml(first args, minPrec),_
    formatHtml(second args, minPrec))
op = "SEGMENT" =>
  -- '..' indicates a range in a list for example
  tmp : Tree S := newNodes("",[formatHtml(first args, minPrec),_
    tree("..")])
  null rest args => tmp
  newNodes("",[tmp,formatHtml(first rest args, minPrec)])
op = "SC" => formatSC(args,minPrec)
op = "MATRIX" => formatMatrix rest args
op = "ZAG" =>
  -- {{+}}{3}{{ZAG}}{1}{7}}{{ZAG}}{1}{15}}{{ZAG}}{1}{1}}{{ZAG}}{1}{25}}_
  --      {{ZAG}}{1}{1}}{{ZAG}}{1}{7}}{{ZAG}}{1}{4}}
  -- to format continued fraction traditionally need to intercept
  -- it at the formatNary of the "+"
  newNodes("",[tree("\zag{",formatHtml(first args, minPrec),
    tree("}{"),
    formatHtml(first rest args,minPrec),tree("}"))])
  tree("formatSpecial not implemented:"op)

formatSuperSub(expr : E, args : L E, opPrec : I) : Tree S ==
  -- This one produces ordinary derivatives with differential notation,
  -- it needs a little more work yet.
  -- first have to divine the semantics, add cases as needed
  if debug then sayTeX$Lisp _
    "formatSuperSub: " concat ["expr=",stringify expr," args=",_
      argsToString(args)," prec=",string(opPrec)$S]
  atomE : L E := atomize(expr)
  op : S := stringify first atomE
  op ~= "SUPERSUB" => tree("Mistake in formatSuperSub: no SUPERSUB")
  #args ~= 1 => tree("Mistake in SuperSub: #args <> 1")

```

```

var : E := first args
-- should be looking at something like {{SUPERSUB}{var}{ }{,,...}}
-- for example here's the second derivative of y w.r.t. x
-- {{SUPERSUB}{y}{ }{,,}}{x}}, expr is the first {} and args is the
-- {}
funcS : S := stringify first rest atomE
bvarS : S := stringify first args
-- count the number of commas
commaS : S := stringify first rest rest rest atomE
commaTest : S := ","
ndiffs : I := 0
while position(commaTest,commaS,1) > 0 repeat
  ndiffs := ndiffs+1
  commaTest := commaTest","
res:Tree S := newNodes("",_
  [tree("&#x02146;"string(ndiffs)"funcS"&#x02146;"),_
    formatHtml(first args,minPrec),tree("&#x02061;"string(ndiffs)"&#x02061;"),_
    formatHtml(first args,minPrec),tree("")])])
res

-- build structure such as integral as a table
buildPlex3(main:Tree S,supsc:Tree S,op:Tree S,subsc:Tree S) : Tree S ==
  if debug then sayTeX$Lisp "buildPlex"
  ssup:Tree S := newNode("td id='plex'",supsc)
  sop:Tree S := newNode("td id='plex'",op)
  ssub:Tree S := newNode("td id='plex'",subsc)
  m:Tree S := newNode("td rowspan='3' id='plex'",main)
  rows:(List Tree S) := [newNodes("tr id='plex'",[ssup,m]),_
    newNode("tr id='plex'",sop),newNode("tr id='plex'",ssub)]
  newNodes("table border='0' id='plex'",rows)

-- build structure such as integral as a table
buildPlex2(main : Tree S,supsc : Tree S,op : Tree S) : Tree S ==
  if debug then sayTeX$Lisp "buildPlex"
  ssup:Tree S := newNode("td id='plex'",supsc)
  sop:Tree S := newNode("td id='plex'",op)
  m:Tree S := newNode("td rowspan='2' id='plex'",main)
  rows:(List Tree S) := [newNodes("tr id='plex'",[sop,m]),_
    newNode("tr id='plex'",ssup)]
  newNodes("table border='0' id='plex'",rows)

-- format an integral
-- args.1 = "NOTHING"
-- args.2 = bound variable
-- args.3 = body, thing being integrated
--
-- axiom replaces the bound variable with something like
-- %A and puts the original variable used
-- in the input command as a superscript on the integral sign.
formatIntSign(args : L E, opPrec : I) : Tree S ==

```

```

-- the original OutputForm expression looks something like this:
-- {{INTSIGN}{NOTHING or lower limit?}
-- {bvar or upper limit?}{{*}{integrand}{{CONCAT}{d}{axiom var}}}}
-- the args list passed here consists of the rest of this list, i.e.
-- starting at the NOTHING or ...
if debug then sayTeX$Lisp "formatIntSign: " concat [" args=",
  argsToString(args)," prec=",string(opPrec)$S]
(stringify first args) = "NOTHING" =>
  buildPlex2(formatHtml(args.3,opPrec),tree("&int;"),
    formatHtml(args.2,opPrec)) -- could use &#x0222B; or &int;
buildPlex3(formatHtml(first args,opPrec),formatHtml(args.3,opPrec),
  tree("&int;"),formatHtml(args.2,opPrec))

-- plex ops are "SIGMA","SIGMA2","PI","PI2","INTSIGN","INDEFINTEGRAL"
-- expects 2 or 3 args
formatPlex(op : S, args : L E, prec : I) : Tree S ==
  if debug then sayTeX$Lisp "formatPlex: " concat ["op=",op," args=",
    argsToString(args)," prec=",string(prec)$S]
  checkarg:Boolean := false
  hold : S
  p : I := position(op,plexOps)
  p < 1 => error "unknown plex op"
  op = "INTSIGN" => formatIntSign(args,minPrec)
  opPrec := plexPrecs.p
  n : I := #args
  (n ~= 2) and (n ~= 3) => error "wrong number of arguments for plex"
  s : Tree S :=
    op = "SIGMA" =>
      checkarg := true
      tree("&#x02211;")
    -- Sum
    op = "SIGMA2" =>
      checkarg := true
      tree("&#x02211;")
    -- Sum
    op = "PI" =>
      checkarg := true
      tree("&#x0220F;")
    -- Product
    op = "PI2" =>
      checkarg := true
      tree("&#x0220F;")
    -- Product
    op = "INTSIGN" => tree("&#x0222B;")
    -- Integral, int
    op = "INDEFINTEGRAL" => tree("&#x0222B;")
    -- Integral, int
    tree("formatPlex: unexpected op:"op)
-- if opPrec < prec then perhaps we should parenthesize?
-- but we need to be careful we don't get loads of unnecessary

```

```

-- brackets
if n=2 then return buildPlex2(formatHtml(first args,minPrec),_
  formatHtml(args.2,minPrec),s)
buildPlex3(formatHtml(first args,minPrec),formatHtml(args.2,minPrec),_
  s,formatHtml(args.3,minPrec))

-- an example is: op=ROW arg={{ROW}{1}{2}}
formatMatrixRow(op : S, arg : E, prec : I,y:I,h:I) : L Tree S ==
  if debug then sayTeX$Lisp "formatMatrixRow: " concat ["op=",op,_
    " args=",stringify arg," prec=",string(prec)$S]
  ATOM(arg)$Lisp@Boolean => [_
    tree("formatMatrixRow does not contain row")]
  l : L E := (arg pretend L E)
  op : S := stringify first l
  args : L E := rest l
  --sayTeX$Lisp "formatMatrixRow op="op" args="argsToString(args)
  w:I := #args
  cells:(List Tree S) := empty()
  for x in 1..w repeat
    --sayTeX$Lisp "formatMatrixRow: x="string(x)$S" width="string(w)$S
    attrib:S := "td id='mat'"
    if x=1 then attrib := "td id='matl'"
    if x=w then attrib := "td id='matr'"
    if y=1 then attrib := "td id='matt'"
    if y=h then attrib := "td id='matb'"
    if x=1 and y=1 then attrib := "td id='matlt'"
    if x=1 and y=h then attrib := "td id='matlb'"
    if x=w and y=1 then attrib := "td id='matrt'"
    if x=w and y=h then attrib := "td id='matrb'"
    cells := append(cells,[newNode(attrib,formatHtml(args.(x),prec))])
  cells

-- an example is: op=MATRIX args={{ROW}{1}{2}}{{ROW}{3}{4}}
formatMatrixContent(op : S, args : L E, prec : I) : L Tree S ==
  if debug then sayTeX$Lisp "formatMatrixContent: " concat ["op=",op,_
    " args=",argsToString(args)," prec=",string(prec)$S]
  y:I := 0
  rows:(List Tree S) := [newNodes("tr id='mat'",_
    formatMatrixRow("ROW",e,prec,y:=y+1,#args)) for e in args]
  rows

formatMatrix(args : L E) : Tree S ==
  -- format for args is [[ROW ...],[ROW ...],[ROW ...]]
  -- generate string for formatting columns (centered)
  if debug then sayTeX$Lisp "formatMatrix: " concat ["args=",_
    argsToString(args)]
  newNodes("table border='1' id='mat'",_
    formatMatrixContent("MATRIX",args,minPrec))

-- output arguments in column table

```

```

buildColumnTable(elements : List Tree S) : Tree S ==
  if debug then sayTeX$Lisp "buildColumnTable"
  cells:(List Tree S) := [newNode("td id='col'",j) for j in elements]
  rows:(List Tree S) := [newNode("tr id='col'",i) for i in cells]
  newNodes("table border='0' id='col'",rows)

-- build superscript structure as either sup tag or
-- if it contains anything that won't go into a
-- sup tag then build it as a table
buildSuperscript(main : Tree S,super : Tree S) : Tree S ==
  if debug then sayTeX$Lisp "buildSuperscript"
  notTable?(super) => newNodes("",[main,newNode("sup",super)])
  m:Tree S := newNode("td rowspan='2' id='sup'",main)
  su:Tree S := newNode("td id='sup'",super)
  e:Tree S := newNode("td id='sup'",tree("&nbsp;"))
  rows:(List Tree S) := [newNodes("tr id='sup'",[m,su]),_
    newNode("tr id='sup'",e)]
  newNodes("table border='0' id='sup'",rows)

-- build subscript structure as either sub tag or
-- if it contains anything that won't go into a
-- sub tag then build it as a table
buildSubscript(main : Tree S,subsc : Tree S) : Tree S ==
  if debug then sayTeX$Lisp "buildSubscript"
  notTable?(subsc) => newNodes("",[main,newNode("sub",subsc)])
  m:Tree S := newNode("td rowspan='2' id='sub'",main)
  su:Tree S := newNode("td id='sub'",subsc)
  e:Tree S := newNode("td id='sub'",tree("&nbsp;"))
  rows:(List Tree S) := [newNodes("tr id='sub'",[m,e]),_
    newNode("tr id='sub'",su)]
  newNodes("table border='0' id='sub'",rows)

formatSub(expr : E, args : L E, opPrec : I) : Tree S ==
  -- format subscript
  -- this function expects expr to start with SUB
  -- it expects first args to be the operator or value that
  -- the subscript is applied to
  -- and the rest args to be the subscript
  if debug then sayTeX$Lisp "formatSub: " concat ["expr=",_
    stringify expr," args=",argsToString(args)," prec=",_
    string(opPrec)$S]
  atomE : L E := atomize(expr)
  if empty?(atomE) then
    if debug then sayTeX$Lisp "formatSub: expr=empty"
    return tree("formatSub: expr=empty")
  op : S := stringify first atomE
  op ~= "SUB" =>
    if debug then sayTeX$Lisp "formatSub: expr~=SUB"
    tree("formatSub: expr~=SUB")
  -- assume args.1 is the expression and args.2 is its subscript

```

```

if #args < 2 then
  if debug then sayTeX$Lisp concat("formatSub: num args=",_
    string(#args)$String)$String
  return tree(concat("formatSub: num args=",_
    string(#args)$String)$String)
if #args > 2 then
  if debug then sayTeX$Lisp concat("formatSub: num args=",_
    string(#args)$String)$String
  return buildSubscript(formatHtml(first args,opPrec),_
    newNodes("",[formatHtml(e,opPrec) for e in rest args]))
buildSubscript(formatHtml(first args,opPrec),_
  formatHtml(args.2,opPrec))

formatFunction(op : Tree S, args : L E, prec : I) : Tree S ==
  if debug then sayTeX$Lisp "formatFunction: " concat ["args=",_
    argsToString(args)," prec=",string(prec)$S]
  newNodes("",[op,tree("("),formatNary(" ",args,minPrec),tree(")")]])

formatNullary(op : S) : Tree S ==
  if debug then sayTeX$Lisp "formatNullary: " concat ["op=",op]
  op = "NOTHING" => empty()$Tree(S)
  tree(op"()")

-- implement operation with single argument
-- an example is minus '-'
-- prec is precedence of operator, used to force brackets where
-- more tightly bound operation is next to less tightly bound operation
formatUnary(op : S, arg : E, prec : I) : Tree S ==
  if debug then sayTeX$Lisp "formatUnary: " concat ["op=",op," arg=",_
    stringify arg," prec=",string(prec)$S]
  p : I := position(op,unaryOps)
  p < 1 => error "unknown unary op"
  opPrec := unaryPrecs.p
  s : Tree S := newNodes("",[tree(op),formatHtml(arg,opPrec)])
  opPrec < prec => newNodes("",[tree("("),s,tree(")")]])
  s

-- output division with numerator above the denominator
-- implemented as a table
buildOver(top : Tree S,bottom : Tree S) : Tree S ==
  if debug then sayTeX$Lisp "buildOver"
  topCell:Tree S := newNode("td",top)
  bottomCell:Tree S := newNode("td style='border-top-style:solid'",_
    bottom)
  rows:(List Tree S) := [newNode("tr id='col'",topCell),_
    newNode("tr id='col'",bottomCell)]
  newNodes("table border='0' id='col'",rows)

-- op may be: "|","^","/","OVER","+>"
-- note: "+" and "*" are n-ary ops

```

```

formatBinary(op : S, args : L E, prec : I) : Tree S ==
  if debug then sayTeX$Lisp "formatBinary: " concat ["op=",op,_,
    " args=",argsToString(args)," prec=",string(prec)$S]
  p : I := position(op,binaryOps)
  p < 1 => error "unknown binary op"
  opPrec := binaryPrecs.p
  -- if base op is product or sum need to add parentheses
  if ATOM(first args)$Lisp@Boolean then
    opa:S := stringify first args
  else
    la : L E := (first args pretend L E)
    opa : S := stringify first la
  if (opa = "SIGMA" or opa = "SIGMA2" or opa = "PI" or opa = "PI2")_
    and op = "^" then
    s1 : Tree S := newNodes("",[tree("("),formatHtml(first args,_,
      opPrec),tree(")")]])
  else
    s1 : Tree S := formatHtml(first args, opPrec)
    s2 : Tree S := formatHtml(first rest args, opPrec)
    op = "|" => newNodes("",[s1,tree(op),s2])
    op = "^" => buildSuperscript(s1,s2)
    op = "/" => newNodes("",[s1,tree(op),s2])
    op = "OVER" => buildOver(s1,s2)
    op = "+->" => newNodes("",[s1,tree("|&mdash;&rsquo;"),s2])
    newNodes("",[s1,tree(op),s2])

-- build a zag from a table with a right part and a
-- upper and lower left part
buildZag(top:Tree S,lowerLeft:Tree S,lowerRight:Tree S) : Tree S ==
  if debug then sayTeX$Lisp "buildZag"
  cellTop:Tree S := _
    newNode("td colspan='2' id='zag' style='border-bottom-style:solid',_,
      top)
  cellLowerLeft:Tree S := newNodes("td id='zag'",[lowerLeft,tree("+")])
  cellLowerRight:Tree S := newNode("td id='zag'",lowerRight)
  row1:Tree S := newNodes("tr id='zag'",[cellTop])
  row2:Tree S := newNodes("tr id='zag'",[cellLowerLeft,cellLowerRight])
  newNodes("table border='0' id='zag',[row1,row2])

formatZag(args : L E,nestLevel:I) : Tree S ==
  -- args will be a list of things like this {{ZAG}{1}{7}}, the ZAG
  -- must be there, the '1' and '7' could conceivably be more complex
  -- expressions
  --
  -- ex 1. continuedFraction(314159/100000)
  -- {{+}}{{3}}{{ZAG}{1}{7}}{{ZAG}{1}{15}}{{ZAG}{1}{1}}{{ZAG}{1}{25}}
  -- {{ZAG}{1}{1}}{{ZAG}{1}{7}}{{ZAG}{1}{4}}
  -- this is the preconditioned output form
  -- including "op", the args list would be the rest of this
  -- i.e op = '+' and args = {{3}}{{ZAG}{1}{7}}{{ZAG}{1}{15}}

```



```

-- {{ZAG}{1}{1}}{{ZAG}{1}{25}}{{ZAG}{1}{1}}{{ZAG}{1}{7}}{{ZAG}{1}{4}}}
--
-- ex 2. continuedFraction(14159/100000)
-- this one doesn't have the leading integer
-- {{+}}{{ZAG}{1}{7}}{{ZAG}{1}{15}}{{ZAG}{1}{1}}{{ZAG}{1}{25}}
-- {{ZAG}{1}{1}}{{ZAG}{1}{7}}{{ZAG}{1}{4}}}
--
-- ex 3. continuedFraction(3,repeating [1], repeating [3,6])
-- {{+}}{{3}}{{ZAG}{1}{3}}{{ZAG}{1}{6}}{{ZAG}{1}{3}}{{ZAG}{1}{6}}}
-- {{ZAG}{1}{3}}{{ZAG}{1}{6}}{{ZAG}{1}{3}}{{ZAG}{1}{6}}}
-- {{ZAG}{1}{3}}{{ZAG}{1}{6}}{...}}
--
-- In each of these examples the args list consists of the terms
-- following the '+' op
-- so the first arg could be a "ZAG" or something
-- else, but the second arg looks like it has to be "ZAG", so maybe
-- test for #args > 1 and args.2 contains "ZAG".
-- Note that since the resulting tables are nested we need
-- to handle the whole continued fraction at once, i.e. we can't
-- just look for, e.g., {{ZAG}{1}{6}}}
--
-- we will assume that the font starts at 16px and reduce it by 4
-- <span style='font-size:16px'>outer zag</span>
-- <span style='font-size:14px'>next zag</span>
-- <span style='font-size:12px'>next zag</span>
-- <span style='font-size:10px'>next zag</span>
-- <span style='font-size:9px'>lowest zag</span>
if debug then sayTeX$Lisp "formatZag: " concat ["args=",_
  argsToString(args)]
tmpZag : L E := first args pretend L E
fontAttrib : S :=
  nestLevel < 2 => "span style='font-size:16px'"
  nestLevel = 2 => "span style='font-size:14px'"
  nestLevel = 3 => "span style='font-size:12px'"
  nestLevel = 4 => "span style='font-size:10px'"
  "span style='font-size:9px'"
-- may want to test that tmpZag contains 'ZAG'
#args > 1 =>
  newNode(fontAttrib,buildZag(formatHtml(first rest tmpZag,minPrec),_
    formatHtml(first rest rest tmpZag,minPrec),_
    formatZag(rest args,nestLevel+1)))
  (first args = "...": E)@Boolean => tree("&#x2026;")
op:S := stringify first args
position("ZAG",op,1) > 0 =>
  newNode(fontAttrib,buildOver(formatHtml(first rest tmpZag,minPrec),_
    formatHtml(first rest rest tmpZag,minPrec)))
  tree("formatZag: Last argument in ZAG construct unknown operator: "op)

-- returns true if this term starts with a minus '-' sign
-- this is used so that we can suppress any plus '+' in front

```

```

-- of the - so we dont get terms like +-
neg?(arg : E) : Boolean ==
  if debug then sayTeX$Lisp "neg?: " concat ["arg=",argsToString([arg])]
  ATOM(arg)$Lisp@Boolean => false
  l : L E := (arg pretend L E)
  op : S := stringify first l
  op = "-" => true
  false

formatNary(op : S, args : L E, prec : I) : Tree S ==
  if debug then sayTeX$Lisp "formatNary: " concat ["op=",op," args=",_
    argsToString(args)," prec=",string(prec)$S]
  formatNaryNoGroup(op, args, prec)

-- possible op values are:
-- ",", ";", "*", " ", "ROW", "+", "-"
-- an example is content of matrix which gives:
-- {{ROW}{1}{2}}{{ROW}{3}{4}}
-- or AGGLST which gives op=, args={{1}{2}}
--
-- need to:
-- format ZAG
-- check for +-
-- add brackets for sigma or pi or root ("SIGMA","SIGMA2","PI","PI2")
formatNaryNoGroup(op : S, args : L E, prec : I) : Tree S ==
  if debug then sayTeX$Lisp "formatNaryNoGroup: " concat ["op=",op,_
    " args=",argsToString(args)," prec=",string(prec)$S]
  checkargs:Boolean := false
  null args => empty()$Tree(S)
  p : I := position(op,naryOps)
  p < 1 => error "unknown nary op"
  -- need to test for "ZAG" case and divert it here
  (#args > 1) and (position("ZAG",stringify first rest args,1) > 0) =>
    tmpS : S := stringify first args
    position("ZAG",tmpS,1) > 0 => formatZag(args,1)
    newNodes("",[formatHtml(first args,minPrec),tree("+"),_
      formatZag(rest args,1)])
  -- At least for the ops "*", "+", "-" we need to test to see if a
  -- sigma or pi is one of their arguments because we might need
  -- parentheses as indicated
  -- by the problem with summation(operator(f)(i),i=1..n)+1 versus
  -- summation(operator(f)(i)+1,i=1..n) having identical displays as of
  -- 2007-12-21
  l := empty()$Tree(S)
  opPrec := naryPrecs.p
  -- if checkargs is true check each arg except last one to see if it's
  -- a sigma or pi and if so add parentheses. Other op's may have to be
  -- checked for in future
  count:I := 1
  tags : (L Tree S)

```

```

if opPrec < prec then tags := [tree("("),formatHtml(args.1,opPrec)]
if opPrec >= prec then tags := [formatHtml(args.1,opPrec)]
for a in rest args repeat
  if op ~= "+" or not neg?(a) then tags := append(tags,[tree(op)])
  tags := append(tags,[formatHtml(a,opPrec)])
if opPrec < prec then tags := append(tags,[tree(")")] )
newNodes("",tags)

-- expr is a tree structure
-- prec is the precision of integers
-- formatHtml returns a string for this node in the tree structure
-- and calls recursively to evaluate sub expressions
formatHtml(arg : E,prec : I) : Tree S ==
  if debug then sayTeX$Lisp "formatHtml: " concat ["arg=",_
    argsToString([arg])," prec=",string(prec)$S]
  i,len : Integer
  intSplitLen : Integer := 20
  ATOM(arg)$Lisp@Boolean =>
    if debug then sayTeX$Lisp "formatHtml atom: " concat ["expr=",_
      stringify arg," prec=",string(prec)$S]
    str := stringify arg
    (i := position(str,specialStrings)) > 0 =>
      tree(specialStringsInHTML.i)
    tree(str)
  l : L E := (arg pretend L E)
  null l => tree(blank)
  op : S := stringify first l
  args : L E := rest l
  nargs : I := #args
  -- need to test here in case first l is SUPERSUB case and then
  -- pass first l and args to formatSuperSub.
  position("SUPERSUB",op,1) > 0 =>
    formatSuperSub(first l,args,minPrec)
  -- now test for SUB
  position("SUB",op,1) > 0 =>
    formatSub(first l,args,minPrec)
  -- special cases
  -- specialOps are:
  -- MATRIX, BRACKET, BRACE, CONCATB, VCONCAT
  -- AGGLST, CONCAT, OVERBAR, ROOT, SUB, TAG
  -- SUPERSUB, ZAG, AGGSET, SC, PAREN
  -- SEGMENT, QUOTE, theMap, SLASH
  member?(op, specialOps) => formatSpecial(op,args,prec)
  -- specialOps are:
  -- SIGMA, SIGMA2, PI, PI2, INTSIGN, INDEFINTEGRAL
  member?(op, plexOps) => formatPlex(op,args,prec)
  -- nullary case: function with no arguments
  0 = nargs => formatNullary op
  -- unary case: function with one argument such as '-'
  (1 = nargs) and member?(op, unaryOps) =>

```



```

--R ?? : (%,Integer) -> S                               elt : (%,Integer,S) -> S
--R empty : () -> %                                       empty? : % -> Boolean
--R entries : % -> List(S)                                eq? : (%,%)-> Boolean
--R index? : (Integer,%)-> Boolean                        indices : % -> List(Integer)
--R map : ((S -> S),%) -> %                               qelt : (%,Integer) -> S
--R sample : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?? : (PositiveInteger,%)-> % if S has ABELSG
--R ?? : (NonNegativeInteger,%)-> % if S has CABMON
--R ?? : (S,%)-> % if S has MONOID
--R ?? : (%S)-> % if S has MONOID
--R ?? : (%,%)-> % if S has MONOID
--R ?? : (Integer,%)-> % if S has RING
--R ??? : (%PositiveInteger)-> % if S has MONOID
--R ??? : (%NonNegativeInteger)-> % if S has MONOID
--R +? : (%,%)-> % if S has ABELSG
--R -? : (%,%)-> % if S has RING
--R /? : (%S)-> % if S has FIELD
--R <? : (%,%)-> Boolean if S has OAMONS or S has ORDRING
--R <=? : (%,%)-> Boolean if S has OAMONS or S has ORDRING
--R ==? : (%,%)-> Boolean if S has SETCAT
--R >? : (%,%)-> Boolean if S has OAMONS or S has ORDRING
--R >=? : (%,%)-> Boolean if S has OAMONS or S has ORDRING
--R D : (%(S -> S))-> % if S has RING
--R D : (%(S -> S),NonNegativeInteger)-> % if S has RING
--R D : (%List(Symbol),List(NonNegativeInteger))-> % if S has PDRING(SYMBOL) and S has RING
--R D : (%Symbol,NonNegativeInteger)-> % if S has PDRING(SYMBOL) and S has RING
--R D : (%List(Symbol))-> % if S has PDRING(SYMBOL) and S has RING
--R D : (%Symbol)-> % if S has PDRING(SYMBOL) and S has RING
--R D : (%NonNegativeInteger)-> % if S has DIFRING and S has RING
--R D : % -> % if S has DIFRING and S has RING
--R ?? : (%PositiveInteger)-> % if S has MONOID
--R ?? : (%NonNegativeInteger)-> % if S has MONOID
--R abs : % -> % if S has ORDRING
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R characteristic : () -> NonNegativeInteger if S has RING
--R coerce : S -> % if S has SETCAT
--R coerce : Fraction(Integer) -> % if S has RETRACT(FRAC(INT)) and S has SETCAT
--R coerce : Integer -> % if S has RETRACT(INT) and S has SETCAT or S has RING
--R coerce : % -> OutputForm if S has SETCAT
--R count : (S,%)-> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R differentiate : (%(S -> S))-> % if S has RING
--R differentiate : (%(S -> S),NonNegativeInteger)-> % if S has RING
--R differentiate : (%List(Symbol),List(NonNegativeInteger))-> % if S has PDRING(SYMBOL) and S
--R differentiate : (%Symbol,NonNegativeInteger)-> % if S has PDRING(SYMBOL) and S has RING
--R differentiate : (%List(Symbol))-> % if S has PDRING(SYMBOL) and S has RING
--R differentiate : (%Symbol)-> % if S has PDRING(SYMBOL) and S has RING
--R differentiate : (%NonNegativeInteger)-> % if S has DIFRING and S has RING
--R differentiate : % -> % if S has DIFRING and S has RING

```

```

--R dimension : () -> CardinalNumber if S has FIELD
--R dot : (%,%) -> S if S has RING
--R entry? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R enumerate : () -> List(%) if S has FINITE
--R eval : (%,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (%,S) -> % if $ has shallowlyMutable
--R first : % -> S if Integer has ORDSET
--R hash : % -> SingleInteger if S has SETCAT
--R index : PositiveInteger -> % if S has FINITE
--R latex : % -> String if S has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R lookup : % -> PositiveInteger if S has FINITE
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R max : (%,%) -> % if S has OAMONS or S has ORDRING
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R members : % -> List(S) if $ has finiteAggregate
--R min : (%,%) -> % if S has OAMONS or S has ORDRING
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (%,NonNegativeInteger) -> Boolean
--R negative? : % -> Boolean if S has ORDRING
--R one? : % -> Boolean if S has MONOID
--R parts : % -> List(S) if $ has finiteAggregate
--R positive? : % -> Boolean if S has ORDRING
--R qsetelt! : (%,Integer,S) -> S if $ has shallowlyMutable
--R random : () -> % if S has FINITE
--R recip : % -> Union(%, "failed") if S has MONOID
--R reducedSystem : Matrix(%) -> Matrix(S) if S has RING
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(S),vec: Vector(S)) if S has RING
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) if S has LIN
--R reducedSystem : Matrix(%) -> Matrix(Integer) if S has LINEXP(INT) and S has RING
--R retract : % -> S if S has SETCAT
--R retract : % -> Fraction(Integer) if S has RETRACT(FRAC(INT)) and S has SETCAT
--R retract : % -> Integer if S has RETRACT(INT) and S has SETCAT
--R retractIfCan : % -> Union(S, "failed") if S has SETCAT
--R retractIfCan : % -> Union(Fraction(Integer), "failed") if S has RETRACT(FRAC(INT)) and S has SETCAT
--R retractIfCan : % -> Union(Integer, "failed") if S has RETRACT(INT) and S has SETCAT
--R setelt : (%,Integer,S) -> S if $ has shallowlyMutable
--R sign : % -> Integer if S has ORDRING
--R size : () -> NonNegativeInteger if S has FINITE
--R size? : (%,NonNegativeInteger) -> Boolean
--R subtractIfCan : (%,%) -> Union(%, "failed") if S has CABMON
--R sup : (%,%) -> % if S has OAMONS
--R swap! : (%,Integer,Integer) -> Void if $ has shallowlyMutable
--R unitVector : PositiveInteger -> % if S has RING
--R zero? : % -> Boolean if S has CABMON

```

```
--R ?~=? : (%,% ) -> Boolean if S has SETCAT
--R
--E 1

)spool
)lisp (bye)
```

— HomogeneousDirectProduct.help —

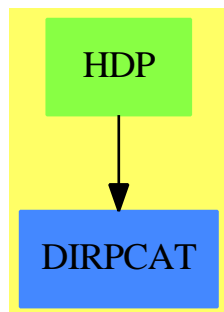
```
=====
HomogeneousDirectProduct examples
=====
```

This type represents the finite direct or cartesian product of an underlying ordered component type. The vectors are ordered first by the sum of their components, and then refined using a reverse lexicographic ordering. This type is a suitable third argument for GeneralDistributedMultivariatePolynomial.

See Also:

o)show HomogeneousDirectProduct

HomogeneousDirectProduct (HDP)



See

⇒ “OrderedDirectProduct” (ODP) [16](#) on page [1816](#)

⇒ “SplitHomogeneousDirectProduct” (SHDP) [20](#) on page [2508](#)

Exports:

0	1	abs	any?	characteristic
coerce	copy	count	D	differentiate
dimension	directProduct	dot	elt	empty
empty?	entries	entry?	eq?	eval
every?	fill!	first	hash	index
index?	indices	latex	less?	lookup
map	map!	max	maxIndex	member?
members	min	minIndex	more?	negative?
one?	parts	positive?	qelt	qsetelt!
random	recip	reducedSystem	retract	retractIfCan
sample	setelt	sign	size	size?
subtractIfCan	sup	swap!	unitVector	zero?
#?	?*	?**?	?+?	?-?
?/?	?<?	?<=?	?=?	?>?
?>=?	?^?	?~=?	~?	..?

— domain HDP HomogeneousDirectProduct —

```

)abbrev domain HDP HomogeneousDirectProduct
++ Author: Mark Botch
++ Description:
++ This type represents the finite direct or cartesian product of an
++ underlying ordered component type. The vectors are ordered first
++ by the sum of their components, and then refined using a reverse
++ lexicographic ordering. This type is a suitable third argument for
++ \spadtype{GeneralDistributedMultivariatePolynomial}.

HomogeneousDirectProduct(dim,S) : T == C where
  dim : NonNegativeInteger
  S      : OrderedAbelianMonoidSup

  T == DirectProductCategory(dim,S)
  C == DirectProduct(dim,S) add
    Rep:=Vector(S)
    v1:% < v2:% ==
-- reverse lexicographical ordering
    n1:S:=0
    n2:S:=0
    for i in 1..dim repeat
      n1:= n1+qelt(v1,i)
      n2:=n2+qelt(v2,i)
    n1<n2 => true
    n2<n1 => false
    for i in reverse(1..dim) repeat
      if qelt(v2,i) < qelt(v1,i) then return true
      if qelt(v1,i) < qelt(v2,i) then return false
    false

```

— HDP.dotabb —

```
"HDP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=HDP"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"HDP" -> "DIRPCAT"
```

domain HDMP HomogeneousDistributedMultivariatePolynomial

— HomogeneousDistributedMultivariatePolynomial.input —

```
)set break resume
)sys rm -f HomogeneousDistributedMultivariatePolynomial.output
)spool HomogeneousDistributedMultivariatePolynomial.output
)set message test on
)set message auto off
)clear all
--S 1 of 10
(d1,d2,d3) : DMP([z,y,x],FRAC INT)
--R
--R
--R                                          Type: Void
--E 1

--S 2 of 10
d1 := -4*z + 4*y**2*x + 16*x**2 + 1
--R
--R
--R              2      2
--R      (2)  - 4z + 4y x + 16x  + 1
--R              Type: DistributedMultivariatePolynomial([z,y,x],Fraction(Integer))
--E 2

--S 3 of 10
d2 := 2*z*y**2 + 4*x + 1
--R
--R
--R              2
--R      (3)  2z y  + 4x + 1
--R              Type: DistributedMultivariatePolynomial([z,y,x],Fraction(Integer))
--E 3
```

DOMAIN HDMP HOMOGENEOUSDISTRIBUTEDMULTIVARIATEPOLYNOMIAL1175

```

--S 4 of 10
d3 := 2*z*x**2 - 2*y**2 - x
--R
--R
--R      2      2
--R  (4)  2z x  - 2y  - x
--R      Type: DistributedMultivariatePolynomial([z,y,x],Fraction(Integer))
--E 4

--S 5 of 10
groebner [d1,d2,d3]
--R
--R
--R  (5)
--R      1568  6      1264  5      6  4      182  3      2047  2      103      2857
--R  [z - ---- x  - ---- x  + --- x  + --- x  - ---- x  - ---- x  - ----,
--R      2745      305      305      549      610      2745      10980
--R      2      112  6      84  5      1264  4      13  3      84  2      1772      2
--R  y  + ---- x  - --- x  - ---- x  - --- x  + --- x  + ---- x  + ----,
--R      2745      305      305      549      305      2745      2745
--R      7  29  6      17  4      11  3      1  2      15      1
--R  x  + -- x  - -- x  - -- x  + -- x  + -- x  + -]
--R      4      16      8      32      16      4
--R      Type: List(DistributedMultivariatePolynomial([z,y,x],Fraction(Integer)))
--E 5

--S 6 of 10
(n1,n2,n3) : HDMP([z,y,x],FRAC INT)
--R
--R
--R                                          Type: Void
--E 6

--S 7 of 10
n1 := d1
--R
--R
--R      2      2
--R  (7)  4y x  + 16x  - 4z + 1
--R      Type: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction(Integer))
--E 7

--S 8 of 10
n2 := d2
--R
--R
--R      2
--R  (8)  2z y  + 4x + 1
--R      Type: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction(Integer))
--E 8

```

```

--S 9 of 10
n3 := d3
--R
--R
--R      2      2
--R      (9) 2z x - 2y - x
--RType: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction(Integer))
--E 9

--S 10 of 10
groebner [n1,n2,n3]
--R
--R
--R      (10)
--R      4      3      3      2      1      1      4      29      3      1      2      7      9      1
--R      [y + 2x - - x + - z - -, x + - x - - y - - z x - - x - -,
--R      2      2      8      4      8      4      16      4
--R      2      1      2      2      1      2      2      1
--R      z y + 2x + -, y x + 4x - z + -, z x - y - - x,
--R      2      4      2
--R      2      2      2      1      3
--R      z - 4y + 2x - - z - - x]
--R      4      2
--RType: List(HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction(Integer)))
--E 10
)spool
)lisp (bye)

```

— HomogeneousDistributedMultivariatePolynomial.help —

```

=====
MultivariatePolynomial
DistributedMultivariatePolynomial
HomogeneousDistributedMultivariatePolynomial
GeneralDistributedMultivariatePolynomial
=====

```

DistributedMultivariatePolynomial which is abbreviated as DMP and HomogeneousDistributedMultivariatePolynomial, which is abbreviated as HDMP, are very similar to MultivariatePolynomial except that they are represented and displayed in a non-recursive manner.

```

(d1,d2,d3) : DMP([z,y,x],FRAC INT)
Type: Void

```

The constructor DMP orders its monomials lexicographically while HDMP orders them by total order refined by reverse lexicographic

order.

```

d1 := -4*z + 4*y**2*x + 16*x**2 + 1
      2      2
    - 4z + 4y x + 16x  + 1
      Type: DistributedMultivariatePolynomial([z,y,x],Fraction Integer)

d2 := 2*z*y**2 + 4*x + 1
      2
    2z y  + 4x + 1
      Type: DistributedMultivariatePolynomial([z,y,x],Fraction Integer)

d3 := 2*z*x**2 - 2*y**2 - x
      2      2
    2z x  - 2y  - x
      Type: DistributedMultivariatePolynomial([z,y,x],Fraction Integer)

```

These constructors are mostly used in Groebner basis calculations.

```

groebner [d1,d2,d3]
      1568 6 1264 5 6 4 182 3 2047 2 103 2857
    [z - ---- x - ---- x + --- x + --- x - ---- x - ---- x - ----,
      2745 305 305 549 610 2745 10980
      2 112 6 84 5 1264 4 13 3 84 2 1772 2
    y + ---- x - --- x - ---- x - --- x + --- x + ---- x + ----,
      2745 305 305 549 305 2745 2745
      7 29 6 17 4 11 3 1 2 15 1
    x + -- x - -- x - -- x + -- x + -- x + -]
      4 16 8 32 16 4
      Type: List DistributedMultivariatePolynomial([z,y,x],Fraction Integer)

(n1,n2,n3) : HDMP([z,y,x],FRAC INT)
      Type: Void

```

```

n1 := d1
      2      2
    4y x + 16x  - 4z + 1
Type: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction Integer)

n2 := d2
      2
    2z y  + 4x + 1
Type: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction Integer)

n3 := d3
      2      2
    2z x  - 2y  - x
Type: HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction Integer)

```

Note that we get a different Groebner basis when we use the HDMP

polynomials, as expected.

```

groebner [n1,n2,n3]
      4      3      3      2      1      1      4      29      3      1      2      7      9      1
[y  + 2x  - - x  + - z - -, x  + -- x  - - y  - - z x - -- x - -,
      2      2      8      4      8      4      16      4
      2      1      2      2      1      2      2      1
z y  + 2x + -, y x + 4x  - z + -, z x  - y  - - x,
      2      2      4      2
      2      2      2      1      3
z  - 4y  + 2x  - - z - - x]
      4      2
Type: List HomogeneousDistributedMultivariatePolynomial([z,y,x],
Fraction Integer)

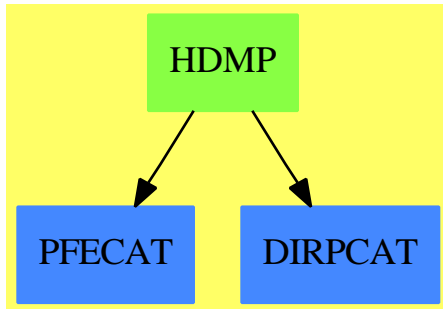
```

GeneralDistributedMultivariatePolynomial is somewhat more flexible in the sense that as well as accepting a list of variables to specify the variable ordering, it also takes a predicate on exponent vectors to specify the term ordering. With this polynomial type the user can experiment with the effect of using completely arbitrary term orderings. This flexibility is mostly important for algorithms such as Groebner basis calculations which can be very sensitive to term ordering.

See Also:

- o)help Polynomial
- o)help UnivariatePolynomial
- o)help MultivariatePolynomial
- o)help DistributedMultivariatePolynomial
- o)help GeneralDistributedMultivariatePolynomial
- o)show HomogeneousDistributedMultivariatePolynomial

HomogeneousDistributedMultivariatePolynomial (HDMP)



See

⇒ “GeneralDistributedMultivariatePolynomial” (GDMP) [8](#) on page [1052](#)

⇒ “DistributedMultivariatePolynomial” (DMP) [5](#) on page [571](#)

Exports:

0	1	associates?
binomThmExpt	characteristic	charthRoot
coefficient	coefficients	coerce
conditionP	content	convert
D	degree	differentiate
discriminant	eval	exquo
factor	factorPolynomial	factorSquareFreePolynomial
gcd	gcdPolynomial	ground
ground?	hash	isExpt
isPlus	isTimes	latex
lcm	leadingCoefficient	leadingMonomial
mainVariable	map	mapExponents
max	min	minimumDegree
monicDivide	monomial	monomial?
monomials	multivariate	numberOfMonomials
one?	patternMatch	pomopo!
prime?	primitiveMonomials	primitivePart
recip	reducedSystem	reductum
reorder	resultant	retract
retractIfCan	sample	solveLinearPolynomialEquation
squareFree	squareFreePart	squareFreePolynomial
subtractIfCan	totalDegree	unit?
unitCanonical	unitNormal	univariate
variables	zero?	?*?
?**?	?+?	?-?
-?	?=?	?^?
?~=?	?/?	?<?
?<=?	?>?	?>=?
?^?		

— **domain HDMP HomogeneousDistributedMultivariatePolynomial**

```

)abbrev domain HDMP HomogeneousDistributedMultivariatePolynomial
++ Author: Barry Trager
++ Description:
++ This type supports distributed multivariate polynomials
++ whose variables are from a user specified list of symbols.
++ The coefficient ring may be non commutative,
++ but the variables are assumed to commute.
++ The term ordering is total degree ordering refined by reverse
++ lexicographic ordering with respect to the position that the variables
++ appear in the list of variables parameter.

HomogeneousDistributedMultivariatePolynomial(vl,R): public == private where
  vl : List Symbol

```

```

R : Ring
E ==> HomogeneousDirectProduct(#v1,NonNegativeInteger)
OV ==> OrderedVariableList(v1)
public == PolynomialCategory(R,E,Ov) with
  reorder: (%,List Integer) -> %
  ++ reorder(p, perm) applies the permutation perm to the variables
  ++ in a polynomial and returns the new correctly ordered polynomial
private ==
  GeneralDistributedMultivariatePolynomial(v1,R,E)

```

— HDMP.dotabb —

```

"HDMP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=HDMP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"HDMP" -> "PFECAT"
"HDMP" -> "DIRPCAT"

```

domain HELLDIV HyperellipticFiniteDivisor

— HyperellipticFiniteDivisor.input —

```

)set break resume
)sys rm -f HyperellipticFiniteDivisor.output
)spool HyperellipticFiniteDivisor.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show HyperellipticFiniteDivisor
--R
--R HyperellipticFiniteDivisor(F: Field,UP: UnivariatePolynomialCategory(F),UPUP: UnivariatePolynomialCategory(F))
--R Abbreviation for HyperellipticFiniteDivisor is HELLDIV
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for HELLDIV
--R
--R----- Operations -----
--R ?? : (Integer,%) -> %          ?? : (PositiveInteger,%) -> %
--R ?+? : (%,%) -> %             ?-? : (%,%) -> %

```



```

--R -? : % -> %
--R 0 : () -> %
--R divisor : (R,UP,UP,UP,F) -> %
--R divisor : (F,F) -> %
--R hash : % -> SingleInteger
--R principal? : % -> Boolean
--R sample : () -> %
--R ?=? : (%,% ) -> Boolean
--R ?*? : (NonNegativeInteger,% ) -> %
--R decompose : % -> Record(id: FractionalIdeal(UP,Fraction(UP),UPUP,R),principalPart: R)
--R divisor : FractionalIdeal(UP,Fraction(UP),UPUP,R) -> %
--R generator : % -> Union(R,"failed")
--R ideal : % -> FractionalIdeal(UP,Fraction(UP),UPUP,R)
--R subtractIfCan : (%,% ) -> Union(%,"failed")
--R
--E 1

)spool
)lisp (bye)

```

— HyperellipticFiniteDivisor.help —

=====

HyperellipticFiniteDivisor examples

=====

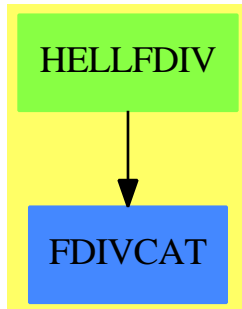
This domains implements finite rational divisors on an hyperelliptic curve, that is finite formal sums $\sum(n * P)$ where the n 's are integers and the P 's are finite rational points on the curve.

The equation of the curve must be $y^2 = f(x)$ and f must have odd degree.

See Also:

o)show HyperellipticFiniteDivisor

HyperellipticFiniteDivisor (HELLFDIV)



See

⇒ “FractionalIdeal” (FRIDEAL) [7](#) on page [994](#)

⇒ “FramedModule” (FRMOD) [7](#) on page [1000](#)

⇒ “FiniteDivisor” (FDIV) [7](#) on page [813](#)

Exports:

0	coerce	decompose	divisor	hash
ideal	generator	latex	principal?	reduce
sample	subtractIfCan	zero?	?~=?	?*?
?+?	?-?	-?	?=?	

— domain HELLFDIV HyperellipticFiniteDivisor —

```

)abbrev domain HELLFDIV HyperellipticFiniteDivisor
++ Author: Manuel Bronstein
++ Date Created: 19 May 1993
++ Date Last Updated: 20 July 1998
++ Description:
++ This domains implements finite rational divisors on an hyperelliptic curve,
++ that is finite formal sums SUM(n * P) where the n's are integers and the
++ P's are finite rational points on the curve.
++ The equation of the curve must be y^2 = f(x) and f must have odd degree.

```

```

HyperellipticFiniteDivisor(F, UP, UPUP, R): Exports == Implementation where

```

```

  F   : Field
  UP   : UnivariatePolynomialCategory F
  UPUP: UnivariatePolynomialCategory Fraction UP
  R    : FunctionFieldCategory(F, UP, UPUP)

```

```

  O ==> OutputForm
  Z ==> Integer
  RF ==> Fraction UP
  ID ==> FractionalIdeal(UP, RF, UPUP, R)
  ERR ==> error "divisor: incomplete implementation for hyperelliptic curves"

```

```

Exports ==> FiniteDivisorCategory(F, UP, UPUP, R)

```

```

Implementation ==> add
  if (uhyper:Union(UP, "failed") := hyperelliptic()$R) case "failed" then
    error "HyperellipticFiniteDivisor: curve must be hyperelliptic"

-- we use the semi-reduced representation from D.Cantor, "Computing in the
-- Jacobian of a HyperellipticCurve", Mathematics of Computation, vol 48,
-- no.177, January 1987, 95-101.
-- The representation [a,b,f] for D means D = [a,b] + div(f)
-- and [a,b] is a semi-reduced representative on the Jacobian
Rep := Record(center:UP, polyPart:UP, principalPart:R, reduced?:Boolean)

hyper:UP := uhyper::UP
gen:Z     := ((degree(hyper)):Z - 1) exquo 2)::Z      -- genus of the curve
dvd:0     := "div"::Symbol::0
zer:0     := 0::Z::0

makeDivisor : (UP, UP, R) -> %
intReduc    : (R, UP) -> R
princ?      : % -> Boolean
polyIfCan   : R -> Union(UP, "failed")
redpolyIfCan : (R, UP) -> Union(UP, "failed")
intReduce   : (R, UP) -> R
mkIdeal     : (UP, UP) -> ID
reducedTimes : (Z, UP, UP) -> %
reducedDouble : (UP, UP) -> %

0 == divisor(1$R)
divisor(g:R) == [1, 0, g, true]
makeDivisor(a, b, g) == [a, b, g, false]
-- princ? d == one?(d.center) and zero?(d.polyPart)
princ? d == (d.center = 1) and zero?(d.polyPart)
ideal d == ideal([d.principalPart]) * mkIdeal(d.center, d.polyPart)
decompose d == [ideal makeDivisor(d.center, d.polyPart, 1), d.principalPart]
mkIdeal(a, b) == ideal [a::RF::R, reduce(monomial(1, 1)$UPUP-b::RF::UPUP)]

-- keep the sum reduced if d1 and d2 are both reduced at the start
d1 + d2 ==
  a1 := d1.center; a2 := d2.center
  b1 := d1.polyPart; b2 := d2.polyPart
  rec := principalIdeal [a1, a2, b1 + b2]
  d := rec.generator
  h := rec.coef -- d = h1 a1 + h2 a2 + h3(b1 + b2)
  a := ((a1 * a2) exquo d**2)::UP
  b:UP:= first(h) * a1 * b2
  b := b + second(h) * a2 * b1
  b := b + third(h) * (b1*b2 + hyper)
  b := (b exquo d)::UP rem a
  dd := makeDivisor(a, b, d::RF * d1.principalPart * d2.principalPart)
  d1.reduced? and d2.reduced? => reduce dd

```

```

dd

-- if is cheaper to keep on reducing as we exponentiate if d is already reduced
n:Z * d:% ==
  zero? n => 0
  n < 0 => (-n) * (-d)
  divisor(d.principalPart ** n) + divisor(mkIdeal(d.center,d.polyPart)**n)

divisor(i:ID) ==
--   one?(n := #(v := basis minimize i)) => divisor v minIndex v
   (n := #(v := basis minimize i)) = 1 => divisor v minIndex v
   n ^= 2 => ERR
   a := v minIndex v
   h := v maxIndex v
   (u := polyIfCan a) case UP =>
     (w := redpolyIfCan(h, u::UP)) case UP => makeDivisor(u::UP, w::UP, 1)
     ERR
   (u := polyIfCan h) case UP =>
     (w := redpolyIfCan(a, u::UP)) case UP => makeDivisor(u::UP, w::UP, 1)
     ERR
   ERR

polyIfCan a ==
  (u := retractIfCan(a)@Union(RF, "failed")) case "failed" => "failed"
  (v := retractIfCan(u::RF)@Union(UP, "failed")) case "failed" => "failed"
  v::UP

redpolyIfCan(h, a) ==
  degree(p := lift h) ^= 1 => "failed"
  q := - coefficient(p, 0) / coefficient(p, 1)
  rec := extendedEuclidean(denom q, a)
  not ground?(rec.generator) => "failed"
  ((numer(q) * rec.coef1) exquo rec.generator)::UP rem a

coerce(d:%):0 ==
  r := bracket [d.center::0, d.polyPart::0]
  g := prefix(dvd, [d.principalPart::0])
--   z := one?(d.principalPart)
  z := (d.principalPart = 1)
  princ? d => (z => zer; g)
  z => r
  r + g

reduce d ==
  d.reduced? => d
  degree(a := d.center) <= gen => (d.reduced? := true; d)
  b := d.polyPart
  a0 := ((hyper - b**2) exquo a)::UP
  b0 := (-b) rem a0
  g := d.principalPart * reduce(b::RF::UPUP-monomial(1,1)$UPUP)/a0::RF::R

```

```

      reduce makeDivisor(a0, b0, g)

generator d ==
  d := reduce d
  princ? d => d.principalPart
  "failed"

- d ==
  a := d.center
  makeDivisor(a, - d.polyPart, inv(a::RF * d.principalPart))

d1 = d2 ==
  d1 := reduce d1
  d2 := reduce d2
  d1.center = d2.center and d1.polyPart = d2.polyPart
  and d1.principalPart = d2.principalPart

divisor(a, b) ==
  x := monomial(1, 1)$UP
  not ground? gcd(d := x - a::UP, retract(discriminant())@UP) =>
    error "divisor: point is singular"
  makeDivisor(d, b::UP, 1)

intReduce(h, b) ==
  v := integralCoordinates(h).num
  integralRepresents(
    [qelt(v, i) rem b for i in minIndex v .. maxIndex v], 1)

-- with hyperelliptic curves, it is cheaper to keep divisors in reduced form
divisor(h, a, dp, g, r) ==
  h := h - (r * dp)::RF::R
  a := gcd(a, retract(norm h)@UP)
  h := intReduce(h, a)
  if not ground? gcd(g, a) then h := intReduce(h ** rank(), a)
  hh := lift h
  b := - coefficient(hh, 0) / coefficient(hh, 1)
  rec := extendedEuclidean(denom b, a)
  not ground?(rec.generator) => ERR
  bb := ((numer(b) * rec.coef1) exquo rec.generator)::UP rem a
  reduce makeDivisor(a, bb, 1)

```

— HELLFDIV.dotabb —

```

"HELLFDIV" [color="#88FF44",href="bookvol10.3.pdf#nameddest=HELLFDIV"]
"FDIVCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FDIVCAT"]
"HELLFDIV" -> "FDIVCAT"

```

—

Chapter 10

Chapter I

domain ICP InfClsPt

— InfClsPt.input —

```
)set break resume
)sys rm -f InfClsPt.output
)spool InfClsPt.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InfClsPt
--R InfClsPt(K: Field,symb: List(Symbol),BLMET: BlowUpMethodCategory) is a domain constructor
--R Abbreviation for InfClsPt is ICP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ICP
--R
--R----- Operations -----
--R ==? : (%,% ) -> Boolean          actualExtensionV : % -> K
--R chartV : % -> BLMET              coerce : % -> OutputForm
--R degree : % -> PositiveInteger    fullOut : % -> OutputForm
--R fullOutput : () -> Boolean        fullOutput : Boolean -> Boolean
--R hash : % -> SingleInteger         latex : % -> String
--R localPointV : % -> AffinePlane(K) multV : % -> NonNegativeInteger
--R pointV : % -> ProjectivePlane(K)  setchart! : (% ,BLMET) -> BLMET
--R symbNameV : % -> Symbol           ?~=? : (%,% ) -> Boolean
--R create : (ProjectivePlane(K),DistributedMultivariatePolynomial(symb,K)) -> %
--R create : (ProjectivePlane(K),DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],K),AffinePlane
--R curveV : % -> DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],K)
--R excpDivV : % -> Divisor(Places(K))
```



```

--R localParamV : % -> List(NeitherSparseOrDensePowerSeries(K))
--R setcurve! : (% , DistributedMultivariatePolynomial([construct, QUOTEX, QUOTEY], K)) -> DistributedMultivariatePolynomial([construct, QUOTEX, QUOTEY], K)
--R setexcpDiv! : (% , Divisor(Places(K))) -> Divisor(Places(K))
--R setlocalParam! : (% , List(NeitherSparseOrDensePowerSeries(K))) -> List(NeitherSparseOrDensePowerSeries(K))
--R setlocalPoint! : (% , AffinePlane(K)) -> AffinePlane(K)
--R setmult! : (% , NonNegativeInteger) -> NonNegativeInteger
--R setpoint! : (% , ProjectivePlane(K)) -> ProjectivePlane(K)
--R setsubmult! : (% , NonNegativeInteger) -> NonNegativeInteger
--R setsymbName! : (% , Symbol) -> Symbol
--R subMultV : % -> NonNegativeInteger
--R
--E 1

)spool
)lisp (bye)

```

— InfClsPt.help —

```

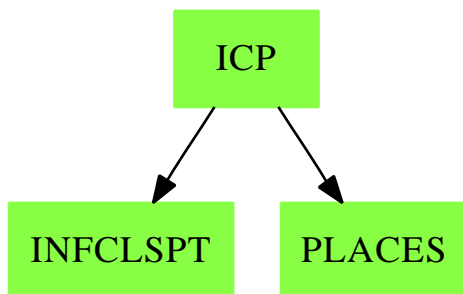
=====
InfClsPt examples
=====

```

This domain is part of the PAFF package

See Also:
o)show InfClsPt

InfClsPt (ICP)



?=?	?~=?	actualExtensionV
chartV	coerce	create
curveV	degree	excpDivV
fullOut	fullOutput	hash
latex	localParamV	localPointV
multV	pointV	setchart!
setcurve!	setexcpDiv!	setlocalParam!
setlocalPoint!	setmult!	setpoint!
setsubmult!	setsymbName!	subMultV
symbNameV		

[illegible]

— ICP.dotabb —

```
"ICP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ICP"]
"INFCLSPT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=INFCLSPT"]
"PLACES" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PLACES"]
"ICP" -> "INFCLSPT"
"ICP" -> "PLACES"
```

domain ICARD IndexCard

— IndexCard.input —

```
)set break resume
)sys rm -f IndexCard.output
)spool IndexCard.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IndexCard
--R IndexCard is a domain constructor
--R Abbreviation for IndexCard is ICARD
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ICARD
--R
--R----- Operations -----
--R ?<? : (%,% ) -> Boolean          ?<=? : (%,% ) -> Boolean
--R ?=? : (%,% ) -> Boolean          ?>? : (%,% ) -> Boolean
--R ?>=? : (%,% ) -> Boolean          coerce : String -> %
--R coerce : % -> OutputForm          display : % -> Void
--R ?? : (% ,Symbol) -> String          fullDisplay : % -> Void
--R hash : % -> SingleInteger          latex : % -> String
--R max : (%,% ) -> %                  min : (%,% ) -> %
--R ?^=? : (%,% ) -> Boolean
--R
--E 1

)spool
)lisp (bye)
```

— IndexCard.help —

=====

IndexCard examples

=====

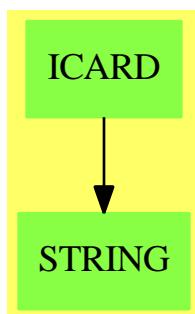
This domain implements a container of information about the AXIOM library

See Also:

o)show IndexCard

—————

IndexCard (ICARD)



See

⇒ “DataList” (DLIST) [5](#) on page [459](#)

⇒ “Database” (DBASE) [5](#) on page [454](#)

⇒ “QueryEquation” (QEQUAT) [18](#) on page [2169](#)

Exports:

coerce	display	fullDisplay	hash	latex
max	min	?~=?	?<?	?<=?
?=?	?>?	?>=?	?..?	

— domain ICARD IndexCard —

)abbrev domain ICARD IndexCard

++ Author: Mark Botch

++ Description:

++ This domain implements a container of information about the AXIOM library

IndexCard() : Exports == Implementation where

Exports == OrderedSet with

```

elt: (% ,Symbol) -> String
  ++ elt(ic,s) selects a particular field from \axiom{ic}. Valid fields
  ++ are \axiom{name, nargs, exposed, type, abbreviation, kind, origin,
  ++ params, condition, doc}.
display: % -> Void
  ++ display(ic) prints a summary of information contained in \axiom{ic}.
fullDisplay: % -> Void
  ++ fullDisplay(ic) prints all of the information contained in \axiom{ic}.
coerce: String -> %
  ++ coerce(s) converts \axiom{s} into an \axiom{IndexCard}. Warning: if
  ++ \axiom{s} is not of the right format then an error will occur
Implementation == add
x<y==(x pretend String) < (y pretend String)
x=y==(x pretend String) = (y pretend String)
display(x) ==
  name : OutputForm := dbName(x)$Lisp
  type : OutputForm := dbPart(x,4,1$Lisp)$Lisp
  output(hconcat(name,hconcat(" : ",type)))$OutputPackage
fullDisplay(x) ==
  name : OutputForm := dbName(x)$Lisp
  type : OutputForm := dbPart(x,4,1$Lisp)$Lisp
  origin:OutputForm :=
    hconcat(alqlGetOrigin(x$Lisp)$Lisp,alqlGetParams(x$Lisp)$Lisp)
  fromPart : OutputForm := hconcat(" from ",origin)
  condition : String := dbPart(x,6,1$Lisp)$Lisp
  ifPart : OutputForm :=
    condition = "" => empty()
    hconcat(" if ",condition::OutputForm)
  exposed? : String := SUBSTRING(dbPart(x,3,1)$Lisp,0,1)$Lisp
  exposedPart : OutputForm :=
    exposed? = "n" => " (unexposed)"
    empty()
  firstPart := hconcat(name,hconcat(" : ",type))
  secondPart := hconcat(fromPart,hconcat(ifPart,exposedPart))
  output(hconcat(firstPart,secondPart))$OutputPackage
coerce(s:String): % == (s pretend %)
coerce(x): OutputForm == (x pretend String)::OutputForm
elt(x,sel) ==
  s := PNAME(sel)$Lisp pretend String
  s = "name" => dbName(x)$Lisp
  s = "nargs" => dbPart(x,2,1$Lisp)$Lisp
  s = "exposed" => SUBSTRING(dbPart(x,3,1)$Lisp,0,1)$Lisp
  s = "type" => dbPart(x,4,1$Lisp)$Lisp
  s = "abbreviation" => dbPart(x,5,1$Lisp)$Lisp
  s = "kind" => alqlGetKindString(x)$Lisp
  s = "origin" => alqlGetOrigin(x)$Lisp
  s = "params" => alqlGetParams(x)$Lisp
  s = "condition" => dbPart(x,6,1$Lisp)$Lisp
  s = "doc" => dbComments(x)$Lisp
  error "unknown selector"

```

domain IBITS IndexedBits

[illegible]

[illegible]

Type: IndexedBits(32)

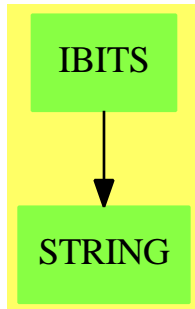
Type: IndexedBits(32)

Type: IndexedBits(32)

Type: IndexedBits(32)

```
=====
IndexedBits
=====
```

```
elt(a,3)
false
```

IndexedBits (IBITS)**See**⇒ “Reference” (REF) [19](#) on page [2249](#)⇒ “Boolean” (BOOLEAN) [3](#) on page [322](#)⇒ “Bits” (BITS) [3](#) on page [314](#)**Exports:**

And	any?	coerce	concat	construct
convert	copy	copyInto!	count	count
delete	elt	empty	empty?	entries
entry?	eq?	eval	every?	fill!
find	first	hash	index?	indices
insert	latex	less?	map	map!
max	maxIndex	member?	members	merge
min	minIndex	more?	nand	new
nor	Not	not?	Or	parts
position	qelt	qsetelt!	reduce	removeDuplicates
reverse	reverse!	sample	select	size?
sort	sort!	sorted?	swap!	xor
#?	?.?	?/\?	?<?	?<=?
?=?	?>?	?>=?	?\\/?	^?
?.?	~?	?~=?	?or?	?and?

— domain IBITS IndexedBits —

```

)abbrev domain IBITS IndexedBits
++ Author: Stephen Watt and Michael Monagan
++ Date Created: July 86
++ Change History: Oct 87
++ Description:
++ \spadtype{IndexedBits} is a domain to compactly represent
++ large quantities of Boolean data.

```

```

IndexedBits(mn:Integer): BitAggregate() with
-- temporaries until parser gets better
Not: % -> %

```

```

    ++ Not(n) returns the bit-by-bit logical Not of n.
Or : (% , %) -> %
    ++ Or(n,m) returns the bit-by-bit logical Or of
    ++ n and m.
And: (% , %) -> %
    ++ And(n,m) returns the bit-by-bit logical And of
    ++ n and m.
== add

range: (% , Integer) -> Integer
    --++ range(j,i) returns the range i of the boolean j.

minIndex u == mn

range(v, i) ==
    i >= 0 and i < #v => i
    error "Index out of range"

coerce(v):OutputForm ==
    t:Character := char "1"
    f:Character := char "0"
    s := new(#v, space()$Character)$String
    for i in minIndex(s)..maxIndex(s) for j in mn.. repeat
        s.i := if v.j then t else f
    s::OutputForm

new(n, b)      == BVEC_-MAKE_-FULL(n,TRUTH_-TO_-BIT(b)$Lisp)$Lisp
empty()        == BVEC_-MAKE_-FULL(0,0)$Lisp
copy v         == BVEC_-COPY(v)$Lisp
#v             == BVEC_-SIZE(v)$Lisp
v = u          == BVEC_-EQUAL(v, u)$Lisp
v < u          == BVEC_-GREATER(u, v)$Lisp
_and(u, v)     == (#v=#u => BVEC_-AND(v,u)$Lisp; map("and",v,u))
_or(u, v)      == (#v=#u => BVEC_-OR(v, u)$Lisp; map("or", v,u))
xor(v,u)       == (#v=#u => BVEC_-XOR(v,u)$Lisp; map("xor",v,u))
setelt(v:%, i:Integer, f:Boolean) ==
    BVEC_-SETELT(v, range(v, abs(i-mn)), TRUTH_-TO_-BIT(f)$Lisp)$Lisp
elt(v:%, i:Integer) ==
    BIT_-TO_-TRUTH(BVEC_-ELT(v, range(v, abs(i-mn)))$Lisp)$Lisp

Not v          == BVEC_-NOT(v)$Lisp
And(u, v)      == (#v=#u => BVEC_-AND(v,u)$Lisp; map("and",v,u))
Or(u, v)       == (#v=#u => BVEC_-OR(v, u)$Lisp; map("or", v,u))

```

```
"IBITS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IBITS"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"IBITS" -> "STRING"
```

domain IDPAG IndexedDirectProductAbelianGroup

— IndexedDirectProductAbelianGroup.input —

```
)set break resume
)sys rm -f IndexedDirectProductAbelianGroup.output
)spool IndexedDirectProductAbelianGroup.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IndexedDirectProductAbelianGroup
--R IndexedDirectProductAbelianGroup(A: AbelianGroup,S: OrderedSet) is a domain constructor
--R Abbreviation for IndexedDirectProductAbelianGroup is IDPAG
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for IDPAG
--R
--R----- Operations -----
--R ?? : (Integer,%) -> %          ?? : (PositiveInteger,%) -> %
--R ?+? : (%,%) -> %              ?-? : (%,%) -> %
--R -? : % -> %                   ?=? : (%,%) -> Boolean
--R 0 : () -> %                   coerce : % -> OutputForm
--R hash : % -> SingleInteger     latex : % -> String
--R leadingCoefficient : % -> A    leadingSupport : % -> S
--R map : ((A -> A),%) -> %        monomial : (A,S) -> %
--R reductum : % -> %              sample : () -> %
--R zero? : % -> Boolean           ?~=? : (%,%) -> Boolean
--R ?? : (NonNegativeInteger,%) -> %
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)
```

— IndexedDirectProductAbelianGroup.help —

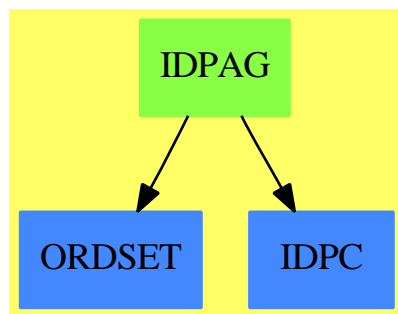
```
=====
IndexedDirectProductAbelianGroup examples
=====
```

Indexed direct products of abelian groups over an abelian group A of generators indexed by the ordered set S . All items have finite support: only non-zero terms are stored.

See Also:

```
o )show IndexedDirectProductAbelianGroup
```

IndexedDirectProductAbelianGroup (IDPAG)



See

- ⇒ “IndexedDirectProductObject” (IDPO) [10](#) on page [1209](#)
- ⇒ “IndexedDirectProductAbelianMonoid” (IDPAM) [10](#) on page [1205](#)
- ⇒ “IndexedDirectProductOrderedAbelianMonoid” (IDPOAM) [10](#) on page [1212](#)
- ⇒ “IndexedDirectProductOrderedAbelianMonoidSup” (IDPOAMS) [10](#) on page [1214](#)

Exports:

0	coerce	hash	latex	leadingCoefficient
leadingSupport	map	monomial	reductum	sample
subtractIfCan	zero?	?~=?	?*?	?+?
?-?	-?	?=?		

— domain IDPAG IndexedDirectProductAbelianGroup —

```
)abbrev domain IDPAG IndexedDirectProductAbelianGroup
++ Author: Mark Botch
++ Description:
++ Indexed direct products of abelian groups over an abelian group \spad{A} of
++ generators indexed by the ordered set S.
++ All items have finite support: only non-zero terms are stored.
```

```

IndexedDirectProductAbelianGroup(A:AbelianGroup,S:OrderedSet):
  Join(AbelianGroup,IndexedDirectProductCategory(A,S))
== IndexedDirectProductAbelianMonoid(A,S) add
--representations
  Term:= Record(k:S,c:A)
  Rep:= List Term
  x,y: %
  r: A
  n: Integer
  f: A -> A
  s: S
  -x == [[u.k,-u.c] for u in x]
  n * x ==
    n = 0 => 0
    n = 1 => x
    [[u.k,a] for u in x | (a:=n*u.c) ^= 0$A]

qsetrest!: (Rep, Rep) -> Rep
qsetrest!(l: Rep, e: Rep): Rep == RPLACD(l, e)$Lisp

x - y ==
  null x => -y
  null y => x
  endcell: Rep := empty()
  res: Rep := empty()
  while not empty? x and not empty? y repeat
    newcell := empty()
    if x.first.k = y.first.k then
      r:= x.first.c - y.first.c
      if not zero? r then
        newcell := cons([x.first.k, r], empty())
      x := rest x
      y := rest y
    else if x.first.k > y.first.k then
      newcell := cons(x.first, empty())
      x := rest x
    else
      newcell := cons([y.first.k,-y.first.c], empty())
      y := rest y
  if not empty? newcell then
    if not empty? endcell then
      qsetrest!(endcell, newcell)
      endcell := newcell
    else
      res := newcell;
      endcell := res
  if empty? x then end := - y
  else end := x
  if empty? res then res := end

```

```

        else qsetrest!(endcell, end)
        res

--      x - y ==
--      empty? x => - y
--      empty? y => x
--      y.first.k > x.first.k => cons([y.first.k,-y.first.c],(x - y.rest))
--      x.first.k > y.first.k => cons(x.first,(x.rest - y))
--      r:= x.first.c - y.first.c
--      r = 0 => x.rest - y.rest
--      cons([x.first.k,r],(x.rest - y.rest))

```

— IDPAG.dotabb —

```

"IDPAG" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IDPAG"]
"ORDSET" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ORDSET"]
"IDPC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=IDPC"]
"IDPAG" -> "IDPC"
"IDPAG" -> "ORDSET"

```

domain IDPAM IndexedDirectProductAbelianMonoid

— IndexedDirectProductAbelianMonoid.input —

```

)set break resume
)sys rm -f IndexedDirectProductAbelianMonoid.output
)spool IndexedDirectProductAbelianMonoid.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IndexedDirectProductAbelianMonoid
--R IndexedDirectProductAbelianMonoid(A: AbelianMonoid,S: OrderedSet) is a domain constructor
--R Abbreviation for IndexedDirectProductAbelianMonoid is IDPAM
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for IDPAM
--R
--R----- Operations -----
--R ?*? : (PositiveInteger,%) -> %      ?+? : (%,%) -> %

```

```

--R ?=? : (%,% ) -> Boolean
--R coerce : % -> OutputForm
--R latex : % -> String
--R leadingSupport : % -> S
--R monomial : (A,S) -> %
--R sample : () -> %
--R ?^=? : (%,% ) -> Boolean
--R ?*? : (NonNegativeInteger,% ) -> %
--R
--E 1

)spool
)lisp (bye)

```

— IndexedDirectProductAbelianMonoid.help —

=====

IndexedDirectProductAbelianMonoid examples

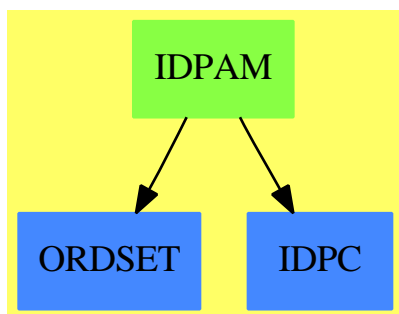
=====

Indexed direct products of abelian monoids over an abelian monoid
A of generators indexed by the ordered set S. All items have
finite support. Only non-zero terms are stored.

See Also:

o)show IndexedDirectProductAbelianMonoid

IndexedDirectProductAbelianMonoid (IDPAM)



See

⇒ “IndexedDirectProductObject” (IDPO) [10](#) on page [1209](#)

⇒ “IndexedDirectProductOrderedAbelianMonoid” (IDPOAM) 10 on page 1212
 ⇒ “IndexedDirectProductOrderedAbelianMonoidSup” (IDPOAMS) 10 on page 1214
 ⇒ “IndexedDirectProductAbelianGroup” (IDPAG) 10 on page 1202

Exports:

0	coerce	hash	latex	leadingCoefficient
leadingSupport	map	monomial	reductum	sample
zero?	?~=?	?*?	?+?	?=?

— domain IDPAM IndexedDirectProductAbelianMonoid —

```

)abbrev domain IDPAM IndexedDirectProductAbelianMonoid
++ Author: Mark Botch
++ Description:
++ Indexed direct products of abelian monoids over an abelian monoid
++ \spad{A} of generators indexed by the ordered set S. All items have
++ finite support. Only non-zero terms are stored.

IndexedDirectProductAbelianMonoid(A:AbelianMonoid,S:OrderedSet):
  Join(AbelianMonoid,IndexedDirectProductCategory(A,S))
== IndexedDirectProductObject(A,S) add
--representations
Term:= Record(k:S,c:A)
Rep:= List Term
x,y: %
r: A
n: NonNegativeInteger
f: A -> A
s: S
0 == []
zero? x == null x

-- PERFORMANCE CRITICAL; Should build list up
-- by merging 2 sorted lists. Doing this will
-- avoid the recursive calls (very useful if there is a
-- large number of vars in a polynomial.
--
x + y ==
--
null x => y
null y => x
y.first.k > x.first.k => cons(y.first,(x + y.rest))
x.first.k > y.first.k => cons(x.first,(x.rest + y))
r:= x.first.c + y.first.c
r = 0 => x.rest + y.rest
--
cons([x.first.k,r],(x.rest + y.rest))
qsetrest!: (Rep, Rep) -> Rep
qsetrest!(l: Rep, e: Rep): Rep == RPLACD(1, e)$Lisp

x + y ==
null x => y
null y => x

```

```

endcell: Rep := empty()
res: Rep := empty()
while not empty? x and not empty? y repeat
  newcell := empty()
  if x.first.k = y.first.k then
    r := x.first.c + y.first.c
    if not zero? r then
      newcell := cons([x.first.k, r], empty())
    x := rest x
    y := rest y
  else if x.first.k > y.first.k then
    newcell := cons(x.first, empty())
    x := rest x
  else
    newcell := cons(y.first, empty())
    y := rest y
  if not empty? newcell then
    if not empty? endcell then
      qsetrest!(endcell, newcell)
      endcell := newcell
    else
      res := newcell;
      endcell := res
  if empty? x then end := y
  else end := x
  if empty? res then res := end
  else qsetrest!(endcell, end)
res

n * x ==
  n = 0 => 0
  n = 1 => x
  [[u.k,a] for u in x | (a:=n*u.c) ^= 0$A]

monomial(r,s) == (r = 0 => 0; [[s,r]])
map(f,x) == [[tm.k,a] for tm in x | (a:=f(tm.c)) ^= 0$A]

reductum x == (null x => 0; rest x)
leadingCoefficient x == (null x => 0; x.first.c)

```

— IDPAM.dotabb —

"IDPAM" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IDPAM"]
 "ORDSET" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ORDSET"]
 "IDPC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=IDPC"]
 "IDPAM" -> "IDPC"

"IDPAM" -> "ORDSET"

domain IDPO IndexedDirectProductObject

— IndexedDirectProductObject.input —

```
)set break resume
)sys rm -f IndexedDirectProductObject.output
)spool IndexedDirectProductObject.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IndexedDirectProductObject
--R IndexedDirectProductObject(A: SetCategory,S: OrderedSet) is a domain constructor
--R Abbreviation for IndexedDirectProductObject is IDPO
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for IDPO
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger        latex : % -> String
--R leadingCoefficient : % -> A      leadingSupport : % -> S
--R map : ((A -> A),%) -> %          monomial : (A,S) -> %
--R reductum : % -> %               ?~=? : (%,% ) -> Boolean
--R
--E 1

)spool
)lisp (bye)
```

— IndexedDirectProductObject.help —

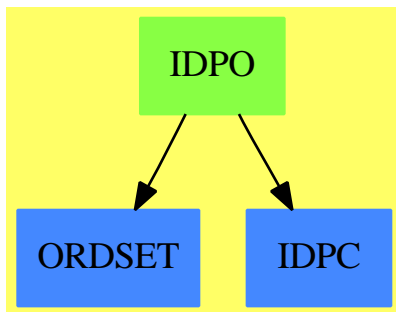
```
=====
IndexedDirectProductObject examples
=====
```

Indexed direct products of objects over a set A of generators indexed by an ordered set S. All items have finite support.

See Also:

o)show IndexedDirectProductObject

IndexedDirectProductObject (IDPO)



See

- ⇒ “IndexedDirectProductAbelianMonoid” (IDPAM) [10](#) on page [1205](#)
- ⇒ “IndexedDirectProductOrderedAbelianMonoid” (IDPOAM) [10](#) on page [1212](#)
- ⇒ “IndexedDirectProductOrderedAbelianMonoidSup” (IDPOAMS) [10](#) on page [1214](#)
- ⇒ “IndexedDirectProductAbelianGroup” (IDPAG) [10](#) on page [1202](#)

Exports:

coerce	hash	latex	leadingCoefficient	leadingSupport
map	monomial	reductum	?=?	?~=?

— domain IDPO IndexedDirectProductObject —

```

)abbrev domain IDPO IndexedDirectProductObject
++ Author: Mark Botch
++ Description:
++ Indexed direct products of objects over a set \spad{A}
++ of generators indexed by an ordered set S. All items have finite support.

```

```

IndexedDirectProductObject(A:SetCategory,S:OrderedSet): _
  IndexedDirectProductCategory(A,S)
== add
  --representations
  Term:= Record(k:S,c:A)
  Rep:= List Term
  --declarations
  x,y: %
  f: A -> A
  s: S

```

```

--define
  x = y ==
    while not null x and _^ null y repeat
      x.first.k ^= y.first.k => return false
      x.first.c ^= y.first.c => return false
      x:=x.rest
      y:=y.rest
    null x and null y

  coerce(x:%):OutputForm ==
    bracket [rarrow(t.k :: OutputForm, t.c :: OutputForm) for t in x]

  -- sample():% == [[sample()$S,sample()$A]$Term]$Rep

  monomial(r,s) == [[s,r]]
  map(f,x) == [[tm.k,f(tm.c)] for tm in x]

  reductum x ==
    rest x
  leadingCoefficient x ==
    null x => error "Can't take leadingCoefficient of empty product element"
    x.first.c
  leadingSupport x ==
    null x => error "Can't take leadingCoefficient of empty product element"
    x.first.k

```

— IDPO.dotabb —

```

"IDPO" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IDPO"]
"ORDSET" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ORDSET"]
"IDPC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=IDPC"]
"IDPO" -> "IDPC"
"IDPO" -> "ORDSET"

```

domain IDPOAM IndexedDirectProductOrderedAbelian-Monoid

— IndexedDirectProductOrderedAbelianMonoid.input —

```

)set break resume

```

```

)sys rm -f IndexedDirectProductOrderedAbelianMonoid.output
)spool IndexedDirectProductOrderedAbelianMonoid.output
)set message test on
)set message auto off
)clear all

```

```
--S 1 of 1
```

```
)show IndexedDirectProductOrderedAbelianMonoid
```

```
--R IndexedDirectProductOrderedAbelianMonoid(A: OrderedAbelianMonoid,S: OrderedSet) is a domain constructor
```

```
--R Abbreviation for IndexedDirectProductOrderedAbelianMonoid is IDPOAM
```

```
--R This constructor is not exposed in this frame.
```

```
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for IDPOAM
```

```
--R
```

```
--R----- Operations -----
```

```

--R ?? : (PositiveInteger,%) -> %      ?+? : (%,%) -> %
--R ?<? : (%,%) -> Boolean             ?<=? : (%,%) -> Boolean
--R ==? : (%,%) -> Boolean             ?>? : (%,%) -> Boolean
--R ?>=? : (%,%) -> Boolean            0 : () -> %
--R coerce : % -> OutputForm           hash : % -> SingleInteger
--R latex : % -> String                 leadingCoefficient : % -> A
--R leadingSupport : % -> S              map : ((A -> A),%) -> %
--R max : (%,%) -> %                   min : (%,%) -> %
--R monomial : (A,S) -> %               reductum : % -> %
--R sample : () -> %                   zero? : % -> Boolean
--R ?^=? : (%,%) -> Boolean
--R ?*? : (NonNegativeInteger,%) -> %

```

```
--R
```

```
--E 1
```

```
)spool
```

```
)lisp (bye)
```

— IndexedDirectProductOrderedAbelianMonoid.help —

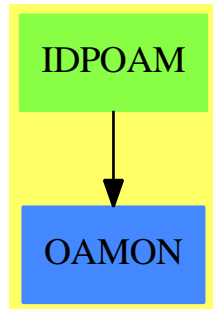
```
=====
IndexedDirectProductOrderedAbelianMonoid examples
=====
```

Indexed direct products of ordered abelian monoids A of generators indexed by the ordered set S. The inherited order is lexicographical. All items have finite support: only non-zero terms are stored.

See Also:

o)show IndexedDirectProductOrderedAbelianMonoid

IndexedDirectProductOrderedAbelianMonoid (IDPOAM)



See

- ⇒ “IndexedDirectProductObject” (IDPO) [10](#) on page [1209](#)
- ⇒ “IndexedDirectProductAbelianMonoid” (IDPAM) [10](#) on page [1205](#)
- ⇒ “IndexedDirectProductOrderedAbelianMonoidSup” (IDPOAMS) [10](#) on page [1214](#)
- ⇒ “IndexedDirectProductAbelianGroup” (IDPAG) [10](#) on page [1202](#)

Exports:

0	coerce	hash	latex	leadingCoefficient
leadingSupport	map	max	min	monomial
reductum	sample	zero?	?~=?	?*?
?+?	?<?	?<=?	?=?	?>?
?>=?				

— domain IDPOAM IndexedDirectProductOrderedAbelianMonoid

```

)abbrev domain IDPOAM IndexedDirectProductOrderedAbelianMonoid
++ Author: Mark Botch
++ Description:
++ Indexed direct products of ordered abelian monoids \spad{A} of
++ generators indexed by the ordered set S.
++ The inherited order is lexicographical.
++ All items have finite support: only non-zero terms are stored.

IndexedDirectProductOrderedAbelianMonoid(A:OrderedAbelianMonoid,S:OrderedSet):
  Join(OrderedAbelianMonoid,IndexedDirectProductCategory(A,S))
== IndexedDirectProductAbelianMonoid(A,S) add
--representations
Term:= Record(k:S,c:A)
Rep:= List Term
x,y: %
x<y ==
  empty? y => false
  empty? x => true -- note careful order of these two lines
  y.first.k > x.first.k => true
  
```

```

y.first.k < x.first.k => false
y.first.c > x.first.c => true
y.first.c < x.first.c => false
x.rest < y.rest

```

— IDPOAM.dotabb —

```

"IDPOAM" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IDPOAM"]
"OAMON" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAMON"]
"IDPOAM" -> "OAMON"

```

domain IDPOAMS IndexedDirectProductOrderedAbelian-MonoidSup

— IndexedDirectProductOrderedAbelianMonoidSup.input —

```

)set break resume
)sys rm -f IndexedDirectProductOrderedAbelianMonoidSup.output
)spool IndexedDirectProductOrderedAbelianMonoidSup.output
)set message test on
)set message auto off
)clear all

```

--S 1 of 1

```
)show IndexedDirectProductOrderedAbelianMonoidSup
```

--R IndexedDirectProductOrderedAbelianMonoidSup(A: OrderedAbelianMonoidSup,S: OrderedSet) is a domain constructor

--R Abbreviation for IndexedDirectProductOrderedAbelianMonoidSup is IDPOAMS

--R This constructor is not exposed in this frame.

--R Issue)edit bookvol10.3.pamphlet to see algebra source code for IDPOAMS

--R

--R----- Operations -----

--R ?? : (PositiveInteger,%) -> % ?+? : (%,%) -> %

--R ?<? : (%,%) -> Boolean ?<=? : (%,%) -> Boolean

--R ?=? : (%,%) -> Boolean ?>? : (%,%) -> Boolean

--R ?>=? : (%,%) -> Boolean 0 : () -> %

--R coerce : % -> OutputForm hash : % -> SingleInteger

--R latex : % -> String leadingCoefficient : % -> A

--R leadingSupport : % -> S map : ((A -> A),%) -> %

--R max : (%,%) -> % min : (%,%) -> %


```

--R monomial : (A,S) -> %          reductum : % -> %
--R sample : () -> %              sup : (%,% ) -> %
--R zero? : % -> Boolean          ?~=? : (%,% ) -> Boolean
--R ??? : (NonNegativeInteger,% ) -> %
--R subtractIfCan : (%,% ) -> Union(%,"failed")
--R
--E 1

)spool
)lisp (bye)

```

— IndexedDirectProductOrderedAbelianMonoidSup.help —

=====

IndexedDirectProductOrderedAbelianMonoidSup examples

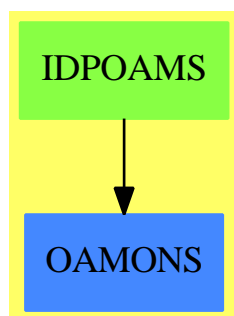
=====

Indexed direct products of ordered abelian monoid sups A, generators indexed by the ordered set S. All items have finite support: only non-zero terms are stored.

See Also:

o)show IndexedDirectProductOrderedAbelianMonoidSup

IndexedDirectProductOrderedAbelianMonoidSup (IDPOAMS)



See

- ⇒ “IndexedDirectProductObject” (IDPO) [10](#) on page [1209](#)
- ⇒ “IndexedDirectProductAbelianMonoid” (IDPAM) [10](#) on page [1205](#)
- ⇒ “IndexedDirectProductOrderedAbelianMonoid” (IDPOAM) [10](#) on page [1212](#)
- ⇒ “IndexedDirectProductAbelianGroup” (IDPAG) [10](#) on page [1202](#)

Exports:

0	coerce	hash	latex	leadingCoefficient
leadingSupport	map	max	min	monomial
reductum	sample	subtractIfCan	sup	zero?
?~=?	?*?	?+?	?<?	?<=?
?=?	?>?	?>=?		

— domain IDPOAMS IndexedDirectProductOrderedAbelianMonoid-Sup —

```
)abbrev domain IDPOAMS IndexedDirectProductOrderedAbelianMonoidSup
++ Author: Mark Botch
++ Description:
++ Indexed direct products of ordered abelian monoid sups \spad{A},
++ generators indexed by the ordered set S.
++ All items have finite support: only non-zero terms are stored.
```

```
IndexedDirectProductOrderedAbelianMonoidSup(A:OrderedAbelianMonoidSup,S:OrderedSet):
  Join(OrderedAbelianMonoidSup,IndexedDirectProductCategory(A,S))
== IndexedDirectProductOrderedAbelianMonoid(A,S) add
--representations
Term:= Record(k:S,c:A)
Rep:= List Term
x,y: %
r: A
s: S

subtractIfCan(x,y) ==
  empty? y => x
  empty? x => "failed"
  x.first.k < y.first.k => "failed"
  x.first.k > y.first.k =>
    t:= subtractIfCan(x.rest, y)
    t case "failed" => "failed"
    cons( x.first, t)
  u:=subtractIfCan(x.first.c, y.first.c)
  u case "failed" => "failed"
  zero? u => subtractIfCan(x.rest, y.rest)
  t:= subtractIfCan(x.rest, y.rest)
  t case "failed" => "failed"
  cons([x.first.k,u],t)

sup(x,y) ==
  empty? y => x
  empty? x => y
  x.first.k < y.first.k => cons(y.first,sup(x,y.rest))
  x.first.k > y.first.k => cons(x.first,sup(x.rest,y))
  u:=sup(x.first.c, y.first.c)
  cons([x.first.k,u],sup(x.rest,y.rest))
```

— IDPOAMS.dotabb —

```
"IDPOAMS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IDPOAMS"]
"OAMONS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAMONS"]
"IDPOAMS" -> "OAMONS"
```

domain INDE IndexedExponents

— IndexedExponents.input —

```
)set break resume
)sys rm -f IndexedExponents.output
)spool IndexedExponents.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IndexedExponents
--R
--R IndexedExponents(Varset: OrderedSet) is a domain constructor
--R Abbreviation for IndexedExponents is INDE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for INDE
--R
--R----- Operations -----
--R ?? : (PositiveInteger,%) -> %      ?+? : (%,%) -> %
--R ?<? : (%,%) -> Boolean             ?<=? : (%,%) -> Boolean
--R ?=? : (%,%) -> Boolean             ?>? : (%,%) -> Boolean
--R ?>=? : (%,%) -> Boolean            0 : () -> %
--R coerce : % -> OutputForm           hash : % -> SingleInteger
--R latex : % -> String                 leadingSupport : % -> Varset
--R max : (%,%) -> %                   min : (%,%) -> %
--R reductum : % -> %                  sample : () -> %
--R sup : (%,%) -> %                   zero? : % -> Boolean
--R ?~=? : (%,%) -> Boolean
--R ?*? : (NonNegativeInteger,%) -> %
--R leadingCoefficient : % -> NonNegativeInteger
--R map : ((NonNegativeInteger -> NonNegativeInteger),%) -> %
```

```

--R monomial : (NonNegativeInteger,Varset) -> %
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)

```

— IndexedExponents.help —

=====

IndexedExponents examples

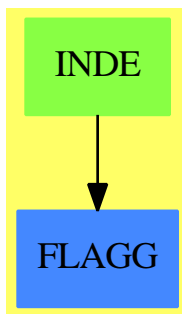
=====

IndexedExponents of an ordered set of variables gives a representation for the degree of polynomials in commuting variables. It gives an ordered pairing of non negative integer exponents with variables

See Also:

o)show IndexedExponents

IndexedExponents (INDE)



See

- ⇒ “Polynomial” (POLY) [17](#) on page [2077](#)
- ⇒ “MultivariatePolynomial” (MPOLY) [14](#) on page [1681](#)
- ⇒ “SparseMultivariatePolynomial” (SMP) [20](#) on page [2422](#)

Exports:

0	coerce	hash	latex	leadingCoefficient
leadingSupport	map	max	min	monomial
reductum	sample	subtractIfCan	sup	zero?
?~=?	?*?	?+?	?<?	?<=?
?=?	?>?	?>=?		

— domain INDE IndexedExponents —

```

)abbrev domain INDE IndexedExponents
++ Author: James Davenport
++ Description:
++ IndexedExponents of an ordered set of variables gives a representation
++ for the degree of polynomials in commuting variables. It gives an ordered
++ pairing of non negative integer exponents with variables

IndexedExponents(Varset:OrderedSet): C == T where
  C == Join(OrderedAbelianMonoidSup,
            IndexedDirectProductCategory(NonNegativeInteger,Varset))
  T == IndexedDirectProductOrderedAbelianMonoidSup(NonNegativeInteger,Varset) add
    Term:= Record(k:Varset,c:NonNegativeInteger)
    Rep:= List Term
    x:%
    t:Term
    coerceOF(t):OutputForm ==      --++ converts term to OutputForm
      t.c = 1 => (t.k)::OutputForm
      (t.k)::OutputForm ** (t.c)::OutputForm
    coerce(x):OutputForm == ++ converts entire exponents to OutputForm
      null x => 1::Integer::OutputForm
      null rest x => coerceOF(first x)
      reduce("",[coerceOF t for t in x])

```

— INDE.dotabb —

```

"INDE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=INDE"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"INDE" -> "FLAGG"

```

domain IFARRAY IndexedFlexibleArray

— IndexedFlexibleArray.input —

```

)set break resume
)sys rm -f IndexedFlexibleArray.output
)spool IndexedFlexibleArray.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IndexedFlexibleArray
--R
--R IndexedFlexibleArray(S: Type,mn: Integer) is a domain constructor
--R Abbreviation for IndexedFlexibleArray is IFARRAY
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for IFARRAY
--R
--R----- Operations -----
--R concat : List(%) -> %          concat : (%,%) -> %
--R concat : (S,%) -> %          concat : (%,S) -> %
--R concat! : (%,S) -> %         concat! : (%,%) -> %
--R construct : List(S) -> %      copy : % -> %
--R delete : (%,Integer) -> %     delete! : (%,Integer) -> %
--R ?? : (%,Integer) -> S        elt : (%,Integer,S) -> S
--R empty : () -> %             empty? : % -> Boolean
--R entries : % -> List(S)       eq? : (%,%) -> Boolean
--R flexibleArray : List(S) -> % index? : (Integer,%) -> Boolean
--R indices : % -> List(Integer) insert : (%,%,Integer) -> %
--R insert : (S,%,Integer) -> %  insert! : (S,%,Integer) -> %
--R insert! : (%,%,Integer) -> % map : (((S,S) -> S),%,%) -> %
--R map : ((S -> S),%) -> %      new : (NonNegativeInteger,S) -> %
--R qelt : (%,Integer) -> S     reverse : % -> %
--R sample : () -> %           shrinkable : Boolean -> Boolean
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?<? : (%,%) -> Boolean if S has ORDSET
--R ?<=? : (%,%) -> Boolean if S has ORDSET
--R ?=? : (%,%) -> Boolean if S has SETCAT
--R ?>? : (%,%) -> Boolean if S has ORDSET
--R ?>=? : (%,%) -> Boolean if S has ORDSET
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if S has SETCAT
--R convert : % -> InputForm if S has KONVERT(INFORM)
--R copyInto! : (%,%,Integer) -> % if $ has shallowlyMutable
--R count : (S,%) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R delete : (%,UniversalSegment(Integer)) -> %
--R delete! : (%,UniversalSegment(Integer)) -> %
--R ?? : (%,UniversalSegment(Integer)) -> %
--R entry? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R eval : (%,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT

```

```

--R eval : (%,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (%,S) -> % if $ has shallowlyMutable
--R find : ((S -> Boolean),%) -> Union(S,"failed")
--R first : % -> S if Integer has ORDSET
--R hash : % -> SingleInteger if S has SETCAT
--R latex : % -> String if S has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R max : (%,%) -> % if S has ORDSET
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R members : % -> List(S) if $ has finiteAggregate
--R merge : (%,%) -> % if S has ORDSET
--R merge : ((S,S) -> Boolean),%,%) -> %
--R merge! : ((S,S) -> Boolean),%,%) -> %
--R merge! : (%,%) -> % if S has ORDSET
--R min : (%,%) -> % if S has ORDSET
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (%,NonNegativeInteger) -> Boolean
--R parts : % -> List(S) if $ has finiteAggregate
--R physicalLength : % -> NonNegativeInteger
--R physicalLength! : (%,Integer) -> %
--R position : (S,%,Integer) -> Integer if S has SETCAT
--R position : (S,%) -> Integer if S has SETCAT
--R position : ((S -> Boolean),%) -> Integer
--R qsetelt! : (%,Integer,S) -> S if $ has shallowlyMutable
--R reduce : ((S,S) -> S),%) -> S if $ has finiteAggregate
--R reduce : ((S,S) -> S),%,S) -> S if $ has finiteAggregate
--R reduce : ((S,S) -> S),%,S,S) -> S if $ has finiteAggregate and S has SETCAT
--R remove : ((S -> Boolean),%) -> % if $ has finiteAggregate
--R remove : (S,%) -> % if $ has finiteAggregate and S has SETCAT
--R remove! : ((S -> Boolean),%) -> %
--R remove! : (S,%) -> % if S has SETCAT
--R removeDuplicates : % -> % if $ has finiteAggregate and S has SETCAT
--R removeDuplicates! : % -> % if S has SETCAT
--R reverse! : % -> % if $ has shallowlyMutable
--R select : ((S -> Boolean),%) -> % if $ has finiteAggregate
--R select! : ((S -> Boolean),%) -> %
--R setelt : (%,UniversalSegment(Integer),S) -> S if $ has shallowlyMutable
--R setelt : (%,Integer,S) -> S if $ has shallowlyMutable
--R size? : (%,NonNegativeInteger) -> Boolean
--R sort : % -> % if S has ORDSET
--R sort : ((S,S) -> Boolean),%) -> %
--R sort! : % -> % if $ has shallowlyMutable and S has ORDSET
--R sort! : ((S,S) -> Boolean),%) -> % if $ has shallowlyMutable
--R sorted? : % -> Boolean if S has ORDSET
--R sorted? : ((S,S) -> Boolean),%) -> Boolean
--R swap! : (%,Integer,Integer) -> Void if $ has shallowlyMutable
--R ?~=? : (%,%) -> Boolean if S has SETCAT

```

```
--R
--E 1
```

```
)spool
)lisp (bye)
```

— IndexedFlexibleArray.help —

```
=====
IndexedFlexibleArray examples
=====
```

A FlexibleArray is the notion of an array intended to allow for growth at the end only. Hence the following efficient operations

```
append(x,a) meaning append item x at the end of the array a
delete(a,n)} meaning delete the last item from the array a
```

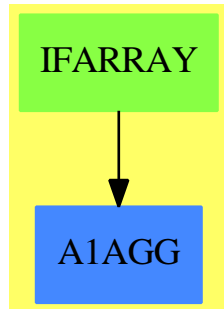
Flexible arrays support the other operations inherited from ExtensibleLinearAggregate. However, these are not efficient.

Flexible arrays combine the $O(1)$ access time property of arrays with growing and shrinking at the end in $O(1)$ (average) time. This is done by using an ordinary array which may have zero or more empty slots at the end. When the array becomes full it is copied into a new larger (50% larger) array. Conversely, when the array becomes less than 1/2 full, it is copied into a smaller array. Flexible arrays provide for an efficient implementation of many data structures in particular heaps, stacks and sets.

See Also:

```
o )show IndexedFlexibleArray
```

IndexedFlexibleArray (IFARRAY)



See

- ⇒ “PrimitiveArray” (PRIMARR) [17](#) on page [2109](#)
- ⇒ “Tuple” (TUPLE) [21](#) on page [2752](#)
- ⇒ “FlexibleArray” (FARRAY) [7](#) on page [885](#)
- ⇒ “IndexedOneDimensionalArray” (IARRAY1) [10](#) on page [1243](#)
- ⇒ “OneDimensionalArray” (ARRAY1) [16](#) on page [1774](#)

Exports:

concat	concat!	construct	copy
delete	delete!	elt	empty
empty?	entries	eq?	flexibleArray
index?	indices	insert	insert!
map	new	qelt	reverse
sample	shrinkable	any?	coerce
convert	copyInto!	count	delete
delete!	entry?	eval	every?
fill!	find	first	hash
latex	less?	map!	max
maxIndex	member?	members	merge
merge!	min	minIndex	more?
parts	physicalLength	physicalLength!	position
qsetelt!	reduce	remove	remove!
removeDuplicates	removeDuplicates!	reverse!	select
select!	setelt	size?	sort
sort!	sorted?	swap!	#?
?<?	?<=?	?=?	?>?
?>=?	?~=?	?..?	

— domain IFARRAY IndexedFlexibleArray —

```

)abbrev domain IFARRAY IndexedFlexibleArray
++ Author: Michael Monagan July/87, modified SMW June/91
++ Description:
++ A FlexibleArray is the notion of an array intended to allow for growth

```

```

++ at the end only. Hence the following efficient operations\br
++ \spad{append(x,a)} meaning append item x at the end of the array \spad{a}\br
++ \spad{delete(a,n)} meaning delete the last item from the array \spad{a}\br
++ Flexible arrays support the other operations inherited from
++ \spadtype{ExtensibleLinearAggregate}. However, these are not efficient.
++ Flexible arrays combine the \spad{0(1)} access time property of arrays
++ with growing and shrinking at the end in \spad{0(1)} (average) time.
++ This is done by using an ordinary array which may have zero or more
++ empty slots at the end. When the array becomes full it is copied
++ into a new larger (50% larger) array. Conversely, when the array
++ becomes less than 1/2 full, it is copied into a smaller array.
++ Flexible arrays provide for an efficient implementation of many
++ data structures in particular heaps, stacks and sets.

```

```

IndexedFlexibleArray(S:Type, mn: Integer): Exports == Implementation where
  A ==> PrimitiveArray S
  I ==> Integer
  N ==> NonNegativeInteger
  U ==> UniversalSegment Integer
Exports ==
Join(OneDimensionalArrayAggregate S,ExtensibleLinearAggregate S) with
flexibleArray : List S -> %
  ++ flexibleArray(l) creates a flexible array from the list of elements l
  ++
  ++X T1:=IndexedFlexibleArray(Integer,20)
  ++X flexibleArray([i for i in 1..10])$T1

physicalLength : % -> NonNegativeInteger
  ++ physicalLength(x) returns the number of elements x can
  ++ accomodate before growing
  ++
  ++X T1:=IndexedFlexibleArray(Integer,20)
  ++X t2:=flexibleArray([i for i in 1..10])$T1
  ++X physicalLength t2

physicalLength_!: (% , I) -> %
  ++ physicalLength!(x,n) changes the physical length of x to be n and
  ++ returns the new array.
  ++
  ++X T1:=IndexedFlexibleArray(Integer,20)
  ++X t2:=flexibleArray([i for i in 1..10])$T1
  ++X physicalLength!(t2,15)

shrinkable: Boolean -> Boolean
  ++ shrinkable(b) sets the shrinkable attribute of flexible arrays to b
  ++ and returns the previous value
  ++
  ++X T1:=IndexedFlexibleArray(Integer,20)
  ++X shrinkable(false)$T1

```

```

Implementation == add
  Rep := Record(physLen:I, logLen:I, f:A)
  shrinkable? : Boolean := true
  growAndFill : (%, I, S) -> %
  growWith    : (%, I, S) -> %
  growAdding  : (%, I, %) -> %
  shrink      : (%, I)   -> %
  newa       : (N, A) -> A

  physicalLength(r) == (r.physLen) pretend NonNegativeInteger
  physicalLength_!(r, n) ==
    r.physLen = 0 => error "flexible array must be non-empty"
    growWith(r, n, r.f.0)

  empty()      == [0, 0, empty()]
  #r           == (r.logLen)::N
  fill_!(r, x) == (fill_!(r.f, x); r)
  maxIndex r   == r.logLen - 1 + mn
  minIndex r   == mn
  new(n, a)    == [n, n, new(n, a)]

  shrinkable(b) ==
    oldval := shrinkable?
    shrinkable? := b
    oldval

  flexibleArray l ==
    n := #l
    n = 0 => empty()
    x := l.1
    a := new(n,x)
    for i in mn + 1..mn + n-1 for y in rest l repeat a.i := y
    a

  -- local utility operations
  newa(n, a) ==
    zero? n => empty()
    new(n, a.0)

  growAdding(r, b, s) ==
    b = 0 => r
    #r > 0 => growAndFill(r, b, (r.f).0)
    #s > 0 => growAndFill(r, b, (s.f).0)
    error "no default filler element"

  growAndFill(r, b, x) ==
    (r.logLen := r.logLen + b) <= r.physLen => r
    -- enlarge by 50% + b
    n := r.physLen + r.physLen quo 2 + 1
    if r.logLen > n then n := r.logLen

```

```

    growWith(r, n, x)

growWith(r, n, x) ==
  y := new(n::N, x)$PrimitiveArray(S)
  a := r.f
  for k in 0 .. r.physLen-1 repeat y.k := a.k
  r.physLen := n
  r.f := y
  r

shrink(r, i) ==
  r.logLen := r.logLen - i
  negative?(n := r.logLen) => error "internal bug in flexible array"
  2*n+2 > r.physLen => r
  not shrinkable? => r
  if n < r.logLen
    then error "cannot shrink flexible array to indicated size"
  n = 0 => empty()
  r.physLen := n
  y := newa(n::N, a := r.f)
  for k in 0 .. n-1 repeat y.k := a.k
  r.f := y
  r

copy r ==
  n := #r
  a := r.f
  v := newa(n, a := r.f)
  for k in 0..n-1 repeat v.k := a.k
  [n, n, v]

elt(r:%, i:I) ==
  i < mn or i >= r.logLen + mn =>
    error "index out of range"
  r.f.(i-mn)

setelt(r:%, i:I, x:S) ==
  i < mn or i >= r.logLen + mn =>
    error "index out of range"
  r.f.(i-mn) := x

-- operations inherited from extensible aggregate
merge(g, a, b) == merge_!(g, copy a, b)
concat(x:S, r:%) == insert_!(x, r, mn)

concat_!(r:%, x:S) ==
  growAndFill(r, 1, x)
  r.f.(r.logLen-1) := x
  r

```

```

concat_!(a:%, b:%) ==
  if eq?(a, b) then b := copy b
  n := #a
  growAdding(a, #b, b)
  copyInto_!(a, b, n + mn)

remove_!(g:(S->Boolean), a:%) ==
  k:I := 0
  for i in 0..maxIndex a - mn repeat
    if not g(a.i) then (a.k := a.i; k := k+1)
  shrink(a, #a - k)

delete_!(r:%, i1:I) ==
  i := i1 - mn
  i < 0 or i > r.logLen => error "index out of range"
  for k in i..r.logLen-2 repeat r.f.k := r.f.(k+1)
  shrink(r, 1)

delete_!(r:%, i:U) ==
  l := lo i - mn; m := maxIndex r - mn
  h := (hasHi i => hi i - mn; m)
  l < 0 or h > m => error "index out of range"
  for j in l.. for k in h+1..m repeat r.f.j := r.f.k
  shrink(r, max(0, h-l+1))

insert_!(x:S, r:%, i1:I):% ==
  i := i1 - mn
  n := r.logLen
  i < 0 or i > n => error "index out of range"
  growAndFill(r, 1, x)
  for k in n-1 .. i by -1 repeat r.f.(k+1) := r.f.k
  r.f.i := x
  r

insert_!(a:%, b:%, i1:I):% ==
  i := i1 - mn
  if eq?(a, b) then b := copy b
  m := #a; n := #b
  i < 0 or i > n => error "index out of range"
  growAdding(b, m, a)
  for k in n-1 .. i by -1 repeat b.f.(m+k) := b.f.k
  for k in m-1 .. 0 by -1 repeat b.f.(i+k) := a.f.k
  b

merge_!(g, a, b) ==
  m := #a; n := #b; growAdding(a, n, b)
  for i in m-1..0 by -1 for j in m+n-1.. by -1 repeat a.f.j := a.f.i
  i := n; j := 0
  for k in 0.. while i < n+m and j < n repeat

```

```

        if g(a.f.i,b.f.j) then (a.f.k := a.f.i; i := i+1)
        else (a.f.k := b.f.j; j := j+1)
    for k in k.. for j in j..n-1 repeat a.f.k := b.f.j
a

select_!(g:(S->Boolean), a:%) ==
k:I := 0
for i in 0..maxIndex a - mn repeat_
    if g(a.f.i) then (a.f.k := a.f.i;k := k+1)
shrink(a, #a - k)

if S has SetCategory then
removeDuplicates_! a ==
    ct := #a
    ct < 2 => a

    i := mn
    nlim := mn + ct
    nlim0 := nlim
    while i < nlim repeat
        j := i+1
        for k in j..nlim-1 | a.k ^= a.i repeat
            a.j := a.k
            j := j+1
        nlim := j
        i := i+1
    nlim ^= nlim0 => delete_!(a, i..)
a

```

— IFARRAY.dotabb —

```

"IFARRAY" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IFARRAY"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"IFARRAY" -> "A1AGG"

```

domain ILIST IndexedList

— IndexedList.input —

```

)set break resume

```

```

)sys rm -f IndexedList.output
)spool IndexedList.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IndexedList
--R
--R IndexedList(S: Type,mn: Integer) is a domain constructor
--R Abbreviation for IndexedList is ILIST
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ILIST
--R
--R----- Operations -----
--R children : % -> List(%)
--R concat : List(%) -> %
--R concat : (%,%) -> %
--R concat! : (%,%) -> %
--R copy : % -> %
--R cycleTail : % -> %
--R delete : (%,Integer) -> %
--R distance : (%,%) -> Integer
--R ?.? : (%,Integer) -> S
--R ?.rest : (%,rest) -> %
--R ?.value : (%,value) -> S
--R empty? : % -> Boolean
--R eq? : (%,%) -> Boolean
--R first : % -> S
--R indices : % -> List(Integer)
--R insert : (%,%,Integer) -> %
--R insert! : (%,%,Integer) -> %
--R leaf? : % -> Boolean
--R list : S -> %
--R map : ((S -> S),%) -> %
--R nodes : % -> List(%)
--R qelt : (%,Integer) -> S
--R reverse : % -> %
--R second : % -> S
--R third : % -> S
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?<? : (%,%) -> Boolean if S has ORDSET
--R ?<=? : (%,%) -> Boolean if S has ORDSET
--R ?=? : (%,%) -> Boolean if S has SETCAT
--R ?>? : (%,%) -> Boolean if S has ORDSET
--R ?>=? : (%,%) -> Boolean if S has ORDSET
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R child? : (%,%) -> Boolean if S has SETCAT
--R coerce : % -> OutputForm if S has SETCAT
--R convert : % -> InputForm if S has KONVERT(INFORM)
--R concat : (%,S) -> %
--R concat : (S,%) -> %
--R concat! : (%,S) -> %
--R construct : List(S) -> %
--R cycleEntry : % -> %
--R cyclic? : % -> Boolean
--R delete! : (%,Integer) -> %
--R elt : (%,Integer,S) -> S
--R ?.last : (%,last) -> S
--R ?.first : (%,first) -> S
--R empty : () -> %
--R entries : % -> List(S)
--R explicitlyFinite? : % -> Boolean
--R index? : (Integer,%) -> Boolean
--R insert : (S,%,Integer) -> %
--R insert! : (S,%,Integer) -> %
--R last : % -> S
--R leaves : % -> List(S)
--R map : (((S,S) -> S),%,%) -> %
--R new : (NonNegativeInteger,S) -> %
--R possiblyInfinite? : % -> Boolean
--R rest : % -> %
--R sample : () -> %
--R tail : % -> %
--R value : % -> S

```

```

--R copyInto! : (%,%,Integer) -> % if $ has shallowlyMutable
--R count : (S,%) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R cycleLength : % -> NonNegativeInteger
--R cycleSplit! : % -> % if $ has shallowlyMutable
--R delete : (% , UniversalSegment(Integer)) -> %
--R delete! : (% , UniversalSegment(Integer)) -> %
--R ?? : (% , UniversalSegment(Integer)) -> %
--R entry? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R eval : (% , List(S), List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (% , S, S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (% , Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (% , List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (% , S) -> % if $ has shallowlyMutable
--R find : ((S -> Boolean),%) -> Union(S, "failed")
--R first : (% , NonNegativeInteger) -> %
--R hash : % -> SingleInteger if S has SETCAT
--R last : (% , NonNegativeInteger) -> %
--R latex : % -> String if S has SETCAT
--R less? : (% , NonNegativeInteger) -> Boolean
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R max : (% , %) -> % if S has ORDSET
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R members : % -> List(S) if $ has finiteAggregate
--R merge : (((S, S) -> Boolean), %, %) -> %
--R merge : (% , %) -> % if S has ORDSET
--R merge! : (((S, S) -> Boolean), %, %) -> %
--R merge! : (% , %) -> % if S has ORDSET
--R min : (% , %) -> % if S has ORDSET
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (% , NonNegativeInteger) -> Boolean
--R node? : (% , %) -> Boolean if S has SETCAT
--R parts : % -> List(S) if $ has finiteAggregate
--R position : ((S -> Boolean),%) -> Integer
--R position : (S,%) -> Integer if S has SETCAT
--R position : (S, % , Integer) -> Integer if S has SETCAT
--R qsetelt! : (% , Integer, S) -> S if $ has shallowlyMutable
--R reduce : (((S, S) -> S), %, S, S) -> S if $ has finiteAggregate and S has SETCAT
--R reduce : (((S, S) -> S), %, S) -> S if $ has finiteAggregate
--R reduce : (((S, S) -> S), %) -> S if $ has finiteAggregate
--R remove : (S,%) -> % if $ has finiteAggregate and S has SETCAT
--R remove : ((S -> Boolean),%) -> % if $ has finiteAggregate
--R remove! : ((S -> Boolean),%) -> %
--R remove! : (S,%) -> % if S has SETCAT
--R removeDuplicates : % -> % if $ has finiteAggregate and S has SETCAT
--R removeDuplicates! : % -> % if S has SETCAT
--R rest : (% , NonNegativeInteger) -> %
--R reverse! : % -> % if $ has shallowlyMutable

```



```

--R select : ((S -> Boolean),%) -> % if $ has finiteAggregate
--R select! : ((S -> Boolean),%) -> %
--R setchildren! : (%,List(%)) -> % if $ has shallowlyMutable
--R setelt : (%,Integer,S) -> S if $ has shallowlyMutable
--R setelt : (%,UniversalSegment(Integer),S) -> S if $ has shallowlyMutable
--R setelt : (%,last,S) -> S if $ has shallowlyMutable
--R setelt : (%,rest,%) -> % if $ has shallowlyMutable
--R setelt : (%,first,S) -> S if $ has shallowlyMutable
--R setelt : (%,value,S) -> S if $ has shallowlyMutable
--R setfirst! : (%,S) -> S if $ has shallowlyMutable
--R setlast! : (%,S) -> S if $ has shallowlyMutable
--R setrest! : (%,%) -> % if $ has shallowlyMutable
--R setvalue! : (%,S) -> S if $ has shallowlyMutable
--R size? : (%,NonNegativeInteger) -> Boolean
--R sort : (((S,S) -> Boolean),%) -> %
--R sort : % -> % if S has ORDSET
--R sort! : (((S,S) -> Boolean),%) -> % if $ has shallowlyMutable
--R sort! : % -> % if $ has shallowlyMutable and S has ORDSET
--R sorted? : (((S,S) -> Boolean),%) -> Boolean
--R sorted? : % -> Boolean if S has ORDSET
--R split! : (%,Integer) -> % if $ has shallowlyMutable
--R swap! : (%,Integer,Integer) -> Void if $ has shallowlyMutable
--R ~=? : (%,%) -> Boolean if S has SETCAT
--R
--E 1

)spool
)lisp (bye)

```

— IndexedList.help —

```

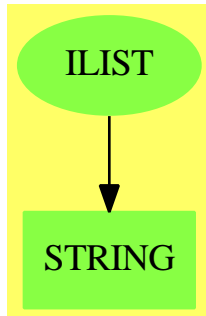
=====
IndexedList examples
=====

```

IndexedList is a basic implementation of the functions in ListAggregate, often using functions in the underlying LISP system. The second parameter to the constructor (mn) is the beginning index of the list. That is, if l is a list, then elt(l,mn) is the first value. This constructor is probably best viewed as the implementation of singly-linked lists that are addressable by index rather than as a mere wrapper for LISP lists.

See Also:

```
o )show IndexedList
```

IndexedList (ILIST)**See**⇒ “List” (LIST) [13](#) on page [1504](#)⇒ “AssociationList” (ALIST) [2](#) on page [235](#)**Exports:**

any?	child?	children	coerce
concat	convert	concat!	copyInto!
construct	copy	count	cycleEntry
cycleLength	cycleSplit!	cycleTail	cyclic?
delete	delete!	distance	elt
empty	empty?	entries	entry?
eq?	eval	every?	explicitlyFinite?
fill!	find	first	hash
index?	indices	insert	insert!
last	latex	leaf?	leaves
less?	list	map	map!
max	maxIndex	member?	members
merge	merge!	min	minIndex
more?	new	node?	nodes
parts	position	possiblyInfinite?	qelt
qsetelt!	reduce	remove	remove!
removeDuplicates	removeDuplicates!	rest	reverse
reverse!	sample	second	select
select!	setchildren!	setelt	setfirst!
setlast!	setrest!	setvalue!	size?
sort	sort!	sorted?	split!
swap!	tail	third	value
#?	?<?	?<=?	?=?
?>?	?>=?	?~=?	?..?
?.last	?.rest	?.first	?.value

— domain ILIST IndexedList —

)abbrev domain ILIST IndexedList

```

++ Author: Michael Monagan
++ Date Created: Sep 1987
++ Description:
++ \spadtype{IndexedList} is a basic implementation of the functions
++ in \spadtype{ListAggregate}, often using functions in the underlying
++ LISP system. The second parameter to the constructor (\spad{mn})
++ is the beginning index of the list. That is, if \spad{l} is a
++ list, then \spad{elt(l,mn)} is the first value. This constructor
++ is probably best viewed as the implementation of singly-linked
++ lists that are addressable by index rather than as a mere wrapper
++ for LISP lists.

```

```

IndexedList(S:Type, mn:Integer): Exports == Implementation where
  cycleMax ==> 1000          -- value used in checking for cycles

```

```

-- The following seems to be a bit out of date, but is kept in case
-- a knowledgeable person wants to update it:
--   The following LISP dependencies are divided into two groups
--   Those that are required
--   CONS, EQ, NIL, NULL, QCAR, QCDR, RPLACA, RPLACD
--   Those that are included for efficiency only
--   NEQ, LIST, CAR, CDR, NCONC2, NREVERSE, LENGTH
--   Also REVERSE, since it's called in Polynomial Ring

```

```

Qfirst ==> QCAR$Lisp
Qrest  ==> QCDR$Lisp
Qnull  ==> NULL$Lisp
Qeq     ==> EQ$Lisp
Qneq    ==> NEQ$Lisp
Qcons   ==> CONS$Lisp
Qpush   ==> PUSH$Lisp

```

```

Exports ==> ListAggregate S

```

```

Implementation ==>

```

```

add
  #x                == LENGTH(x)$Lisp
  concat(s:S,x:%)   == CONS(s,x)$Lisp
  eq?(x,y)           == EQ(x,y)$Lisp
  first x            == SPADfirst(x)$Lisp
  elt(x,"first")     == SPADfirst(x)$Lisp
  empty()            == NIL$Lisp
  empty? x           == NULL(x)$Lisp
  rest x             == CDR(x)$Lisp
  elt(x,"rest")      == CDR(x)$Lisp
  setfirst_!(x,s)    ==
    empty? x => error "Cannot update an empty list"
    Qfirst RPLACA(x,s)$Lisp
  setelt(x,"first",s) ==
    empty? x => error "Cannot update an empty list"
    Qfirst RPLACA(x,s)$Lisp

```

```

setrest_!(x,y)      ==
  empty? x => error "Cannot update an empty list"
  Qrest RPLACD(x,y)$Lisp
setelt(x,"rest",y)  ==
  empty? x => error "Cannot update an empty list"
  Qrest RPLACD(x,y)$Lisp
construct l          == l pretend %
parts s              == s pretend List S
reverse_! x          == NREVERSE(x)$Lisp
reverse x            == REVERSE(x)$Lisp
minIndex x           == mn

rest(x, n) ==
  for i in 1..n repeat
    if Qnull x then error "index out of range"
    x := Qrest x
  x

copy x ==
  y := empty()
  for i in 0.. while not Qnull x repeat
    if Qeq(i,cycleMax) and cyclic? x then error "cyclic list"
    y := Qcons(Qfirst x,y)
    x := Qrest x
  (NREVERSE(y)$Lisp)%

if S has SetCategory then
  coerce(x):OutputForm ==
    -- displays cycle with overbar over the cycle
    y := empty()$List(OutputForm)
    s := cycleEntry x
    while Qneq(x, s) repeat
      y := concat((first x)::OutputForm, y)
      x := rest x
    y := reverse_! y
    empty? s => bracket y
    -- cyclic case: z is cyclic part
    z := list((first x)::OutputForm)
    while Qneq(s, rest x) repeat
      x := rest x
      z := concat((first x)::OutputForm, z)
    bracket concat_!(y, overbar commaSeparate reverse_! z)

x = y ==
  Qeq(x,y) => true
  while not Qnull x and not Qnull y repeat
    Qfirst x ^=$S Qfirst y => return false
    x := Qrest x
    y := Qrest y
  Qnull x and Qnull y

```

```

latex(x : %): String ==
  s : String := "\left["
  while not Qnull x repeat
    s := concat(s, latex(Qfirst x)$S)$String
    x := Qrest x
    if not Qnull x then s := concat(s, ", ")$String
  concat(s, " \right]")$String

member?(s,x) ==
  while not Qnull x repeat
    if s = Qfirst x then return true else x := Qrest x
  false

-- Lots of code from parts of AGGCAT, repeated here to
-- get faster compilation
concat_!(x:%,y:%) ==
  Qnull x =>
    Qnull y => x
    Qpush(first y,x)
    QRPLACD(x,rest y)$Lisp
  x
  z:=x
  while not Qnull Qrest z repeat
    z:=Qrest z
    QRPLACD(z,y)$Lisp
  x

-- Then a quicky:
if S has SetCategory then
  removeDuplicates_! l ==
    p := l
    while not Qnull p repeat
      p := setrest_!(p, remove_!(#1 = Qfirst p, Qrest p))
-- far too expensive - builds closures etc.
  pp:=p
  f:S:=Qfirst p
  p:=Qrest p
  while not Qnull (pr:=Qrest pp) repeat
    if (Qfirst pr)@S = f then QRPLACD(pp,Qrest pr)$Lisp
    else pp:=pr
  l

-- then sorting
mergeSort: ((S, S) -> Boolean, %, Integer) -> %

sort_!(f, l)      == mergeSort(f, l, #1)

merge_!(f, p, q) ==
  Qnull p => q

```

```

Qnull q => p
Qeq(p, q) => error "cannot merge a list into itself"
if f(Qfirst p, Qfirst q)
  then (r := t := p; p := Qrest p)
  else (r := t := q; q := Qrest q)
while not Qnull p and not Qnull q repeat
  if f(Qfirst p, Qfirst q)
    then (QRPLACD(t, p)$Lisp; t := p; p := Qrest p)
    else (QRPLACD(t, q)$Lisp; t := q; q := Qrest q)
QRPLACD(t, if Qnull p then q else p)$Lisp
r

split_!(p, n) ==
  n < 1 => error "index out of range"
  p := rest(p, (n - 1)::NonNegativeInteger)
  q := Qrest p
  QRPLACD(p, NIL$Lisp)$Lisp
  q

mergeSort(f, p, n) ==
  if n = 2 and f(first rest p, first p) then p := reverse_! p
  n < 3 => p
  l := (n quo 2)::NonNegativeInteger
  q := split_!(p, l)
  p := mergeSort(f, p, l)
  q := mergeSort(f, q, n - l)
  merge_!(f, p, q)

```

— ILIST.dotabb —

```

"ILIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ILIST",
        shape=ellipse]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"ILIST" -> "STRING"

```

domain IMATRIX IndexedMatrix

— IndexedMatrix.input —

```
)set break resume
```

```

)sys rm -f IndexedMatrix.output
)spool IndexedMatrix.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IndexedMatrix
--R IndexedMatrix(R: Ring,mnRow: Integer,mnCol: Integer) is a domain constructor
--R Abbreviation for IndexedMatrix is IMATRIX
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for IMATRIX
--R
--R----- Operations -----
--R ???: (Integer,%) -> %          ??? : (%,R) -> %
--R ??? : (R,%) -> %              ??? : (%,%) -> %
--R ?+? : (%,%) -> %              -? : % -> %
--R ?-? : (%,%) -> %              antisymmetric? : % -> Boolean
--R copy : % -> %                 diagonal? : % -> Boolean
--R diagonalMatrix : List(%) -> % diagonalMatrix : List(R) -> %
--R elt : (%,Integer,Integer,R) -> R elt : (%,Integer,Integer) -> R
--R empty : () -> %               empty? : % -> Boolean
--R eq? : (%,%) -> Boolean         fill! : (%,R) -> %
--R horizConcat : (%,%) -> %      listOfLists : % -> List(List(R))
--R map : (((R,R) -> R),%,%,R) -> % map : (((R,R) -> R),%,%) -> %
--R map : ((R -> R),%) -> %        map! : ((R -> R),%) -> %
--R matrix : List(List(R)) -> %    maxColIndex : % -> Integer
--R maxRowIndex : % -> Integer      minColIndex : % -> Integer
--R minRowIndex : % -> Integer      ncols : % -> NonNegativeInteger
--R nrows : % -> NonNegativeInteger parts : % -> List(R)
--R qelt : (%,Integer,Integer) -> R sample : () -> %
--R square? : % -> Boolean          squareTop : % -> %
--R symmetric? : % -> Boolean        transpose : % -> %
--R vertConcat : (%,%) -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ??? : (IndexedVector(R,mnCol),%) -> IndexedVector(R,mnCol)
--R ??? : (%,IndexedVector(R,mnRow)) -> IndexedVector(R,mnRow)
--R ??? : (%,Integer) -> % if R has FIELD
--R ??? : (%,NonNegativeInteger) -> %
--R ?/? : (%,R) -> % if R has FIELD
--R ==? : (%,%) -> Boolean if R has SETCAT
--R any? : ((R -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : IndexedVector(R,mnRow) -> %
--R coerce : % -> OutputForm if R has SETCAT
--R column : (%,Integer) -> IndexedVector(R,mnRow)
--R columnSpace : % -> List(IndexedVector(R,mnRow)) if R has EUCLDOM
--R count : (R,%) -> NonNegativeInteger if $ has finiteAggregate and R has SETCAT
--R count : ((R -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R determinant : % -> R if R has commutative(*)
--R elt : (%,List(Integer),List(Integer)) -> %

```

```

--R eval : (% ,List(R),List(R)) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (% ,R,R) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (% ,Equation(R)) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (% ,List(Equation(R))) -> % if R has EVALAB(R) and R has SETCAT
--R every? : ((R -> Boolean),%) -> Boolean if $ has finiteAggregate
--R exquo : (% ,R) -> Union(%,"failed") if R has INTDOM
--R hash : % -> SingleInteger if R has SETCAT
--R inverse : % -> Union(%,"failed") if R has FIELD
--R latex : % -> String if R has SETCAT
--R less? : (% ,NonNegativeInteger) -> Boolean
--R matrix : (NonNegativeInteger,NonNegativeInteger,((Integer,Integer) -> R)) -> %
--R member? : (R,%) -> Boolean if $ has finiteAggregate and R has SETCAT
--R members : % -> List(R) if $ has finiteAggregate
--R minordet : % -> R if R has commutative(*)
--R more? : (% ,NonNegativeInteger) -> Boolean
--R new : (NonNegativeInteger,NonNegativeInteger,R) -> %
--R nullSpace : % -> List(IndexedVector(R,mnRow)) if R has INTDOM
--R nullity : % -> NonNegativeInteger if R has INTDOM
--R pfaffian : % -> R if R has COMRING
--R qsetelt! : (% ,Integer,Integer,R) -> R
--R rank : % -> NonNegativeInteger if R has INTDOM
--R row : (% ,Integer) -> IndexedVector(R,mnCol)
--R rowEchelon : % -> % if R has EUCDOM
--R scalarMatrix : (NonNegativeInteger,R) -> %
--R setColumn! : (% ,Integer,IndexedVector(R,mnRow)) -> %
--R setRow! : (% ,Integer,IndexedVector(R,mnCol)) -> %
--R setelt : (% ,List(Integer),List(Integer),%) -> %
--R setelt : (% ,Integer,Integer,R) -> R
--R setsubMatrix! : (% ,Integer,Integer,%) -> %
--R size? : (% ,NonNegativeInteger) -> Boolean
--R subMatrix : (% ,Integer,Integer,Integer,Integer) -> %
--R swapColumns! : (% ,Integer,Integer) -> %
--R swapRows! : (% ,Integer,Integer) -> %
--R transpose : IndexedVector(R,mnCol) -> %
--R zero : (NonNegativeInteger,NonNegativeInteger) -> %
--R ?~=? : (% ,%) -> Boolean if R has SETCAT
--R
--E 1

```

```

)spool
)lisp (bye)

```

— IndexedMatrix.help —

```

=====
IndexedMatrix examples
=====

```

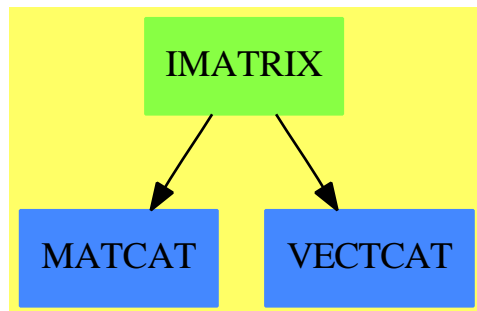

An `IndexedMatrix` is a matrix where the minimal row and column indices are parameters of the type. The domains `Row` and `Col` are both `IndexedVectors`.

The index of the 'first' row may be obtained by calling the function `minRowIndex`. The index of the 'first' column may be obtained by calling the function `minColIndex`. The index of the first element of a 'Row' is the same as the index of the first column in a matrix and vice versa.

See Also:

- o `)show IndexedMatrix`

IndexedMatrix (IMATRIX)



See

- ⇒ “Matrix” (MATRIX) [14](#) on page [1622](#)
- ⇒ “RectangularMatrix” (RMATRIX) [19](#) on page [2245](#)
- ⇒ “SquareMatrix” (SQMATRIX) [20](#) on page [2547](#)

Exports:

any?	antisymmetric?	coerce	column	copy
count	determinant	diagonal?	diagonalMatrix	elt
empty	empty?	eq?	eval	every?
exquo	fill!	hash	horizConcat	inverse
latex	less?	listOfLists	map	map!
matrix	maxColIndex	maxRowIndex	member?	members
minColIndex	minordet	minRowIndex	more?	ncols
new	nrows	nullSpace	nullity	parts
qelt	qsetelt!	rank	row	rowEchelon
sample	scalarMatrix	setColumn!	setRow!	setelt
setsubMatrix!	size?	square?	squareTop	subMatrix
swapColumns!	swapRows!	symmetric?	transpose	vertConcat
zero	#?	?*?	?**?	?/?
?=?	?~=?	?+?	-?	?-?

— domain IMATRIX IndexedMatrix —

```
)abbrev domain IMATRIX IndexedMatrix
++ Author: Grabmeier, Gschnitzer, Williamson
++ Date Created: 1987
++ Date Last Updated: July 1990
++ Description:
++ An \spad{IndexedMatrix} is a matrix where the minimal row and column
++ indices are parameters of the type. The domains Row and Col
++ are both IndexedVectors.
++ The index of the 'first' row may be obtained by calling the
++ function \spadfun{minRowIndex}. The index of the 'first' column may
++ be obtained by calling the function \spadfun{minColIndex}. The index of
++ the first element of a 'Row' is the same as the index of the
++ first column in a matrix and vice versa.
```

```
IndexedMatrix(R,mnRow,mnCol): Exports == Implementation where
```

```
  R : Ring
  mnRow, mnCol : Integer
  Row ==> IndexedVector(R,mnCol)
  Col ==> IndexedVector(R,mnRow)
  MATLIN ==> MatrixLinearAlgebraFunctions(R,Row,Col,$)
```

```
Exports ==> MatrixCategory(R,Row,Col)
```

```
Implementation ==>
```

```
  InnerIndexedTwoDimensionalArray(R,mnRow,mnCol,Row,Col) add
```

```
  swapRows_!(x,i1,i2) ==
    (i1 < minRowIndex(x)) or (i1 > maxRowIndex(x)) or _
    (i2 < minRowIndex(x)) or (i2 > maxRowIndex(x)) =>
      error "swapRows!: index out of range"
  i1 = i2 => x
```

```

minRow := minRowIndex x
xx := x pretend PrimitiveArray(PrimitiveArray(R))
n1 := i1 - minRow; n2 := i2 - minRow
row1 := qelt(xx,n1)
qsetelt_!(xx,n1,qelt(xx,n2))
qsetelt_!(xx,n2,row1)
xx pretend $

if R has commutative("*") then

    determinant x == determinant(x)$MATLIN
    minordet     x == minordet(x)$MATLIN

if R has EuclideanDomain then

    rowEchelon x == rowEchelon(x)$MATLIN

if R has IntegralDomain then

    rank      x == rank(x)$MATLIN
    nullity   x == nullity(x)$MATLIN
    nullSpace x == nullSpace(x)$MATLIN

if R has Field then

    inverse    x == inverse(x)$MATLIN

```

— IMATRIX.dotabb —

```

"IMATRIX" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IMATRIX"]
"MATCAT"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=MATCAT"]
"VECTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=VECTCAT"]
"IMATRIX" -> "MATCAT"
"IMATRIX" -> "VECTCAT"

```

domain IARRAY1 IndexedOneDimensionalArray

— IndexedOneDimensionalArray.input —

```

)set break resume

```

```

)sys rm -f IndexedOneDimensionalArray.output
)spool IndexedOneDimensionalArray.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IndexedOneDimensionalArray
--R
--R IndexedOneDimensionalArray(S: Type,mn: Integer) is a domain constructor
--R Abbreviation for IndexedOneDimensionalArray is IARRAY1
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for IARRAY1
--R
--R----- Operations -----
--R concat : List(%) -> %          concat : (%,%) -> %
--R concat : (S,%) -> %          concat : (%,S) -> %
--R construct : List(S) -> %      copy : % -> %
--R delete : (%,Integer) -> %    ?.? : (%,Integer) -> S
--R elt : (%,Integer,S) -> S     empty : () -> %
--R empty? : % -> Boolean        entries : % -> List(S)
--R eq? : (%,%) -> Boolean       index? : (Integer,%) -> Boolean
--R indices : % -> List(Integer) insert : (%,%,Integer) -> %
--R insert : (S,%,Integer) -> %  map : (((S,S) -> S),%,%) -> %
--R map : ((S -> S),%) -> %      new : (NonNegativeInteger,S) -> %
--R qelt : (%,Integer) -> S     reverse : % -> %
--R sample : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?<? : (%,%) -> Boolean if S has ORDSET
--R ?<=? : (%,%) -> Boolean if S has ORDSET
--R ?=? : (%,%) -> Boolean if S has SETCAT
--R ?>? : (%,%) -> Boolean if S has ORDSET
--R ?>=? : (%,%) -> Boolean if S has ORDSET
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if S has SETCAT
--R convert : % -> InputForm if S has KONVERT(INFORM)
--R copyInto! : (%,%,Integer) -> % if $ has shallowlyMutable
--R count : (S,%) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R delete : (%,UniversalSegment(Integer)) -> %
--R ?.? : (%,UniversalSegment(Integer)) -> %
--R entry? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R eval : (%,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (%,S) -> % if $ has shallowlyMutable
--R find : ((S -> Boolean),%) -> Union(S,"failed")
--R first : % -> S if Integer has ORDSET

```

```

--R hash : % -> SingleInteger if S has SETCAT
--R latex : % -> String if S has SETCAT
--R less? : (% , NonNegativeInteger) -> Boolean
--R map! : ((S -> S), %) -> % if $ has shallowlyMutable
--R max : (% , %) -> % if S has ORDSET
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (S , %) -> Boolean if $ has finiteAggregate and S has SETCAT
--R members : % -> List(S) if $ has finiteAggregate
--R merge : (% , %) -> % if S has ORDSET
--R merge : (((S , S) -> Boolean), % , %) -> %
--R min : (% , %) -> % if S has ORDSET
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (% , NonNegativeInteger) -> Boolean
--R parts : % -> List(S) if $ has finiteAggregate
--R position : (S , % , Integer) -> Integer if S has SETCAT
--R position : (S , %) -> Integer if S has SETCAT
--R position : ((S -> Boolean), %) -> Integer
--R qsetelt! : (% , Integer , S) -> S if $ has shallowlyMutable
--R reduce : (((S , S) -> S), %) -> S if $ has finiteAggregate
--R reduce : (((S , S) -> S), % , S) -> S if $ has finiteAggregate
--R reduce : (((S , S) -> S), % , S , S) -> S if $ has finiteAggregate and S has SETCAT
--R remove : ((S -> Boolean), %) -> % if $ has finiteAggregate
--R remove : (S , %) -> % if $ has finiteAggregate and S has SETCAT
--R removeDuplicates : % -> % if $ has finiteAggregate and S has SETCAT
--R reverse! : % -> % if $ has shallowlyMutable
--R select : ((S -> Boolean), %) -> % if $ has finiteAggregate
--R setelt : (% , UniversalSegment(Integer) , S) -> S if $ has shallowlyMutable
--R setelt : (% , Integer , S) -> S if $ has shallowlyMutable
--R size? : (% , NonNegativeInteger) -> Boolean
--R sort : % -> % if S has ORDSET
--R sort : (((S , S) -> Boolean), %) -> %
--R sort! : % -> % if $ has shallowlyMutable and S has ORDSET
--R sort! : (((S , S) -> Boolean), %) -> % if $ has shallowlyMutable
--R sorted? : % -> Boolean if S has ORDSET
--R sorted? : (((S , S) -> Boolean), %) -> Boolean
--R swap! : (% , Integer , Integer) -> Void if $ has shallowlyMutable
--R ?~=?: (% , %) -> Boolean if S has SETCAT
--R
--E 1

)spool
)lisp (bye)

```

— IndexedOneDimensionalArray.help —

=====

IndexedOneDimensionalArray examples

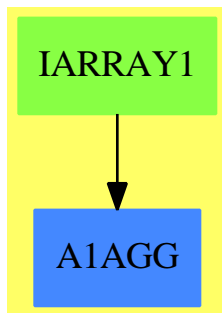
=====

This is the basic one dimensional array data type.

See Also:

o)show IndexedOneDimensionalArray

IndexedOneDimensionalArray (IARRAY1)



See

- ⇒ “PrimitiveArray” (PRIMARR) [17](#) on page [2109](#)
- ⇒ “Tuple” (TUPLE) [21](#) on page [2752](#)
- ⇒ “IndexedFlexibleArray” (IFARRAY) [10](#) on page [1222](#)
- ⇒ “FlexibleArray” (FARRAY) [7](#) on page [885](#)
- ⇒ “OneDimensionalArray” (ARRAY1) [16](#) on page [1774](#)

Exports:

concat	construct	copy	delete	elt
empty	empty?	entries	eq?	index?
indices	insert	insert	map	map
new	qelt	reverse	sample	any?
coerce	convert	copyInto!	count	count
delete	entry?	eval	eval	eval
eval	every?	fill!	find	first
hash	latex	less?	map!	max
maxIndex	member?	members	merge	merge
min	minIndex	more?	parts	position
position	position	qsetelt!	reduce	reduce
reduce	remove	remove	removeDuplicates	reverse!
select	setelt	setelt	size?	sort
sort	sort!	sort!	sorted?	sorted?
swap!	#?	?<?	?<=?	?=?
?>?	?>=?	?~=?	?..?	

— domain IARRAY1 IndexedOneDimensionalArray —

```

)abbrev domain IARRAY1 IndexedOneDimensionalArray
++ Author Micheal Monagan Aug/87
++ Description:
++ This is the basic one dimensional array data type.

IndexedOneDimensionalArray(S:Type, mn:Integer):
  OneDimensionalArrayAggregate S == add
    Qmax ==> QVMAXINDEX$Lisp
    Qsize ==> QVSIZE$Lisp
  -- Qelt ==> QVELT$Lisp
  -- Qsetelt ==> QSETVELT$Lisp
    Qelt ==> ELT$Lisp
    Qsetelt ==> SETELT$Lisp
  -- Qelt1 ==> QVELT_-1$Lisp
  -- Qsetelt1 ==> QSETVELT_-1$Lisp
    Qnew ==> MAKE_-ARRAY$Lisp
    I ==> Integer

  #x == Qsize x
  fill_!(x, s) == (for i in 0..Qmax x repeat Qsetelt(x, i, s); x)
  minIndex x == mn

  empty() == Qnew(0$Lisp)
  new(n, s) == fill_!(Qnew n,s)

  map_!(f, s1) ==
    n:Integer := Qmax(s1)
    n < 0 => s1
    for i in 0..n repeat Qsetelt(s1, i, f(Qelt(s1,i)))

```

```

s1

map(f, s1) ==
  n:Integer := Qmax(s1)
  n < 0 => s1
  ss2:% := Qnew(n+1)
  for i in 0..n repeat Qsetelt(ss2, i, f(Qelt(s1,i)))
  ss2

map(f, a, b) ==
  maxind:Integer := min(Qmax a, Qmax b)
  maxind < 0 => empty()
  c:% := Qnew(maxind+1)
  for i in 0..maxind repeat
    Qsetelt(c, i, f(Qelt(a,i),Qelt(b,i)))
  c

if zero? mn then
  qelt(x, i) == Qelt(x, i)
  qsetelt_!(x, i, s) == Qsetelt(x, i, s)

elt(x:%, i:I) ==
  negative? i or i > maxIndex(x) => error "index out of range"
  qelt(x, i)

setelt(x:%, i:I, s:S) ==
  negative? i or i > maxIndex(x) => error "index out of range"
  qsetelt_!(x, i, s)

-- else if one? mn then
else if (mn = 1) then
  maxIndex x == Qsize x
  qelt(x, i) == Qelt(x, i-1)
  qsetelt_!(x, i, s) == Qsetelt(x, i-1, s)

  elt(x:%, i:I) ==
    QSLESSP(i,1$Lisp)$Lisp or QSLESSP(Qsize x,i)$Lisp =>
      error "index out of range"
    Qelt(x, i-1)

  setelt(x:%, i:I, s:S) ==
    QSLESSP(i,1$Lisp)$Lisp or QSLESSP(Qsize x,i)$Lisp =>
      error "index out of range"
    Qsetelt(x, i-1, s)

else
  qelt(x, i) == Qelt(x, i - mn)
  qsetelt_!(x, i, s) == Qsetelt(x, i - mn, s)

  elt(x:%, i:I) ==

```



```

i < mn or i > maxIndex(x) => error "index out of range"
qelt(x, i)

setelt(x:%, i:I, s:S) ==
i < mn or i > maxIndex(x) => error "index out of range"
qsetelt_!(x, i, s)

```

— IARRAY1.dotabb —

```

"IARRAY1" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IARRAY1"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"IARRAY1" -> "A1AGG"

```

domain ISTRING IndexedString

— IndexedString.input —

```

)set break resume
)sys rm -f IndexedString.output
)spool IndexedString.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IndexedString
--R
--R IndexedString(mn: Integer) is a domain constructor
--R Abbreviation for IndexedString is ISTRING
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ISTRING
--R
--R----- Operations -----
--R coerce : Character -> %                concat : List(%) -> %
--R concat : (%,%) -> %                    concat : (Character,%) -> %
--R concat : (%,Character) -> %            construct : List(Character) -> %
--R copy : % -> %                          delete : (%,Integer) -> %
--R ?.? : (%,%) -> %                      ?.? : (%,Integer) -> Character
--R empty : () -> %                       empty? : % -> Boolean
--R entries : % -> List(Character)         eq? : (%,%) -> Boolean

```

```

--R hash : % -> Integer                                index? : (Integer,%) -> Boolean
--R indices : % -> List(Integer)                        insert : (%,%,Integer) -> %
--R leftTrim : (%,Character) -> %                      lowerCase : % -> %
--R lowerCase! : % -> %                                prefix? : (%,%) -> Boolean
--R qelt : (%,Integer) -> Character                    reverse : % -> %
--R rightTrim : (%,Character) -> %                    sample : () -> %
--R split : (%,Character) -> List(%)                  suffix? : (%,%) -> Boolean
--R trim : (%,CharacterClass) -> %                   trim : (%,Character) -> %
--R upperCase : % -> %                                upperCase! : % -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?<? : (%,%) -> Boolean if Character has ORDSET
--R ?<=? : (%,%) -> Boolean if Character has ORDSET
--R ?=? : (%,%) -> Boolean if Character has SETCAT
--R ?>? : (%,%) -> Boolean if Character has ORDSET
--R ?>=? : (%,%) -> Boolean if Character has ORDSET
--R any? : ((Character -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if Character has SETCAT
--R convert : % -> InputForm if Character has KONVERT(INFORM)
--R copyInto! : (%,%,Integer) -> % if $ has shallowlyMutable
--R count : (Character,%) -> NonNegativeInteger if $ has finiteAggregate and Character has SETCAT
--R count : ((Character -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R delete : (%,UniversalSegment(Integer)) -> %
--R ?.? : (%,UniversalSegment(Integer)) -> %
--R elt : (%,Integer,Character) -> Character
--R entry? : (Character,%) -> Boolean if $ has finiteAggregate and Character has SETCAT
--R eval : (%,List(Character),List(Character)) -> % if Character has EVALAB(CHAR) and Character has SETCAT
--R eval : (%,Character,Character) -> % if Character has EVALAB(CHAR) and Character has SETCAT
--R eval : (%,Equation(Character)) -> % if Character has EVALAB(CHAR) and Character has SETCAT
--R eval : (%,List(Equation(Character))) -> % if Character has EVALAB(CHAR) and Character has SETCAT
--R every? : ((Character -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (%,Character) -> % if $ has shallowlyMutable
--R find : ((Character -> Boolean),%) -> Union(Character,"failed")
--R first : % -> Character if Integer has ORDSET
--R hash : % -> SingleInteger if Character has SETCAT
--R insert : (Character,%,Integer) -> %
--R latex : % -> String if Character has SETCAT
--R leftTrim : (%,CharacterClass) -> %
--R less? : (%,NonNegativeInteger) -> Boolean
--R map : (((Character,Character) -> Character),%,%) -> %
--R map : ((Character -> Character),%) -> %
--R map! : ((Character -> Character),%) -> % if $ has shallowlyMutable
--R match : (%,%,Character) -> NonNegativeInteger
--R match? : (%,%,Character) -> Boolean
--R max : (%,%) -> % if Character has ORDSET
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (Character,%) -> Boolean if $ has finiteAggregate and Character has SETCAT
--R members : % -> List(Character) if $ has finiteAggregate
--R merge : (%,%) -> % if Character has ORDSET
--R merge : (((Character,Character) -> Boolean),%,%) -> %
--R min : (%,%) -> % if Character has ORDSET

```

```

--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (% , NonNegativeInteger) -> Boolean
--R new : (NonNegativeInteger, Character) -> %
--R parts : % -> List(Character) if $ has finiteAggregate
--R position : (CharacterClass, % , Integer) -> Integer
--R position : (% , % , Integer) -> Integer
--R position : (Character, % , Integer) -> Integer if Character has SETCAT
--R position : (Character, %) -> Integer if Character has SETCAT
--R position : ((Character -> Boolean), %) -> Integer
--R qsetelt! : (% , Integer, Character) -> Character if $ has shallowlyMutable
--R reduce : (((Character, Character) -> Character), %) -> Character if $ has finiteAggregate
--R reduce : (((Character, Character) -> Character), % , Character) -> Character if $ has finiteAggregate
--R reduce : (((Character, Character) -> Character), % , Character, Character) -> Character if $ has finiteAggregate
--R remove : ((Character -> Boolean), %) -> % if $ has finiteAggregate
--R remove : (Character, %) -> % if $ has finiteAggregate and Character has SETCAT
--R removeDuplicates : % -> % if $ has finiteAggregate and Character has SETCAT
--R replace : (% , UniversalSegment(Integer), %) -> %
--R reverse! : % -> % if $ has shallowlyMutable
--R rightTrim : (% , CharacterClass) -> %
--R select : ((Character -> Boolean), %) -> % if $ has finiteAggregate
--R setelt : (% , UniversalSegment(Integer), Character) -> Character if $ has shallowlyMutable
--R setelt : (% , Integer, Character) -> Character if $ has shallowlyMutable
--R size? : (% , NonNegativeInteger) -> Boolean
--R sort : % -> % if Character has ORDSET
--R sort : (((Character, Character) -> Boolean), %) -> %
--R sort! : % -> % if $ has shallowlyMutable and Character has ORDSET
--R sort! : (((Character, Character) -> Boolean), %) -> % if $ has shallowlyMutable
--R sorted? : % -> Boolean if Character has ORDSET
--R sorted? : (((Character, Character) -> Boolean), %) -> Boolean
--R split : (% , CharacterClass) -> List(%)
--R substring? : (% , % , Integer) -> Boolean
--R swap! : (% , Integer, Integer) -> Void if $ has shallowlyMutable
--R ~=? : (% , %) -> Boolean if Character has SETCAT
--R
--E 1

)spool
)lisp (bye)

```

— IndexedString.help —

```

=====
IndexedString examples
=====

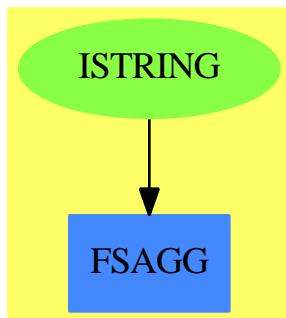
```

This domain implements low-level strings

See Also:

o)show IndexedString

IndexedString (ISTRING)



See

⇒ “Character” (CHAR) [4](#) on page [373](#)

⇒ “CharacterClass” (CCLASS) [4](#) on page [381](#)

⇒ “String” (STRING) [20](#) on page [2606](#)

Exports:

any?	coerce	concat	construct	convert
copy	copyInto!	count	delete	elt
empty	empty?	entries	entry?	eq?
eval	every?	fill!	find	first
hash	index?	indices	insert	latex
leftTrim	less?	lowerCase	lowerCase!	map
map!	match	match?	max	maxIndex
member?	members	merge	min	minIndex
more?	new	parts	prefix?	position
qelt	qsetelt!	reduce	remove	removeDuplicates
replace	reverse	reverse!	rightTrim	sample
select	setelt	size?	sort	sort!
sorted?	split	suffix?	substring?	swap!
trim	upperCase	upperCase!	#?	?<?
?<=?	?=?	?>?	?>=?	?~=?
?.?				

— domain ISTRING IndexedString —

)abbrev domain ISTRING IndexedString

++ Authors: Stephen Watt, Michael Monagan, Manuel Bronstein 1986 .. 1991

```

++ Description:
++ This domain implements low-level strings

IndexedString(mn:Integer): Export == Implementation where
  B ==> Boolean
  C ==> Character
  I ==> Integer
  N ==> NonNegativeInteger
  U ==> UniversalSegment Integer

Export ==> StringAggregate() with
  hash: % -> I
  ++ hash(x) provides a hashing function for strings

Implementation ==> add
  -- These assume Character's Rep is Small I
  Qelt    ==> QENUM$Lisp
  Qequal  ==> EQUAL$Lisp
  Qsetelt ==> QESSET$Lisp
  Qsize   ==> QCSIZE$Lisp
  Cheq    ==> EQL$Lisp
  Chlt    ==> QSLESSP$Lisp
  Chgt    ==> QSGREATERP$Lisp

  c: Character
  cc: CharacterClass

-- new n                == MAKE_-FULL_-CVEC(n, space$C)$Lisp
new(n, c)               == MAKE_-FULL_-CVEC(n, c)$Lisp
empty()                 == MAKE_-FULL_-CVEC(0$Lisp)$Lisp
empty?(s)               == Qsize(s) = 0
#s                      == Qsize(s)
s = t                   == Qequal(s, t)
s < t                   == CGREATERP(t,s)$Lisp
concat(s:%,t:%)         == STRCONC(s,t)$Lisp
copy s                  == COPY_-SEQ(s)$Lisp
insert(s:%, t:%, i:I)   == concat(concat(s(mn..i-1), t), s(i..))
coerce(s:%):OutputForm == outputForm(s pretend String)
minIndex s              == mn
upperCase_! s           == map_!(upperCase, s)
lowerCase_! s           == map_!(lowerCase, s)

latex s                 == concat("\mbox{'", concat(s pretend String, "'}"))

replace(s, sg, t) ==
  l := lo(sg) - mn
  m := #s
  n := #t
  h:I := if hasHi sg then hi(sg) - mn else maxIndex s - mn
  l < 0 or h >= m or h < l-1 => error "index out of range"

```

```

r := new((m-(h-1+1)+n)::N, space$C)
for k in 0.. for i in 0..l-1 repeat Qsetelt(r, k, Qelt(s, i))
for k in k.. for i in 0..n-1 repeat Qsetelt(r, k, Qelt(t, i))
for k in k.. for i in h+1..m-1 repeat Qsetelt(r, k, Qelt(s, i))
r

setelt(s:%, i:I, c:C) ==
  i < mn or i > maxIndex(s) => error "index out of range"
  Qsetelt(s, i - mn, c)
c

substring?(part, whole, startpos) ==
  np:I := Qsize part
  nw:I := Qsize whole
  (startpos := startpos - mn) < 0 => error "index out of bounds"
  np > nw - startpos => false
  for ip in 0..np-1 for iw in startpos.. repeat
    not Cheq(Qelt(part, ip), Qelt(whole, iw)) => return false
  true

position(s:%, t:%, startpos:I) ==
  (startpos := startpos - mn) < 0 => error "index out of bounds"
  startpos >= Qsize t => mn - 1
  r:I := STRPOS(s, t, startpos, NIL$Lisp)$Lisp
  EQ(r, NIL$Lisp)$Lisp => mn - 1
  r + mn

position(c: Character, t: %, startpos: I) ==
  (startpos := startpos - mn) < 0 => error "index out of bounds"
  startpos >= Qsize t => mn - 1
  for r in startpos..Qsize t - 1 repeat
    if Cheq(Qelt(t, r), c) then return r + mn
  mn - 1

position(cc: CharacterClass, t: %, startpos: I) ==
  (startpos := startpos - mn) < 0 => error "index out of bounds"
  startpos >= Qsize t => mn - 1
  for r in startpos..Qsize t - 1 repeat
    if member?(Qelt(t,r), cc) then return r + mn
  mn - 1

suffix?(s, t) ==
  (m := maxIndex s) > (n := maxIndex t) => false
  substring?(s, t, mn + n - m)

split(s, c) ==
  n := maxIndex s
  for i in mn..n while s.i = c repeat 0
  l := empty()$List(%)
  j:Integer -- j is conditionally intialized

```

```

while i <= n and (j := position(c, s, i)) >= mn repeat
  l := concat(s(i..j-1), l)
  for i in j..n while s.i = c repeat 0
if i <= n then l := concat(s(i..n), l)
reverse_! l

split(s, cc) ==
  n := maxIndex s
  for i in mn..n while member?(s.i,cc) repeat 0
  l := empty()$List(%)
  j:Integer -- j is conditionally intialized
  while i <= n and (j := position(cc, s, i)) >= mn repeat
    l := concat(s(i..j-1), l)
    for i in j..n while member?(s.i,cc) repeat 0
  if i <= n then l := concat(s(i..n), l)
  reverse_! l

leftTrim(s, c) ==
  n := maxIndex s
  for i in mn .. n while s.i = c repeat 0
  s(i..n)

leftTrim(s, cc) ==
  n := maxIndex s
  for i in mn .. n while member?(s.i,cc) repeat 0
  s(i..n)

rightTrim(s, c) ==
  for j in maxIndex s .. mn by -1 while s.j = c repeat 0
  s(minIndex(s)..j)

rightTrim(s, cc) ==
  for j in maxIndex s .. mn by -1 while member?(s.j, cc) repeat 0
  s(minIndex(s)..j)

concat l ==
  t := new(+/#s for s in l], space$C)
  i := mn
  for s in l repeat
    copyInto_!(t, s, i)
    i := i + #s
  t

copyInto_!(y, x, s) ==
  m := #x
  n := #y
  s := s - mn
  s < 0 or s+m > n => error "index out of range"
  RPLACSTR(y, s, m, x, 0, m)$Lisp
  y

```

```

elt(s:%, i:I) ==
  i < mn or i > maxIndex(s) => error "index out of range"
  Qelt(s, i - mn)

elt(s:%, sg:U) ==
  l := lo(sg) - mn
  h := if hasHi sg then hi(sg) - mn else maxIndex s - mn
  l < 0 or h >= #s => error "index out of bound"
  SUBSTRING(s, l, max(0, h-1+1))$Lisp

hash(s:$):Integer ==
  n:I := Qsize s
  zero? n => 0
--   one? n => ord(s.mn)
  (n = 1) => ord(s.mn)
  ord(s.mn) * ord s(mn+n-1) * ord s(mn + n quo 2)

match(pattern,target,wildcard) ==
  stringMatch(pattern,target,CHARACTER(wildcard)$Lisp)$Lisp

match?(pattern, target, dontcare) ==
  n := maxIndex pattern
  p := position(dontcare, pattern, m := minIndex pattern)::N
  p = m-1 => pattern = target
  (p ^= m) and not prefix?(pattern(m..p-1), target) => false
  i := p      -- index into target
  q := position(dontcare, pattern, p + 1)::N
  while q ^= m-1 repeat
    s := pattern(p+1..q-1)
    i := position(s, target, i)::N
    i = m-1 => return false
    i := i + #s
    p := q
    q := position(dontcare, pattern, q + 1)::N
  (p ^= n) and not suffix?(pattern(p+1..n), target) => false
  true

```

— ISTRING.dotabb —

```

"ISTRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ISTRING",
           shape=ellipse]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"ISTRING" -> "FSAGG"

```

domain IARRAY2 IndexedTwoDimensionalArray

An IndexedTwoDimensionalArray is a 2-dimensional array where the minimal row and column indices are parameters of the type. Rows and columns are returned as IndexedOneDimensionalArray's with minimal indices matching those of the IndexedTwoDimensionalArray. The index of the 'first' row may be obtained by calling the function 'minRowIndex'. The index of the 'first' column may be obtained by calling the function 'minColIndex'. The index of the first element of a 'Row' is the same as the index of the first column in an array and vice versa.

— IndexedTwoDimensionalArray.input —

```
)set break resume
)sys rm -f IndexedTwoDimensionalArray.output
)spool IndexedTwoDimensionalArray.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IndexedTwoDimensionalArray
--R
--R IndexedTwoDimensionalArray(R: Type,mnRow: Integer,mnCol: Integer) is a domain constructor
--R Abbreviation for IndexedTwoDimensionalArray is IARRAY2
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for IARRAY2
--R
--R----- Operations -----
--R copy : % -> %                               elt : (%,Integer,Integer,R) -> R
--R elt : (%,Integer,Integer) -> R               empty : () -> %
--R empty? : % -> Boolean                        eq? : (%,% ) -> Boolean
--R fill! : (% ,R) -> %                          map : (((R,R) -> R),%,%,R) -> %
--R map : (((R,R) -> R),%,%) -> %               map : ((R -> R),%) -> %
--R map! : ((R -> R),%) -> %                    maxColIndex : % -> Integer
--R maxRowIndex : % -> Integer                  minColIndex : % -> Integer
--R minRowIndex : % -> Integer                  ncols : % -> NonNegativeInteger
--R nrows : % -> NonNegativeInteger              parts : % -> List(R)
--R qelt : (% ,Integer,Integer) -> R            sample : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?? : (% ,%) -> Boolean if R has SETCAT
--R any? : ((R -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if R has SETCAT
--R column : (% ,Integer) -> IndexedOneDimensionalArray(R,mnRow)
--R count : (R,% ) -> NonNegativeInteger if $ has finiteAggregate and R has SETCAT
--R count : ((R -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R eval : (% ,List(R),List(R)) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (% ,R,R) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (% ,Equation(R)) -> % if R has EVALAB(R) and R has SETCAT
```

```

--R eval : (% ,List(Equation(R))) -> % if R has EVALAB(R) and R has SETCAT
--R every? : ((R -> Boolean),%) -> Boolean if $ has finiteAggregate
--R hash : % -> SingleInteger if R has SETCAT
--R latex : % -> String if R has SETCAT
--R less? : (% ,NonNegativeInteger) -> Boolean
--R member? : (R,%) -> Boolean if $ has finiteAggregate and R has SETCAT
--R members : % -> List(R) if $ has finiteAggregate
--R more? : (% ,NonNegativeInteger) -> Boolean
--R new : (NonNegativeInteger,NonNegativeInteger,R) -> %
--R qsetelt! : (% ,Integer,Integer,R) -> R
--R row : (% ,Integer) -> IndexedOneDimensionalArray(R,mnCol)
--R setColumn! : (% ,Integer,IndexedOneDimensionalArray(R,mnRow)) -> %
--R setRow! : (% ,Integer,IndexedOneDimensionalArray(R,mnCol)) -> %
--R setelt : (% ,Integer,Integer,R) -> R
--R size? : (% ,NonNegativeInteger) -> Boolean
--R ?~=? : (% ,%) -> Boolean if R has SETCAT
--R
--E 1

```

```

)spool
)lisp (bye)

```

— IndexedTwoDimensionalArray.help —

```

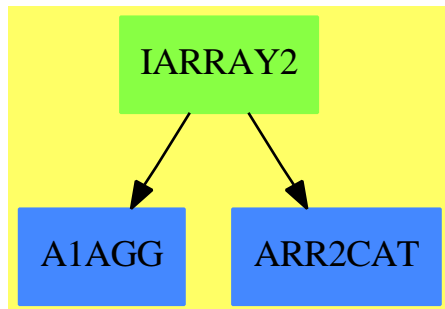
=====
IndexedTwoDimensionalArray examples
=====

```

This domain implements two dimensional arrays

See Also:

o)show IndexedTwoDimensionalArray

IndexedTwoDimensionalArray (IARRAY2)

See

⇒ “InnerIndexedTwoDimensionalArray” (IARRAY2) [10](#) on page [1290](#)

⇒ “TwoDimensionalArray” (ARRAY2) [21](#) on page [2763](#)

Exports:

any?	coerce	column	copy	count
count	elt	empty	empty?	eq?
eval	every?	fill!	hash	latex
less?	maxColIndex	maxRowIndex	map	map!
member?	members	minColIndex	minRowIndex	more?
ncols	new	nrows	parts	qelt
qsetelt!	row	sample	setColumn!	setRow!
setelt	size?	#?	?=?	?~=?

— domain **IARRAY2** IndexedTwoDimensionalArray —

```

)abbrev domain IARRAY2 IndexedTwoDimensionalArray
++ Author: Mark Botch
++ Description:
++ This domain implements two dimensional arrays
  
```

```

IndexedTwoDimensionalArray(R,mnRow,mnCol):Exports == Implementation where
  
```

```

  R : Type
  mnRow, mnCol : Integer
  Row ==> IndexedOneDimensionalArray(R,mnCol)
  Col ==> IndexedOneDimensionalArray(R,mnRow)
  
```

```

Exports ==> TwoDimensionalArrayCategory(R,Row,Col)
  
```

```

Implementation ==>
  InnerIndexedTwoDimensionalArray(R,mnRow,mnCol,Row,Col)
  
```

—

— IARRAY2.dotabb —

```

"IARRAY2" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IARRAY2"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"ARR2CAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ARR2CAT"]
"IARRAY2" -> "ARR2CAT"
"IARRAY2" -> "A1AGG"

```

domain IVECTOR IndexedVector

— IndexedVector.input —

```

)set break resume
)sys rm -f IndexedVector.output
)spool IndexedVector.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IndexedVector
--R
--R IndexedVector(R: Type,mn: Integer) is a domain constructor
--R Abbreviation for IndexedVector is IVECTOR
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for IVECTOR
--R
--R----- Operations -----
--R -? : % -> % if R has ABELGRP          concat : List(%) -> %
--R concat : (%,%) -> %                  concat : (R,%) -> %
--R concat : (%,R) -> %                  construct : List(R) -> %
--R copy : % -> %                        delete : (%,Integer) -> %
--R ?.? : (%,Integer) -> R              elt : (%,Integer,R) -> R
--R empty : () -> %                     empty? : % -> Boolean
--R entries : % -> List(R)              eq? : (%,%) -> Boolean
--R index? : (Integer,%) -> Boolean      indices : % -> List(Integer)
--R insert : (%,%,Integer) -> %         insert : (R,%,Integer) -> %
--R map : (((R,R) -> R),%,%) -> %       map : ((R -> R),%) -> %
--R new : (NonNegativeInteger,R) -> %   qelt : (%,Integer) -> R
--R reverse : % -> %                    sample : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?? : (%,R) -> % if R has MONOID
--R ?? : (R,%) -> % if R has MONOID

```

```

--R ?? : (Integer,%) -> % if R has ABELGRP
--R ?+? : (%,%) -> % if R has ABELSG
--R ?-? : (%,%) -> % if R has ABELGRP
--R ?<? : (%,%) -> Boolean if R has ORDSET
--R ?<=? : (%,%) -> Boolean if R has ORDSET
--R ?=? : (%,%) -> Boolean if R has SETCAT
--R ?>? : (%,%) -> Boolean if R has ORDSET
--R ?>=? : (%,%) -> Boolean if R has ORDSET
--R any? : ((R -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if R has SETCAT
--R convert : % -> InputForm if R has KONVERT(INFORM)
--R copyInto! : (%,%,Integer) -> % if $ has shallowlyMutable
--R count : (R,%) -> NonNegativeInteger if $ has finiteAggregate and R has SETCAT
--R count : ((R -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R cross : (%,%) -> % if R has RING
--R delete : (%,UniversalSegment(Integer)) -> %
--R dot : (%,%) -> R if R has RING
--R ?.? : (%,UniversalSegment(Integer)) -> %
--R entry? : (R,%) -> Boolean if $ has finiteAggregate and R has SETCAT
--R eval : (%,List(R),List(R)) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (%,R,R) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (%,Equation(R)) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (%,List(Equation(R))) -> % if R has EVALAB(R) and R has SETCAT
--R every? : ((R -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (%,R) -> % if $ has shallowlyMutable
--R find : ((R -> Boolean),%) -> Union(R,"failed")
--R first : % -> R if Integer has ORDSET
--R hash : % -> SingleInteger if R has SETCAT
--R latex : % -> String if R has SETCAT
--R length : % -> R if R has RADCAT and R has RING
--R less? : (%,NonNegativeInteger) -> Boolean
--R magnitude : % -> R if R has RADCAT and R has RING
--R map! : ((R -> R),%) -> % if $ has shallowlyMutable
--R max : (%,%) -> % if R has ORDSET
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (R,%) -> Boolean if $ has finiteAggregate and R has SETCAT
--R members : % -> List(R) if $ has finiteAggregate
--R merge : (%,%) -> % if R has ORDSET
--R merge : ((R,R) -> Boolean),%,%) -> %
--R min : (%,%) -> % if R has ORDSET
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (%,NonNegativeInteger) -> Boolean
--R outerProduct : (%,%) -> Matrix(R) if R has RING
--R parts : % -> List(R) if $ has finiteAggregate
--R position : (R,%,Integer) -> Integer if R has SETCAT
--R position : (R,%) -> Integer if R has SETCAT
--R position : ((R -> Boolean),%) -> Integer
--R qsetelt! : (%,Integer,R) -> R if $ has shallowlyMutable
--R reduce : ((R,R) -> R),%) -> R if $ has finiteAggregate
--R reduce : ((R,R) -> R),%,R) -> R if $ has finiteAggregate

```

```

--R reduce : (((R,R) -> R),%,R,R) -> R if $ has finiteAggregate and R has SETCAT
--R remove : ((R -> Boolean),%) -> % if $ has finiteAggregate
--R remove : (R,%) -> % if $ has finiteAggregate and R has SETCAT
--R removeDuplicates : % -> % if $ has finiteAggregate and R has SETCAT
--R reverse! : % -> % if $ has shallowlyMutable
--R select : ((R -> Boolean),%) -> % if $ has finiteAggregate
--R setelt : (%,UniversalSegment(Integer),R) -> R if $ has shallowlyMutable
--R setelt : (%,Integer,R) -> R if $ has shallowlyMutable
--R size? : (%,NonNegativeInteger) -> Boolean
--R sort : % -> % if R has ORDSET
--R sort : (((R,R) -> Boolean),%) -> %
--R sort! : % -> % if $ has shallowlyMutable and R has ORDSET
--R sort! : (((R,R) -> Boolean),%) -> % if $ has shallowlyMutable
--R sorted? : % -> Boolean if R has ORDSET
--R sorted? : (((R,R) -> Boolean),%) -> Boolean
--R swap! : (%,Integer,Integer) -> Void if $ has shallowlyMutable
--R zero : NonNegativeInteger -> % if R has ABELMON
--R ?~=? : (%,%) -> Boolean if R has SETCAT
--R
--E 1

```

```

)spool
)lisp (bye)

```

— IndexedVector.help —

```

=====
IndexedVector examples
=====

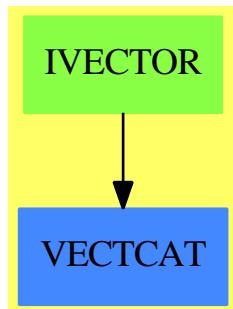
```

This type represents vector like objects with varying lengths
and a user-specified initial index.

See Also:

o)show IndexedVector

IndexedVector (IVECTOR)



Exports:

any?	coerce	concat	construct	convert
copy	copyInto!	count	cross	delete
dot	elt	empty	empty?	entries
entry?	eq?	eval	every?	fill!
find	first	hash	index?	indices
insert	latex	length	less?	magnitude
map!	max	maxIndex	member?	members
merge	min	minIndex	more?	new
outerProduct	parts	position	qelt	qsetelt!
reduce	remove	removeDuplicates	reverse	reverse!
sample	select	setelt	size?	sort
sort!	sorted?	swap!	zero	#?
?*?	?+?	?-?	?<?	?<=?
?=?	?>?	?>=?	?~=?	-?
?..?				

— domain IVECTOR IndexedVector —

```

)abbrev domain IVECTOR IndexedVector
++ Author: Mark Botch
++ Description:
++ This type represents vector like objects with varying lengths
++ and a user-specified initial index.
  
```

```

IndexedVector(R:Type, mn:Integer):
  VectorCategory R == IndexedOneDimensionalArray(R, mn)
  
```

—————

— IVECTOR.dotabb —

"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]

"VECTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=VECTCAT"]
 "IVECTOR" -> "VECTCAT"

domain ITUPLE InfiniteTuple

— InfiniteTuple.input —

```
)set break resume
)sys rm -f InfiniteTuple.output
)spool InfiniteTuple.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InfiniteTuple
--R
--R InfiniteTuple(S: Type) is a domain constructor
--R Abbreviation for InfiniteTuple is ITUPLE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ITUPLE
--R
--R----- Operations -----
--R coerce : % -> OutputForm          construct : % -> Stream(S)
--R generate : ((S -> S),S) -> %      map : ((S -> S),%) -> %
--R select : ((S -> Boolean),%) -> %
--R filterUntil : ((S -> Boolean),%) -> %
--R filterWhile : ((S -> Boolean),%) -> %
--R
--E 1

)spool
)lisp (bye)
```

— InfiniteTuple.help —

```
=====
InfiniteTuple examples
=====
```

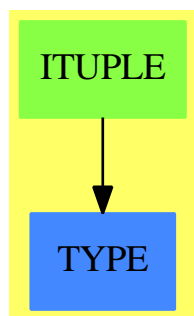
This package implements 'infinite tuples' for the interpreter.

The representation is a stream.

See Also:

o)show InfiniteTuple

InfiniteTuple (ITUPLE)



Exports:

coerce construct filterUntil filterWhile generate map select

— domain ITUPLE InfiniteTuple —

```

)abbrev domain ITUPLE InfiniteTuple
++ Author: Clifton J. Williamson
++ Date Created: 16 February 1990
++ Date Last Updated: 16 February 1990
++ Description:
++ This package implements 'infinite tuples' for the interpreter.
++ The representation is a stream.
  
```

InfiniteTuple(S:Type): Exports == Implementation where

```

Exports ==> CoercibleTo OutputForm with
  map: (S -> S, %) -> %
    ++ map(f,t) replaces the tuple t
    ++ by \spad{[f(x) for x in t]}.
  filterWhile: (S -> Boolean, %) -> %
    ++ filterWhile(p,t) returns \spad{[x for x in t while p(x)]}.
  filterUntil: (S -> Boolean, %) -> %
    ++ filterUntil(p,t) returns \spad{[x for x in t while not p(x)]}.
  select: (S -> Boolean, %) -> %
    ++ select(p,t) returns \spad{[x for x in t | p(x)]}.
  
```

```

generate: (S -> S,S) -> %
  ++ generate(f,s) returns \spad{[s,f(s),f(f(s)),...]}.
construct: % -> Stream S
  ++ construct(t) converts an infinite tuple to a stream.

Implementation ==> Stream S add
  generate(f,x) == generate(f,x)$Stream(S) pretend %
  filterWhile(f, x) == filterWhile(f,x pretend Stream(S))$Stream(S) pretend %
  filterUntil(f, x) == filterUntil(f,x pretend Stream(S))$Stream(S) pretend %
  select(f, x) == select(f,x pretend Stream(S))$Stream(S) pretend %
  construct x == x pretend Stream(S)
--      coerce x ==
--      coerce(x)$Stream(S)

```

— ITUPLE.dotabb —

```

"ITUPLE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ITUPLE"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"ITUPLE" -> "TYPE"

```

domain INFCLSPT InfinitelyClosePoint

— InfinitelyClosePoint.input —

```

)set break resume
)sys rm -f InfinitelyClosePoint.output
)spool InfinitelyClosePoint.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InfinitelyClosePoint
--R
--R InfinitelyClosePoint(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory(K,E,OrderedVariableList(syn
--R Abbreviation for InfinitelyClosePoint is INFCLSPT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for INFCLSPT
--R
--R----- Operations -----

```

```

--R ?=? : (%,% ) -> Boolean
--R chartV : % -> BLMET
--R create : (ProjPt,PolyRing) -> %
--R excpDivV : % -> DIVISOR
--R fullOutput : () -> Boolean
--R hash : % -> SingleInteger
--R localParamV : % -> List(PCS)
--R multV : % -> NonNegativeInteger
--R setchart! : (% ,BLMET) -> BLMET
--R symbNameV : % -> Symbol
--R create : (ProjPt,DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],K),AffinePlane(K)) -> DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],K)
--R curveV : % -> DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],K)
--R setcurve! : (% ,DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],K)) -> DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],K)
--R setexcpDiv! : (% ,DIVISOR) -> DIVISOR
--R setlocalParam! : (% ,List(PCS)) -> List(PCS)
--R setlocalPoint! : (% ,AffinePlane(K)) -> AffinePlane(K)
--R setmult! : (% ,NonNegativeInteger) -> NonNegativeInteger
--R setsubmult! : (% ,NonNegativeInteger) -> NonNegativeInteger
--R setsymbName! : (% ,Symbol) -> Symbol
--R subMultV : % -> NonNegativeInteger
--R
--E 1
)spool
)lisp (bye)

```

— InfinitelyClosePoint.help —

```

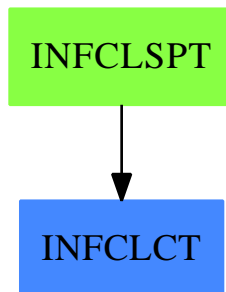
=====
InfinitelyClosePoint examples
=====

```

This domain is part of the PAFF package

See Also:

o)show InfinitelyClosePoint

InfinitelyClosePoint (INFCLSPT)**Exports:**

?=?	?~=?	actualExtensionV
chartV	coerce	create
curveV	degree	excpDivV
fullOut	fullOutput	fullOutput
hash	latex	localParamV
localPointV	multV	pointV
setchart!	setcurve!	setexcpDiv!
setlocalParam!	setlocalPoint!	setmult!
setpoint!	setsubmult!	setsymbName!
subMultV	symbNameV	

— domain **INFCLSPT** **InfinitelyClosePoint** —

```

)abbrev domain INFCLSPT InfinitelyClosePoint
++ Authors: Gaetan Hache
++ Date Created: june 1996
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ This domain is part of the PAFF package
InfinitelyClosePoint(K,symb,PolyRing,E,ProjPt,PCS,Plc,DIVISOR,BLMET):Exports == Implementation where
  K:Field
  symb: List Symbol
  E:DirectProductCategory(#symb,NonNegativeInteger)
  OV ==> OrderedVariableList(symb)
  PolyRing: PolynomialCategory(K,E,OV)

  bls      ==> ['X','Y']
  BlUpRing ==> DistributedMultivariatePolynomial(bls , K)
  E2       ==> DirectProduct( #bls , NonNegativeInteger )
  outRec   ==> Record(name:Symbol,mult:NonNegativeInteger)
  AFP      ==> AffinePlane(K)
  OV2      ==> OrderedVariableList( bls )

  PCS: LocalPowerSeriesCategory(K)

```

```

ProjPt:ProjectiveSpaceCategory(K)
Plc: PlacesCategory(K,PCS)
DIVISOR: DivisorCategory(Plc)
BLMET : BlowUpMethodCategory

bigoutRecBLQT ==> Record(dominate:ProjPt,_
                        name:Symbol,_
                        mult:NonNegativeInteger,_
                        defCurve:BLUpRing,_
                        localPoint:AFP,_
                        chart:BLMET,_
                        expD:DIVISOR)

bigoutRecHN ==> Record(dominate:ProjPt,_
                      name:Symbol,_
                      mult:NonNegativeInteger,_
                      defCurve:BLUpRing,_
                      localPoint:AFP,_
                      chart:BLMET,_
                      subMultip: NonNegativeInteger,_
                      expD:DIVISOR)

representation ==> Record(point:ProjPt,_
                          curve:BLUpRing,_
                          localPoint:AFP,_
                          mult:NonNegativeInteger,_
                          chrt:BLMET,_
                          subMultiplicity:NonNegativeInteger,_
                          excpDiv:DIVISOR,_
                          localParam:List(PCS),_
                          actualExtension:K,_
                          symbName:Symbol)

Exports == InfinitelyClosePointCategory(K,symb,PolyRing,E,ProjPt,PCS,Plc,DIVISOR,BLMET) with

fullOut: % -> OutputForm
++ fullOut(tr) yields a full output of tr (see function fullOutput).

fullOutput: Boolean -> Boolean
++ fullOutput(b) sets a flag such that when true, a coerce to
++ OutputForm yields the full output of tr, otherwise encode(tr) is
++ output (see encode function). The default is false.

fullOutput: () -> Boolean
++ fullOutput returns the value of the flag set by fullOutput(b).

Implementation == representation add
Rep := representation

```

```

polyRing2BiRing: (PolyRing, Integer) -> B1UpRing
polyRing2BiRing(pol,nV)==
  zero? pol => 0$B1UpRing
  d:= degree pol
  lc:= leadingCoefficient pol
  dd: List NonNegativeInteger := entries d
  ddr:=vector([dd.i for i in 1..#dd | ^(i=nV)])$Vector(NonNegativeInteger)
  ddre:E2 := directProduct( ddr )$E2
  monomial(lc,ddre)$B1UpRing + polyRing2BiRing( reductum pol , nV )

projPt2affPt: (ProjPt, Integer) -> AFP
projPt2affPt(pt,nV)==
  ll:= pt :: List(K)
  l:= [ ll.i for i in 1..#ll | ^(i = nV )]
  affinePoint( l)

fullOut(a)==
  oo: bigoutRecBLQT
  oo2: bigoutRecHN
  BLMET has BlowUpWithQuadTrans =>
    oo:= [ pointV(a), symbNameV(a), multV(a), curveV(a), _
          localPointV(a), chartV(a), excpDivV(a) ]$bigoutRecBLQT
    oo :: OutputForm
  BLMET has BlowUpWithHamburgerNoether =>
    oo2:= [ pointV(a), symbNameV(a), multV(a), curveV(a), _
          localPointV(a), chartV(a), subMultV(a), excpDivV(a) ]$bigoutRecHN
    oo2 :: OutputForm

fullOutputFlag:Boolean:=false()

fullOutput(f)== fullOutputFlag:=f

fullOutput == fullOutputFlag

coerce(a:%):OutputForm==
  fullOutput() => fullOut(a)
  oo:outRec:= [ symbNameV(a) , multV(a) ]$outRec
  oo :: OutputForm

degree(a)==
  K has PseudoAlgebraicClosureOfPerfectFieldCategory => extDegree actualExtensionV a
  1

create(pointA,curveA,localPointA,multA,chartA,subM,excpDivA,atcL,aName)== -- CHH
  ([pointA,curveA,localPointA,multA,chartA,subM,excpDivA,empty()$List(PCS),atcL,aName]$Rep)::%

create(pointA,curveA)==
  nV := lastNonNul pointA
  localPointA := projPt2affPt(pointA,nV)

```

```

multA:NonNegativeInteger:=0$NonNegativeInteger
chartA:BLMET
if BLMET has QuadraticTransform then chartA:=( [0,0, nV] :: List Integer ) :: BLMET -- C
if BLMET has HamburgerNoether then
  chartA := createHN( 0,0,nV,0,0,true,"right") -- A changer le "right"
excpDivA:DIVISOR:= 0$DIVISOR
actL:K:=definingField pointA
aName:Symbol:=new(P)$Symbol
affCurvA : BlUpRing := polyRing2BiRing(curveA,nV)
create(pointA,affCurvA,localPointA,multA,chartA,0$NonNegativeInteger,excpDivA,actL,aName)

subMultV(a:%)== (a:Rep)(subMultiplicity)

setsubmult_!(a:%,sm:NonNegativeInteger)== (a:Rep)(subMultiplicity) := sm

pointV(a:%)      ==(a:Rep)(point)

symbNameV(a:%)    ==(a:Rep)(symbName)

curveV(a:%)      ==(a:Rep)(curve)

localPointV(a:%)  ==(a:Rep)(localPoint)

multV(a:%)        ==(a:Rep)(mult)

chartV(a:%)       ==(a:Rep)(chrt) -- CHH

excpDivV(a:%)     ==(a:Rep)(excpDiv)

localParamV(a:%)  ==(a:Rep)(localParam)

actualExtensionV(a:%) == (a:Rep)(actualExtension)

setpoint_!(a:%,n:ProjPt)      ==(a:Rep)(point):=n

setcurve_!(a:%,n:BlUpRing)    ==(a:Rep)(curve):=n

setlocalPoint_!(a:%,n:AFP)    ==(a:Rep)(localPoint):=n

setmult_!(a:%,n:NonNegativeInteger)      ==(a:Rep)(mult):=n

setchart_!(a:%,n:BLMET)      ==(a:Rep)(chrt):=n -- CHH

setlocalParam_!(a:%,n:List(PCS)) ==(a:Rep)(localParam):=n

setexcpDiv_!(a:%,n:DIVISOR) ==(a:Rep)(excpDiv):=n

setsymbName_!(a:%,n:Symbol) ==(a:Rep)(symbName):=n

```

— INFCLSPT.dotabb —

```
"INFCLSPT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=INFCLSPT"]
"INFCLCT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=INFCLCT"]
"INFCLSPT" -> "INFCLCT"
```

domain INFCLSPS InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField

— InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField.input

```
)set break resume
)sys rm -f InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField.output
)spool InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField
--R
--R InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField(K: FiniteFieldCategory,symb: List(Symbol),BL
--R Abbreviation for InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField is INFCLSPS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for INFCLSPS
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          chartV : % -> BLMET
--R coerce : % -> OutputForm          degree : % -> PositiveInteger
--R fullOut : % -> OutputForm          fullOutput : () -> Boolean
--R fullOutput : Boolean -> Boolean    hash : % -> SingleInteger
--R latex : % -> String                multV : % -> NonNegativeInteger
--R setchart! : (%,BLMET) -> BLMET    symbNameV : % -> Symbol
--R ?~=? : (%,% ) -> Boolean
--R actualExtensionV : % -> PseudoAlgebraicClosureOfFiniteField(K)
--R create : (ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K),DistributedMultivariatePolynomial(s
--R create : (ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K),DistributedMultivariatePolynomial([
--R curveV : % -> DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],PseudoAlgebraicClosureOfFinite
--R excpDivV : % -> Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K))
--R localParamV : % -> List(NeitherSparseOrDensePowerSeries(PseudoAlgebraicClosureOfFiniteField(K)))
```



```

--R localPointV : % -> AffinePlane(PseudoAlgebraicClosureOfFiniteField(K))
--R pointV : % -> ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K)
--R setcurve! : (% ,DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],PseudoAlgebraicCl
--R setexcpDiv! : (% ,Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K))) -> Divisor(Place
--R setlocalParam! : (% ,List(NeitherSparseOrDensePowerSeries(PseudoAlgebraicClosureOfFiniteField
--R setlocalPoint! : (% ,AffinePlane(PseudoAlgebraicClosureOfFiniteField(K))) -> AffinePlane(Pseu
--R setmult! : (% ,NonNegativeInteger) -> NonNegativeInteger
--R setpoint! : (% ,ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K)) -> ProjectivePlane
--R setsubmult! : (% ,NonNegativeInteger) -> NonNegativeInteger
--R setsymbName! : (% ,Symbol) -> Symbol
--R subMultV : % -> NonNegativeInteger
--R
--E 1

)spool
)lisp (bye)

```

— **InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField.help**

```

=====
InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField examples
=====

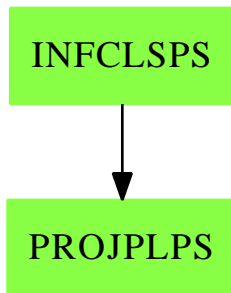
```

This domain is part of the PAFF package

See Also:

o)show InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField

InfinitelyClosePointOverPseudoAlgebraicClosureOfFiniteField (INF-CLSPS)



Exports:

?=?	?~=?	actualExtensionV
chartV	coerce	create
curveV	degree	excpDivV
fullOut	fullOutput	hash
latex	localParamV	localPointV
multV	pointV	setchart!
setcurve!	setexcpDiv!	setlocalParam!
setlocalPoint!	setmult!	setpoint!
setsubmult!	setsymbName!	subMultV
symbNameV		

— domain INFCLSPS InfinitelyClosePointOverPseudoAlgebraicClosureOfFiniteField —

```

)abbrev domain INFCLSPS InfinitelyClosePointOverPseudoAlgebraicClosureOfFiniteField
++ Authors: Gaetan Hache
++ Date Created: june 1996
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ This domain is part of the PAFF package
InfinitelyClosePointOverPseudoAlgebraicClosureOfFiniteField(K,symb,BLMET):_
  Exports == Implementation where

  K:FiniteFieldCategory
  symb: List Symbol
  BLMET : BlowUpMethodCategory

  E      ==> DirectProduct(#symb,NonNegativeInteger)
  KK     ==> PseudoAlgebraicClosureOfFiniteField(K)
  PolyRing ==> DistributedMultivariatePolynomial(symb,KK)
  ProjPt ==> ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K)
  PCS    ==> NeitherSparseOrDensePowerSeries(KK)
  
```

```

Plc          ==> PlacesOverPseudoAlgebraicClosureOfFiniteField(K)
DIVISOR      ==> Divisor(Plc)

Exports == InfinitelyClosePointCategory(KK,symb,PolyRing,E,ProjPt,_,
                                           PCS,Plc,DIVISOR,BLMET) with

fullOut: % -> OutputForm
++ fullOut(tr) yields a full output of tr (see function fullOutput).

fullOutput: Boolean -> Boolean

++ fullOutput(b) sets a flag such that when true, a coerce to OutputForm
++ yields the full output of tr, otherwise encode(tr) is output
++ (see encode function). The default is false.

fullOutput: () -> Boolean
++ fullOutput returns the value of the flag set by fullOutput(b).

Implementation == InfinitelyClosePoint(KK,symb,PolyRing,E,ProjPt,_,
                                           PCS,Plc,DIVISOR,BLMET)

```

— INFCLSPS.dotabb —

```

"INFCLSPS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=INFCLSPS"]
"PROJPLPS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PROJPLPS"]
"INFCLSPS" -> "PROJPLPS"

```

domain IAN InnerAlgebraicNumber

— InnerAlgebraicNumber.input —

```

)set break resume
)sys rm -f InnerAlgebraicNumber.output
)spool InnerAlgebraicNumber.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerAlgebraicNumber
--R

```

```

--R InnerAlgebraicNumber is a domain constructor
--R Abbreviation for InnerAlgebraicNumber is IAN
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for IAN
--R
--R----- Operations -----
--R ?? : (PositiveInteger,%) -> %      ?? : (Integer,%) -> %
--R ?? : (%,%) -> %                  ?? : (%,Fraction(Integer)) -> %
--R ?? : (Fraction(Integer),%) -> %    ??? : (%,PositiveInteger) -> %
--R ??? : (%,Integer) -> %           +? : (%,%) -> %
--R -? : % -> %                      -? : (%,%) -> %
--R ?/? : (%,%) -> %                 ?<? : (%,%) -> Boolean
--R ?<=? : (%,%) -> Boolean           ?=? : (%,%) -> Boolean
--R ?>? : (%,%) -> Boolean           ?>=? : (%,%) -> Boolean
--R D : % -> %                       D : (%,NonNegativeInteger) -> %
--R 1 : () -> %                      0 : () -> %
--R ?? : (%,PositiveInteger) -> %    ?? : (%,Integer) -> %
--R associates? : (%,%) -> Boolean    belong? : BasicOperator -> Boolean
--R box : List(%) -> %                box : % -> %
--R coerce : Integer -> %             coerce : % -> %
--R coerce : Fraction(Integer) -> %   coerce : Kernel(%) -> %
--R coerce : % -> OutputForm          convert : % -> Complex(Float)
--R convert : % -> DoubleFloat         convert : % -> Float
--R differentiate : % -> %            distribute : (%,%) -> %
--R distribute : % -> %               elt : (BasicOperator,%,%) -> %
--R elt : (BasicOperator,%) -> %      eval : (%,%,%) -> %
--R eval : (%,Equation(%)) -> %       eval : (%,Kernel(%),%) -> %
--R factor : % -> Factored(%)         freeOf? : (%,Symbol) -> Boolean
--R freeOf? : (%,%) -> Boolean         gcd : (%,%) -> %
--R gcd : List(%) -> %                hash : % -> SingleInteger
--R height : % -> NonNegativeInteger   inv : % -> %
--R is? : (%,Symbol) -> Boolean        kernel : (BasicOperator,%) -> %
--R kernels : % -> List(Kernel(%))     latex : % -> String
--R lcm : (%,%) -> %                  lcm : List(%) -> %
--R max : (%,%) -> %                  min : (%,%) -> %
--R norm : (%,List(Kernel(%))) -> %    norm : (%,Kernel(%)) -> %
--R nthRoot : (%,Integer) -> %         one? : % -> Boolean
--R paren : List(%) -> %              paren : % -> %
--R prime? : % -> Boolean              ?quo? : (%,%) -> %
--R recip : % -> Union(%, "failed")    reduce : % -> %
--R ?rem? : (%,%) -> %                retract : % -> Fraction(Integer)
--R retract : % -> Integer             retract : % -> Kernel(%)
--R rootOf : Polynomial(%) -> %        sample : () -> %
--R sizeLess? : (%,%) -> Boolean       sqrt : % -> %
--R squareFree : % -> Factored(%)      squareFreePart : % -> %
--R subst : (%,Equation(%)) -> %      tower : % -> List(Kernel(%))
--R trueEqual : (%,%) -> Boolean       unit? : % -> Boolean
--R unitCanonical : % -> %            zero? : % -> Boolean
--R zeroOf : Polynomial(%) -> %       ?~=? : (%,%) -> Boolean
--R ?? : (NonNegativeInteger,%) -> %

```

```

--R ??? : (% , NonNegativeInteger) -> %
--R ??? : (% , Fraction(Integer)) -> %
--R ?? : (% , NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R coerce : SparseMultivariatePolynomial(Integer, Kernel(%)) -> %
--R definingPolynomial : % -> % if $ has RING
--R denom : % -> SparseMultivariatePolynomial(Integer, Kernel(%))
--R differentiate : (% , NonNegativeInteger) -> %
--R divide : (% , %) -> Record(quotient: % , remainder: %)
--R elt : (BasicOperator, List(%)) -> %
--R elt : (BasicOperator, % , % , % , %) -> %
--R elt : (BasicOperator, % , % , %) -> %
--R euclideanSize : % -> NonNegativeInteger
--R eval : (% , BasicOperator, (% -> %)) -> %
--R eval : (% , BasicOperator, (List(% -> %)) -> %
--R eval : (% , List(BasicOperator), List((List(% -> %))) -> %
--R eval : (% , List(BasicOperator), List((% -> %))) -> %
--R eval : (% , Symbol, (% -> %)) -> %
--R eval : (% , Symbol, (List(% -> %)) -> %
--R eval : (% , List(Symbol), List((List(% -> %))) -> %
--R eval : (% , List(Symbol), List((% -> %))) -> %
--R eval : (% , List(%), List(%)) -> %
--R eval : (% , List(Equation(%))) -> %
--R eval : (% , List(Kernel(%)), List(%)) -> %
--R even? : % -> Boolean if $ has RETRACT(INT)
--R expressIdealMember : (List(%), %) -> Union(List(%), "failed")
--R exquo : (% , %) -> Union(% , "failed")
--R extendedEuclidean : (% , %) -> Record(coef1: % , coef2: % , generator: %)
--R extendedEuclidean : (% , % , %) -> Union(Record(coef1: % , coef2: %), "failed")
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUniva
--R is? : (% , BasicOperator) -> Boolean
--R kernel : (BasicOperator, List(%)) -> %
--R mainKernel : % -> Union(Kernel(%), "failed")
--R map : ((% -> %), Kernel(%)) -> %
--R minPoly : Kernel(% -> SparseUnivariatePolynomial(%)) if $ has RING
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed")
--R norm : (SparseUnivariatePolynomial(%), List(Kernel(%))) -> SparseUnivariatePolynomial(%)
--R norm : (SparseUnivariatePolynomial(%), Kernel(%)) -> SparseUnivariatePolynomial(%)
--R numer : % -> SparseMultivariatePolynomial(Integer, Kernel(%))
--R odd? : % -> Boolean if $ has RETRACT(INT)
--R operator : BasicOperator -> BasicOperator
--R operators : % -> List(BasicOperator)
--R principalIdeal : List(% -> Record(coef: List(%), generator: %))
--R reducedSystem : Matrix(% -> Matrix(Fraction(Integer)))
--R reducedSystem : (Matrix(%), Vector(%)) -> Record(mat: Matrix(Fraction(Integer)), vec: Vector(F
--R reducedSystem : Matrix(% -> Matrix(Integer))
--R reducedSystem : (Matrix(%), Vector(%)) -> Record(mat: Matrix(Integer), vec: Vector(Integer))
--R retractIfCan : % -> Union(Fraction(Integer), "failed")
--R retractIfCan : % -> Union(Integer, "failed")
--R retractIfCan : % -> Union(Kernel(%), "failed")

```

```

--R rootOf : SparseUnivariatePolynomial(%) -> %
--R rootOf : (SparseUnivariatePolynomial(%),Symbol) -> %
--R rootsOf : Polynomial(%) -> List(%)
--R rootsOf : SparseUnivariatePolynomial(%) -> List(%)
--R rootsOf : (SparseUnivariatePolynomial(%),Symbol) -> List(%)
--R subst : (%,List(Kernel(%)),List(%)) -> %
--R subst : (%,List(Equation(%))) -> %
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R zeroOf : SparseUnivariatePolynomial(%) -> %
--R zeroOf : (SparseUnivariatePolynomial(%),Symbol) -> %
--R zerosOf : Polynomial(%) -> List(%)
--R zerosOf : SparseUnivariatePolynomial(%) -> List(%)
--R zerosOf : (SparseUnivariatePolynomial(%),Symbol) -> List(%)
--R
--E 1

```

```

)spool
)lisp (bye)

```

— InnerAlgebraicNumber.help —

```

=====
InnerAlgebraicNumber examples
=====

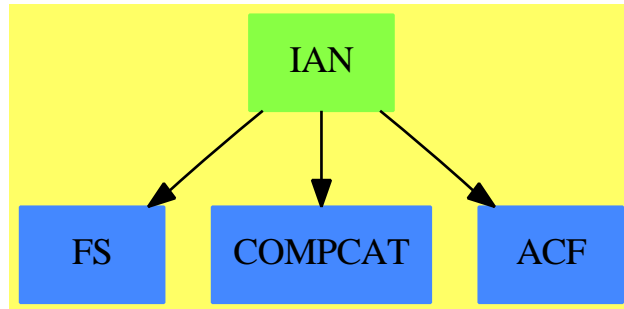
```

Algebraic closure of the rational numbers.

See Also:

o)show InnerAlgebraicNumber

InnerAlgebraicNumber (IAN)



See

⇒ “AlgebraicNumber” (AN) 2 on page 35

Exports:

0	1	associates?	belong?
box	characteristic	coerce	convert
D	definingPolynomial	denom	differentiate
distribute	divide	elt	euclideanSize
eval	even?	expressIdealMember	exquo
extendedEuclidean	factor	freeOf?	gcd
gcdPolynomial	hash	height	inv
is?	kernel	kernels	latex
lcm	mainKernel	map	max
min	minPoly	multiEuclidean	norm
nthRoot	numer	odd?	one?
operator	operators	paren	prime?
principalIdeal	recip	reduce	reducedSystem
retract	retractIfCan	rootOf	rootsOf
sample	sizeLess?	sqrt	squareFree
squareFreePart	subst	subtractIfCan	tower
trueEqual	unit?	unitCanonical	unitNormal
zero?	zeroOf	zerosOf	?*?
?**?	?+?	-?	?-?
?/?	?<?	?<=?	?=?
?>?	?>=?	?^?	?~=?
?*?	?**?	?quo?	?rem?

— domain IAN InnerAlgebraicNumber —

```

)abbrev domain IAN InnerAlgebraicNumber
++ Author: Manuel Bronstein
++ Date Created: 22 March 1988
++ Date Last Updated: 4 October 1995 (JHD)
++ Description:
  
```

```

++ Algebraic closure of the rational numbers.

InnerAlgebraicNumber(): Exports == Implementation where
  Z ==> Integer
  FE ==> Expression Z
  K ==> Kernel %
  P ==> SparseMultivariatePolynomial(Z, K)
  ALGOP ==> "%alg"
  SUP ==> SparseUnivariatePolynomial

Exports ==> Join(ExpressionSpace, AlgebraicallyClosedField,
  RetractableTo Z, RetractableTo Fraction Z,
  LinearlyExplicitRingOver Z, RealConstant,
  LinearlyExplicitRingOver Fraction Z,
  CharacteristicZero,
  ConvertibleTo Complex Float, DifferentialRing) with
coerce : P -> %
  ++ coerce(p) returns p viewed as an algebraic number.
numer : % -> P
  ++ numer(f) returns the numerator of f viewed as a
  ++ polynomial in the kernels over Z.
denom : % -> P
  ++ denom(f) returns the denominator of f viewed as a
  ++ polynomial in the kernels over Z.
reduce : % -> %
  ++ reduce(f) simplifies all the unreduced algebraic numbers
  ++ present in f by applying their defining relations.
trueEqual : (%,% ) -> Boolean
  ++ trueEqual(x,y) tries to determine if the two numbers are equal
norm : (SUP(%),Kernel %) -> SUP(%)
  ++ norm(p,k) computes the norm of the polynomial p
  ++ with respect to the extension generated by kernel k
norm : (SUP(%),List Kernel %) -> SUP(%)
  ++ norm(p,l) computes the norm of the polynomial p
  ++ with respect to the extension generated by kernels l
norm : (% ,Kernel %) -> %
  ++ norm(f,k) computes the norm of the algebraic number f
  ++ with respect to the extension generated by kernel k
norm : (% ,List Kernel %) -> %
  ++ norm(f,l) computes the norm of the algebraic number f
  ++ with respect to the extension generated by kernels l
Implementation ==> FE add

Rep := FE

-- private
mainRatDenom(f:%):% ==
  ratDenom(f::Rep::FE)$AlgebraicManipulations(Integer, FE)::Rep::%
--      mv:= mainVariable denom f
--      mv case "failed" => f

```



```

--      algv:=mv::K
--      q:=univariate(f, algv, minPoly(algv))_
--      $PolynomialCategoryQuotientFunctions(IndexedExponents K,K,Integer,P,%)
--      q(algv::%)

findDenominator(z:SUP %):Record(num:SUP %,den:%) ==
  zz:=z
  while not(zz=0) repeat
    dd:=(denom leadingCoefficient zz)::%
    not(dd=1) =>
      rec:=findDenominator(dd*z)
      return [rec.num,rec.den*dd]
    zz:=reductum zz
  [z,1]
makeUnivariate(p:P,k:Kernel %):SUP % ==
  map(x+>x::%,univariate(p,k))$SparseUnivariatePolynomialFunctions2(P,%)
-- public
a,b:%
differentiate(x::%):% == 0
zero? a == zero? numer a
-- one? a == one? numer a and one? denom a
one? a == (numer a = 1) and (denom a = 1)
x:% / y:%      == mainRatDenom(x /$Rep y)
x:% ** n:Integer ==
  n < 0 => mainRatDenom (x **$Rep n)
  x **$Rep n
trueEqual(a,b) ==
  -- if two algebraic numbers have the same norm (after deleting repeated
  -- roots, then they are certainly conjugates. Note that we start with a
  -- monic polynomial, so don't have to check for constant factors.
  -- this will be fooled by sqrt(2) and -sqrt(2), but the = in
  -- AlgebraicNumber knows what to do about this.
  ka:=reverse tower a
  kb:=reverse tower b
  empty? ka and empty? kb => retract(a)@Fraction Z = retract(b)@Fraction Z
  pa,pb:$SparseUnivariatePolynomial %
  pa:=monomial(1,1)-monomial(a,0)
  pb:=monomial(1,1)-monomial(b,0)
  na:=map(retract,norm(pa,ka))_
  $SparseUnivariatePolynomialFunctions2(%,Fraction Z)
  nb:=map(retract,norm(pb,kb))_
  $SparseUnivariatePolynomialFunctions2(%,Fraction Z)
  (sa:=squareFreePart(na)) = (sb:=squareFreePart(nb)) => true
  g:=gcd(sa,sb)
  (dg:=degree g) = 0 => false
  -- of course, if these have a factor in common, then the
  -- answer is really ambiguous, so we ought to be using Duval-type
  -- technology
  dg = degree sa or dg = degree sb => true
  false

```

```

norm(z:%,k:Kernel %): % ==
  p:=minPoly k
  n:=makeUnivariate( numer z,k)
  d:=makeUnivariate( denom z,k)
  resultant(n,p)/resultant(d,p)
norm(z:%,l:List Kernel %): % ==
  for k in l repeat
    z:=norm(z,k)
  z
norm(z:SUP %,k:Kernel %):SUP % ==
  p:=map(x +-> x::SUP %,minPoly k)_
  $SparseUnivariatePolynomialFunctions2(% ,SUP %)
  f:=findDenominator z
  zz:=map(x +-> makeUnivariate( numer x,k),f.num)_
  $SparseUnivariatePolynomialFunctions2( % ,SUP %)
  zz:=swap(zz)$CommuteUnivariatePolynomialCategory(% ,SUP %,SUP SUP %)
  resultant(p,zz)/norm(f.den,k)
norm(z:SUP %,l:List Kernel %): SUP % ==
  for k in l repeat
    z:=norm(z,k)
  z
belong? op == belong?(op)$ExpressionSpace_&%(%) or has?(op, ALGOP)

convert(x:%):Float ==
  retract map(y +-> y::Float, x pretend FE)$ExpressionFunctions2(Z,Float)

convert(x:%):DoubleFloat ==
  retract map(y +-> y::DoubleFloat,x pretend FE)_
  $ExpressionFunctions2(Z, DoubleFloat)

convert(x:%):Complex(Float) ==
  retract map(y +-> y::Complex(Float),x pretend FE)_
  $ExpressionFunctions2(Z, Complex Float)

```

— IAN.dotabb —

```

"IAN" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IAN"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"IAN" -> "ACF"
"IAN" -> "FS"
"IAN" -> "COMPCAT"

```

domain IFF InnerFiniteField

— InnerFiniteField.input —

```

)set break resume
)sys rm -f InnerFiniteField.output
)spool InnerFiniteField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerFiniteField
--R InnerFiniteField(p: PositiveInteger,n: PositiveInteger) is a domain constructor
--R Abbreviation for InnerFiniteField is IFF
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for IFF
--R
--R----- Operations -----
--R ?? : (Fraction(Integer),%) -> %      ?? : (%,Fraction(Integer)) -> %
--R ?? : (%,%) -> %                      ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %        ??? : (%,Integer) -> %
--R ??? : (%,PositiveInteger) -> %      ?+? : (%,%) -> %
--R ?-? : (%,%) -> %                    -? : % -> %
--R ?/? : (%,%) -> %                    ?=? : (%,%) -> Boolean
--R 1 : () -> %                         0 : () -> %
--R ?? : (%,Integer) -> %               ?? : (%,PositiveInteger) -> %
--R algebraic? : % -> Boolean           associates? : (%,%) -> Boolean
--R basis : () -> Vector(%)             coerce : InnerPrimeField(p) -> %
--R coerce : Fraction(Integer) -> %     coerce : % -> %
--R coerce : Integer -> %               coerce : % -> OutputForm
--R degree : % -> PositiveInteger        dimension : () -> CardinalNumber
--R factor : % -> Factored(%)            gcd : List(%) -> %
--R gcd : (%,%) -> %                    hash : % -> SingleInteger
--R inGroundField? : % -> Boolean        inv : % -> %
--R latex : % -> String                  lcm : List(%) -> %
--R lcm : (%,%) -> %                    norm : % -> InnerPrimeField(p)
--R one? : % -> Boolean                  prime? : % -> Boolean
--R ?quo? : (%,%) -> %                   recip : % -> Union(%, "failed")
--R ?rem? : (%,%) -> %                   retract : % -> InnerPrimeField(p)
--R sample : () -> %                     sizeLess? : (%,%) -> Boolean
--R squareFree : % -> Factored(%)        squareFreePart : % -> %
--R trace : % -> InnerPrimeField(p)      transcendent? : % -> Boolean
--R unit? : % -> Boolean                  unitCanonical : % -> %
--R zero? : % -> Boolean                  ?~=? : (%,%) -> Boolean
--R ?? : (InnerPrimeField(p),%) -> %
--R ?? : (%,InnerPrimeField(p)) -> %
--R ?? : (NonNegativeInteger,%) -> %

```

```

--R ??? : (% , NonNegativeInteger) -> %
--R ?/? : (% , InnerPrimeField(p)) -> %
--R D : (% , NonNegativeInteger) -> % if InnerPrimeField(p) has FINITE
--R D : % -> % if InnerPrimeField(p) has FINITE
--R Frobenius : (% , NonNegativeInteger) -> % if InnerPrimeField(p) has FINITE
--R Frobenius : % -> % if InnerPrimeField(p) has FINITE
--R ?? : (% , NonNegativeInteger) -> %
--R basis : PositiveInteger -> Vector(%)
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if InnerPrimeField(p) has CHARNZ or InnerPrimeField(p) has FINITE
--R charthRoot : % -> % if InnerPrimeField(p) has FINITE
--R conditionP : Matrix(%) -> Union(Vector(%, "failed") if InnerPrimeField(p) has FINITE
--R coordinates : Vector(%) -> Matrix(InnerPrimeField(p))
--R coordinates : % -> Vector(InnerPrimeField(p))
--R createNormalElement : () -> % if InnerPrimeField(p) has FINITE
--R createPrimitiveElement : () -> % if InnerPrimeField(p) has FINITE
--R definingPolynomial : () -> SparseUnivariatePolynomial(InnerPrimeField(p))
--R degree : % -> OnePointCompletion(PositiveInteger)
--R differentiate : (% , NonNegativeInteger) -> % if InnerPrimeField(p) has FINITE
--R differentiate : % -> % if InnerPrimeField(p) has FINITE
--R discreteLog : (% , %) -> Union(NonNegativeInteger, "failed") if InnerPrimeField(p) has CHARNZ or InnerPrimeField(p) has FINITE
--R discreteLog : % -> NonNegativeInteger if InnerPrimeField(p) has FINITE
--R divide : (% , %) -> Record(quotient: %, remainder: %)
--R enumerate : () -> List(%) if InnerPrimeField(p) has FINITE
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(%), %) -> Union(List(%), "failed")
--R exquo : (% , %) -> Union(%, "failed")
--R extendedEuclidean : (% , %, %) -> Union(Record(coef1: %, coef2: %), "failed")
--R extendedEuclidean : (% , %) -> Record(coef1: %, coef2: %, generator: %)
--R extensionDegree : () -> PositiveInteger
--R extensionDegree : () -> OnePointCompletion(PositiveInteger)
--R factorsOfCyclicGroupSize : () -> List(Record(factor: Integer, exponent: Integer)) if InnerPrimeField(p) has FINITE
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(InnerPrimeField(p))
--R generator : () -> % if InnerPrimeField(p) has FINITE
--R index : PositiveInteger -> % if InnerPrimeField(p) has FINITE
--R init : () -> % if InnerPrimeField(p) has FINITE
--R linearAssociatedExp : (% , SparseUnivariatePolynomial(InnerPrimeField(p))) -> % if InnerPrimeField(p) has FINITE
--R linearAssociatedLog : (% , %) -> Union(SparseUnivariatePolynomial(InnerPrimeField(p)), "failed") if InnerPrimeField(p) has FINITE
--R linearAssociatedLog : % -> SparseUnivariatePolynomial(InnerPrimeField(p)) if InnerPrimeField(p) has FINITE
--R linearAssociatedOrder : % -> SparseUnivariatePolynomial(InnerPrimeField(p)) if InnerPrimeField(p) has FINITE
--R lookup : % -> PositiveInteger if InnerPrimeField(p) has FINITE
--R minimalPolynomial : (% , PositiveInteger) -> SparseUnivariatePolynomial(InnerPrimeField(p)) if InnerPrimeField(p) has FINITE
--R minimalPolynomial : % -> SparseUnivariatePolynomial(InnerPrimeField(p))
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed")
--R nextItem : % -> Union(%, "failed") if InnerPrimeField(p) has FINITE
--R norm : (% , PositiveInteger) -> % if InnerPrimeField(p) has FINITE
--R normal? : % -> Boolean if InnerPrimeField(p) has FINITE
--R normalElement : () -> % if InnerPrimeField(p) has FINITE
--R order : % -> OnePointCompletion(PositiveInteger) if InnerPrimeField(p) has CHARNZ or InnerPrimeField(p) has FINITE
--R order : % -> PositiveInteger if InnerPrimeField(p) has FINITE

```

```

--R primeFrobenius : % -> % if InnerPrimeField(p) has CHARNZ or InnerPrimeField(p) has FINITE
--R primeFrobenius : (% , NonNegativeInteger) -> % if InnerPrimeField(p) has CHARNZ or InnerPrimeField(p) has FINITE
--R primitive? : % -> Boolean if InnerPrimeField(p) has FINITE
--R primitiveElement : () -> % if InnerPrimeField(p) has FINITE
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %)
--R random : () -> % if InnerPrimeField(p) has FINITE
--R representationType : () -> Union("prime", polynomial, normal, cyclic) if InnerPrimeField(p) has FINITE
--R represents : Vector(InnerPrimeField(p)) -> %
--R retractIfCan : % -> Union(InnerPrimeField(p), "failed")
--R size : () -> NonNegativeInteger if InnerPrimeField(p) has FINITE
--R subtractIfCan : (% , %) -> Union(% , "failed")
--R tableForDiscreteLogarithm : Integer -> Table(PositiveInteger, NonNegativeInteger) if InnerPrimeField(p) has FINITE
--R trace : (% , PositiveInteger) -> % if InnerPrimeField(p) has FINITE
--R transcendenceDegree : () -> NonNegativeInteger
--R unitNormal : % -> Record(unit: % , canonical: % , associate: %)
--R
--E 1

)spool
)lisp (bye)

```

— InnerFiniteField.help —

```

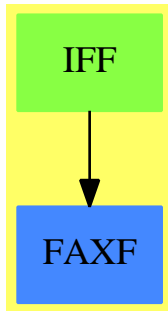
=====
InnerFiniteField examples
=====

```

InnerFiniteField(p,n) implements finite fields with p^{*n} elements
 where p is assumed prime but does not check.
 For a version which checks that p is prime, see FiniteField.

See Also:

- o)show InnerFiniteField
- o)show FiniteField

InnerFiniteField (IFF)**See**

- ⇒ “FiniteFieldExtensionByPolynomial” (FFP) [7](#) on page [850](#)
- ⇒ “FiniteFieldExtension” (FFX) [7](#) on page [845](#)
- ⇒ “FiniteField” (FF) [7](#) on page [819](#)

Exports:

0	1	algebraic?
associates?	basis	characteristic
charthRoot	coerce	conditionP
coordinates	createNormalElement	createPrimitiveElement
D	definingPolynomial	degree
dimension	differentiate	discreteLog
divide	euclideanSize	expressIdealMember
exquo	extendedEuclidean	extensionDegree
factor	factorsOfCyclicGroupSize	Frobenius
gcd	gcdPolynomial	generator
hash	index	inGroundField?
init	inv	latex
lcm	linearAssociatedExp	linearAssociatedLog
linearAssociatedOrder	lookup	minimalPolynomial
multiEuclidean	nextItem	norm
normal?	normalElement	one?
order	prime?	primeFrobenius
primitive?	primitiveElement	principalIdeal
random	recip	representationType
represents	retract	retractIfCan
sample	size	sizeLess?
squareFree	squareFreePart	subtractIfCan
tableForDiscreteLogarithm	trace	transcendenceDegree
transcendent?	unit?	unitCanonical
unitNormal	zero?	?*?
?**?	?+?	?-?
-?	?/?	?=?
?^?	?~=?	?quo?
?rem?		

— domain IFF InnerFiniteField —

```

)abbrev domain IFF InnerFiniteField
++ Author: Mark Botch
++ Date Last Updated: 29 May 1990
++ Reference:
++ R.Lidl, H.Niederreiter: Finite Field, Encyclopedia of Mathematics and
++ Its Applications, Vol. 20, Cambridge Univ. Press, 1983, ISBN 0 521 30240 4
++ J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++ AXIOM Technical Report Series, ATR/5 NP2522.
++ Description:
++ InnerFiniteField(p,n) implements finite fields with \spad{p**n} elements
++ where p is assumed prime but does not check.
++ For a version which checks that p is prime, see \spadtype{FiniteField}.

InnerFiniteField(p:PositiveInteger, n:PositiveInteger) ==

```

```
FiniteFieldExtension(InnerPrimeField p, n)
```

— IFF.dotabb —

```
"IFF" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IFF"]
"FAXF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAXF"]
"IFF" -> "FAXF"
```

domain IFAMON InnerFreeAbelianMonoid

— InnerFreeAbelianMonoid.input —

```
)set break resume
)sys rm -f InnerFreeAbelianMonoid.output
)spool InnerFreeAbelianMonoid.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerFreeAbelianMonoid
--R
--R InnerFreeAbelianMonoid(S: SetCategory,E: CancellationAbelianMonoid,un: E) is a domain constructor
--R Abbreviation for InnerFreeAbelianMonoid is IFAMON
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for IFAMON
--R
--R----- Operations -----
--R ?? : (E,S) -> %           ?? : (PositiveInteger,%) -> %
--R ?? : (S,%) -> %           ?+? : (%,%) -> %
--R ?? : (%,%) -> Boolean     0 : () -> %
--R coefficient : (S,%) -> E   coerce : S -> %
--R coerce : % -> OutputForm   hash : % -> SingleInteger
--R latex : % -> String        mapCoef : ((E -> E),%) -> %
--R mapGen : ((S -> S),%) -> % nthCoef : (%,Integer) -> E
--R nthFactor : (%,Integer) -> S retract : % -> S
--R sample : () -> %           size : % -> NonNegativeInteger
--R zero? : % -> Boolean       ?~=? : (%,%) -> Boolean
--R ?? : (NonNegativeInteger,%) -> %
--R highCommonTerms : (%,%) -> % if E has OAMON
```



```

--R retractIfCan : % -> Union(S,"failed")
--R subtractIfCan : (%,% ) -> Union(%,"failed")
--R terms : % -> List(Record(gen: S,exp: E))
--R
--E 1

)spool
)lisp (bye)

```

— InnerFreeAbelianMonoid.help —

```

=====
InnerFreeAbelianMonoid examples
=====

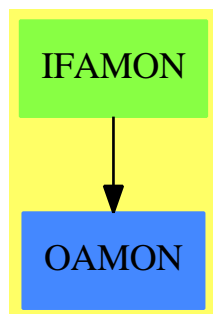
```

Internal implementation of a free abelian monoid on any set of generators

See Also:

o)show InnerFreeAbelianMonoid

InnerFreeAbelianMonoid (IFAMON)



See

- ⇒ “ListMonoidOps” (LMOPS) [13](#) on page [1509](#)
- ⇒ “FreeMonoid” (FMONOID) [7](#) on page [1021](#)
- ⇒ “FreeGroup” (FGROUP) [7](#) on page [1010](#)
- ⇒ “FreeAbelianMonoid” (FAMONOID) [7](#) on page [1007](#)
- ⇒ “FreeAbelianGroup” (FAGROUP) [7](#) on page [1004](#)

Exports:

0	coefficient	coerce	hash	highCommonTerms
latex	mapCoef	mapGen	nthCoef	nthFactor
retract	retractIfCan	sample	size	subtractIfCan
terms	zero?	?~=?	?*?	?+?
?=?				

— domain IFAMON InnerFreeAbelianMonoid —

```

)abbrev domain IFAMON InnerFreeAbelianMonoid
++ Author: Manuel Bronstein
++ Date Created: November 1989
++ Date Last Updated: 6 June 1991
++ Description:
++ Internal implementation of a free abelian monoid on any set of generators

InnerFreeAbelianMonoid(S: SetCategory, E: CancellationAbelianMonoid, un:E):
  FreeAbelianMonoidCategory(S, E) == ListMonoidOps(S, E, un) add
    Rep := ListMonoidOps(S, E, un)

    0 == makeUnit()
    zero? f == empty? listOfMonoms f
    terms f == copy listOfMonoms f
    nthCoef(f, i) == nthExpon(f, i)
    nthFactor(f, i) == nthFactor(f, i)$Rep
    s:S + f:$ == plus(s, un, f)
    f:$ + g:$ == plus(f, g)
    (f:$ = g:$):Boolean == commutativeEquality(f,g)
    n:E * s:S == makeTerm(s, n)
    n:NonNegativeInteger * f:$ == mapExpon(x +-> n*x, f)
    coerce(f:$):OutputForm == outputForm(f, "+", (x,y) +-> y*x, 0)
    mapCoef(f, x) == mapExpon(f, x)
    mapGen(f, x) == mapGen(f, x)$Rep

    coefficient(s, f) ==
      for x in terms f repeat
        x.gen = s => return(x.exp)
      0

    if E has OrderedAbelianMonoid then
      highCommonTerms(f, g) ==
        makeMulti [[x.gen, min(x.exp, n)] for x in listOfMonoms f |
          (n := coefficient(x.gen, g)) > 0]

```

— IFAMON.dotabb —

```
"IFAMON" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IFAMON"]
"OAMON" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAMON"]
"IFAMON" -> "OAMON"
```

domain IIARRAY2 InnerIndexedTwoDimensionalArray

This is an internal type which provides an implementation of 2-dimensional arrays as PrimitiveArray's of PrimitiveArray's.

— InnerIndexedTwoDimensionalArray.input —

```
)set break resume
)sys rm -f InnerIndexedTwoDimensionalArray.output
)spool InnerIndexedTwoDimensionalArray.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerIndexedTwoDimensionalArray
--R
--R InnerIndexedTwoDimensionalArray(R: Type,mnRow: Integer,mnCol: Integer,Row: FiniteLinearAggregate)
--R Abbreviation for InnerIndexedTwoDimensionalArray is IIARRAY2
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for IIARRAY2
--R
--R----- Operations -----
--R column : (%,Integer) -> Col          copy : % -> %
--R elt : (%,Integer,Integer,R) -> R      elt : (%,Integer,Integer) -> R
--R empty : () -> %                      empty? : % -> Boolean
--R eq? : (%,% ) -> Boolean              fill! : (% ,R) -> %
--R map : (((R,R) -> R),% ,%,R) -> %      map : (((R,R) -> R),% ,%) -> %
--R map : ((R -> R),%) -> %              map! : ((R -> R),%) -> %
--R maxColIndex : % -> Integer            maxRowIndex : % -> Integer
--R minColIndex : % -> Integer            minRowIndex : % -> Integer
--R ncols : % -> NonNegativeInteger       nrows : % -> NonNegativeInteger
--R parts : % -> List(R)                  qelt : (% ,Integer,Integer) -> R
--R row : (% ,Integer) -> Row              sample : () -> %
--R setRow! : (% ,Integer,Row) -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?=? : (% ,%) -> Boolean if R has SETCAT
--R any? : ((R -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if R has SETCAT
--R count : (R,% ) -> NonNegativeInteger if $ has finiteAggregate and R has SETCAT
```

```

--R count : ((R -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R eval : (%,List(R),List(R)) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (%,R,R) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (%,Equation(R)) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (%,List(Equation(R))) -> % if R has EVALAB(R) and R has SETCAT
--R every? : ((R -> Boolean),%) -> Boolean if $ has finiteAggregate
--R hash : % -> SingleInteger if R has SETCAT
--R latex : % -> String if R has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R member? : (R,%) -> Boolean if $ has finiteAggregate and R has SETCAT
--R members : % -> List(R) if $ has finiteAggregate
--R more? : (%,NonNegativeInteger) -> Boolean
--R new : (NonNegativeInteger,NonNegativeInteger,R) -> %
--R qsetelt! : (%,Integer,Integer,R) -> R
--R setColumn! : (%,Integer,Col) -> %
--R setelt : (%,Integer,Integer,R) -> R
--R size? : (%,NonNegativeInteger) -> Boolean
--R ?~=? : (%,%) -> Boolean if R has SETCAT
--R
--E 1

```

```

)spool
)lisp (bye)

```

— InnerIndexedTwoDimensionalArray.help —

```

=====
InnerIndexedTwoDimensionalArray examples
=====

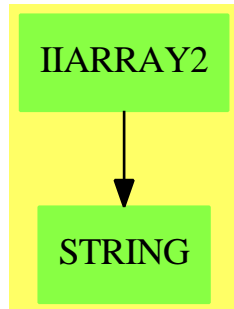
```

There is no description for this domain

See Also:

o)show InnerIndexedTwoDimensionalArray

InnerIndexedTwoDimensionalArray (IIARRAY2)



See

⇒ “IndexedTwoDimensionalArray” (IARRAY2) [10](#) on page [1256](#)

⇒ “TwoDimensionalArray” (ARRAY2) [21](#) on page [2763](#)

Exports:

any?	coerce	column	copy	count
elt	empty	empty?	eq?	eval
every?	fill!	hash	latex	less?
map	map!	maxColIndex	maxRowIndex	member?
members	minColIndex	minRowIndex	more?	ncols
new	nrows	parts	qelt	qsetelt!
row	sample	setColumn!	setelt	setRow!
size?	#?	?=?	?~=?	

— domain IIARRAY2 InnerIndexedTwoDimensionalArray —

```

)abbrev domain IIARRAY2 InnerIndexedTwoDimensionalArray
++ Author: Mark Botch
++ Description:
++ There is no description for this domain

InnerIndexedTwoDimensionalArray(R,mnRow,mnCol,Row,Col):_
  Exports == Implementation where
    R : Type
    mnRow, mnCol : Integer
    Row : FiniteLinearAggregate R
    Col : FiniteLinearAggregate R

  Exports ==> TwoDimensionalArrayCategory(R,Row,Col)

  Implementation ==> add

    Rep := PrimitiveArray PrimitiveArray R

--% Predicates

```

```

    empty? m == empty?(m)$Rep

--% Primitive array creation

    empty() == empty()$Rep

    new(rows,cols,a) ==
        rows = 0 =>
            error "new: arrays with zero rows are not supported"
        cols = 0 =>
--            error "new: arrays with zero columns are not supported"
        arr : PrimitiveArray PrimitiveArray R := new(rows,empty())
        for i in minIndex(arr)..maxIndex(arr) repeat
            qsetelt_!(arr,i,new(cols,a))
        arr

--% Size inquiries

    minRowIndex m == mnRow
    minColIndex m == mnCol
    maxRowIndex m == nrow m + mnRow - 1
    maxColIndex m == ncol m + mnCol - 1

    nrow m == (# m)$Rep

    ncol m ==
        empty? m => 0
        # m(minIndex(m)$Rep)

--% Part selection/assignment

    qelt(m,i,j) ==
        qelt(qelt(m,i - minRowIndex m)$Rep,j - minColIndex m)

    elt(m:%,i:Integer,j:Integer) ==
        i < minRowIndex(m) or i > maxRowIndex(m) =>
            error "elt: index out of range"
        j < minColIndex(m) or j > maxColIndex(m) =>
            error "elt: index out of range"
        qelt(m,i,j)

    qsetelt_!(m,i,j,r) ==
        setelt(qelt(m,i - minRowIndex m)$Rep,j - minColIndex m,r)

    setelt(m:%,i:Integer,j:Integer,r:R) ==
        i < minRowIndex(m) or i > maxRowIndex(m) =>
            error "setelt: index out of range"
        j < minColIndex(m) or j > maxColIndex(m) =>
            error "setelt: index out of range"

```

```

qsetelt_!(m,i,j,r)

if R has SetCategory then
  latex(m : %) : String ==
    s : String := "\left[ \begin{array}{l}"
    j : Integer
    for j in minColIndex(m)..maxColIndex(m) repeat
      s := concat(s,"c")$String
    s := concat(s,"} ")$String
    i : Integer
    for i in minRowIndex(m)..maxRowIndex(m) repeat
      for j in minColIndex(m)..maxColIndex(m) repeat
        s := concat(s, latex(qelt(m,i,j))$R)$String
        if j < maxColIndex(m) then s := concat(s, " & ")$String
        if i < maxRowIndex(m) then s := concat(s, " \\ ")$String
      concat(s, "\end{array} \right]")$String

-----

— IIARRAY2.dotabb —

"IIARRAY2" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IIARRAY2"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"IIARRAY2" -> "STRING"

```

domain IPADIC InnerPAdicInteger

```

— InnerPAdicInteger.input —

)set break resume
)sys rm -f InnerPAdicInteger.output
)spool InnerPAdicInteger.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerPAdicInteger
--R
--R InnerPAdicInteger(p: Integer,unBalanced?: Boolean) is a domain constructor
--R Abbreviation for InnerPAdicInteger is IPADIC
--R This constructor is not exposed in this frame.

```

```
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for IPADIC
```

```
--R
```

```
--R----- Operations -----
```

```
--R ?? : (% , %) -> %           ?? : (Integer , %) -> %
--R ?? : (PositiveInteger , %) -> %   ??? : (% , PositiveInteger) -> %
--R +? : (% , %) -> %           ?-? : (% , %) -> %
--R -? : % -> %                 ?=? : (% , %) -> Boolean
--R 1 : () -> %                 0 : () -> %
--R ?? : (% , PositiveInteger) -> %   associates? : (% , %) -> Boolean
--R coerce : % -> %               coerce : Integer -> %
--R coerce : % -> OutputForm       complete : % -> %
--R digits : % -> Stream(Integer)   extend : (% , Integer) -> %
--R gcd : List(%) -> %             gcd : (% , %) -> %
--R hash : % -> SingleInteger       latex : % -> String
--R lcm : List(%) -> %             lcm : (% , %) -> %
--R moduloP : % -> Integer          modulus : () -> Integer
--R one? : % -> Boolean            order : % -> NonNegativeInteger
--R ?quo? : (% , %) -> %           quotientByP : % -> %
--R recip : % -> Union(%, "failed") ?rem? : (% , %) -> %
--R sample : () -> %              sizeLess? : (% , %) -> Boolean
--R sqrt : (% , Integer) -> %       unit? : % -> Boolean
--R unitCanonical : % -> %          zero? : % -> Boolean
--R ?~=? : (% , %) -> Boolean
--R ?? : (NonNegativeInteger , %) -> %
--R ??? : (% , NonNegativeInteger) -> %
--R ?? : (% , NonNegativeInteger) -> %
--R approximate : (% , Integer) -> Integer
--R characteristic : () -> NonNegativeInteger
--R divide : (% , %) -> Record(quotient: %, remainder: %)
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(%), %) -> Union(List(%), "failed")
--R exquo : (% , %) -> Union(%, "failed")
--R extendedEuclidean : (% , %, %) -> Union(Record(coef1: %, coef2: %), "failed")
--R extendedEuclidean : (% , %) -> Record(coef1: %, coef2: %, generator: %)
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed")
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %)
--R root : (SparseUnivariatePolynomial(Integer), Integer) -> %
--R subtractIfCan : (% , %) -> Union(%, "failed")
--R unitNormal : % -> Record(unit: %, canonical: %, associate: %)
--R
--E 1
```

```
)spool
```

```
)lisp (bye)
```



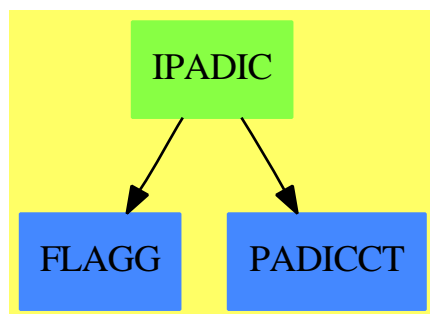
```
=====
InnerPAdicInteger examples
=====
```

This domain implements \mathbb{Z}_p , the p-adic completion of the integers.
This is an internal domain.

See Also:

```
o )show InnerPAdicInteger
```

InnerPAdicInteger (IPADIC)



See

- ⇒ “PAdicInteger” (PADIC) [17](#) on page [1881](#)
- ⇒ “BalancedPAdicInteger” (BPADIC) [3](#) on page [258](#)
- ⇒ “PAdicRationalConstructor” (PADICRC) [17](#) on page [1890](#)
- ⇒ “PAdicRational” (PADICRAT) [17](#) on page [1885](#)
- ⇒ “BalancedPAdicRational” (BPADICRT) [3](#) on page [262](#)

Exports:

0	1	approximate	associates?
characteristic	coerce	complete	digits
divide	euclideanSize	expressIdealMember	exquo
extend	extendedEuclidean	gcd	gcdPolynomial
hash	latex	lcm	multiEuclidean
moduloP	modulus	one?	order
principalIdeal	quotientByP	recip	root
sample	sizeLess?	sqrt	subtractIfCan
unit?	unitCanonical	unitNormal	zero?
?^=?	?*?	?**?	?^?
?+?	?-?	-?	?=?
?quo?	?rem?		

— domain IPADIC InnerPAdicInteger —

```

)abbrev domain IPADIC InnerPAdicInteger
++ Author: Clifton J. Williamson
++ Date Created: 20 August 1989
++ Date Last Updated: 15 May 1990
++ Description:
++ This domain implements  $\mathbb{Z}_p$ , the p-adic completion of the integers.
++ This is an internal domain.

InnerPAdicInteger(p,unBalanced?): Exports == Implementation where
  p          : Integer
  unBalanced? : Boolean
  I  ==> Integer
  NNI ==> NonNegativeInteger
  OUT ==> OutputForm
  L  ==> List
  ST ==> Stream
  SUP ==> SparseUnivariatePolynomial

Exports ==> PAdicIntegerCategory p

Implementation ==> add

  PEXPR := p :: OUT

  Rep := ST I

  characteristic() == 0
  euclideanSize(x) == order(x)

  stream(x:ST I):ST I == x pretend ST(I)
  padic(x:ST I):% == x pretend %
  digits x == stream x

  extend(x,n) == extend(x,n + 1)$Rep
  complete x == complete(x)$Rep

--      notBalanced?:() -> Boolean
--      notBalanced?() == unBalanced?

modP:I -> I
modP n ==
  unBalanced? or (p = 2) => positiveRemainder(n,p)
  symmetricRemainder(n,p)

modPInfo:I -> Record(digit:I,carry:I)
modPInfo n ==
  dv := divide(n,p)

```

```

r0 := dv.remainder; q := dv.quotient
if (r := modP r0) ^= r0 then q := q + ((r0 - r) quo p)
[r,q]

invModP: I -> I
invModP n == invmod(n,p)

modulus()      == p
moduloP x      == (empty? x => 0; frst x)
quotientByP x == (empty? x => x; rst x)

approximate(x,n) ==
  n <= 0 or empty? x => 0
  frst x + p * approximate(rst x,n - 1)

x = y ==
  st : ST I := stream(x - y)
  n : I := _$streamCount$Lisp
  for i in 0..n repeat
    empty? st => return true
    frst st ^= 0 => return false
    st := rst st
  empty? st

order x ==
  st := stream x
  for i in 0..1000 repeat
    empty? st => return 0
    frst st ^= 0 => return i
    st := rst st
  error "order: series has more than 1000 leading zero coeffs"

0 == padic concat(0$I,empty())
1 == padic concat(1$I,empty())

intToPAAdic: I -> ST I
intToPAAdic n == delay
  n = 0 => empty()
  modp := modPInfo n
  concat(modp.digit,intToPAAdic modp.carry)

intPlusPAAdic: (I,ST I) -> ST I
intPlusPAAdic(n,x) == delay
  empty? x => intToPAAdic n
  modp := modPInfo(n + frst x)
  concat(modp.digit,intPlusPAAdic(modp.carry,rst x))

intMinusPAAdic: (I,ST I) -> ST I
intMinusPAAdic(n,x) == delay
  empty? x => intToPAAdic n

```

```

modp := modPInfo(n - frst x)
concat(modp.digit,intMinusPAdic(modp.carry,rst x))

plusAux: (I,ST I,ST I) -> ST I
plusAux(n,x,y) == delay
  empty? x and empty? y => intToPAdic n
  empty? x => intPlusPAdic(n,y)
  empty? y => intPlusPAdic(n,x)
  modp := modPInfo(n + frst x + frst y)
  concat(modp.digit,plusAux(modp.carry,rst x,rst y))

minusAux: (I,ST I,ST I) -> ST I
minusAux(n,x,y) == delay
  empty? x and empty? y => intToPAdic n
  empty? x => intMinusPAdic(n,y)
  empty? y => intPlusPAdic(n,x)
  modp := modPInfo(n + frst x - frst y)
  concat(modp.digit,minusAux(modp.carry,rst x,rst y))

x + y == padic plusAux(0,stream x,stream y)
x - y == padic minusAux(0,stream x,stream y)
- y == padic intMinusPAdic(0,stream y)
coerce(n:I) == padic intToPAdic n

intMult: (I,ST I) -> ST I
intMult(n,x) == delay
  empty? x => empty()
  modp := modPInfo(n * frst x)
  concat(modp.digit,intPlusPAdic(modp.carry,intMult(n,rst x)))

(n:I) * (x:%) ==
  padic intMult(n,stream x)

timesAux: (ST I,ST I) -> ST I
timesAux(x,y) == delay
  empty? x or empty? y => empty()
  modp := modPInfo(frst x * frst y)
  car := modp.digit
  cdr : ST I --!!
  cdr := plusAux(modp.carry,intMult(frst x,rst y),timesAux(rst x,y))
  concat(car,cdr)

(x:%) * (y:%) == padic timesAux(stream x,stream y)

quotientAux: (ST I,ST I) -> ST I
quotientAux(x,y) == delay
  empty? x => error "quotientAux: first argument"
  empty? y => empty()
  modP frst x = 0 =>
    modP frst y = 0 => quotientAux(rst x,rst y)

```

```

    error "quotient: quotient not integral"
  z0 := modP(invModP first x * first y)
  yy : ST I --!!
  yy := rest minusAux(0,y,intMult(z0,x))
  concat(z0,quotientAux(x,yy))

recip x ==
  empty? x or modP first x = 0 => "failed"
  padic quotientAux(stream x,concat(1,empty()))

iExquo: (%,%,I) -> Union(%, "failed")
iExquo(xx,yy,n) ==
  n > 1000 =>
    error "exquo: quotient by series with many leading zero coefs"
  empty? yy => "failed"
  empty? xx => 0
  zero? first yy =>
    zero? first xx => iExquo(rst xx,rst yy,n + 1)
    "failed"
  (rec := recip yy) case "failed" => "failed"
  xx * (rec :: %)

x exquo y == iExquo(stream x,stream y,0)

divide(x,y) ==
  (z:=x exquo y) case "failed" => [0,x]
  [z, 0]

iSqrt: (I,I,I,%) -> %
iSqrt(pn,an,bn,bSt) == delay
  bn1 := (empty? bSt => bn; bn + pn * first(bSt))
  c := (bn1 - an*an) quo pn
  aa := modP(c * invmod(2*an,p))
  nSt := (empty? bSt => bSt; rst bSt)
  concat(aa,iSqrt(pn*p,an + pn*aa,bn1,nSt))

sqrt(b,a) ==
  p = 2 =>
    error "sqrt: no square roots in Z2 yet"
  not zero? modP(a*a - (bb := moduloP b)) =>
    error "sqrt: not a square root (mod p)"
  b := (empty? b => b; rst b)
  a := modP a
  concat(a,iSqrt(p,a,bb,b))

iRoot: (SUP I,I,I,I) -> ST I
iRoot(f,alpha,invFpx0,pPow) == delay
  num := -((f(alpha) exquo pPow) :: I)
  digit := modP(num * invFpx0)
  concat(digit,iRoot(f,alpha + digit * pPow,invFpx0,p * pPow))

```

```

root(f,x0) ==
  x0 := modP x0
  not zero? modP f(x0) =>
    error "root: not a root (mod p)"
  fpx0 := modP (differentiate f)(x0)
  zero? fpx0 =>
    error "root: approximate root must be a simple root (mod p)"
  invFpx0 := modP invModP fpx0
  padic concat(x0,iRoot(f,x0,invFpx0,p))

termOutput:(I,I) -> OUT
termOutput(k,c) ==
  k = 0 => c :: OUT
  mon := (k = 1 => PEXPR; PEXPR ** (k :: OUT))
  c = 1 => mon
  c = -1 => -mon
  (c :: OUT) * mon

showAll?:() -> Boolean
-- check a global Lisp variable
showAll?() == true

coerce(x:%):OUT ==
  empty?(st := stream x) => 0 :: OUT
  n : NNI ; count : NNI := _$streamCount$Lisp
  l : L OUT := empty()
  for n in 0..count while not empty? st repeat
    if first(st) ^= 0 then
      l := concat(termOutput(n :: I,first st),l)
      st := rst st
  if showAll?() then
    for n in (count + 1).. while explicitEntries? st and _
      not eq?(st,rst st) repeat
      if first(st) ^= 0 then
        l := concat(termOutput(n pretend I,first st),l)
        st := rst st
  l :=
    explicitlyEmpty? st => l
    eq?(st,rst st) and first st = 0 => l
    concat(prefix("0" :: OUT,[PEXPR ** (n :: OUT)]),l)
  empty? l => 0 :: OUT
  reduce("+",reverse_! l)

```

— IPADIC.dotabb —

```

"IPADIC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IPADIC"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PADICCT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PADICCT"]
"IPADIC" -> "PADICCT"
"IPADIC" -> "FLAGG"

```

domain IPF InnerPrimeField

— InnerPrimeField.input —

```

)set break resume
)sys rm -f InnerPrimeField.output
)spool InnerPrimeField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerPrimeField
--R InnerPrimeField(p: PositiveInteger) is a domain constructor
--R Abbreviation for InnerPrimeField is IPF
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for IPF
--R
--R----- Operations -----
--R ?? : (Fraction(Integer),%) -> %      ?? : (%,Fraction(Integer)) -> %
--R ?? : (%,%) -> %                      ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %        *** : (%,Integer) -> %
--R *** : (%,PositiveInteger) -> %      ?+? : (%,%) -> %
--R ?-? : (%,%) -> %                    -? : % -> %
--R ?/? : (%,%) -> %                    ?=? : (%,%) -> Boolean
--R D : % -> %                          D : (%,NonNegativeInteger) -> %
--R 1 : () -> %                          0 : () -> %
--R ?? : (%,Integer) -> %                ?? : (%,PositiveInteger) -> %
--R algebraic? : % -> Boolean             associates? : (%,%) -> Boolean
--R basis : () -> Vector(%)               charthRoot : % -> %
--R coerce : Fraction(Integer) -> %        coerce : % -> %
--R coerce : Integer -> %                 coerce : % -> OutputForm
--R convert : % -> Integer                 coordinates : % -> Vector(%)
--R createPrimitiveElement : () -> %        degree : % -> PositiveInteger
--R differentiate : % -> %                 dimension : () -> CardinalNumber
--R enumerate : () -> List(%)              factor : % -> Factored(%)
--R gcd : List(%) -> %                     gcd : (%,%) -> %
--R hash : % -> SingleInteger              inGroundField? : % -> Boolean

```

```

--R index : PositiveInteger -> %
--R inv : % -> %
--R lcm : List(%) -> %
--R lookup : % -> PositiveInteger
--R one? : % -> Boolean
--R prime? : % -> Boolean
--R primitive? : % -> Boolean
--R ?quo? : (%,%) -> %
--R recip : % -> Union(%, "failed")
--R represents : Vector(%) -> %
--R sample : () -> %
--R sizeLess? : (%,%) -> Boolean
--R squareFreePart : % -> %
--R transcendental? : % -> Boolean
--R unitCanonical : % -> %
--R ?~=?: (%,%) -> Boolean
--R ?*?: (NonNegativeInteger,%) -> %
--R ?**?: (%,NonNegativeInteger) -> %
--R Frobenius : % -> % if $ has FINITE
--R Frobenius : (%,NonNegativeInteger) -> % if $ has FINITE
--R ?^?: (%,NonNegativeInteger) -> %
--R basis : PositiveInteger -> Vector(%)
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed")
--R conditionP : Matrix(%) -> Union(Vector(%), "failed")
--R coordinates : Vector(%) -> Matrix(%)
--R createNormalElement : () -> % if $ has FINITE
--R definingPolynomial : () -> SparseUnivariatePolynomial(%)
--R degree : % -> OnePointCompletion(PositiveInteger)
--R differentiate : (%,NonNegativeInteger) -> %
--R discreteLog : % -> NonNegativeInteger
--R discreteLog : (%,%) -> Union(NonNegativeInteger, "failed")
--R divide : (%,%) -> Record(quotient: %, remainder: %)
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(%),%) -> Union(List(%), "failed")
--R exquo : (%,%) -> Union(%, "failed")
--R extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %), "failed")
--R extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %)
--R extensionDegree : () -> OnePointCompletion(PositiveInteger)
--R extensionDegree : () -> PositiveInteger
--R factorsOfCyclicGroupSize : () -> List(Record(factor: Integer,exponent: Integer))
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R generator : () -> % if $ has FINITE
--R linearAssociatedExp : (%,SparseUnivariatePolynomial(%)) -> % if $ has FINITE
--R linearAssociatedLog : % -> SparseUnivariatePolynomial(%) if $ has FINITE
--R linearAssociatedLog : (%,%) -> Union(SparseUnivariatePolynomial(%), "failed") if $ has FINITE
--R linearAssociatedOrder : % -> SparseUnivariatePolynomial(%) if $ has FINITE
--R minimalPolynomial : % -> SparseUnivariatePolynomial(%)
--R minimalPolynomial : (%,PositiveInteger) -> SparseUnivariatePolynomial(%) if $ has FINITE
--R multiEuclidean : (List(%),%) -> Union(List(%), "failed")

```



```

--R nextItem : % -> Union(%, "failed")
--R norm : (%, PositiveInteger) -> % if $ has FINITE
--R normal? : % -> Boolean if $ has FINITE
--R normalElement : () -> % if $ has FINITE
--R order : % -> OnePointCompletion(PositiveInteger)
--R primeFrobenius : (%, NonNegativeInteger) -> %
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %)
--R representationType : () -> Union("prime", polynomial, normal, cyclic)
--R retractIfCan : % -> Union(%, "failed")
--R subtractIfCan : (%, %) -> Union(%, "failed")
--R tableForDiscreteLogarithm : Integer -> Table(PositiveInteger, NonNegativeInteger)
--R trace : (%, PositiveInteger) -> % if $ has FINITE
--R transcendenceDegree : () -> NonNegativeInteger
--R unitNormal : % -> Record(unit: %, canonical: %, associate: %)
--R
--E 1

)spool
)lisp (bye)

```

— InnerPrimeField.help —

=====

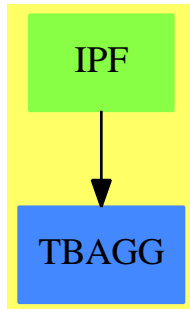
InnerPrimeField examples

=====

InnerPrimeField(p) implements the field with p elements.
 Note: argument p MUST be a prime (this domain does not check).
 See PrimeField for a domain that does check.

See Also:

- o)show InnerPrimeField
- o)show PrimeField

InnerPrimeField (IPF)**See**⇒ “PrimeField” (PF) [17](#) on page [2104](#)**Exports:**

0	1	algebraic?
associates?	basis	characteristic
charthRoot	coerce	conditionP
convert	coordinates	createPrimitiveElement
createNormalElement	D	definingPolynomial
degree	differentiate	dimension
discreteLog	divide	euclideanSize
expressIdealMember	exquo	extendedEuclidean
extensionDegree	factor	factorsOfCyclicGroupSize
Frobenius	gcd	gcdPolynomial
generator	hash	inGroundField?
index	init	inv
latex	lcm	linearAssociatedExp
linearAssociatedLog	linearAssociatedOrder	lookup
minimalPolynomial	multiEuclidean	nextItem
norm	normal?	normalElement
one?	order	prime?
primeFrobenius	primitive?	primitiveElement
principalIdeal	random	recip
representationType	represents	retract
retractIfCan	sample	size
sizeLess?	squareFree	squareFreePart
subtractIfCan	tableForDiscreteLogarithm	trace
transcendenceDegree	transcendent?	unit?
unitCanonical	unitNormal	zero?
?*?	?**?	?+?
?-?	-?	?/?
?=?	?^?	?~=?
?quo?	?rem?	

— domain IPF InnerPrimeField —

```

)abbrev domain IPF InnerPrimeField
++ Authors: N.N., J.Grabmeier, A.Scheerhorn
++ Date Created: ?, November 1990, 26.03.1991
++ Date Last Updated: 12 April 1991
++ References:
++ R.Lidl, H.Niederreiter: Finite Field, Encycloedia of Mathematics and
++ Its Applications, Vol. 20, Cambridge Univ. Press, 1983, ISBN 0 521 30240 4
++ AXIOM Technical Report Series, to appear.
++ Description:
++ InnerPrimeField(p) implements the field with p elements.
++ Note: argument p MUST be a prime (this domain does not check).
++ See \spadtype{PrimeField} for a domain that does check.

InnerPrimeField(p:PositiveInteger): Exports == Implementation where

  I ==> Integer
  NNI ==> NonNegativeInteger
  PI ==> PositiveInteger
  TBL ==> Table(PI,NNI)
  R ==> Record(key:PI,entry:NNI)
  SUP ==> SparseUnivariatePolynomial
  OUT ==> OutputForm

Exports ==> Join(FiniteFieldCategory,FiniteAlgebraicExtensionField($),_
  ConvertibleTo(Integer))

Implementation ==> IntegerMod p add

  initializeElt:() -> Void
  initializeLog:() -> Void

-- global variables =====

primitiveElt:PI:=1
-- for the lookup the primitive Element computed by createPrimitiveElement()

sizeCG :=(p-1) pretend NonNegativeInteger
-- the size of the cyclic group

facOfGroupSize := nil()$(List Record(factor:Integer,exponent:Integer))
-- the factorization of the cyclic group size

initlog?:Boolean:=true
-- gets false after initialization of the logarithm table

initelt?:Boolean:=true
-- gets false after initialization of the primitive Element

```

```

discLogTable:Table(PI,TBL):=table()$Table(PI,TBL)
-- tables indexed by the factors of the size q of the cyclic group
-- discLogTable.factor is a table of with keys
-- primitiveElement() ** (i * (q quo factor)) and entries i for
-- i in 0..n-1, n computed in initialize() in order to use
-- the minimal size limit 'limit' optimal.

-- functions =====

generator() == 1

-- This uses x**(p-1)=1 (mod p), so x**(q(p-1)+r) = x**r (mod p)
x:$ ** n:Integer ==
  zero?(n) => 1
  zero?(x) => 0
  r := positiveRemainder(n,p-1)::NNI
  ((x pretend IntegerMod p) **$IntegerMod(p) r) pretend $

if p <= convert(max())$SingleInteger@Integer then
  q := p::SingleInteger

  recip x ==
    zero?(y := convert(x)@Integer :: SingleInteger) => "failed"
    invmod(y, q)::Integer::$
else
  recip x ==
    zero?(y := convert(x)@Integer) => "failed"
    invmod(y, p)::Integer::$

convert(x:$) == x pretend I

normalElement() == 1

createNormalElement() == 1

characteristic() == p

factorsOfCyclicGroupSize() ==
  p=2 => facOfGroupSize -- this fixes an infinite loop of functions
                        -- calls, problem was that factors factor(1)
                        -- is the empty list
  if empty? facOfGroupSize then initializeElt()
  facOfGroupSize

representationType() == "prime"

tableForDiscreteLogarithm(fac) ==
  if initlog? then initializeLog()

```

```

tbl:=search(fac::PI, discLogTable)$Table(PI, TBL)
tbl case "failed" =>
    error "tableForDiscreteLogarithm: argument must be prime divisor_
of the order of the multiplicative group"
tbl pretend TBL

primitiveElement() ==
    if initelt? then initializeElt()
    index(primitiveElt)

initializeElt() ==
    facOfGroupSize:=factors(factor(sizeCG)$I)$(Factored I)
    -- get a primitive element
    primitiveElt:=lookup(createPrimitiveElement())
    -- set initialization flag
    initelt? := false
    void$Void

initializeLog() ==
    if initelt? then initializeElt()
    -- set up tables for discrete logarithm
    limit:Integer:=30
    -- the minimum size for the discrete logarithm table
    for f in facOfGroupSize repeat
        fac:=f.factor
        base:=$:=primitiveElement() ** (sizeCG quo fac)
        l:Integer:=length(fac)$Integer
        n:Integer:=0
        if odd?(l)$Integer then n:=shift(fac, -(l quo 2))
            else n:=shift(1, (l quo 2))

        if n < limit then
            d:=(fac-1) quo limit + 1
            n:=(fac-1) quo d + 1
            tbl:TBL:=table()$TBL
            a:=$:=1
            for i in (0::NNI)..(n-1)::NNI repeat
                insert_!([lookup(a), i::NNI]$R, tbl)$TBL
                a:=a*base
            insert_!([fac::PI, copy(tbl)$TBL]_
                $Record(key:PI, entry:TBL), discLogTable)$Table(PI, TBL)
    -- tell user about initialization
    -- print("discrete logarithm table initialized"::OUT)
    -- set initialization flag
    initlog? := false
    void$Void

degree(x):PI == 1::PositiveInteger
extensionDegree():PI == 1::PositiveInteger

-- sizeOfGroundField() == p::NonNegativeInteger

```

```

inGroundField?(x) == true

coordinates(x) == new(1,x)$(Vector $)

represents(v) == v.1

retract(x) == x

retractIfCan(x) == x

basis() == new(1,1::)$(Vector $)
basis(n:PI) ==
  n = 1 => basis()
  error("basis: argument must divide extension degree")

definingPolynomial() ==
  monomial(1,1)$(SUP $) - monomial(1,0)$(SUP $)

minimalPolynomial(x) ==
  monomial(1,1)$(SUP $) - monomial(x,0)$(SUP $)

charthRoot x == x

```

— IPF.dotabb —

```

"IPF" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IPF"]
"TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]
"IPF" -> "TBAGG"

```

domain ISUPS InnerSparseUnivariatePowerSeries

— InnerSparseUnivariatePowerSeries.input —

```

)set break resume
)sys rm -f InnerSparseUnivariatePowerSeries.output
)spool InnerSparseUnivariatePowerSeries.output
)set message test on
)set message auto off

```

```
)clear all
```

```
--S 1 of 1
```

```
)show InnerSparseUnivariatePowerSeries
```

```
--R
```

```
--R InnerSparseUnivariatePowerSeries(Coef: Ring) is a domain constructor
```

```
--R Abbreviation for InnerSparseUnivariatePowerSeries is ISUPS
```

```
--R This constructor is not exposed in this frame.
```

```
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ISUPS
```

```
--R
```

```
--R----- Operations -----
```

```
--R ?? : (Coef,%) -> %
```

```
?? : (%,Coef) -> %
```

```
--R ?? : (%,%) -> %
```

```
?? : (Integer,%) -> %
```

```
--R ?? : (PositiveInteger,%) -> %
```

```
?? : (%,PositiveInteger) -> %
```

```
--R ?? : (%,%) -> %
```

```
?-? : (%,%) -> %
```

```
--R -? : % -> %
```

```
?=? : (%,%) -> Boolean
```

```
--R 1 : () -> %
```

```
0 : () -> %
```

```
--R ?? : (%,PositiveInteger) -> %
```

```
center : % -> Coef
```

```
--R coefficient : (%,Integer) -> Coef
```

```
coerce : Integer -> %
```

```
--R coerce : % -> OutputForm
```

```
complete : % -> %
```

```
--R degree : % -> Integer
```

```
?..? : (%,Integer) -> Coef
```

```
--R extend : (%,Integer) -> %
```

```
hash : % -> SingleInteger
```

```
--R iCompose : (%,%) -> %
```

```
latex : % -> String
```

```
--R leadingCoefficient : % -> Coef
```

```
leadingMonomial : % -> %
```

```
--R map : ((Coef -> Coef),%) -> %
```

```
monomial : (Coef,Integer) -> %
```

```
--R monomial? : % -> Boolean
```

```
one? : % -> Boolean
```

```
--R order : (%,Integer) -> Integer
```

```
order : % -> Integer
```

```
--R pole? : % -> Boolean
```

```
recip : % -> Union(%, "failed")
```

```
--R reductum : % -> %
```

```
sample : () -> %
```

```
--R taylorQuoByVar : % -> %
```

```
truncate : (%,Integer) -> %
```

```
--R variable : % -> Symbol
```

```
zero? : % -> Boolean
```

```
--R ~=? : (%,%) -> Boolean
```

```
--R ?? : (%,Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
```

```
--R ?? : (Fraction(Integer),%) -> % if Coef has ALGEBRA(FRAC(INT))
```

```
--R ?? : (NonNegativeInteger,%) -> %
```

```
--R ?? : (%,NonNegativeInteger) -> %
```

```
--R ?? : (%,Coef) -> % if Coef has FIELD
```

```
--R D : % -> % if Coef has *: (Integer,Coef) -> Coef
```

```
--R D : (%,NonNegativeInteger) -> % if Coef has *: (Integer,Coef) -> Coef
```

```
--R D : (%,Symbol) -> % if Coef has *: (Integer,Coef) -> Coef and Coef has PDRING(SYMBOL)
```

```
--R D : (%,List(Symbol)) -> % if Coef has *: (Integer,Coef) -> Coef and Coef has PDRING(SYMBOL)
```

```
--R D : (%,Symbol,NonNegativeInteger) -> % if Coef has *: (Integer,Coef) -> Coef and Coef has PD
```

```
--R D : (%,List(Symbol),List(NonNegativeInteger)) -> % if Coef has *: (Integer,Coef) -> Coef and
```

```
--R ?? : (%,NonNegativeInteger) -> %
```

```
--R approximate : (%,Integer) -> Coef if Coef has **: (Coef,Integer) -> Coef and Coef has coerce
```

```
--R associates? : (%,%) -> Boolean if Coef has INTDOM
```

```
--R cAcos : % -> % if Coef has ALGEBRA(FRAC(INT))
```

```
--R cAcosh : % -> % if Coef has ALGEBRA(FRAC(INT))
```

```
--R cAcot : % -> % if Coef has ALGEBRA(FRAC(INT))
```

```
--R cAcoth : % -> % if Coef has ALGEBRA(FRAC(INT))
```

```

--R cAcsc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cAcsch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cAsec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cAsech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cAsin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cAsinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cAtan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cAtanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cCos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cCosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cCot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cCoth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cCsc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cCsch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cExp : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cLog : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cPower : (% ,Coef) -> % if Coef has ALGEBRA(FRAC(INT))
--R cRationalPower : (% ,Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R cSec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cSech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cSin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cSinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cTan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cTanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%,"failed") if Coef has CHARNZ
--R coerce : Coef -> % if Coef has COMRING
--R coerce : % -> % if Coef has INTDOM
--R coerce : Fraction(Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R differentiate : % -> % if Coef has *: (Integer,Coef) -> Coef
--R differentiate : (% ,NonNegativeInteger) -> % if Coef has *: (Integer,Coef) -> Coef
--R differentiate : (% ,Symbol) -> % if Coef has *: (Integer,Coef) -> Coef and Coef has PDRING(SYMBOL)
--R differentiate : (% ,List(Symbol)) -> % if Coef has *: (Integer,Coef) -> Coef and Coef has PDRING(SYMBOL)
--R differentiate : (% ,Symbol,NonNegativeInteger) -> % if Coef has *: (Integer,Coef) -> Coef and Coef has P
--R differentiate : (% ,List(Symbol),List(NonNegativeInteger)) -> % if Coef has *: (Integer,Coef) -> Coef and
--R ?.? : (% ,%) -> % if Integer has SGROUP
--R eval : (% ,Coef) -> Stream(Coef) if Coef has **: (Coef,Integer) -> Coef
--R exquo : (% ,%) -> Union(%,"failed") if Coef has INTDOM
--R getRef : % -> Reference(OrderedCompletion(Integer))
--R getStream : % -> Stream(Record(k: Integer,c: Coef))
--R iExquo : (% ,%,Boolean) -> Union(%,"failed")
--R integrate : % -> % if Coef has ALGEBRA(FRAC(INT))
--R makeSeries : (Reference(OrderedCompletion(Integer)),Stream(Record(k: Integer,c: Coef))) -> %
--R monomial : (% ,List(SingletonAsOrderedSet),List(Integer)) -> %
--R monomial : (% ,SingletonAsOrderedSet,Integer) -> %
--R multiplyCoefficients : ((Integer -> Coef),%) -> %
--R multiplyExponents : (% ,PositiveInteger) -> %
--R series : Stream(Record(k: Integer,c: Coef)) -> %
--R seriesToOutputForm : (Stream(Record(k: Integer,c: Coef)),Reference(OrderedCompletion(Integer)),Symbol,C
--R subtractIfCan : (% ,%) -> Union(%,"failed")

```



```

--R terms : % -> Stream(Record(k: Integer,c: Coef))
--R truncate : (%,Integer,Integer) -> %
--R unit? : % -> Boolean if Coef has INTDOM
--R unitCanonical : % -> % if Coef has INTDOM
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %) if Coef has INTDOM
--R variables : % -> List(SingletonAsOrderedSet)
--R
--E 1

)spool
)lisp (bye)

```

— InnerSparseUnivariatePowerSeries.help —

```

=====
InnerSparseUnivariatePowerSeries examples
=====

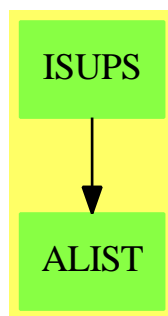
```

InnerSparseUnivariatePowerSeries is an internal domain used for creating sparse Taylor and Laurent series.

See Also:

o)show InnerSparseUnivariatePowerSeries

InnerSparseUnivariatePowerSeries (ISUPS)



Exports:

0	1	approximate	associates?
cAcos	cAcosh	cAcot	cAcoth
cAcsc	cAcsch	cAsec	cAsech
cAsin	cAsinh	cAtan	cAtanh
cCos	cCosh	cCot	cCoth
cCsc	cCsch	center	cExp
cLog	coefficient	cPower	cRationalPower
cSec	cSech	cSin	cSinh
cTan	cTanh	characteristic	charthRoot
coerce	complete	D	differentiate
degree	eval	exquo	extend
getRef	getStream	hash	iCompose
iExquo	integrate	latex	leadingCoefficient
leadingMonomial	makeSeries	map	monomial
monomial?	multiplyCoefficients	multiplyExponents	one?
order	pole?	recip	reductum
sample	series	seriesToOutputForm	subtractIfCan
taylorQuoByVar	terms	truncate	unit?
unitCanonical	unitNormal	variable	variables
zero?	?*?	?**?	?+?
?-?	-?	?=?	?^?
??	?~=?	?/?	?^?
??			

— domain ISUPS InnerSparseUnivariatePowerSeries —

```
)abbrev domain ISUPS InnerSparseUnivariatePowerSeries
```

```
++ Author: Clifton J. Williamson
```

```
++ Date Created: 28 October 1994
```

```
++ Date Last Updated: 9 March 1995
```

```
++ Description:
```

```
++ InnerSparseUnivariatePowerSeries is an internal domain
```

```
++ used for creating sparse Taylor and Laurent series.
```

```
InnerSparseUnivariatePowerSeries(Coef): Exports == Implementation where
```

```
Coef : Ring
```

```
B ==> Boolean
```

```
COM ==> OrderedCompletion Integer
```

```
I ==> Integer
```

```
L ==> List
```

```
NNI ==> NonNegativeInteger
```

```
OUT ==> OutputForm
```

```
PI ==> PositiveInteger
```

```
REF ==> Reference OrderedCompletion Integer
```

```
RN ==> Fraction Integer
```

```
Term ==> Record(k:Integer,c:Coef)
```

```
SG ==> String
```

ST ==> Stream Term

```
Exports ==> UnivariatePowerSeriesCategory(Coef,Integer) with
makeSeries: (REF,ST) -> %
  ++ makeSeries(refer,str) creates a power series from the reference
  ++ \spad{refer} and the stream \spad{str}.
getRef: % -> REF
  ++ getRef(f) returns a reference containing the order to which the
  ++ terms of f have been computed.
getStream: % -> ST
  ++ getStream(f) returns the stream of terms representing the series f.
series: ST -> %
  ++ series(st) creates a series from a stream of non-zero terms,
  ++ where a term is an exponent-coefficient pair. The terms in the
  ++ stream should be ordered by increasing order of exponents.
monomial?: % -> B
  ++ monomial?(f) tests if f is a single monomial.
multiplyCoefficients: (I -> Coef,%) -> %
  ++ multiplyCoefficients(fn,f) returns the series
  ++ \spad{sum(fn(n) * an * x^n,n = n0..)},
  ++ where f is the series \spad{sum(an * x^n,n = n0..)}.
iExquo: (%,% ,B) -> Union(%,"failed")
  ++ iExquo(f,g,taylor?) is the quotient of the power series f and g.
  ++ If \spad{taylor?} is \spad{true}, then we must have
  ++ \spad{order(f) >= order(g)}.
taylorQuoByVar: % -> %
  ++ taylorQuoByVar(a0 + a1 x + a2 x**2 + ...)
  ++ returns \spad{a1 + a2 x + a3 x**2 + ...}
iCompose: (%,% ) -> %
  ++ iCompose(f,g) returns \spad{f(g(x))}. This is an internal function
  ++ which should only be called for Taylor series \spad{f(x)} and
  ++ \spad{g(x)} such that the constant coefficient of \spad{g(x)} is zero.
seriesToOutputForm: (ST,REF,Symbol,Coef,RN) -> OutputForm
  ++ seriesToOutputForm(st,refer,var,cen,r) prints the series
  ++ \spad{f((var - cen)^r)}.
if Coef has Algebra Fraction Integer then
integrate: % -> %
  ++ integrate(f(x)) returns an anti-derivative of the power series
  ++ \spad{f(x)} with constant coefficient 0.
  ++ Warning: function does not check for a term of degree -1.
cPower: (% ,Coef) -> %
  ++ cPower(f,r) computes \spad{f^r}, where f has constant coefficient 1.
  ++ For use when the coefficient ring is commutative.
cRationalPower: (% ,RN) -> %
  ++ cRationalPower(f,r) computes \spad{f^r}.
  ++ For use when the coefficient ring is commutative.
cExp: % -> %
  ++ cExp(f) computes the exponential of the power series f.
  ++ For use when the coefficient ring is commutative.
cLog: % -> %
```

```

    ++ cLog(f) computes the logarithm of the power series f.
    ++ For use when the coefficient ring is commutative.
cSin: % -> %
    ++ cSin(f) computes the sine of the power series f.
    ++ For use when the coefficient ring is commutative.
cCos: % -> %
    ++ cCos(f) computes the cosine of the power series f.
    ++ For use when the coefficient ring is commutative.
cTan: % -> %
    ++ cTan(f) computes the tangent of the power series f.
    ++ For use when the coefficient ring is commutative.
cCot: % -> %
    ++ cCot(f) computes the cotangent of the power series f.
    ++ For use when the coefficient ring is commutative.
cSec: % -> %
    ++ cSec(f) computes the secant of the power series f.
    ++ For use when the coefficient ring is commutative.
cCsc: % -> %
    ++ cCsc(f) computes the cosecant of the power series f.
    ++ For use when the coefficient ring is commutative.
cAsin: % -> %
    ++ cAsin(f) computes the arcsine of the power series f.
    ++ For use when the coefficient ring is commutative.
cAcos: % -> %
    ++ cAcos(f) computes the arccosine of the power series f.
    ++ For use when the coefficient ring is commutative.
cAtan: % -> %
    ++ cAtan(f) computes the arctangent of the power series f.
    ++ For use when the coefficient ring is commutative.
cAcot: % -> %
    ++ cAcot(f) computes the arccotangent of the power series f.
    ++ For use when the coefficient ring is commutative.
cAsec: % -> %
    ++ cAsec(f) computes the arcsecant of the power series f.
    ++ For use when the coefficient ring is commutative.
cAcsc: % -> %
    ++ cAcsc(f) computes the arccosecant of the power series f.
    ++ For use when the coefficient ring is commutative.
cSinh: % -> %
    ++ cSinh(f) computes the hyperbolic sine of the power series f.
    ++ For use when the coefficient ring is commutative.
cCosh: % -> %
    ++ cCosh(f) computes the hyperbolic cosine of the power series f.
    ++ For use when the coefficient ring is commutative.
cTanh: % -> %
    ++ cTanh(f) computes the hyperbolic tangent of the power series f.
    ++ For use when the coefficient ring is commutative.
cCoth: % -> %
    ++ cCoth(f) computes the hyperbolic cotangent of the power series f.
    ++ For use when the coefficient ring is commutative.

```

```

cSech: % -> %
  ++ cSech(f) computes the hyperbolic secant of the power series f.
  ++ For use when the coefficient ring is commutative.
cSch: % -> %
  ++ cSch(f) computes the hyperbolic cosecant of the power series f.
  ++ For use when the coefficient ring is commutative.
cAsinh: % -> %
  ++ cAsinh(f) computes the inverse hyperbolic sine of the power
  ++ series f. For use when the coefficient ring is commutative.
cAcosh: % -> %
  ++ cAcosh(f) computes the inverse hyperbolic cosine of the power
  ++ series f. For use when the coefficient ring is commutative.
cAtanh: % -> %
  ++ cAtanh(f) computes the inverse hyperbolic tangent of the power
  ++ series f. For use when the coefficient ring is commutative.
cAcoth: % -> %
  ++ cAcoth(f) computes the inverse hyperbolic cotangent of the power
  ++ series f. For use when the coefficient ring is commutative.
cAsech: % -> %
  ++ cAsech(f) computes the inverse hyperbolic secant of the power
  ++ series f. For use when the coefficient ring is commutative.
cAcsch: % -> %
  ++ cAcsch(f) computes the inverse hyperbolic cosecant of the power
  ++ series f. For use when the coefficient ring is commutative.

Implementation ==> add
import REF

Rep := Record(%ord: REF,%str: Stream Term)
-- when the value of 'ord' is n, this indicates that all non-zero
-- terms of order up to and including n have been computed;
-- when 'ord' is plusInfinity, all terms have been computed;
-- lazy evaluation of 'str' has the side-effect of modifying the value
-- of 'ord'

--% Local functions

makeTerm:      (Integer,Coef) -> Term
getCoef:       Term -> Coef
getExpon:      Term -> Integer
iSeries:       (ST,REF) -> ST
iExtend:       (ST,COM,REF) -> ST
iTruncate0:    (ST,REF,REF,COM,I,I) -> ST
iTruncate:     (%,COM,I) -> %
iCoefficient:  (ST,Integer) -> Coef
iOrder:       (ST,COM,REF) -> I
iMap1:        ((Coef,I) -> Coef,I -> I,B,ST,REF,REF,Integer) -> ST
iMap2:        ((Coef,I) -> Coef,I -> I,B,%) -> %
iPlus1:       ((Coef,Coef) -> Coef,ST,REF,ST,REF,REF,I) -> ST
iPlus2:       ((Coef,Coef) -> Coef,%,%) -> %

```

```

productByTerm: (Coef,I,ST,REF,REF,I) -> ST
productLazyEval: (ST,REF,ST,REF,COM) -> Void
iTimes: (ST,REF,ST,REF,REF,I) -> ST
iDivide: (ST,REF,ST,REF,Coef,I,REF,I) -> ST
divide: (% ,I,% ,I,Coef) -> %
compose0: (ST,REF,ST,REF,I,% ,% ,I,REF,I) -> ST
factorials?: () -> Boolean
termOutput: (RN,Coef,OUT) -> OUT
showAll?: () -> Boolean

--% macros

makeTerm(exp,coef) == [exp,coef]
getCoef term == term.c
getExpon term == term.k

makeSeries(refer,x) == [refer,x]
getRef ups == ups.%ord
getStream ups == ups.%str

--% creation and destruction of series

monomial(coef,expon) ==
  nix : ST := empty()
  st :=
    zero? coef => nix
    concat(makeTerm(expon,coef),nix)
  makeSeries(ref plusInfinity(),st)

monomial? ups == (not empty? getStream ups) and (empty? rst getStream ups)

coerce(n:I) == n :: Coef :: %
coerce(r:Coef) == monomial(r,0)

iSeries(x,refer) ==
  empty? x => (setelt(refer,plusInfinity()); empty())
  setelt(refer,(getExpon first x) :: COM)
  concat(first x,iSeries(rst x,refer))

series(x:ST) ==
  empty? x => 0
  n := getExpon first x; refer := ref(n :: COM)
  makeSeries(refer,iSeries(x,refer))

--% values

characteristic() == characteristic()$Coef

0 == monomial(0,0)
1 == monomial(1,0)

```

```

iExtend(st,n,refer) ==
  (elt refer) < n =>
    explicitlyEmpty? st => (setelt(refer,plusInfinity()); st)
    explicitEntries? st => iExtend(rst st,n,refer)
    iExtend(lazyEvaluate st,n,refer)
  st

extend(x,n) == (iExtend(getStream x,n :: COM,getRef x); x)
complete x == (iExtend(getStream x,plusInfinity(),getRef x); x)

iTruncate0(x,xRefer,refer,minExp,maxExp,n) == delay
explicitlyEmpty? x => (setelt(refer,plusInfinity()); empty())
nn := n :: COM
while (elt xRefer) < nn repeat lazyEvaluate x
explicitEntries? x =>
  (nx := getExpon(xTerm := frst x)) > maxExp =>
    (setelt(refer,plusInfinity()); empty())
  setelt(refer,nx :: COM)
  (nx :: COM) >= minExp =>
    concat(makeTerm(nx,getCoef xTerm),_
      iTruncate0(rst x,xRefer,refer,minExp,maxExp,nx + 1))
  iTruncate0(rst x,xRefer,refer,minExp,maxExp,nx + 1)
-- can't have elt(xRefer) = infy unless all terms have been computed
degr := retract(elt xRefer)@I
setelt(refer,degr :: COM)
iTruncate0(x,xRefer,refer,minExp,maxExp,degr + 1)

iTruncate(ups,minExp,maxExp) ==
  x := getStream ups; xRefer := getRef ups
  explicitlyEmpty? x => 0
  explicitEntries? x =>
    deg := getExpon frst x
    refer := ref((deg - 1) :: COM)
    makeSeries(refer,iTruncate0(x,xRefer,refer,minExp,maxExp,deg))
  -- can't have elt(xRefer) = infy unless all terms have been computed
  degr := retract(elt xRefer)@I
  refer := ref(degr :: COM)
  makeSeries(refer,iTruncate0(x,xRefer,refer,minExp,maxExp,degr + 1))

truncate(ups,n) == iTruncate(ups,minusInfinity(),n)
truncate(ups,n1,n2) ==
  if n1 > n2 then (n1,n2) := (n2,n1)
  iTruncate(ups,n1 :: COM,n2)

iCoefficient(st,n) ==
  explicitEntries? st =>
    term := frst st
    (expon := getExpon term) > n => 0
    expon = n => getCoef term

```

```

    iCoefficient(rst st,n)
  0

coefficient(x,n) == (extend(x,n); iCoefficient(getStream x,n))
elt(x:%,n:Integer) == coefficient(x,n)

iOrder(st,n,refer) ==
  explicitlyEmpty? st => error "order: series has infinite order"
  explicitEntries? st =>
    ((r := getExpon first st) :: COM) >= n => retract(n)@Integer
  r
  -- can't have elt(xRefer) = infy unless all terms have been computed
  degr := retract(elt refer)@I
  (degr :: COM) >= n => retract(n)@Integer
  iOrder(lazyEvaluate st,n,refer)

order x == iOrder(getStream x,plusInfinity(),getRef x)
order(x,n) == iOrder(getStream x,n :: COM,getRef x)

terms x == getStream x

--% predicates

zero? ups ==
  x := getStream ups; ref := getRef ups
  whatInfinity(n := elt ref) = 1 => explicitlyEmpty? x
  count : NNI := _$streamCount$Lisp
  for i in 1..count repeat
    explicitlyEmpty? x => return true
    explicitEntries? x => return false
  lazyEvaluate x
  false

ups1 = ups2 == zero?(ups1 - ups2)

--% arithmetic

iMap1(cFcn,eFcn,check?,x,xRefer,refer,n) == delay
  -- when this function is called, all terms in 'x' of order < n have been
  -- computed and we compute the eFcn(n)th order coefficient of the result
  explicitlyEmpty? x => (setelt(refer,plusInfinity()); empty())
  -- if terms in 'x' up to order n have not been computed,
  -- apply lazy evaluation
  nn := n :: COM
  while (elt xRefer) < nn repeat lazyEvaluate x
  -- 'x' may now be empty: retest
  explicitlyEmpty? x => (setelt(refer,plusInfinity()); empty())
  -- must have nx >= n
  explicitEntries? x =>
    xCoef := getCoef(xTerm := first x); nx := getExpon xTerm

```



```

newCoef := cFcn(xCoef,nx); m := eFcn nx
setelt(refer,m :: COM)
not check? =>
    concat(makeTerm(m,newCoef),_
        iMap1(cFcn,eFcn,check?,rst x,xRefer,refer,nx + 1))
zero? newCoef => iMap1(cFcn,eFcn,check?,rst x,xRefer,refer,nx + 1)
concat(makeTerm(m,newCoef),_
    iMap1(cFcn,eFcn,check?,rst x,xRefer,refer,nx + 1))
-- can't have elt(xRefer) = infty unless all terms have been computed
degr := retract(elt xRefer)@I
setelt(refer,eFcn(degr) :: COM)
iMap1(cFcn,eFcn,check?,x,xRefer,refer,degr + 1)

iMap2(cFcn,eFcn,check?,ups) ==
-- 'eFcn' must be a strictly increasing function,
-- i.e. i < j => eFcn(i) < eFcn(j)
xRefer := getRef ups; x := getStream ups
explicitlyEmpty? x => 0
explicitEntries? x =>
    deg := getExpon frst x
    refer := ref(eFcn(deg - 1) :: COM)
    makeSeries(refer,iMap1(cFcn,eFcn,check?,x,xRefer,refer,deg))
-- can't have elt(xRefer) = infty unless all terms have been computed
degr := retract(elt xRefer)@I
refer := ref(eFcn(degr) :: COM)
makeSeries(refer,iMap1(cFcn,eFcn,check?,x,xRefer,refer,degr + 1))

map(fcn,x) == iMap2((y,n) +-> fcn(y), z +-> z, true, x)
differentiate x == iMap2((y,n) +-> n*y, z +-> z - 1, true, x)
multiplyCoefficients(f,x) == iMap2((y,n) +-> f(n)*y, z +-> z, true, x)
multiplyExponents(x,n) == iMap2((y,m) +-> y, z +-> n*z, false, x)

iPlus1(op,x,xRefer,y,yRefer,refer,n) == delay
-- when this function is called, all terms in 'x' and 'y' of order < n
-- have been computed and we are computing the nth order coefficient of
-- the result; note the 'op' is either '+' or '-'
explicitlyEmpty? x =>
    iMap1((x1,m) +-> op(0,x1), z +-> z, false, y, yRefer, refer, n)
explicitlyEmpty? y =>
    iMap1((x1,m) +-> op(x1,0), z +-> z, false, x, xRefer, refer, n)
-- if terms up to order n have not been computed,
-- apply lazy evaluation
nn := n :: COM
while (elt xRefer) < nn repeat lazyEvaluate x
while (elt yRefer) < nn repeat lazyEvaluate y
-- 'x' or 'y' may now be empty: retest
explicitlyEmpty? x =>
    iMap1((x1,m) +-> op(0,x1), z +-> z, false, y, yRefer, refer, n)
explicitlyEmpty? y =>
    iMap1((x1,m) +-> op(x1,0), z +-> z, false, x, xRefer, refer, n)

```

```

-- must have nx >= n, ny >= n
-- both x and y have explicit terms
explicitEntries?(x) and explicitEntries?(y) =>
  xCoef := getCoef(xTerm := frst x); nx := getExpon xTerm
  yCoef := getCoef(yTerm := frst y); ny := getExpon yTerm
  nx = ny =>
    setelt(refer,nx :: COM)
    zero? (coef := op(xCoef,yCoef)) =>
      iPlus1(op,rst x,xRefer,rst y,yRefer,refer,nx + 1)
    concat(makeTerm(nx,coef),_
      iPlus1(op,rst x,xRefer,rst y,yRefer,refer,nx + 1))
  nx < ny =>
    setelt(refer,nx :: COM)
    concat(makeTerm(nx,op(xCoef,0)),_
      iPlus1(op,rst x,xRefer,y,yRefer,refer,nx + 1))
    setelt(refer,ny :: COM)
    concat(makeTerm(ny,op(0,yCoef)),_
      iPlus1(op,x,xRefer,rst y,yRefer,refer,ny + 1))
-- y has no term of degree n
explicitEntries? x =>
  xCoef := getCoef(xTerm := frst x); nx := getExpon xTerm
  -- can't have elt(yRefer) = infy unless all terms have been computed
  (degr := retract(elt yRefer)@I) < nx =>
    setelt(refer,elt yRefer)
    iPlus1(op,x,xRefer,y,yRefer,refer,degr + 1)
  setelt(refer,nx :: COM)
  concat(makeTerm(nx,op(xCoef,0)),_
    iPlus1(op,rst x,xRefer,y,yRefer,refer,nx + 1))
-- x has no term of degree n
explicitEntries? y =>
  yCoef := getCoef(yTerm := frst y); ny := getExpon yTerm
  -- can't have elt(xRefer) = infy unless all terms have been computed
  (degr := retract(elt xRefer)@I) < ny =>
    setelt(refer,elt xRefer)
    iPlus1(op,x,xRefer,y,yRefer,refer,degr + 1)
  setelt(refer,ny :: COM)
  concat(makeTerm(ny,op(0,yCoef)),_
    iPlus1(op,x,xRefer,rst y,yRefer,refer,ny + 1))
-- neither x nor y has a term of degree n
setelt(refer,xyRef := min(elt xRefer,elt yRefer))
-- can't have xyRef = infy unless all terms have been computed
iPlus1(op,x,xRefer,y,yRefer,refer,retract(xyRef)@I + 1)

iPlus2(op,ups1,ups2) ==
  xRefer := getRef ups1; x := getStream ups1
  xDeg :=
    explicitlyEmpty? x => return map(z +-> op(0$Coef,z),ups2)
  explicitEntries? x => (getExpon frst x) - 1
  -- can't have elt(xRefer) = infy unless all terms have been computed
  retract(elt xRefer)@I

```

```

yRefer := getRef ups2; y := getStream ups2
yDeg :=
  explicitlyEmpty? y => return map(z +-> op(z,0$Coef),ups1)
  explicitEntries? y => (getExpon first y) - 1
  -- can't have elt(yRefer) = infty unless all terms have been computed
  retract(elt yRefer)@I
deg := min(xDeg,yDeg); refer := ref(deg :: COM)
makeSeries(refer,iPlus1(op,x,xRefer,y,yRefer,refer,deg + 1))

x + y == iPlus2((xi,yi) +-> xi + yi, x, y)
x - y == iPlus2((xi,yi) +-> xi - yi, x, y)
- y == iMap2((x,n) +-> -x, z +-> z, false, y)

-- gives correct defaults for I, NNI and PI
n:I * x:% == (zero? n => 0; map(z +-> n*z, x))
n:NNI * x:% == (zero? n => 0; map(z +-> n*z, x))
n:PI * x:% == (zero? n => 0; map(z +-> n*z, x))

productByTerm(coef,expon,x,xRefer,refer,n) ==
  iMap1((y,m) +-> coef*y, z +-> z+expon, true, x, xRefer, refer, n)

productLazyEval(x,xRefer,y,yRefer,nn) ==
  explicitlyEmpty?(x) or explicitlyEmpty?(y) => void()
  explicitEntries? x =>
    explicitEntries? y => void()
    xDeg := (getExpon first x) :: COM
    while (xDeg + elt(yRefer)) < nn repeat lazyEvaluate y
    void()
  explicitEntries? y =>
    yDeg := (getExpon first y) :: COM
    while (yDeg + elt(xRefer)) < nn repeat lazyEvaluate x
    void()
  lazyEvaluate x
  -- if x = y, then y may now have explicit entries
  if lazy? y then lazyEvaluate y
  productLazyEval(x,xRefer,y,yRefer,nn)

iTimes(x,xRefer,y,yRefer,refer,n) == delay
  -- when this function is called, we are computing the nth order
  -- coefficient of the product
  productLazyEval(x,xRefer,y,yRefer,n :: COM)
  explicitlyEmpty?(x) or explicitlyEmpty?(y) =>
    (setelt(refer,plusInfinity()); empty())
  -- must have nx + ny >= n
  explicitEntries?(x) and explicitEntries?(y) =>
    xCoef := getCoef(xTerm := first x); xExpon := getExpon xTerm
    yCoef := getCoef(yTerm := first y); yExpon := getExpon yTerm
    expon := xExpon + yExpon
    setelt(refer,expon :: COM)
    scRefer := ref(expon :: COM)

```

```

    scMult := productByTerm(xCoef,xExpon,rst y,yRefer,scRefer,yExpon + 1)
    prRefer := ref(expon :: COM)
    pr := iTimes(rst x,xRefer,y,yRefer,prRefer,expon + 1)
    sm := iPlus1((a,b) +-> a+b,scMult,scRefer,pr,prRefer,refer,expon + 1)
    zero?(coef := xCoef * yCoef) => sm
    concat(makeTerm(expon,coef),sm)
explicitEntries? x =>
    xExpon := getExpon frst x
    -- can't have elt(yRefer) = infty unless all terms have been computed
    degr := retract(elt yRefer)@I
    setelt(refer,(xExpon + degr) :: COM)
    iTimes(x,xRefer,y,yRefer,refer,xExpon + degr + 1)
explicitEntries? y =>
    yExpon := getExpon frst y
    -- can't have elt(xRefer) = infty unless all terms have been computed
    degr := retract(elt xRefer)@I
    setelt(refer,(yExpon + degr) :: COM)
    iTimes(x,xRefer,y,yRefer,refer,yExpon + degr + 1)
-- can't have elt(xRefer) = infty unless all terms have been computed
xDegr := retract(elt xRefer)@I
yDegr := retract(elt yRefer)@I
setelt(refer,(xDegr + yDegr) :: COM)
iTimes(x,xRefer,y,yRefer,refer,xDegr + yDegr + 1)

ups1:% * ups2:% ==
xRefer := getRef ups1; x := getStream ups1
xDeg :=
    explicitlyEmpty? x => return 0
    explicitEntries? x => (getExpon frst x) - 1
    -- can't have elt(xRefer) = infty unless all terms have been computed
    retract(elt xRefer)@I
yRefer := getRef ups2; y := getStream ups2
yDeg :=
    explicitlyEmpty? y => return 0
    explicitEntries? y => (getExpon frst y) - 1
    -- can't have elt(yRefer) = infty unless all terms have been computed
    retract(elt yRefer)@I
deg := xDeg + yDeg + 1; refer := ref(deg :: COM)
makeSeries(refer,iTimes(x,xRefer,y,yRefer,refer,deg + 1))

iDivide(x,xRefer,y,yRefer,rym,m,refer,n) == delay
-- when this function is called, we are computing the nth order
-- coefficient of the result
explicitlyEmpty? x => (setelt(refer,plusInfinity()); empty())
-- if terms up to order n - m have not been computed,
-- apply lazy evaluation
nm := (n + m) :: COM
while (elt xRefer) < nm repeat lazyEvaluate x
-- 'x' may now be empty: retest
explicitlyEmpty? x => (setelt(refer,plusInfinity()); empty())

```

```

-- must have nx >= n + m
explicitEntries? x =>
  newCoef := getCoef(xTerm := frst x) * rym; nx := getExpon xTerm
  prodRefer := ref(nx :: COM)
  prod := productByTerm(-newCoef,nx - m,rst y,yRefer,prodRefer,1)
  sumRefer := ref(nx :: COM)
  sum := iPlus1((a,b)+->a+b,rst x,xRefer,prod,prodRefer,sumRefer,nx + 1)
  setelt(refer,(nx - m) :: COM); term := makeTerm(nx - m,newCoef)
  concat(term,iDivide(sum,sumRefer,y,yRefer,rym,m,refer,nx - m + 1))
-- can't have elt(xRefer) = infty unless all terms have been computed
degr := retract(elt xRefer)@I
setelt(refer,(degr - m) :: COM)
iDivide(x,xRefer,y,yRefer,rym,m,refer,degr - m + 1)

divide(ups1,deg1,ups2,deg2,r) ==
  xRefer := getRef ups1; x := getStream ups1
  yRefer := getRef ups2; y := getStream ups2
  refer := ref((deg1 - deg2) :: COM)
  makeSeries(refer,iDivide(x,xRefer,y,yRefer,r,deg2,refer,deg1 - deg2 + 1))

iExquo(ups1,ups2,taylor?) ==
  xRefer := getRef ups1; x := getStream ups1
  yRefer := getRef ups2; y := getStream ups2
  n : I := 0
  -- try to find first non-zero term in y
  -- give up after 1000 lazy evaluations
  while not explicitEntries? y repeat
    explicitlyEmpty? y => return "failed"
    lazyEvaluate y
    (n := n + 1) > 1000 => return "failed"
  yCoef := getCoef(yTerm := frst y); ny := getExpon yTerm
  (ry := recip yCoef) case "failed" => "failed"
  nn := ny :: COM
  if taylor? then
    while (elt(xRefer) < nn) repeat
      explicitlyEmpty? x => return 0
      explicitEntries? x => return "failed"
      lazyEvaluate x
  -- check if ups2 is a monomial
  empty? rst y => iMap2((y1,m) +-> y1*(ry::Coef),z +->z-ny, false, ups1)
  explicitlyEmpty? x => 0
  nx :=
    explicitEntries? x =>
      ((deg := getExpon frst x) < ny) and taylor? => return "failed"
      deg - 1
    -- can't have elt(xRefer) = infty unless all terms have been computed
    retract(elt xRefer)@I
  divide(ups1,nx,ups2,ny,ry :: Coef)

taylorQuoByVar ups ==

```

```

iMap2((y,n) +-> y, z +-> z-1,false,ups - monomial(coefficient(ups,0),0))

compose0(x,xRefer,y,yRefer,y0rd,y1,yn0,n0,refer,n) == delay
-- when this function is called, we are computing the nth order
-- coefficient of the composite
explicitlyEmpty? x => (setelt(refer,plusInfinity()); empty())
-- if terms in 'x' up to order n have not been computed,
-- apply lazy evaluation
nn := n :: COM; yy0rd := y0rd :: COM
while (yy0rd * elt(xRefer)) < nn repeat lazyEvaluate x
explicitEntries? x =>
  xCoef := getCoef(xTerm := first x); n1 := getExpon xTerm
  zero? n1 =>
    setelt(refer,n1 :: COM)
    concat(makeTerm(n1,xCoef),_
      compose0(rst x,xRefer,y,yRefer,y0rd,y1,yn0,n0,refer,n1 + 1))
  yn1 := yn0 * y1 ** ((n1 - n0) :: NNI)
  z := getStream yn1; zRefer := getRef yn1
  degr := y0rd * n1; prodRefer := ref((degr - 1) :: COM)
  prod := iMap1((s,k)+->xCoef*s,m+>m,true,z,zRefer,prodRefer,degr)
  coRefer := ref((degr + y0rd - 1) :: COM)
  co := compose0(rst x,xRefer,y,yRefer,y0rd,y1,yn1,n1,coRefer,degr+y0rd)
  setelt(refer,(degr - 1) :: COM)
  iPlus1((a,b)+->a+b,prod,prodRefer,co,coRefer,refer,degr)
-- can't have elt(xRefer) = infity unless all terms have been computed
degr := y0rd * (retract(elt xRefer)@I + 1)
setelt(refer,(degr - 1) :: COM)
compose0(x,xRefer,y,yRefer,y0rd,y1,yn0,n0,refer,degr)

iCompose(ups1,ups2) ==
  x := getStream ups1; xRefer := getRef ups1
  y := getStream ups2; yRefer := getRef ups2
  -- try to compute the order of 'ups2'
  n : I := _$streamCount$Lisp
  for i in 1..n while not explicitEntries? y repeat
    explicitlyEmpty? y => coefficient(ups1,0) :: %
    lazyEvaluate y
  explicitlyEmpty? y => coefficient(ups1,0) :: %
  y0rd : I :=
    explicitEntries? y => getExpon first y
    retract(elt yRefer)@I
  compRefer := ref((-1) :: COM)
  makeSeries(compRefer,_
    compose0(x,xRefer,y,yRefer,y0rd,ups2,1,0,compRefer,0))

if Coef has Algebra Fraction Integer then

  integrate x == iMap2((y,n) +-> 1/(n+1)*y, z +-> z+1, true, x)

--% Fixed point computations

```

```

Ys ==> Y$ParadoxicalCombinatorsForStreams(Term)

integ0: (ST,REF,REF,I) -> ST
integ0(x,intRef,ansRef,n) == delay
  nLess1 := (n - 1) :: COM
  while (elt intRef) < nLess1 repeat lazyEvaluate x
  explicitlyEmpty? x => (setelt(ansRef,plusInfinity()); empty())
  explicitEntries? x =>
    xCoef := getCoef(xTerm := first x); nx := getExpon xTerm
    setelt(ansRef,(n1 := (nx + 1)) :: COM)
    concat(makeTerm(n1,inv(n1 :: RN) * xCoef),_
      integ0(rst x,intRef,ansRef,n1))
  -- can't have elt(intRef) = infy unless all terms have been computed
  degr := retract(elt intRef)@I; setelt(ansRef,(degr + 1) :: COM)
  integ0(x,intRef,ansRef,degr + 2)

integ1: (ST,REF,REF) -> ST
integ1(x,intRef,ansRef) == integ0(x,intRef,ansRef,1)

lazyInteg: (Coef,() -> ST,REF,REF) -> ST
lazyInteg(a,xf,intRef,ansRef) ==
  ansStr : ST := integ1(delay xf,intRef,ansRef)
  concat(makeTerm(0,a),ansStr)

cPower(f,r) ==
  -- computes f^r. f should have constant coefficient 1.
  fp := differentiate f
  fInv := iExquo(1,f,false) :: %; y := r * fp * fInv
  yRef := getRef y; yStr := getStream y
  intRef := ref((-1) :: COM); ansRef := ref(0 :: COM)
  ansStr :=
    Ys(s+>lazyInteg(1,iTimes(s,ansRef,yStr,yRef,intRef,0),intRef,ansRef))
  makeSeries(ansRef,ansStr)

iExp: (% ,Coef) -> %
iExp(f,cc) ==
  -- computes exp(f). cc = exp coefficient(f,0)
  fp := differentiate f
  fpRef := getRef fp; fpStr := getStream fp
  intRef := ref((-1) :: COM); ansRef := ref(0 :: COM)
  ansStr :=
    Ys(s+>lazyInteg(cc,
      iTimes(s,ansRef,fpStr,fpRef,intRef,0),intRef,ansRef))
  makeSeries(ansRef,ansStr)

sincos0: (Coef,Coef,L ST,REF,REF,ST,REF,ST,REF) -> L ST
sincos0(sinc,cosc,list,sinRef,cosRef,fpStr,fpRef,fpStr2,fpRef2) ==
  sinStr := first list; cosStr := second list
  prodRef1 := ref((-1) :: COM); prodRef2 := ref((-1) :: COM)

```

```

    prodStr1 := iTimes(cosStr,cosRef,fpStr,fpRef,prodRef1,0)
    prodStr2 := iTimes(sinStr,sinRef,fpStr2,fpRef2,prodRef2,0)
    [lazyInteg(sinc,prodStr1,prodRef1,sinRef),_
     lazyInteg(cosc,prodStr2,prodRef2,cosRef)]

iSincos: (%,Coef,Coef,I) -> Record(%sin: %, %cos: %)
iSincos(f,sinc,cosc,sign) ==
  fp := differentiate f
  fpRef := getRef fp; fpStr := getStream fp
--   fp2 := (one? sign => fp; -fp)
  fp2 := ((sign = 1) => fp; -fp)
  fpRef2 := getRef fp2; fpStr2 := getStream fp2
  sinRef := ref(0 :: COM); cosRef := ref(0 :: COM)
  sincos :=
    Ys(s+>sincos0(sinc,cosc,s,sinRef,cosRef,fpStr,fpRef,fpStr2,fpRef2),2)
  sinStr := (zero? sinc => rst first sincos; first sincos)
  cosStr := (zero? cosc => rst second sincos; second sincos)
  [makeSeries(sinRef,sinStr),makeSeries(cosRef,cosStr)]

tan0: (Coef,ST,REF,ST,REF,I) -> ST
tan0(cc,ansStr,ansRef,fpStr,fpRef,sign) ==
  sqRef := ref((-1) :: COM)
  sqStr := iTimes(ansStr,ansRef,ansStr,ansRef,sqRef,0)
  one : % := 1; oneStr := getStream one; oneRef := getRef one
  yRef := ref((-1) :: COM)
  yStr : ST :=
--   one? sign => iPlus1(#1 + #2,oneStr,oneRef,sqStr,sqRef,yRef,0)
    (sign = 1) => iPlus1((a,b)+->a+b,oneStr,oneRef,sqStr,sqRef,yRef,0)
    iPlus1((a,b)+->a-b,oneStr,oneRef,sqStr,sqRef,yRef,0)
  intRef := ref((-1) :: COM)
  lazyInteg(cc,iTimes(yStr,yRef,fpStr,fpRef,intRef,0),intRef,ansRef)

iTan: (%,%,Coef,I) -> %
iTan(f,fp,cc,sign) ==
  -- computes the tangent (and related functions) of f.
  fpRef := getRef fp; fpStr := getStream fp
  ansRef := ref(0 :: COM)
  ansStr := Ys(s+>tan0(cc,s,ansRef,fpStr,fpRef,sign))
  zero? cc => makeSeries(ansRef,rst ansStr)
  makeSeries(ansRef,ansStr)

--% Error Reporting

TRCONST : SG := "series expansion involves transcendental constants"
NPOWERS : SG := "series expansion has terms of negative degree"
FPOWERS : SG := "series expansion has terms of fractional degree"
MAYFPOW : SG := "series expansion may have terms of fractional degree"
LOGS : SG := "series expansion has logarithmic term"
NPOWLOG : SG :=
  "series expansion has terms of negative degree or logarithmic term"

```



```

NOTINV : SG := "leading coefficient not invertible"

--% Rational powers and transcendental functions

orderOrFailed : % -> Union(I,"failed")
orderOrFailed uts ==
-- returns the order of x or "failed"
-- if -1 is returned, the series is identically zero
x := getStream uts
for n in 0..1000 repeat
  explicitlyEmpty? x => return -1
  explicitEntries? x => return getExpon first x
  lazyEvaluate x
"failed"

RATPOWERS : Boolean := Coef has "***": (Coef,RN) -> Coef
TRANSFCN : Boolean := Coef has TranscendentalFunctionCategory

cRationalPower(uts,r) ==
(ord0 := orderOrFailed uts) case "failed" =>
  error "***: series with many leading zero coefficients"
order := ord0 :: I
(n := order exquo denom(r)) case "failed" =>
  error "***: rational power does not exist"
cc := coefficient(uts,order)
(ccInv := recip cc) case "failed" => error concat("***: ",NOTINV)
ccPow :=
--   one? cc => cc
  (cc = 1) => cc
--   one? denom r =>
  (denom r) = 1 =>
    not negative?(num := numer r) => cc ** (num :: NNI)
    (ccInv :: Coef) ** ((-num) :: NNI)
  RATPOWERS => cc ** r
  error "*** rational power of coefficient undefined"
uts1 := (ccInv :: Coef) * uts
uts2 := uts1 * monomial(1,-order)
monomial(ccPow,(n :: I) * numer(r)) * cPower(uts2,r :: Coef)

cExp uts ==
zero?(cc := coefficient(uts,0)) => iExp(uts,1)
TRANSFCN => iExp(uts,exp cc)
error concat("exp: ",TRCONST)

cLog uts ==
zero?(cc := coefficient(uts,0)) =>
  error "log: constant coefficient should not be 0"
--   one? cc => integrate(differentiate(uts) * (iExquo(1,uts,true) :: %))
  (cc = 1) => integrate(differentiate(uts) * (iExquo(1,uts,true) :: %))
TRANSFCN =>

```

```

    y := iExquo(1,uts,true) :: %
    (log(cc) :: %) + integrate(y * differentiate(uts))
    error concat("log: ",TRCONST)

sincos: % -> Record(%sin: %, %cos: %)
sincos uts ==
    zero?(cc := coefficient(uts,0)) => iSincos(uts,0,1,-1)
    TRANSFCN => iSincos(uts,sin cc,cos cc,-1)
    error concat("sincos: ",TRCONST)

cSin uts == sincos(uts).%sin
cCos uts == sincos(uts).%cos

cTan uts ==
    zero?(cc := coefficient(uts,0)) => iTan(uts,differentiate uts,0,1)
    TRANSFCN => iTan(uts,differentiate uts,tan cc,1)
    error concat("tan: ",TRCONST)

cCot uts ==
    zero? uts => error "cot: cot(0) is undefined"
    zero?(cc := coefficient(uts,0)) => error error concat("cot: ",NPOWERS)
    TRANSFCN => iTan(uts,-differentiate uts,cot cc,1)
    error concat("cot: ",TRCONST)

cSec uts ==
    zero?(cc := coefficient(uts,0)) => iExquo(1,cCos uts,true) :: %
    TRANSFCN =>
        cosUts := cCos uts
        zero? coefficient(cosUts,0) => error concat("sec: ",NPOWERS)
        iExquo(1,cosUts,true) :: %
    error concat("sec: ",TRCONST)

cCsc uts ==
    zero? uts => error "csc: csc(0) is undefined"
    TRANSFCN =>
        sinUts := cSin uts
        zero? coefficient(sinUts,0) => error concat("csc: ",NPOWERS)
        iExquo(1,sinUts,true) :: %
    error concat("csc: ",TRCONST)

cAsin uts ==
    zero?(cc := coefficient(uts,0)) =>
        integrate(cRationalPower(1 - uts*uts,-1/2) * differentiate(uts))
    TRANSFCN =>
        x := 1 - uts * uts
        cc = 1 or cc = -1 =>
            -- compute order of 'x'
            (ord := orderOrFailed x) case "failed" =>
                error concat("asin: ",MAYFPOW)
            (order := ord :: I) = -1 => return asin(cc) :: %

```

```

    odd? order => error concat("asin: ",FPOWERS)
    c0 := asin(cc) :: %
    c0 + integrate(cRationalPower(x,-1/2) * differentiate(uts))
    c0 := asin(cc) :: %
    c0 + integrate(cRationalPower(x,-1/2) * differentiate(uts))
    error concat("asin: ",TRCONST)

cAcos uts ==
zero? uts =>
    TRANSFCN => acos(0)$Coef :: %
    error concat("acos: ",TRCONST)
TRANSFCN =>
    x := 1 - uts * uts
    cc := coefficient(uts,0)
    cc = 1 or cc = -1 =>
        -- compute order of 'x'
        (ord := orderOrFailed x) case "failed" =>
            error concat("acos: ",MAYFPOW)
        (order := ord :: I) = -1 => return acos(cc) :: %
    odd? order => error concat("acos: ",FPOWERS)
    c0 := acos(cc) :: %
    c0 + integrate(-cRationalPower(x,-1/2) * differentiate(uts))
    c0 := acos(cc) :: %
    c0 + integrate(-cRationalPower(x,-1/2) * differentiate(uts))
    error concat("acos: ",TRCONST)

cAtan uts ==
zero?(cc := coefficient(uts,0)) =>
    y := iExquo(1,(1 :: %) + uts*uts,true) :: %
    integrate(y * (differentiate uts))
TRANSFCN =>
    (y := iExquo(1,(1 :: %) + uts*uts,true)) case "failed" =>
        error concat("atan: ",LOGS)
    (atan(cc) :: %) + integrate((y :: %) * (differentiate uts))
    error concat("atan: ",TRCONST)

cAcot uts ==
TRANSFCN =>
    (y := iExquo(1,(1 :: %) + uts*uts,true)) case "failed" =>
        error concat("acot: ",LOGS)
    cc := coefficient(uts,0)
    (acot(cc) :: %) + integrate(-(y :: %) * (differentiate uts))
    error concat("acot: ",TRCONST)

cAsec uts ==
zero?(cc := coefficient(uts,0)) =>
    error "asec: constant coefficient should not be 0"
TRANSFCN =>
    x := uts * uts - 1
    y :=

```

```

cc = 1 or cc = -1 =>
  -- compute order of 'x'
  (ord := orderOrFailed x) case "failed" =>
    error concat("asec: ",MAYFPOW)
  (order := ord :: I) = -1 => return asec(cc) :: %
  odd? order => error concat("asec: ",FPOWERS)
  cRationalPower(x,-1/2) * differentiate(uts)
  cRationalPower(x,-1/2) * differentiate(uts)
  (z := iExquo(y,uts,true)) case "failed" =>
    error concat("asec: ",NOTINV)
  (asec(cc) :: %) + integrate(z :: %)
error concat("asec: ",TRCONST)

cAsec uts ==
zero?(cc := coefficient(uts,0)) =>
  error "acsc: constant coefficient should not be 0"
TRANSFCN =>
  x := uts * uts - 1
  y :=
    cc = 1 or cc = -1 =>
      -- compute order of 'x'
      (ord := orderOrFailed x) case "failed" =>
        error concat("acsc: ",MAYFPOW)
      (order := ord :: I) = -1 => return acsc(cc) :: %
      odd? order => error concat("acsc: ",FPOWERS)
      -cRationalPower(x,-1/2) * differentiate(uts)
      -cRationalPower(x,-1/2) * differentiate(uts)
      (z := iExquo(y,uts,true)) case "failed" =>
        error concat("asec: ",NOTINV)
      (acsc(cc) :: %) + integrate(z :: %)
    error concat("acsc: ",TRCONST)

sinhcosh: % -> Record(%sinh: %, %cosh: %)
sinhcosh uts ==
  zero?(cc := coefficient(uts,0)) =>
    tmp := iSincos(uts,0,1,1)
    [tmp.%sin,tmp.%cos]
  TRANSFCN =>
    tmp := iSincos(uts,sinh cc,cosh cc,1)
    [tmp.%sin,tmp.%cos]
  error concat("sinhcosh: ",TRCONST)

cSinh uts == sinhcosh(uts).%sinh
cCosh uts == sinhcosh(uts).%cosh

cTanh uts ==
  zero?(cc := coefficient(uts,0)) => iTan(uts,differentiate uts,0,-1)
  TRANSFCN => iTan(uts,differentiate uts,tanh cc,-1)
  error concat("tanh: ",TRCONST)

```

```

cCoth uts ==
  tanhUts := cTanh uts
  zero? tanhUts => error "coth: coth(0) is undefined"
  zero? coefficient(tanhUts,0) => error concat("coth: ",NPOWERS)
  iExquo(1,tanhUts,true) :: %

cSech uts ==
  coshUts := cCosh uts
  zero? coefficient(coshUts,0) => error concat("sech: ",NPOWERS)
  iExquo(1,coshUts,true) :: %

cCsch uts ==
  sinhUts := cSinh uts
  zero? coefficient(sinhUts,0) => error concat("csch: ",NPOWERS)
  iExquo(1,sinhUts,true) :: %

cAsinh uts ==
  x := 1 + uts * uts
  zero?(cc := coefficient(uts,0)) => cLog(uts + cRationalPower(x,1/2))
  TRANSFCN =>
    (ord := orderOrFailed x) case "failed" =>
      error concat("asinh: ",MAYFPOW)
    (order := ord :: I) = -1 => return asinh(cc) :: %
    odd? order => error concat("asinh: ",FPOWERS)
    -- the argument to 'log' must have a non-zero constant term
    cLog(uts + cRationalPower(x,1/2))
    error concat("asinh: ",TRCONST)

cAcosh uts ==
  zero? uts =>
    TRANSFCN => acosh(0)$Coef :: %
    error concat("acosh: ",TRCONST)
  TRANSFCN =>
    cc := coefficient(uts,0); x := uts*uts - 1
    cc = 1 or cc = -1 =>
      -- compute order of 'x'
      (ord := orderOrFailed x) case "failed" =>
        error concat("acosh: ",MAYFPOW)
      (order := ord :: I) = -1 => return acosh(cc) :: %
      odd? order => error concat("acosh: ",FPOWERS)
      -- the argument to 'log' must have a non-zero constant term
      cLog(uts + cRationalPower(x,1/2))
      cLog(uts + cRationalPower(x,1/2))
      error concat("acosh: ",TRCONST)

cAtanh uts ==
  half := inv(2 :: RN) :: Coef
  zero?(cc := coefficient(uts,0)) =>
    half * (cLog(1 + uts) - cLog(1 - uts))
  TRANSFCN =>

```

```

cc = 1 or cc = -1 => error concat("atanh: ",LOGS)
half * (cLog(1 + uts) - cLog(1 - uts))
error concat("atanh: ",TRCONST)

cAcoth uts ==
zero? uts =>
  TRANSFCN => acoth(0)$Coef :: %
  error concat("acoth: ",TRCONST)
TRANSFCN =>
  cc := coefficient(uts,0); half := inv(2 :: RN) :: Coef
  cc = 1 or cc = -1 => error concat("acoth: ",LOGS)
  half * (cLog(uts + 1) - cLog(uts - 1))
  error concat("acoth: ",TRCONST)

cAsech uts ==
zero? uts => error "asech: asech(0) is undefined"
TRANSFCN =>
  zero?(cc := coefficient(uts,0)) =>
    error concat("asech: ",NPOWLOG)
  x := 1 - uts * uts
  cc = 1 or cc = -1 =>
    -- compute order of 'x'
    (ord := orderOrFailed x) case "failed" =>
      error concat("asech: ",MAYFPOW)
    (order := ord :: I) = -1 => return asech(cc) :: %
    odd? order => error concat("asech: ",FPOWERS)
    (utsInv := iExquo(1,uts,true)) case "failed" =>
      error concat("asech: ",NOTINV)
    cLog((1 + cRationalPower(x,1/2)) * (utsInv :: %))
    (utsInv := iExquo(1,uts,true)) case "failed" =>
      error concat("asech: ",NOTINV)
    cLog((1 + cRationalPower(x,1/2)) * (utsInv :: %))
    error concat("asech: ",TRCONST)

cAcsch uts ==
zero? uts => error "acsch: acsch(0) is undefined"
TRANSFCN =>
  zero?(cc := coefficient(uts,0)) => error concat("acsch: ",NPOWLOG)
  x := uts * uts + 1
  -- compute order of 'x'
  (ord := orderOrFailed x) case "failed" =>
    error concat("acsc: ",MAYFPOW)
  (order := ord :: I) = -1 => return acsch(cc) :: %
  odd? order => error concat("acsch: ",FPOWERS)
  (utsInv := iExquo(1,uts,true)) case "failed" =>
    error concat("acsch: ",NOTINV)
  cLog((1 + cRationalPower(x,1/2)) * (utsInv :: %))
  error concat("acsch: ",TRCONST)

```

--% Output forms

```

-- check a global Lisp variable
factorials?() == false

termOutput(k,c,vv) ==
-- creates a term c * vv ** k
  k = 0 => c :: OUT
  mon := (k = 1 => vv; vv ** (k :: OUT))
--      if factorials?() and k > 1 then
--          c := factorial(k)$IntegerCombinatoricFunctions * c
--          mon := mon / hconcat(k :: OUT,"!" :: OUT)
  c = 1 => mon
  c = -1 => -mon
  (c :: OUT) * mon

-- check a global Lisp variable
showAll?() == true

seriesToOutputForm(st,refer,var,cen,r) ==
  vv :=
    zero? cen => var :: OUT
    paren(var :: OUT - cen :: OUT)
  l : L OUT := empty()
  while explicitEntries? st repeat
    term := first st
    l := concat(termOutput(getExpon(term) * r,getCoef term,vv),l)
    st := rst st
  l :=
    explicitlyEmpty? st => l
    (deg := retractIfCan(elt refer)@Union(I,"failed")) case I =>
      concat(prefix("0" :: OUT,[vv ** (((deg :: I) + 1) * r) :: OUT)],l)
  l
  empty? l => (0$Coef) :: OUT
  reduce("+",reverse_! l)

```

— ISUPS.dotabb —

```

"ISUPS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ISUPS"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ISUPS" -> "ALIST"

```

domain INTABL InnerTable

— InnerTable.input —

```

)set break resume
)sys rm -f InnerTable.output
)spool InnerTable.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerTable
--R
--R InnerTable(Key: SetCategory,Entry: SetCategory,addDom) where
--R   addDom: TableAggregate(Key,Entry) with
--R       finiteAggregate is a domain constructor
--R Abbreviation for InnerTable is INTABL
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for INTABL
--R
--R----- Operations -----
--R copy : % -> %                dictionary : () -> %
--R elt : (% ,Key,Entry) -> Entry    ?.? : (% ,Key) -> Entry
--R empty : () -> %                empty? : % -> Boolean
--R entries : % -> List(Entry)      eq? : (% ,%) -> Boolean
--R index? : (Key,% ) -> Boolean    indices : % -> List(Key)
--R key? : (Key,% ) -> Boolean      keys : % -> List(Key)
--R map : ((Entry -> Entry),%) -> %  qelt : (% ,Key) -> Entry
--R sample : () -> %              setelt : (% ,Key,Entry) -> Entry
--R table : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ==? : (% ,%) -> Boolean if Record(key: Key,entry: Entry) has SETCAT or Entry has SETCAT
--R any? : ((Entry -> Boolean),%) -> Boolean if $ has finiteAggregate
--R any? : ((Record(key: Key,entry: Entry) -> Boolean),%) -> Boolean if $ has finiteAggregate
--R bag : List(Record(key: Key,entry: Entry)) -> %
--R coerce : % -> OutputForm if Record(key: Key,entry: Entry) has SETCAT or Entry has SETCAT
--R construct : List(Record(key: Key,entry: Entry)) -> %
--R convert : % -> InputForm if Record(key: Key,entry: Entry) has KONVERT(INFORM)
--R count : ((Entry -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R count : (Entry,% ) -> NonNegativeInteger if $ has finiteAggregate and Entry has SETCAT
--R count : (Record(key: Key,entry: Entry),%) -> NonNegativeInteger if $ has finiteAggregate and Record(key
--R count : ((Record(key: Key,entry: Entry) -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R dictionary : List(Record(key: Key,entry: Entry)) -> %
--R entry? : (Entry,% ) -> Boolean if $ has finiteAggregate and Entry has SETCAT
--R eval : (% ,List(Equation(Entry))) -> % if Entry has EVALAB(Entry) and Entry has SETCAT
--R eval : (% ,Equation(Entry)) -> % if Entry has EVALAB(Entry) and Entry has SETCAT
--R eval : (% ,Entry,Entry) -> % if Entry has EVALAB(Entry) and Entry has SETCAT

```



```

--R eval : (% , List(Entry), List(Entry)) -> % if Entry has EVALAB(Entry) and Entry has SETCAT
--R eval : (% , List(Record(key: Key, entry: Entry)), List(Record(key: Key, entry: Entry))) -> % if R
--R eval : (% , Record(key: Key, entry: Entry), Record(key: Key, entry: Entry)) -> % if Record(key: K
--R eval : (% , Equation(Record(key: Key, entry: Entry))) -> % if Record(key: Key, entry: Entry) has
--R eval : (% , List(Equation(Record(key: Key, entry: Entry)))) -> % if Record(key: Key, entry: Entr
--R every? : ((Entry -> Boolean), %) -> Boolean if $ has finiteAggregate
--R every? : ((Record(key: Key, entry: Entry) -> Boolean), %) -> Boolean if $ has finiteAggregate
--R extract! : % -> Record(key: Key, entry: Entry)
--R fill! : (% , Entry) -> % if $ has shallowlyMutable
--R find : ((Record(key: Key, entry: Entry) -> Boolean), %) -> Union(Record(key: Key, entry: Entry)
--R first : % -> Entry if Key has ORDSET
--R hash : % -> SingleInteger if Record(key: Key, entry: Entry) has SETCAT or Entry has SETCAT
--R insert! : (Record(key: Key, entry: Entry), %) -> %
--R inspect : % -> Record(key: Key, entry: Entry)
--R latex : % -> String if Record(key: Key, entry: Entry) has SETCAT or Entry has SETCAT
--R less? : (% , NonNegativeInteger) -> Boolean
--R map : (((Entry, Entry) -> Entry), %, %) -> %
--R map : ((Record(key: Key, entry: Entry) -> Record(key: Key, entry: Entry)), %) -> %
--R map! : ((Entry -> Entry), %) -> % if $ has shallowlyMutable
--R map! : ((Record(key: Key, entry: Entry) -> Record(key: Key, entry: Entry)), %) -> % if $ has sh
--R maxIndex : % -> Key if Key has ORDSET
--R member? : (Entry, %) -> Boolean if $ has finiteAggregate and Entry has SETCAT
--R member? : (Record(key: Key, entry: Entry), %) -> Boolean if $ has finiteAggregate and Record(k
--R members : % -> List(Entry) if $ has finiteAggregate
--R members : % -> List(Record(key: Key, entry: Entry)) if $ has finiteAggregate
--R minIndex : % -> Key if Key has ORDSET
--R more? : (% , NonNegativeInteger) -> Boolean
--R parts : % -> List(Entry) if $ has finiteAggregate
--R parts : % -> List(Record(key: Key, entry: Entry)) if $ has finiteAggregate
--R qsetelt! : (% , Key, Entry) -> Entry if $ has shallowlyMutable
--R reduce : (((Record(key: Key, entry: Entry), Record(key: Key, entry: Entry)) -> Record(key: Key,
--R reduce : (((Record(key: Key, entry: Entry), Record(key: Key, entry: Entry)) -> Record(key: Key,
--R reduce : (((Record(key: Key, entry: Entry), Record(key: Key, entry: Entry)) -> Record(key: Key,
--R remove : ((Record(key: Key, entry: Entry) -> Boolean), %) -> % if $ has finiteAggregate
--R remove : (Record(key: Key, entry: Entry), %) -> % if $ has finiteAggregate and Record(key: Key
--R remove! : (Key, %) -> Union(Entry, "failed")
--R remove! : ((Record(key: Key, entry: Entry) -> Boolean), %) -> % if $ has finiteAggregate
--R remove! : (Record(key: Key, entry: Entry), %) -> % if $ has finiteAggregate
--R removeDuplicates : % -> % if $ has finiteAggregate and Record(key: Key, entry: Entry) has SET
--R search : (Key, %) -> Union(Entry, "failed")
--R select : ((Record(key: Key, entry: Entry) -> Boolean), %) -> % if $ has finiteAggregate
--R select! : ((Record(key: Key, entry: Entry) -> Boolean), %) -> % if $ has finiteAggregate
--R size? : (% , NonNegativeInteger) -> Boolean
--R swap! : (% , Key, Key) -> Void if $ has shallowlyMutable
--R table : List(Record(key: Key, entry: Entry)) -> %
--R ?~=? : (% , %) -> Boolean if Record(key: Key, entry: Entry) has SETCAT or Entry has SETCAT
--R
--E 1

```

```

)spool

```

)lisp (bye)

— InnerTable.help —

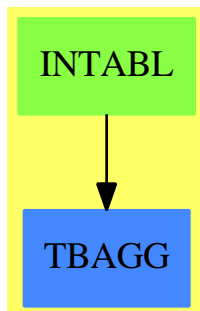
```
=====
InnerTable examples
=====
```

This domain is used to provide a conditional "add" domain
for the implementation of Table.

See Also:

o)show InnerTable

InnerTable (INTABL)



See

- ⇒ "HashTable" (HASHTBL) [9](#) on page [1120](#)
- ⇒ "Table" (TABLE) [21](#) on page [2663](#)
- ⇒ "EqTable" (EQTBL) [6](#) on page [698](#)
- ⇒ "StringTable" (STRTBL) [20](#) on page [2610](#)
- ⇒ "GeneralSparseTable" (GSTBL) [8](#) on page [1078](#)
- ⇒ "SparseTable" (STBL) [20](#) on page [2450](#)

Exports:

any?	any?	bag	coerce	construct
convert	copy	count	dictionary	entry?
elt	empty	empty?	entries	eq?
eval	every?	extract!	fill!	find
first	hash	index?	indices	insert!
inspect	key?	keys	latex	less?
map	map!	maxIndex	member?	members
minIndex	more?	parts	sample	qelt
qsetelt!	reduce	remove	remove!	removeDuplicates
search	select	select!	setelt	size?
swap!	table	??	#?	?=?
?^=?				

— domain INTABL InnerTable —

```

)abbrev domain INTABL InnerTable
++ Author: Barry Trager
++ Date Created: 1992
++ Date Last Updated: Sept 15, 1992
++ Description:
++ This domain is used to provide a conditional "add" domain
++ for the implementation of \spadtype{Table}.

InnerTable(Key: SetCategory, Entry: SetCategory, addDom):Exports == Implementation where
  addDom : TableAggregate(Key, Entry) with
    finiteAggregate
  Exports ==> TableAggregate(Key, Entry) with
    finiteAggregate
  Implementation ==> addDom

```

— INTABL.dotabb —

```

"INTABL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=INTABL"]
"TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]
"INTABL" -> "TBAGG"

```

domain ITAYLOR InnerTaylorSeries

— InnerTaylorSeries.input —

```

)set break resume
)sys rm -f InnerTaylorSeries.output
)spool InnerTaylorSeries.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerTaylorSeries
--R
--R InnerTaylorSeries(Coef: Ring) is a domain constructor
--R Abbreviation for InnerTaylorSeries is ITAYLOR
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ITAYLOR
--R
--R----- Operations -----
--R ??? : (%,Integer) -> %          ??? : (%,Coef) -> %
--R ??? : (Coef,%) -> %            ??? : (%,%) -> %
--R ??? : (Integer,%) -> %         ??? : (PositiveInteger,%) -> %
--R ??? : (%,PositiveInteger) -> % ?+? : (%,%) -> %
--R ?-? : (%,%) -> %              -? : % -> %
--R ?? : (%,%) -> Boolean         1 : () -> %
--R 0 : () -> %                  ?? : (%,PositiveInteger) -> %
--R coefficients : % -> Stream(Coef)  coerce : Integer -> %
--R coerce : % -> OutputForm          hash : % -> SingleInteger
--R latex : % -> String              one? : % -> Boolean
--R order : % -> NonNegativeInteger   pole? : % -> Boolean
--R recip : % -> Union(%, "failed")   sample : () -> %
--R series : Stream(Coef) -> %        zero? : % -> Boolean
--R ?^=? : (%,%) -> Boolean
--R ??? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R ?? : (%,NonNegativeInteger) -> %
--R associates? : (%,%) -> Boolean if Coef has INTDOM
--R characteristic : () -> NonNegativeInteger
--R coerce : % -> % if Coef has INTDOM
--R exquo : (%,%) -> Union(%, "failed") if Coef has INTDOM
--R order : (%,NonNegativeInteger) -> NonNegativeInteger
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R unit? : % -> Boolean if Coef has INTDOM
--R unitCanonical : % -> % if Coef has INTDOM
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %) if Coef has INTDOM
--R
--E 1

)spool
)lisp (bye)

```

— InnerTaylorSeries.help —

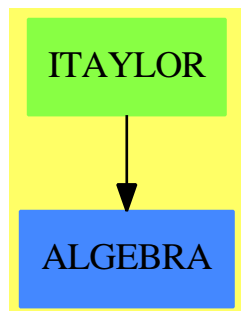
```
=====
InnerTaylorSeries examples
=====
```

Internal package for dense Taylor series. This is an internal Taylor series type in which Taylor series are represented by a Stream of Ring elements. For univariate series, the Stream elements are the Taylor coefficients. For multivariate series, the n-th Stream element is a form of degree n in the power series variables.

See Also:

```
o )show InnerTaylorSeries
```

InnerTaylorSeries (ITAYLOR)



See

⇒ “UnivariateTaylorSeries” (UTS) [22](#) on page [2876](#)

Exports:

0	1	associates?	characteristic	coefficients
coerce	exquo	hash	latex	one?
order	pole?	recip	sample	series
subtractIfCan	unit?	unitCanonical	unitNormal	zero?
?^=?	?*?	?**?	?^?	?+?
?-?	-?	?=?		

— domain ITAYLOR InnerTaylorSeries —

```
)abbrev domain ITAYLOR InnerTaylorSeries
++ Author: Clifton J. Williamson
```

```

++ Date Created: 21 December 1989
++ Date Last Updated: 25 February 1989
++ Description:
++ Internal package for dense Taylor series.
++ This is an internal Taylor series type in which Taylor series
++ are represented by a \spadtype{Stream} of \spadtype{Ring} elements.
++ For univariate series, the \spad{Stream} elements are the Taylor
++ coefficients. For multivariate series, the \spad{n}th Stream element
++ is a form of degree n in the power series variables.

InnerTaylorSeries(Coef): Exports == Implementation where
  Coef : Ring
  I ==> Integer
  NNI ==> NonNegativeInteger
  ST ==> Stream Coef
  STT ==> StreamTaylorSeriesOperations Coef

Exports ==> Ring with
  coefficients: % -> Stream Coef
    ++\spad{coefficients(x)} returns a stream of ring elements.
    ++ When x is a univariate series, this is a stream of Taylor
    ++ coefficients. When x is a multivariate series, the
    ++ \spad{n}th element of the stream is a form of
    ++ degree n in the power series variables.
  series: Stream Coef -> %
    ++\spad{series(s)} creates a power series from a stream of
    ++ ring elements.
    ++ For univariate series types, the stream s should be a stream
    ++ of Taylor coefficients. For multivariate series types, the
    ++ stream s should be a stream of forms the \spad{n}th element
    ++ of which is a
    ++ form of degree n in the power series variables.
  pole?: % -> Boolean
    ++\spad{pole?(x)} tests if the series x has a pole.
    ++ Note: this is false when x is a Taylor series.
  order: % -> NNI
    ++\spad{order(x)} returns the order of a power series x,
    ++ i.e. the degree of the first non-zero term of the series.
  order: (% , NNI) -> NNI
    ++\spad{order(x,n)} returns the minimum of n and the order of x.
  "*" : (Coef, %) -> %
    ++\spad{c*x} returns the product of c and the series x.
  "*" : (% , Coef) -> %
    ++\spad{x*c} returns the product of c and the series x.
  "*" : (% , Integer) -> %
    ++\spad{x*i} returns the product of integer i and the series x.
  if Coef has IntegralDomain then IntegralDomain
    --++ An IntegralDomain provides 'exquo'

Implementation ==> add

```

```

Rep := Stream Coef

--% declarations
x,y: %

--% definitions

-- In what follows, we will be calling operations on Streams
-- which are NOT defined in the package Stream. Thus, it is
-- necessary to explicitly pass back and forth between Rep and %.
-- This will be done using the functions 'stream' and 'series'.

stream : % -> Stream Coef
stream x == x pretend Stream(Coef)
series st == st @ %

0 == coerce(0)$STT
1 == coerce(1)$STT

x = y ==
  -- tests if two power series are equal
  -- difference must be a finite stream of zeroes of length <= n + 1,
  -- where n = $streamCount$Lisp
  st : ST := stream(x - y)
  n : I := _$streamCount$Lisp
  for i in 0..n repeat
    empty? st => return true
    first st ^= 0 => return false
    st := rst st
  empty? st

coefficients x == stream x

x + y          == stream(x) +$STT stream(y)
x - y          == stream(x) -$STT stream(y)
(x:%) * (y:%)  == stream(x) *$STT stream(y)
- x            == -$STT (stream x)
(i:I) * (x:%)  == (i::Coef) *$STT stream x
(x:%) * (i:I)  == stream(x) *$STT (i::Coef)
(c:Coef) * (x:%) == c *$STT stream x
(x:%) * (c:Coef) == stream(x) *$STT c

recip x ==
  (rec := recip$STT stream x) case "failed" => "failed"
  series(rec :: ST)

if Coef has IntegralDomain then

  x exquo y ==

```

```

    (quot := stream(x) exquo$STT stream(y)) case "failed" => "failed"
    series(quot :: ST)

x:% ** n:NNI ==
  n = 0 => 1
  expt(x,n :: PositiveInteger)$RepeatedSquaring(%)

characteristic() == characteristic()$Coef
pole? x == false

iOrder: (ST,NNI,NNI) -> NNI
iOrder(st,n,n0) ==
  (n = n0) or (empty? st) => n0
  zero? first st => iOrder(rst st,n + 1,n0)
  n

order(x,n) == iOrder(stream x,0,n)

iOrder2: (ST,NNI) -> NNI
iOrder2(st,n) ==
  empty? st => error "order: series has infinite order"
  zero? first st => iOrder2(rst st,n + 1)
  n

order x == iOrder2(stream x,0)

```

— ITAYLOR.dotabb —

```

"ITAYLOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ITAYLOR"]
"ALGEBRA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ALGEBRA"]
"ITAYLOR" -> "ALGEBRA"

```

domain INFORM InputForm

— InputForm.input —

```

)set break resume
)sys rm -f InputForm.output
)spool InputForm.output
)set message test on

```



```

)set message auto off
)clear all

--S 1 of 1
)show InputForm
--R
--R InputForm is a domain constructor
--R Abbreviation for InputForm is INFORM
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for INFORM
--R
--R----- Operations -----
--R #? : % -> Integer          ?? : (%,% ) -> %
--R ??? : (%,Integer) -> %    ?+? : (%,% ) -> %
--R ?/? : (%,% ) -> %        ?=? : (%,% ) -> Boolean
--R 1 : () -> %              0 : () -> %
--R atom? : % -> Boolean     binary : (% ,List(%)) -> %
--R car : % -> %             cdr : % -> %
--R coerce : % -> OutputForm convert : SExpression -> %
--R convert : % -> SExpression convert : OutputForm -> %
--R convert : DoubleFloat -> % convert : Integer -> %
--R convert : Symbol -> %    convert : String -> %
--R convert : List(% ) -> %  declare : List(% ) -> Symbol
--R destruct : % -> List(% ) ?.? : (% ,List(Integer)) -> %
--R ?.? : (% ,Integer) -> %  eq : (% ,%) -> Boolean
--R expr : % -> OutputForm   flatten : % -> %
--R float : % -> DoubleFloat float? : % -> Boolean
--R hash : % -> SingleInteger integer : % -> Integer
--R integer? : % -> Boolean  interpret : % -> Any
--R lambda : (% ,List(Symbol)) -> % latex : % -> String
--R list? : % -> Boolean     null? : % -> Boolean
--R pair? : % -> Boolean     parse : String -> %
--R string : % -> String     string? : % -> Boolean
--R symbol : % -> Symbol     symbol? : % -> Boolean
--R unparse : % -> String    ?~=? : (% ,%) -> Boolean
--R ??? : (% ,NonNegativeInteger) -> %
--R compile : (Symbol,List(%)) -> Symbol
--R function : (% ,List(Symbol),Symbol) -> %
--R
--E 1

)spool
)lisp (bye)

```

— InputForm.help —

=====

InputForm examples

=====

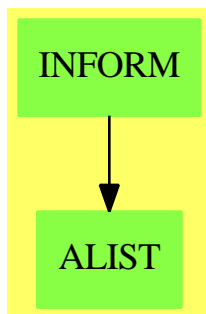
Domain of parsed forms which can be passed to the interpreter.
 This is also the interface between algebra code and facilities
 in the interpreter.

See Also:

o)show InputForm

—————→

InputForm (INFORM)



Exports:

0	1	atom?	binary	car
cdr	coerce	compile	convert	declare
destruct	eq	expr	flatten	float
float?	function	hash	integer	integer?
interpret	lambda	latex	list?	null?
pair?	parse	string	string?	symbol
symbol?	unparse	#?	?~=?	?**?
?*?	?+?	?/?	?=?	?..?

— domain INFORM InputForm —

```

)abbrev domain INFORM InputForm
++ Author: Manuel Bronstein
++ Date Last Updated: 19 April 1991
++ Description:
++ Domain of parsed forms which can be passed to the interpreter.
++ This is also the interface between algebra code and facilities
++ in the interpreter.
  
```

```

-->boot $noSubsumption := true

InputForm():
  Join(SExpressionCategory(String,Symbol,Integer,DoubleFloat,OutputForm),
    ConvertibleTo SExpression) with
  interpret: % -> Any
    ++ interpret(f) passes f to the interpreter.
  convert : SExpression -> %
    ++ convert(s) makes s into an input form.
  binary : (%, List %) -> %
    ++ \spad{binary(op, [a1,...,an])} returns the input form
    ++ corresponding to \spad{a1 op a2 op ... op an}.
    ++
    ++X a:=[1,2,3]::List(InputForm)
    ++X binary(_+::InputForm,a)

function : (%, List Symbol, Symbol) -> %
  ++ \spad{function(code, [x1,...,xn], f)} returns the input form
  ++ corresponding to \spad{f(x1,...,xn) == code}.
lambda : (%, List Symbol) -> %
  ++ \spad{lambda(code, [x1,...,xn])} returns the input form
  ++ corresponding to \spad{(x1,...,xn) +-> code} if \spad{n > 1},
  ++ or to \spad{x1 +-> code} if \spad{n = 1}.
"+" : (%, %) -> %
  ++ \spad{a + b} returns the input form corresponding to \spad{a + b}.
"*" : (%, %) -> %
  ++ \spad{a * b} returns the input form corresponding to \spad{a * b}.
"/" : (%, %) -> %
  ++ \spad{a / b} returns the input form corresponding to \spad{a / b}.
"%" : (%, NonNegativeInteger) -> %
  ++ \spad{a ** b} returns the input form corresponding to \spad{a ** b}.
"%" : (%, Integer) -> %
  ++ \spad{a ** b} returns the input form corresponding to \spad{a ** b}.
0 : constant -> %
  ++ \spad{0} returns the input form corresponding to 0.
1 : constant -> %
  ++ \spad{1} returns the input form corresponding to 1.
flatten : % -> %
  ++ flatten(s) returns an input form corresponding to s with
  ++ all the nested operations flattened to triples using new
  ++ local variables.
  ++ If s is a piece of code, this speeds up
  ++ the compilation tremendously later on.
unparse : % -> String
  ++ unparse(f) returns a string s such that the parser
  ++ would transform s to f.
  ++ Error: if f is not the parsed form of a string.
parse : String -> %
  ++ parse(s) is the inverse of unparse. It parses a string to InputForm.
declare : List % -> Symbol

```

```

++ declare(t) returns a name f such that f has been
++ declared to the interpreter to be of type t, but has
++ not been assigned a value yet.
++ Note: t should be created as \spad{devaluate(T)$Lisp} where T is the
++ actual type of f (this hack is required for the case where
++ T is a mapping type).
compile : (Symbol, List %) -> Symbol
++ \spad{compile(f, [t1,...,tn])} forces the interpreter to compile
++ the function f with signature \spad{(t1,...,tn) -> ?}.
++ returns the symbol f if successful.
++ Error: if f was not defined beforehand in the interpreter,
++ or if the ti's are not valid types, or if the compiler fails.
== SExpression add
Rep := SExpression

mkProperOp: Symbol -> %
strsym    : % -> String
tuplify   : List Symbol -> %
flatten0  : (% , Symbol, NonNegativeInteger) ->
                                         Record(lst: List %, symb:%)

0          == convert(0::Integer)
1          == convert(1::Integer)
convert(x:%):SExpression == x pretend SExpression
convert(x:SExpression):% == x

conv(ll : List %): % ==
  convert(ll pretend List SExpression)$SExpression pretend %

lambda(f,l) == conv([convert("+-> "::Symbol),tuplify l,f]$List(%))

interpret x ==
  v := interpret(x)$Lisp
  mkObj(unwrap(objVal(v)$Lisp)$Lisp, objMode(v)$Lisp)$Lisp

convert(x:DoubleFloat):% ==
  zero? x => 0
--   one? x => 1
  (x = 1) => 1
  convert(x)$Rep

flatten s ==
  -- will not compile if I use 'or'
  atom? s => s
  every?(atom?,destruct s)$List(%) => s
  sy := new()$Symbol
  n:NonNegativeInteger := 0
  l2 := [flatten0(x, sy, n := n + 1) for x in rest(l := destruct s)]
  conv(concat(convert("SEQ"::Symbol)@%,
    concat(concat [u.lst for u in l2], conv(

```

```

[convert("exit"::Symbol)@%, 1$%, conv(concat(first l,
[u.symb for u in l2]))@%]$List(%)@%))@%

flatten0(s, sy, n) ==
  atom? s => [nil(), s]
  a := convert(concat(string sy, convert(n)@String)::Symbol)@%
  l2 := [flatten0(x, sy, n := n+1) for x in rest(l := destruct s)]
  [concat(concat [u.lst for u in l2], conv([convert(
    "LET"::Symbol)@%, a, conv(concat(first l,
    [u.symb for u in l2]))@%]$List(%)@%), a]

strsym s ==
  string? s => string s
  symbol? s => string symbol s
  error "strsym: form is neither a string or symbol"

-- given a function this will attempt to recreate the input string
unparse x ==
  atom?(s:% := unparseInputForm(x)$Lisp) => strsym s
  concat [strsym a for a in destruct s]

parse(s:String):% ==
  ncParseFromString(s)$Lisp

declare signature ==
  declare(name := new()$Symbol, signature)$Lisp
  name

compile(name, types) ==
  symbol car cdr car
  selectLocalMms(mkProperOp name, convert(name)@%,
  types, nil$List(%)$Lisp

mkProperOp name ==
  op := mkAtree(nme := convert(name)@%)$Lisp
  transferPropsToNode(nme, op)$Lisp
  convert op

binary(op, args) ==
  (n := #args) < 2 => error "Need at least 2 arguments"
  n = 2 => convert([op, first args, last args]$List(%)
  convert([op, first args, binary(op, rest args)]$List(%)

tuplify l ==
  empty? rest l => convert first l
  conv
  concat(convert("Tuple"::Symbol), [convert x for x in l]$List(%)

function(f, l, name) ==
  nn := convert(new(1 + #l, convert(nil()$List(%)$List(%)@%

```

```

conv([convert("DEF"::Symbol), conv(cons(convert(name)@%,
[convert(x)@% for x in l])), nn, nn, f]$List(%))

s1 + s2 ==
  s1 = 0 => s2
  s2 = 0 => s1
  conv [convert("+ "::Symbol), s1, s2]$List(%)

s1 * s2 ==
  s1 = 0 or s2 = 0 => 0
  s1 = 1 => s2
  s2 = 1 => s1
  conv [convert("* "::Symbol), s1, s2]$List(%)

s1:% ** n:Integer ==
  s1 = 0 and n > 0 => 0
  s1 = 1 or zero? n => 1
--   one? n => s1
  (n = 1) => s1
  conv [convert("** "::Symbol), s1, convert n]$List(%)

s1:% ** n:NonNegativeInteger == s1 ** (n::Integer)

s1 / s2 ==
  s2 = 1 => s1
  conv [convert("/ "::Symbol), s1, s2]$List(%)

```

— INFORM.dotabb —

```

"INFORM" [color="#88FF44",href="bookvol10.3.pdf#nameddest=INFORM"]
"ALIST"  [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"INFORM" -> "ALIST"

```

domain INT Integer

The function **one?** has been rewritten back to its original form. The NAG version called a lisp primitive that exists only in Codemist Common Lisp and is not defined in Common Lisp.

— Integer.input —

```

)set break resume
)sys rm -f Integer.output
)spool Integer.output
)set message test on
)set message auto off
)clear all
--S 1 of 42
2**(5678 - 4856 + 2 * 17)
--R
--R
--R (1)
--R 4804810770435008147181540925125924391239526139871682263473855610088084200076_
--R 308293086342527091412083743074572278211496076276922026433435687527334980249_
--R 539302425425230458177649495442143929053063884787051467457680738771416988598_
--R 15495632935288783334250628775936
--R
--R Type: PositiveInteger
--E 1

--S 2 of 42
x := -101
--R
--R
--R (2) - 101
--R
--R Type: Integer
--E 2

--S 3 of 42
abs(x)
--R
--R
--R (3) 101
--R
--R Type: PositiveInteger
--E 3

--S 4 of 42
sign(x)
--R
--R
--R (4) - 1
--R
--R Type: Integer
--E 4

--S 5 of 42
x < 0
--R
--R
--R (5) true
--R
--R Type: Boolean
--E 5

```

--S 6 of 42

x <= -1

--R

--R

--R (6) true

--R

Type: Boolean

--E 6

--S 7 of 42

negative?(x)

--R

--R

--R (7) true

--R

Type: Boolean

--E 7

--S 8 of 42

x > 0

--R

--R

--R (8) false

--R

Type: Boolean

--E 8

--S 9 of 42

x >= 1

--R

--R

--R (9) false

--R

Type: Boolean

--E 9

--S 10 of 42

positive?(x)

--R

--R

--R (10) false

--R

Type: Boolean

--E 10

--S 11 of 42

zero?(x)

--R

--R

--R (11) false

--R

Type: Boolean

--E 11

--S 12 of 42

one?(x)


```
--R
--R
--R (12) false
--R
--R                                          Type: Boolean
--E 12

--S 13 of 42
(x = -101)@Boolean
--R
--R
--R (13) true
--R
--R                                          Type: Boolean
--E 13

--S 14 of 42
odd?(x)
--R
--R
--R (14) true
--R
--R                                          Type: Boolean
--E 14

--S 15 of 42
even?(x)
--R
--R
--R (15) false
--R
--R                                          Type: Boolean
--E 15

--S 16 of 42
gcd(56788,43688)
--R
--R
--R (16) 4
--R
--R                                          Type: PositiveInteger
--E 16

--S 17 of 42
lcm(56788,43688)
--R
--R
--R (17) 620238536
--R
--R                                          Type: PositiveInteger
--E 17

--S 18 of 42
max(678,567)
--R
--R
```

```
--R (18) 678
--R                                         Type: PositiveInteger
--E 18

--S 19 of 42
min(678,567)
--R
--R
--R (19) 567
--R                                         Type: PositiveInteger
--E 19

--S 20 of 42
reduce(max,[2,45,-89,78,100,-45])
--R
--R
--R (20) 100
--R                                         Type: PositiveInteger
--E 20

--S 21 of 42
reduce(min,[2,45,-89,78,100,-45])
--R
--R
--R (21) - 89
--R                                         Type: Integer
--E 21

--S 22 of 42
reduce(gcd,[2,45,-89,78,100,-45])
--R
--R
--R (22) 1
--R                                         Type: PositiveInteger
--E 22

--S 23 of 42
reduce(lcm,[2,45,-89,78,100,-45])
--R
--R
--R (23) 1041300
--R                                         Type: PositiveInteger
--E 23

--S 24 of 42
13 / 4
--R
--R
--R 13
--R (24) --
```

[illegible]

--E 30

--S 31 of 42
factor 102400

--R
--R
--R 12 2
--R (31) 2 5
--R
--E 31

Type: Factored(Integer)

--S 32 of 42
prime? 7

--R
--R
--R (32) true
--R
--E 32

Type: Boolean

--S 33 of 42
prime? 8

--R
--R
--R (33) false
--R
--E 33

Type: Boolean

--S 34 of 42
nextPrime 100

--R
--R
--R (34) 101
--R
--E 34

Type: PositiveInteger

--S 35 of 42
prevPrime 100

--R
--R
--R (35) 97
--R
--E 35

Type: PositiveInteger

--S 36 of 42
primes(100,175)

--R
--R
--R (36) [173,167,163,157,151,149,139,137,131,127,113,109,107,103,101]
--R
--E 36

Type: List(Integer)

[illegible]

)lisp (bye)

— Integer.help —

=====

Integer examples

=====

Axiom provides many operations for manipulating arbitrary precision integers. In this section we will show some of those that come from Integer itself plus some that are implemented in other packages.

\subsection{Basic Functions}

The size of an integer in Axiom is only limited by the amount of computer storage you have available. The usual arithmetic operations are available.

```
2**(5678 - 4856 + 2 * 17)
4804810770435008147181540925125924391239526139871682263473855610088084200076_
308293086342527091412083743074572278211496076276922026433435687527334980249_
539302425425230458177649495442143929053063884787051467457680738771416988598_
15495632935288783334250628775936
Type: PositiveInteger
```

There are a number of ways of working with the sign of an integer. Let's use this x as an example.

```
x := -101
- 101
Type: Integer
```

First of all, there is the absolute value function.

```
abs(x)
101
Type: PositiveInteger
```

The sign operation returns -1 if its argument is negative, 0 if zero and 1 if positive.

```
sign(x)
- 1
Type: Integer
```

You can determine if an integer is negative in several other ways.

```
x < 0
true
Type: Boolean

x <= -1
true
Type: Boolean

negative?(x)
true
Type: Boolean
```

Similarly, you can find out if it is positive.

```
x > 0
false
Type: Boolean

x >= 1
false
Type: Boolean

positive?(x)
false
Type: Boolean
```

This is the recommended way of determining whether an integer is zero.

```
zero?(x)
false
Type: Boolean
```

Use the `zero?` operation whenever you are testing any mathematical object for equality with zero. This is usually more efficient than using `=` (think of matrices: it is easier to tell if a matrix is zero by just checking term by term than constructing another "zero" matrix and comparing the two matrices term by term) and also avoids the problem that `=` is usually used for creating equations.

This is the recommended way of determining whether an integer is equal to one.

```
one?(x)
false
Type: Boolean
```

This syntax is used to test equality using `=`. It says that you want a Boolean (true or false) answer rather than an equation.

```
(x = -101)@Boolean
```

```
true
```

```
Type: Boolean
```

The operations `odd?` and `even?` determine whether an integer is odd or even, respectively. They each return a Boolean object.

```
odd?(x)
```

```
true
```

```
Type: Boolean
```

```
even?(x)
```

```
false
```

```
Type: Boolean
```

The operation `gcd` computes the greatest common divisor of two integers.

```
gcd(56788,43688)
```

```
4
```

```
Type: PositiveInteger
```

The operation `lcm` computes their least common multiple.

```
lcm(56788,43688)
```

```
620238536
```

```
Type: PositiveInteger
```

To determine the maximum of two integers, use `max`.

```
max(678,567)
```

```
678
```

```
Type: PositiveInteger
```

To determine the minimum, use `min`.

```
min(678,567)
```

```
567
```

```
Type: PositiveInteger
```

The `reduce` operation is used to extend binary operations to more than two arguments. For example, you can use `reduce` to find the maximum integer in a list or compute the least common multiple of all integers in the list.

```
reduce(max,[2,45,-89,78,100,-45])
```

```
100
```

```
Type: PositiveInteger
```

```
reduce(min,[2,45,-89,78,100,-45])
```

```
- 89
```

```
Type: Integer
```



```
reduce(gcd,[2,45,-89,78,100,-45])
1
Type: PositiveInteger
```

```
reduce(lcm,[2,45,-89,78,100,-45])
1041300
Type: PositiveInteger
```

The infix operator "/" is not used to compute the quotient of integers. Rather, it is used to create rational numbers as described in Fraction.

```
13 / 4
13
--
4
Type: Fraction Integer
```

The infix operation quo computes the integer quotient.

```
13 quo 4
3
Type: PositiveInteger
```

The infix operation rem computes the integer remainder.

```
13 rem 4
1
Type: PositiveInteger
```

One integer is evenly divisible by another if the remainder is zero. The operation exquo can also be used.

```
zero?(167604736446952 rem 2003644)
true
Type: Boolean
```

The operation divide returns a record of the quotient and remainder and thus is more efficient when both are needed.

```
d := divide(13,4)
[quotient= 3,remainder= 1]
Type: Record(quotient: Integer,remainder: Integer)

d.quotient
3
Type: PositiveInteger
```

See help on Records for details on Records.

```
d.remainder
1
```

Type: PositiveInteger

```
=====
Primes and Factorization
=====
```

Use the operation factor to factor integers. It returns an object of type Factored Integer.

```
factor 102400
12 2
2 5
```

Type: Factored Integer

The operation prime? returns true or false depending on whether its argument is a prime.

```
prime? 7
true
```

Type: Boolean

```
prime? 8
false
```

Type: Boolean

The operation nextPrime returns the least prime number greater than its argument.

```
nextPrime 100
101
```

Type: PositiveInteger

The operation prevPrime returns the greatest prime number less than its argument.

```
prevPrime 100
97
```

Type: PositiveInteger

To compute all primes between two integers (inclusively), use the operation primes.

```
primes(100,175)
[173,167,163,157,151,149,139,137,131,127,113,109,107,103,101]
```

Type: List Integer

You might sometimes want to see the factorization of an integer when it is considered a Gaussian integer.

```

factor(2 :: Complex Integer)
      2
- %i (1 + %i)
      Type: Factored Complex Integer

```

=====

Some Number Theoretic Functions

=====

Axiom provides several number theoretic operations for integers.

The operation `fibonacci` computes the Fibonacci numbers. The algorithm has running time $O(\log^3 n)$ for argument n .

```

[fibonacci(k) for k in 0..]
[0,1,1,2,3,5,8,13,21,34,...]
      Type: Stream Integer

```

The operation `legendre` computes the Legendre symbol for its two integer arguments where the second one is prime. If you know the second argument to be prime, use `jacobi` instead where no check is made.

```

[legendre(i,11) for i in 0..10]
[0,1,- 1,1,1,1,- 1,- 1,- 1,1,- 1]
      Type: List Integer

```

The operation `jacobi` computes the Jacobi symbol for its two integer arguments. By convention, 0 is returned if the greatest common divisor of the numerator and denominator is not 1.

```

[jacobi(i,15) for i in 0..9]
[0,1,1,0,1,0,0,- 1,1,0]
      Type: List Integer

```

The operation `eulerPhi` computes the values of Euler's ϕ -function where $\phi(n)$ equals the number of positive integers less than or equal to n that are relatively prime to the positive integer n .

```

[eulerPhi i for i in 1..]
[1,1,2,2,4,2,6,4,6,4,...]
      Type: Stream Integer

```

The operation `moebiusMu` computes the Moebius μ function.

```

[moebiusMu i for i in 1..]
[1,- 1,- 1,0,- 1,1,- 1,0,0,1,...]
      Type: Stream Integer

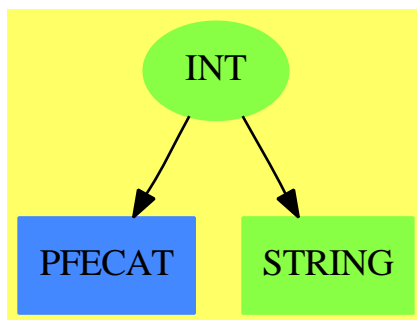
```

See Also:

- o)help Complex
- o)help Factored
- o)help Records
- o)help Fraction
- o)help RadixExpansion
- o)help HexadecimalExpansion
- o)help BinaryExpansion
- o)help DecimalExpansion
- o)help IntegerNumberTheoryFunctions
- o)help RomanNumeral
- o)show Integer

—————

Integer (INT)



See

- ⇒ “NonNegativeInteger” (NNI) [15](#) on page [1739](#)
- ⇒ “PositiveInteger” (PI) [17](#) on page [2100](#)
- ⇒ “RomanNumeral” (ROMAN) [19](#) on page [2326](#)

Exports:

0	1	abs	addmod
associates?	base	binomial	bit?
characteristic	coerce	convert	copy
D	dec	differentiate	divide
euclideanSize	even?	expressIdealMember	exquo
extendedEuclidean	extendedEuclidean	factor	factorial
gcd	gcdPolynomial	hash	inc
init	invmod	latex	lcm
length	mask	max	min
mulmod	multiEuclidean	negative?	nextItem
odd?	OMwrite	one?	patternMatch
permutation	positive?	positiveRemainder	powmod
prime?	principalIdeal	random	rational
rational?	rationalIfCan	recip	reducedSystem
retract	retractIfCan	sample	shift
sign	sizeLess?	squareFree	squareFreePart
submod	subtractIfCan	symmetricRemainder	unit?
unitCanonical	unitNormal	zero?	?*?
?**?	?+?	?-?	-?
?<?	?<=?	?=?	?>?
?>=?	?^?	?~=?	?quo?
?rem?			

— domain INT Integer —

```

)abbrev domain INT Integer
++ Author: Mark Botch
++ Description:
++ \spadtype{Integer} provides the domain of arbitrary precision integers.

Integer: Join(IntegerNumberSystem, ConvertibleTo String, OpenMath) with
  random      : % -> %
  ++ random(n) returns a random integer from 0 to \spad{n-1}.
  canonical
  ++ mathematical equality is data structure equality.
  canonicalsClosed
  ++ two positives multiply to give positive.
  noetherian
  ++ ascending chain condition on ideals.
  infinite
  ++ nextItem never returns "failed".
== add
  ZP ==> SparseUnivariatePolynomial %
  ZZP ==> SparseUnivariatePolynomial Integer
  x,y: %
  n: NonNegativeInteger

```

```

writeOMInt(dev: OpenMathDevice, x: %): Void ==
  if x < 0 then
    MputApp(dev)
    MputSymbol(dev, "arith1", "unary__minus")
    MputInteger(dev, (-x) pretend Integer)
    MputEndApp(dev)
  else
    MputInteger(dev, x pretend Integer)

OMwrite(x: %): String ==
  s: String := ""
  sp := OM_STRINGTOSTRINGPTR(s)$Lisp
  dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML)
  MputObject(dev)
  writeOMInt(dev, x)
  MputEndObject(dev)
  Mclose(dev)
  s := OM_STRINGPTRTOSTRING(sp)$Lisp pretend String
  s

OMwrite(x: %, wholeObj: Boolean): String ==
  s: String := ""
  sp := OM_STRINGTOSTRINGPTR(s)$Lisp
  dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML)
  if wholeObj then
    MputObject(dev)
  writeOMInt(dev, x)
  if wholeObj then
    MputEndObject(dev)
  Mclose(dev)
  s := OM_STRINGPTRTOSTRING(sp)$Lisp pretend String
  s

OMwrite(dev: OpenMathDevice, x: %): Void ==
  MputObject(dev)
  writeOMInt(dev, x)
  MputEndObject(dev)

OMwrite(dev: OpenMathDevice, x: %, wholeObj: Boolean): Void ==
  if wholeObj then
    MputObject(dev)
  writeOMInt(dev, x)
  if wholeObj then
    MputEndObject(dev)

zero? x == ZEROP(x)$Lisp
-- one? x == ONEP(x)$Lisp
one? x == x = 1
0 == 0$Lisp
1 == 1$Lisp

```

```

base() == 2$Lisp
copy x == x
inc x == x + 1
dec x == x - 1
hash x == SXHASH(x)$Lisp
negative? x == MINUSP(x)$Lisp
coerce(x):OutputForm == outputForm(x pretend Integer)
coerce(m:Integer):% == m pretend %
convert(x:%):Integer == x pretend Integer
length a == INTEGER-LENGTH(a)$Lisp
addmod(a, b, p) ==
  (c:=a + b) >= p => c - p
  c
submod(a, b, p) ==
  (c:=a - b) < 0 => c + p
  c
mulmod(a, b, p) == (a * b) rem p
convert(x:%):Float == coerce(x pretend Integer)$Float
convert(x:%):DoubleFloat == coerce(x pretend Integer)$DoubleFloat
convert(x:%):InputForm == convert(x pretend Integer)$InputForm
convert(x:%):String == string(x pretend Integer)$String

latex(x:%):String ==
  s : String := string(x pretend Integer)$String
  (-1 < (x pretend Integer)) and ((x pretend Integer) < 10) => s
  concat("{", concat(s, "}")$String)$String

positiveRemainder(a, b) ==
  negative?(r := a rem b) =>
    negative? b => r - b
    r + b
  r

reducedSystem(m:Matrix %):Matrix(Integer) ==
  m pretend Matrix(Integer)

reducedSystem(m:Matrix %, v:Vector %):
  Record(mat:Matrix(Integer), vec:Vector(Integer)) ==
    [m pretend Matrix(Integer), vec pretend Vector(Integer)]

abs(x) == ABS(x)$Lisp
random() == random()$Lisp
random(x) == RANDOM(x)$Lisp
x = y == EQL(x,y)$Lisp
x < y == (x<y)$Lisp
-x == (-x)$Lisp
x + y == (x+y)$Lisp
x - y == (x-y)$Lisp
x * y == (x*y)$Lisp
(m:Integer) * (y:%) == (m*y)$Lisp -- for subsumption problem

```

```

x ** n == EXPT(x,n)$Lisp
odd? x == ODDP(x)$Lisp
max(x,y) == MAX(x,y)$Lisp
min(x,y) == MIN(x,y)$Lisp
divide(x,y) == DIVIDE2(x,y)$Lisp
x quo y == QUOTIENT2(x,y)$Lisp
x rem y == REMAINDER2(x,y)$Lisp
shift(x, y) == ASH(x,y)$Lisp
x exquo y ==
  zero? y => "failed"
  zero?(x rem y) => x quo y
  "failed"
-- recip(x) == if one? x or x=-1 then x else "failed"
recip(x) == if (x = 1) or x=-1 then x else "failed"
gcd(x,y) == GCD(x,y)$Lisp
UCA ==> Record(unit:%,canonical:%,associate:%)
unitNormal x ==
  x < 0 => [-1,-x,-1]$UCA
  [1,x,1]$UCA
unitCanonical x == abs x
solveLinearPolynomialEquation(lp>List ZP,p:ZP):Union(List ZP,"failed") ==
  solveLinearPolynomialEquation(lp pretend List ZZP,
    p pretend ZZP)$IntegerSolveLinearPolynomialEquation pretend
    Union(List ZP,"failed")
squareFreePolynomial(p:ZP):Factored ZP ==
  squareFree(p)$UnivariatePolynomialSquareFree(%,ZP)
factorPolynomial(p:ZP):Factored ZP ==
  -- GaloisGroupFactorizer doesn't factor the content
  -- so we have to do this by hand
  pp:=primitivePart p
  leadingCoefficient pp = leadingCoefficient p =>
    factor(p)$GaloisGroupFactorizer(ZP)
  mergeFactors(factor(pp)$GaloisGroupFactorizer(ZP),
    map((x1:%):ZP+>->x1::ZP,
      factor((leadingCoefficient p exquo
        leadingCoefficient pp)
        ::%))$FactoredFunctions2(%,ZP)
    )$FactoredFunctionUtilities(ZP)
factorSquareFreePolynomial(p:ZP):Factored ZP ==
  factorSquareFree(p)$GaloisGroupFactorizer(ZP)
gcdPolynomial(p:ZP, q:ZP):ZP ==
  zero? p => unitCanonical q
  zero? q => unitCanonical p
  gcd([p,q])$HeuGcd(ZP)
-- myNextPrime: (%,NonNegativeInteger) -> %
-- myNextPrime(x,n) ==
--   nextPrime(x)$IntegerPrimesPackage(%)
-- TT:=InnerModularGcd(%,ZP,67108859 pretend %,myNextPrime)
-- gcdPolynomial(p,q) == modularGcd(p,q)$TT

```

— INT.dotabb —

```
"INT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=INT",
      shape=ellipse]
"PFECA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECA"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"INT" -> "STRING"
"INT" -> "PFECA"
```

domain ZMOD IntegerMod

— IntegerMod.input —

```
)set break resume
)sys rm -f IntegerMod.output
)spool IntegerMod.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntegerMod
--R IntegerMod(p: PositiveInteger) is a domain constructor
--R Abbreviation for IntegerMod is ZMOD
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ZMOD
--R
--R----- Operations -----
--R ?? : (%,%) -> %               ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R +? : (%,%) -> %               ?-? : (%,%) -> %
--R -? : % -> %                   ?=? : (%,%) -> Boolean
--R 1 : () -> %                   0 : () -> %
--R ?? : (%,PositiveInteger) -> %   coerce : Integer -> %
--R coerce : % -> OutputForm         convert : % -> Integer
--R enumerate : () -> List(%)       hash : % -> SingleInteger
--R index : PositiveInteger -> %    init : () -> %
--R latex : % -> String             lookup : % -> PositiveInteger
--R one? : % -> Boolean             random : () -> %
--R recip : % -> Union(%, "failed") sample : () -> %
--R size : () -> NonNegativeInteger zero? : % -> Boolean
```

```

--R ?~=? : (%,% ) -> Boolean
--R ?*? : (NonNegativeInteger,% ) -> %
--R ?**? : (% ,NonNegativeInteger) -> %
--R ?^? : (% ,NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R nextItem : % -> Union(%, "failed")
--R subtractIfCan : (% ,%) -> Union(%, "failed")
--R
--E 1

```

```

)spool
)lisp (bye)

```

— IntegerMod.help —

```

=====
IntegerMod examples
=====

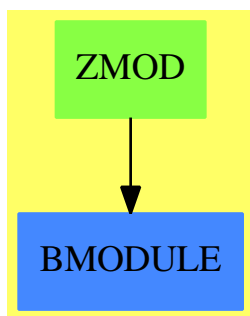
```

IntegerMod(n) creates the ring of integers reduced modulo the integer n.

See Also:

o)show IntegerMod

IntegerMod (ZMOD)



Exports:

0	1	characteristic	coerce	convert
hash	index	init	latex	lookup
nextItem	one?	random	recip	sample
size	subtractIfCan	zero?	?~=?	?*?
?**?	?^?	?+?	?-?	-?
?=?				

— domain ZMOD IntegerMod —

```

)abbrev domain ZMOD IntegerMod
++ Author: Mark Botch
++ Description:
++ IntegerMod(n) creates the ring of integers reduced modulo the integer n.

IntegerMod(p:PositiveInteger):
Join(CommutativeRing, Finite, ConvertibleTo Integer, StepThrough) == add
size() == p
characteristic() == p
lookup x == (zero? x => p; (convert(x)@Integer) :: PositiveInteger)

-- Code is duplicated for the optimizer to kick in.
if p <= convert(max())$SingleInteger@Integer then
  Rep:= SingleInteger
  q := p::SingleInteger

  bloodyCompiler: Integer -> %
  bloodyCompiler n == positiveRemainder(n, p)$Integer :: Rep

  convert(x:%):Integer == convert(x)$Rep
  coerce(x):OutputForm == coerce(x)$Rep
  coerce(n:Integer):% == bloodyCompiler n
  0 == 0$Rep
  1 == 1$Rep
  init == 0$Rep
  nextItem(n) ==
      m:=n+1
      m=0 => "failed"
      m
  x = y == x =$Rep y
  x:% * y:% == mulmod(x, y, q)
  n:Integer * x:% == mulmod(bloodyCompiler n, x, q)
  x + y == addmod(x, y, q)
  x - y == submod(x, y, q)
  random() == random(q)$Rep
  index a == positiveRemainder(a::%, q)
  - x == (zero? x => 0; q -$Rep x)

  x:% ** n:NonNegativeInteger ==

```

```

n < p => powmod(x, n::Rep, q)
powmod(convert(x)@Integer, n, p)$Integer :: Rep

recip x ==
  (c1, c2, g) := extendedEuclidean(x, q)$Rep
--   not one? g => "failed"
   not (g = 1) => "failed"
   positiveRemainder(c1, q)

else
  Rep:= Integer

  convert(x:%):Integer == convert(x)$Rep
  coerce(n:Integer):% == positiveRemainder(n::Rep, p)
  coerce(x):OutputForm == coerce(x)$Rep
  0 == 0$Rep
  1 == 1$Rep
  init == 0$Rep
  nextItem(n) ==
    m:=n+1
    m=0 => "failed"
    m
  x = y == x =$Rep y
  x:% * y:% == mulmod(x, y, p)
  n:Integer * x:% == mulmod(positiveRemainder(n::Rep, p), x, p)
  x + y == addmod(x, y, p)
  x - y == submod(x, y, p)
  random() == random(p)$Rep
  index a == positiveRemainder(a::Rep, p)
  - x == (zero? x => 0; p -$Rep x)
  x:% ** n:NonNegativeInteger == powmod(x, n::Rep, p)

  recip x ==
    (c1, c2, g) := extendedEuclidean(x, p)$Rep
--   not one? g => "failed"
   not (g = 1) => "failed"
   positiveRemainder(c1, p)

```

— ZMOD.dotabb —

"ZMOD" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ZMOD"]
 "BMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BMODULE"]
 "ZMOD" -> "BMODULE"

domain INTFTBL IntegrationFunctionsTable

— IntegrationFunctionsTable.input —

```

)set break resume
)sys rm -f IntegrationFunctionsTable.output
)spool IntegrationFunctionsTable.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntegrationFunctionsTable
--R
--R IntegrationFunctionsTable is a domain constructor
--R Abbreviation for IntegrationFunctionsTable is INTFTBL
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for INTFTBL
--R
--R----- Operations -----
--R clearTheFTable : () -> Void          showTheFTable : () -> %
--R entries : % -> List(Record(key: Record(var: Symbol,fn: Expression(DoubleFloat),range: Segmen
--R entry : Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedCompletion(Doub
--R fTable : List(Record(key: Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(Orde
--R insert! : Record(key: Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedC
--R keys : % -> List(Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedComple
--R showAttributes : Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedComple
--R
--E 1

)spool
)lisp (bye)

```

— IntegrationFunctionsTable.help —

```

=====
IntegrationFunctionsTable examples
=====

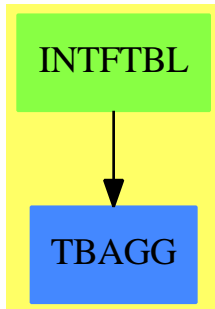
```

There is no description for this domain

See Also:

o)show IntegrationFunctionsTable

IntegrationFunctionsTable (INTFTBL)

**Exports:**

clearTheFTable entries entry fTable insert! keys showAttributes showTheFTable

— domain INTFTBL IntegrationFunctionsTable —

```

)abbrev domain INTFTBL IntegrationFunctionsTable
++ Author: Brian Dupee
++ Date Created: March 1995
++ Date Last Updated: June 1995
++ Description:
++ There is no description for this domain

IntegrationFunctionsTable(): E == I where
  EF2 ==> ExpressionFunctions2
  EFI ==> Expression Fraction Integer
  FI ==> Fraction Integer
  LEDF ==> List Expression DoubleFloat
  KEDF ==> Kernel Expression DoubleFloat
  EEDF ==> Equation Expression DoubleFloat
  EDF ==> Expression DoubleFloat
  PDF ==> Polynomial DoubleFloat
  LDF ==> List DoubleFloat
  SDF ==> Stream DoubleFloat
  DF ==> DoubleFloat
  F ==> Float
  ST ==> String
  LST ==> List String
  SI ==> SingleInteger
  SOCDF ==> Segment OrderedCompletion DoubleFloat
  OCDF ==> OrderedCompletion DoubleFloat
  OCEDF ==> OrderedCompletion Expression DoubleFloat
  EOCEFI ==> Equation OrderedCompletion Expression Fraction Integer
  OCEFI ==> OrderedCompletion Expression Fraction Integer
  OCFI ==> OrderedCompletion Fraction Integer
  NIA ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
  
```

```

INT ==> Integer
CTYPE ==> Union(continuous: "Continuous at the end points",
    lowerSingular: "There is a singularity at the lower end point",
    upperSingular: "There is a singularity at the upper end point",
    bothSingular: "There are singularities at both end points",
    notEvaluated: "End point continuity not yet evaluated")
RTYPE ==> Union(finite: "The range is finite",
    lowerInfinite: "The bottom of range is infinite",
    upperInfinite: "The top of range is infinite",
    bothInfinite: "Both top and bottom points are infinite",
    notEvaluated: "Range not yet evaluated")
STYPE ==> Union(str:SDF,
    notEvaluated:"Internal singularities not yet evaluated")
ATT ==> Record(endPointContinuity:CTYPE,
    singularitiesStream:STYPE,range:RTYPE)
ROA ==> Record(key:NIA,entry:ATT)

E ==> with

    showTheFTable:() -> $
    ++ showTheFTable() returns the current table of functions.
    clearTheFTable : () -> Void
    ++ clearTheFTable() clears the current table of functions.
    keys : $ -> List(NIA)
    ++ keys(f) returns the list of keys of f
    fTable: List Record(key:NIA,entry:ATT) -> $
    ++ fTable(l) creates a functions table from the elements of l.
    insert!:Record(key:NIA,entry:ATT) -> $
    ++ insert!(r) inserts an entry r into theFTable
    showAttributes:NIA -> Union(ATT,"failed")
    ++ showAttributes(x) is not documented
    entries : $ -> List Record(key:NIA,entry:ATT)
    ++ entries(x) is not documented
    entry:NIA -> ATT
    ++ entry(n) is not documented
I ==> add

Rep := Table(NIA,ATT)
import Rep

theFTable:$ := empty()$Rep

showTheFTable():$ ==
    theFTable

clearTheFTable():Void ==
    theFTable := empty()$Rep
    void()$Void

fTable(l:List Record(key:NIA,entry:ATT)):$ ==

```

```

theFTable := table(1)$Rep

insert!(r:Record(key:NIA,entry:ATT)):$ ==
  insert!(r,theFTable)$Rep

keys(t:$):List NIA ==
  keys(t)$Rep

showAttributes(k:NIA):Union(ATT,"failed") ==
  search(k,theFTable)$Rep

entries(t:$):List Record(key:NIA,entry:ATT) ==
  members(t)$Rep

entry(k:NIA):ATT ==
  qelt(theFTable,k)$Rep

```

— INTFTBL.dotabb —

```

"INTFTBL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=INTFTBL"]
"TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]
"INTFTBL" -> "TBAGG"

```

domain IR IntegrationResult

— IntegrationResult.input —

```

)set break resume
)sys rm -f IntegrationResult.output
)spool IntegrationResult.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntegrationResult
--R
--R IntegrationResult(F: Field) is a domain constructor
--R Abbreviation for IntegrationResult is IR
--R This constructor is not exposed in this frame.

```



```

--R Issue )edit bookvol10.3.pamphlet to see algebra source code for IR
--R
--R----- Operations -----
--R ?? : (% , Fraction(Integer)) -> %      ?? : (Fraction(Integer), %) -> %
--R ?? : (Integer, %) -> %                  ?? : (PositiveInteger, %) -> %
--R +? : (% , %) -> %                      ?-? : (% , %) -> %
--R -? : % -> %                             ?=? : (% , %) -> Boolean
--R 0 : () -> %                             coerce : F -> %
--R coerce : % -> OutputForm                differentiate : (% , (F -> F)) -> F
--R elem? : % -> Boolean                    hash : % -> SingleInteger
--R integral : (F, F) -> %                  latex : % -> String
--R ratpart : % -> F                        retract : % -> F
--R sample : () -> %                       zero? : % -> Boolean
--R ?^=? : (% , %) -> Boolean
--R ?? : (NonNegativeInteger, %) -> %
--R differentiate : (% , Symbol) -> F if F has PDRING(SYMBOL)
--R integral : (F, Symbol) -> % if F has RETRACT(SYMBOL)
--R logpart : % -> List(Record(scalar: Fraction(Integer), coeff: SparseUnivariatePolynomial(F), log
--R mkAnswer : (F, List(Record(scalar: Fraction(Integer), coeff: SparseUnivariatePolynomial(F), log
--R notelem : % -> List(Record(integrand: F, intvar: F))
--R retractIfCan : % -> Union(F, "failed")
--R subtractIfCan : (% , %) -> Union(% , "failed")
--R
--E 1

)spool
)lisp (bye)

```

— IntegrationResult.help —

=====

IntegrationResult examples

=====

The result of a transcendental integration.

If a function f has an elementary integral g , then g can be written in the form

$$g = h + c_1 \log(u_1) + c_2 \log(u_2) + \dots + c_n \log(u_n)$$

where h , which is in the same field than f , is called the rational part of the integral, and

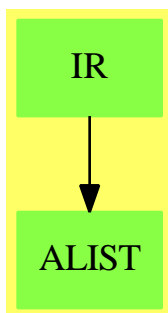
$$c_1 \log(u_1) + \dots + c_n \log(u_n)$$

is called the logarithmic part of the integral. This domain manipulates integrals represented in that form, by keeping both parts separately. The logs are not explicitly computed.

See Also:

```
o )show IntegrationResult
```

IntegrationResult (IR)



Exports:

0	coerce	differentiate	elem?	hash
integral	latex	logpart	mkAnswer	notelem
ratpart	retract	retractIfCan	subtractIfCan	sample
zero?	?~=?	?*?	?+?	?-?
-?	?=?			

— domain IR IntegrationResult —

```
)abbrev domain IR IntegrationResult
++ Author: Barry Trager, Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 12 August 1992
++ Description:
++ The result of a transcendental integration.
++ If a function f has an elementary integral g, then g can be written
++ in the form \spad{g = h + c1 log(u1) + c2 log(u2) + ... + cn log(un)}
++ where h, which is in the same field than f, is called the rational
++ part of the integral, and \spad{c1 log(u1) + ... cn log(un)} is called the
++ logarithmic part of the integral. This domain manipulates integrals
++ represented in that form, by keeping both parts separately. The logs
++ are not explicitly computed.
```

```
IntegrationResult(F:Field): Exports == Implementation where
```

```
  O ==> OutputForm
  B ==> Boolean
  Z ==> Integer
  Q ==> Fraction Integer
  SE ==> Symbol
```

```

UP ==> SparseUnivariatePolynomial F
LOG ==> Record(scalar:Q, coeff:UP, logand:UP)
NE ==> Record(integrand:F, intvar:F)

Exports ==> (Module Q, RetractableTo F) with
mkAnswer: (F, List LOG, List NE) -> %
  ++ mkAnswer(r,l,ne) creates an integration result from
  ++ a rational part r, a logarithmic part l, and a non-elementary part ne.
ratpart : % -> F
  ++ ratpart(ir) returns the rational part of an integration result
logpart : % -> List LOG
  ++ logpart(ir) returns the logarithmic part of an integration result
notelem : % -> List NE
  ++ notelem(ir) returns the non-elementary part of an integration result
elem? : % -> B
  ++ elem?(ir) tests if an integration result is elementary over F?
integral: (F, F) -> %
  ++ integral(f,x) returns the formal integral of f with respect to x
differentiate: (% , F -> F) -> F
  ++ differentiate(ir,D) differentiates ir with respect to the derivation D.
if F has PartialDifferentialRing(SE) then
  differentiate: (% , Symbol) -> F
  ++ differentiate(ir,x) differentiates ir with respect to x
if F has RetractableTo Symbol then
  integral: (F, Symbol) -> %
  ++ integral(f,x) returns the formal integral of f with respect to x

Implementation ==> add
Rep := Record(ratp: F, logp: List LOG, nelem: List NE)

timelog : (Q, LOG) -> LOG
timene : (Q, NE) -> NE
LOG2O : LOG -> O
NE2O : NE -> O
Q2F : Q -> F
nesimp : List NE -> List NE
neselect: (List NE, F) -> F
pLogDeriv: (LOG, F -> F) -> F
pNeDeriv : (NE, F -> F) -> F

alpha:O := new()$Symbol :: O

- u == (-1$Z) * u
0 == mkAnswer(0, empty(), empty())
coerce(x:F):% == mkAnswer(x, empty(), empty())
ratpart u == u.ratp
logpart u == u.logp
notelem u == u.nelem
elem? u == empty? notelem u

```

```

mkAnswer(x, l, n) == [x, l, nesimp n]
timelog(r, lg)    == [r * lg.scalar, lg.coeff, lg.logand]
integral(f:F,x:F) == (zero? f => 0; mkAnswer(0, empty(), [[f, x]]))
timene(r, ne)     == [Q2F(r) * ne.integrand, ne.intvar]
n:Z * u:%         == (n::Q) * u
Q2F r             == numer(r)::F / denom(r)::F
neselect(l, x)    == _+/[ne.integrand for ne in l | ne.intvar = x]

if F has RetractableTo Symbol then
  integral(f:F, x:Symbol):% == integral(f, x:F)

LOG20 rec ==
--
  one? degree rec.coeff =>
    (degree rec.coeff) = 1 =>
      -- deg 1 minimal poly doesn't get sigma
      lastc := - coefficient(rec.coeff, 0) / coefficient(rec.coeff, 1)
      lg    := (rec.logand) lastc
      logandp := prefix("log"::Symbol::0, [lg::0])
      (cc := Q2F(rec.scalar) * lastc) = 1 => logandp
      cc = -1 => - logandp
      cc::0 * logandp
      coeffp:0 := (outputForm(rec.coeff, alpha) = 0::Z::0)@0
      logandp :=
        alpha * prefix("log"::Symbol::0, [outputForm(rec.logand, alpha)])
      if (cc := Q2F(rec.scalar)) ^= 1 then
        logandp := cc::0 * logandp
      sum(logandp, coeffp)

nesimp l ==
  [[u,x] for x in removeDuplicates_!([ne.intvar for ne in l]$List(F))
    | (u := neselect(l, x)) ^= 0]

if (F has LiouvillianFunctionCategory) and (F has RetractableTo Symbol) then
  retractIfCan u ==
    empty? logpart u =>
      ratpart u +
        _+/[integral(ne.integrand, retract(ne.intvar)@Symbol)$F
          for ne in notelem u]
    "failed"

else
  retractIfCan u ==
    elem? u and empty? logpart u => ratpart u
    "failed"

r:Q * u:% ==
  r = 0 => 0
  mkAnswer(Q2F(r) * ratpart u, map(x1+>timelog(r, x1), logpart u),
    map(x2+>timene(r, x2), notelem u))

```

```

-- Initial attempt, quick and dirty, no simplification
u + v ==
  mkAnswer(ratpart u + ratpart v, concat(logpart u, logpart v),
            nesimp concat(notelem u, notelem v))

if F has PartialDifferentialRing(Symbol) then
  differentiate(u:%, x:Symbol):F ==
    differentiate(u, x1+>differentiate(x1, x))

differentiate(u:%, derivation:F -> F):F ==
  derivation ratpart u +
    _+/[pLogDeriv(log, derivation) for log in logpart u]
    + _+/[pNeDeriv(ne, derivation) for ne in notelem u]

pNeDeriv(ne, derivation) ==
--   one? derivation(ne.intvar) => ne.integrand
  (derivation(ne.intvar) = 1) => ne.integrand
  zero? derivation(ne.integrand) => 0
  error "pNeDeriv: cannot differentiate not elementary part into F"

pLogDeriv(log, derivation) ==
  map(derivation, log.coeff) ^= 0 =>
    error "pLogDeriv: can only handle logs with constant coefficients"
--   one?(n := degree(log.coeff)) =>
  ((n := degree(log.coeff)) = 1) =>
    c := - (leadingCoefficient reductum log.coeff)
           / (leadingCoefficient log.coeff)

    ans := (log.logand) c
    Q2F(log.scalar) * c * derivation(ans) / ans
  numlog := map(derivation, log.logand)
  diflog := extendedEuclidean(log.logand, log.coeff,
                               numlog)::Record(coef1:UP, coef2:UP)

  algans := diflog.coef1
  ans:F := 0
  for i in 0..(n-1) repeat
    algans := algans * monomial(1, 1) rem log.coeff
    ans := ans + coefficient(algans, i)
  Q2F(log.scalar) * ans

coerce(u:%):0 ==
  (r := retractIfCan u) case F => r::F::0
  l := reverse_! [LOG20 f for f in logpart u]$List(0)
  if ratpart u ^= 0 then l := concat(ratpart(u)::0, l)
  if not elem? u then l := concat([NE20 f for f in notelem u], l)
  null l => 0::0
  reduce("+", l)

NE20 ne ==
  int((ne.integrand)::0 * hconcat ["d "::Symbol::0, (ne.intvar)::0])

```

— IR.dotabb —

— Interval.input —

Type: Interval(Float)

Type: Interval(Float)

Type: Interval(Float)

```

--S 4 of 13
t4:=1/4*t2
--R
--R
--R      [1.0,1.0]
--R      (4)  -----
--R      [4.0,4.0]
--R
--R                                          Type: Fraction(Interval(Float))
--E 4

--S 5 of 13
acos(t3)
--R
--R
--R      (5)  [1.5707963267 948966192,1.5707963267 948966192]
--R
--R                                          Type: Interval(Float)
--E 5

--S 6 of 13
t6:=t4*t4
--R
--R
--R      [1.0,1.0]
--R      (6)  -----
--R      [16.0,16.0]
--R
--R                                          Type: Fraction(Interval(Float))
--E 6

--S 7 of 13
sup(t6)
--R
--R
--R      (7)  0.0625000000 0000000000 1
--R
--R                                          Type: Float
--E 7

--S 8 of 13
inf(t6)
--R
--R
--R      (8)  0.0624999999 9999999999 9
--R
--R                                          Type: Float
--E 8

--S 9 of 13
width(t6)
--R
--R
--R      (9)  0.1 E -20
--R
--R                                          Type: Float

```

--E 9

--S 10 of 13

positive? t3

--R

--R

--R (10) false

--R

Type: Boolean

--E 10

--S 11 of 13

negative? t3

--R

--R

--R (11) false

--R

Type: Boolean

--E 11

--S 12 of 13

contains?(t3,0.3)

--R

--R

--R (12) false

--R

Type: Boolean

--E 12

--S 13 of 13

)show Interval

--R

--R Interval(R: Join(FloatingPointSystem,TranscendentalFunctionCategory)) is a domain constructor

--R Abbreviation for Interval is INTRVL

--R This constructor is exposed in this frame.

--R Issue)edit bookvol10.3.pamphlet to see algebra source code for INTRVL

--R

--R----- Operations -----

--R ?? : (%,%) -> %

?? : (Integer,%) -> %

--R ?? : (PositiveInteger,%) -> %

?? : (%,%) -> %

--R ?? : (% ,PositiveInteger) -> %

?+? : (%,%) -> %

--R ?-? : (%,%) -> %

-? : % -> %

--R ?<? : (%,%) -> Boolean

?<=? : (%,%) -> Boolean

--R ?=? : (%,%) -> Boolean

?>? : (%,%) -> Boolean

--R ?>=? : (%,%) -> Boolean

1 : () -> %

--R 0 : () -> %

?^? : (% ,PositiveInteger) -> %

--R acos : % -> %

acosh : % -> %

--R acot : % -> %

acoth : % -> %

--R acsc : % -> %

acsch : % -> %

--R asec : % -> %

asech : % -> %

--R asin : % -> %

asinh : % -> %

--R associates? : (%,%) -> Boolean

atan : % -> %

--R atanh : % -> %

coerce : Integer -> %


```

--R coerce : % -> %
--R coerce : % -> OutputForm
--R cos : % -> %
--R cot : % -> %
--R csc : % -> %
--R exp : % -> %
--R gcd : (% , %) -> %
--R inf : % -> R
--R interval : R -> %
--R latex : % -> String
--R lcm : (% , %) -> %
--R max : (% , %) -> %
--R negative? : % -> Boolean
--R one? : % -> Boolean
--R positive? : % -> Boolean
--R recip : % -> Union(% , "failed")
--R sample : () -> %
--R sech : % -> %
--R sinh : % -> %
--R sup : % -> R
--R tanh : % -> %
--R unitCanonical : % -> %
--R zero? : % -> Boolean
--R ?? : (NonNegativeInteger , %) -> %
--R ??? : (% , Fraction(Integer)) -> %
--R ??? : (% , NonNegativeInteger) -> %
--R ?? : (% , NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R exquo : (% , %) -> Union(% , "failed")
--R gcdPolynomial : (SparseUnivariatePolynomial(% ) , SparseUnivariatePolynomial(% )) -> SparseUnivariatePolynomial(% )
--R retractIfCan : % -> Union(Integer , "failed")
--R subtractIfCan : (% , %) -> Union(% , "failed")
--R unitNormal : % -> Record(unit : % , canonical : % , associate : %)
--R
--E 13

)spool
)lisp (bye)

```

— Interval.help —

Interval examples

This domain is an implementation of interval arithmetic and transcendental

functions over intervals.

```
t1:=0::Interval(Float)
  [0.0,0.0]
```

```
t2:=1::Interval(Float)
  [1.0,1.0]
```

```
t3:=3*t1
  [- 0.3 E -20,0.3 E -20]
```

```
t4:=1/4*t2
  [1.0,1.0]
  -----
  [4.0,4.0]
```

```
acos(t3)
  [1.5707963267 948966192,1.5707963267 948966192]
```

```
t6:=t4*t4
  [1.0,1.0]
  -----
  [16.0,16.0]
```

```
sup(t6)
  0.0625000000 0000000000 1
```

```
inf(t6)
  0.0624999999 9999999999 9
```

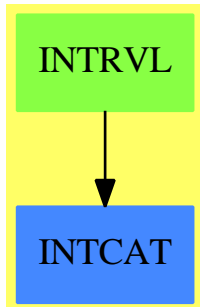
```
width(t6)
  0.1 E -20
```

```
positive? t3
  false
```

```
negative? t3
  false
```

```
contains?(t3,0.3)
  false
```

Interval (INTRVL)

**Exports:**

0	1	acos	acosh	acot
acoth	acsc	acsch	asec	asech
asin	asinh	associates?	atan	atanh
characteristic	coerce	contains?	cos	cosh
cot	coth	csc	csch	exp
exquo	gcd	gcdPolynomial	hash	inf
interval	latex	lcm	log	max
min	negative?	nthRoot	one?	pi
positive?	qinterval	recip	retract	retractIfCan
sample	sec	sech	sin	sinh
sqrt	subtractIfCan	sup	tan	tanh
unit?	unitCanonical	unitNormal	width	zero?
?*?	?**?	?+?	?-?	-?
?<?	?<=?	?=?	?>?	?>=?
?^?	?~=?			

— domain INTRVL Interval —

```

)abbrev domain INTRVL Interval
++ Author: Mike Dewar
++ Date Created: November 1996
++ Description:
++ This domain is an implementation of interval arithmetic and transcendental
++ functions over intervals.

```

```

Interval(R:Join(FloatingPointSystem,TranscendentalFunctionCategory)): IntervalCategory(R) == add

```

```

    import Integer
--    import from R

    Rep := Record(Inf:R, Sup:R)

    roundDown(u:R):R ==

```

```

    if zero?(u) then float(-1,-(bits())@Integer))
      else float(mantissa(u) - 1,exponent(u))

roundUp(u:R):R ==
  if zero?(u) then float(1, -(bits())@Integer)
    else float(mantissa(u) + 1,exponent(u))

-- Sometimes the float representation does not use all the bits (e.g. when
-- representing an integer in software using arbitrary-length Integers as
-- your mantissa it is convenient to keep them exact). This function
-- normalises things so that rounding etc. works as expected. It is only
-- called when creating new intervals.
normaliseFloat(u:R):R ==
  zero? u => u
  m : Integer := mantissa u
  b : Integer := bits()@Integer
  l : Integer := length(m)
  if l < b then
    BASE : Integer := base()$R@Integer
    float(m*BASE**((b-l) pretend PositiveInteger),exponent(u)-b+1)
  else
    u

interval(i:R,s:R):% ==
  i > s => [roundDown normaliseFloat s,roundUp normaliseFloat i]
  [roundDown normaliseFloat i,roundUp normaliseFloat s]

interval(f:R):% ==
  zero?(f) => 0
  one?(f) => 1
  -- This next part is necessary to allow e.g. mapping between Expressions:
  -- AXIOM assumes that Integers stay as Integers!
--   import from Union(value1:Integer,failed:"failed")
  fnew : R := normaliseFloat f
  retractIfCan(f)@Union(Integer,"failed") case "failed" =>
    [roundDown fnew, roundUp fnew]
  [fnew,fnew]

qinterval(i:R,s:R):% ==
  [roundDown normaliseFloat i,roundUp normaliseFloat s]

exactInterval(i:R,s:R):% == [i,s]
exactSupInterval(i:R,s:R):% == [roundDown i,s]
exactInfInterval(i:R,s:R):% == [i,roundUp s]

inf(u:%):R == u.Inf
sup(u:%):R == u.Sup
width(u:%):R == u.Sup - u.Inf

contains?(u:%,f:R):Boolean == (f > inf(u)) and (f < sup(u))

```

```

positive?(u:%):Boolean == inf(u) > 0
negative?(u:%):Boolean == sup(u) < 0

_< (a:%,b:%):Boolean ==
  if inf(a) < inf(b) then
    true
  else if inf(a) > inf(b) then
    false
  else
    sup(a) < sup(b)

_+ (a:%,b:%):% ==
  -- A couple of blatant hacks to preserve the Ring Axioms!
  if zero?(a) then return(b) else if zero?(b) then return(a)
  if a = b then return qinterval(2*inf(a),2*sup(a))
  qinterval(inf(a) + inf(b), sup(a) + sup(b))

_- (a:%,b:%):% ==
  if zero?(a) then return(-b) else if zero?(b) then return(a)
  if a = b then 0 else qinterval(inf(a) - sup(b), sup(a) - inf(b))

_* (a:%,b:%):% ==
  -- A couple of blatant hacks to preserve the Ring Axioms!
  if one?(a) then return(b) else if one?(b) then return(a)
  if zero?(a) then return(0) else if zero?(b) then return(0)
  prods : List R := sort [inf(a)*inf(b),sup(a)*sup(b),
                          inf(a)*sup(b),sup(a)*inf(b)]
  qinterval(first prods, last prods)

_* (a:Integer,b:%):% ==
  if (a > 0) then
    qinterval(a*inf(b),a*sup(b))
  else if (a < 0) then
    qinterval(a*sup(b),a*inf(b))
  else
    0

_* (a:PositiveInteger,b:%):% == qinterval(a*inf(b),a*sup(b))

_** (a:%,n:PositiveInteger):% ==
  contains?(a,0) and zero?((n@Integer) rem 2) =>
    interval(0,max(inf(a)**n,sup(a)**n))
  interval(inf(a)**n,sup(a)**n)

_^ (a:%,n:PositiveInteger):% ==

```

```

contains?(a,0) and zero?((n@Integer) rem 2) =>
  interval(0,max(inf(a)**n,sup(a)**n))
interval(inf(a)**n,sup(a)**n)

_- (a:%):% == exactInterval(-sup(a),-inf(a))

_ = (a:%,b:%):Boolean == (inf(a)=inf(b)) and (sup(a)=sup(b))
_~_ = (a:%,b:%):Boolean == (inf(a)~=inf(b)) or (sup(a)~=sup(b))

1 ==
  one : R := normaliseFloat 1
  [one,one]

0 == [0,0]

recip(u:%):Union(%, "failed") ==
  contains?(u,0) => "failed"
vals:List R := sort [1/inf(u),1/sup(u)]$List(R)
qinterval(first vals, last vals)

unit?(u:%):Boolean == contains?(u,0)

_exquo(u:%,v:%):Union(%, "failed") ==
  contains?(v,0) => "failed"
  one?(v) => u
  u=v => 1
  u=-v => -1
  vals:List R := sort [inf(u)/inf(v),inf(u)/sup(v),sup(u)/inf(v),sup(u)/sup(v)]$List(R)
  qinterval(first vals, last vals)

gcd(u:%,v:%):% == 1

coerce(u:Integer):% ==
  ur := normaliseFloat(u::R)
  exactInterval(ur,ur)

interval(u:Fraction Integer):% ==
--   import    log2 : % -> %
--           coerce : Integer -> %
--   retractIfCan : % -> Union(value1:Integer,failed:"failed")
--   from Float
  flt := u::R

-- Test if the representation in R is exact
--den := denom(u)::Float
bin : Union(Integer,"failed") := retractIfCan(log2(denom(u)::Float))
bin case Integer and length(numer u)$Integer < (bits())@Integer =>

```

```

    flt := normaliseFloat flt
    exactInterval(flt,flt)

qinterval(flt,flt)

retractIfCan(u:%):Union(Integer,"failed") ==
  not zero? width(u) => "failed"
  retractIfCan inf u

retract(u:%):Integer ==
  not zero? width(u) =>
    error "attempt to retract a non-Integer interval to an Integer"
  retract inf u

coerce(u:%):OutputForm ==
  bracket([coerce inf(u), coerce sup(u)]$List(OutputForm))

characteristic():NonNegativeInteger == 0

-- Explicit export from TranscendentalFunctionCategory
pi():% == qinterval(pi(),pi())

-- From ElementaryFunctionCategory
log(u:%):% ==
  positive?(u) => qinterval(log inf u, log sup u)
  error "negative logs in interval"

exp(u:%):% == qinterval(exp inf u, exp sup u)

_ *_ (u:%,v:%):% ==
  zero?(v) => if zero?(u) then error "0**0 is undefined" else 1
  one?(u)  => 1
  expts : List R := sort [inf(u)**inf(v),sup(u)**sup(v),
                        inf(u)**sup(v),sup(u)**inf(v)]
  qinterval(first expts, last expts)

-- From TrigonometricFunctionCategory

-- This function checks whether an interval contains a value of the form
-- 'offset + 2 n pi'.
hasTwoPiMultiple(offset:R,ipi:R,i:%):Boolean ==
  next : Integer := retract ceiling( (inf(i) - offset)/(2*ipi) )
  contains?(i,offset+2*next*ipi)

```

```

-- This function checks whether an interval contains a value of the form
-- 'offset + n pi'.
hasPiMultiple(offset:R,ipi:R,i:%):Boolean ==
  next : Integer := retract ceiling( (inf(i) - offset)/ipi )
  contains?(i,offset+next*ipi)

sin(u:%):% ==
  ipi : R := pi()$R
  hasOne? : Boolean := hasTwoPiMultiple(ipi/(2:R),ipi,u)
  hasMinusOne? : Boolean := hasTwoPiMultiple(3*ipi/(2:R),ipi,u)

  if hasOne? and hasMinusOne? then
    exactInterval(-1,1)
  else
    vals : List R := sort [sin inf u, sin sup u]
    if hasOne? then
      exactSupInterval(first vals, 1)
    else if hasMinusOne? then
      exactInfInterval(-1,last vals)
    else
      qinterval(first vals, last vals)

cos(u:%):% ==
  ipi : R := pi()
  hasOne? : Boolean := hasTwoPiMultiple(0,ipi,u)
  hasMinusOne? : Boolean := hasTwoPiMultiple(ipi,ipi,u)

  if hasOne? and hasMinusOne? then
    exactInterval(-1,1)
  else
    vals : List R := sort [cos inf u, cos sup u]
    if hasOne? then
      exactSupInterval(first vals, 1)
    else if hasMinusOne? then
      exactInfInterval(-1,last vals)
    else
      qinterval(first vals, last vals)

tan(u:%):% ==
  ipi : R := pi()
  if width(u) > ipi then
    error "Interval contains a singularity"
  else
    -- Since we know the interval is less than pi wide, monotonicity implies
    -- that there is no singularity. If there is a singularity on a endpoint

```



```

-- of the interval the user will see the error generated by R.
lo : R := tan inf u
hi : R := tan sup u

lo > hi => error "Interval contains a singularity"
qinterval(lo,hi)

csc(u:R):R ==
  ipi : R := pi()
  if width(u) > ipi then
    error "Interval contains a singularity"
  else
--    import from Integer
--    singularities are at multiples of Pi
    if hasPiMultiple(0,ipi,u) then error "Interval contains a singularity"
    vals : List R := sort [csc inf u, csc sup u]
    if hasTwoPiMultiple(ipi/(2::R),ipi,u) then
      exactInfInterval(1,last vals)
    else if hasTwoPiMultiple(3*ipi/(2::R),ipi,u) then
      exactSupInterval(first vals,-1)
    else
      qinterval(first vals, last vals)

sec(u:R):R ==
  ipi : R := pi()
  if width(u) > ipi then
    error "Interval contains a singularity"
  else
--    import from Integer
--    singularities are at Pi/2 + n Pi
    if hasPiMultiple(ipi/(2::R),ipi,u) then
      error "Interval contains a singularity"
    vals : List R := sort [sec inf u, sec sup u]
    if hasTwoPiMultiple(0,ipi,u) then
      exactInfInterval(1,last vals)
    else if hasTwoPiMultiple(ipi,ipi,u) then
      exactSupInterval(first vals,-1)
    else
      qinterval(first vals, last vals)

cot(u:R):R ==
  ipi : R := pi()
  if width(u) > ipi then

```

```

    error "Interval contains a singularity"
  else
    -- Since we know the interval is less than pi wide, monotonicity implies
    -- that there is no singularity.  If there is a singularity on a endpoint
    -- of the interval the user will see the error generated by R.
    hi : R := cot inf u
    lo : R := cot sup u

    lo > hi => error "Interval contains a singularity"
    qinterval(lo,hi)

-- From ArcTrigonometricFunctionCategory

asin(u:%):% ==
  lo : R := inf(u)
  hi : R := sup(u)
  if (lo < -1) or (hi > 1) then error "asin only defined on the region -1..1"
  qinterval(asin lo,asin hi)

acos(u:%):% ==
  lo : R := inf(u)
  hi : R := sup(u)
  if (lo < -1) or (hi > 1) then error "acos only defined on the region -1..1"
  qinterval(acos hi,acos lo)

atan(u:%):% == qinterval(atan inf u, atan sup u)

acot(u:%):% == qinterval(acot sup u, acot inf u)

acsc(u:%):% ==
  lo : R := inf(u)
  hi : R := sup(u)
  if ((lo <= -1) and (hi >= -1)) or ((lo <= 1) and (hi >= 1)) then
    error "acsc not defined on the region -1..1"
  qinterval(acsc hi, acsc lo)

asec(u:%):% ==
  lo : R := inf(u)
  hi : R := sup(u)
  if ((lo < -1) and (hi > -1)) or ((lo < 1) and (hi > 1)) then
    error "asec not defined on the region -1..1"
  qinterval(asec lo, asec hi)

-- From HyperbolicFunctionCategory

```

```

tanh(u:%) := qinterval(tanh inf u, tanh sup u)

sinh(u:%) := qinterval(sinh inf u, sinh sup u)

sech(u:%) :=
  negative? u => qinterval(sech inf u, sech sup u)
  positive? u => qinterval(sech sup u, sech inf u)
  vals : List R := sort [sech inf u, sech sup u]
  exactSupInterval(first vals, 1)

cosh(u:%) :=
  negative? u => qinterval(cosh sup u, cosh inf u)
  positive? u => qinterval(cosh inf u, cosh sup u)
  vals : List R := sort [cosh inf u, cosh sup u]
  exactInfInterval(1, last vals)

csch(u:%) :=
  contains?(u, 0) => error "csch: singularity at zero"
  qinterval(csch sup u, csch inf u)

coth(u:%) :=
  contains?(u, 0) => error "coth: singularity at zero"
  qinterval(coth sup u, coth inf u)

-- From ArchHyperbolicFunctionCategory

acosh(u:%) :=
  inf(u) < 1 => error "invalid argument: acosh only defined on the region 1.."
  qinterval(acosh inf u, acosh sup u)

acoth(u:%) :=
  lo : R := inf(u)
  hi : R := sup(u)
  if ((lo <= -1) and (hi >= -1)) or ((lo <= 1) and (hi >= 1)) then
    error "acoth not defined on the region -1..1"
  qinterval(acoth hi, acoth lo)

acsch(u:%) :=
  contains?(u, 0) => error "acsch: singularity at zero"
  qinterval(acsch sup u, acsch inf u)

asech(u:%) :=

```

```

lo : R := inf(u)
hi : R := sup(u)
if (lo <= 0) or (hi > 1) then
  error "asech only defined on the region 0 < x <= 1"
qinterval(asech hi, asech lo)

asinh(u:%) := qinterval(asinh inf u, asinh sup u)

atanh(u:%) :=
  lo : R := inf(u)
  hi : R := sup(u)
  if (lo <= -1) or (hi >= 1) then
    error "atanh only defined on the region -1 < x < 1"
  qinterval(atanh lo, atanh hi)

-- From RadicalCategory
_.*_ (u:%, n:Fraction Integer) := interval(inf(u)**n, sup(u)**n)

```

— INTRVL.dotabb —

```

"INTRVL" [color="#88FF44", href="bookvol10.3.pdf#nameddest=INTRVL"]
"INTCAT" [color="#4488FF", href="bookvol10.2.pdf#nameddest=INTCAT"]
"INTRVL" -> "INTCAT"

```

Chapter 11

Chapter J

Chapter 12

Chapter K

domain KERNEL Kernel

— Kernel.input —

```
)set break resume
)sys rm -f Kernel.output
)spool Kernel.output
)set message test on
)set message auto off
)clear all
--S 1 of 19
x :: Expression Integer
--R
--R
--R (1) x
--R
--R                                          Type: Expression(Integer)
--E 1

--S 2 of 19
kernel x
--R
--R
--R (2) x
--R
--R                                          Type: Kernel(Expression(Integer))
--E 2

--S 3 of 19
sin(x) + cos(x)
--R
--R
--R (3) sin(x) + cos(x)
```


[illegible]

--E 9

--S 10 of 19

height mainKernel(sin x)

--R

--R

--R (10) 2

--R

Type: PositiveInteger

--E 10

--S 11 of 19

height mainKernel(sin cos x)

--R

--R

--R (11) 3

--R

Type: PositiveInteger

--E 11

--S 12 of 19

height mainKernel(sin cos (tan x + sin x))

--R

--R

--R (12) 4

--R

Type: PositiveInteger

--E 12

--S 13 of 19

operator mainKernel(sin cos (tan x + sin x))

--R

--R

--R (13) sin

--R

Type: BasicOperator

--E 13

--S 14 of 19

name mainKernel(sin cos (tan x + sin x))

--R

--R

--R (14) sin

--R

Type: Symbol

--E 14

--S 15 of 19

f := operator 'f

--R

--R

--R (15) f

--R

Type: BasicOperator

--E 15

```

--S 16 of 19
e := f(x, y, 10)
--R
--R
--R (16) f(x,y,10)
--R
--R                                          Type: Expression(Integer)
--E 16

--S 17 of 19
is?(e, f)
--R
--R
--R (17) true
--R
--R                                          Type: Boolean
--E 17

--S 18 of 19
is?(e, 'f)
--R
--R
--R (18) true
--R
--R                                          Type: Boolean
--E 18

--S 19 of 19
argument mainKernel e
--R
--R
--R (19) [x,y,10]
--R
--R                                          Type: List(Expression(Integer))
--E 19
)spool
)lisp (bye)

```

— Kernel.help —

=====

Kernel examples

=====

A kernel is a symbolic function application (such as $\sin(x+y)$) or a symbol (such as x). More precisely, a non-symbol kernel over a set S is an operator applied to a given list of arguments from S . The operator has type `BasicOperator` and the kernel object is usually part of an `Expression` object.

Kernels are created implicitly for you when you create expressions.

```
x :: Expression Integer
x
Type: Expression Integer
```

You can directly create a "symbol" kernel by using the kernel operation.

```
kernel x
x
Type: Kernel Expression Integer
```

This expression has two different kernels.

```
sin(x) + cos(x)
sin(x) + cos(x)
Type: Expression Integer
```

The operator kernels returns a list of the kernels in an object of type Expression.

```
kernels %
[sin(x),cos(x)]
Type: List Kernel Expression Integer
```

This expression also has two different kernels.

```
sin(x)**2 + sin(x) + cos(x)
2
sin(x) + sin(x) + cos(x)
Type: Expression Integer
```

The sin(x) kernel is used twice.

```
kernels %
[sin(x),cos(x)]
Type: List Kernel Expression Integer
```

An expression need not contain any kernels.

```
kernels(1 :: Expression Integer)
[]
Type: List Kernel Expression Integer
```

If one or more kernels are present, one of them is designated the main kernel.

```
mainKernel(cos(x) + tan(x))
tan(x)
Type: Union(Kernel Expression Integer,...)
```

Kernels can be nested. Use `height` to determine the nesting depth.

```
height kernel x
1
Type: PositiveInteger
```

This has height 2 because the `x` has height 1 and then we apply an operator to that.

```
height mainKernel(sin x)
2
Type: PositiveInteger
```

```
height mainKernel(sin cos x)
3
Type: PositiveInteger
```

```
height mainKernel(sin cos (tan x + sin x))
4
Type: PositiveInteger
```

Use the `operator` operation to extract the operator component of the kernel. The operator has type `BasicOperator`.

```
operator mainKernel(sin cos (tan x + sin x))
sin
Type: BasicOperator
```

Use the `name` operation to extract the name of the operator component of the kernel. The name has type `Symbol`. This is really just a shortcut for a two-step process of extracting the operator and then calling `name` on the operator.

```
name mainKernel(sin cos (tan x + sin x))
sin
Type: Symbol
```

Axiom knows about functions such as `sin`, `cos` and so on and can make kernels and then expressions using them. To create a kernel and expression using an arbitrary operator, use `operator`.

Now `f` can be used to create symbolic function applications.

```
f := operator 'f
f
Type: BasicOperator

e := f(x, y, 10)
f(x,y,10)
Type: Expression Integer
```

Use the `is?` operation to learn if the operator component of a kernel is equal to a given operator.

```
is?(e, f)
true
```

Type: Boolean

You can also use a symbol or a string as the second argument to `is?`.

```
is?(e, 'f')
true
```

Type: Boolean

Use the `argument` operation to get a list containing the argument component of a kernel.

```
argument mainKernel e
[x,y,10]
```

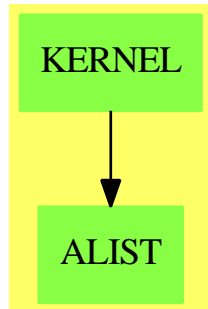
Type: List Expression Integer

Conceptually, an object of type `Expression` can be thought of a quotient of multivariate polynomials, where the "variables" are kernels. The arguments of the kernels are again expressions and so the structure recurses. See `Expression` for examples of using kernels to take apart expression objects.

See Also:

- o `)help Expression`
- o `)help BasicOperator`
- o `)show Kernel`

Kernel (KERNEL)



See

⇒ “MakeCachableSet” (MKCHSET) [14](#) on page [1570](#)

Exports:

argument	coerce	convert	hash	height
is?	kernel	latex	max	min
name	operator	position	setPosition	symbolIfCan
?~=?	?<?	?<=?	?=?	?>?
?>=?				

— domain **KERNEL** Kernel —

```

)abbrev domain KERNEL Kernel
++ Author: Manuel Bronstein
++ Date Created: 22 March 1988
++ Date Last Updated: 10 August 1994
++ Description:
++ A kernel over a set S is an operator applied to a given list
++ of arguments from S.

```

```

Kernel(S:OrderedSet): Exports == Implementation where

```

```

  O ==> OutputForm
  N ==> NonNegativeInteger
  OP ==> BasicOperator

```

```

  SYMBOL ==> "%symbol"
  PMPRED ==> "%pmpredicate"
  PMOPT ==> "%pmpoptional"
  PMMULT ==> "%pmmultiple"
  PMCONST ==> "%pmconstant"
  SPECIALDISP ==> "%specialDisp"
  SPECIALEQUAL ==> "%specialEqual"
  SPECIALINPUT ==> "%specialInput"

```

```

Exports ==> Join(CachableSet, Patternable S) with
  name      : % -> Symbol

```

```

    ++ name(op(a1,...,an)) returns the name of op.
operator: % -> OP
    ++ operator(op(a1,...,an)) returns the operator op.
argument: % -> List S
    ++ argument(op(a1,...,an)) returns \spad{[a1,...,an]}.
height : % -> N
    ++ height(k) returns the nesting level of k.
kernel : (OP, List S, N) -> %
    ++ kernel(op, [a1,...,an], m) returns the kernel \spad{op(a1,...,an)}
    ++ of nesting level m.
    ++ Error: if op is k-ary for some k not equal to m.
kernel : Symbol -> %
    ++ kernel(x) returns x viewed as a kernel.
symbolIfCan: % -> Union(Symbol, "failed")
    ++ symbolIfCan(k) returns k viewed as a symbol if k is a symbol, and
    ++ "failed" otherwise.
is?      : (% , OP) -> Boolean
    ++ is?(op(a1,...,an), f) tests if op = f.
is?      : (% , Symbol) -> Boolean
    ++ is?(op(a1,...,an), s) tests if the name of op is s.
if S has ConvertibleTo InputForm then ConvertibleTo InputForm

Implementation ==> add
import SortedCache(%)

Rep := Record(op:OP, arg:List S, nest:N, posit:N)

clearCache()

B2Z : Boolean -> Integer
trriage: (% , %) -> Integer
preds : OP -> List Any

is?(k:%, s:Symbol) == is?(operator k, s)
is?(k:%, o:OP) == (operator k) = o
name k == name operator k
height k == k.nest
operator k == k.op
argument k == k.arg
position k == k.posit
setPosition(k, n) == k.posit := n
B2Z flag == (flag => -1; 1)
kernel s == kernel(assert(operator(s,0),SYMBOL), nil(), 1)

preds o ==
  (u := property(o, PMPRED)) case "failed" => nil()
  (u::None) pretend List(Any)

symbolIfCan k ==
  has?(operator k, SYMBOL) => name operator k

```



```

"failed"

k1 = k2 ==
  if k1.posit = 0 then enterInCache(k1, triage)
  if k2.posit = 0 then enterInCache(k2, triage)
  k1.posit = k2.posit

k1 < k2 ==
  if k1.posit = 0 then enterInCache(k1, triage)
  if k2.posit = 0 then enterInCache(k2, triage)
  k1.posit < k2.posit

kernel(fn, x, n) ==
  ((u := arity fn) case N) and (#x ^= u::N)
  => error "Wrong number of arguments"
  enterInCache([fn, x, n, 0]$Rep, triage)

-- SPECIALDISP contains a map List S -> OutputForm
-- it is used when the converting the arguments first is not good,
-- for instance with formal derivatives.
coerce(k:%):OutputForm ==
  (v := symbolIfCan k) case Symbol => v::Symbol::OutputForm
  (f := property(o := operator k, SPECIALDISP)) case None =>
    ((f::None) pretend (List S -> OutputForm)) (argument k)
  l := [x::OutputForm for x in argument k]$List(OutputForm)
  (u := display o) case "failed" => prefix(name(o)::OutputForm, l)
  (u::(List OutputForm -> OutputForm)) l

triage(k1, k2) ==
  k1.nest ^= k2.nest => B2Z(k1.nest < k2.nest)
  k1.op ^= k2.op => B2Z(k1.op < k2.op)
  (n1 := #(argument k1)) ^= (n2 := #(argument k2)) => B2Z(n1 < n2)
  ((func := property(operator k1, SPECIALEQUAL)) case None) and
  (((func::None) pretend ((%, %) -> Boolean)) (k1, k2)) => 0
  for x1 in argument(k1) for x2 in argument(k2) repeat
    x1 ^= x2 => return B2Z(x1 < x2)
  0

if S has ConvertibleTo InputForm then
  convert(k:%):InputForm ==
    (v := symbolIfCan k) case Symbol => convert(v::Symbol)@InputForm
    (f := property(o := operator k, SPECIALINPUT)) case None =>
      ((f::None) pretend (List S -> InputForm)) (argument k)
    l := [convert x for x in argument k]$List(InputForm)
    (u := input operator k) case "failed" =>
      convert concat(convert name operator k, l)
    (u::(List InputForm -> InputForm)) l

if S has ConvertibleTo Pattern Integer then
  convert(k:%):Pattern(Integer) ==

```

```

o := operator k
(v := symbolIfCan k) case Symbol =>
  s := patternVariable(v::Symbol,
    has?(o, PMCONST), has?(o, PMOPT), has?(o, PMMULT))
  empty?(l := preds o) => s
  setPredicates(s, l)
o [convert x for x in k.arg]$List(Pattern Integer)

if S has ConvertibleTo Pattern Float then
convert(k:%):Pattern(Float) ==
o := operator k
(v := symbolIfCan k) case Symbol =>
  s := patternVariable(v::Symbol,
    has?(o, PMCONST), has?(o, PMOPT), has?(o, PMMULT))
  empty?(l := preds o) => s
  setPredicates(s, l)
o [convert x for x in k.arg]$List(Pattern Float)

```

— KERNEL.dotabb —

```

"KERNEL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=KERNEL"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"KERNEL" -> "ALIST"

```

domain KAFILE KeyedAccessFile

— KeyedAccessFile.input —

```

)set break resume
)sys rm -f KeyedAccessFile.output
)spool KeyedAccessFile.output
)set message test on
)set message auto off
)clear all

--S 1 of 20
ey: KeyedAccessFile(Integer) := open("editor.year", "output")
--R
--R
--R (1) "editor.year"

```

[illegible]

```

--S 8 of 20
search("Char", ey)
--R
--R
--R (8) 1986
--R
--R                                         Type: Union(Integer,...)
--E 8

```

```

--S 9 of 20
search("Smith", ey)
--R
--R
--R (9) "failed"
--R
--R                                         Type: Union("failed",...)
--E 9

```

```

--S 10 of 20
remove!("Char", ey)
--R
--R
--R (10) 1986
--R
--R                                         Type: Union(Integer,...)
--E 10

```

```

--S 11 of 20
keys ey
--R
--R
--R (11) ["Fitch","Caviness"]
--R
--R                                         Type: List(String)
--E 11

```

```

--S 12 of 20
#ey
--R
--R
--R (12) 2
--R
--R                                         Type: PositiveInteger
--E 12

```

```

--S 13 of 20
KE := Record(key: String, entry: Integer)
--R
--R
--R (13) Record(key: String,entry: Integer)
--R
--R                                         Type: Domain
--E 13

```

```

--S 14 of 20

```

```

reopen!(ey, "output")
--R
--R
--R (14) "editor.year"
--R
--R                                          Type: KeyedAccessFile(Integer)
--E 14

--S 15 of 20
write!(ey, ["van Hulzen", 1983]$KE)
--R
--R
--R (15) [key= "van Hulzen",entry= 1983]
--R
--R                                          Type: Record(key: String,entry: Integer)
--E 15

--S 16 of 20
write!(ey, ["Calmet", 1982]$KE)
--R
--R
--R (16) [key= "Calmet",entry= 1982]
--R
--R                                          Type: Record(key: String,entry: Integer)
--E 16

--S 17 of 20
write!(ey, ["Wang", 1981]$KE)
--R
--R
--R (17) [key= "Wang",entry= 1981]
--R
--R                                          Type: Record(key: String,entry: Integer)
--E 17

--S 18 of 20
close! ey
--R
--R
--R (18) "editor.year"
--R
--R                                          Type: KeyedAccessFile(Integer)
--E 18

--S 19 of 20
keys ey
--R
--R
--R (19) ["Wang","Calmet","van Hulzen","Fitch","Caviness"]
--R
--R                                          Type: List(String)
--E 19

--S 20 of 20
members ey
--R

```

```

--R
--R   (20)  [1981,1982,1983,1984,1985]
--R
--R                                          Type: List(Integer)
--E 20

)system rm -r editor.year

)spool
)lisp (bye)

```

— KeyedAccessFile.help —

=====

KeyedAccessFile examples

=====

The domain KeyedAccessFile(S) provides files which can be used as associative tables. Data values are stored in these files and can be retrieved according to their keys. The keys must be strings so this type behaves very much like the StringTable(S) domain. The difference is that keyed access files reside in secondary storage while string tables are kept in memory.

Before a keyed access file can be used, it must first be opened. A new file can be created by opening it for output.

```
ey: KeyedAccessFile(Integer) := open("editor.year", "output")
```

Just as for vectors, tables or lists, values are saved in a keyed access file by setting elements.

```
ey."Char" := 1986
```

```
ey."Caviness" := 1985
```

```
ey."Fitch" := 1984
```

Values are retrieved using application, in any of its syntactic forms.

```
ey."Char"
```

```
ey("Char")
```

```
ey "Char"
```

Attempting to retrieve a non-existent element in this way causes an error. If it is not known whether a key exists, you should use the

search operation.

```
search("Char", ey)

search("Smith", ey)
```

When an entry is no longer needed, it can be removed from the file.

```
remove!("Char", ey)
```

The keys operation returns a list of all the keys for a given file.

```
keys ey
```

The # operation gives the number of entries.

```
#ey
```

The table view of keyed access files provides safe operations. That is, if the Axiom program is terminated between file operations, the file is left in a consistent, current state. This means, however, that the operations are somewhat costly. For example, after each update the file is closed.

Here we add several more items to the file, then check its contents.

```
KE := Record(key: String, entry: Integer)

reopen!(ey, "output")
```

If many items are to be added to a file at the same time, then it is more efficient to use the write operation.

```
write!(ey, ["van Hulzen", 1983]$KE)

write!(ey, ["Calmet", 1982]$KE)

write!(ey, ["Wang", 1981]$KE)

close! ey
```

The read operation is also available from the file view, but it returns elements in a random order. It is generally clearer and more efficient to use the keys operation and to extract elements by key.

```
keys ey

members ey

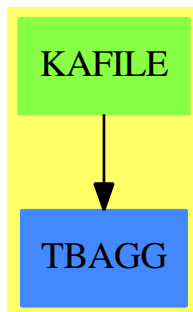
)system rm -r editor.year
```

See Also:

- o `)help Table`
- o `)help StringTable`
- o `)help File`
- o `)help TextFile`
- o `)help Library`
- o `)show KeyedAccessFile`

—————→

KeyedAccessFile (KAFILE)



See

- ⇒ “File” (FILE) [7](#) on page [802](#)
- ⇒ “TextFile” (TEXTFILE) [21](#) on page [2692](#)
- ⇒ “BinaryFile” (BINFILE) [3](#) on page [295](#)
- ⇒ “Library” (LIB) [13](#) on page [1428](#)

Exports:

any?	any?	bag	close!	coerce
construct	convert	copy	count	count
dictionary	elt	empty	empty?	entries
entry?	eq?	eval	every?	every?
extract!	fill!	find	first	hash
index?	indices	insert!	inspect	iomode
key?	keys	latex	less?	map
map!	maxIndex	member?	members	minIndex
more?	name	open	pack!	parts
parts	qelt	qsetelt!	read!	reduce
remove	remove!	removeDuplicates	reopen!	sample
search	select	select!	setelt	size?
swap!	table	write!	#?	?~=?
?=?	?.?			

— domain KAFfile KeyedAccessFile —

```

)abbrev domain KAFfile KeyedAccessFile
++ Author: Stephen M. Watt
++ Date Created: 1985
++ Date Last Updated: June 4, 1991
++ Description:
++ This domain allows a random access file to be viewed both as a table
++ and as a file object. The KeyedAccessFile format is a directory
++ containing a single file called "index.kaf". This file is a random
++ access file. The first thing in the file is an integer which is the
++ byte offset of an association list (the dictionary) at the end of
++ the file. The association list is of the form
++ ((key . byteoffset) (key . byteoffset)...)
++ where the byte offset is the number of bytes from the beginning of
++ the file. This offset contains an s-expression for the value of the key.

KeyedAccessFile(Entry): KAFcategory == KAFcapsule where
  Name ==> FileName
  Key ==> String
  Entry : SetCategory

KAFcategory ==
  Join(FileCategory(Name, Record(key: Key, entry: Entry)),
    TableAggregate(Key, Entry)) with
    finiteAggregate
    pack_!: % -> %
    ++ pack!(f) reorganizes the file f on disk to recover
    ++ unused space.

KAFcapsule == add

CLASS ==> 131 -- an arbitrary no. greater than 127

```

```

FileState ==> SExpression
IOMode      ==> String

Cons:= Record(car: SExpression, cdr: SExpression)
Rep := Record(fileName:  Name,      _
               fileState: FileState, _
               fileIOmode: IOMode)

defstream(fn: Name, mode: IOMode): FileState ==
  kafstring:=concat(fn::String,"/index.kaf")::FileName
  mode = "input" =>
    not readable? kafstring => error ["File is not readable", fn]
    RDEFINSTREAM(fn)$Lisp
  mode = "output" =>
    not writable? fn => error ["File is not writable", fn]
    RDEFOUTSTREAM(fn)$Lisp
  error ["IO mode must be input or output", mode]

---- From Set ----
f1 = f2 ==
  f1.fileName = f2.fileName
coerce(f: %): OutputForm ==
  f.fileName::OutputForm

---- From FileCategory ----
open fname ==
  open(fname, "either")
open(fname, mode) ==
  mode = "either" =>
    exists? fname =>
      open(fname, "input")
    writable? fname =>
      reopen_!(open(fname, "output"), "input")
    error "File does not exist and cannot be created"
  [fname, defstream(fname, mode), mode]
reopen_!(f, mode) ==
  close_! f
  if mode ^= "closed" then
    f.fileState := defstream(f.fileName, mode)
    f.fileIOmode := mode
  f
close_! f ==
  if f.fileIOmode ^= "closed" then
    RSHUT(f.fileState)$Lisp
    f.fileIOmode := "closed"
  f
read_! f ==
  f.fileIOmode ^= "input" => error ["File not in read state",f]
  ks: List Symbol := RKEYIDS(f.fileName)$Lisp

```

```

    null ks => error ["Attempt to read empty file", f]
    ix := random()$Integer rem #ks
    k: String := PNAME(ks.ix)$Lisp
    [k, SPADRREAD(k, f.fileState)$Lisp]
write_!(f, pr) ==
    f.fileIOmode ^= "output" => error ["File not in write state",f]
    SPADRWRITE(pr.key, pr.entry, f.fileState)$Lisp
    pr
name f ==
    f.fileName
iomode f ==
    f.fileIOmode

---- From TableAggregate ----
empty() ==
    fn := new("", "kaf", "sdata")$Name
    open fn
keys f ==
    close_! f
    l: List SExpression := RKEYIDS(f.fileName)$Lisp
    [PNAME(n)$Lisp for n in l]
# f ==
    # keys f
elt(f,k) ==
    reopen_!(f, "input")
    SPADRREAD(k, f.fileState)$Lisp
setelt(f,k,e) ==
    -- Leaves f in a safe, closed state. For speed use "write".
    reopen_!(f, "output")
    UNWIND_-PROTECT(write_!(f, [k,e]), close_! f)$Lisp
    close_! f
    e
search(k,f) ==
    not member?(k, keys f) => "failed" -- can't trap RREAD error
    reopen_!(f, "input")
    (SPADRREAD(k, f.fileState)$Lisp)@Entry
remove_!(k:String,f:%) ==
    result := search(k,f)
    result case "failed" => result
    close_! f
    RDROPITEMS(NAMESTRING(f.fileName)$Lisp, LIST(k)$Lisp)$Lisp
    result
pack_! f ==
    close_! f
    RPACKFILE(f.fileName)$Lisp
    f

```

— KAFILE.dotabb —

```
"KAFILE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=KAFILE"]  
"TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]  
"KAFILE" -> "TBAGG"
```

—————

Chapter 13

Chapter L

domain LAUPOL LaurentPolynomial

— LaurentPolynomial.input —

```
)set break resume
)sys rm -f LaurentPolynomial.output
)spool LaurentPolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LaurentPolynomial
--R
--R LaurentPolynomial(R: IntegralDomain,UP: UnivariatePolynomialCategory(R)) is a domain constructor
--R Abbreviation for LaurentPolynomial is LAUPOL
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for LAUPOL
--R
--R----- Operations -----
--R ?? : (% ,%) -> %
--R ?? : (PositiveInteger,% ) -> %
--R ?? : (% ,%) -> %
--R -? : % -> %
--R D : % -> % if UP has DIFRING
--R 1 : () -> %
--R ?? : (% ,PositiveInteger) -> %
--R coefficient : (% ,Integer) -> R
--R coerce : R -> %
--R coerce : Integer -> %
--R convert : % -> Fraction(UP)
--R ?? : (Integer,% ) -> %
--R ??? : (% ,PositiveInteger) -> %
--R ?-? : (% ,%) -> %
--R ?=? : (% ,%) -> Boolean
--R D : (% ,(UP -> UP)) -> %
--R 0 : () -> %
--R associates? : (% ,%) -> Boolean
--R coerce : UP -> %
--R coerce : % -> %
--R coerce : % -> OutputForm
--R degree : % -> Integer
```

```

--R hash : % -> SingleInteger
--R leadingCoefficient : % -> R
--R monomial? : % -> Boolean
--R order : % -> Integer
--R reductum : % -> %
--R retract : % -> R
--R trailingCoefficient : % -> R
--R unitCanonical : % -> %
--R ?^=? : (%,%) -> Boolean
--R ?*? : (NonNegativeInteger,%) -> %
--R ***? : (%,NonNegativeInteger) -> %
--R D : (%,NonNegativeInteger) -> % if UP has DIFRING
--R D : (%,Symbol) -> % if UP has PDRING(SYMBOL)
--R D : (%,List(Symbol)) -> % if UP has PDRING(SYMBOL)
--R D : (%,Symbol,NonNegativeInteger) -> % if UP has PDRING(SYMBOL)
--R D : (%,List(Symbol),List(NonNegativeInteger)) -> % if UP has PDRING(SYMBOL)
--R D : (%,(UP -> UP),NonNegativeInteger) -> %
--R ?^? : (%,NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if R has CHARNZ
--R coerce : Fraction(Integer) -> % if R has RETRACT(FRAC(INT))
--R differentiate : % -> % if UP has DIFRING
--R differentiate : (%,NonNegativeInteger) -> % if UP has DIFRING
--R differentiate : (%,Symbol) -> % if UP has PDRING(SYMBOL)
--R differentiate : (%,List(Symbol)) -> % if UP has PDRING(SYMBOL)
--R differentiate : (%,Symbol,NonNegativeInteger) -> % if UP has PDRING(SYMBOL)
--R differentiate : (%,List(Symbol),List(NonNegativeInteger)) -> % if UP has PDRING(SYMBOL)
--R differentiate : (%,(UP -> UP),NonNegativeInteger) -> %
--R differentiate : (%,(UP -> UP)) -> %
--R divide : (%,%) -> Record(quotient: %,remainder: %) if R has FIELD
--R euclideanSize : % -> NonNegativeInteger if R has FIELD
--R expressIdealMember : (List(%),%) -> Union(List(%), "failed") if R has FIELD
--R exquo : (%,%) -> Union(%, "failed")
--R extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %) if R has FIELD
--R extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %), "failed") if R has FIELD
--R gcd : (%,%) -> % if R has FIELD
--R gcd : List(%) -> % if R has FIELD
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R lcm : (%,%) -> % if R has FIELD
--R lcm : List(%) -> % if R has FIELD
--R multiEuclidean : (List(%),%) -> Union(List(%), "failed") if R has FIELD
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %) if R has FIELD
--R ?quo? : (%,%) -> % if R has FIELD
--R ?rem? : (%,%) -> % if R has FIELD
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retract : % -> Integer if R has RETRACT(INT)
--R retractIfCan : % -> Union(UP, "failed")
--R retractIfCan : % -> Union(R, "failed")
--R retractIfCan : % -> Union(Fraction(Integer), "failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(Integer, "failed") if R has RETRACT(INT)

```

```
--R separate : Fraction(UP) -> Record(polyPart: %,fracPart: Fraction(UP)) if R has FIELD
--R sizeLess? : (%,%)-> Boolean if R has FIELD
--R subtractIfCan : (%,%)-> Union(%,"failed")
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R
--E 1

)spool
)lisp (bye)
```

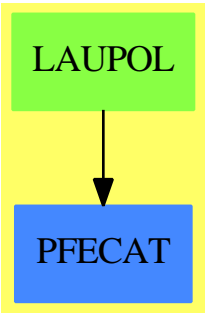
— LaurentPolynomial.help —

=====
LaurentPolynomial examples
=====

Univariate polynomials with negative and positive exponents.

See Also:
o)show LaurentPolynomial

LaurentPolynomial (LAUPOL)



Exports:

0	1	associates?
characteristic	charthRoot	coefficient
coerce	convert	D
degree	differentiate	divide
euclideanSize	expressIdealMember	exquo
extendedEuclidean	gcd	gcdPolynomial
hash	latex	lcm
leadingCoefficient	monomial	monomial?
multiEuclidean	one?	order
principalIdeal	recip	reductum
retract	retractIfCan	sample
separate	sizeLess?	subtractIfCan
trailingCoefficient	unit?	unitCanonical
unitNormal	zero?	?*?
?**?	?+?	?-?
-?	?=?	?^?
?~=?	?quo?	?rem?

— domain LAUPOL LaurentPolynomial —

```

)abbrev domain LAUPOL LaurentPolynomial
++ Author: Manuel Bronstein
++ Date Created: May 1988
++ Date Last Updated: 26 Apr 1990
++ Description:
++ Univariate polynomials with negative and positive exponents.

LaurentPolynomial(R, UP): Exports == Implementation where
  R : IntegralDomain
  UP: UnivariatePolynomialCategory R

O  ==> OutputForm
B  ==> Boolean
N  ==> NonNegativeInteger
Z  ==> Integer
RF ==> Fraction UP

Exports ==> Join(DifferentialExtension UP, IntegralDomain,
  ConvertibleTo RF, FullyRetractableTo R, RetractableTo UP) with
monomial?      : % -> B
  ++ monomial?(x) is not documented
degree         : % -> Z
  ++ degree(x) is not documented
order          : % -> Z
  ++ order(x) is not documented
reductum       : % -> %
  ++ reductum(x) is not documented

```

```

leadingCoefficient : % -> R
  ++ leadingCoefficient is not documented
trailingCoefficient: % -> R
  ++ trailingCoefficient is not documented
coefficient        : (%, Z) -> R
  ++ coefficient(x,n) is not documented
monomial           : (R, Z) -> %
  ++ monomial(x,n) is not documented
if R has CharacteristicZero then CharacteristicZero
if R has CharacteristicNonZero then CharacteristicNonZero
if R has Field then
  EuclideanDomain
  separate: RF -> Record(polyPart:%, fracPart:RF)
  ++ separate(x) is not documented

Implementation ==> add
Rep := Record(polypart: UP, order0: Z)

poly   : % -> UP
check0 : (Z, UP) -> %
mkgpol : (Z, UP) -> %
gpol   : (UP, Z) -> %
toutput: (R, Z, 0) -> 0
monTerm: (R, Z, 0) -> 0

0          == [0, 0]
1          == [1, 0]
p = q      == p.order0 = q.order0 and p.polypart = q.polypart
poly p     == p.polypart
order p    == p.order0
gpol(p, n) == [p, n]
monomial(r, n) == check0(n, r::UP)
coerce(p:UP):% == mkgpol(0, p)
reductum p  == check0(order p, reductum poly p)
n:Z * p:%   == check0(order p, n * poly p)
characteristic() == characteristic()$R
coerce(n:Z):% == n::R::%
degree p    == degree(poly p)::Z + order p
monomial? p == monomial? poly p
coerce(r:R):% == gpol(r::UP, 0)
convert(p:%):RF == poly(p) * (monomial(1, 1)$UP)::RF ** order p
p:% * q:%    == check0(order p + order q, poly p * poly q)
- p          == gpol(- poly p, order p)
check0(n, p) == (zero? p => 0; gpol(p, n))
trailingCoefficient p == coefficient(poly p, 0)
leadingCoefficient p == leadingCoefficient poly p

coerce(p:%):0 ==
  zero? p => 0::Z::0
  1 := nil()$List(0)

```

```

v := monomial(1, 1)$UP :: 0
while p ^= 0 repeat
  l := concat(1, toutput(leadingCoefficient p, degree p, v))
  p := reductum p
  reduce("+", l)

coefficient(p, n) ==
  (m := n - order p) < 0 => 0
  coefficient(poly p, m::N)

differentiate(p:%, derivation:UP -> UP) ==
  t := monomial(1, 1)$UP
  mkgpol(order(p) - 1,
    derivation(poly p) * t + order(p) * poly(p) * derivation t)

monTerm(r, n, v) ==
  zero? n => r::0
--   one? n => v
  (n = 1) => v
  v ** (n::0)

toutput(r, n, v) ==
  mon := monTerm(r, n, v)
--   zero? n or one? r => mon
  zero? n or (r = 1) => mon
  r = -1 => - mon
  r::0 * mon

recip p ==
  (q := recip poly p) case "failed" => "failed"
  gpol(q::UP, - order p)

p + q ==
  zero? q => p
  zero? p => q
  (d := order p - order q) > 0 =>
    gpol(poly(p) * monomial(1, d::N) + poly q, order q)
  d < 0 => gpol(poly(p) + poly(q) * monomial(1, (-d)::N), order p)
  mkgpol(order p, poly(p) + poly q)

mkgpol(n, p) ==
  zero? p => 0
  d := order(p, monomial(1, 1)$UP)
  gpol((p exquo monomial(1, d))::UP, n + d::Z)

p exquo q ==
  (r := poly(p) exquo poly q) case "failed" => "failed"
  check0(order p - order q, r::UP)

retractIfCan(p:%):Union(UP, "failed") ==

```

```

order(p) < 0 => error "Not retractable"
poly(p) * monomial(1, order(p)::N)$UP

retractIfCan(p:%):Union(R, "failed") ==
  order(p) ^= 0 => "failed"
  retractIfCan poly p

if R has Field then
  gcd(p, q) == gcd(poly p, poly q)::%

separate f ==
  n := order(q := denom f, monomial(1, 1))
  q := (q exquo (tn := monomial(1, n)$UP))::UP
  bc := extendedEuclidean(tn,q,numer f)::Record(coef1:UP,coef2:UP)
  qr := divide(bc.coef1, q)
  [mkgpol(-n, bc.coef2 + tn * qr.quotient), qr.remainder / q]

-- returns (z, r) s.t. p = q z + r,
-- and degree(r) < degree(q), order(r) >= min(order(p), order(q))
divide(p, q) ==
  c := min(order p, order q)
  qr := divide(poly(p) * monomial(1, (order p - c)::N)$UP, poly q)
  [mkgpol(c - order q, qr.quotient), mkgpol(c, qr.remainder)]

euclideanSize p == degree poly p

extendedEuclidean(a, b, c) ==
  (bc := extendedEuclidean(poly a, poly b, poly c)) case "failed"
  => "failed"
  [mkgpol(order c - order a, bc.coef1),
   mkgpol(order c - order b, bc.coef2)]

-----

— LAUPOL.dotabb —

"LAUPOL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=LAUPOL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"LAUPOL" -> "PFECAT"

-----

```

domain LIB Library

— Library.input —

```

)set break resume
)sys rm -f Library.output
)spool Library.output
)set message test on
)set message auto off
)clear all

```

```

--S 1 of 7
stuff := library "Neat.stuff"
--R
--R
--R (1) "Neat.stuff"
--R
--E 1

```

Type: Library

```

--S 2 of 7
stuff.int := 32**2
--R
--R
--R (2) 1024
--R
--E 2

```

Type: PositiveInteger

```

--S 3 of 7
stuff."poly" := x**2 + 1
--R
--R
--R      2
--R (3) x  + 1
--R
--E 3

```

Type: Polynomial(Integer)

```

--S 4 of 7
stuff.str := "Hello"
--R
--R
--R (4) "Hello"
--R
--E 4

```

Type: String

```

--S 5 of 7
keys stuff
--R
--R
--R (5) ["str","poly","int"]
--R
--E 5

```

Type: List(String)

```

--S 6 of 7
stuff.poly

```

```
--R
--R
--R      2
--R (6)  x  + 1
--R
--R                                          Type: Polynomial(Integer)
--E 6
```

```
--S 7 of 7
stuff("poly")
--R
--R
--R      2
--R (7)  x  + 1
--R
--R                                          Type: Polynomial(Integer)
--E 7
```

```
)system rm -rf Neat.stuff
)spool
)lisp (bye)
```

— Library.help —

```
=====
Library examples
=====
```

The Library domain provides a simple way to store Axiom values in a file. This domain is similar to KeyedAccessFile but fewer declarations are needed and items of different types can be saved together in the same file.

To create a library, you supply a file name.

```
stuff := library "Neat.stuff"
```

Now values can be saved by key in the file. The keys should be mnemonic, just as the field names are for records. They can be given either as strings or symbols.

```
stuff.int := 32**2

stuff."poly" := x**2 + 1

stuff.str := "Hello"
```

You obtain the set of available keys using the keys operation.

```
keys stuff
```

You extract values by giving the desired key in this way.

```
stuff.poly
```

```
stuff("poly")
```

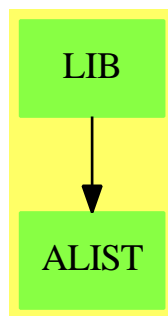
When the file is no longer needed, you should remove it from the file system.

```
)system rm -rf Neat.stuff
```

See Also:

- o)help File
- o)help TextFile
- o)help KeyedAccessFile
- o)show Library

Library (LIB)



See

- ⇒ “File” (FILE) [7](#) on page [802](#)
- ⇒ “TextFile” (TEXTFILE) [21](#) on page [2692](#)
- ⇒ “BinaryFile” (BINFILE) [3](#) on page [295](#)
- ⇒ “KeyedAccessFile” (KAFILE) [12](#) on page [1413](#)

Exports:

any?	any?	bag	close!	coerce
copy	construct	convert	count	dictionary
elt	empty	empty?	entries	entry?
eq?	eval	every?	every?	extract!
fill!	find	first	hash	index?
indices	insert!	inspect	key?	keys
latex	less?	library	map	map!
maxIndex	member?	members	minIndex	more?
pack!	parts	qelt	qsetelt!	reduce
remove	remove!	removeDuplicates	sample	search
select	select!	setelt	size?	swap!
table	#?	?=?	?~=?	?..?

— domain LIB Library —

```

)abbrev domain LIB Library
++ Author: Stephen M. Watt
++ Date Created: 1985
++ Date Last Updated: June 4, 1991
++ Description:
++ This domain provides a simple way to save values in files.

Library(): TableAggregate(String, Any) with
  library: FileName -> %
    ++ library(ln) creates a new library file.
  pack_!: % -> %
    ++ pack!(f) reorganizes the file f on disk to recover
    ++ unused space.

  elt : (% , Symbol) -> Any
    ++ elt(lib,k) or lib.k extracts the value corresponding to
    ++ the key \spad{k} from the library \spad{lib}.

  setelt : (% , Symbol, Any) -> Any
    ++ \spad{lib.k := v} saves the value \spad{v} in the library
    ++ \spad{lib}. It can later be extracted using the key \spad{k}.

  close_!: % -> %
    ++ close!(f) returns the library f closed to input and output.

== KeyedAccessFile(Any) add
  Rep := KeyedAccessFile(Any)
  library f == open f
  elt(f:%,v:Symbol) == elt(f, string v)
  setelt(f:%, v:Symbol, val:Any) == setelt(f, string v, val)

```

— LIB.dotabb —

```
"LIB" [color="#88FF44",href="bookvol10.3.pdf#nameddest=LIB"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"LIB" -> "ALIST"
```

—

domain LEXP LieExponentials

— LieExponentials.input —

```
)set break resume
)sys rm -f LieExponentials.output
)spool LieExponentials.output
)set message test on
)set message auto off
)clear all
--S 1 of 13
a: Symbol := 'a
--R
--R
--R (1) a
--R
--R                                          Type: Symbol
--E 1

--S 2 of 13
b: Symbol := 'b
--R
--R
--R (2) b
--R
--R                                          Type: Symbol
--E 2

--S 3 of 13
coef := Fraction(Integer)
--R
--R
--R (3) Fraction(Integer)
--R
--R                                          Type: Domain
--E 3

--S 4 of 13
group := LieExponentials(Symbol, coef, 3)
--R
```

```

--R
--R (4) LieExponentials(Symbol,Fraction(Integer),3)
--R                                         Type: Domain
--E 4

--S 5 of 13
lpoly := LiePolynomial(Symbol, coef)
--R
--R
--R (5) LiePolynomial(Symbol,Fraction(Integer))
--R                                         Type: Domain
--E 5

--S 6 of 13
poly := XPBWPolynomial(Symbol, coef)
--R
--R
--R (6) XPBWPolynomial(Symbol,Fraction(Integer))
--R                                         Type: Domain
--E 6

--S 7 of 13
ea := exp(a::lpoly)$group
--R
--R
--R [a]
--R (7) e
--R                                         Type: LieExponentials(Symbol,Fraction(Integer),3)
--E 7

--S 8 of 13
eb := exp(b::lpoly)$group
--R
--R
--R [b]
--R (8) e
--R                                         Type: LieExponentials(Symbol,Fraction(Integer),3)
--E 8

--S 9 of 13
g: group := ea*eb
--R
--R
--R          1      2      1      2
--R        - [a b ] - [a b ]
--R      [b] 2      [a b] 2      [a]
--R (9) e e      e      e
--R                                         Type: LieExponentials(Symbol,Fraction(Integer),3)
--E 9

```

```

--S 10 of 13
g :: poly
--R
--R
--R (10)
--R      1      1      1
--R      1 + [a] + [b] + - [a][a] + [a b] + [b][a] + - [b][b] + - [a][a][a]
--R      2      2      6
--R      +
--R      1 2      1 2      1
--R      - [a b] + [a b][a] + - [a b ] + - [b][a][a] + [b][a b] + - [b][b][a]
--R      2      2      2      2
--R      +
--R      1
--R      - [b][b][b]
--R      6
--R
--R                                          Type: XPBWPolynomial(Symbol,Fraction(Integer))
--E 10

--S 11 of 13
log(g)$group
--R
--R
--R (11)
--R      1      1 2      1 2
--R      [a] + [b] + - [a b] + -- [a b] + -- [a b ]
--R      2      12      12
--R
--R                                          Type: LiePolynomial(Symbol,Fraction(Integer))
--E 11

--S 12 of 13
g1: group := inv(g)
--R
--R
--R      - [b] - [a]
--R      (12) e      e
--R
--R                                          Type: LieExponentials(Symbol,Fraction(Integer),3)
--E 12

--S 13 of 13
g*g1
--R
--R
--R (13) 1
--R
--R                                          Type: LieExponentials(Symbol,Fraction(Integer),3)
--E 13
)spool
)lisp (bye)

```

— LieExponentials.help —

===== LieExponentials examples =====

Management of the Lie Group associated with a free nilpotent Lie algebra. Every Lie bracket with length greater than Order are assumed to be null. The implementation inherits from the XPBWPolynomial domain constructor: Lyndon coordinates are exponential coordinates of the second kind.

```
a: Symbol := 'a
a
Type: Symbol
```

```
b: Symbol := 'b
b
Type: Symbol
```

===== Declarations of domains =====

```
coef := Fraction(Integer)
Fraction Integer
Type: Domain

group := LieExponentials(Symbol, coef, 3)
LieExponentials(Symbol, Fraction Integer, 3)
Type: Domain

lpoly := LiePolynomial(Symbol, coef)
LiePolynomial(Symbol, Fraction Integer)
Type: Domain

poly := XPBWPolynomial(Symbol, coef)
XPBWPolynomial(Symbol, Fraction Integer)
Type: Domain
```

===== Calculations =====

```
ea := exp(a::lpoly)$group
[a]
e
Type: LieExponentials(Symbol, Fraction Integer, 3)
```

```

eb := exp(b::lpoly)$group
      [b]
      e
                                     Type: LieExponentials(Symbol,Fraction Integer,3)

g: group := ea*eb
      1      2      1      2
      - [a b ] - [a b ]
      [b] 2      [a b] 2      [a]
      e  e      e  e      e
                                     Type: LieExponentials(Symbol,Fraction Integer,3)

g :: poly
      1      1      1
      1 + [a] + [b] + - [a][a] + [a b] + [b][a] + - [b][b] + - [a][a][a]
      2      2      6
+
      1      2      1      2      1      1
      - [a b] + [a b][a] + - [a b ] + - [b][a][a] + [b][a b] + - [b][b][a]
      2      2      2      2
+
      1
      - [b][b][b]
      6
                                     Type: XPBWPolynomial(Symbol,Fraction Integer)

log(g)$group
      1      1      2      1      2
      [a] + [b] + - [a b] + -- [a b] + -- [a b ]
      2      12      12
                                     Type: LiePolynomial(Symbol,Fraction Integer)

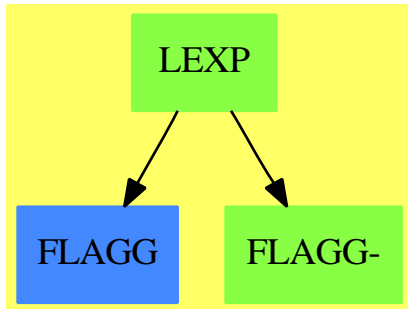
g1: group := inv(g)
      - [b] - [a]
      e      e
                                     Type: LieExponentials(Symbol,Fraction Integer,3)

g*g1
      1
                                     Type: LieExponentials(Symbol,Fraction Integer,3)

```

See Also:

o)show LieExponentials

LieExponentials (LEXP)**Exports:**

1	coerce	commutator	conjugate
exp	hash	identification	inv
latex	log	listOfTerms	LyndonBasis
LyndonCoordinates	mirror	one?	recip
sample	varList	?~=?	?^?
?*?	?**?	?/?	?=?

— domain **LEXP** LieExponentials —

```

)abbrev domain LEXP LieExponentials
++ Author: Michel Petitot (petitot@lifl.fr).
++ Date Created: 91
++ Date Last Updated: 7 Juillet 92
++ Fix History: compilation v 2.1 le 13 dec 98
++ Description:
++ Management of the Lie Group associated with a
++ free nilpotent Lie algebra. Every Lie bracket with
++ length greater than \axiom{Order} are assumed to be null.
++ The implementation inherits from the \spadtype{XPBWPolynomial}
++ domain constructor: Lyndon coordinates are exponential coordinates
++ of the second kind.

```

```

LieExponentials(VarSet, R, Order): XDPcat == XDPdef where

```

```

EX      ==> OutputForm
PI      ==> PositiveInteger
NNI     ==> NonNegativeInteger
I       ==> Integer
RN      ==> Fraction(I)
R       : Join(CommutativeRing, Module RN)
Order  : PI
VarSet  : OrderedSet
LWORD   ==> LyndonWord(VarSet)
LWORDS  ==> List LWORD

```

```

BASIS ==> PoincareBirkhoffWittLyndonBasis(VarSet)
TERM  ==> Record(k:BASIS, c:R)
LTERMS ==> List(TERM)
LPOLY ==> LiePolynomial(VarSet,R)
XDPOLY ==> XDistributedPolynomial(VarSet,R)
PBWPOLY==> XPBWPolynomial(VarSet, R)
TERM1 ==> Record(k:LWORD, c:R)
EQ    ==> Equation(R)

XDPcat == Group with
  exp      : LPOLY -> $
  ++ \axiom{exp(p)} returns the exponential of \axiom{p}.
  log      : $ -> LPOLY
  ++ \axiom{log(p)} returns the logarithm of \axiom{p}.
  listOfTerms : $ -> LTERMS
  ++ \axiom{listOfTerms(p)} returns the internal representation of \axiom{p}.
  coerce    : $ -> XDPOLY
  ++ \axiom{coerce(g)} returns the internal representation of \axiom{g}.
  coerce    : $ -> PBWPOLY
  ++ \axiom{coerce(g)} returns the internal representation of \axiom{g}.
  mirror    : $ -> $
  ++ \axiom{mirror(g)} is the mirror of the internal representation of \axiom{g}.
  varList   : $ -> List VarSet
  ++ \axiom{varList(g)} returns the list of variables of \axiom{g}.
  LyndonBasis : List VarSet -> List LPOLY
  ++ \axiom{LyndonBasis(lv)} returns the Lyndon basis of the nilpotent free
  ++ Lie algebra.
  LyndonCoordinates: $ -> List TERM1
  ++ \axiom{LyndonCoordinates(g)} returns the exponential coordinates of \axiom{g}.
  identification: ($,$) -> List EQ
  ++ \axiom{identification(g,h)} returns the list of equations \axiom{g_i = h_i},
  ++ where \axiom{g_i} (resp. \axiom{h_i}) are exponential coordinates
  ++ of \axiom{g} (resp. \axiom{h}).

XDPdef == PBWPOLY add

-- Representation
Rep := PBWPOLY

-- local functions
compareTerm1s: (TERM1, TERM1) -> Boolean
out: TERM1 -> EX
ident: (List TERM1, List TERM1) -> List EQ

-- functions locales
ident(l1, l2) ==
  import(TERM1)
  null l1 => [equation(0$R,t.c)$EQ for t in l2]
  null l2 => [equation(t.c, 0$R)$EQ for t in l1]
  u1 : LWORD := l1.first.k; c1 :R := l1.first.c

```

```

u2 : LWORD := l2.first.k; c2 :R := l2.first.c
u1 = u2 =>
  r: R := c1 - c2
  r = 0 => ident(rest l1, rest l2)
  cons(equation(c1,c2)$EQ , ident(rest l1, rest l2))
lexico(u1, u2)$LWORD =>
  cons(equation(0$R,c2)$EQ , ident(l1, rest l2))
cons(equation(c1,0$R)$EQ , ident(rest l1, l2))

-- ordre lexico decroissant
compareTerms(u:TERM1, v:TERM1):Boolean == lexico(v.k, u.k)$LWORD

out(t:TERM1):EX ==
  t.c =$R 1 => char("e")$Character :: EX ** t.k ::EX
  char("e")$Character :: EX ** (t.c::EX * t.k::EX)

-- definitions
identification(x,y) ==
  l1: List TERM1 := LyndonCoordinates x
  l2: List TERM1 := LyndonCoordinates y
  ident(l1, l2)

LyndonCoordinates x ==
  lt: List TERM1 := [[l::LWORD, t.c]$TERM1 for t in listOfTerms x | _
    (l := retractIfCan(t.k)$BASIS) case LWORD ]
  lt := sort(compareTerms,lt)

x:$ * y:$ == product(x::Rep, y::Rep, Order::I::NNI)$Rep

exp p == exp(p::Rep , Order::I::NNI)$Rep

log p == LiePolyIfCan(log(p,Order::I::NNI))$Rep :: LPOLY

coerce(p:$):EX ==
  p = 1$$ => 1$R :: EX
  lt : List TERM1 := LyndonCoordinates p
  reduce(*, [out t for t in lt])$List(EX)

LyndonBasis(lv) ==
  [LiePoly(l)$LPOLY for l in LyndonWordsList(lv,Order)$LWORD]

coerce(p:$):PBWPOLY == p::Rep

inv x ==
  x = 1 => 1
  lt:LTERMS := listOfTerms mirror x
  lt:= [[t.k, (odd? length(t.k)$BASIS => - t.c; t.c)]$TERM for t in lt ]
  lt pretend $

```



```

--S 4 of 28
Lword := LyndonWord Symbol
--R
--R
--R (4) LyndonWord(Symbol)
--R
--R                                          Type: Domain
--E 4

--S 5 of 28
a:Symbol := 'a
--R
--R
--R (5) a
--R
--R                                          Type: Symbol
--E 5

--S 6 of 28
b:Symbol := 'b
--R
--R
--R (6) b
--R
--R                                          Type: Symbol
--E 6

--S 7 of 28
c:Symbol := 'c
--R
--R
--R (7) c
--R
--R                                          Type: Symbol
--E 7

--S 8 of 28
aa: Lpoly := a
--R
--R
--R (8) [a]
--R
--R                                          Type: LiePolynomial(Symbol,Fraction(Integer))
--E 8

--S 9 of 28
bb: Lpoly := b
--R
--R
--R (9) [b]
--R
--R                                          Type: LiePolynomial(Symbol,Fraction(Integer))
--E 9

--S 10 of 28

```

```

cc: Lpoly := c
--R
--R
--R (10) [c]
--R
--R                                         Type: LiePolynomial(Symbol,Fraction(Integer))
--E 10

--S 11 of 28
p : Lpoly := [aa,bb]
--R
--R
--R (11) [a b]
--R
--R                                         Type: LiePolynomial(Symbol,Fraction(Integer))
--E 11

--S 12 of 28
q : Lpoly := [p,bb]
--R
--R
--R
--R      2
--R (12) [a b ]
--R
--R                                         Type: LiePolynomial(Symbol,Fraction(Integer))
--E 12

--S 13 of 28
liste : List Lword := LyndonWordsList([a,b], 4)
--R
--R
--R
--R      2      2      3      2 2      3
--R (13) [[a],[b],[a b],[a b],[a b ],[a b],[a b ],[a b ]]
--R
--R                                         Type: List(LyndonWord(Symbol))
--E 13

--S 14 of 28
r: Lpoly := p + q + 3*LiePoly(liste.4)$Lpoly
--R
--R
--R
--R      2      2
--R (14) [a b] + 3[a b] + [a b ]
--R
--R                                         Type: LiePolynomial(Symbol,Fraction(Integer))
--E 14

--S 15 of 28
s:Lpoly := [p,r]
--R
--R
--R
--R      2      2
--R (15) - 3[a b a b] + [a b a b ]
--R
--R                                         Type: LiePolynomial(Symbol,Fraction(Integer))
--E 15

```

```

--S 16 of 28
t:Lpoly := s + 2*LiePoly(liste.3) - 5*LiePoly(liste.5)
--R
--R
--R
--R      2      2      2
--R      (16)  2[a b] - 5[a b] - 3[a b a b] + [a b a b]
--R
--R      Type: LiePolynomial(Symbol,Fraction(Integer))
--E 16

--S 17 of 28
degree t
--R
--R
--R      (17)  5
--R
--R      Type: PositiveInteger
--E 17

--S 18 of 28
mirror t
--R
--R
--R
--R      2      2      2
--R      (18)  - 2[a b] - 5[a b] - 3[a b a b] + [a b a b]
--R
--R      Type: LiePolynomial(Symbol,Fraction(Integer))
--E 18

--S 19 of 28
Jacobi(p: Lpoly, q: Lpoly, r: Lpoly): Lpoly == _
  [ [p,q]$Lpoly, r] + [ [q,r]$Lpoly, p] + [ [r,p]$Lpoly, q]
--R
--R      Function declaration Jacobi : (LiePolynomial(Symbol,Fraction(Integer)
--R      )),LiePolynomial(Symbol,Fraction(Integer)),LiePolynomial(Symbol,
--R      Fraction(Integer))) -> LiePolynomial(Symbol,Fraction(Integer))
--R      has been added to workspace.
--R
--R      Type: Void
--E 19

--S 20 of 28
test: Lpoly := Jacobi(a,b,b)
--R
--R      Compiling function Jacobi with type (LiePolynomial(Symbol,Fraction(
--R      Integer)),LiePolynomial(Symbol,Fraction(Integer)),LiePolynomial(
--R      Symbol,Fraction(Integer))) -> LiePolynomial(Symbol,Fraction(
--R      Integer))
--R
--R      (20)  0
--R
--R      Type: LiePolynomial(Symbol,Fraction(Integer))
--E 20

```

```

--S 21 of 28
test: Lpoly := Jacobi(p,q,r)
--R
--R
--R (21)  0
--R
--R                                          Type: LiePolynomial(Symbol,Fraction(Integer))
--E 21

--S 22 of 28
test: Lpoly := Jacobi(r,s,t)
--R
--R
--R (22)  0
--R
--R                                          Type: LiePolynomial(Symbol,Fraction(Integer))
--E 22

--S 23 of 28
eval(p, a, p)$Lpoly
--R
--R
--R          2
--R (23)  [a b ]
--R
--R                                          Type: LiePolynomial(Symbol,Fraction(Integer))
--E 23

--S 24 of 28
eval(p, [a,b], [2*bb, 3*aa])$Lpoly
--R
--R
--R (24)  - 6[a b]
--R
--R                                          Type: LiePolynomial(Symbol,Fraction(Integer))
--E 24

--S 25 of 28
r: Lpoly := [p,c]
--R
--R
--R (25)  [a b c] + [a c b]
--R
--R                                          Type: LiePolynomial(Symbol,Fraction(Integer))
--E 25

--S 26 of 28
r1: Lpoly := eval(r, [a,b,c], [bb, cc, aa])$Lpoly
--R
--R
--R (26)  - [a b c]
--R
--R                                          Type: LiePolynomial(Symbol,Fraction(Integer))
--E 26

--S 27 of 28

```

```

r2: Lpoly := eval(r, [a,b,c], [cc, aa, bb])$Lpoly
--R
--R
--R (27) - [a c b]
--R
--R                                         Type: LiePolynomial(Symbol,Fraction(Integer))
--E 27

--S 28 of 28
r + r1 + r2
--R
--R
--R (28)  0
--R
--R                                         Type: LiePolynomial(Symbol,Fraction(Integer))
--E 28
)spool

)lisp (bye)

```

— LiePolynomial.help —

```

=====
LiePolynomial examples
=====

```

```

=====
Declaration of domains
=====

```

```

RN := Fraction Integer
    Fraction Integer
        Type: Domain

Lpoly := LiePolynomial(Symbol,RN)
        LiePolynomial(Symbol,Fraction Integer)
            Type: Domain

Dpoly := XDPOLY(Symbol,RN)
        XDistributedPolynomial(Symbol,Fraction Integer)
            Type: Domain

Lword := LyndonWord Symbol
        LyndonWord Symbol
            Type: Domain

```

```

=====
Initialisation
=====

```

```

a:Symbol := 'a
a
Type: Symbol

b:Symbol := 'b
b
Type: Symbol

c:Symbol := 'c
c
Type: Symbol

aa: Lpoly := a
[a]
Type: LiePolynomial(Symbol,Fraction Integer)

bb: Lpoly := b
[b]
Type: LiePolynomial(Symbol,Fraction Integer)

cc: Lpoly := c
[c]
Type: LiePolynomial(Symbol,Fraction Integer)

p : Lpoly := [aa,bb]
[a b]
Type: LiePolynomial(Symbol,Fraction Integer)

q : Lpoly := [p,bb]
2
[a b ]
Type: LiePolynomial(Symbol,Fraction Integer)

All the Lyndon words of order 4

liste : List Lword := LyndonWordsList([a,b], 4)
2 2 3 2 2 3
[[a],[b],[a b],[a b],[a b ],[a b],[a b ],[a b ]]
Type: List LyndonWord Symbol

r: Lpoly := p + q + 3*LiePoly(liste.4)$Lpoly
2 2
[a b] + 3[a b] + [a b ]
Type: LiePolynomial(Symbol,Fraction Integer)

s:Lpoly := [p,r]
2 2
- 3[a b a b] + [a b a b ]
Type: LiePolynomial(Symbol,Fraction Integer)

```

```
t:Lpoly := s + 2*LiePoly(liste.3) - 5*LiePoly(liste.5)
          2      2      2
2[a b] - 5[a b] - 3[a b a b] + [a b a b]
          Type: LiePolynomial(Symbol,Fraction Integer)
```

```
degree t
5
          Type: PositiveInteger
```

```
mirror t
          2      2      2
- 2[a b] - 5[a b] - 3[a b a b] + [a b a b]
          Type: LiePolynomial(Symbol,Fraction Integer)
```

```
=====
Jacobi Relation
=====
```

```
Jacobi(p: Lpoly, q: Lpoly, r: Lpoly): Lpoly == _
[ [p,q]\$Lpoly, r] + [ [q,r]\$Lpoly, p] + [ [r,p]\$Lpoly, q]
          Type: Void
```

```
=====
Tests
=====
```

```
test: Lpoly := Jacobi(a,b,b)
0
          Type: LiePolynomial(Symbol,Fraction Integer)
```

```
test: Lpoly := Jacobi(p,q,r)
0
          Type: LiePolynomial(Symbol,Fraction Integer)
```

```
test: Lpoly := Jacobi(r,s,t)
0
          Type: LiePolynomial(Symbol,Fraction Integer)
```

```
=====
Evaluation
=====
```

```
eval(p, a, p)$Lpoly
2
[a b]
          Type: LiePolynomial(Symbol,Fraction Integer)
```

```
eval(p, [a,b], [2*bb, 3*aa])$Lpoly
- 6[a b]
```



```

                                Type: LiePolynomial(Symbol,Fraction Integer)

r: Lpoly := [p,c]
[a b c] + [a c b]
                                Type: LiePolynomial(Symbol,Fraction Integer)

r1: Lpoly := eval(r, [a,b,c], [bb, cc, aa])$Lpoly
- [a b c]
                                Type: LiePolynomial(Symbol,Fraction Integer)

r2: Lpoly := eval(r, [a,b,c], [cc, aa, bb])$Lpoly
- [a c b]
                                Type: LiePolynomial(Symbol,Fraction Integer)

r + r1 + r2
0
                                Type: LiePolynomial(Symbol,Fraction Integer)

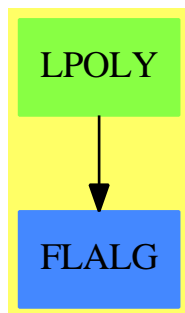
```

See Also:

- o)help LyndonWord
- o)help XDistributedPolynomial
- o)show LiePolynomial

—————

LiePolynomial (LPOLY)



Exports:

0	coef	coefficient	coefficients
coerce	construct	degree	eval
hash	latex	leadingCoefficient	leadingMonomial
leadingTerm	LiePoly	LiePolyIfCan	listOfTerms
lquo	map	mirror	monom
monomial?	monomials	numberOfMonomials	reductum
retract	retractIfCan	rquo	sample
subtractIfCan	trunc	varList	zero?
?^=?	?*?	?/?	?+?
?-?	-?	?=?	

— domain LPOLY LiePolynomial —

```

)abbrev domain LPOLY LiePolynomial
++ Author: Michel Petitot (petitot@lifl.fr).
++ Date Created: 91
++ Date Last Updated: 7 Juillet 92
++ Fix History: compilation v 2.1 le 13 dec 98
++ References:
++ Free Lie Algebras by C. Reutenauer (Oxford science publications).
++ Description:
++ This type supports Lie polynomials in Lyndon basis
++ see Free Lie Algebras by C. Reutenauer
++ (Oxford science publications).

LiePolynomial(VarSet:OrderedSet, R:CommutativeRing) : Public == Private where
  MAGMA    ==> Magma(VarSet)
  LWORD    ==> LyndonWord(VarSet)
  WORD     ==> OrderedFreeMonoid(VarSet)
  XDPOLY   ==> XDistributedPolynomial(VarSet,R)
  XRPOLY   ==> XRecursivePolynomial(VarSet,R)
  NNI      ==> NonNegativeInteger
  RN       ==> Fraction Integer
  EX       ==> OutputForm
  TERM     ==> Record(k: LWORD, c: R)

Public == Join(FreeLieAlgebra(VarSet,R), FreeModuleCat(R,LWORD)) with
  LiePolyIfCan: XDPOLY -> Union($, "failed")
    ++ \axiom{LiePolyIfCan(p)} returns \axiom{p} in Lyndon basis
    ++ if \axiom{p} is a Lie polynomial, otherwise \axiom{"failed"}
    ++ is returned.
  construct: (LWORD, LWORD) -> $
    ++ \axiom{construct(x,y)} returns the Lie bracket \axiom{[x,y]}.
  construct: (LWORD, $) -> $
    ++ \axiom{construct(x,y)} returns the Lie bracket \axiom{[x,y]}.
  construct: ($, LWORD) -> $
    ++ \axiom{construct(x,y)} returns the Lie bracket \axiom{[x,y]}.

```

```

Private == FreeModule1(R, LWORD) add
import(TERM)

--representation
Rep := List TERM

-- fonctions locales
cr1 : (LWORD, $ ) -> $
cr2 : ($, LWORD ) -> $
crw : (LWORD, LWORD) -> $      -- crochet de 2 mots de Lyndon
DPoly: LWORD -> XDPOLY
lquo1: (XRPOLY , LWORD) -> XRPOLY
lyndon: (LWORD, LWORD) -> $
makeLyndon: (LWORD, LWORD) -> LWORD
rquo1: (XRPOLY , LWORD) -> XRPOLY
RPoly: LWORD -> XRPOLY
eval1: (LWORD, VarSet, $) -> $      -- 08/03/98
eval2: (LWORD, List VarSet, List $) -> $      -- 08/03/98

-- Evaluation
eval1(lw,v,nv) ==      -- 08/03/98
  not member?(v, varList(lw)$LWORD) => LiePoly lw
  (s := retractIfCan(lw)$LWORD) case VarSet =>
    if (s::VarSet) = v then nv else LiePoly lw
  l: LWORD := left lw
  r: LWORD := right lw
  construct(eval1(l,v,nv), eval1(r,v,nv))

eval2(lw,lv,lnv) ==      -- 08/03/98
  p: Integer
  (s := retractIfCan(lw)$LWORD) case VarSet =>
    p := position(s::VarSet, lv)$List(VarSet)
    if p=0 then lw::$ else elt(lnv,p)$List($
  l: LWORD := left lw
  r: LWORD := right lw
  construct(eval2(l,lv,lnv), eval2(r,lv,lnv))

eval(p:$, v: VarSet, nv: $): $ ==      -- 08/03/98
  +/ [t.c * eval1(t.k, v, nv) for t in p]

eval(p:$, lv: List(VarSet), lnv: List($)): $ ==      -- 08/03/98
  +/ [t.c * eval2(t.k, lv, lnv) for t in p]

lquo1(p,lw) ==
  constant? p => 0$XRPOLY
  retractable? lw => lquo(p, retract lw)$XRPOLY
  lquo1(lquo1(p, left lw),right lw) - lquo1(lquo1(p, right lw),left lw)
rquo1(p,lw) ==
  constant? p => 0$XRPOLY

```

```

    retractable? lw => rquo(p, retract lw)$XRPOLY
    rquo1(rquo1(p, left lw),right lw) - rquo1(rquo1(p, right lw),left lw)

coef(p, lp) == coef(p, lp::XRPOLY)$XRPOLY

lquo(p, lp) ==
  lp = 0 => 0$XRPOLY
  +/ [t.c * lquo1(p,t.k) for t in lp]

rquo(p, lp) ==
  lp = 0 => 0$XRPOLY
  +/ [t.c * rquo1(p,t.k) for t in lp]

LiePolyIfCan p ==          -- inefficace a cause de la rep. de XDPOLY
  not quasiRegular? p => "failed"
  p1: XDPOLY := p ; r:$ := 0
  while p1 ^= 0 repeat
    t: Record(k:WORD, c:R) := mindegTerm p1
    w: WORD := t.k; coef:R := t.c
    (l := lyndonIfCan(w)$LWORD) case "failed" => return "failed"
    lp:$ := coef * LiePoly(l::LWORD)
    r := r + lp
    p1 := p1 - lp::XDPOLY
  r

--definitions locales
makeLyndon(u,v) == (u::MAGMA * v::MAGMA) pretend LWORD

crw(u,v) ==                -- u et v sont des mots de Lyndon
  u = v => 0
  lexico(u,v) => lyndon(u,v)
  - lyndon (v,u)

lyndon(u,v) ==              -- u et v sont des mots de Lyndon tq u < v
  retractable? u => monom(makeLyndon(u,v),1)
  u1: LWORD := left u
  u2: LWORD := right u
  lexico(u2,v) => cr1(u1, lyndon(u2,v)) + cr2(lyndon(u1,v), u2)
  monom(makeLyndon(u,v),1)

cr1 (l, p) ==
  +/[t.c * crw(l, t.k) for t in p]

cr2 (p, l) ==
  +/[t.c * crw(t.k, l) for t in p]

DPoly w ==
  retractable? w => retract(w) :: XDPOLY
  l:XDPOLY := DPoly left w
  r:XDPOLY := DPoly right w

```

```

l*r - r*l

RPoly w ==
  retractable? w => retract(w) :: XRPOLY
  l:XRPOLY := RPoly left w
  r:XRPOLY := RPoly right w
  l*r - r*l

-- definitions

coerce(v:VarSet) == monom(v::LWORD , 1)

construct(x:$ , y:$):$ ==
  +/[t.c * cr1(t.k, y) for t in x]

construct(l:LWORD , p:$):$ == cr1(l,p)
construct(p:$ , l:LWORD):$ == cr2(p,l)
construct(u:LWORD , v:LWORD):$ == crw(u,v)

coerce(p:$):XDPOLY ==
  +/ [t.c * DPoly(t.k) for t in p]

coerce(p:$):XRPOLY ==
  +/ [t.c * RPoly(t.k) for t in p]

LiePoly(l) == monom(l,1)

varList p ==
  le : List VarSet := "setUnion"/[varList(t.k)$LWORD for t in p]
  sort(le)$List(VarSet)

mirror p ==
  [[t.k, (odd? length t.k => t.c; -t.c)]$TERM for t in p]

trunc(p, n) ==
  degree(p) > n => trunc( reductum p , n)
  p

degree p ==
  null p => 0
  length( p.first.k)$LWORD

-- listOfTerms p == p pretend List TERM

--
--  coerce(x) : EX ==
--    null x => (0$R) :: EX
--    le : List EX := nil
--    for rec in x repeat
--      rec.c = 1$R => le := cons(rec.k :: EX, le)
--      le := cons(mkBinary("*"::EX, rec.c :: EX, rec.k :: EX), le)

```

```
--      1 = #le => first le
--      mkNary("+" :: EX,le)
```

— LPOLY.dotabb —

```
"LPOLY" [color="#88FF44",href="bookvol10.3.pdf#nameddest=LPOLY"]
"FLALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLALG"]
"LPOLY" -> "FLALG"
```

domain LSQM LieSquareMatrix

— LieSquareMatrix.input —

```
)set break resume
)sys rm -f LieSquareMatrix.output
)spool LieSquareMatrix.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LieSquareMatrix
--R
--R LieSquareMatrix(n: PositiveInteger,R: CommutativeRing) is a domain constructor
--R Abbreviation for LieSquareMatrix is LSQM
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for LSQM
--R
--R----- Operations -----
--R ??? : (R,%) -> %          ??? : (%,R) -> %
--R ??? : (%,%) -> %          ??? : (Integer,%) -> %
--R ??? : (PositiveInteger,%) -> %    ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %          ?-? : (%,%) -> %
--R -? : % -> %              ?=? : (%,%) -> Boolean
--R D : % -> % if R has DIFRING      D : (%,(R -> R)) -> %
--R 1 : () -> %                0 : () -> %
--R ??? : (%,PositiveInteger) -> %    alternative? : () -> Boolean
--R antiAssociative? : () -> Boolean    antiCommutative? : () -> Boolean
--R antiCommutator : (%,%) -> %        antisymmetric? : % -> Boolean
--R apply : (Matrix(R),%) -> %        associative? : () -> Boolean
```

```

--R associator : (%,%,%) -> %
--R coerce : % -> Matrix(R)
--R coerce : Integer -> %
--R commutative? : () -> Boolean
--R convert : % -> Vector(R)
--R coordinates : % -> Vector(R)
--R diagonal? : % -> Boolean
--R diagonalProduct : % -> R
--R elt : (%,Integer,Integer) -> R
--R empty : () -> %
--R eq? : (%,%) -> Boolean
--R hash : % -> SingleInteger
--R jordanAdmissible? : () -> Boolean
--R latex : % -> String
--R leftDiscriminant : Vector(%) -> R
--R leftNorm : % -> R
--R leftTraceMatrix : () -> Matrix(R)
--R lieAlgebra? : () -> Boolean
--R map : ((R -> R),%) -> %
--R matrix : List(List(R)) -> %
--R maxRowIndex : % -> Integer
--R minRowIndex : % -> Integer
--R nrows : % -> NonNegativeInteger
--R powerAssociative? : () -> Boolean
--R rank : () -> PositiveInteger
--R represents : Vector(R) -> %
--R rightAlternative? : () -> Boolean
--R rightNorm : % -> R
--R rightTraceMatrix : () -> Matrix(R)
--R scalarMatrix : R -> %
--R square? : % -> Boolean
--R trace : % -> R
--R ~=?: (%,%) -> Boolean
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?? : (DirectProduct(n,R),%) -> DirectProduct(n,R)
--R ?? : (%,DirectProduct(n,R)) -> DirectProduct(n,R)
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%,Integer) -> % if R has FIELD
--R ??? : (%,NonNegativeInteger) -> %
--R ?/? : (%,R) -> % if R has FIELD
--R D : (%,NonNegativeInteger) -> % if R has DIFRING
--R D : (%,Symbol) -> % if R has PDRING(SYMBOL)
--R D : (%,List(Symbol)) -> % if R has PDRING(SYMBOL)
--R D : (%,Symbol,NonNegativeInteger) -> % if R has PDRING(SYMBOL)
--R D : (%,List(Symbol),List(NonNegativeInteger)) -> % if R has PDRING(SYMBOL)
--R D : (%,(R -> R),NonNegativeInteger) -> %
--R ?? : (%,NonNegativeInteger) -> %
--R any? : ((R -> Boolean),%) -> Boolean if $ has finiteAggregate
--R associatorDependence : () -> List(Vector(R)) if R has INTDOM
--R characteristic : () -> NonNegativeInteger
--R basis : () -> Vector(%)
--R coerce : R -> %
--R coerce : % -> OutputForm
--R commutator : (%,%) -> %
--R convert : Vector(R) -> %
--R copy : % -> %
--R diagonalMatrix : List(R) -> %
--R ?.? : (%,Integer) -> R
--R elt : (%,Integer,Integer,R) -> R
--R empty? : % -> Boolean
--R flexible? : () -> Boolean
--R jacobIdentity? : () -> Boolean
--R jordanAlgebra? : () -> Boolean
--R leftAlternative? : () -> Boolean
--R leftDiscriminant : () -> R
--R leftTrace : % -> R
--R lieAdmissible? : () -> Boolean
--R listOfLists : % -> List(List(R))
--R map : (((R,R) -> R),%,%) -> %
--R maxColIndex : % -> Integer
--R minColIndex : % -> Integer
--R ncols : % -> NonNegativeInteger
--R one? : % -> Boolean
--R qelt : (%,Integer,Integer) -> R
--R recip : % -> Union(%, "failed")
--R retract : % -> R
--R rightDiscriminant : () -> R
--R rightTrace : % -> R
--R sample : () -> %
--R someBasis : () -> Vector(%)
--R symmetric? : % -> Boolean
--R zero? : % -> Boolean

```

```

--R coerce : Fraction(Integer) -> % if R has RETRACT(FRAC(INT))
--R column : (%,Integer) -> DirectProduct(n,R)
--R conditionsForIdempotents : Vector(%) -> List(Polynomial(R))
--R conditionsForIdempotents : () -> List(Polynomial(R))
--R coordinates : (%,Vector(%)) -> Vector(R)
--R coordinates : (Vector(%),Vector(%)) -> Matrix(R)
--R coordinates : Vector(%) -> Matrix(R)
--R count : (R,%) -> NonNegativeInteger if $ has finiteAggregate and R has SETCAT
--R count : ((R -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R determinant : % -> R if R has commutative(*)
--R diagonal : % -> DirectProduct(n,R)
--R differentiate : % -> % if R has DIFRING
--R differentiate : (% ,NonNegativeInteger) -> % if R has DIFRING
--R differentiate : (% ,Symbol) -> % if R has PDRING(SYMBOL)
--R differentiate : (% ,List(Symbol)) -> % if R has PDRING(SYMBOL)
--R differentiate : (% ,Symbol,NonNegativeInteger) -> % if R has PDRING(SYMBOL)
--R differentiate : (% ,List(Symbol),List(NonNegativeInteger)) -> % if R has PDRING(SYMBOL)
--R differentiate : (% ,(R -> R),NonNegativeInteger) -> %
--R differentiate : (% ,(R -> R)) -> %
--R eval : (% ,List(R),List(R)) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (% ,R,R) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (% ,Equation(R)) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (% ,List(Equation(R))) -> % if R has EVALAB(R) and R has SETCAT
--R every? : ((R -> Boolean),%) -> Boolean if $ has finiteAggregate
--R exquo : (% ,R) -> Union(% ,"failed") if R has INTDOM
--R inverse : % -> Union(% ,"failed") if R has FIELD
--R leftCharacteristicPolynomial : % -> SparseUnivariatePolynomial(R)
--R leftMinimalPolynomial : % -> SparseUnivariatePolynomial(R) if R has INTDOM
--R leftPower : (% ,PositiveInteger) -> %
--R leftRankPolynomial : () -> SparseUnivariatePolynomial(Polynomial(R)) if R has FIELD
--R leftRecip : % -> Union(% ,"failed") if R has INTDOM
--R leftRegularRepresentation : (% ,Vector(%)) -> Matrix(R)
--R leftRegularRepresentation : % -> Matrix(R)
--R leftTraceMatrix : Vector(%) -> Matrix(R)
--R leftUnit : () -> Union(% ,"failed") if R has INTDOM
--R leftUnits : () -> Union(Record(particular: % ,basis: List(%)) ,"failed") if R has INTDOM
--R less? : (% ,NonNegativeInteger) -> Boolean
--R map! : ((R -> R),%) -> % if $ has shallowlyMutable
--R member? : (R,%) -> Boolean if $ has finiteAggregate and R has SETCAT
--R members : % -> List(R) if $ has finiteAggregate
--R minordet : % -> R if R has commutative(*)
--R more? : (% ,NonNegativeInteger) -> Boolean
--R noncommutativeJordanAlgebra? : () -> Boolean
--R nullSpace : % -> List(DirectProduct(n,R)) if R has INTDOM
--R nullity : % -> NonNegativeInteger if R has INTDOM
--R parts : % -> List(R) if $ has finiteAggregate
--R plenaryPower : (% ,PositiveInteger) -> %
--R rank : % -> NonNegativeInteger if R has INTDOM
--R reducedSystem : Matrix(%) -> Matrix(R)
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(R),vec: Vector(R))

```



```

--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) i
--R reducedSystem : Matrix(%) -> Matrix(Integer) if R has LINEXP(INT)
--R represents : (Vector(R),Vector(%)) -> %
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retract : % -> Integer if R has RETRACT(INT)
--R retractIfCan : % -> Union(R,"failed")
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(Integer,"failed") if R has RETRACT(INT)
--R rightCharacteristicPolynomial : % -> SparseUnivariatePolynomial(R)
--R rightDiscriminant : Vector(%) -> R
--R rightMinimalPolynomial : % -> SparseUnivariatePolynomial(R) if R has INTDOM
--R rightPower : (% ,PositiveInteger) -> %
--R rightRankPolynomial : () -> SparseUnivariatePolynomial(Polynomial(R)) if R has FIELD
--R rightRecip : % -> Union(%,"failed") if R has INTDOM
--R rightRegularRepresentation : (% ,Vector(%)) -> Matrix(R)
--R rightRegularRepresentation : % -> Matrix(R)
--R rightTraceMatrix : Vector(%) -> Matrix(R)
--R rightUnit : () -> Union(%,"failed") if R has INTDOM
--R rightUnits : () -> Union(Record(particular: %,basis: List(%)),"failed") if R has INTDOM
--R row : (% ,Integer) -> DirectProduct(n,R)
--R rowEchelon : % -> % if R has EUCLDOM
--R size? : (% ,NonNegativeInteger) -> Boolean
--R structuralConstants : Vector(%) -> Vector(Matrix(R))
--R structuralConstants : () -> Vector(Matrix(R))
--R subtractIfCan : (% ,%) -> Union(%,"failed")
--R unit : () -> Union(%,"failed") if R has INTDOM
--R
--E 1

)spool
)lisp (bye)

```

— LieSquareMatrix.help —

=====

LieSquareMatrix examples

=====

LieSquareMatrix(n,R) implements the Lie algebra of the n by n matrices over the commutative ring R.

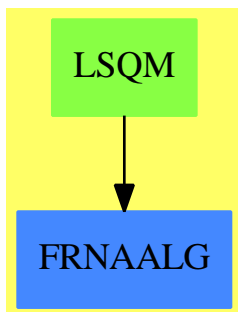
The Lie bracket (commutator) of the algebra is given by

$$a*b := (a * \$\$MATRIX(n,R) b - b * \$\$MATRIX(n,R) a)$$

where $\$\$MATRIX(n,R)$ is the usual matrix multiplication.

See Also:

o)show LieSquareMatrix

LieSquareMatrix (LSQM)**See**

- ⇒ “AssociatedLieAlgebra” (LIE) [2](#) on page [228](#)
- ⇒ “AssociatedJordanAlgebra” (JORDAN) [2](#) on page [222](#)

Exports:

0	1	alternative?
antiAssociative?	antiCommutator	antisymmetric?
any?	apply	associative?
associator	associatorDependence	basis
characteristic	coerce	column
commutative?	commutator	conditionsForIdempotents
convert	coordinates	copy
count	D	determinant
diagonal	diagonal?	diagonalMatrix
diagonalProduct	differentiate	elt
empty	empty?	eq?
eval	every?	exquo
flexible?	hash	inverse
jacobiIdentity?	jordanAdmissible?	jordanAlgebra?
latex	leftAlternative?	leftCharacteristicPolynomial
leftDiscriminant	leftDiscriminant	leftMinimalPolynomial
leftNorm	leftPower	leftRankPolynomial
leftRecip	leftRegularRepresentation	leftRegularRepresentation
leftTrace	leftTraceMatrix	leftUnit
leftUnits	less?	lieAdmissible?
lieAlgebra?	listOfLists	map
map!	matrix	maxColIndex
maxRowIndex	member?	members
minColIndex	minordet	minRowIndex
more?	ncols	noncommutativeJordanAlgebra?
nrows	nullSpace	nullity
one?	parts	plenaryPower
powerAssociative?	qelt	rank
recip	reducedSystem	represents
retract	retractIfCan	rightAlternative?
rightCharacteristicPolynomial	rightDiscriminant	rightMinimalPolynomial
rightNorm	rightPower	rightRankPolynomial
rightRecip	rightRegularRepresentation	rightTrace
rightTraceMatrix	rightUnit	rightUnits
row	rowEchelon	sample
scalarMatrix	size?	someBasis
square?	structuralConstants	structuralConstants
subtractIfCan	symmetric?	trace
unit	zero?	#?
?~=?	?*?	?**?
?+?	?-?	-?
?/?	?=?	?^?
?.?		

```

)abbrev domain LSQM LieSquareMatrix
++ Author: J. Grabmeier
++ Date Created: 07 March 1991
++ Date Last Updated: 08 March 1991
++ Description:
++ LieSquareMatrix(n,R) implements the Lie algebra of the n by n
++ matrices over the commutative ring R.
++ The Lie bracket (commutator) of the algebra is given by\br
++ \spad{a*b := (a *$SQMATRIX(n,R) b - b *$SQMATRIX(n,R) a)},\br
++ where \spadfun{*$SQMATRIX(n,R)} is the usual matrix multiplication.

LieSquareMatrix(n,R): Exports == Implementation where

  n      : PositiveInteger
  R      : CommutativeRing

  Row ==> DirectProduct(n,R)
  Col ==> DirectProduct(n,R)

  Exports ==> Join(SquareMatrixCategory(n,R,Row,Col), CoercibleTo Matrix R,
    FramedNonAssociativeAlgebra R) --with

  Implementation ==> AssociatedLieAlgebra (R,SquareMatrix(n, R)) add

  Rep := AssociatedLieAlgebra (R,SquareMatrix(n, R))
    -- local functions
  n2 : PositiveInteger := n*n

  convDM : DirectProduct(n2,R) -> %
  conv : DirectProduct(n2,R) -> SquareMatrix(n,R)
    --++ converts n2-vector to (n,n)-matrix row by row
  conv v ==
    cond : Matrix(R) := new(n,n,0$R)$Matrix(R)
    z : Integer := 0
    for i in 1..n repeat
      for j in 1..n repeat
        z := z+1
        setelt(cond,i,j,v.z)
    squareMatrix(cond)$SquareMatrix(n, R)

  coordinates(a:%,b:Vector(%)):Vector(R) ==
    -- only valid for b canonicalBasis
    res : Vector R := new(n2,0$R)
    z : Integer := 0
    for i in 1..n repeat
      for j in 1..n repeat
        z := z+1
        res.z := elt(a,i,j)$%
    res

```

```

convDM v ==
  sq := conv v
  coerce(sq)$Rep :: %

basis() ==
  n2 : PositiveInteger := n*n
  ldp : List DirectProduct(n2,R) :=
    [unitVector(i::PositiveInteger)$DirectProduct(n2,R) for i in 1..n2]
  res:Vector % := vector map(convDM,_
    ldp)$ListFunctions2(DirectProduct(n2,R), %)

someBasis() == basis()
rank() == n*n

-- transpose: % -> %
-- ++ computes the transpose of a matrix
-- squareMatrix: Matrix R -> %
-- ++ converts a Matrix to a LieSquareMatrix
-- coerce: % -> Matrix R
-- ++ converts a LieSquareMatrix to a Matrix
-- symdecomp : % -> Record(sym:%,antisym:%)
-- if R has commutative("*") then
--   minorsVect: -> Vector(Union(R,"uncomputed")) --range: 1..2**n-1
-- if R has commutative("*") then central
-- if R has commutative("*") and R has unitsKnown then unitsKnown

-----

— LSQM.dotabb —

"LSQM" [color="#88FF44",href="bookvol10.3.pdf#nameddest=LSQM"]
"FRNAALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRNAALG"]
"LSQM" -> "FRNAALG"

-----

```

domain LODO LinearOrdinaryDifferentialOperator

— LinearOrdinaryDifferentialOperator.input —

```
)set break resume
```



```

phi2 == phi1 *x**(n+3)
--R
--R
--R                                          Type: Void
--E 7

--S 8 of 16
phi3 == retract(phi2)@(POLY INT)
--R
--R
--R                                          Type: Void
--E 8

--S 9 of 16
pans == phi3 ::UP(x,POLY INT)
--R
--R
--R                                          Type: Void
--E 9

--S 10 of 16
pans1 == [coefficient(pans, (n+3-i) :: NNI) for i in 2..n+1]
--R
--R
--R                                          Type: Void
--E 10

--S 11 of 16
leq == solve(pans1,[subscript(s,[i]) for i in 1..n])
--R
--R
--R                                          Type: Void
--E 11

--S 12 of 16
leq
--R
--R   Compiling body of rule n to compute value of type PositiveInteger
--R   Compiling body of rule phi to compute value of type Expression(
--R       Integer)
--R   Compiling body of rule phi1 to compute value of type Expression(
--R       Integer)
--R   Compiling body of rule phi2 to compute value of type Expression(
--R       Integer)
--R   Compiling body of rule phi3 to compute value of type Polynomial(
--R       Integer)
--R   Compiling body of rule pans to compute value of type
--R       UnivariatePolynomial(x,Polynomial(Integer))
--R   Compiling body of rule pans1 to compute value of type List(
--R       Polynomial(Integer))
--R   Compiling body of rule leq to compute value of type List(List(
--R       Equation(Fraction(Polynomial(Integer))))))
--R   Compiling function G3491 with type Integer -> Boolean
--R
--R   (12)

```

```

--R
--R      2      3      2
--R      s G      3s H + s G + 6s G      (9s G + 54s )H + s G + 18s G + 72s G
--R      0      0      0      0      0      0      0      0      0
--R      [[s = ---, s = -----, s = -----]]
--R      1      3      2      18      3      162
--R      Type: List(List(Equation(Fraction(Polynomial(Integer)))))
--E 12

```

```

--S 13 of 16

```

```

n==4

```

```

--R
--R      Compiled code for n has been cleared.
--R      Compiled code for leq has been cleared.
--R      Compiled code for pans1 has been cleared.
--R      Compiled code for phi2 has been cleared.
--R      Compiled code for phi has been cleared.
--R      Compiled code for phi3 has been cleared.
--R      Compiled code for phi1 has been cleared.
--R      Compiled code for pans has been cleared.
--R      1 old definition(s) deleted for function or rule n
--R
--R      Type: Void
--E 13

```

```

--S 14 of 16

```

```

leq

```

```

--R
--R      Compiling body of rule n to compute value of type PositiveInteger
--R      Compiling body of rule phi to compute value of type Expression(
--R      Integer)
--R      Compiling body of rule phi1 to compute value of type Expression(
--R      Integer)
--R      Compiling body of rule phi2 to compute value of type Expression(
--R      Integer)
--R      Compiling body of rule phi3 to compute value of type Polynomial(
--R      Integer)
--R      Compiling body of rule pans to compute value of type
--R      UnivariatePolynomial(x,Polynomial(Integer))
--R      Compiling body of rule pans1 to compute value of type List(
--R      Polynomial(Integer))
--R      Compiling body of rule leq to compute value of type List(List(
--R      Equation(Fraction(Polynomial(Integer)))))
--R
--R      (14)
--R      [
--R
--R      2
--R      s G      3s H + s G + 6s G
--R      0      0      0      0
--R      [s = ---, s = -----,
--R      1      3      2      18      3      2
--R

```



```

--R      (9s G + 54s )H + s G + 18s G + 72s G
--R      0      0      0      0      0
--R      s = -----,
--R      3      162
--R
--R      s =
--R      4
--R      2      2      4      3      2
--R      27s H + (18s G + 378s G + 1296s )H + s G + 36s G + 396s G
--R      0      0      0      0      0      0      0
--R      +
--R      1296s G
--R      0
--R      /
--R      1944
--R      ]
--R      ]
--R
--R      Type: List(List(Equation(Fraction(Polynomial(Integer))))))
--E 14

```

```

--S 15 of 16

```

```

n==7

```

```

--R
--R      Compiled code for n has been cleared.
--R      Compiled code for leq has been cleared.
--R      Compiled code for pans1 has been cleared.
--R      Compiled code for phi2 has been cleared.
--R      Compiled code for phi has been cleared.
--R      Compiled code for phi3 has been cleared.
--R      Compiled code for phi1 has been cleared.
--R      Compiled code for pans has been cleared.
--R      1 old definition(s) deleted for function or rule n

```

```

Type: Void

```

```

--E 15

```

```

--S 16 of 16

```

```

leq

```

```

--R
--R      Compiling body of rule n to compute value of type PositiveInteger
--R      Compiling body of rule phi to compute value of type Expression(
--R      Integer)
--R      Compiling body of rule phi1 to compute value of type Expression(
--R      Integer)
--R      Compiling body of rule phi2 to compute value of type Expression(
--R      Integer)
--R      Compiling body of rule phi3 to compute value of type Polynomial(
--R      Integer)
--R      Compiling body of rule pans to compute value of type
--R      UnivariatePolynomial(x,Polynomial(Integer))
--R      Compiling body of rule pans1 to compute value of type List(

```

```

--R      Polynomial(Integer))
--R      Compiling body of rule leq to compute value of type List(List(
--R      Equation(Fraction(Polynomial(Integer)))))
--R
--R      (16)
--R      [
--R      
$$\frac{s^2 G^2 + 3s H + s^2 G^2 + 6s G}{0 \quad 0 \quad 0 \quad 0},$$

--R      
$$[s = \frac{1}{3}, s = \frac{2}{18},$$

--R      
$$(9s^3 G + 54s^2)H + s^3 G^2 + 18s^2 G + 72s G$$

--R      
$$s = \frac{0 \quad 0 \quad 0 \quad 0 \quad 0}{3 \quad 162},$$

--R
--R      
$$s = \frac{27s^2 H^2 + (18s^2 G + 378s G + 1296s)H + s^4 G + 36s^3 G + 396s^2 G + 1296s G}{0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0}$$

--R      /
--R      1944
--R      ,
--R
--R      
$$s = \frac{(135s^2 G + 2268s)H + (30s^3 G + 1350s^2 G + 16416s G + 38880s)H + s^5 G + 60s^4 G + 1188s^3 G + 9504s^2 G + 25920s G}{0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0}$$

--R      /
--R      29160
--R      ,
--R
--R      
$$s = \frac{405s^3 H + (405s^2 G + 18468s G + 174960s)H + (45s^4 G + 3510s^3 G + 88776s^2 G + 777600s G + 1166400s)H + s^6 G}{0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0}$$

--R

```

```

--R      0      0      0      0      0      0
--R      +
--R      5      4      3      2
--R      90s G + 2628s G + 27864s G + 90720s G
--R      0      0      0      0
--R      /
--R      524880
--R      ,
--R      s =
--R      7
--R      3
--R      (2835s G + 91854s )H
--R      0      0
--R      +
--R      3      2      2
--R      (945s G + 81648s G + 2082996s G + 14171760s )H
--R      0      0      0      0
--R      +
--R      5      4      3      2
--R      (63s G + 7560s G + 317520s G + 5554008s G + 34058880s G)H
--R      0      0      0      0      0
--R      +
--R      7      6      5      4      3      2
--R      s G + 126s G + 4788s G + 25272s G - 1744416s G - 26827200s G
--R      0      0      0      0      0      0
--R      +
--R      - 97977600s G
--R      0
--R      /
--R      11022480
--R      ]
--R      ]
--R      Type: List(List(Equation(Fraction(Polynomial(Integer))))))
--E 16
)spool

```

— LinearOrdinaryDifferentialOperator.help —

```

=====
LinearOrdinaryDifferentialOperator examples
=====

```

LinearOrdinaryDifferentialOperator(A, diff) is the domain of linear ordinary differential operators with coefficients in a ring A with a given derivation.

```
=====
Differential Operators with Series Coefficients
=====
```

Problem:

Find the first few coefficients of $\exp(x)/x^i$ of Dop phi where

```
Dop := D^3 + G/x^2 * D + H/x^3 - 1
phi := sum(s[i]*exp(x)/x^i, i = 0..)
```

Solution:

Define the differential.

```
Dx: LODO(EXPR INT, f +-> D(f, x))
      Type: Void

Dx := D()
D
Type: LinearOrdinaryDifferentialOperator(Expression Integer,
theMap LAMBDA-CLOSURE(NIL,NIL,NIL,G1404 envArg,
SPADCALL(G1404,QUOTE x,
ELT(*1;anonymousFunction;0;frame0;internal;MV,0))))
```

Now define the differential operator Dop.

```
Dop:= Dx^3 + G/x^2*Dx + H/x^3 - 1
      3
      G      - x  + H
D  + -- D + -----
      2          3
      x          x
Type: LinearOrdinaryDifferentialOperator(Expression Integer,
theMap LAMBDA-CLOSURE(NIL,NIL,NIL,G1404 envArg,
SPADCALL(G1404,QUOTE x,
ELT(*1;anonymousFunction;0;frame0;internal;MV,0))))
```

n == 3

Type: Void

```
phi == reduce(+,[subscript(s,[i])*exp(x)/x^i for i in 0..n])
      Type: Void
```

```
phi1 == Dop(phi) / exp x
      Type: Void
```

```
phi2 == phi1 *x**(n+3)
      Type: Void
```

```

phi3 == retract(phi2)@(POLY INT)
                                     Type: Void

pans == phi3 ::UP(x,POLY INT)
                                     Type: Void

pans1 == [coefficient(pans, (n+3-i) :: NNI) for i in 2..n+1]
                                     Type: Void

leq == solve(pans1,[subscript(s,[i]) for i in 1..n])
                                     Type: Void

```

Evaluate this for several values of n.

```

leq
      2      3      2
      s G    3s H + s G + 6s G    (9s G + 54s )H + s G + 18s G + 72s G
      0      0      0      0      0      0      0      0      0
[[s = ---, s = ---, s = ---]]
  1   3   2      18      3      162
                                     Type: List List Equation Fraction Polynomial Integer

```

```

n==4
                                     Type: Void

```

```

leq
[
      2
      s G    3s H + s G + 6s G
      0      0      0      0
[s = ---, s = ---,
  1   3   2      18
      3      2
      (9s G + 54s )H + s G + 18s G + 72s G
      0      0      0      0      0
s = ---,
  3      162
      s =
      4
      2      2      4      3      2
      27s H + (18s G + 378s G + 1296s )H + s G + 36s G + 396s G
      0      0      0      0      0      0      0
      +
      1296s G
      0
      /
      1944
]
]

```

Type: List List Equation Fraction Polynomial Integer

n==7

Type: Void

leq

[

$$\begin{aligned}
 & \begin{array}{c} s^2 G^2 \\ 0 \end{array} \quad \begin{array}{c} 3s^2 H + s^2 G^2 + 6s^2 G^2 \\ 0 \end{array} \\
 [s = \frac{\quad}{1^3}, s = \frac{\quad}{2^{18}}, \\
 & \begin{array}{c} (9s^3 G + 54s^2)H + s^3 G^2 + 18s^2 G^2 + 72s^2 G^2 \\ 0 \end{array} \\
 s = \frac{\quad}{3^{162}}, \\
 & s = \\
 & \begin{array}{c} 27s^2 H + (18s^2 G + 378s^2 G + 1296s^2)H + s^4 G^2 + 36s^3 G^2 + 396s^2 G^2 \\ 0 \end{array} \\
 + \\
 & \begin{array}{c} 1296s^2 G \\ 0 \end{array} \\
 / \\
 & 1944 \\
 , \\
 & s = \\
 & \begin{array}{c} (135s^2 G + 2268s^2)H + (30s^3 G + 1350s^2 G + 16416s^2 G + 38880s^2)H \\ 0 \end{array} \\
 + \\
 & \begin{array}{c} s^5 G + 60s^4 G + 1188s^3 G + 9504s^2 G + 25920s^2 G \\ 0 \end{array} \\
 / \\
 & 29160 \\
 , \\
 & s = \\
 & \begin{array}{c} 405s^3 H + (405s^2 G + 18468s^2 G + 174960s^2)H \\ 0 \end{array} \\
 + \\
 & \begin{array}{c} 4 \quad 3 \quad 2 \quad 6 \end{array}
 \end{aligned}$$

```

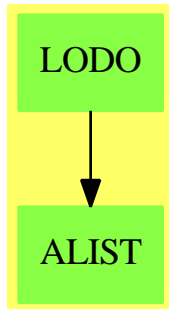
      (45s G + 3510s G + 88776s G + 777600s G + 1166400s )H + s G
      0          0          0          0          0          0
+
      5      4      3      2
    90s G + 2628s G + 27864s G + 90720s G
      0          0          0          0
/
524880
,
s =
7
      3
    (2835s G + 91854s )H
      0          0
+
      3      2      2
    (945s G + 81648s G + 2082996s G + 14171760s )H
      0          0          0          0
+
      5      4      3      2
    (63s G + 7560s G + 317520s G + 5554008s G + 34058880s G)H
      0          0          0          0          0
+
      7      6      5      4      3      2
    s G + 126s G + 4788s G + 25272s G - 1744416s G - 26827200s G
      0          0          0          0          0          0
+
    - 97977600s G
      0
/
11022480
]
]

```

Type: List List Equation Fraction Polynomial Integer

See Also:

o)show LinearOrdinaryDifferentialOperator

LinearOrdinaryDifferentialOperator (LODO)**See**⇒ “LinearOrdinaryDifferentialOperator1” (LODO1) [13](#) on page [1479](#)⇒ “LinearOrdinaryDifferentialOperator2” (LODO2) [13](#) on page [1491](#)**Exports:**

0	1	adjoint
apply	characteristic	coefficient
coefficients	coerce	content
D	degree	directSum
exquo	hash	latex
leadingCoefficient	leftDivide	leftExactQuotient
leftExtendedGcd	leftGcd	leftLcm
leftQuotient	leftRemainder	minimumDegree
monicLeftDivide	monicRightDivide	monomial
one?	primitivePart	recip
reductum	retract	retractIfCan
rightDivide	rightExactQuotient	rightExtendedGcd
rightGcd	rightLcm	rightQuotient
rightRemainder	sample	subtractIfCan
symmetricPower	symmetricProduct	symmetricSquare
zero?	?*?	?**?
?+?	?-?	-?
?=?	?^?	?..?
?~=?		

— domain LODO LinearOrdinaryDifferentialOperator —

```

)abbrev domain LODO LinearOrdinaryDifferentialOperator
++ Author: Manuel Bronstein
++ Date Created: 9 December 1993
++ Date Last Updated: 15 April 1994
++ Description:
++ \spad{LinearOrdinaryDifferentialOperator} defines a ring of
++ differential operators with coefficients in a ring A with a given

```



```

++ derivation.
++ Multiplication of operators corresponds to functional composition:\br
++ \spad{(L1 * L2).(f) = L1 L2 f}

LinearOrdinaryDifferentialOperator(A:Ring, diff: A -> A):
  LinearOrdinaryDifferentialOperatorCategory A
  == SparseUnivariateSkewPolynomial(A, 1, diff) add
  Rep := SparseUnivariateSkewPolynomial(A, 1, diff)

  outputD := "D"@String :: Symbol :: OutputForm

  coerce(1:%):OutputForm == outputForm(1, outputD)
  elt(p:%, a:A):A        == apply(p, 0, a)

  if A has Field then
    import LinearOrdinaryDifferentialOperatorsOps(A, %)

    symmetricProduct(a, b) == symmetricProduct(a, b, diff)
    symmetricPower(a, n)   == symmetricPower(a, n, diff)
    directSum(a, b)        == directSum(a, b, diff)

  -----

  — LODO.dotabb —

"LODO" [color="#88FF44",href="bookvol10.3.pdf#nameddest=LODO"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"LODO" -> "ALIST"

```

domain LODO1 LinearOrdinaryDifferentialOperator1

— LinearOrdinaryDifferentialOperator1.input —

```

)set break resume
)sys rm -f LinearOrdinaryDifferentialOperator1.output
)spool LinearOrdinaryDifferentialOperator1.output
)set message test on
)set message auto off
)clear all
--S 1 of 20
RFZ := Fraction UnivariatePolynomial('x, Integer)
--R

```

```

--R
--R (1) Fraction(UnivariatePolynomial(x,Integer))
--R
--R                                          Type: Domain
--E 1

--S 2 of 20
x : RFZ := 'x
--R
--R
--R (2) x
--R
--R                                          Type: Fraction(UnivariatePolynomial(x,Integer))
--E 2

--S 3 of 20
Dx : LOD01 RFZ := D()
--R
--R
--R (3) D
--RType: LinearOrdinaryDifferentialOperator1(Fraction(UnivariatePolynomial(x,Integer)))
--E 3

--S 4 of 20
b : LOD01 RFZ := 3*x**2*Dx**2 + 2*Dx + 1/x
--R
--R
--R          2 2      1
--R (4) 3x D  + 2D + -
--R                      x
--RType: LinearOrdinaryDifferentialOperator1(Fraction(UnivariatePolynomial(x,Integer)))
--E 4

--S 5 of 20
a : LOD01 RFZ := b*(5*x*Dx + 7)
--R
--R
--R          3 3      2      2      7
--R (5) 15x D  + (51x  + 10x)D  + 29D + -
--R                      x
--RType: LinearOrdinaryDifferentialOperator1(Fraction(UnivariatePolynomial(x,Integer)))
--E 5

--S 6 of 20
p := x**2 + 1/x**2
--R
--R
--R          4
--R      x  + 1
--R (6) -----
--R          2
--R         x

```

```
--R                                         Type: Fraction(UnivariatePolynomial(x,Integer))
--E 6
```

```
--S 7 of 20
(a*b - b*a) p
```

```
--R
--R
--R          4
--R      - 75x  + 540x - 75
--R (7)  -----
--R          4
--R         x
```

```
--R                                         Type: Fraction(UnivariatePolynomial(x,Integer))
--E 7
```

```
--S 8 of 20
ld := leftDivide(a,b)
```

```
--R
--R
--R (8) [quotient= 5x D + 7,remainder= 0]
--RType: Record(quotient: LinearOrdinaryDifferentialOperator1(Fraction(...
--E 8
```

```
--S 9 of 20
a = b * ld.quotient + ld.remainder
```

```
--R
--R
--R          3 3      2      2      7      3 3      2      2      7
--R (9)  15x D  + (51x  + 10x)D  + 29D + -- 15x D  + (51x  + 10x)D  + 29D + -
--R                                         x                                         x
--RType: Equation(LinearOrdinaryDifferentialOperator1(Fraction(UnivariatePolynomial(x,Integer)))
--E 9
```

```
--S 10 of 20
rd := rightDivide(a,b)
```

```
--R
--R
--R          5
--R (10) [quotient= 5x D + 7,remainder= 10D + -]
--R                                         x
--RType: Record(quotient: LinearOrdinaryDifferentialOperator1(...)
--E 10
```

```
--S 11 of 20
a = rd.quotient * b + rd.remainder
```

```
--R
--R
--R          3 3      2      2      7      3 3      2      2      7
--R (11) 15x D  + (51x  + 10x)D  + 29D + -- 15x D  + (51x  + 10x)D  + 29D + -
--R                                         x                                         x
```

```
--RType: Equation(LinearOrdinaryDifferentialOperator1(Fraction(UnivariatePolynomial(x,Integer))))
--E 11
```

```
--S 12 of 20
rightQuotient(a,b)
--R
--R
--R (12)  $5x D + 7$ 
--RType: LinearOrdinaryDifferentialOperator1(Fraction(UnivariatePolynomial(x,Integer)))
--E 12
```

```
--S 13 of 20
rightRemainder(a,b)
--R
--R
--R (13)  $10D + \frac{5}{x}$ 
--RType: LinearOrdinaryDifferentialOperator1(Fraction(UnivariatePolynomial(x,Integer)))
--E 13
```

```
--S 14 of 20
leftExactQuotient(a,b)
--R
--R
--R (14)  $5x D + 7$ 
--RType: Union(LinearOrdinaryDifferentialOperator1(Fraction(UnivariatePolynomial(x,Integer))),...)
--E 14
```

```
--S 15 of 20
e := leftGcd(a,b)
--R
--R
--R (15)  $3x^2 D^2 + 2D + \frac{1}{x}$ 
--RType: LinearOrdinaryDifferentialOperator1(Fraction(UnivariatePolynomial(x,Integer)))
--E 15
```

```
--S 16 of 20
leftRemainder(a, e)
--R
--R
--R (16) 0
--RType: LinearOrdinaryDifferentialOperator1(Fraction(UnivariatePolynomial(x,Integer)))
--E 16
```

```
--S 17 of 20
rightRemainder(a, e)
--R
```

```

--R
--R      5
--R (17) 10D + -
--R      x
--RType: LinearOrdinaryDifferentialOperator1(Fraction(UnivariatePolynomial(x,Integer)))
--E 17

--S 18 of 20
f := rightLcm(a,b)
--R
--R
--R      3 3      2      2      7
--R (18) 15x D + (51x + 10x)D + 29D + -
--R      x
--RType: LinearOrdinaryDifferentialOperator1(Fraction(UnivariatePolynomial(x,Integer)))
--E 18

--S 19 of 20
rightRemainder(f, b)
--R
--R
--R      5
--R (19) 10D + -
--R      x
--RType: LinearOrdinaryDifferentialOperator1(Fraction(UnivariatePolynomial(x,Integer)))
--E 19

--S 20 of 20
leftRemainder(f, b)
--R
--R
--R (20) 0
--RType: LinearOrdinaryDifferentialOperator1(Fraction(UnivariatePolynomial(x,Integer)))
--E 20
)spool
)lisp (bye)

```

— LinearOrdinaryDifferentialOperator1.help —

```

=====
LinearOrdinaryDifferentialOperator1 example
=====

```

LinearOrdinaryDifferentialOperator1(A) is the domain of linear ordinary differential operators with coefficients in the differential ring A.

```
=====
Differential Operators with Rational Function Coefficients
=====
```

This example shows differential operators with rational function coefficients. In this case operator multiplication is non-commutative and, since the coefficients form a field, an operator division algorithm exists.

We begin by defining RFZ to be the rational functions in x with integer coefficients and Dx to be the differential operator for d/dx.

```
RFZ := Fraction UnivariatePolynomial('x, Integer)
      Fraction UnivariatePolynomial(x,Integer)
      Type: Domain

x : RFZ := 'x
  x
                                     Type: Fraction UnivariatePolynomial(x,Integer)

Dx : LODO1 RFZ := D()
  D
                                     Type: LinearOrdinaryDifferentialOperator1
                                     Fraction UnivariatePolynomial(x,Integer)
```

Operators are created using the usual arithmetic operations.

```
b : LODO1 RFZ := 3*x**2*Dx**2 + 2*Dx + 1/x
      2 2      1
      3x D  + 2D + -
                        x
                                     Type: LinearOrdinaryDifferentialOperator1
                                     Fraction UnivariatePolynomial(x,Integer)

a : LODO1 RFZ := b*(5*x*Dx + 7)
      3 3      2      2      7
      15x D  + (51x  + 10x)D  + 29D + -
                        x
                                     Type: LinearOrdinaryDifferentialOperator1
                                     Fraction UnivariatePolynomial(x,Integer)
```

Operator multiplication corresponds to functional composition.

```
p := x**2 + 1/x**2
      4
      x  + 1
      ----
      2
      x
                                     Type: Fraction UnivariatePolynomial(x,Integer)
```

Since operator coefficients depend on x , the multiplication is not commutative.

$$\frac{(a*b - b*a) p}{x^4 - 75x^3 + 540x^2 - 75x}$$

Type: Fraction UnivariatePolynomial(x,Integer)

When the coefficients of operator polynomials come from a field, as in this case, it is possible to define operator division. Division on the left and division on the right yield different results when the multiplication is non-commutative.

The results of `leftDivide` and `rightDivide` are quotient-remainder pairs satisfying:

$$\begin{aligned} \text{leftDivide}(a,b) &= [q, r] \text{ such that } a = b*q + r \\ \text{rightDivide}(a,b) &= [q, r] \text{ such that } a = q*b + r \end{aligned}$$

In both cases, the degree of the remainder, r , is less than the degree of b .

```
ld := leftDivide(a,b)
[quotient= 5x D + 7,remainder= 0]
Type: Record(quotient: LinearOrdinaryDifferentialOperator1
             Fraction UnivariatePolynomial(x,Integer),
             remainder: LinearOrdinaryDifferentialOperator1
             Fraction UnivariatePolynomial(x,Integer))

a = b * ld.quotient + ld.remainder
3 3      2      2      7      3 3      2      2      7
15x D + (51x + 10x)D + 29D + -= 15x D + (51x + 10x)D + 29D + -
Type: Equation LinearOrdinaryDifferentialOperator1
Fraction UnivariatePolynomial(x,Integer)
```

The operations of left and right division are so-called because the quotient is obtained by dividing a on that side by b .

```
rd := rightDivide(a,b)
[quotient= 5x D + 7,remainder= 10D + -]
Type: Record(quotient: LinearOrdinaryDifferentialOperator1
             Fraction UnivariatePolynomial(x,Integer),
             remainder: LinearOrdinaryDifferentialOperator1
             Fraction UnivariatePolynomial(x,Integer))

a = rd.quotient * b + rd.remainder
3 3      2      2      7      3 3      2      2      7
```

```

15x D + (51x + 10x)D + 29D + -= 15x D + (51x + 10x)D + 29D + -
Type: Equation LinearOrdinaryDifferentialOperator1
      Fraction UnivariatePolynomial(x,Integer)

```

Operations `rightQuotient` and `rightRemainder` are available if only one of the quotient or remainder are of interest to you. This is the quotient from right division.

```

rightQuotient(a,b)
5x D + 7
Type: LinearOrdinaryDifferentialOperator1
      Fraction UnivariatePolynomial(x,Integer)

```

This is the remainder from right division. The corresponding "left" functions, `leftQuotient` and `leftRemainder` are also available.

```

rightRemainder(a,b)
5
10D + -
x
Type: LinearOrdinaryDifferentialOperator1
      Fraction UnivariatePolynomial(x,Integer)

```

For exact division, operations `leftExactQuotient` and `rightExactQuotient` are supplied. These return the quotient but only if the remainder is zero. The call `rightExactQuotient(a,b)` would yield an error.

```

leftExactQuotient(a,b)
5x D + 7
Type: Union(LinearOrdinaryDifferentialOperator1
      Fraction UnivariatePolynomial(x,Integer),...)

```

The division operations allow the computation of left and right greatest common divisors, `leftGcd` and `rightGcd` via remainder sequences, and consequently the computation of left and right least common multiples, `rightLcm` and `leftLcm`.

```

e := leftGcd(a,b)
2 2      1
3x D + 2D + -
x
Type: LinearOrdinaryDifferentialOperator1
      Fraction UnivariatePolynomial(x,Integer)

```

Note that a greatest common divisor doesn't necessarily divide `a` and `b` on both sides. Here the left greatest common divisor does not divide `a` on the right.

```

leftRemainder(a, e)
0

```



```

Type: LinearOrdinaryDifferentialOperator1
      Fraction UnivariatePolynomial(x,Integer)

```

```

rightRemainder(a, e)
      5
10D + -
      x
Type: LinearOrdinaryDifferentialOperator1
      Fraction UnivariatePolynomial(x,Integer)

```

Similarly, a least common multiple is not necessarily divisible from both sides.

```

f := rightLcm(a,b)
      3 3      2      2      7
15x D + (51x + 10x)D + 29D + -
Type: LinearOrdinaryDifferentialOperator1
      Fraction UnivariatePolynomial(x,Integer)

```

```

rightRemainder(f, b)
      5
10D + -
      x
Type: LinearOrdinaryDifferentialOperator1
      Fraction UnivariatePolynomial(x,Integer)

```

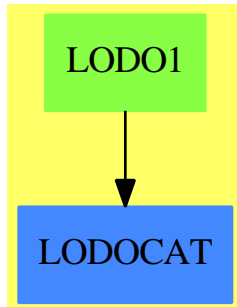
```

leftRemainder(f, b)
0
Type: LinearOrdinaryDifferentialOperator1
      Fraction UnivariatePolynomial(x,Integer)

```

See Also:

```
o )show LinearOrdinaryDifferentialOperator1
```

LinearOrdinaryDifferentialOperator1 (LODO1)

See

⇒ “LinearOrdinaryDifferentialOperator” (LODO) [13](#) on page [1469](#)

⇒ “LinearOrdinaryDifferentialOperator2” (LODO2) [13](#) on page [1491](#)

Exports:

0	1	adjoint	apply
characteristic	coefficient	coefficients	coerce
content	D	degree	directSum
exquo	hash	latex	leadingCoefficient
leftDivide	leftExactQuotient	leftExtendedGcd	leftGcd
leftLcm	leftQuotient	leftRemainder	minimumDegree
monicLeftDivide	monicRightDivide	monomial	one?
primitivePart	recip	reductum	retract
retractIfCan	rightDivide	rightExactQuotient	rightExtendedGcd
rightGcd	rightLcm	rightQuotient	rightRemainder
sample	subtractIfCan	symmetricPower	symmetricProduct
symmetricSquare	zero?	?*?	?**?
?+?	?-?	-?	?=?
?^?	?.??	?~=?	

— domain LODO1 LinearOrdinaryDifferentialOperator1 —

```
)abbrev domain LODO1 LinearOrdinaryDifferentialOperator1
++ Author: Manuel Bronstein
++ Date Created: 9 December 1993
++ Date Last Updated: 31 January 1994
++ Description:
++ \spad{LinearOrdinaryDifferentialOperator1} defines a ring of
++ differential operators with coefficients in a differential ring A.
++ Multiplication of operators corresponds to functional composition:\br
++ \spad{(L1 * L2).(f) = L1 L2 f}
```

```
LinearOrdinaryDifferentialOperator1(A:DifferentialRing) ==
  LinearOrdinaryDifferentialOperator(A, differentiate$A)
```

— LODO1.dotabb —

```
"LODO1" [color="#88FF44",href="bookvol10.3.pdf#nameddest=LODO1"]
"LODOCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LODOCAT"]
"LODO1" -> "LODOCAT"
```

domain LODO2 LinearOrdinaryDifferentialOperator2

— LinearOrdinaryDifferentialOperator2.input —

```
)set break resume
)sys rm -f LinearOrdinaryDifferentialOperator2.output
)spool LinearOrdinaryDifferentialOperator2.output
)set message test on
)set message auto off
)clear all
--S 1 of 26
Q := Fraction Integer
--R
--R
--R (1) Fraction(Integer)
--R
--R                                          Type: Domain
--E 1

--S 2 of 26
PQ := UnivariatePolynomial('x, Q)
--R
--R
--R (2) UnivariatePolynomial(x,Fraction(Integer))
--R
--R                                          Type: Domain
--E 2

--S 3 of 26
x: PQ := 'x
--R
--R
--R (3) x
--R
--R                                          Type: UnivariatePolynomial(x,Fraction(Integer))
--E 3

--S 4 of 26
```

Dx: LOD02(Q, PQ) := D()

--R

--R

--R (4) D

--RType: LinearOrdinaryDifferentialOperator2(Fraction(Integer),UnivariatePolynomial(x,Fraction(Integer)))

--E 4

--S 5 of 26

a := Dx + 1

--R

--R

--R (5) D + 1

--RType: LinearOrdinaryDifferentialOperator2(Fraction(Integer),UnivariatePolynomial(x,Fraction(Integer)))

--E 5

--S 6 of 26

b := a + 1/2*Dx**2 - 1/2

--R

--R

--R
$$(6) \quad -\frac{1}{2}D^2 + D + -\frac{1}{2}$$

--R

--RType: LinearOrdinaryDifferentialOperator2(Fraction(Integer),UnivariatePolynomial(x,Fraction(Integer)))

--E 6

--S 7 of 26

p := 4*x**2 + 2/3

--R

--R

--R
$$(7) \quad 4x^2 + -\frac{2}{3}$$

--R

--R Type: UnivariatePolynomial(x,Fraction(Integer))

--E 7

--S 8 of 26

a p

--R

--R

--R
$$(8) \quad 4x^2 + 8x + -\frac{2}{3}$$

--R

--R Type: UnivariatePolynomial(x,Fraction(Integer))

--E 8

--S 9 of 26

(a * b) p = a b p

--R

--R

--R
$$2 \quad 37 \quad 2 \quad 37$$

```

--R (9) 
$$\frac{2x^3 + 12x^2 + 12x + 3}{3} = \frac{2x^3 + 12x^2 + 12x + 3}{3}$$

--R
--R Type: Equation(UnivariatePolynomial(x,Fraction(Integer)))
--E 9

--S 10 of 26
c := (1/9)*b*(a + b)^2
--R
--R
--R (10) 
$$\frac{1}{72} D^6 + \frac{5}{36} D^5 + \frac{13}{24} D^4 + \frac{19}{18} D^3 + \frac{79}{72} D^2 + \frac{7}{12} D + \frac{1}{8}$$

--R
--RType: LinearOrdinaryDifferentialOperator2(Fraction(Integer),UnivariatePolynomial(x,Fraction(Integer)))
--E 10

--S 11 of 26
(a**2 - 3/4*b + c) (p + 1)
--R
--R
--R (11) 
$$3x^2 + \frac{44}{3}x + \frac{541}{36}$$

--R
--R Type: UnivariatePolynomial(x,Fraction(Integer))
--E 11

)clear all
--S 12 of 26
PZ := UnivariatePolynomial(x,Integer)
--R
--R
--R (1) UnivariatePolynomial(x,Integer)
--R
--R Type: Domain
--E 12

--S 13 of 26
x:PZ := 'x
--R
--R
--R (2) x
--R
--R Type: UnivariatePolynomial(x,Integer)
--E 13

--S 14 of 26
Mat := SquareMatrix(3,PZ)
--R
--R
--R (3) SquareMatrix(3,UnivariatePolynomial(x,Integer))
--R
--R Type: Domain
--E 14

--S 15 of 26

```

```

Vect := DPMM(3, PZ, Mat, PZ)
--R
--R
--R (4)
--R DirectProductMatrixModule(3,UnivariatePolynomial(x,Integer),SquareMatrix(3,Un
--R ivariatePolynomial(x,Integer)),UnivariatePolynomial(x,Integer))
--R
--R                                         Type: Domain
--E 15

--S 16 of 26
Modo := LOD02(Mat, Vect)
--R
--R
--R (5)
--R LinearOrdinaryDifferentialOperator2(SquareMatrix(3,UnivariatePolynomial(x,Int
--R eger)),DirectProductMatrixModule(3,UnivariatePolynomial(x,Integer),SquareMatr
--R ix(3,UnivariatePolynomial(x,Integer)),UnivariatePolynomial(x,Integer)))
--R
--R                                         Type: Domain
--E 16

--S 17 of 26
m:Mat := matrix [ [x^2,1,0],[1,x^4,0],[0,0,4*x^2] ]
--R
--R
--R
--R      + 2      +
--R      |x      1      0 |
--R      |          |
--R (6)  |          4      |
--R      |1      x      0 |
--R      |          |
--R      |          2|
--R      +0      0      4x +
--R
--R                                         Type: SquareMatrix(3,UnivariatePolynomial(x,Integer))
--E 17

--S 18 of 26
p:Vect := directProduct [3*x^2+1,2*x,7*x^3+2*x]
--R
--R
--R
--R      2      3
--R (7)  [3x  + 1,2x,7x  + 2x]
--R
--R                                         Type: DirectProductMatrixModule(3,UnivariatePolynomial(x,Integer),...
--E 18

--S 19 of 26
q: Vect := m * p
--R
--R
--R
--R      4      2      5      2      5      3
--R (8)  [3x  + x  + 2x,2x  + 3x  + 1,28x  + 8x ]

```

```
--IType: DirectProductMatrixModule(3,UnivariatePolynomial(x,Integer),...
--E 19
```

```
--S 20 of 26
Dx : Mod0 := D()
--R
--R
--R (9) D
--IType: LinearOrdinaryDifferentialOperator2(SquareMatrix(...)
--E 20
```

```
--S 21 of 26
a : Mod0 := Dx + m
--R
--R
--R (10) D + 
$$\begin{vmatrix} x^2 + 1 & 0 \\ 1 & x^2 + 0 \\ 0 & 0 & 4x + 2 \end{vmatrix}$$

--IType: LinearOrdinaryDifferentialOperator2(SquareMatrix(...)
--E 21
```

```
--S 22 of 26
b : Mod0 := m*Dx + 1
--R
--R
--R (11) 
$$\begin{vmatrix} x^2 + 1 & 0 & +1 & 0 & 0 \\ 1 & x^2 + 0 & |D + 0 & 1 & 0 \\ 0 & 0 & +0 & 0 & 1 \end{vmatrix} + 2 \begin{vmatrix} x^2 + 1 & 0 \\ 1 & x^2 + 0 \\ 0 & 0 & 4x + 2 \end{vmatrix}$$

--IType: LinearOrdinaryDifferentialOperator2(SquareMatrix(...)
--E 22
```

```
--S 23 of 26
c := a*b
--R
--R
--R (12) 
$$\begin{vmatrix} x^2 + 1 & 0 & |x^4 + 2x^2 + 2 & x^4 + x^2 & 0 \\ 1 & x^2 + 0 & |2 & 4 & 2 & 8 & 3 \\ 0 & 0 & |D + 4 & 2 & 8 & 3 \end{vmatrix} + 2 \begin{vmatrix} x^2 + 1 & 0 \\ 1 & x^2 + 0 \\ 0 & 0 & 4x + 2 \end{vmatrix}$$

--R
```

```
)lisp (bye)
```

— LinearOrdinaryDifferentialOperator2.help —

LinearOrdinaryDifferentialOperator2

`LinearOrdinaryDifferentialOperator2(A, M)` is the domain of linear ordinary differential operators with coefficients in the differential ring A and operating on M , an A -module. This includes the cases of operators which are polynomials in D acting upon scalar or vector expressions of a single variable. The coefficients of the operator polynomials can be integers, rational functions, matrices or elements of other domains.

```
=====
Differential Operators with Constant Coefficients
=====
```

This example shows differential operators with rational number coefficients operating on univariate polynomials.

We begin by making type assignments so we can conveniently refer to univariate polynomials in x over the rationals.

```
Q := Fraction Integer
Fraction Integer
                                Type: Domain

PQ := UnivariatePolynomial('x, Q)
UnivariatePolynomial(x,Fraction Integer)
                                Type: Domain

x: PQ := 'x
x
                                Type: UnivariatePolynomial(x,Fraction Integer)
```

Now we assign Dx to be the differential operator D corresponding to d/dx .

```
Dx: LOD02(Q, PQ) := D()
D
Type: LinearOrdinaryDifferentialOperator2(Fraction Integer,
UnivariatePolynomial(x,Fraction Integer))
```

New operators are created as polynomials in $D()$.

```
a := Dx + 1
D + 1
Type: LinearOrdinaryDifferentialOperator2(Fraction Integer,
UnivariatePolynomial(x,Fraction Integer))

b := a + 1/2*Dx**2 - 1/2
1 2      1
- D + D + -
2      2
Type: LinearOrdinaryDifferentialOperator2(Fraction Integer,
UnivariatePolynomial(x,Fraction Integer))
```

To apply the operator a to the value p the usual function call syntax is used.

```
p := 4*x**2 + 2/3
      2      2
      4x  + -
          3
Type: UnivariatePolynomial(x,Fraction Integer)
```

```
a p
      2      2
      4x  + 8x + -
          3
Type: UnivariatePolynomial(x,Fraction Integer)
```

Operator multiplication is defined by the identity $(a*b)p = a(b(p))$

```
(a * b) p = a b p
      2      37      2      37
      2x  + 12x + --- 2x  + 12x + ---
          3          3
Type: Equation UnivariatePolynomial(x,Fraction Integer)
```

Exponentiation follows from multiplication.

```
c := (1/9)*b*(a + b)^2
      1  6      5  5      13  4      19  3      79  2      7      1
      -- D  + -- D  + -- D  + -- D  + -- D  + -- D  + -
      72      36      24      18      72      12      8
Type: LinearOrdinaryDifferentialOperator2(Fraction Integer,
UnivariatePolynomial(x,Fraction Integer))
```

Finally, note that operator expressions may be applied directly.

```
(a**2 - 3/4*b + c) (p + 1)
      2      44      541
      3x  + -- x + ---
          3      36
Type: UnivariatePolynomial(x,Fraction Integer)
```

```
=====
Differential Operators with Matrix Coefficients Operating on Vectors}
=====
```

This is another example of linear ordinary differential operators with non-commutative multiplication. Unlike the rational function case, the differential ring of square matrices (of a given dimension) with univariate polynomial entries does not form a field. Thus the number of operations available is more limited.

In this section, the operators have three by three matrix coefficients with polynomial entries.

```
PZ := UnivariatePolynomial(x,Integer)
      UnivariatePolynomial(x,Integer)
      Type: Domain

x:PZ := 'x
      x
      Type: UnivariatePolynomial(x,Integer)

Mat := SquareMatrix(3,PZ)
      SquareMatrix(3,UnivariatePolynomial(x,Integer))
      Type: Domain
```

The operators act on the vectors considered as a Mat-module.

```
Vect := DPMM(3, PZ, Mat, PZ)
DirectProductMatrixModule(3,UnivariatePolynomial(x,Integer),SquareMatrix(3,UnivariatePolynomial(x,Integer)),UnivariatePolynomial(x,Integer))
      Type: Domain

Modo := LODD2(Mat, Vect)
LinearOrdinaryDifferentialOperator2(SquareMatrix(3,UnivariatePolynomial(x,Integer)),DirectProductMatrixModule(3,UnivariatePolynomial(x,Integer),SquareMatrix(3,UnivariatePolynomial(x,Integer)),UnivariatePolynomial(x,Integer)))
      Type: Domain
```

The matrix m is used as a coefficient and the vectors p and q are operated upon.

```
m:Mat := matrix [ [x^2,1,0],[1,x^4,0],[0,0,4*x^2] ]
      + 2      +
      |x  1  0 |
      |      |
      |  4  |
      |1  x  0 |
      |      |
      |      2|
      +0  0  4x +
      Type: SquareMatrix(3,UnivariatePolynomial(x,Integer))

p:Vect := directProduct [3*x^2+1,2*x,7*x^3+2*x]
      2      3
      [3x  + 1,2x,7x  + 2x]
      Type: DirectProductMatrixModule(3,UnivariatePolynomial(x,Integer),
      SquareMatrix(3,UnivariatePolynomial(x,Integer)),
      UnivariatePolynomial(x,Integer))
```

```

q: Vect := m * p
      4      2      5      2      5      3
      [3x  + x  + 2x, 2x  + 3x  + 1, 28x  + 8x ]
      Type: DirectProductMatrixModule(3,UnivariatePolynomial(x,Integer),
      SquareMatrix(3,UnivariatePolynomial(x,Integer)),
      UnivariatePolynomial(x,Integer))

```

Now form a few operators.

```

Dx : Mod0 := D()
D
      Type: LinearOrdinaryDifferentialOperator2(
      SquareMatrix(3,UnivariatePolynomial(x,Integer)),
      DirectProductMatrixModule(3,UnivariatePolynomial(x,Integer),
      SquareMatrix(3,UnivariatePolynomial(x,Integer)),
      UnivariatePolynomial(x,Integer)))

```

```

a : Mod0 := Dx + m
      + 2      +
      |x  1    0 |
      |      |
      D + |      4 |
      |1   x   0 |
      |      |
      |      2|
      +0  0  4x +
      Type: LinearOrdinaryDifferentialOperator2(
      SquareMatrix(3,UnivariatePolynomial(x,Integer)),
      DirectProductMatrixModule(3,UnivariatePolynomial(x,Integer),
      SquareMatrix(3,UnivariatePolynomial(x,Integer)),
      UnivariatePolynomial(x,Integer)))

```

```

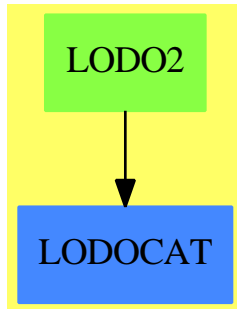
b : Mod0 := m*Dx + 1
      + 2      +
      |x  1    0 |      +1  0  0+
      |      |      |      | |
      |      4 |      |D + |0  1  0|
      |1   x   0 |      |      |
      |      |      +0  0  1+
      |      2|
      +0  0  4x +
      Type: LinearOrdinaryDifferentialOperator2(
      SquareMatrix(3,UnivariatePolynomial(x,Integer)),
      DirectProductMatrixModule(3,UnivariatePolynomial(x,Integer),
      SquareMatrix(3,UnivariatePolynomial(x,Integer)),
      UnivariatePolynomial(x,Integer)))

```

```

c := a*b
      + 2      +      + 4      4      2      +      + 2      +
      |x  1    0 |      |x  + 2x + 2      x  + x      0      |      |x  1    0 |

```


LinearOrdinaryDifferentialOperator2 (LODO2)

See

⇒ “LinearOrdinaryDifferentialOperator” (LODO) [13](#) on page [1469](#)

⇒ “LinearOrdinaryDifferentialOperator1” (LODO1) [13](#) on page [1479](#)

Exports:

0	1	adjoint	apply
characteristic	coefficient	coefficients	coerce
content	D	degree	directSum
exquo	hash	latex	leadingCoefficient
leftDivide	leftExactQuotient	leftExtendedGcd	leftGcd
leftLcm	leftQuotient	leftRemainder	minimumDegree
monicLeftDivide	monicRightDivide	monomial	one?
primitivePart	recip	reductum	retract
retractIfCan	rightDivide	rightExactQuotient	rightExtendedGcd
rightGcd	rightLcm	rightQuotient	rightRemainder
sample	subtractIfCan	symmetricPower	symmetricProduct
symmetricSquare	zero?	?*?	?**?
?+?	?-?	-?	?=?
?^?	?.??	?~=?	

— domain LODO2 LinearOrdinaryDifferentialOperator2 —

```

)abbrev domain LODO2 LinearOrdinaryDifferentialOperator2
++ Author: Stephen M. Watt, Manuel Bronstein
++ Date Created: 1986
++ Date Last Updated: 1 February 1994
++ Description:
++ \spad{LinearOrdinaryDifferentialOperator2} defines a ring of
++ differential operators with coefficients in a differential ring A
++ and acting on an A-module M.
++ Multiplication of operators corresponds to functional composition:\br
++ \spad{(L1 * L2).(f) = L1 L2 f}

LinearOrdinaryDifferentialOperator2(A, M): Exports == Implementation where
  A: DifferentialRing

```


--E 2

--S 3 of 34

list(1)

--R

--R

--R (3) [1]

--R

Type: List(PositiveInteger)

--E 3

--S 4 of 34

append([1,2,3],[5,6,7])

--R

--R

--R (4) [1,2,3,5,6,7]

--R

Type: List(PositiveInteger)

--E 4

--S 5 of 34

cons(10,[9,8,7])

--R

--R

--R (5) [10,9,8,7]

--R

Type: List(PositiveInteger)

--E 5

--S 6 of 34

empty? [x+1]

--R

--R

--R (6) false

--R

Type: Boolean

--E 6

--S 7 of 34

([] = nil)@Boolean

--R

--R

--R (7) true

--R

Type: Boolean

--E 7

--S 8 of 34

k := [4,3,7,3,8,5,9,2]

--R

--R

--R (8) [4,3,7,3,8,5,9,2]

--R

Type: List(PositiveInteger)

--E 8


```
--S 9 of 34
first k
--R
--R
--R (9)  4
--R
--E 9
```

Type: PositiveInteger

```
--S 10 of 34
k.first
--R
--R
--R (10)  4
--R
--E 10
```

Type: PositiveInteger

```
--S 11 of 34
k.1
--R
--R
--R (11)  4
--R
--E 11
```

Type: PositiveInteger

```
--S 12 of 34
k(1)
--R
--R
--R (12)  4
--R
--E 12
```

Type: PositiveInteger

```
--S 13 of 34
n := #k
--R
--R
--R (13)  8
--R
--E 13
```

Type: PositiveInteger

```
--S 14 of 34
last k
--R
--R
--R (14)  2
--R
--E 14
```

Type: PositiveInteger

```
--S 15 of 34
k.last
```

```

--R
--R
--R (15)  2
--R
--R                                         Type: PositiveInteger
--E 15

--S 16 of 34
k. (#k)
--R
--R
--R (16)  2
--R
--R                                         Type: PositiveInteger
--E 16

--S 17 of 34
k := [4,3,7,3,8,5,9,2]
--R
--R
--R (17)  [4,3,7,3,8,5,9,2]
--R
--R                                         Type: List(PositiveInteger)
--E 17

--S 18 of 34
k.1 := 999
--R
--R
--R (18)  999
--R
--R                                         Type: PositiveInteger
--E 18

--S 19 of 34
k
--R
--R
--R (19)  [999,3,7,3,8,5,9,2]
--R
--R                                         Type: List(PositiveInteger)
--E 19

--S 20 of 34
k := [1,2]
--R
--R
--R (20)  [1,2]
--R
--R                                         Type: List(PositiveInteger)
--E 20

--S 21 of 34
m := cons(0,k)
--R
--R

```

```
--R (21) [0,1,2]
```

```
--R
```

```
Type: List(Integer)
```

```
--E 21
```

```
--S 22 of 34
```

```
m.2 := 99
```

```
--R
```

```
--R
```

```
--R (22) 99
```

```
--R
```

```
Type: PositiveInteger
```

```
--E 22
```

```
--S 23 of 34
```

```
m
```

```
--R
```

```
--R
```

```
--R (23) [0,99,2]
```

```
--R
```

```
Type: List(Integer)
```

```
--E 23
```

```
--S 24 of 34
```

```
k
```

```
--R
```

```
--R
```

```
--R (24) [99,2]
```

```
--R
```

```
Type: List(PositiveInteger)
```

```
--E 24
```

```
--S 25 of 34
```

```
k := [1,2,3]
```

```
--R
```

```
--R
```

```
--R (25) [1,2,3]
```

```
--R
```

```
Type: List(PositiveInteger)
```

```
--E 25
```

```
--S 26 of 34
```

```
rest k
```

```
--R
```

```
--R
```

```
--R (26) [2,3]
```

```
--R
```

```
Type: List(PositiveInteger)
```

```
--E 26
```

```
--S 27 of 34
```

```
removeDuplicates [4,3,4,3,5,3,4]
```

```
--R
```

```
--R
```

```
--R (27) [4,3,5]
```

```
--R
```

```
Type: List(PositiveInteger)
```

--E 27

--S 28 of 34

reverse [1,2,3,4,5,6]

--R

--R

--R (28) [6,5,4,3,2,1]

--R

Type: List(PositiveInteger)

--E 28

--S 29 of 34

member?(1/2, [3/4,5/6,1/2])

--R

--R

--R (29) true

--R

Type: Boolean

--E 29

--S 30 of 34

member?(1/12, [3/4,5/6,1/2])

--R

--R

--R (30) false

--R

Type: Boolean

--E 30

--S 31 of 34

reverse(rest(reverse(k)))

--R

--R

--R (31) [1,2]

--R

Type: List(PositiveInteger)

--E 31

--S 32 of 34

[1..3,10,20..23]

--R

--R

--R (32) [1..3,10..10,20..23]

--R

Type: List(Segment(PositiveInteger))

--E 32

--S 33 of 34

expand [1..3,10,20..23]

--R

--R

--R (33) [1,2,3,10,20,21,22,23]

--R

Type: List(Integer)

--E 33

```

--S 34 of 34
expand [1..]
--R
--R
--R (34) [1,2,3,4,5,6,7,8,9,10,...]
--R
--R                                          Type: Stream(Integer)
--E 34
)spool
)lisp (bye)

```

— List.help —

```

=====
List examples
=====

```

A list is a finite collection of elements in a specified order that can contain duplicates. A list is a convenient structure to work with because it is easy to add or remove elements and the length need not be constant. There are many different kinds of lists in Axiom, but the default types (and those used most often) are created by the List constructor. For example, there are objects of type List Integer, List Float and List Polynomial Fraction Integer. Indeed, you can even have List List List Boolean (that is, lists of lists of lists of Boolean values). You can have lists of any type of Axiom object.

```

=====
Creating Lists
=====

```

The easiest way to create a list with, for example, the elements 2, 4, 5, 6 is to enclose the elements with square brackets and separate the elements with commas.

The spaces after the commas are optional, but they do improve the readability.

```

[2, 4, 5, 6]
[2,4,5,6]

```

Type: List PositiveInteger

To create a list with the single element 1, you can use either [1] or the operation list.

```

[1]
list 1

```

Type: List PositiveInteger

```
list(1)
[1]
```

Type: List PositiveInteger

Once created, two lists *k* and *m* can be concatenated by issuing `append(k,m)`. `append` does not physically join the lists, but rather produces a new list with the elements coming from the two arguments.

```
append([1,2,3],[5,6,7])
[1,2,3,5,6,7]
```

Type: List PositiveInteger

Use `cons` to append an element onto the front of a list.

```
cons(10,[9,8,7])
[10,9,8,7]
```

Type: List PositiveInteger

=====

Accessing List Elements

=====

To determine whether a list has any elements, use the operation `empty?`.

```
empty? [x+1]
false
```

Type: Boolean

Alternatively, equality with the list constant `nil` can be tested.

```
([] = nil)@Boolean
true
```

Type: Boolean

We'll use this in some of the following examples.

```
k := [4,3,7,3,8,5,9,2]
[4,3,7,3,8,5,9,2]
```

Type: List PositiveInteger

Each of the next four expressions extracts the first element of *k*.

```
first k
4
```

Type: PositiveInteger

```
k.first
4
```

Type: PositiveInteger

```
k.1
  4
                                Type: PositiveInteger
```

```
k(1)
  4
                                Type: PositiveInteger
```

The last two forms generalize to `k.i` and `k(i)`, respectively, where $1 \leq i \leq n$ and n equals the length of `k`.

This length is calculated by `#`.

```
n := #k
  8
                                Type: PositiveInteger
```

Performing an operation such as `k.i` is sometimes referred to as indexing into `k` or elting into `k`. The latter phrase comes about because the name of the operation that extracts elements is called `elt`. That is, `k.3` is just alternative syntax for `elt(k,3)`. It is important to remember that list indices begin with 1. If we issue `k := [1,3,2,9,5]` then `k.4` returns 9. It is an error to use an index that is not in the range from 1 to the length of the list.

The last element of a list is extracted by any of the following three expressions.

```
last k
  2
                                Type: PositiveInteger
```

```
k.last
  2
                                Type: PositiveInteger
```

This form computes the index of the last element and then extracts the element from the list.

```
k.(#k)
  2
                                Type: PositiveInteger
```

```
=====
Changing List Elements
=====
```

We'll use this in some of the following examples.

```
k := [4,3,7,3,8,5,9,2]
      [4,3,7,3,8,5,9,2]
```

Type: List PositiveInteger

List elements are reset by using the `k.i` form on the left-hand side of an assignment. This expression resets the first element of `k` to 999.

```
k.1 := 999
      999
```

Type: PositiveInteger

As with indexing into a list, it is an error to use an index that is not within the proper bounds. Here you see that `k` was modified.

```
k
      [999,3,7,3,8,5,9,2]
```

Type: List PositiveInteger

The operation that performs the assignment of an element to a particular position in a list is called `setelt`. This operation is destructive in that it changes the list. In the above example, the assignment returned the value 999 and `k` was modified. For this reason, lists are called mutable objects: it is possible to change part of a list (mutate it) rather than always returning a new list reflecting the intended modifications.

Moreover, since lists can share structure, changes to one list can sometimes affect others.

```
k := [1,2]
      [1,2]
```

Type: List PositiveInteger

```
m := cons(0,k)
      [0,1,2]
```

Type: List Integer

Change the second element of `m`.

```
m.2 := 99
      99
```

Type: PositiveInteger

See, `m` was altered.

```
m
      [0,99,2]
```

Type: List Integer

But what about `k`? It changed too!


```
k
[99,2]

Type: List PositiveInteger
```

Other Functions

An operation that is used frequently in list processing is that which returns all elements in a list after the first element.

```
k := [1,2,3]
[1,2,3]

Type: List PositiveInteger
```

Use the rest operation to do this.

```
rest k
[2,3]

Type: List PositiveInteger
```

To remove duplicate elements in a list k , use `removeDuplicates`.

```
removeDuplicates [4,3,4,3,5,3,4]
[4,3,5]

Type: List PositiveInteger
```

To get a list with elements in the order opposite to those in a list k , use `reverse`.

```
reverse [1,2,3,4,5,6]
[6,5,4,3,2,1]

Type: List PositiveInteger
```

To test whether an element is in a list, use `member?`: `member?(a,k)` returns true or false depending on whether a is in k or not.

```
member?(1/2, [3/4,5/6,1/2])
true

Type: Boolean
```

```
member?(1/12, [3/4,5/6,1/2])
false

Type: Boolean
```

We can get a list containing all but the last of the elements in a given non-empty list k .

```
reverse(rest(reverse(k)))
```

[1,2]

Type: List PositiveInteger

```
=====
Dot, Dot
=====
```

Certain lists are used so often that Axiom provides an easy way of constructing them. If n and m are integers, then `expand [n..m]` creates a list containing $n, n+1, \dots, m$. If $n > m$ then the list is empty. It is actually permissible to leave off the m in the dot-dot construction (see below).

The dot-dot notation can be used more than once in a list construction and with specific elements being given. Items separated by dots are called segments.

[1..3,10,20..23]

[1..3,10..10,20..23]

Type: List Segment PositiveInteger

Segments can be expanded into the range of items between the endpoints by using `expand`.

`expand [1..3,10,20..23]`

[1,2,3,10,20,21,22,23]

Type: List Integer

What happens if we leave off a number on the right-hand side of `..`?

`expand [1..]`

[1,2,3,4,5,6,7,8,9,10,...]

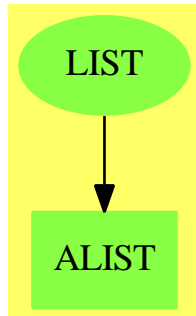
Type: Stream Integer

What is created in this case is a Stream which is a generalization of a list.

See Also:

o `)help Stream`

o `)show List`

List (LIST)**See**⇒ “IndexedList” (ILIST) [10](#) on page [1231](#)⇒ “AssociationList” (ALIST) [2](#) on page [235](#)**Exports:**

any?	append	child?	children	coerce
concat	concat!	cons	construct	convert
copy	copyInto!	count	cycleEntry	cycleLength
cycleSplit!	cycleTail	cyclic?	delete	delete!
distance	elt	empty	empty?	entries
entry?	eq?	eval	every?	explicitlyFinite?
fill!	find	first	hash	index?
indices	insert	insert!	last	latex
leaf?	leaves	less?	list	map
map!	max	maxIndex	member?	members
merge	merge!	min	minIndex	more?
node?	new	nil	nodes	null
OMwrite	parts	position	possiblyInfinite?	qelt
qsetelt!	reduce	remove	remove!	removeDuplicates
removeDuplicates!	rest	reverse	reverse!	sample
second	select	select!	setDifference	setIntersection
setUnion	setchildren!	setelt	setfirst!	setlast!
setrest!	setvalue!	size?	sort	sort!
sorted?	split!	swap!	tail	third
value	#?	?<?	?<=?	?=?
?>?	?>=?	?..?	?~=?	?..?
?.last	?.rest	?.first	?.value	

— domain LIST List —

```

)abbrev domain LIST List
++ Author: Michael Monagan
++ Date Created: Sep 1987
++ Description:

```

```

++ \spadtype{List} implements singly-linked lists that are
++ addressable by indices; the index of the first element
++ is 1. In addition to the operations provided by
++ \spadtype{IndexedList}, this constructor provides some
++ LISP-like functions such as \spadfun{null} and \spadfun{cons}.

```

```

List(S:Type): Exports == Implementation where
LISTMININDEX ==> 1      -- this is the minimum list index

```

```

Exports ==> ListAggregate S with
nil      : ()      -> %
  ++ nil() returns the empty list.
null     : %      -> Boolean
  ++ null(u) tests if list \spad{u} is the
  ++ empty list.
cons     : (S, %) -> %
  ++ cons(element,u) appends \spad{element} onto the front
  ++ of list \spad{u} and returns the new list. This new list
  ++ and the old one will share some structure.
append   : (%, %) -> %
  ++ append(u1,u2) appends the elements of list \spad{u1}
  ++ onto the front of list \spad{u2}. This new list
  ++ and \spad{u2} will share some structure.
if S has SetCategory then
  setUnion : (%, %) -> %
    ++ setUnion(u1,u2) appends the two lists u1 and u2, then
    ++ removes all duplicates. The order of elements in the
    ++ resulting list is unspecified.
  setIntersection : (%, %) -> %
    ++ setIntersection(u1,u2) returns a list of the elements
    ++ that lists \spad{u1} and \spad{u2} have in common.
    ++ The order of elements in the resulting list is unspecified.
  setDifference : (%, %) -> %
    ++ setDifference(u1,u2) returns a list of the elements
    ++ of \spad{u1} that are not also in \spad{u2}.
    ++ The order of elements in the resulting list is unspecified.
if S has OpenMath then OpenMath

```

```

Implementation ==>
  IndexedList(S, LISTMININDEX) add
    nil()          == NIL$Lisp
    null l         == NULL(1)$Lisp
    cons(s, l)     == CONS(s, 1)$Lisp
    append(l:%, t:%) == APPEND(l, t)$Lisp

  if S has OpenMath then
    writeOMList(dev: OpenMathDevice, x: %): Void ==
      OMputApp(dev)
      OMputSymbol(dev, "list1", "list")
      -- The following didn't compile because the compiler isn't

```

```

-- convinced that 'xval' is a S.  Duhhh! MCD.
--for xval in x repeat
--  OMwrite(dev, xval, false)
while not null x repeat
  OMwrite(dev,first x,false)
  x := rest x
OMputEndApp(dev)

OMwrite(x: %): String ==
s: String := ""
sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
dev: OpenMathDevice := OMopenString(sp @ String, OMencodingXML)
OMputObject(dev)
writeOMList(dev, x)
OMputEndObject(dev)
OMclose(dev)
s := OM_-STRINGPTRTOSTRING(sp)$Lisp @ String
s

OMwrite(x: %, wholeObj: Boolean): String ==
s: String := ""
sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
dev: OpenMathDevice := OMopenString(sp @ String, OMencodingXML)
if wholeObj then
  OMputObject(dev)
  writeOMList(dev, x)
if wholeObj then
  OMputEndObject(dev)
OMclose(dev)
s := OM_-STRINGPTRTOSTRING(sp)$Lisp @ String
s

OMwrite(dev: OpenMathDevice, x: %): Void ==
  OMputObject(dev)
  writeOMList(dev, x)
  OMputEndObject(dev)

OMwrite(dev: OpenMathDevice, x: %, wholeObj: Boolean): Void ==
  if wholeObj then
    OMputObject(dev)
    writeOMList(dev, x)
  if wholeObj then
    OMputEndObject(dev)

if S has SetCategory then
  setUnion(l1:%,l2:%) == removeDuplicatess concat(l1,l2)

  setIntersection(l1:%,l2:%) ==
    u :% := empty()
    l1 := removeDuplicatess l1

```

```

while not empty? l1 repeat
  if member?(first l1,l2) then u := cons(first l1,u)
  l1 := rest l1
u

setDifference(l1:%,l2:%) ==
l1 := removeDuplicates l1
lu:% := empty()
while not empty? l1 repeat
  l11:=l1.1
  if not member?(l11,l2) then lu := concat(l11,lu)
  l1 := rest l1
lu

if S has ConvertibleTo InputForm then
convert(x:%):InputForm ==
  convert concat(convert("construct"::Symbol)@InputForm,
    [convert a for a in (x pretend List S)]$List(InputForm))

```

— LIST.dotabb —

```

"LIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=LIST",
  shape=ellipse]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"LIST" -> "ALIST"

```

domain LMOPS ListMonoidOps

— ListMonoidOps.input —

```

)set break resume
)sys rm -f ListMonoidOps.output
)spool ListMonoidOps.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ListMonoidOps
--R

```

```

--R ListMonoidOps(S: SetCategory,E: AbelianMonoid,un: E) is a domain constructor
--R Abbreviation for ListMonoidOps is LMOPS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for LMOPS
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : S -> %
--R coerce : % -> OutputForm          hash : % -> SingleInteger
--R latex : % -> String                leftMult : (S,%) -> %
--R makeTerm : (S,E) -> %              makeUnit : () -> %
--R mapExpon : ((E -> E),%) -> %        mapGen : ((S -> S),%) -> %
--R nthExpon : (%,Integer) -> E         nthFactor : (%,Integer) -> S
--R plus : (%,% ) -> %                plus : (S,E,%) -> %
--R retract : % -> S                  reverse : % -> %
--R reverse! : % -> %                 rightMult : (%,S) -> %
--R size : % -> NonNegativeInteger      ?~=? : (%,% ) -> Boolean
--R commutativeEquality : (%,% ) -> Boolean
--R listOfMonoms : % -> List(Record(gen: S,exp: E))
--R makeMulti : List(Record(gen: S,exp: E)) -> %
--R outputForm : (%,(OutputForm,OutputForm) -> OutputForm),((OutputForm,OutputForm) -> OutputForm)
--R retractIfCan : % -> Union(S,"failed")
--R
--E 1

)spool
)lisp (bye)

```

— ListMonoidOps.help —

```

=====
ListMonoidOps examples
=====

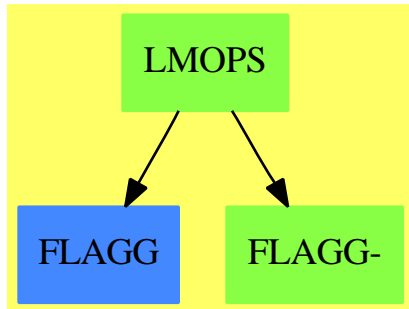
```

This internal package represents monoid (abelian or not, with or without inverses) as lists and provides some common operations to the various flavors of monoids.

See Also:

o)show ListMonoidOps

ListMonoidOps (LMOPS)



See

- ⇒ “FreeMonoid” (FMONOID) [7](#) on page [1021](#)
- ⇒ “FreeGroup” (FGROUP) [7](#) on page [1010](#)
- ⇒ “InnerFreeAbelianMonoid” (IFAMON) [10](#) on page [1286](#)
- ⇒ “FreeAbelianMonoid” (FAMONOID) [7](#) on page [1007](#)
- ⇒ “FreeAbelianGroup” (FAGROUP) [7](#) on page [1004](#)

Exports:

coerce	commutativeEquality	hash	latex	leftMult
listOfMonoms	makeTerm	makeUnit	mapExpon	mapGen
makeMulti	nthExpon	nthFactor	outputForm	plus
retract	retractIfCan	reverse	reverse!	rightMult
size	?=?	?~=?		

— domain LMOPS ListMonoidOps —

```

)abbrev domain LMOPS ListMonoidOps
++ Author: Manuel Bronstein
++ Date Created: November 1989
++ Date Last Updated: 6 June 1991
++ Description:
++ This internal package represents monoid (abelian or not, with or
++ without inverses) as lists and provides some common operations
++ to the various flavors of monoids.

```

```

ListMonoidOps(S, E, un): Exports == Implementation where
  S : SetCategory
  E : AbelianMonoid
  un: E

```

```

REC ==> Record(gen:S, exp: E)
0    ==> OutputForm

```

```

Exports ==> Join(SetCategory, RetractableTo S) with
  outputForm : ($, (0, 0) -> 0, (0, 0) -> 0, Integer) -> 0

```



```

++ outputForm(l, fop, fexp, unit) converts the monoid element
++ represented by l to an \spadtype{OutputForm}.
++ Argument unit is the output form
++ for the \spadignore{unit} of the monoid (e.g. 0 or 1),
++ \spad{fop(a, b)} is the
++ output form for the monoid operation applied to \spad{a} and b
++ (e.g. \spad{a + b}, \spad{a * b}, \spad{ab}),
++ and \spad{fexp(a, n)} is the output form
++ for the exponentiation operation applied to \spad{a} and n
++ (e.g. \spad{n a}, \spad{n * a}, \spad{a ** n}, \spad{a^~n}).
listOfMonoms : $ -> List REC
++ listOfMonoms(l) returns the list of the monomials forming l.
makeTerm      : (S, E) -> $
++ makeTerm(s, e) returns the monomial s exponentiated by e
++ (e.g. s^e or e * s).
makeMulti     : List REC -> $
++ makeMulti(l) returns the element whose list of monomials is l.
nthExpon      : ($, Integer) -> E
++ nthExpon(l, n) returns the exponent of the n^th monomial of l.
nthFactor     : ($, Integer) -> S
++ nthFactor(l, n) returns the factor of the n^th monomial of l.
reverse       : $ -> $
++ reverse(l) reverses the list of monomials forming l. This
++ has some effect if the monoid is non-abelian, i.e.
++ \spad{reverse(a1^~e1 ... an^~en) = an^~en ... a1^~e1} which is different.
reverse_!     : $ -> $
++ reverse!(l) reverses the list of monomials forming l, destroying
++ the element l.
size          : $ -> NonNegativeInteger
++ size(l) returns the number of monomials forming l.
makeUnit      : () -> $
++ makeUnit() returns the unit element of the monomial.
rightMult     : ($, S) -> $
++ rightMult(a, s) returns \spad{a * s} where \spad{*}
++ is the monoid operation,
++ which is assumed non-commutative.
leftMult      : (S, $) -> $
++ leftMult(s, a) returns \spad{s * a} where
++ \spad{*} is the monoid operation,
++ which is assumed non-commutative.
plus          : (S, E, $) -> $
++ plus(s, e, x) returns \spad{e * s + x} where \spad{+}
++ is the monoid operation,
++ which is assumed commutative.
plus          : ($, $) -> $
++ plus(x, y) returns \spad{x + y} where \spad{+}
++ is the monoid operation,
++ which is assumed commutative.
commutativeEquality: ($, $) -> Boolean
++ commutativeEquality(x,y) returns true if x and y are equal

```

```

    ++ assuming commutativity
mapExpon      : (E -> E, $) -> $
    ++ mapExpon(f, a1\^e1 ... an\^en) returns \spad{a1\^f(e1) ... an\^f(en)}.
mapGen        : (S -> S, $) -> $
    ++ mapGen(f, a1\^e1 ... an\^en) returns \spad{f(a1)\^e1 ... f(an)\^en}.

Implementation ==> add
Rep := List REC

localplus: ($, $) -> $

makeUnit()      == empty()$Rep
size l           == # listOfMonoms l
coerce(s:S):$    == [[s, un]]
coerce(l:$):0    == coerce(l)$Rep
makeTerm(s, e)   == (zero? e => makeUnit(); [[s, e]])
makeMulti l      == l
f = g            == f =$Rep g
listOfMonoms l   == l pretend List(REC)
nthExpon(f, i)   == f.(i-1+minIndex f).exp
nthFactor(f, i)  == f.(i-1+minIndex f).gen
reverse l        == reverse(l)$Rep
reverse_! l      == reverse_!(l)$Rep
mapGen(f, l)     == [[f(x.gen), x.exp] for x in l]

mapExpon(f, l) ==
  ans:List(REC) := empty()
  for x in l repeat
    if (a := f(x.exp)) ^= 0 then ans := concat([x.gen, a], ans)
  reverse_! ans

outputForm(l, op, opexp, id) ==
  empty? l => id::OutputForm
  l:List(0) :=
    [(p.exp = un => p.gen::0; opexp(p.gen::0, p.exp::0)) for p in l]
  reduce(op, l)

retractIfCan(l:$):Union(S, "failed") ==
  not empty? l and empty? rest l and l.first.exp = un => l.first.gen
  "failed"

rightMult(f, s) ==
  empty? f => s::$
  s = f.last.gen => (setlast_!(h := copy f, [s, f.last.exp + un]); h)
  concat(f, [s, un])

leftMult(s, f) ==
  empty? f => s::$
  s = f.first.gen => concat([s, f.first.exp + un], rest f)
  concat([s, un], f)

```

```

commutativeEquality(s1:$, s2:$):Boolean ==
  #s1 ^= #s2 => false
  for t1 in s1 repeat
    if not member?(t1,s2) then return false
  true

plus_!(s:S, n:E, f:$):$ ==
  h := g := concat([s, n], f)
  h1 := rest h
  while not empty? h1 repeat
    s = h1.first.gen =>
      l :=
        zero?(m := n + h1.first.exp) => rest h1
        concat([s, m], rest h1)
      setrest_!(h, l)
      return rest g
    h := h1
    h1 := rest h1
  g

plus(s, n, f) == plus_!(s,n,copy f)

plus(f, g) ==
  #f < #g => localplus(f, g)
  localplus(g, f)

localplus(f, g) ==
  g := copy g
  for x in f repeat
    g := plus(x.gen, x.exp, g)
  g

```

— LMOPS.dotabb —

```

"LMOPS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=LMOPS"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"LMOPS" -> "FLAGG"
"LMOPS" -> "FLAGG-"

```

domain LMDICT ListMultiDictionary

— ListMultiDictionary.input —

```

)set break resume
)sys rm -f ListMultiDictionary.output
)spool ListMultiDictionary.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ListMultiDictionary
--R
--R ListMultiDictionary(S: SetCategory) is a domain constructor
--R Abbreviation for ListMultiDictionary is LMDICT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for LMDICT
--R
--R----- Operations -----
--R bag : List(S) -> %                construct : List(S) -> %
--R copy : % -> %                    dictionary : List(S) -> %
--R dictionary : () -> %              duplicates? : % -> Boolean
--R empty : () -> %                  empty? : % -> Boolean
--R eq? : (%,% ) -> Boolean          extract! : % -> S
--R insert! : (S,% ) -> %            inspect : % -> S
--R map : ((S -> S),%) -> %          removeDuplicates! : % -> %
--R sample : () -> %                 substitute : (S,S,% ) -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?=? : (%,% ) -> Boolean if S has SETCAT
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if S has SETCAT
--R convert : % -> InputForm if S has KONVERT(INFORM)
--R count : (S,% ) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R duplicates : % -> List(Record(entry: S,count: NonNegativeInteger))
--R eval : (% ,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (% ,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (% ,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (% ,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R find : ((S -> Boolean),%) -> Union(S,"failed")
--R hash : % -> SingleInteger if S has SETCAT
--R insert! : (S,% ,NonNegativeInteger) -> %
--R latex : % -> String if S has SETCAT
--R less? : (% ,NonNegativeInteger) -> Boolean
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R member? : (S,% ) -> Boolean if $ has finiteAggregate and S has SETCAT

```

```

--R members : % -> List(S) if $ has finiteAggregate
--R more? : (% , NonNegativeInteger) -> Boolean
--R parts : % -> List(S) if $ has finiteAggregate
--R reduce : ((S,S) -> S) , % -> S if $ has finiteAggregate
--R reduce : ((S,S) -> S) , % , S -> S if $ has finiteAggregate
--R reduce : ((S,S) -> S) , % , S , S -> S if $ has finiteAggregate and S has SETCAT
--R remove : ((S -> Boolean) , %) -> % if $ has finiteAggregate
--R remove : (S , %) -> % if $ has finiteAggregate and S has SETCAT
--R remove! : ((S -> Boolean) , %) -> % if $ has finiteAggregate
--R remove! : (S , %) -> % if $ has finiteAggregate
--R removeDuplicates : % -> % if $ has finiteAggregate and S has SETCAT
--R select : ((S -> Boolean) , %) -> % if $ has finiteAggregate
--R select! : ((S -> Boolean) , %) -> % if $ has finiteAggregate
--R size? : (% , NonNegativeInteger) -> Boolean
--R ?~=? : (% , %) -> Boolean if S has SETCAT
--R
--E 1

)spool
)lisp (bye)

```

— ListMultiDictionary.help —

```

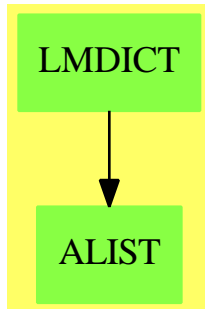
=====
ListMultiDictionary examples
=====

```

The ListMultiDictionary domain implements a dictionary with duplicates allowed. The representation is a list with duplicates represented explicitly. Hence most operations will be relatively inefficient when the number of entries in the dictionary becomes large. If the objects in the dictionary belong to an ordered set, the entries are maintained in ascending order.

See Also:
o)show ListMultiDictionary

ListMultiDictionary (LMDICT)

**Exports:**

any?	bag	coerce	construct	convert
copy	count	dictionary	dictionary	duplicates
duplicates?	empty	empty?	eq?	eval
extract!	every?	find	hash	insert!
inspect	latex	less?	map	map!
member?	members	more?	parts	reduce
remove	remove!	removeDuplicates	removeDuplicates!	sample
select	select!	size?	substitute	#?
?~=?	?=?			

— domain LMDICT ListMultiDictionary —

```

)abbrev domain LMDICT ListMultiDictionary
++ Author: Mark Botch
++ Date Last Updated: 13 June 1994 Frederic Lehobey
++ Description:
++ The \spadtype{ListMultiDictionary} domain implements a
++ dictionary with duplicates
++ allowed. The representation is a list with duplicates represented
++ explicitly. Hence most operations will be relatively inefficient
++ when the number of entries in the dictionary becomes large.
++ If the objects in the dictionary belong to an ordered set,
++ the entries are maintained in ascending order.

```

```
ListMultiDictionary(S:SetCategory): EE == II where
```

```

NNI ==> NonNegativeInteger
D ==> Record(entry:S, count:NonNegativeInteger)

EE ==> MultiDictionary(S) with
  finiteAggregate
  duplicates?: % -> Boolean
  ++ duplicates?(d) tests if dictionary d has duplicate entries.
  substitute : (S, S, %) -> %

```

```

    ++ substitute(x,y,d) replace x's with y's in dictionary d.
II ==> add
    Rep := Reference List S

sub: (S, S, S) -> S

coerce(s:%):OutputForm ==
    prefix("dictionary":OutputForm, [x:OutputForm for x in parts s])

#s                == # parts s
copy s            == dictionary copy parts s
empty? s          == empty? parts s
bag l             == dictionary l
dictionary()      == dictionary empty()

empty():% == ref empty()

dictionary(ls:List S):% ==
    empty? ls => empty()
    lmd := empty()
    for x in ls repeat insert_!(x,lmd)
    lmd

if S has ConvertibleTo InputForm then
    convert(lmd:%):InputForm ==
        convert [convert("dictionary":Symbol)@InputForm,
            convert(parts lmd)@InputForm]

map(f, s)          == dictionary map(f, parts s)
map_!(f, s)        == dictionary map_!(f, parts s)
parts s            == deref s
sub(x, y, z)       == (z = x => y; z)
insert_!(x, s, n)  == (for i in 1..n repeat insert_!(x, s); s)
substitute(x, y, s) == dictionary map(z1 +-> sub(x, y, z1), parts s)
removeDuplicates_! s == dictionary removeDuplicates_! parts s

inspect s ==
    empty? s => error "empty dictionary"
    first parts s

extract_! s ==
    empty? s => error "empty dictionary"
    x := first(p := parts s)
    setref(s, rest p)
    x

duplicates? s ==
    empty?(p := parts s) => false
    q := rest p
    while not empty? q repeat

```

```

    first p = first q => return true
    p := q
    q := rest q
    false

remove_!(p: S->Boolean, lmd:%):% ==
  for x in removeDuplicates parts lmd | p(x) repeat remove_!(x,lmd)
  lmd

select_!(p: S->Boolean, lmd:%):% == remove_!((z:S):Boolean+-->not p(z), lmd)

duplicates(lmd:%):List D ==
  ld: List D := empty()
  for x in removeDuplicates parts lmd | (n := count(x, lmd)) >
    1$NonNegativeInteger repeat
    ld := cons([x, n], ld)
  ld

if S has OrderedSet then
  s = t == parts s = parts t

  remove_!(x:S, s:%) ==
    p := deref s
    while not empty? p and x = first p repeat p := rest p
    setref(s, p)
    empty? p => s
    q := rest p
    while not empty? q and x > first q repeat (p := q; q := rest q)
    while not empty? q and x = first q repeat q := rest q
    p.rest := q
    s

  insert_!(x, s) ==
    p := deref s
    empty? p or x < first p =>
      setref(s, concat(x, p))
    s
    q := rest p
    while not empty? q and x > first q repeat (p := q; q := rest q)
    p.rest := concat(x, q)
    s

else
  remove_!(x:S, s:%) == (setref(s, remove_!(x, parts s))); s

  s = t ==
    a := copy s
    while not empty? a repeat
      x := inspect a
      count(x, s) ^= count(x, t) => return false

```



```

        remove_!(x, a)
    true

insert_!(x, s) ==
    p := deref s
    while not empty? p repeat
        x = first p =>
            p.rest := concat(x, rest p)
            s
        p := rest p
    setref(s, concat(x, deref s))
    s

```

— LMDICT.dotabb —

```

"LMDICT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=LMDICT"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"LMDICT" -> "ALIST"

```

domain LA LocalAlgebra

— LocalAlgebra.input —

```

)set break resume
)sys rm -f LocalAlgebra.output
)spool LocalAlgebra.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LocalAlgebra
--R
--R LocalAlgebra(A: Algebra(R),R: CommutativeRing,S: SubsetCategory(Monoid,R)) is a domain cons
--R Abbreviation for LocalAlgebra is LA
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for LA
--R
--R----- Operations -----
--R ?? : (R,%) -> %
--R ?? : (% ,R) -> %

```

```

--R ?? : (%,% ) -> %
--R ?? : (PositiveInteger,% ) -> %
--R +? : (%,% ) -> %
--R -? : % -> %
--R ?/? : (% ,S) -> %
--R 1 : () -> %
--R ^? : (% ,PositiveInteger) -> %
--R coerce : Integer -> %
--R denom : % -> S
--R latex : % -> String
--R one? : % -> Boolean
--R sample : () -> %
--R ^=? : (%,% ) -> Boolean
--R ?? : (NonNegativeInteger,% ) -> %
--R ***? : (% ,NonNegativeInteger) -> %
--R ?<? : (%,% ) -> Boolean if A has ORDRING
--R ?<=? : (%,% ) -> Boolean if A has ORDRING
--R ?>? : (%,% ) -> Boolean if A has ORDRING
--R ?>=? : (%,% ) -> Boolean if A has ORDRING
--R ^? : (% ,NonNegativeInteger) -> %
--R abs : % -> % if A has ORDRING
--R characteristic : () -> NonNegativeInteger
--R max : (%,% ) -> % if A has ORDRING
--R min : (%,% ) -> % if A has ORDRING
--R negative? : % -> Boolean if A has ORDRING
--R positive? : % -> Boolean if A has ORDRING
--R sign : % -> Integer if A has ORDRING
--R subtractIfCan : (%,% ) -> Union(%,"failed")
--R
--E 1

)spool
)lisp (bye)

```

— LocalAlgebra.help —

=====

LocalAlgebra examples

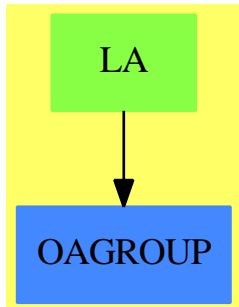
=====

LocalAlgebra produces the localization of an algebra, i.e.
fractions whose numerators come from some R algebra.

See Also:

o)show LocalAlgebra

LocalAlgebra (LA)



See

⇒ “Localize” (LO) [13](#) on page [1522](#)

⇒ “Fraction” (FRAC) [7](#) on page [985](#)

Exports:

0	1	abs	characteristic	coerce
denom	hash	latex	max	min
negative?	numer	one?	positive?	recip
sample	sign	subtractIfCan	zero?	?~=?
?*?	?**?	?<?	?<=?	?>?
?>=?	?^?	?+?	?-?	-?
?/?	?=?			

— domain LA LocalAlgebra —

```
)abbrev domain LA LocalAlgebra
++ Author: Dave Barton, Barry Trager
++ Description:
++ LocalAlgebra produces the localization of an algebra, i.e.
++ fractions whose numerators come from some R algebra.

LocalAlgebra(A: Algebra R,
             R: CommutativeRing,
             S: SubsetCategory(Monoid, R)): Algebra R with
if A has OrderedRing then OrderedRing
_ / : (% , S) -> %
++ x / d divides the element x by d.
_ / : (A, S) -> %
++ a / d divides the element \spad{a} by d.
numer: % -> A
++ numer x returns the numerator of x.
denom: % -> S
++ denom x returns the denominator of x.
== Localize(A, R, S) add
1 == 1$A / 1$S
x:% * y:% == (numer(x) * numer(y)) / (denom(x) * denom(y))
```

```
characteristic() == characteristic()$A
```

— LA.dotabb —

```
"LA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=LA"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"LA" -> "OAGROUP"
```

domain LO Localize

— Localize.input —

```
)set break resume
)sys rm -f Localize.output
)spool Localize.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Localize
--R
--R Localize(M: Module(R),R: CommutativeRing,S: SubsetCategory(Monoid,R)) is a domain constructor
--R Abbreviation for Localize is LO
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for LO
--R
--R----- Operations -----
--R ?? : (%,R) -> %           ?? : (R,%) -> %
--R ?? : (Integer,%) -> %     ?? : (PositiveInteger,%) -> %
--R ?? : (%,%) -> %          ?-? : (%,%) -> %
--R -? : % -> %              ?/? : (M,S) -> %
--R ?/? : (%,S) -> %         ?=? : (%,%) -> Boolean
--R 0 : () -> %              coerce : % -> OutputForm
--R denom : % -> S           hash : % -> SingleInteger
--R latex : % -> String      numer : % -> M
--R sample : () -> %         zero? : % -> Boolean
--R ?~=? : (%,%) -> Boolean
--R ?? : (NonNegativeInteger,%) -> %
--R ?<? : (%,%) -> Boolean if M has OAGROUP
```

```

--R ?<=? : (%,% ) -> Boolean if M has OAGROUP
--R ?>? : (%,% ) -> Boolean if M has OAGROUP
--R ?>=? : (%,% ) -> Boolean if M has OAGROUP
--R max : (%,% ) -> % if M has OAGROUP
--R min : (%,% ) -> % if M has OAGROUP
--R subtractIfCan : (%,% ) -> Union(%, "failed")
--R
--E 1

```

```

)spool
)lisp (bye)

```

— Localize.help —

=====

Localize examples

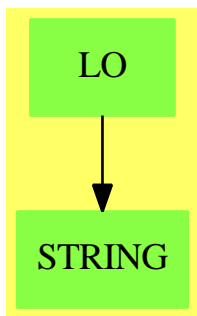
=====

Localize(M,R,S) produces fractions with numerators from an R module M and denominators from some multiplicative subset D of R.

See Also:

- o)show Localize

Localize (LO)



See

⇒ “LocalAlgebra” (LA) [13](#) on page [1520](#)

⇒ “Fraction” (FRAC) [7](#) on page [985](#)

Exports:

0	coerce	denom	hash	latex
max	min	numer	sample	subtractIfCan
zero?	?~=?	?*?	?<?	?<=?
?>?	?>=?	?+?	?-?	-?
?/?	?=?			

— domain LO Localize —

```

)abbrev domain LO Localize
++ Author: Dave Barton, Barry Trager
++ Description:
++ Localize(M,R,S) produces fractions with numerators
++ from an R module M and denominators from some multiplicative subset D of R.

Localize(M:Module R,
  R:CommutativeRing,
  S:SubsetCategory(Monoid, R)): Module R with
if M has OrderedAbelianGroup then OrderedAbelianGroup
_/: (% , S) -> %
  ++ x / d divides the element x by d.
_/: (M, S) -> %
  ++ m / d divides the element m by d.
numer: % -> M
  ++ numer x returns the numerator of x.
denom: % -> S
  ++ denom x returns the denominator of x.
==
add
--representation
Rep:= Record(num:M,den:S)
--declarations
x,y: %
n: Integer
m: M
r: R
d: S
--definitions
0 == [0,1]
zero? x == zero? (x.num)
-x== [-x.num,x.den]
x=y == y.den*x.num = x.den*y.num
numer x == x.num
denom x == x.den
if M has OrderedAbelianGroup then
  x < y ==
--      if y.den::R < 0 then (x,y):=(y,x)
--      if x.den::R < 0 then (x,y):=(y,x)
      y.den*x.num < x.den*y.num
x+y == [y.den*x.num+x.den*y.num, x.den*y.den]

```

```

n*x == [n*x.num,x.den]
r*x == if r=x.den then [x.num,1] else [r*x.num,x.den]
x/d ==
  zero?(u:S:=d*x.den) => error "division by zero"
  [x.num,u]
m/d == if zero? d then error "division by zero" else [m,d]
coerce(x:%):OutputForm ==
--   one?(xd:=x.den) => (x.num)::OutputForm
   ((xd:=x.den) = 1) => (x.num)::OutputForm
   (x.num)::OutputForm / (xd::OutputForm)
latex(x:%): String ==
--   one?(xd:=x.den) => latex(x.num)
   ((xd:=x.den) = 1) => latex(x.num)
nl : String := concat("{", concat(latex(x.num), "}")$String)$String
dl : String := concat("{", concat(latex(x.den), "}")$String)$String
concat("{ ", concat(nl, concat(" \over ", concat(dl, " }")$String)$String)$String)$String

```

— LO.dotabb —

```

"LO" [color="#88FF44",href="bookvol10.3.pdf#nameddest=LO"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"LO" -> "STRING"

```

domain LWORD LyndonWord

— LyndonWord.input —

```

)set break resume
)sys rm -f LyndonWord.output
)spool LyndonWord.output
)set message test on
)set message auto off
)clear all
--S 1 of 22
a:Symbol := 'a
--R
--R
--R   (1)  a
--R
--E 1

```

Type: Symbol

```

--S 2 of 22
b:Symbol := 'b
--R
--R
--R (2) b
--R
--R                                          Type: Symbol
--E 2

--S 3 of 22
c:Symbol := 'c
--R
--R
--R (3) c
--R
--R                                          Type: Symbol
--E 3

--S 4 of 22
lword:= LyndonWord(Symbol)
--R
--R
--R (4) LyndonWord(Symbol)
--R
--R                                          Type: Domain
--E 4

--S 5 of 22
magma := Magma(Symbol)
--R
--R
--R (5) Magma(Symbol)
--R
--R                                          Type: Domain
--E 5

--S 6 of 22
word := OrderedFreeMonoid(Symbol)
--R
--R
--R (6) OrderedFreeMonoid(Symbol)
--R
--R                                          Type: Domain
--E 6

--S 7 of 22
LyndonWordsList1([a,b,c],3)$lword
--R
--R
--R (7)
--R [[a], [b], [c]], [[a b], [a c], [b c]],
--R      2      2      2      2      2      2
--R [[a b], [a c], [a b ], [a b c], [a c b], [a c ], [b c], [b c ]]
--R                                          Type: OneDimensionalArray(List(LyndonWord(Symbol)))

```


--S 8 of 22

--R

--R

$$--\mathbb{R}$$

--R

 $-\mathbb{R}$

--E 8

--S 9 of 22

$$--R$$
$$--\mathbb{R}$$
$$--R$$

--R.

--R

--E 9

--S 10 of 22

--R

 $-\mathbb{R}$ $-\mathbb{R}$

--E 10

--S 11 of 22

 $-\mathbb{R}$

--R

--R

--R

--E 11

--S 12 of 22

--R

--R

--R

Type: List(LyndonWord(Symbol))

--E 12

--S 13 of 22

factor(w1*w2)\$lword

--R

--R

--R 2 2
--R (13) [[a b a b]]

--R

Type: List(LyndonWord(Symbol))

--E 13

--S 14 of 22

factor(w2*w2)\$lword

--R

--R

--R 2 2
--R (14) [[a b],[a b]]

--R

Type: List(LyndonWord(Symbol))

--E 14

--S 15 of 22

factor(w2*w1)\$lword

--R

--R

--R 2 2
--R (15) [[a b],[a b]]

--R

Type: List(LyndonWord(Symbol))

--E 15

--S 16 of 22

lyndon?(w1)\$lword

--R

--R

--R (16) true

--R

Type: Boolean

--E 16

--S 17 of 22

lyndon?(w1*w2)\$lword

--R

--R

--R (17) true

--R

Type: Boolean

--E 17

--S 18 of 22

lyndon?(w2*w1)\$lword

--R

--R

--R (18) false

```

--R                                                    Type: Boolean
--E 18

--S 19 of 22
lyndonIfCan(w1)$lword
--R
--R
--R          2
--R   (19)  [a b]
--R
--R                                                    Type: Union(LyndonWord(Symbol),...)
--E 19

--S 20 of 22
lyndonIfCan(w2*w1)$lword
--R
--R
--R   (20)  "failed"
--R
--R                                                    Type: Union("failed",...)
--E 20

--S 21 of 22
lyndon(w1)$lword
--R
--R
--R          2
--R   (21)  [a b]
--R
--R                                                    Type: LyndonWord(Symbol)
--E 21

--S 22 of 22
lyndon(w1*w2)$lword
--R
--R
--R          2      2
--R   (22)  [a b a b ]
--R
--R                                                    Type: LyndonWord(Symbol)
--E 22
)spool
)lisp (bye)

```

— LyndonWord.help —

=====

LyndonWord examples

=====

A function f in $[0,1]$ is called acyclic if $C(F)$ consists of n

different objects. The canonical representative of the orbit of an acyclic function is usually called a Lyndon Word. If f is acyclic, then all elements in the orbit $C(f)$ are acyclic as well, and we call $C(f)$ an acyclic orbit.

=====

Initialisations

=====

```
a:Symbol := 'a
a
Type: Symbol
```

```
b:Symbol := 'b
b
Type: Symbol
```

```
c:Symbol := 'c
c
Type: Symbol
```

```
lword:= LyndonWord(Symbol)
LyndonWord Symbol
Type: Domain
```

```
magma := Magma(Symbol)
Magma Symbol
Type: Domain
```

```
word := OrderedFreeMonoid(Symbol)
OrderedFreeMonoid Symbol
Type: Domain
```

All Lyndon words of with a, b, c to order 3

```
LyndonWordsList1([a,b,c],3)$lword
[[[a],[b],[c]], [[a b],[a c],[b c]],
  2      2      2      2      2      2
 [[a b],[a c],[a b ],[a b c],[a c b],[a c ],[b c],[b c ]]]
Type: OneDimensionalArray List LyndonWord Symbol
```

All Lyndon words of with a, b, c to order 3 in flat list

```
LyndonWordsList([a,b,c],3)$lword
2      2      2
[[a], [b], [c], [a b], [a c], [b c], [a b ], [a c ], [a b c], [a c b],
  2      2      2
 [a c ], [b c ], [b c ]]
Type: List LyndonWord Symbol
```

All Lyndon words of with a, b to order 5

```
lw := LyndonWordsList([a,b],5)$lword
      2      2      3      2 2      3      4      3 2
[[a], [b], [a b], [a b], [a b ], [a b], [a b ], [a b ], [a b ], [a b ],
      2      2 3      2      4
[a b a b], [a b ], [a b a b ], [a b ]]
```

Type: List LyndonWord Symbol

```
w1 : word := lw.4 :: word
      2
a b
```

Type: OrderedFreeMonoid Symbol

```
w2 : word := lw.5 :: word
      2
a b
```

Type: OrderedFreeMonoid Symbol

Let's try factoring

```
factor(a::word)$lword
[[a]]
```

Type: List LyndonWord Symbol

```
factor(w1*w2)$lword
      2      2
[[a b a b ]]
```

Type: List LyndonWord Symbol

```
factor(w2*w2)$lword
      2      2
[[a b ],[a b ]]
```

Type: List LyndonWord Symbol

```
factor(w2*w1)$lword
      2      2
[[a b ],[a b ]]
```

Type: List LyndonWord Symbol

=====

Checks and coercions

=====

```
lyndon?(w1)$lword
true
```

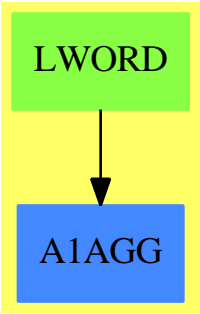
Type: Boolean

```
lyndon?(w1*w2)$lword
true
```

	Type: Boolean
lyndon?(w2*w1)\$lword false	
	Type: Boolean
lyndonIfCan(w1)\$lword 2 [a b]	
	Type: Union(LyndonWord Symbol,...)
lyndonIfCan(w2*w1)\$lword "failed"	
	Type: Union("failed",...)
lyndon(w1)\$lword 2 [a b]	
	Type: LyndonWord Symbol
lyndon(w1*w2)\$lword 2 2 [a b a b]	
	Type: LyndonWord Symbol

See Also:
o)show LyndonWord

LyndonWord (LWORD)



Exports:

coerce	factor	hash	latex	left
length	lexico	lyndon	lyndon?	lyndonIfCan
LyndonWordsList	LyndonWordsList1	max	min	retract
retractIfCan	retractable?	right	varList	?<?
?<=?	?=?	?>?	?>=?	?~=?

— domain LWORD LyndonWord —

```

)abbrev domain LWORD LyndonWord
++ Author: Michel Petitot (petitot@lifl.fr).
++ Date Created: 91
++ Date Last Updated: 7 Juillet 92
++ Fix History: compilation v 2.1 le 13 dec 98
++ References:
++ Free Lie Algebras by C. Reutenauer (Oxford science publications).
++ Description:
++ Lyndon words over arbitrary (ordered) symbols:
++ see Free Lie Algebras by C. Reutenauer (Oxford science publications).
++ A Lyndon word is a word which is smaller than any of its right factors
++ w.r.t. the pure lexicographical ordering.
++ If \axiom{a} and \axiom{b} are two Lyndon words such that \axiom{a < b}
++ holds w.r.t lexicographical ordering then \axiom{a*b} is a Lyndon word.
++ Parenthesized Lyndon words can be generated from symbols by using the
++ following rule:\br
++ \axiom{[[a,b],c]} is a Lyndon word iff \axiom{a*b < c <= b} holds.\br
++ Lyndon words are internally represented by binary trees using the
++ \spadtype{Magma} domain constructor.
++ Two ordering are provided: lexicographic and
++ length-lexicographic.

LyndonWord(VarSet:OrderedSet):Public == Private where
  OFMON ==> OrderedFreeMonoid(VarSet)
  PI    ==> PositiveInteger
  NNI   ==> NonNegativeInteger
  I     ==> Integer
  OF    ==> OutputForm
  ARRAY1==> OneDimensionalArray

Public == Join(OrderedSet,RetractableTo VarSet) with
  retractable? : $ -> Boolean
    ++ \axiom{retractable?(x)} tests if \axiom{x} is a tree
    ++ with only one entry.
  left        : $ -> $
    ++ \axiom{left(x)} returns left subtree of \axiom{x} or
    ++ error if retractable?(x) is true.
  right       : $ -> $
    ++ \axiom{right(x)} returns right subtree of \axiom{x} or
    ++ error if retractable?(x) is true.
  length      : $ -> PI

```

```

    ++ \axiom{length(x)} returns the number of entries in \axiom{x}.
lexico : ($,$) -> Boolean
    ++ \axiom{lexico(x,y)} returns \axiom{true} iff \axiom{x} is smaller than
    ++ \axiom{y} w.r.t. the lexicographical ordering induced by \axiom{VarSet}.
coerce : $ -> OFMON
    ++ \axiom{coerce(x)} returns the element of \axiomType{OrderedFreeMonoid}(VarSet)
    ++ corresponding to \axiom{x}.
coerce : $ -> Magma VarSet
    ++ \axiom{coerce(x)} returns the element of \axiomType{Magma}(VarSet)
    ++ corresponding to \axiom{x}.
factor : OFMON -> List $
    ++ \axiom{factor(x)} returns the decreasing factorization into Lyndon words.
lyndon?: OFMON -> Boolean
    ++ \axiom{lyndon?(w)} test if \axiom{w} is a Lyndon word.
lyndon : OFMON -> $
    ++ \axiom{lyndon(w)} convert \axiom{w} into a Lyndon word,
    ++ error if \axiom{w} is not a Lyndon word.
lyndonIfCan : OFMON -> Union($, "failed")
    ++ \axiom{lyndonIfCan(w)} convert \axiom{w} into a Lyndon word.
varList : $ -> List VarSet
    ++ \axiom{varList(x)} returns the list of distinct entries of \axiom{x}.
LyndonWordsList1: (List VarSet, PI) -> ARRAY1 List $
    ++ \axiom{LyndonWordsList1(vl, n)} returns an array of lists of Lyndon
    ++ words over the alphabet \axiom{vl}, up to order \axiom{n}.
LyndonWordsList : (List VarSet, PI) -> List $
    ++ \axiom{LyndonWordsList(vl, n)} returns the list of Lyndon
    ++ words over the alphabet \axiom{vl}, up to order \axiom{n}.

Private == Magma(VarSet) add
-- Representation
Rep:= Magma(VarSet)

-- Fonctions locales
LetterList : OFMON -> List VarSet
factor1 : (List $, $, List $) -> List $

-- Definitions
lyndon? w ==
    w = 1$OFMON => false
    f: OFMON := rest w
    while f ^= 1$OFMON repeat
        not lexico(w,f) => return false
        f := rest f
    true

lyndonIfCan w ==
    l: List $ := factor w
    # l = 1 => first l
    "failed"

```



```

lyndon w ==
  l: List $ := factor w
  # l = 1 => first l
  error "This word is not a Lyndon word"

LetterList w ==
  w = 1 => []
  cons(first w , LetterList rest w)

factor1 (gauche, x, droite) ==
  g: List $ := gauche; d: List $ := droite
  while not null g repeat
    lexico( g.first , x ) =>
      x := g.first *$Rep x
      null(d) => g := rest g
      g := cons( x, rest g)
      x := first d
      d := rest d
      d := cons( x , d)
      x := first g
      g := rest g
  return cons(x, d)

factor w ==
  w = 1 => []
  l : List $ := reverse [ u:$ for u in LetterList w]
  factor1( rest l, first l , [] )

x < y ==
  lx,ly: PI
  lx:= length x ; ly:= length y
  lx = ly => lexico(x,y)
  lx < ly

coerce(x:$):OF == bracket(x::OFMON::OF)
coerce(x:$):Magma VarSet == x::Rep

LyndonWordsList1 (vl,n) ==
  null vl => error "empty list"
  base: ARRAY1 List $ := new(n::I::NNI ,[])

  -- mots de longueur 1
  lbase1:List $ := [w:$ for w in sort(vl)]
  base.1 := lbase1

  -- calcul des mots de longueur ll
  for ll in 2..n:I repeat
    lbase1 := []
    for a in base(1) repeat
      for b in base(ll-1) repeat

```

```

        if lexico(a,b) then lbase1:=cons(a*b,lbase1)

    for i in 2..ll-1 repeat          -- mot + mot
    for a in base(i) repeat
    for b in base(ll-i) repeat
    if lexico(a,b) and (lexico(b,right a) or b = right a )
    then lbase1:=cons(a*b,lbase1)

    base(ll):= sort_!(lexico, lbase1)
return base

LyndonWordsList (v1,n) ==
v:ARRAY1 List $ := LyndonWordsList1(v1,n)
"append"/ [v.i for i in 1..n]

```

— LWORD.dotabb —

```

"LWORD" [color="#88FF44",href="bookvol10.3.pdf#nameddest=LWORD"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"LWORD" -> "A1AGG"

```

Chapter 14

Chapter M

domain MCMPLX MachineComplex

— MachineComplex.input —

```
)set break resume
)sys rm -f MachineComplex.output
)spool MachineComplex.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MachineComplex
--R MachineComplex is a domain constructor
--R Abbreviation for MachineComplex is MCMPLX
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for MCMPLX
--R
--R----- Operations -----
--R ??? : (%,MachineFloat) -> %          ??? : (MachineFloat,%) -> %
--R ??? : (%,%) -> %                    ??? : (Integer,%) -> %
--R ??? : (PositiveInteger,%) -> %      ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %                    ?-? : (%,%) -> %
--R -? : % -> %                         ?<? : (%,%) -> Boolean
--R ?<=? : (%,%) -> Boolean              ?=? : (%,%) -> Boolean
--R ?>? : (%,%) -> Boolean              ?>=? : (%,%) -> Boolean
--R 1 : () -> %                         0 : () -> %
--R ??? : (%,PositiveInteger) -> %      associates? : (%,%) -> Boolean
--R basis : () -> Vector(%)              coerce : % -> Complex(Float)
--R coerce : Complex(Integer) -> %        coerce : Complex(Float) -> %
--R coerce : MachineFloat -> %           coerce : Integer -> %
```

```

--R coerce : % -> %
--R coerce : % -> OutputForm
--R discriminant : () -> MachineFloat
--R hash : % -> SingleInteger
--R imaginary : () -> %
--R max : (%,%) -> %
--R norm : % -> MachineFloat
--R rank : () -> PositiveInteger
--R recip : % -> Union(%, "failed")
--R retract : % -> Integer
--R trace : % -> MachineFloat
--R unitCanonical : % -> %
--R ?^=? : (%,%) -> Boolean
--R ?*? : (%, Fraction(Integer)) -> % if MachineFloat has FIELD
--R ?*? : (Fraction(Integer), %) -> % if MachineFloat has FIELD
--R ?*? : (NonNegativeInteger, %) -> %
--R ?**? : (%, Integer) -> % if MachineFloat has FIELD
--R ?**? : (%, Fraction(Integer)) -> % if MachineFloat has RADCAT and MachineFloat has TRANFUN
--R ?**? : (%, %) -> % if MachineFloat has TRANFUN
--R ?**? : (%, NonNegativeInteger) -> %
--R ?/? : (%, %) -> % if MachineFloat has FIELD
--R D : % -> % if MachineFloat has DIFRING
--R D : (%, NonNegativeInteger) -> % if MachineFloat has DIFRING
--R D : (%, Symbol) -> % if MachineFloat has PDRING(SYMBOL)
--R D : (%, List(Symbol)) -> % if MachineFloat has PDRING(SYMBOL)
--R D : (%, Symbol, NonNegativeInteger) -> % if MachineFloat has PDRING(SYMBOL)
--R D : (%, List(Symbol), List(NonNegativeInteger)) -> % if MachineFloat has PDRING(SYMBOL)
--R D : (%, (MachineFloat -> MachineFloat), NonNegativeInteger) -> %
--R D : (%, (MachineFloat -> MachineFloat)) -> %
--R ?^? : (%, Integer) -> % if MachineFloat has FIELD
--R ?^? : (%, NonNegativeInteger) -> %
--R abs : % -> % if MachineFloat has RNS
--R acos : % -> % if MachineFloat has TRANFUN
--R acosh : % -> % if MachineFloat has TRANFUN
--R acot : % -> % if MachineFloat has TRANFUN
--R acoth : % -> % if MachineFloat has TRANFUN
--R acsc : % -> % if MachineFloat has TRANFUN
--R acsch : % -> % if MachineFloat has TRANFUN
--R argument : % -> MachineFloat if MachineFloat has TRANFUN
--R asec : % -> % if MachineFloat has TRANFUN
--R asech : % -> % if MachineFloat has TRANFUN
--R asin : % -> % if MachineFloat has TRANFUN
--R asinh : % -> % if MachineFloat has TRANFUN
--R atan : % -> % if MachineFloat has TRANFUN
--R atanh : % -> % if MachineFloat has TRANFUN
--R characteristic : () -> NonNegativeInteger
--R characteristicPolynomial : % -> SparseUnivariatePolynomial(MachineFloat)
--R charthRoot : % -> Union(%, "failed") if $ has CHARNZ and MachineFloat has EUCDOM and MachineF
--R charthRoot : % -> % if MachineFloat has FFIELDC
--R coerce : Fraction(Integer) -> % if MachineFloat has FIELD or MachineFloat has RETRACT(FRAC(I

```

```

--R coerce : Complex(MachineInteger) -> %
--R coerce : Complex(MachineFloat) -> %
--R complex : (MachineFloat,MachineFloat) -> %
--R conditionP : Matrix(%) -> Union(Vector(%) ,"failed") if $ has CHARNZ and MachineFloat has EUCDOM and Mac
--R convert : % -> Vector(MachineFloat)
--R convert : Vector(MachineFloat) -> %
--R convert : % -> SparseUnivariatePolynomial(MachineFloat)
--R convert : SparseUnivariatePolynomial(MachineFloat) -> %
--R convert : % -> Pattern(Integer) if MachineFloat has KONVERT(PATTERN(INT))
--R convert : % -> Pattern(Float) if MachineFloat has KONVERT(PATTERN(FLOAT))
--R convert : % -> Complex(Float) if MachineFloat has REAL
--R convert : % -> Complex(DoubleFloat) if MachineFloat has REAL
--R convert : % -> InputForm if MachineFloat has KONVERT(INFORM)
--R coordinates : (%,Vector(%)) -> Vector(MachineFloat)
--R coordinates : (Vector(%),Vector(%)) -> Matrix(MachineFloat)
--R coordinates : % -> Vector(MachineFloat)
--R coordinates : Vector(%) -> Matrix(MachineFloat)
--R cos : % -> % if MachineFloat has TRANFUN
--R cosh : % -> % if MachineFloat has TRANFUN
--R cot : % -> % if MachineFloat has TRANFUN
--R coth : % -> % if MachineFloat has TRANFUN
--R createPrimitiveElement : () -> % if MachineFloat has FFIELDC
--R csc : % -> % if MachineFloat has TRANFUN
--R csch : % -> % if MachineFloat has TRANFUN
--R definingPolynomial : () -> SparseUnivariatePolynomial(MachineFloat)
--R derivationCoordinates : (Vector(%),(MachineFloat -> MachineFloat)) -> Matrix(MachineFloat) if MachineFl
--R differentiate : % -> % if MachineFloat has DIFRING
--R differentiate : (%,NonNegativeInteger) -> % if MachineFloat has DIFRING
--R differentiate : (%,Symbol) -> % if MachineFloat has PDRING(SYMBOL)
--R differentiate : (%,List(Symbol)) -> % if MachineFloat has PDRING(SYMBOL)
--R differentiate : (%,Symbol,NonNegativeInteger) -> % if MachineFloat has PDRING(SYMBOL)
--R differentiate : (%,List(Symbol),List(NonNegativeInteger)) -> % if MachineFloat has PDRING(SYMBOL)
--R differentiate : (%,(MachineFloat -> MachineFloat),NonNegativeInteger) -> %
--R differentiate : (%,(MachineFloat -> MachineFloat)) -> %
--R discreteLog : % -> NonNegativeInteger if MachineFloat has FFIELDC
--R discreteLog : (%,%) -> Union(NonNegativeInteger,"failed") if MachineFloat has FFIELDC
--R discriminant : Vector(%) -> MachineFloat
--R divide : (%,%) -> Record(quotient: %,remainder: %) if MachineFloat has EUCDOM
--R ?? : (%,MachineFloat) -> % if MachineFloat has ELTAB(MFLOAT,MFLOAT)
--R enumerate : () -> List(%) if MachineFloat has FINITE
--R euclideanSize : % -> NonNegativeInteger if MachineFloat has EUCDOM
--R eval : (%,List(MachineFloat),List(MachineFloat)) -> % if MachineFloat has EVALAB(MFLOAT)
--R eval : (%,MachineFloat,MachineFloat) -> % if MachineFloat has EVALAB(MFLOAT)
--R eval : (%,Equation(MachineFloat)) -> % if MachineFloat has EVALAB(MFLOAT)
--R eval : (%,List(Equation(MachineFloat))) -> % if MachineFloat has EVALAB(MFLOAT)
--R eval : (%,List(Symbol),List(MachineFloat)) -> % if MachineFloat has IEVALAB(SYMBOL,MFLOAT)
--R eval : (%,Symbol,MachineFloat) -> % if MachineFloat has IEVALAB(SYMBOL,MFLOAT)
--R exp : % -> % if MachineFloat has TRANFUN
--R expressIdealMember : (List(%),%) -> Union(List(%),"failed") if MachineFloat has EUCDOM
--R exquo : (%,MachineFloat) -> Union(%,"failed") if MachineFloat has INTDOM

```

```

--R exquo : (% , %) -> Union(%, "failed")
--R extendedEuclidean : (% , %) -> Record(coef1: %, coef2: %, generator: %) if MachineFloat has EUCLD
--R extendedEuclidean : (% , %, %) -> Union(Record(coef1: %, coef2: %), "failed") if MachineFloat has
--R factor : % -> Factored(%) if MachineFloat has EUCDOM and MachineFloat has PFECAT or MachineF
--R factorPolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%))
--R factorSquareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolyn
--R factorsOfCyclicGroupSize : () -> List(Record(factor: Integer, exponent: Integer)) if MachineF
--R gcd : (% , %) -> % if MachineFloat has EUCDOM
--R gcd : List(%) -> % if MachineFloat has EUCDOM
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUniva
--R index : PositiveInteger -> % if MachineFloat has FINITE
--R init : () -> % if MachineFloat has FFIELDC
--R inv : % -> % if MachineFloat has FIELD
--R lcm : (% , %) -> % if MachineFloat has EUCDOM
--R lcm : List(%) -> % if MachineFloat has EUCDOM
--R lift : % -> SparseUnivariatePolynomial(MachineFloat)
--R log : % -> % if MachineFloat has TRANFUN
--R lookup : % -> PositiveInteger if MachineFloat has FINITE
--R map : ((MachineFloat -> MachineFloat), %) -> %
--R minimalPolynomial : % -> SparseUnivariatePolynomial(MachineFloat) if MachineFloat has FIELD
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed") if MachineFloat has EUCDOM
--R nextItem : % -> Union(%, "failed") if MachineFloat has FFIELDC
--R nthRoot : (% , Integer) -> % if MachineFloat has RADCAT and MachineFloat has TRANFUN
--R order : % -> PositiveInteger if MachineFloat has FFIELDC
--R order : % -> OnePointCompletion(PositiveInteger) if MachineFloat has FFIELDC
--R patternMatch : (% , Pattern(Integer), PatternMatchResult(Integer, %)) -> PatternMatchResult(Inte
--R patternMatch : (% , Pattern(Float), PatternMatchResult(Float, %)) -> PatternMatchResult(Float, %)
--R pi : () -> % if MachineFloat has TRANFUN
--R polarCoordinates : % -> Record(r: MachineFloat, phi: MachineFloat) if MachineFloat has RNS an
--R prime? : % -> Boolean if MachineFloat has EUCDOM and MachineFloat has PFECAT or MachineFloat
--R primeFrobenius : (% , NonNegativeInteger) -> % if MachineFloat has FFIELDC
--R primeFrobenius : % -> % if MachineFloat has FFIELDC
--R primitive? : % -> Boolean if MachineFloat has FFIELDC
--R primitiveElement : () -> % if MachineFloat has FFIELDC
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %) if MachineFloat has EUCDOM
--R ?quo? : (% , %) -> % if MachineFloat has EUCDOM
--R random : () -> % if MachineFloat has FINITE
--R rational : % -> Fraction(Integer) if MachineFloat has INS
--R rational? : % -> Boolean if MachineFloat has INS
--R rationalIfCan : % -> Union(Fraction(Integer), "failed") if MachineFloat has INS
--R reduce : SparseUnivariatePolynomial(MachineFloat) -> %
--R reduce : Fraction(SparseUnivariatePolynomial(MachineFloat)) -> Union(%, "failed") if MachineF
--R reducedSystem : Matrix(%) -> Matrix(Integer) if MachineFloat has LINEXP(INT)
--R reducedSystem : (Matrix(%), Vector(%)) -> Record(mat: Matrix(Integer), vec: Vector(Integer)) i
--R reducedSystem : (Matrix(%), Vector(%)) -> Record(mat: Matrix(MachineFloat), vec: Vector(Machin
--R reducedSystem : Matrix(%) -> Matrix(MachineFloat)
--R regularRepresentation : (% , Vector(%)) -> Matrix(MachineFloat)
--R regularRepresentation : % -> Matrix(MachineFloat)
--R ?rem? : (% , %) -> % if MachineFloat has EUCDOM
--R representationType : () -> Union("prime", polynomial, normal, cyclic) if MachineFloat has FFIEL

```

```

--R represents : (Vector(MachineFloat),Vector(%)) -> %
--R represents : Vector(MachineFloat) -> %
--R retract : % -> Fraction(Integer) if MachineFloat has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if MachineFloat has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(MachineFloat,"failed")
--R retractIfCan : % -> Union(Integer,"failed")
--R sec : % -> % if MachineFloat has TRANFUN
--R sech : % -> % if MachineFloat has TRANFUN
--R sin : % -> % if MachineFloat has TRANFUN
--R sinh : % -> % if MachineFloat has TRANFUN
--R size : () -> NonNegativeInteger if MachineFloat has FINITE
--R sizeLess? : (%,%) -> Boolean if MachineFloat has EUCDOM
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(%)),SparseUnivariatePolynomial(%)) -> %
--R sqrt : % -> % if MachineFloat has RADCAT and MachineFloat has TRANFUN
--R squareFree : % -> Factored(%) if MachineFloat has EUCDOM and MachineFloat has PFECAT or MachineFloat has FIEL
--R squareFreePart : % -> % if MachineFloat has EUCDOM and MachineFloat has PFECAT or MachineFloat has FIEL
--R squareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if Mach
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R tableForDiscreteLogarithm : Integer -> Table(PositiveInteger,NonNegativeInteger) if MachineFloat has FF
--R tan : % -> % if MachineFloat has TRANFUN
--R tanh : % -> % if MachineFloat has TRANFUN
--R traceMatrix : Vector(%) -> Matrix(MachineFloat)
--R traceMatrix : () -> Matrix(MachineFloat)
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R
--E 1

)spool
)lisp (bye)

```

— MachineComplex.help —

```

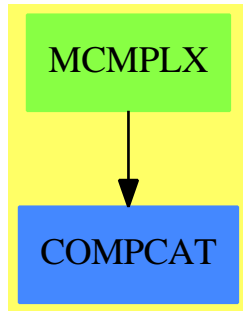
=====
MachineComplex examples
=====

```

A domain which models the complex number representation used by machines in the AXIOM-NAG link.

See Also:

o)show MachineComplex

MachineComplex (MCMPLX)**See**

- ⇒ “MachineInteger” (MINT) [14](#) on page [1557](#)
- ⇒ “MachineFloat” (MFLOAT) [14](#) on page [1547](#)

Exports:

0	1	abs
acos	acosh	acot
acoth	acsc	acsch
argument	asec	asech
asin	asinh	associates?
atan	atanh	basis
characteristic	characteristicPolynomial	charthRoot
coerce	complex	conditionP
conjugate	convert	coordinates
cos	cosh	cot
coth	createPrimitiveElement	csc
csch	D	definingPolynomial
derivationCoordinates	differentiate	discreteLog
discriminant	divide	euclideanSize
eval	exp	expressIdealMember
exquo	extendedEuclidean	factor
factorPolynomial	factorSquareFreePolynomial	gcdPolynomial
gcd	factorsOfCyclicGroupSize	generator
hash	imag	imaginary
index	init	inv
latex	lcm	lift
log	lookup	map
max	min	minimalPolynomial
multiEuclidean	nextItem	norm
nthRoot	one?	order
patternMatch	pi	polarCoordinates
prime?	primeFrobenius	primitive?
primitiveElement	principalIdeal	random
rank	rational	rational?
rationalIfCan	real	recip
reduce	reducedSystem	regularRepresentation
representationType	represents	retract
retractIfCan	sample	sec
sech	sin	sinh
size	solveLinearPolynomialEquation	sizeLess?
sqrt	squareFree	squareFreePart
squareFreePolynomial	tableForDiscreteLogarithm	subtractIfCan
tan	tanh	trace
traceMatrix	traceMatrix	unit?
unitCanonical	unitNormal	zero?
?*?	?**?	?+?
?-?	-?	?<?
?<=?	?=?	?>?
?>=?	?^?	?~=?
?/?	?..?	?quo?
?rem?		

— domain MCMPLX MachineComplex —

```

)abbrev domain MCMPLX MachineComplex
++ Date Created:  December 1993
++ Description:
++ A domain which models the complex number representation
++ used by machines in the AXIOM-NAG link.

```

```

MachineComplex():Exports == Implementation where

```

```

Exports ==> Join (FortranMachineTypeCategory,
                  ComplexCategory(MachineFloat)) with
coerce : Complex Float -> $
  ++ coerce(u) transforms u into a MachineComplex
coerce : Complex Integer -> $
  ++ coerce(u) transforms u into a MachineComplex
coerce : Complex MachineFloat -> $
  ++ coerce(u) transforms u into a MachineComplex
coerce : Complex MachineInteger -> $
  ++ coerce(u) transforms u into a MachineComplex
coerce : $ -> Complex Float
  ++ coerce(u) transforms u into a CComplex Float

```

```

Implementation ==> Complex MachineFloat add

```

```

coerce(u:Complex Float):$ ==
  complex(real(u)::MachineFloat,imag(u)::MachineFloat)

coerce(u:Complex Integer):$ ==
  complex(real(u)::MachineFloat,imag(u)::MachineFloat)

coerce(u:Complex MachineInteger):$ ==
  complex(real(u)::MachineFloat,imag(u)::MachineFloat)

coerce(u:Complex MachineFloat):$ ==
  complex(real(u),imag(u))

coerce(u:$):Complex Float ==
  complex(real(u)::Float,imag(u)::Float)

```

— MCMPLX.dotabb —

```

"MCMPLX" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MCMPLX"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"MCMPLX" -> "COMPCAT"

```

domain MFLOAT MachineFloat

— MachineFloat.input —

```

)set break resume
)sys rm -f MachineFloat.output
)spool MachineFloat.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MachineFloat
--R
--R MachineFloat is a domain constructor
--R Abbreviation for MachineFloat is MFLOAT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for MFLOAT
--R
--R----- Operations -----
--R ??? : (Fraction(Integer),%) -> %      ??? : (%,Fraction(Integer)) -> %
--R ??? : (%,%) -> %                      ??? : (Integer,%) -> %
--R ??? : (PositiveInteger,%) -> %        ??? : (%,Integer) -> %
--R ??? : (%,PositiveInteger) -> %        ?+? : (%,%) -> %
--R ?-? : (%,%) -> %                      -? : % -> %
--R ?/? : (%,%) -> %                      ?<? : (%,%) -> Boolean
--R ?<=? : (%,%) -> Boolean                ?=? : (%,%) -> Boolean
--R ?>? : (%,%) -> Boolean                ?>=? : (%,%) -> Boolean
--R 1 : () -> %                           0 : () -> %
--R ?^? : (%,Integer) -> %                 ?^? : (%,PositiveInteger) -> %
--R abs : % -> %                           associates? : (%,%) -> Boolean
--R base : () -> PositiveInteger            bits : () -> PositiveInteger
--R ceiling : % -> %                       coerce : MachineInteger -> %
--R coerce : % -> Float                     coerce : Float -> %
--R coerce : Fraction(Integer) -> %         coerce : Integer -> %
--R coerce : Fraction(Integer) -> %         coerce : % -> %
--R coerce : Integer -> %                   coerce : % -> OutputForm
--R convert : % -> Pattern(Float)           convert : % -> DoubleFloat
--R convert : % -> Float                     digits : () -> PositiveInteger
--R exponent : % -> Integer                  factor : % -> Factored(%)
--R float : (Integer,Integer) -> %          floor : % -> %
--R fractionPart : % -> %                   gcd : List(%) -> %
--R gcd : (%,%) -> %                       hash : % -> SingleInteger

```

```

--R inv : % -> %
--R lcm : List(%) -> %
--R mantissa : % -> Integer
--R maximumExponent : () -> Integer
--R minimumExponent : () -> Integer
--R norm : % -> %
--R one? : % -> Boolean
--R positive? : % -> Boolean
--R prime? : % -> Boolean
--R recip : % -> Union(%, "failed")
--R retract : % -> Float
--R retract : % -> Integer
--R sample : () -> %
--R sizeLess? : (%, %) -> Boolean
--R squareFree : % -> Factored(%)
--R truncate : % -> %
--R unitCanonical : % -> %
--R zero? : % -> Boolean
--R ?? : (NonNegativeInteger, %) -> %
--R ***? : (%, Fraction(Integer)) -> %
--R ***? : (%, NonNegativeInteger) -> %
--R ??^? : (%, NonNegativeInteger) -> %
--R base : PositiveInteger -> PositiveInteger
--R bits : PositiveInteger -> PositiveInteger if $ has arbitraryPrecision
--R changeBase : (Integer, Integer, PositiveInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R decreasePrecision : Integer -> PositiveInteger if $ has arbitraryPrecision
--R digits : PositiveInteger -> PositiveInteger if $ has arbitraryPrecision
--R divide : (%, %) -> Record(quotient: %, remainder: %)
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(%), %) -> Union(List(%), "failed")
--R exquo : (%, %) -> Union(%, "failed")
--R extendedEuclidean : (%, %, %) -> Union(Record(coef1: %, coef2: %), "failed")
--R extendedEuclidean : (%, %) -> Record(coef1: %, coef2: %, generator: %)
--R float : (Integer, Integer, PositiveInteger) -> %
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R increasePrecision : Integer -> PositiveInteger if $ has arbitraryPrecision
--R max : () -> % if not(has($, arbitraryExponent)) and not(has($, arbitraryPrecision))
--R maximumExponent : Integer -> Integer
--R min : () -> % if not(has($, arbitraryExponent)) and not(has($, arbitraryPrecision))
--R minimumExponent : Integer -> Integer
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed")
--R patternMatch : (%, Pattern(Float), PatternMatchResult(Float, %)) -> PatternMatchResult(Float, %)
--R precision : PositiveInteger -> PositiveInteger
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %)
--R retractIfCan : % -> Union(Float, "failed")
--R retractIfCan : % -> Union(Fraction(Integer), "failed")
--R retractIfCan : % -> Union(Integer, "failed")
--R subtractIfCan : (%, %) -> Union(%, "failed")
--R unitNormal : % -> Record(unit: %, canonical: %, associate: %)
--R latex : % -> String
--R lcm : (%, %) -> %
--R max : (%, %) -> %
--R min : (%, %) -> %
--R negative? : % -> Boolean
--R nthRoot : (%, Integer) -> %
--R order : % -> Integer
--R precision : () -> PositiveInteger
--R ?quo? : (%, %) -> %
--R ?rem? : (%, %) -> %
--R retract : % -> Fraction(Integer)
--R round : % -> %
--R sign : % -> Integer
--R sqrt : % -> %
--R squareFreePart : % -> %
--R unit? : % -> Boolean
--R wholePart : % -> Integer
--R ~=? : (%, %) -> Boolean

```

```
--R
--E 1
```

```
)spool
)lisp (bye)
```

— MachineFloat.help —

=====

MachineFloat examples

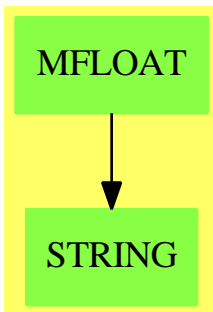
=====

A domain which models the floating point representation used by machines in the AXIOM-NAG link.

See Also:

o)show MachineFloat

MachineFloat (MFLOAT)



See

⇒ “MachineInteger” (MINT) [14](#) on page [1557](#)
 ⇒ “MachineComplex” (MCMPLX) [14](#) on page [1542](#)

Exports:

1	0	abs	associates?
base	bits	ceiling	coerce
changeBase	characteristic	convert	decreasePrecision
digits	exponent	divide	euclideanSize
expressIdealMember	exquo	extendedEuclidean	factor
float	floor	fractionPart	gcd
gcdPolynomial	hash	increasePrecision	inv
latex	lcm	mantissa	max
maximumExponent	min	minimumExponent	multiEuclidean
negative?	norm	nthRoot	one?
order	patternMatch	positive?	precision
prime?	principalIdeal	recip	retract
retractIfCan	round	sample	sign
sizeLess?	sqrt	squareFree	squareFreePart
subtractIfCan	truncate	unit?	unitCanonical
unitNormal	wholePart	zero?	?*?
?**?	?+?	?-?	-?
?/?	?<?	?<=?	?=?
?>?	?>=?	?~=?	?^?
?quo?	?rem?		

— domain MFLOAT MachineFloat —

```

)abbrev domain MFLOAT MachineFloat
++ Author: Mike Dewar
++ Date Created: December 1993
++ Description:
++ A domain which models the floating point representation
++ used by machines in the AXIOM-NAG link.

MachineFloat(): Exports == Implementation where

  PI ==> PositiveInteger
  NNI ==> NonNegativeInteger
  F ==> Float
  I ==> Integer
  S ==> String
  FI ==> Fraction Integer
  SUP ==> SparseUnivariatePolynomial
  SF ==> DoubleFloat

Exports ==> Join(FloatingPointSystem, FortranMachineTypeCategory, Field,
  RetractableTo(Float), RetractableTo(Fraction(Integer)), CharacteristicZero) with
  precision : PI -> PI
    ++ precision(p) sets the number of digits in the model to p
  precision : () -> PI
    ++ precision() returns the number of digits in the model

```

```

base    : PI -> PI
  ++ base(b) sets the base of the model to b
base    : () -> PI
  ++ base() returns the base of the model
maximumExponent : I -> I
  ++ maximumExponent(e) sets the maximum exponent in the model to e
maximumExponent : () -> I
  ++ maximumExponent() returns the maximum exponent in the model
minimumExponent : I -> I
  ++ minimumExponent(e) sets the minimum exponent in the model to e
minimumExponent : () -> I
  ++ minimumExponent() returns the minimum exponent in the model
coerce   : $ -> F
  ++ coerce(u) transforms a MachineFloat to a standard Float
coerce   : MachineInteger -> $
  ++ coerce(u) transforms a MachineInteger into a MachineFloat
mantissa : $ -> I
  ++ mantissa(u) returns the mantissa of u
exponent : $ -> I
  ++ exponent(u) returns the exponent of u
changeBase : (I,I,PI) -> $
  ++ changeBase(exp,man,base) is not documented

Implementation ==> add

import F
import FI

Rep := Record(mantissa:I,exponent:I)

-- Parameters of the Floating Point Representation
P : PI := 16      -- Precision
B : PI := 2       -- Base
EMIN : I := -1021 -- Minimum Exponent
EMAX : I := 1024  -- Maximum Exponent

-- Useful constants
POWER : PI := 53 -- The maximum power of B which will yield P
              -- decimal digits.
MMAX  : PI := B**POWER

-- locals
locRound:(FI)->I
checkExponent:($)->$
normalise:($)->$
newPower:(PI,PI)->Void

retractIfCan(u:$):Union(FI,"failed") ==
  mantissa(u)*(B/1)**(exponent(u))

```



```

wholePart(u:$):Integer ==
  man:I:=mantissa u
  exp:I:=exponent u
  f:=
    positive? exp => man*B**(exp pretend PI)
    zero? exp => man
    wholePart(man/B**((-exp) pretend PI))
normalise(u:$):$ ==
  -- We want the largest possible mantissa, to ensure a canonical
  -- representation.
  exp : I := exponent u
  man : I := mantissa u
  BB  : I := B @ I
  sgn : I := sign man ; man := abs man
  zero? man => [0,0]$Rep
  if man < MMAX then
    while man < MMAX repeat
      exp := exp - 1
      man := man * BB
  if man > MMAX then
    q1:FI:= man/1
    BBF:FI:=BB/1
    while wholePart(q1) > MMAX repeat
      q1:= q1 / BBF
      exp:=exp + 1
      man := locRound(q1)
  positive?(sgn) => checkExponent [man,exp]$Rep
  checkExponent [-man,exp]$Rep

mantissa(u:$):I == elt(u,mantissa)$Rep
exponent(u:$):I == elt(u,exponent)$Rep

newPower(base:PI,prec:PI):Void ==
  power : PI := 1
  target : PI := 10**prec
  current : PI := base
  while (current := current*base) < target repeat power := power+1
  POWER := power
  MMAX := B**POWER
  void()

changeBase(exp:I,man:I,base:PI):$ ==
  newExp : I := 0
  f : FI := man*(base @ I)::FI**exp
  sign : I := sign f
  f : FI := abs f
  newMan : I := wholePart f
  zero? f => [0,0]$Rep
  BB : FI := (B @ I)::FI

```

```

    if newMan < MMAX then
      while newMan < MMAX repeat
        newExp := newExp - 1
        f := f*BB
        newMan := wholePart f
    if newMan > MMAX then
      while newMan > MMAX repeat
        newExp := newExp + 1
        f := f/BB
        newMan := wholePart f
    [sign*newMan,newExp]$Rep

checkExponent(u:$):$ ==
  exponent(u) < EMIN or exponent(u) > EMAX =>
    message :S := concat(["Exponent out of range: ",
                          convert(EMIN)@S, "..", convert(EMAX)@S])$S
    error message
  u

coerce(u:$):OutputForm ==
  coerce(u::F)

coerce(u:MachineInteger):$ ==
  checkExponent changeBase(0,retract(u)@Integer,10)

coerce(u:$):F ==
  oldDigits : PI := digits(P)$F
  r : F := float(mantissa u,exponent u,B)$Float
  digits(oldDigits)$F
  r

coerce(u:F):$ ==
  checkExponent changeBase(exponent(u)$F,mantissa(u)$F,base())$F

coerce(u:I):$ ==
  checkExponent changeBase(0,u,10)

coerce(u:FI):$ == (numer u)::$(denom u)::$

retract(u:$):FI ==
  value : Union(FI,"failed") := retractIfCan(u)
  value case "failed" => error "Cannot retract to a Fraction Integer"
  value::FI

retract(u:$):F == u::F

retractIfCan(u:$):Union(F,"failed") == u::F::Union(F,"failed")

retractIfCan(u:$):Union(I,"failed") ==
  value:FI := mantissa(u)*(B @ I)::FI**exponent(u)

```

```

zero? fractionPart(value) => wholePart(value)::Union(I,"failed")
"failed"::Union(I,"failed")

retract(u:$):I ==
  result : Union(I,"failed") := retractIfCan u
  result = "failed" => error "Not an Integer"
  result::I

precision(p: PI):PI ==
  old : PI := P
  newPower(B,p)
  P := p
  old

precision():PI == P

base(b:PI):PI ==
  old : PI := b
  newPower(b,P)
  B := b
  old

base():PI == B

maximumExponent(u:I):I ==
  old : I := EMAX
  EMAX := u
  old

maximumExponent():I == EMAX

minimumExponent(u:I):I ==
  old : I := EMIN
  EMIN := u
  old

minimumExponent():I == EMIN

0 == [0,0]$Rep
1 == changeBase(0,1,10)

zero?(u:$):Boolean == u=[0,0]$Rep

f1:$
f2:$

locRound(x:FI):I ==

```

```

    abs(fractionPart(x)) >= 1/2 => wholePart(x)+sign(x)
    wholePart(x)

    recip f1 ==
      zero? f1 => "failed"
      normalise [ locRound(B**(2*POWER)/mantissa f1),-(exponent f1 + 2*POWER)]

    f1 * f2 ==
      normalise [mantissa(f1)*mantissa(f2),exponent(f1)+exponent(f2)]$Rep

    f1 **(p:FI) ==
      ((f1::F)**p)::%

--inline
    f1 / f2 ==
      zero? f2 => error "division by zero"
      zero? f1 => 0
      f1=f2 => 1
      normalise [locRound(mantissa(f1)*B**(2*POWER)/mantissa(f2)),
        exponent(f1)-(exponent f2 + 2*POWER)]

    inv(f1) == 1/f1

    f1 exquo f2 == f1/f2

    divide(f1,f2) == [ f1/f2,0]

    f1 quo f2 == f1/f2
    f1 rem f2 == 0
    u:I * f1 ==
      normalise [u*mantissa(f1),exponent(f1)]$Rep

    f1 = f2 == mantissa(f1)=mantissa(f2) and exponent(f1)=exponent(f2)

    f1 + f2 ==
      m1 : I := mantissa f1
      m2 : I := mantissa f2
      e1 : I := exponent f1
      e2 : I := exponent f2
      e1 > e2 =>
--insignificance
        e1 > e2 + POWER + 2 =>
          zero? f1 => f2
          f1
          normalise [m1*(B @ I)**((e1-e2) pretend NNI)+m2,e2]$Rep
        e2 > e1 + POWER +2 =>
          zero? f2 => f1
          f2
          normalise [m2*(B @ I)**((e2-e1) pretend NNI)+m1,e1]$Rep

```

```

- f1 == [- mantissa f1,exponent f1]$Rep

f1 - f2 == f1 + (-f2)

f1 < f2 ==
  m1 : I := mantissa f1
  m2 : I := mantissa f2
  e1 : I := exponent f1
  e2 : I := exponent f2
  sign(m1) = sign(m2) =>
    e1 < e2 => true
    e1 = e2 and m1 < m2 => true
    false
  sign(m1) = 1 => false
  sign(m1) = 0 and sign(m2) = -1 => false
  true

characteristic():NNI == 0

```

— MFLOAT.dotabb —

```

"MFLOAT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MFLOAT"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"MFLOAT" -> "STRING"

```

domain MINT MachineInteger

— MachineInteger.input —

```

)set break resume
)sys rm -f MachineInteger.output
)spool MachineInteger.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MachineInteger
--R
--R MachineInteger  is a domain constructor

```

```

--R Abbreviation for MachineInteger is MINT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for MINT
--R
--R----- Operations -----
--R ?? : (%,% ) -> %
--R ?? : (PositiveInteger,% ) -> %
--R ?+? : (%,% ) -> %
--R -? : % -> %
--R ?<=? : (%,% ) -> Boolean
--R ?>? : (%,% ) -> Boolean
--R D : (% ,NonNegativeInteger) -> %
--R 1 : () -> %
--R ?? : (% ,PositiveInteger) -> %
--R addmod : (% ,% ,% ) -> %
--R base : () -> %
--R bit? : (% ,% ) -> Boolean
--R coerce : % -> %
--R coerce : % -> OutputForm
--R convert : % -> InputForm
--R convert : % -> Float
--R copy : % -> %
--R differentiate : % -> %
--R factor : % -> Factored(%)
--R gcd : (% ,% ) -> %
--R hash : % -> %
--R inc : % -> %
--R invmod : (% ,% ) -> %
--R lcm : (% ,% ) -> %
--R length : % -> %
--R max : (% ,% ) -> %
--R min : (% ,% ) -> %
--R negative? : % -> Boolean
--R one? : % -> Boolean
--R positive? : % -> Boolean
--R powmod : (% ,% ,% ) -> %
--R ?quo? : (% ,% ) -> %
--R random : % -> %
--R rational? : % -> Boolean
--R ?rem? : (% ,% ) -> %
--R sample : () -> %
--R sign : % -> Integer
--R squareFree : % -> Factored(%)
--R submod : (% ,% ,% ) -> %
--R unit? : % -> Boolean
--R zero? : % -> Boolean
--R ?? : (NonNegativeInteger,% ) -> %
--R ??? : (% ,NonNegativeInteger) -> %
--R ?? : (% ,NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger

?? : (Integer,% ) -> %
??? : (% ,PositiveInteger) -> %
?-? : (% ,% ) -> %
?<? : (% ,% ) -> Boolean
?=? : (% ,% ) -> Boolean
?>=? : (% ,% ) -> Boolean
D : % -> %
0 : () -> %
abs : % -> %
associates? : (% ,% ) -> Boolean
binomial : (% ,% ) -> %
coerce : Integer -> %
coerce : Integer -> %
convert : % -> Integer
convert : % -> Pattern(Integer)
convert : % -> DoubleFloat
dec : % -> %
even? : % -> Boolean
factorial : % -> %
gcd : List(%) -> %
hash : % -> SingleInteger
init : () -> %
latex : % -> String
lcm : List(%) -> %
mask : % -> %
maxint : () -> PositiveInteger
mulmod : (% ,% ,% ) -> %
odd? : % -> Boolean
permutation : (% ,% ) -> %
positiveRemainder : (% ,% ) -> %
prime? : % -> Boolean
random : () -> %
rational : % -> Fraction(Integer)
recip : % -> Union(%, "failed")
retract : % -> Integer
shift : (% ,% ) -> %
sizeLess? : (% ,% ) -> Boolean
squareFreePart : % -> %
symmetricRemainder : (% ,% ) -> %
unitCanonical : % -> %
?~=? : (% ,% ) -> Boolean

```

```

--R coerce : Expression(Integer) -> Expression(%)
--R differentiate : (%,NonNegativeInteger) -> %
--R divide : (%,%) -> Record(quotient: %,remainder: %)
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(%),%) -> Union(List(%),"failed")
--R exquo : (%,%) -> Union(%,"failed")
--R extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %),"failed")
--R extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %)
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R maxint : PositiveInteger -> PositiveInteger
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed")
--R nextItem : % -> Union(%,"failed")
--R patternMatch : (%,Pattern(Integer),PatternMatchResult(Integer,%)) -> PatternMatchResult(Integer,%)
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %)
--R rationalIfCan : % -> Union(Fraction(Integer),"failed")
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer))
--R reducedSystem : Matrix(%) -> Matrix(Integer)
--R retractIfCan : % -> Union(Integer,"failed")
--R subtractIfCan : (%,%) -> Union(%,"failed")
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R
--E 1

)spool
)lisp (bye)

```

— MachineInteger.help —

=====

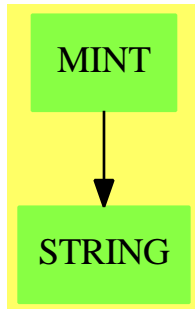
MachineInteger examples

=====

A domain which models the integer representation used by machines in the AXIOM-NAG link.

See Also:

o)show MachineInteger

MachineInteger (MINT)**See**⇒ “MachineFloat” (MFLOAT) [14](#) on page [1547](#)⇒ “MachineComplex” (MCMLPX) [14](#) on page [1542](#)**Exports:**

0	1	abs	addmod	associates?
base	binomial	bit?	characteristic	coerce
convert	copy	D	dec	differentiate
divide	euclideanSize	even?	expressIdealMember	exquo
extendedEuclidean	factor	factorial	gcd	gcdPolynomial
hash	inc	init	invmod	latex
lcm	length	mask	max	maxint
min	mulmod	multiEuclidean	negative?	nextItem
odd?	one?	patternMatch	permutation	positive?
positiveRemainder	powmod	prime?	principalIdeal	random
rational	rationalIfCan	rational?	recip	reducedSystem
retract	retractIfCan	sample	shift	sign
sizeLess?	squareFree	squareFreePart	submod	subtractIfCan
symmetricRemainder	unit?	unitCanonical	unitNormal	zero?
?~=?	?*?	?**?	?^?	?+?
?-?	-?	?<?	?<=?	?=?
?>?	?>=?	?quo?	?rem?	

— domain MINT MachineInteger —

```

)abbrev domain MINT MachineInteger
++ Author: Mike Dewar
++ Date Created:  December 1993
++ Description:
++ A domain which models the integer representation
++ used by machines in the AXIOM-NAG link.

```

MachineInteger(): Exports == Implementation where

S ==> String


```

Exports ==> Join(FortranMachineTypeCategory,IntegerNumberSystem) with
  maxint : PositiveInteger -> PositiveInteger
  ++ maxint(u) sets the maximum integer in the model to u
  maxint : () -> PositiveInteger
  ++ maxint() returns the maximum integer in the model
  coerce : Expression Integer -> Expression $
  ++ coerce(x) returns x with coefficients in the domain

Implementation ==> Integer add

MAXINT : PositiveInteger := 2**32

maxint():PositiveInteger == MAXINT

maxint(new:PositiveInteger):PositiveInteger ==
  old := MAXINT
  MAXINT := new
  old

coerce(u:Expression Integer):Expression($) ==
  map(coerce,u)$ExpressionFunctions2(Integer,$)

coerce(u:Integer):$ ==
  import S
  abs(u) > MAXINT =>
    message: S := concat [convert(u)@S," > MAXINT(",convert(MAXINT)@S,")"]
    error message
  u pretend $

retract(u:$):Integer == u pretend Integer

retractIfCan(u:$):Union(Integer,"failed") == u pretend Integer

```

— MINT.dotabb —

```

"MINT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MINT"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"MINT" -> "STRING"

```

--E 5

--S 6 of 22

a:tree := x*x

--R

--R

--R (6) [x,x]

--R

Type: Magma(Symbol)

--E 6

--S 7 of 22

b:tree := y*y

--R

--R

--R (7) [y,y]

--R

Type: Magma(Symbol)

--E 7

--S 8 of 22

c:tree := a*b

--R

--R

--R (8) [[x,x],[y,y]]

--R

Type: Magma(Symbol)

--E 8

--S 9 of 22

left c

--R

--R

--R (9) [x,x]

--R

Type: Magma(Symbol)

--E 9

--S 10 of 22

right c

--R

--R

--R (10) [y,y]

--R

Type: Magma(Symbol)

--E 10

--S 11 of 22

length c

--R

--R

--R (11) 4

--R

Type: PositiveInteger

--E 11

```
--S 12 of 22
c::word
--R
--R
--R      2 2
--R (12)  x y
--R
--R                                          Type: OrderedFreeMonoid(Symbol)
--E 12

--S 13 of 22
a < b
--R
--R
--R (13)  true
--R
--R                                          Type: Boolean
--E 13

--S 14 of 22
a < c
--R
--R
--R (14)  true
--R
--R                                          Type: Boolean
--E 14

--S 15 of 22
b < c
--R
--R
--R (15)  true
--R
--R                                          Type: Boolean
--E 15

--S 16 of 22
first c
--R
--R
--R (16)  x
--R
--R                                          Type: Symbol
--E 16

--S 17 of 22
rest c
--R
--R
--R (17)  [x,[y,y]]
--R
--R                                          Type: Magma(Symbol)
--E 17

--S 18 of 22
```

```

rest rest c
--R
--R
--R (18) [y,y]
--R
--R                                          Type: Magma(Symbol)
--E 18

--S 19 of 22
ax:tree := a*x
--R
--R
--R (19) [[x,x],x]
--R
--R                                          Type: Magma(Symbol)
--E 19

--S 20 of 22
xa:tree := x*a
--R
--R
--R (20) [x,[x,x]]
--R
--R                                          Type: Magma(Symbol)
--E 20

--S 21 of 22
xa < ax
--R
--R
--R (21) true
--R
--R                                          Type: Boolean
--E 21

--S 22 of 22
lexico(xa,ax)
--R
--R
--R (22) false
--R
--R                                          Type: Boolean
--E 22
)spool
)lisp (bye)

```

— Magma.help —

```

=====
Magma examples
=====

```

Initialisations

```
x:Symbol := 'x
  x
                                Type: Symbol
```

```
y:Symbol := 'y
  y
                                Type: Symbol
```

```
z:Symbol := 'z
  z
                                Type: Symbol
```

```
word := OrderedFreeMonoid(Symbol)
      OrderedFreeMonoid Symbol
                                Type: Domain
```

```
tree := Magma(Symbol)
      Magma Symbol
                                Type: Domain
```

Let's make some trees

```
a:tree := x*x
  [x,x]
                                Type: Magma Symbol
```

```
b:tree := y*y
  [y,y]
                                Type: Magma Symbol
```

```
c:tree := a*b
  [[x,x],[y,y]]
                                Type: Magma Symbol
```

Query the trees

```
left c
  [x,x]
                                Type: Magma Symbol
```

```
right c
  [y,y]
                                Type: Magma Symbol
```

```
length c
  4
                                Type: PositiveInteger
```

Coerce to the monoid

```
c::word
  2 2
  x y
```

Type: OrderedFreeMonoid Symbol

Check ordering

```
a < b
true
```

Type: Boolean

```
a < c
true
```

Type: Boolean

```
b < c
true
```

Type: Boolean

Navigate the tree

```
first c
  x
```

Type: Symbol

```
rest c
  [x,[y,y]]
```

Type: Magma Symbol

```
rest rest c
  [y,y]
```

Type: Magma Symbol

Check ordering

```
ax:tree := a*x
  [[x,x],x]
```

Type: Magma Symbol

```
xa:tree := x*a
  [x,[x,x]]
```

Type: Magma Symbol

```
xa < ax
true
```

Type: Boolean

```
lexico(xa,ax)
```

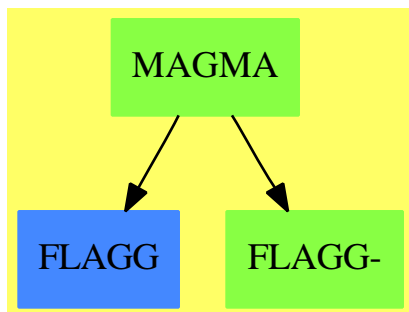
false

Type: Boolean

See Also:

o)show Magma

Magma (MAGMA)



Exports:

coerce	first	hash	latex	left
length	lexico	max	min	mirror
rest	retract	retractIfCan	retractable?	right
varList	?~=?	?*?	?<?	?<=?
?=?	?>?	?>=?		

— domain MAGMA Magma —

```

)abbrev domain MAGMA Magma
++ Author: Michel Petitot (petitot@lifl.fr).
++ Date Created: 91
++ Date Last Updated: 7 Juillet 92
++ Fix History: compilation v 2.1 le 13 dec 98
++ Description:
++ This type is the basic representation of
++ parenthesized words (binary trees over arbitrary symbols)
++ useful in \spadtype{LiePolynomial}.

```

```

Magma(VarSet:OrderedSet):Public == Private where
  WORD ==> OrderedFreeMonoid(VarSet)
  EX   ==> OutputForm

```

```

Public == Join(OrderedSet,RetractableTo VarSet) with

```



```

"*"      : ($,$) -> $
  ++ \axiom{x*y} returns the tree \axiom{[x,y]}.
coerce   : $ -> WORD
  ++ \axiom{coerce(x)} returns the element of
  ++ \axiomType{OrderedFreeMonoid}(VarSet)
  ++ corresponding to \axiom{x} by removing parentheses.
first    : $ -> VarSet
  ++ \axiom{first(x)} returns the first entry of the tree \axiom{x}.
left     : $ -> $
  ++ \axiom{left(x)} returns left subtree of \axiom{x} or
  ++ error if retractable?(x) is true.
length   : $ -> PositiveInteger
  ++ \axiom{length(x)} returns the number of entries in \axiom{x}.
lexico   : ($,$) -> Boolean
  ++ \axiom{lexico(x,y)} returns \axiom{true} iff \axiom{x} is smaller
  ++ than \axiom{y} w.r.t. the lexicographical ordering induced by
  ++ \axiom{VarSet}.
  ++ N.B. This operation does not take into account the tree structure of
  ++ its arguments. Thus this is not a total ordering.
mirror   : $ -> $
  ++ \axiom{mirror(x)} returns the reversed word of \axiom{x}.
  ++ That is \axiom{x} itself if retractable?(x) is true and
  ++ \axiom{mirror(z) * mirror(y)} if \axiom{x} is \axiom{y*z}.
rest     : $ -> $
  ++ \axiom{rest(x)} return \axiom{x} without the first entry or
  ++ error if retractable?(x) is true.
retractable? : $ -> Boolean
  ++ \axiom{retractable?(x)} tests if \axiom{x} is a tree with
  ++ only one entry.
right    : $ -> $
  ++ \axiom{right(x)} returns right subtree of \axiom{x} or
  ++ error if retractable?(x) is true.
varList  : $ -> List VarSet
  ++ \axiom{varList(x)} returns the list of distinct entries of \axiom{x}.

Private == add
-- representation
VWORD := Record(left:$ ,right:$)
Rep:= Union(VarSet,VWORD)

recursif: ($,$) -> Boolean

-- define
x:$ = y:$ ==
  x case VarSet =>
    y case VarSet => x::VarSet = y::VarSet
    false
  y case VWORD => x::VWORD = y::VWORD
  false

```

```

varList x ==
  x case VarSet => [x::VarSet]
  lv: List VarSet := setUnion(varList x.left, varList x.right)
  sort_!(lv)

left x ==
  x case VarSet => error "x has only one entry"
  x.left

right x ==
  x case VarSet => error "x has only one entry"
  x.right
retractable? x == (x case VarSet)

retract x ==
  x case VarSet => x::VarSet
  error "Not retractable"

retractIfCan x == (retractable? x => x::VarSet ; "failed")
coerce(l:VarSet):$ == l

mirror x ==
  x case VarSet => x
  [mirror x.right, mirror x.left]$VWORD

coerce(x:$): WORD ==
  x case VarSet => x::VarSet::WORD
  x.left::WORD * x.right::WORD

coerce(x:$):EX ==
  x case VarSet => x::VarSet::EX
  bracket [x.left::EX, x.right::EX]

x * y == [x,y]$VWORD

first x ==
  x case VarSet => x::VarSet
  first x.left

rest x ==
  x case VarSet => error "rest$Magma: inexistant rest"
  lx:$ := x.left
  lx case VarSet => x.right
  [rest lx , x.right]$VWORD

length x ==
  x case VarSet => 1
  length(x.left) + length(x.right)

recursif(x,y) ==

```

```

x case VarSet =>
  y case VarSet => x::VarSet < y::VarSet
    true
  y case VarSet => false
x.left = y.left => x.right < y.right
x.left < y.left

lexico(x,y) ==      -- peut etre amelioree !!!!!!!!!!!
x case VarSet =>
  y case VarSet => x::VarSet < y::VarSet
    x::VarSet <= first y
  y case VarSet => first x < retract y
fx:VarSet := first x ; fy:VarSet := first y
fx = fy => lexico(rest x , rest y)
fx < fy

x < y ==      -- recursif par longueur
lx,ly: PositiveInteger
lx:= length x ; ly:= length y
lx = ly => recursif(x,y)
lx < ly

```

— MAGMA.dotabb —

```

"MAGMA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MAGMA"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"MAGMA" -> "FLAGG"
"MAGMA" -> "FLAGG-"

```

domain MKCHSET MakeCachableSet

— MakeCachableSet.input —

```

)set break resume
)sys rm -f MakeCachableSet.output
)spool MakeCachableSet.output
)set message test on
)set message auto off
)clear all

```

```

--S 1 of 1
)show MakeCachableSet
--R
--R MakeCachableSet(S: SetCategory) is a domain constructor
--R Abbreviation for MakeCachableSet is MKCHSET
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for MKCHSET
--R
--R----- Operations -----
--R ?<? : (%,% ) -> Boolean          ?<=? : (%,% ) -> Boolean
--R ?=? : (%,% ) -> Boolean          ?>? : (%,% ) -> Boolean
--R ?>=? : (%,% ) -> Boolean          coerce : S -> %
--R coerce : % -> S                  coerce : % -> OutputForm
--R hash : % -> SingleInteger         latex : % -> String
--R max : (%,% ) -> %                 min : (%,% ) -> %
--R ?~=? : (%,% ) -> Boolean
--R position : % -> NonNegativeInteger
--R setPosition : (% ,NonNegativeInteger) -> Void
--R
--E 1

```

```

)spool
)lisp (bye)

```

— MakeCachableSet.help —

```

=====
MakeCachableSet examples
=====

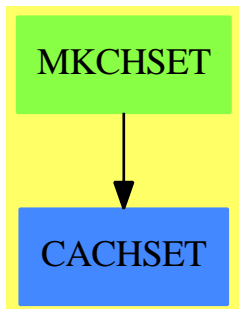
```

MakeCachableSet(S) returns a cachable set which is equal to S as a set.

See Also:

o)show MakeCachableSet

MakeCachableSet (MKCHSET)



See

⇒ “Kernel” (KERNEL) [12](#) on page [1404](#)

Exports:

coerce	hash	latex	max	min	position
setPosition	?~=?	?<?	?<=?	?=?	?>?
?>=?					

— domain MKCHSET MakeCachableSet —

```

)abbrev domain MKCHSET MakeCachableSet
++ Author: Manuel Bronstein
++ Date Created: ???
++ Date Last Updated: 14 May 1991
++ Description:
++ MakeCachableSet(S) returns a cachable set which is equal to S as a set.

```

```

MakeCachableSet(S:SetCategory): Exports == Implementation where
  Exports ==> Join(CachableSet, CoercibleTo S) with
    coerce: S -> %
      ++ coerce(s) returns s viewed as an element of %.

```

```

Implementation ==> add
  import SortedCache(%)

```

```

Rep := Record(setpart: S, pos: NonNegativeInteger)

```

```

clearCache()

```

```

position x                == x.pos
setPosition(x, n)         == (x.pos := n; void)
coerce(x:%):S             == x.setpart
coerce(x:%):OutputForm    == x::S::OutputForm
coerce(s:S):%             == enterInCache([s, 0]$Rep, x+-->(s = x::S))

```

```

x < y ==

```

```

if position(x) = 0 then enterInCache(x, x1+-->(x::S = x1::S))
if position(y) = 0 then enterInCache(y, x1+-->(y::S = x1::S))
position(x) < position(y)

x = y ==
if position(x) = 0 then enterInCache(x, x1+-->(x::S = x1::S))
if position(y) = 0 then enterInCache(y, x1+-->(y::S = x1::S))
position(x) = position(y)

```

— MKCHSET.dotabb —

```

"MKCHSET" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MKCHSET"]
"CACHSET" [color="#4488FF",href="bookvol10.2.pdf#nameddest=CACHSET"]
"MKCHSET" -> "CACHSET"

```

domain MMLFORM MathMLFormat

— MathMLFormat.input —

```

)set break resume
)sys rm -f MathMLFormat.output
)spool MathMLFormat.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MathMLFormat
--R MathMLFormat is a domain constructor
--R Abbreviation for MathMLFormat is MMLFORM
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for MMLFORM
--R
--R----- Operations -----
--R ?=? : (%,%) -> Boolean          coerce : OutputForm -> String
--R coerce : % -> OutputForm        coerceL : OutputForm -> String
--R coerceS : OutputForm -> String  display : String -> Void
--R exprex : OutputForm -> String   hash : % -> SingleInteger
--R latex : % -> String             ?~=? : (%,%) -> Boolean
--R

```

```
--E 1

)spool
)lisp (bye)
```

— MathMLFormat.help —

```
=====
MathMLFormat examples
=====
```

This package is based on the TeXFormat domain by Robert S. Sutor
 MathMLFormat provides a coercion from OutputForm to MathML format.

See Also:

o)show MathMLFormat

Both this code and documentation are still under development and I don't pretend they are anywhere close to perfect or even finished. However the code does work and I hope it might be useful to somebody both for it's ability to output MathML from Axiom and as an example of how to write a new output form.

Introduction to Mathematical Markup Language

MathML exists in two forms: presentation and content. At this time (2007-02-11) the package only has a presentation package. A content package is in the works however it is more difficult. Unfortunately Axiom does not make its semantics easily available. The `OutputForm` domain mediates between the individual Axiom domains and the user visible output but `OutputForm` does not provide full semantic information. From my currently incomplete understanding of Axiom it appears that remedying this would entail going back to the individual domains and rewriting a lot of code. However some semantics are conveyed directly by `OutputForm` and other things can be deduced from `OutputForm` or from the original user command.

Displaying MathML

The MathML string produced by `)set output mathml on` can be pasted directly into an appropriate xhtml page and then viewed in Firefox or some other MathML aware browser. The boiler plate code needed for a test page, `testmathml.xml`, is:

```
<?xml version="1.0" ?>
```

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1 plus MathML 2.0//EN"
    "http://www.w3.org/Math/DTD/mathml2/xhtml-math11-f.dtd" [
<!ENTITY mathml "http://www.w3.org/1998/Math/MathML">
]>

<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:xlink="http://www.w3.org/1999/xlink" >

    <head>
        <title>MathML Test </title>
    </head>

    <body>

    </body>
</html>

```

Paste the MathML string into the body element and it should display nicely in Firefox.

Test Cases

Here's a list of test cases that currently format correctly:

1. $(x+y)^{**2}$
2. $\text{integrate}(x^{**x}, x)$
3. $\text{integral}(x^{**x}, x)$
4. $(5 + \sqrt{63} + \sqrt{847})^{**}(1/3)$
5. $\text{set } [1, 2, 3]$
6. $\text{multiset } [x \bmod 5 \text{ for } x \text{ in primes}(2, 1000)]$
7. $\text{series}(\sin(a \cdot x), x=0)$
8. $\text{matrix } [[x^{**i} + y^{**j} \text{ for } i \text{ in } 1..10] \text{ for } j \text{ in } 1..10]$
9. $y := \text{operator } 'y \text{ a. } D(y(x, z), [x, x, z, x]) \text{ b. } D(y \ x, x, 2)$
10. $x := \text{series } 'x \text{ a. } \sin(1+x)$
11. $\text{series}(1/\log(y), y=1)$
12. $y:\text{UTS}(\text{FLOAT}, 'z, 0) := \exp(z)$
13. a. $c := \text{continuedFraction}(314159/100000)$ b. $c := \text{continuedFraction}(314159/100000)$

The `TexFormat` domain has the capability to format an object with subscripts, superscripts, presubscripts and presuperscripts however I don't know of any Axiom command that produces such an object. In fact at present I see the case of "SUPERSUB" being used for putting primes in the superscript position to denote ordinary differentiation. I also only see the "SUB" case being used to denote partial derivatives.

)set output mathml on

Making mathml appear as output during a normal Axiom session by invoking ")set output mathml on" proved to be a bit tedious and seems to be undocumented. I document my experience here in case it proves useful to somebody else trying to get a new output format from Axiom.

In MathMLFormat the functions **coerce(expr : OutputForm) : String** and **display(s : String) : Void** provide the desired mathml output. Note that this package was constructed by close examination of Robert Sutor's **TexFormat** domain and much remains from that source. To have mathml displayed as output we need to get Axiom to call **display(coerce(expr))** at the appropriate place. Here's what I did to get that to happen. Note that my starting point here was an attempt by Andrey Grozin to do the same. To figure things out I searched through files for "tex" to see what was done for the **TexFormat** domain, and used **grep** to find which files had mention of **TexFormat**.

File src/interp/setvars.boot.pamphlet

Create an output mathml section by analogy to the tex section. Remember to add the code chunk "outputmathmlCode" at the end.

setvars.boot is a bootstrap file which means that it has to be precompiled into lisp code and then that code has to be inserted back into setvars.boot. To do this extract the boot code by running "notangle" on it. I did this from the "tmp" directory. From inside axiom run ")lisp (boottran::boottool "tmp/setvars.boot")" which put "setvars.clisp" into "int/interp/setvars.clisp". Then replace the lisp in "setvars.boot.pamphlet" with that in the newly generated "setvars.clisp".

The relevant code chunks appearing in "setvars.boot.pamphlet" are:

```
outputmathmlCode
setOutputMathml
describeSetOutputMathml
```

and the relevant variables are:

```
setOutputMathml
$mathmlOutputStream
$mathmlOutputFile
$mathmlFormat
describeSetOutputMathml
```

File setvart.boot.pamphlet

Create an output mathml section in "setvart.boot.pamphlet" again patterned after the tex section. I changed the default file extension from ".stex" to ".smml".

To the "sectionoutput" table I added the line

```
mathml                                created output in MathML style      Off:CONSOLE
```

Added the code chunk "outputmathml" to the code chunk "output" in "sectionoutput".

Relevant code chunks:

```
outputmathml
```

Relevant variables:

```
setOutputMathml
$mathmlFormat
$mathmlOutputFile
```

Note when copying the tex stuff I changed occurrences of "tex" to "mathml", "Tex" to "Mathml" and "TeX" to "MathML".

File src/algebra/Makefile.pamphlet

The file "src/algebra/tex.spad.pamphlet" contains the domain `TexFormat` (TEX) and the package `TexFormat1` (TEX1). However the sole function of `TexFormat1` is to **coerce** objects from a domain into `OutputForm` and then apply `TexFormat` to them. It is to save programmers the trouble of doing the coercion themselves from inside spad code. It does not appear to be used for the main purpose of delivering Axiom output in TeX format. In order to keep the mathml package as simple as possible, and because I didn't see much use for this, I didn't copy the `TexFormat1` package. So no analog of the TEX1 entries in "Makefile.pamphlet" were needed. One curiosity I don't understand is why TEX1 appears in layer 4 when it seems to depend on TEX which appears in layer 14.

Initially I added "\$OUT/MMLFORM.o" to layer 14 and "mathml.spad.pamphlet" to completed spad files in layer 14. When trying to compile the build failed at MMLFORM. It left "MMLFORM.erlib" in "int/algebra" instead of "MMLFORM.NRLIB" which confused me at first because mathml.spad compiled under a running axiom. By examining the file "obj/tmp/trace" I saw that a new dependency had been introduced, compared to `TexFormat`, with the function `eltName` depending on the domain `FSAGG` in layer 16. So the lines had to be moved from layer 14 to layer 17.

Added appropriate lines to "SPADFILES" and "DOCFILES".

File src/algebra/exposed.lsp.pamphlet

Add the line "(|MathMLFormat| . MMLFORM)"

File src/algebra/Lattice.pamphlet

I don't see that this file is used anywhere but I made the appropriate changes anyway by searching for "TEX" and mimicing everything for MMLFORM.

File `src/doc/axiom.bib.pamphlet`

Added `mathml.spad` subsection to `"src/doc/axiom.bib.pamphlet"`.

File `interp/i-output.boot.pamphlet`

This is where the **coerce** and **display** functions from `MathMLFormat` actually get called. The following was added:

```
mathmlFormat expr ==
  mml := '(MathMLFormat)
  mmlrep := '(String)
  formatFn := getFunctionFromDomain("coerce",mml,[$OutputForm])
  displayFn := getFunctionFromDomain("display",mml,[mmlrep])
  SPADCALL(SPADCALL(expr,formatFn),displayFn)
  TERPRI $mathmlOutputStream
  FORCE_-OUTPUT $mathmlOutputStream
  NIL
```

Note that compared to the `texFormat` function there are a couple of differences. Since `MathMLFormat` is currently a package rather than a domain there is the `"mmlrep"` variable whereas in `texFormat` the argument of the `"display"` function is an instance of the domain. Also the **coerce** function here only has one argument, namely `"$OutputForm"`.

Also for the function `"output(expr,domain)"` add lines for `mathml`, e.g. `"if $mathmlFormat then mathmlFormat expr"`.

After these changes Axiom compiled with `mathml` enabled under `)set output`.

Public Declarations

The declarations

```
E      ==> OutputForm
I      ==> Integer
L      ==> List
S      ==> String
US     ==> UniversalSegment(Integer)
```

provide abbreviations for domains used heavily in the code. The publicly exposed functions are:

coerce: E -> S This function is the main one for converting an expression in domain `OutputForm` into a MathML string.

coerceS: E -> S This function is for use from the command line. It converts an `OutputForm` expression into a MathML string and does some formatting so that the output is not one long line. If you take the output from this function, stick it in an emacs buffer in `nxml-mode`

and then indent according to mode, you'll get something that's nicer to look at than what comes from `coerce`. Note that `coerceS` returns the same value as `coerce` but invokes a display function as well so that the result will be printed twice in different formats. The need for this is that the output from `coerce` is automatically formatted with line breaks by Axiom's output routine that are not in the right place.

coerceL: E -> S Similar to `coerceS` except that the displayed result is the MathML string in one long line. These functions can be used, for instance, to get the MathML for the previous result by typing `coerceL(`

expres: E -> S Converts `OutputForm` to `String` with the structure preserved with braces. This is useful in developing this package. Actually this is not quite accurate. The function **precondition** is first applied to the `OutputForm` expression before **expres**. Raw `OutputForm` and the nature of the **precondition** function is still obscure to me at the time of this writing (2007-02-14), however I probably need to understand it to make sure I'm not missing any semantics. The `spad` function **precondition** is just a wrapper for the lisp function `outputTran$Lisp`, which I guess is compiled from `boot`.

display: S -> Void This one prints the string returned by `coerce` as one long line, adding "math" tags: `$...$`. Thus the output from this can be stuck directly into an appropriate `html/xhtml` page and will be displayed nicely by a MathML aware browser.

displayF: S -> Void This function doesn't exist yet but it would be nice to have a humanly readable formatted output as well. The basics do exist in the `coerceS` function however the formatting still needs some work to be really good.

— public declarations —

```
)abbrev domain MMLFORM MathMLFormat
++ Author: Arthur C. Ralfs
++ Date: January 2007
++ Description:
++ This package is based on the TeXFormat domain by Robert S. Sutor
++ \spadtype{MathMLFormat} provides a coercion from \spadtype{OutputForm}
++ to MathML format.
```

```
MathMLFormat(): public == private where
```

```
  E      ==> OutputForm
  I      ==> Integer
  L      ==> List
  S      ==> String
  US     ==> UniversalSegment(Integer)
```

```
public == SetCategory with
```

```
  coerce:   E -> S
    ++ coerceS(o) changes o in the standard output format to MathML
    ++ format.
  coerceS:  E -> S
    ++ coerceS(o) changes o in the standard output format to MathML
    ++ format and displays formatted result.
```

```

coerceL:  E -> S
  ++ coerceS(o) changes o in the standard output format to MathML
  ++ format and displays result as one long string.
expres:   E -> S
  ++ converts \spadtype{OutputForm} to \spadtype{String} with the
  ++ structure preserved with braces.  Actually this is not quite
  ++ accurate.  The function \spadfun{precondition} is first
  ++ applied to the
  ++ \spadtype{OutputForm} expression before \spadfun{expres}.
  ++ The raw \spadtype{OutputForm} and
  ++ the nature of the \spadfun{precondition} function is
  ++ still obscure to me
  ++ at the time of this writing (2007-02-14).
display:  S -> Void
  ++ prints the string returned by coerce, adding <math ...> tags.

```

Private Constant Declarations

— private constant declarations —

```

private == add
  import OutputForm
  import Character
  import Integer
  import List OutputForm
  import List String

-- local variable declarations and definitions

expr: E
prec,opPrec: I
str: S
blank      : S := " \ "

maxPrec    : I := 1000000
minPrec    : I := 0

unaryOps    : L S := ["-","^"]$(L S)
unaryPrecs  : L I := [700,260]$(L I)

-- the precedence of / in the following is relatively low because
-- the bar obviates the need for parentheses.
binaryOps   : L S := ["+>","|","**","/","<",">","=","OVER"]$(L S)
binaryPrecs : L I := [0,0,900, 700,400,400,400, 700]$(L I)

```

```

naryOps      : L S := ["-", "+", "*", blank, ",", ";", " ", "ROW", "",
  " \cr ", "&", "</mtd></mtr><mtr><mtd>"]$(L S)
naryPrecs    : L I := [700, 700, 800, 800, 110, 110, 0, 0, 0, 0, 0]$(L I)
naryNGOps    : L S := ["ROW", "&"]$(L S)

plexOps      : L S := ["SIGMA", "SIGMA2", "PI", "PI2", "INTSIGN", "INDEFINTEGRAL"]$(L S)
plexPrecs    : L I := [700, 800, 700, 800, 700, 700]$(L I)

specialOps   : L S := ["MATRIX", "BRACKET", "BRACE", "CONCATB", "VCONCAT", _
  "AGGLST", "CONCAT", "OVERBAR", "ROOT", "SUB", "TAG", _
  "SUPERSUB", "ZAG", "AGGSET", "SC", "PAREN", _
  "SEGMENT", "QUOTE", "theMap", "SLASH" ]

-- the next two lists provide translations for some strings for
-- which MML provides special macros.

specialStrings : L S :=
  ["cos", "cot", "csc", "log", "sec", "sin", "tan",
   "cosh", "coth", "csch", "sech", "sinh", "tanh",
   "acos", "asin", "atan", "erf", "...", "$", "infinity"]
specialStringsInMML : L S :=
  ["<mo>cos</mo>", "<mo>cot</mo>", "<mo>csc</mo>", "<mo>log</mo>", "<mo>sec</mo>", "<mo>sin</mo>", "<mo>tan</mo>",
   "<mo>cosh</mo>", "<mo>coth</mo>", "<mo>csch</mo>", "<mo>sech</mo>", "<mo>sinh</mo>", "<mo>tanh</mo>",
   "<mo>arccos</mo>", "<mo>arcsin</mo>", "<mo>arctan</mo>", "<mo>erf</mo>", "<mo>&#x2026;</mo>", "<mo>$</mo>"]

```

Private Function Declarations

These are the local functions:

addBraces:S -> S

addBrackets:S -> S

atomize:E -> L E

displayElt:S -> Void function for recursively displaying mathml nicely formatted

eltLimit:(S,I,S) -> I demarcates end postion of mathml element with name:S starting at position i:I in mathml string s:S and returns end of end tag as i:I position in mathml string, i.e. find start and end of substring: <name ...>...</name>

eltName:(I,S) -> S find name of mathml element starting at position i:I in string s:S

group:S -> S

formatBinary:(S,L E, I) -> S

formatFunction:(S,L E, I) -> S

formatMatrix:L E -> S

```

formatNary:(S,L E, I) -> S
formatNaryNoGroup:(S,L E, I) -> S
formatNullary:S -> S
formatPlex:(S,L E, I) -> S
formatSpecial:(S,L E, I) -> S
formatUnary:(S, E, I) -> S
formatMml:(E,I) -> S
newWithNum:I -> $ this is a relic from tex.spad and is not used here so far. I'll probably
remove it.
parenthesize:S -> S
precondition:E -> E this function is applied to the OutputForm expression before doing
anything else.
postcondition:S -> S this function is applied after all other OutputForm -> MathML trans-
formations. In the TexFormat domain the ungroup function first peels off the outermost set
of braces however I have replaced braces with <mrow>s here and sometimes the outermost
set of <mrow>s is necessary to get proper display in Firefox. For instance with getting the
correct size of brackets on a matrix the whole expression needs to be enclosed in a mrow
element. It also checks for +- and removes the +.
stringify:E -> S
tagEnd:(S,I,S) -> I finds closing ">" of start or end tag for mathML element for formatting
MathML string for human readability. No analog in TexFormat.
ungroup:S -> S

```

— private function declarations —

```

-- local function signatures

addBraces:      S -> S
addBrackets:    S -> S
atomize:        E -> L E
displayElt:     S -> Void
  ++ function for recursively displaying mathml nicely formatted
eltLimit:       (S,I,S) -> I
  ++ demarcates end postion of mathml element with name:S starting at
  ++ position i:I in mathml string s:S and returns end of end tag as
  ++ i:I position in mathml string, i.e. find start and end of
  ++ substring: <name ...>...</name>
eltName:        (I,S) -> S
  ++ find name of mathml element starting at position i:I in string s:S
group:          S -> S
formatBinary:   (S,L E, I) -> S
formatFunction: (S,L E, I) -> S

```

```

formatIntSign: (L E, I) -> S
formatMatrix:  L E -> S
formatNary:    (S,L E, I) -> S
formatNaryNoGroup: (S,L E, I) -> S
formatNullary: S -> S
formatPlex:    (S,L E, I) -> S
formatSpecial: (S,L E, I) -> S
formatSub:     (E, L E, I) -> S
formatSuperSub: (E, L E, I) -> S
formatSuperSub1: (E, L E, I) -> S
formatUnary:   (S, E, I) -> S
formatMml:     (E,I) -> S
formatZag:     L E -> S
formatZag1:    L E -> S
newWithNum:    I -> $
parenthesize:  S -> S
precondition:  E -> E
postcondition: S -> S
stringify:     E -> S
tagEnd:        (S,I,S) -> I
  ++ finds closing ">" of start or end tag for mathML element
ungroup:       S -> S

```

Public Function Definitions

Note that I use the function `sayTeX$Lisp` much as I would `printf` in a C program. I've noticed in grepping the code that there are other "say" functions, `sayBrightly` and `sayMessage` for instance, but I have no idea what the difference is between them at this point. `sayTeX$Lisp` does the job so for the time being I'll use that until I learn more.

The functions `coerceS` and `coerceL` should probably be changed to display functions, *i.e.* **`displayS`** and **`display L`**, returning `Void`. I really only need the one `coerce` function.

— public function definitions —

```
-- public function definitions
```

```

coerce(expr : E): S ==
  s : S := postcondition formatMml(precondition expr, minPrec)
  s

```

```

coerceS(expr : E): S ==
  s : S := postcondition formatMml(precondition expr, minPrec)
  sayTeX$Lisp "<math xmlns='http://www.w3.org/1998/Math/MathML' mathsize='big' display='block'>"
  displayElt(s)

```



```

sayTeX$Lisp "</math>"
s

coerceL(expr : E): S ==
s : S := postcondition formatMml(precondition expr, minPrec)
sayTeX$Lisp "<math xmlns=_\"http://www.w3.org/1998/Math/MathML_\" mathsize=_\"big_\" display=_\"
sayTeX$Lisp s
sayTeX$Lisp "</math>"
s

display(mathml : S): Void ==
sayTeX$Lisp "<math xmlns=_\"http://www.w3.org/1998/Math/MathML_\" mathsize=_\"big_\" display=_\"
sayTeX$Lisp mathml
sayTeX$Lisp "</math>"
void()$Void

exprex(expr : E): S ==
-- This breaks down an expression into atoms and returns it as
-- a string. It's for developmental purposes to help understand
-- the expressions.
a : E
expr := precondition expr
--
sayTeX$Lisp "0: "stringify expr
(ATOM(expr)$Lisp@Boolean) or (stringify expr = "NOTHING") =>
concat ["{",stringify expr,"}"]
le : L E := (expr pretend L E)
op := first le
sop : S := exprex op
args : L E := rest le
nargs : I := #args
--
sayTeX$Lisp concat ["1: ",stringify first le," : ",string(nargs)$S]
s : S := concat ["{",sop]
if nargs > 0 then
for a in args repeat
--
sayTeX$Lisp concat ["2: ",stringify a]
s1 : S := exprex a
s := concat [s,s1]
s := concat [s,"}"]

```

Private Function Definitions

Display Functions

displayElt(mathml:S):Void

```
eltName(pos:I,mathml:S):S
eltLimit(name:S,pos:I,mathml:S):I
tagEnd(name:S,pos:I,mathml:S):I
```

— display functions —

```
displayElt(mathML:S): Void ==
  -- Takes a string of syntactically complete mathML
  -- and formats it for display.
  --      sayTeX$Lisp "****displayElt1****"
  --      sayTeX$Lisp mathML
  enT:I -- marks end of tag, e.g. "<name>"
  enE:I -- marks end of element, e.g. "<name> ... </name>"
  end:I -- marks end of mathML string
  u:US
  end := #mathML
  length:I := 60
  --      sayTeX$Lisp "****displayElt1.1****"
  name:S := eltName(1,mathML)
  --      sayTeX$Lisp name
  --      sayTeX$Lisp concat("****displayElt1.2****",name)
  enE := eltLimit(name,2+#name,mathML)
  --      sayTeX$Lisp "****displayElt2****"
  if enE < length then
  --      sayTeX$Lisp "****displayElt3****"
    u := segment(1,enE)$US
    sayTeX$Lisp mathML.u
  else
  --      sayTeX$Lisp "****displayElt4****"
    enT := tagEnd(name,1,mathML)
    u := segment(1,enT)$US
    sayTeX$Lisp mathML.u
    u := segment(enT+1,enE-#name-3)$US
    displayElt(mathML.u)
    u := segment(enE-#name-2,enE)$US
    sayTeX$Lisp mathML.u
  if end > enE then
  --      sayTeX$Lisp "****displayElt5****"
    u := segment(enE+1,end)$US
    displayElt(mathML.u)

  void()$Void

eltName(pos:I,mathML:S): S ==
  -- Assuming pos is the position of "<" for a start tag of a mathML
  -- element finds and returns the element's name.
  i:I := pos+1
  --sayTeX$Lisp "eltName:mathmML string: "mathML
```

```

while member?(mathML.i,lowerCase()$CharacterClass)$CharacterClass repeat
  i := i+1
u:US := segment(pos+1,i-1)
name:S := mathML.u

eltLimit(name:S,pos:I,mathML:S): I ==
-- Finds the end of a mathML element like "<name ...> ... </name>"
-- where pos is the position of the space after name in the start tag
-- although it could point to the closing ">". Returns the position
-- of the ">" in the end tag.
pI:I := pos
startI:I
endI:I
startS:S := concat ["<",name]
endS:S := concat ["</",name,">"]
level:I := 1
--sayTeX$Lisp "eltLimit: element name: "name
while (level > 0) repeat
  startI := position(startS,mathML,pI)$String

  endI := position(endS,mathML,pI)$String

  if (startI = 0) then
    level := level-1
    --sayTeX$Lisp "****eltLimit 1*****"
    pI := tagEnd(name,endI,mathML)
  else
    if (startI < endI) then
      level := level+1
      pI := tagEnd(name,startI,mathML)
    else
      level := level-1
      pI := tagEnd(name,endI,mathML)
pI

tagEnd(name:S,pos:I,mathML:S):I ==
-- Finds the closing ">" for either a start or end tag of a mathML
-- element, so the return value is the position of ">" in mathML.
pI:I := pos
while (mathML.pI ^= char ">") repeat
  pI := pI+1
u:US := segment(pos,pI)$US
--sayTeX$Lisp "tagEnd: "mathML.u
pI

```

Formatting Functions

Still need to format `\zag` in `formatSpecial`!

In `formatPlex` the case `op = "INTSIGN"` is now passed off to `formatIntSign` which is a change from the `TexFormat` domain. This is done here for presentation mark up to replace the ugly bound variable that Axiom delivers. For content mark up this has to be done anyway.

The `formatPlex` function also allows for `op = "INDEFINTEGRAL"`. However I don't know what Axiom command gives rise to this case. The `INTSIGN` case already allows for both definite and indefinite integrals.

In the function `formatSpecial` various cases are handled including `SUB` and `SUPERSUB`. These cases are now caught in `formatMml` and so the code in `formatSpecial` doesn't get executed. The only cases I know of using these are partial derivatives for `SUB` and ordinary derivatives or `SUPERSUB` however in `TexFormat` the capability is there to handle multi-scripts, i.e. an object with subscripts, superscripts, pre-subscripts and pre-superscripts but I am so far unaware of any Axiom command that produces such a multiscripted object.

Another question is how to represent derivatives. At present I have differential notation for partials and prime notation for ordinary derivatives, but it would be nice to allow for different derivative notations in different circumstances, maybe some options to `)set output mathml on`.

Ordinary derivatives are formatted in `formatSuperSub` and there are 2 versions, `formatSuperSub` and `formatSuperSub1`, which at this point have to be switched by swapping names.

— formatting functions —

```
atomize(expr : E): L E ==
-- This breaks down an expression into a flat list of atomic expressions.
-- expr should be preconditioned.
le : L E := nil()
a : E
letmp : L E
(ATOM(expr)$Lisp@Boolean) or (stringify expr = "NOTHING") =>
  le := append(le,list(expr))
letmp := expr pretend L E
for a in letmp repeat
  le := append(le,atomize a)
le

ungroup(str: S): S ==
len : I := #str
len < 14 => str
lrow : S := "<mrow>"
rrow : S := "</mrow>"
-- drop leading and trailing mrows
u1 : US := segment(1,6)$US
u2 : US := segment(len-6,len)$US
if (str.u1 =$S lrow) and (str.u2 =$S rrow) then
```

```

    u : US := segment(7,len-7)$US
    str := str.u
    str

postcondition(str: S): S ==
--   str := ungroup str
   len : I := #str
   plusminus : S := "<mo>+</mo><mo>-</mo>"
   pos : I := position(plusminus,str,1)
   if pos > 0 then
       ustart:US := segment(1,pos-1)$US
       uend:US := segment(pos+20,len)$US
       str := concat [str.ustrart,"<mo>-</mo>",str.uend]
       if pos < len-18 then
           str := postcondition(str)
   str

stringify expr == (mathObject2String$Lisp expr)@S

group str ==
   concat ["<mrow>",str,"</mrow>"]

addBraces str ==
   concat ["<mo>{</mo>",str,"<mo>}</mo>"]

addBrackets str ==
   concat ["<mo>[</mo>",str,"<mo>]</mo>"]

parenthesize str ==
   concat ["<mo>(</mo>",str,"<mo>)</mo>"]

precondition expr ==
   outputTran$Lisp expr

formatSpecial(op : S, args : L E, prec : I) : S ==
   arg : E
   prescript : Boolean := false
   op = "theMap" => "<mtext>theMap(...)</mtext>"
   op = "AGGLST" =>
       formatNary(",",args,prec)
   op = "AGGSET" =>
       formatNary(";",args,prec)
   op = "TAG" =>
       group concat [formatMml(first args,prec),
                     "<mo>&#x02192;</mo>",
                     formatMml(second args,prec)]
       --RightArrow
   op = "SLASH" =>
       group concat [formatMml(first args,prec),
                     "<mo>/</mo>",formatMml(second args,prec)]

```

```

op = "VCONCAT" =>
  group concat("<table><mtr>",
    concat(concat([concat("<td>",concat(formatMml(u, minPrec),"</td>"))
      for u in args]::L S),
      "</mtr></table>"))
op = "CONCATB" =>
  formatNary(" ",args,prec)
op = "CONCAT" =>
  formatNary("",args,minPrec)
op = "QUOTE" =>
  group concat("<mo>'</mo>",formatMml(first args, minPrec))
op = "BRACKET" =>
  group addBrackets ungroup formatMml(first args, minPrec)
op = "BRACE" =>
  group addBraces ungroup formatMml(first args, minPrec)
op = "PAREN" =>
  group parenthesize ungroup formatMml(first args, minPrec)
op = "OVERBAR" =>
  null args => ""
  group concat ["<mover accent='true'><mrow>","_
    formatMml(first args,minPrec),_
    "</mrow><mo stretchy='true'>&#x00AF;</mo></mover>"]
  --OverBar
op = "ROOT" =>
  null args => ""
  tmp : S := group formatMml(first args, minPrec)
  null rest args => concat ["<msqrt>",tmp,"</msqrt>"]
  group concat
    ["<mroot><mrow>",tmp,"</mrow>","_
      formatMml(first rest args, minPrec),"</mroot>"]
op = "SEGMENT" =>
  tmp : S := concat [formatMml(first args, minPrec),"<mo>.</mo>"]
  group
    null rest args => tmp
    concat [tmp,formatMml(first rest args, minPrec)]
-- SUB should now be diverted in formatMml although I'll leave
-- the code here for now.
op = "SUB" =>
  group concat ["<msub>",formatMml(first args, minPrec),
    formatSpecial("AGGLST",rest args,minPrec),"</msub>"]
-- SUPERSUB should now be diverted in formatMml although I'll leave
-- the code here for now.
op = "SUPERSUB" =>
  base:S := formatMml(first args, minPrec)
  args := rest args
  if #args = 1 then
    "<msub><mrow>"base"</mrow><mrow>"_
      formatMml(first args, minPrec)"</mrow></msub>"
  else if #args = 2 then
    -- it would be nice to substitute &#x2032; for , in the case of

```

```

-- an ordinary derivative, it looks a lot better.
"<msubsup><mrow>"base"</mrow><mrow>"_
  formatMml(first args,minPrec)_
  "</mrow><mrow>"_
  formatMml(first rest args, minPrec)_
  "</mrow></msubsup>"
else if #args = 3 then
  "<mmultiscripts><mrow>"base"</mrow><mrow>"_
  formatMml(first args,minPrec)"</mrow><mrow>"_
  formatMml(first rest args,minPrec)"</mrow><mprescripts/><mrow>"_
  formatMml(first rest rest args,minPrec)_
  "</mrow><none/></mmultiscripts>"
else if #args = 4 then
  "<mmultiscripts><mrow>"base"</mrow><mrow>"_
  formatMml(first args,minPrec)"</mrow><mrow>"_
  formatMml(first rest args,minPrec)"</mrow><mprescripts/><mrow>"_
  formatMml(first rest rest args,minPrec)_
  "</mrow><mrow>"formatMml(first rest rest rest args,minPrec)_
  "</mrow></mmultiscripts>"
else
  "<mtext>Problem with multiscript object</mtext>"
op = "SC" =>
  -- need to handle indentation someday
  null args => ""
  tmp := formatNaryNoGroup("</mtd></mtr><mtr><mtd>", args, minPrec)
  group concat ["<mtable><mtr><mtd>",tmp,"</mtd></mtr></mtable>"]
op = "MATRIX" => formatMatrix rest args
op = "ZAG" =>
-- {{+}}{3}{{ZAG}}{1}{7}}{{ZAG}}{1}{15}}{{ZAG}}{1}{1}}
-- {{ZAG}}{1}{25}}{{ZAG}}{1}{1}}{{ZAG}}{1}{7}}{{ZAG}}{1}{4}}
-- to format continued fraction traditionally need to intercept it at the
-- formatNary of the "+"
  concat [" \zag{" ,formatMml(first args, minPrec),"}{" ,
    formatMml(first rest args,minPrec),"}"]
  concat ["<mtext>not done yet for: " ,op,"</mtext>"]

formatSub(expr : E, args : L E, opPrec : I) : S ==
  -- This one produces differential notation partial derivatives.
  -- It doesn't work in all cases and may not be workable, use
  -- formatSub1 below for now.
  -- At this time this is only to handle partial derivatives.
  -- If the SUB case handles anything else I'm not aware of it.
  -- This an example of the 4th partial of y(x,z) w.r.t. x,x,z,x
  -- {{{SUB}}{y}}{{CONCAT}}{{CONCAT}}{{CONCAT}}{{CONCAT}}{,}{1}}
  -- {{CONCAT}}{,}{1}}{{CONCAT}}{,}{2}}{{CONCAT}}{,}{1}}{x}{z}}
  atomE : L E := atomize(expr)
  op : S := stringify first atomE
  op ~= "SUB" => "<mtext>Mistake in formatSub: no SUB</mtext>"
  stringify first rest atomE ~= "CONCAT" => _
    "<mtext>Mistake in formatSub: no CONCAT</mtext>"

```

```

-- expecting form for atomE like
-- [{SUB}{func}{CONCAT}...{CONCAT}{,}{n}{CONCAT}{,}{n}...{CONCAT}{,}{n}],
-- counting the first CONCATs before the comma gives the number of
-- derivatives
ndiffs : I := 0
tmpLE : L E := rest rest atomE
while stringify first tmpLE = "CONCAT" repeat
  ndiffs := ndiffs+1
  tmpLE := rest tmpLE
numLS : L S := nil
i : I := 1
while i < ndiffs repeat
  numLS := append(numLS,list(stringify first rest tmpLE))
  tmpLE := rest rest rest tmpLE
  i := i+1
numLS := append(numLS,list(stringify first rest tmpLE))
-- numLS contains the numbers of the bound variables as strings
-- for the differentiations, thus for the differentiation [x,x,z,x]
-- for y(x,z) numLS = ["1","1","2","1"]
posLS : L S := nil
i := 0
-- sayTeX$Lisp "formatSub: nargs = "string(#args)
while i < #args repeat
  posLS := append(posLS,list(string(i+1)))
  i := i+1
-- posLS contains the positions of the bound variables in args
-- as a list of strings, e.g. for the above example ["1","2"]
tmpS: S := stringify atomE.2
if ndiffs = 1 then
  s : S := "<mfrac><mo>&#x02202;</mo><mi>"tmpS"</mi><mrow>"
else
  s : S := "<mfrac><mrow><msup><mo>&#x02202;</mo><mn>"string(ndiffs)"</mn></msup><mi>"tmpS"</mi></mrow>"
-- need to find the order of the differentiation w.r.t. the i-th
-- variable
i := 1
j : I
k : I
tmpS: S
while i < #posLS+1 repeat
  j := 0
  k := 1
  while k < #numLS + 1 repeat
    if numLS.k = string i then j := j + 1
    k := k+1
  if j > 0 then
    tmpS := stringify args.i
    if j = 1 then
      s := s"<mo>&#x02202;</mo><mi>"tmpS"</mi>"
    else
      s := s"<mo>&#x02202;</mo><msup><mi>"tmpS_

```



```

            "</mi><mn>"string(j)"</mn></msup>"
        i := i + 1
    s := s"</mrow></mfrac><mo>(</mo>"
    i := 1
    while i < #posLS+1 repeat
        tmpS := stringify args.i
        s := s"<mi>"tmpS"</mi>"
        if i < #posLS then s := s"<mo>,</mo>"
        i := i+1
    s := s"<mo>></mo>"

formatSub1(expr : E, args : L E, opPrec : I) : S ==
-- This one produces partial derivatives notated by ",n" as
-- subscripts.
-- At this time this is only to handle partial derivatives.
-- If the SUB case handles anything else I'm not aware of it.
-- This an example of the 4th partial of y(x,z) w.r.t. x,x,z,x
-- {{{SUB}{y}{{CONCAT}}{{CONCAT}}{{CONCAT}}{{CONCAT}{{,}{1}}
-- {{CONCAT}{{,}{1}}}}{{CONCAT}{{,}{2}}}}{{CONCAT}{{,}{1}}}}{x}{z}},
-- here expr is everything in the first set of braces and
-- args is {{x}{z}}
atomE : L E := atomize(expr)
op : S := stringify first atomE
op ^= "SUB" => "<mtext>Mistake in formatSub: no SUB</mtext>"
stringify first rest rest atomE ^= "CONCAT" => "<mtext>Mistake in formatSub: no CONCAT</mt
-- expecting form for atomE like
--[{{SUB}{{func}{{CONCAT}}...{{CONCAT}{{,}{n}{{CONCAT}{{,}{n}...{{CONCAT}{{,}{n}}},
--counting the first CONCATs before the comma gives the number of
--derivatives
ndiffs : I := 0
tmpLE : L E := rest rest atomE
while stringify first tmpLE = "CONCAT" repeat
    ndiffs := ndiffs+1
    tmpLE := rest tmpLE
numLS : L S := nil
i : I := 1
while i < ndiffs repeat
    numLS := append(numLS,list(stringify first rest tmpLE))
    tmpLE := rest rest rest tmpLE
    i := i+1
numLS := append(numLS,list(stringify first rest tmpLE))
-- numLS contains the numbers of the bound variables as strings
-- for the differentiations, thus for the differentiation [x,x,z,x]
-- for y(x,z) numLS = ["1","1","2","1"]
posLS : L S := nil
i := 0
-- sayTeX$Lisp "formatSub: nargs = "string(#args)
while i < #args repeat
    posLS := append(posLS,list(string(i+1)))
    i := i+1

```

```

-- posLS contains the positions of the bound variables in args
-- as a list of strings, e.g. for the above example ["1","2"]
funcS: S := stringify atomE.2
s : S := "<msub><mi>funcS</mi><mrow>"
i := 1
while i < #numLS+1 repeat
  s := s"<mo>,</mo><mn>numLS.i</mn>"
  i := i + 1
s := s"</mrow></msub><mo>(</mo>"
i := 1
while i < #posLS+1 repeat
--   tmpS := stringify args.i
  tmpS := formatMml(first args,minPrec)
  args := rest args
  s := s"<mi>tmpS</mi>"
  if i < #posLS then s := s"<mo>,</mo>"
  i := i+1
s := s"<mo></mo>"

formatSuperSub(expr : E, args : L E, opPrec : I) : S ==
-- this produces prime notation ordinary derivatives.
-- first have to divine the semantics, add cases as needed
--   WriteLine$Lisp "SuperSub1 begin"
atomE : L E := atomize(expr)
op : S := stringify first atomE
--   WriteLine$Lisp "op: "op
op ^ = "SUPERSUB" => _
  "<mtext>Mistake in formatSuperSub: no SUPERSUB1</mtext>"
#args ^ = 1 => "<mtext>Mistake in SuperSub1: #args <> 1</mtext>"
var : E := first args
-- should be looking at something like {{SUPERSUB}{var}{ }{,,...}} for
-- example here's the second derivative of y w.r.t. x
-- {{SUPERSUB}{y}{ }{,,}}{x}}, expr is the first {} and args is the
-- {x}
funcS : S := stringify first rest atomE
--   WriteLine$Lisp "funcS: "funcS
bvarS : S := stringify first args
--   WriteLine$Lisp "bvarS: "bvarS
-- count the number of commas
commaS : S := stringify first rest rest rest atomE
commaTest : S := ","
i : I := 0
while position(commaTest,commaS,1) > 0 repeat
  i := i+1
  commaTest := commaTest","
s : S := "<msup><mi>funcS</mi><mrow>"
--   WriteLine$Lisp "s: "s
j : I := 0
while j < i repeat
  s := s"<mo>&#x02032;</mo>"

```

```

j := j + 1
s := s"</mrow></msup><mo>&#x02061;</mo><mo></mo>"_
      formatMml(first args,minPrec)"<mo></mo>"

formatSuperSub1(expr : E, args : L E, opPrec : I) : S ==
-- This one produces ordinary derivatives with differential notation,
-- it needs a little more work yet.
-- first have to divine the semantics, add cases as needed
-- WriteLine$Lisp "SuperSub begin"
atomE : L E := atomize(expr)
op : S := stringify first atomE
op ^= "SUPERSUB" => _
      "<mtext>Mistake in formatSuperSub: no SUPERSUB</mtext>"
#args ^= 1 => "<mtext>Mistake in SuperSub: #args <> 1</mtext>"
var : E := first args
-- should be looking at something like {{SUPERSUB}{var}{ }{,,...}} for
-- example here's the second derivative of y w.r.t. x
-- {{SUPERSUB}{y}{ }{,,}}{x}}, expr is the first {} and args is the
-- {x}
funcS : S := stringify first rest atomE
bvarS : S := stringify first args
-- count the number of commas
commaS : S := stringify first rest rest rest atomE
commaTest : S := ","
ndiffs : I := 0
while position(commaTest,commaS,1) > 0 repeat
  ndiffs := ndiffs+1
  commaTest := commaTest","
s : S := "<mfrac><mrow><msup><mo>&#x02146;</mo><mn>"string(ndiffs)_
      "</mn></msup><mi>"funcS"</mi></mrow><mrow><mo>&#x02146;</mo><msup><mi>"_
      formatMml(first args,minPrec)"</mi><mn>"string(ndiffs)_
      "</mn></msup></mrow></mfrac><mo>&#x02061;</mo><mo></mo><mi>"_
      formatMml(first args,minPrec)"</mi><mo></mo>"

formatPlex(op : S, args : L E, prec : I) : S ==
checkarg:Boolean := false
hold : S
p : I := position(op,plexOps)
p < 1 => error "unknown plex op"
op = "INTSIGN" => formatIntSign(args,minPrec)
opPrec := plexPrecs.p
n : I := #args
(n ^= 2) and (n ^= 3) => error "wrong number of arguments for plex"
s : S :=
  op = "SIGMA" =>
    checkarg := true
    "<mo>&#x02211;</mo>"
  -- Sum
  op = "SIGMA2" =>
    checkarg := true

```

```

    "<mo>&#x02211;</mo>"
-- Sum
op = "PI"      =>
  checkarg := true
  "<mo>&#x0220F;</mo>"
-- Product
op = "PI2"     =>
  checkarg := true
  "<mo>&#x0220F;</mo>"
-- Product
op = "INTSIGN" => "<mo>&#x0222B;</mo>"
-- Integral, int
op = "INDEFINTEGRAL" => "<mo>&#x0222B;</mo>"
-- Integral, int
"???"
hold := formatMml(first args,minPrec)
args := rest args
if op ^= "INDEFINTEGRAL" then
  if hold ^= "" then
    s := concat ["<munderover>",s,group hold]
  else
    s := concat ["<munderover>",s,group " "]
  if not null rest args then
    hold := formatMml(first args,minPrec)
    if hold ^= "" then
      s := concat [s,group hold,"</munderover>"]
    else
      s := concat [s,group " ","</munderover>"]
    args := rest args
  -- if checkarg true need to test op arg for "+" at least
  -- and wrap parentheses if so
  if checkarg then
    la : L E := (first args pretend L E)
    opa : S := stringify first la
    if opa = "+" then
      s :=
        concat [s,"<mo>(</mo>",formatMml(first args,minPrec),"<mo></mo>"]
      else s := concat [s,formatMml(first args,minPrec)]
    else s := concat [s,formatMml(first args,minPrec)]
  else
    hold := group concat [hold,formatMml(first args,minPrec)]
    s := concat [s,hold]
  -- if opPrec < prec then s := parenthesize s
-- getting ugly parentheses on fractions
group s

formatIntSign(args : L E, opPrec : I) : S ==
-- the original OutputForm expression looks something like this:
-- {{INTSIGN}}{NOTHING or lower limit?}
-- {bvar or upper limit?}{{*}}{integrand}{{CONCAT}}{d}{{axiom var}}}}

```

```

-- the args list passed here consists of the rest of this list, i.e.
-- starting at the NOTHING or ...
(stringify first args) = "NOTHING" =>
  -- the bound variable is the second one in the argument list
  bvar : E := first rest args
  bvarS : S := stringify bvar
  tmpS : S
  i : I := 0
  u1 : US
  u2 : US
  -- this next one atomizes the integrand plus differential
  atomE : L E := atomize(first rest rest args)
  -- pick out the bound variable used by axiom
  varRS : S := stringify last(atomE)
  tmpLE : L E := ((first rest rest args) pretend L E)
  integrand : S := formatMml(first rest tmpLE,minPrec)
  -- replace the bound variable, i.e. axiom uses something of the form
  -- %A for the bound variable and puts the original variable used
  -- in the input command as a superscript on the integral sign.
  -- I'm assuming that the axiom variable is 2 characters.
  while (i := position(varRS,integrand,i+1)) > 0 repeat
    u1 := segment(1,i-1)$US
    u2 := segment(i+2,#integrand)$US
    integrand := concat [integrand.u1,bvarS,integrand.u2]
  concat ["<mrow><mo>&#x0222B;</mo>" integrand _
    "<mo>&#x02146;</mo><mi>" bvarS "</mi></mrow>"]

lowlim : S := stringify first args
highlim : S := stringify first rest args
bvar : E := last atomize(first rest rest args)
bvarS : S := stringify bvar
tmpLE : L E := ((first rest rest args) pretend L E)
integrand : S := formatMml(first rest tmpLE,minPrec)
concat ["<mrow><munderover><mo>&#x0222B;</mo><mi>" lowlim "</mi><mi>" highlim "</mi></munderover>"]

formatMatrix(args : L E) : S ==
  -- format for args is [[ROW ...],[ROW ...],[ROW ...]]
  -- generate string for formatting columns (centered)
  group addBrackets concat
    ["<mtable><mtr><mtd>",formatNaryNoGroup("</mtd></mtr><mtr><mtd>",args,minPrec),
      "</mtd></mtr></mtable>"]

formatFunction(op : S, args : L E, prec : I) : S ==
  group concat ["<mo>",op,"</mo>",parenthesize formatNary(",",args,minPrec)]

formatNullary(op : S) ==
  op = "NOTHING" => ""
  group concat ["<mo>",op,"</mo><mo>(</mo><mo>)</mo>"]

```

```

formatUnary(op : S, arg : E, prec : I) ==
  p : I := position(op, unaryOps)
  p < 1 => error "unknown unary op"
  opPrec := unaryPrecs.p
  s : S := concat ["<mo>", op, "</mo>", formatMml(arg, opPrec)]
  opPrec < prec => group parenthesize s
  op = "-" => s
  group s

formatBinary(op : S, args : L E, prec : I) : S ==
  p : I := position(op, binaryOps)
  p < 1 => error "unknown binary op"
  opPrec := binaryPrecs.p
  -- if base op is product or sum need to add parentheses
  if ATOM(first args)$Lisp@Boolean then
    opa:S := stringify first args
  else
    la : L E := (first args pretend L E)
    opa : S := stringify first la
  if (opa = "SIGMA" or opa = "SIGMA2" or opa = "PI" or opa = "PI2") _
    and op = "***" then
    s1:S:=concat ["<mo>(</mo>", formatMml(first args, opPrec), "<mo>)</mo>"]
  else
    s1 : S := formatMml(first args, opPrec)
    s2 : S := formatMml(first rest args, opPrec)
    op :=
      op = "|" => s := concat ["<mrow>", s1, "</mrow><mo>", op, "</mo><mrow>", s2, "</mrow>"]
      op = "***" => s := concat ["<msup><mrow>", s1, "</mrow><mrow>", s2, "</mrow></msup>"]
      op = "/" => s := concat ["<mfrac><mrow>", s1, "</mrow><mrow>", s2, "</mrow></mfrac>"]
      op = "OVER" => s := concat ["<mfrac><mrow>", s1, "</mrow><mrow>", s2, "</mrow></mfrac>"]
      op = "+->" => s := concat ["<mrow>", s1, "</mrow><mo>", op, "</mo><mrow>", s2, "</mrow>"]
      s := concat ["<mrow>", s1, "</mrow><mo>", op, "</mo><mrow>", s2, "</mrow>"]
    group
    op = "OVER" => s
  --
  opPrec < prec => parenthesize s
  -- ugly parentheses?
  s

formatNary(op : S, args : L E, prec : I) : S ==
  group formatNaryNoGroup(op, args, prec)

formatNaryNoGroup(op : S, args : L E, prec : I) : S ==
  checkargs:Boolean := false
  null args => ""
  p : I := position(op, naryOps)
  p < 1 => error "unknown nary op"
  -- need to test for "ZAG" case and divert it here
  -- ex 1. continuedFraction(314159/100000)
  -- {{+}}{3}{{ZAG}}{1}{7}}{{ZAG}}{1}{15}}{{ZAG}}{1}{1}}{{ZAG}}{1}{25}}
  -- {{ZAG}}{1}{1}}{{ZAG}}{1}{7}}{{ZAG}}{1}{4}}

```

```

-- this is the preconditioned output form
-- including "op", the args list would be the rest of this
-- i.e op = '+' and args = {{3}}{{ZAG}{1}{7}}{{ZAG}{1}{15}}
-- {{ZAG}{1}{1}}{{ZAG}{1}{25}}{{ZAG}{1}{1}}{{ZAG}{1}{7}}{{ZAG}{1}{4}}}
-- ex 2. continuedFraction(14159/100000)
-- this one doesn't have the leading integer
-- {{+}}{{ZAG}{1}{7}}{{ZAG}{1}{15}}{{ZAG}{1}{1}}{{ZAG}{1}{25}}
-- {{ZAG}{1}{1}}{{ZAG}{1}{7}}{{ZAG}{1}{4}}}
--
-- ex 3. continuedFraction(3,repeating [1], repeating [3,6])
-- {{+}}{{3}}{{ZAG}{1}{3}}{{ZAG}{1}{6}}{{ZAG}{1}{3}}{{ZAG}{1}{6}}
-- {{ZAG}{1}{3}}{{ZAG}{1}{6}}{{ZAG}{1}{3}}{{ZAG}{1}{6}}
-- {{ZAG}{1}{3}}{{ZAG}{1}{6}}{...}}
-- In each of these examples the args list consists of the terms
-- following the '+' op
-- so the first arg could be a "ZAG" or something
-- else, but the second arg looks like it has to be "ZAG", so maybe
-- test for #args > 1 and args.2 contains "ZAG".
-- Note that since the resulting MathML <mfrac>s are nested we need
-- to handle the whole continued fraction at once, i.e. we can't
-- just look for, e.g., {{ZAG}{1}{6}}
(#args > 1) and (position("ZAG",stringify first rest args,1) > 0) =>
  tmpS : S := stringify first args
  position("ZAG",tmpS,1) > 0 => formatZag(args)
-- position("ZAG",tmpS,1) > 0 => formatZag1(args)
  concat [formatMml(first args,minPrec) "<mo>+</mo>" _
    formatZag(rest args)]
-- At least for the ops "*", "+", "-" we need to test to see if a sigma
-- or pi is one of their arguments because we might need parentheses
-- as indicated by the problem with
-- summation(operator(f)(i),i=1..n)+1 versus
-- summation(operator(f)(i)+1,i=1..n) having identical displays as
-- of 2007-21-21
op :=
  op = ","      => "<mo>,</mo>" --originally , \:
  op = ";"      => "<mo>;</mo>" --originally ; \: should figure these out
  op = "*"      => "<mspace width='0.3em'/>"
-- InvisibleTimes
  op = " "      => "<mspace width='0.5em'/>"
  op = "ROW"    => "</mtd><mtd>"
  op = "+"      =>
    checkargs := true
    "<mo>+</mo>"
  op = "-"      =>
    checkargs := true
    "<mo>-</mo>"
  op
l : L S := nil
opPrec := naryPrecs.p
-- if checkargs is true check each arg except last one to see if it's

```

```

-- a sigma or pi and if so add parentheses. Other op's may have to be
-- checked for in future
count:I := 1
for a in args repeat
--   WriteLine$Lisp "checking args"
   if checkargs then
     if count < #args then
       -- check here for sum or product
       if ATOM(a)$Lisp@Boolean then
         opa:S := stringify a
       else
         la : L E := (a pretend L E)
         opa : S := stringify first la
       if opa = "SIGMA" or opa = "SIGMA2" or _
         opa = "PI" or opa = "PI2" then
         l := concat(op,concat(_
           concat ["<mo>(</mo>",formatMml(a,opPrec),_
             "<mo>)</mo>"],1)$L(S))$L(S)
         else l := concat(op,concat(formatMml(a,opPrec),1)$L(S))$L(S)
         else l := concat(op,concat(formatMml(a,opPrec),1)$L(S))$L(S)
         else l := concat(op,concat(formatMml(a,opPrec),1)$L(S))$L(S)
       count := count + 1
     s : S := concat reverse rest l
     opPrec < prec => parenthesize s
   s

formatZag(args : L E) : S ==
-- args will be a list of things like this {{ZAG}{1}{7}}, the ZAG
-- must be there, the '1' and '7' could conceivably be more complex
-- expressions
tmpZag : L E := first args pretend L E
-- may want to test that tmpZag contains 'ZAG'
#args > 1 => "<mfrac>"formatMml(first rest tmpZag,minPrec)"<mrow><mn>"formatMml(first rest rest tmpZag,minPrec)"</mrow></mfrac>"
-- EQUAL(tmpZag, "...")$Lisp => "<mo>&#x2026;</mo>"
(first args = "...":E)@Boolean => "<mo>&#x2026;</mo>"
position("ZAG",stringify first args,1) > 0 =>
  "<mfrac>"formatMml(first rest tmpZag,minPrec)formatMml(first rest rest tmpZag,minPrec)"</mfrac>"
  "<mtext>formatZag: Unexpected kind of ZAG</mtext>"

formatZag1(args : L E) : S ==
-- make alternative ZAG format without diminishing fonts, maybe
-- use a table
-- {{ZAG}{1}{7}}
tmpZag : L E := first args pretend L E
#args > 1 => "<mfrac>"formatMml(first rest tmpZag,minPrec)"<mrow><mn>"formatMml(first rest rest tmpZag,minPrec)"</mrow></mfrac>"
(first args = "...":E)@Boolean => "<mo>&#x2026;</mo>"
error "formatZag1: Unexpected kind of ZAG"

```



```

formatMml(expr : E, prec : I) ==
  i, len : Integer
  intSplitLen : Integer := 20
  ATOM(expr)$Lisp@Boolean =>
    str := stringify expr
    len := #str
    -- this bit seems to deal with integers
    INTEGERP$Lisp expr =>
      i := expr pretend Integer
      if (i < 0) or (i > 9)
      then
        group
          nstr : String := ""
          -- insert some blanks into the string, if too long
          while ((len := #str) > intSplitLen) repeat
            nstr := concat [nstr, " ",
              elt(str, segment(1, intSplitLen)$US)]
            str := elt(str, segment(intSplitLen+1)$US)
          empty? nstr => concat ["<mn>", str, "</mn>"]
          nstr :=
            empty? str => nstr
            concat [nstr, " ", str]
            concat ["<mn>", elt(nstr, segment(2)$US), "</mn>"]
          else str := concat ["<mn>", str, "</mn>"]
      str = "%pi" => "<mi>&#x003C0;</mi>"
      -- pi
      str = "%e" => "<mi>&#x02147;</mi>"
      -- ExponentialE
      str = "%i" => "<mi>&#x02148;</mi>"
      -- ImaginaryI
      len > 0 and str.1 = char "%" => concat(concat("<mi>", str), "</mi>")
      -- should handle floats
      len > 1 and digit? str.1 => concat ["<mn>", str, "</mn>"]
      -- presumably this is a literal string
      len > 0 and str.1 = char "_" =>
        concat(concat("<mtext>", str), "</mtext>")
      len = 1 and str.1 = char " " => " "
      (i := position(str, specialStrings)) > 0 =>
        specialStringsInMML.i
      (i := position(char " ", str)) > 0 =>
        -- We want to preserve spacing, so use a roman font.
        -- What's this for? Leave the \rm in for now so I can see
        -- where it arises. Removed 2007-02-14
        concat(concat("<mtext>", str), "</mtext>")
        -- if we get to here does that mean it's a variable?
        concat ["<mi>", str, "</mi>"]
      l : L E := (expr pretend L E)
      null l => blank
      op : S := stringify first l
      args : L E := rest l

```

```

nargs : I := #args
-- need to test here in case first l is SUPERSUB case and then
-- pass first l and args to formatSuperSub.
position("SUPERSUB",op,1) > 0 =>
  formatSuperSub(first l,args,minPrec)
-- now test for SUB
position("SUB",op,1) > 0 =>
  formatSub1(first l,args,minPrec)

-- special cases
member?(op, specialOps) => formatSpecial(op,args,prec)
member?(op, plexOps)    => formatPlex(op,args,prec)

-- nullary case
0 = nargs => formatNullary op

-- unary case
(1 = nargs) and member?(op, unaryOps) =>
  formatUnary(op, first args, prec)

-- binary case
(2 = nargs) and member?(op, binaryOps) =>
  formatBinary(op, args, prec)

-- nary case
member?(op,naryNGOps) => formatNaryNoGroup(op,args, prec)
member?(op,naryOps) => formatNary(op,args, prec)

op := formatMml(first l,minPrec)
formatFunction(op,args,prec)

```

Mathematical Markup Language Form

— MathMLFormat.input —

```

)set break resume
)spool MathMLFormat.output
)set message test on
)set message auto off
)clear all

--S 1 of 5
)set output mathml on

--R

```

--E 1

--S 2 of 5

1/2

--R

--R

--R 1

--R (1) -

--R 2

--R<math xmlns="http://www.w3.org/1998/Math/MathML" mathsize="big" display="block">

--R<mrow><mfrac><mrow><mn>1</mn></mrow><mrow><mn>2</mn></mrow></mfrac></mrow>

--R</math>

--R

--R

Type: Fraction Integer

--E 2

--S 3 of 5

1/(x+5)

--R

--R

--R 1

--R (2) -----

--R x + 5

--R<math xmlns="http://www.w3.org/1998/Math/MathML" mathsize="big" display="block">

--R<mrow><mfrac><mrow><mn>1</mn></mrow><mrow><mrow><mi>x</mi><mo>+</mo><mn>5</mn></mrow></mrow></mfrac></mrow>

--R</math>

--R

--R

Type: Fraction Polynomial Integer

--E 3

--S 4 of 5

(x+3)/(y-5)

--R

--R

--R x + 3

--R (3) -----

--R y - 5

--R<math xmlns="http://www.w3.org/1998/Math/MathML" mathsize="big" display="block">

--R<mrow><mfrac><mrow><mrow><mi>x</mi><mo>+</mo><mn>3</mn></mrow></mrow><mrow><mrow><mi>y</mi><mo>-</mo><mn>5</mn></mrow></mrow></mfrac></mrow>

--R</math>

--R

--R

Type: Fraction Polynomial Integer

--E 4

--S 5 of 5

)show MathMLFormat

--R MathMLFormat is a domain constructor

--R Abbreviation for MathMLFormat is MMLFORM

--R This constructor is exposed in this frame.

--R Issue)edit bookvol10.3.pamphlet to see algebra source code for MMLFORM

```

--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : OutputForm -> String
--R coerce : % -> OutputForm          coerceL : OutputForm -> String
--R coerceS : OutputForm -> String    display : String -> Void
--R exprex : OutputForm -> String     hash : % -> SingleInteger
--R latex : % -> String               ?~=? : (%,% ) -> Boolean
--R
--E 5

```

```

)spool
)lisp (bye)

```

— MathMLFormat.help —

=====

MathMLFormat examples

=====

MathML is an HTML-like markup language for mathematics. It uses the "knuckle" syntax of HTML such as "<mo>" to introduce a math operator and "</mo>" to mark the end of the operator. Axiom can generate MathML output and does so when it communicates to the browser front end.

This output is enabled by

```
)set output mathml on
```

after which you'll see the MathML markup as well as the algebra.
Note that you can turn off the algebra output with

```
)set output algebra off
```

but we don't do that here so you can compare the output.

1/2

1/2

```

<math xmlns="http://www.w3.org/1998/Math/MathML" mathsize="big"
  display="block">
  <mrow>
    <mn>1</mn>
    <mo>/</mo>
    <mn>2</mn>
  </mrow>

```

</math>

$1/(x+5)$

$1/(x + 5)$

```
<math xmlns="http://www.w3.org/1998/Math/MathML" mathsize="big"
  display="block">
  <mrow>
    <mn>1</mn>
    <mo>/</mo>
    <mrow>
      <mo>(</mo>
        <mi>x</mi>
        <mo>+</mo>
        <mn>5</mn>
      <mo>)</mo>
    </mrow>
  </mrow>
</math>
```

$(x+3)/(y-5)$

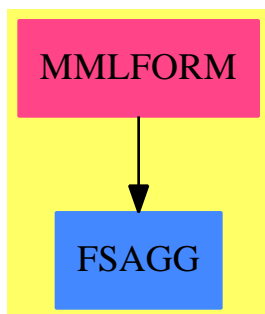
$(x + 3)/(y - 5)$

```
<math xmlns="http://www.w3.org/1998/Math/MathML" mathsize="big"
  display="block">
  <mrow>
    <mrow>
      <mo>(</mo>
        <mi>x</mi>
        <mo>+</mo>
        <mn>3</mn>
      <mo>)</mo>
    </mrow>
    <mo>/</mo>
    <mrow>
      <mo>(</mo>
        <mi>y</mi>
        <mo>-</mo>
        <mn>5</mn>
      <mo>)</mo>
    </mrow>
  </mrow>
</math>
```

See Also:

o)show MathMLFormat

MathMLForm (MMLFORM)



Exports:

coerce coerceL coerceS display exprex
 hash latex ?=? ?~=?

— domain MMLFORM MathMLFormat —

```

\getchunk{public declarations}
\getchunk{private constant declarations}
\getchunk{private function declarations}
\getchunk{public function definitions}
\getchunk{display functions}
\getchunk{formatting functions}
  
```

— MMLFORM.dotabb —

```

"MMLFORM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MMLFORM"]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"MMLFORM" -> "FSAGG"
  
```

domain MATRIX Matrix

— Matrix.input —

```

)set break resume
)sys rm -f Matrix.output
)spool Matrix.output
)set message test on
)set message auto off
)clear all
--S 1 of 38
m : Matrix(Integer) := new(3,3,0)
--R
--R
--R      +0  0  0+
--R      |      |
--R  (1) |0  0  0|
--R      |      |
--R      +0  0  0+
--R
--R                                          Type: Matrix(Integer)
--E 1

--S 2 of 38
setelt(m,2,3,5)
--R
--R
--R  (2)  5
--R
--R                                          Type: PositiveInteger
--E 2

--S 3 of 38
m(1,2) := 10
--R
--R
--R  (3)  10
--R
--R                                          Type: PositiveInteger
--E 3

--S 4 of 38
m
--R
--R
--R      +0  10  0+
--R      |      |
--R  (4) |0  0  5|
--R      |      |
--R      +0  0  0+
--R
--R                                          Type: Matrix(Integer)
--E 4

--S 5 of 38
matrix [ [1,2,3,4],[0,9,8,7] ]
--R
--R

```

```

--R      +1  2  3  4+
--R  (5) |      |
--R      +0  9  8  7+
--R
--R                                          Type: Matrix(Integer)
--E 5

```

```

--S 6 of 38
dm := diagonalMatrix [1,x**2,x**3,x**4,x**5]
--R
--R
--R      +1  0  0  0  0  +
--R      |      |
--R      |      2      |
--R      |0  x  0  0  0  |
--R      |      |
--R      |      3      |
--R  (6) |0  0  x  0  0  |
--R      |      |
--R      |      4      |
--R      |0  0  0  x  0  |
--R      |      |
--R      |      5      |
--R      +0  0  0  0  x  +
--R
--R                                          Type: Matrix(Polynomial(Integer))
--E 6

```

```

--S 7 of 38
setRow!(dm,5,vector [1,1,1,1,1])
--R
--R
--R      +1  0  0  0  0+
--R      |      |
--R      |      2      |
--R      |0  x  0  0  0|
--R      |      |
--R      |      3      |
--R  (7) |0  0  x  0  0|
--R      |      |
--R      |      4      |
--R      |0  0  0  x  0|
--R      |      |
--R      +1  1  1  1  1+
--R
--R                                          Type: Matrix(Polynomial(Integer))
--E 7

```

```

--S 8 of 38
setColumn!(dm,2,vector [y,y,y,y,y])
--R
--R
--R      +1  y  0  0  0+

```



```

--R      |
--R      |0 y 0 0 0|
--R      |
--R      |      3
--R (8) |0 y x 0 0|
--R      |
--R      |      4
--R      |0 y 0 x 0|
--R      |
--R      +1 y 1 1 1+
--R
--E 8

```

Type: Matrix(Polynomial(Integer))

```

--S 9 of 38
cdm := copy(dm)
--R
--R
--R      +1 y 0 0 0+
--R      |
--R      |0 y 0 0 0|
--R      |
--R      |      3
--R (9) |0 y x 0 0|
--R      |
--R      |      4
--R      |0 y 0 x 0|
--R      |
--R      +1 y 1 1 1+
--R
--E 9

```

Type: Matrix(Polynomial(Integer))

```

--S 10 of 38
setelt(dm,4,1,1-x**7)
--R
--R
--R      7
--R (10) - x + 1
--R
--E 10

```

Type: Polynomial(Integer)

```

--S 11 of 38
[dm,cdm]
--R
--R
--R      + 1      y 0 0 0+ +1 y 0 0 0+
--R      |      |      |
--R      | 0      y 0 0 0| |0 y 0 0 0|
--R      |      |      |
--R      |      3      |      3
--R (11) [| 0      y x 0 0|, |0 y x 0 0|]

```

```

--R      |      |      |      |
--R      |  7      4      |      4      |
--R      |- x  + 1  y  0  x  0| 0  y  0  x  0|
--R      |      |      |      |
--R      +  1      y  1  1  1+ +1  y  1  1  1+
--R                                          Type: List(Matrix(Polynomial(Integer)))
--E 11

```

```

--S 12 of 38
subMatrix(dm,2,3,2,4)
--R
--R
--R      +y  0  0+
--R  (12) |      |
--R      |  3  |
--R      +y  x  0+
--R                                          Type: Matrix(Polynomial(Integer))
--E 12

```

```

--S 13 of 38
d := diagonalMatrix [1.2,-1.3,1.4,-1.5]
--R
--R
--R      +1.2  0.0  0.0  0.0 +
--R      |      |      |      |
--R      |0.0  - 1.3  0.0  0.0 |
--R  (13) |      |      |      |
--R      |0.0  0.0  1.4  0.0 |
--R      |      |      |      |
--R      +0.0  0.0  0.0  - 1.5+
--R                                          Type: Matrix(Float)
--E 13

```

```

--S 14 of 38
e := matrix [ [6.7,9.11],[-31.33,67.19] ]
--R
--R
--R      + 6.7  9.11 +
--R  (14) |      |
--R      +- 31.33 67.19+
--R                                          Type: Matrix(Float)
--E 14

```

```

--S 15 of 38
setsubMatrix!(d,1,2,e)
--R
--R
--R      +1.2  6.7  9.11  0.0 +
--R      |      |      |      |
--R      |0.0  - 31.33 67.19  0.0 |

```

```

--R (15) |
--R |0.0 0.0 1.4 0.0 |
--R |
--R +0.0 0.0 0.0 - 1.5+
--R
--R                                          Type: Matrix(Float)
--E 15

```

```

--S 16 of 38
d
--R
--R
--R +1.2 6.7 9.11 0.0 +
--R |
--R |0.0 - 31.33 67.19 0.0 |
--R (16) |
--R |0.0 0.0 1.4 0.0 |
--R |
--R +0.0 0.0 0.0 - 1.5+
--R
--R                                          Type: Matrix(Float)
--E 16

```

```

--S 17 of 38
a := matrix [ [1/2,1/3,1/4],[1/5,1/6,1/7] ]
--R
--R
--R +1 1 1+
--R |- - -|
--R |2 3 4|
--R (17) |
--R |1 1 1|
--R |- - -|
--R +5 6 7+
--R
--R                                          Type: Matrix(Fraction(Integer))
--E 17

```

```

--S 18 of 38
b := matrix [ [3/5,3/7,3/11],[3/13,3/17,3/19] ]
--R
--R
--R +3 3 3+
--R |- - --|
--R |5 7 11|
--R (18) |
--R | 3 3 3|
--R |-- -- --|
--R +13 17 19+
--R
--R                                          Type: Matrix(Fraction(Integer))
--E 18

```

```

--S 19 of 38

```

```
horizConcat(a,b)
```

```
--R
--R
--R      +1  1  1  3  3  3+
--R      |- - - - - --|
--R      |2  3  4  5  7  11|
--R  (19) |          |
--R      |1  1  1  3  3  3|
--R      |- - - -- -- --|
--R      +5  6  7  13  17  19+
--R
--E 19
```

Type: Matrix(Fraction(Integer))

```
--S 20 of 38
```

```
vab := vertConcat(a,b)
```

```
--R
--R
--R      +1  1  1 +
--R      |- - - |
--R      |2  3  4 |
--R      |          |
--R      |1  1  1 |
--R      |- - - |
--R      |5  6  7 |
--R  (20) |          |
--R      |3  3  3|
--R      |- - --|
--R      |5  7  11|
--R      |          |
--R      | 3  3  3|
--R      |-- -- --|
--R      +13  17  19+
--R
--E 20
```

Type: Matrix(Fraction(Integer))

```
--S 21 of 38
```

```
transpose vab
```

```
--R
--R
--R      +1  1  3  3+
--R      |- - - --|
--R      |2  5  5  13|
--R      |          |
--R      |1  1  3  3|
--R  (21) |- - - --|
--R      |3  6  7  17|
--R      |          |
--R      |1  1  3  3|
--R      |- - -- --|
--R      +4  7  11  19+
```

```

--R
--E 21
Type: Matrix(Fraction(Integer))

--S 22 of 38
m := matrix [ [1,2],[3,4] ]
--R
--R
--R      +1  2+
--R  (22) |   |
--R      +3  4+
--R
--R
--E 22
Type: Matrix(Integer)

--S 23 of 38
4 * m * (-5)
--R
--R
--R      +- 20 - 40+
--R  (23) |       |
--R      +- 60 - 80+
--R
--R
--E 23
Type: Matrix(Integer)

--S 24 of 38
n := matrix([ [1,0,-2],[-3,5,1] ])
--R
--R
--R      + 1  0 - 2+
--R  (24) |       |
--R      +- 3  5  1 +
--R
--R
--E 24
Type: Matrix(Integer)

--S 25 of 38
m * n
--R
--R
--R      +- 5  10  0 +
--R  (25) |       |
--R      +- 9  20 - 2+
--R
--R
--E 25
Type: Matrix(Integer)

--S 26 of 38
vec := column(n,3)
--R
--R
--R  (26) [- 2,1]
--R
--R
--E 26
Type: Vector(Integer)

```

```

--S 27 of 38
vec * m
--R
--R
--R (27) [1,0]
--R
--R                                         Type: Vector(Integer)
--E 27

--S 28 of 38
m * vec
--R
--R
--R (28) [0,- 2]
--R
--R                                         Type: Vector(Integer)
--E 28

--S 29 of 38
hilb := matrix([ [1/(i + j) for i in 1..3] for j in 1..3])
--R
--R
--R
--R      +1  1  1+
--R      |-  -  -|
--R      |2  3  4|
--R      |      |
--R      |1  1  1|
--R (29) |-  -  -|
--R      |3  4  5|
--R      |      |
--R      |1  1  1|
--R      |-  -  -|
--R      +4  5  6+
--R
--R                                         Type: Matrix(Fraction(Integer))
--E 29

--S 30 of 38
inverse(hilb)
--R
--R
--R
--R      + 72      - 240      180 +
--R      |          |          |
--R (30) |- 240      900      - 720|
--R      |          |          |
--R      + 180      - 720      600 +
--R
--R                                         Type: Union(Matrix(Fraction(Integer)),...)
--E 30

--S 31 of 38
mm := matrix([ [1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16] ])
--R

```

Type: Matrix(Integer)

```
Type: Union("failed",...)
```

Type: NonNegativeInteger

Type: PositiveInteger

Type: PositiveInteger

Type: PositiveInteger

```

--S 37 of 38
nullSpace(mm)
--R
--R
--R (37) [[1,- 2,1,0],[2,- 3,0,1]]
--R
--R                                         Type: List(Vector(Integer))
--E 37

--S 38 of 38
rowEchelon(mm)
--R
--R
--R      +1  2  3  4  +
--R      |          |
--R      |0  4  8 12|
--R (38) |          |
--R      |0  0  0  0 |
--R      |          |
--R      +0  0  0  0 +
--R
--R                                         Type: Matrix(Integer)
--E 38
)spool
)lisp (bye)

```

— Matrix.help —

=====

Matrix examples

=====

The Matrix domain provides arithmetic operations on matrices and standard functions from linear algebra. This domain is similar to the TwoDimensionalArray domain, except that the entries for Matrix must belong to a Ring.

=====

Creating Matrices

=====

There are many ways to create a matrix from a collection of values or from existing matrices.

If the matrix has almost all items equal to the same value, use new to create a matrix filled with that value and then reset the entries that are different.


```

m : Matrix(Integer) := new(3,3,0)
+0  0  0+
|    |
|0  0  0|
|    |
+0  0  0+
Type: Matrix Integer

```

To change the entry in the second row, third column to 5, use `setelt`.

```

setelt(m,2,3,5)
5
Type: PositiveInteger

```

An alternative syntax is to use assignment.

```

m(1,2) := 10
10
Type: PositiveInteger

```

The matrix was destructively modified.

```

m
+0  10  0+
|    |
|0  0  5|
|    |
+0  0  0+
Type: Matrix Integer

```

If you already have the matrix entries as a list of lists, use `matrix`.

```

matrix [ [1,2,3,4],[0,9,8,7] ]
+1  2  3  4+
|    |
+0  9  8  7+
Type: Matrix Integer

```

If the matrix is diagonal, use `diagonalMatrix`.

```

dm := diagonalMatrix [1,x**2,x**3,x**4,x**5]
+1  0  0  0  0 +
|    |
|  2  |
|0  x  0  0  0 |
|    |
|  3  |
|0  0  x  0  0 |
|    |
|  4  |

```

```

|0 0 0 x 0|
|          |
|          5|
+0 0 0 0 x+
Type: Matrix Polynomial Integer

```

Use `setRow` and `setColumn` to change a row or column of a matrix.

```

setRow!(dm,5,vector [1,1,1,1,1])
+1 0 0 0 0+
|          |
|      2    |
|0 x 0 0 0|
|          |
|      3    |
|0 0 x 0 0|
|          |
|      4    |
|0 0 0 x 0|
|          |
+1 1 1 1 1+
Type: Matrix Polynomial Integer

```

```

setColumn!(dm,2,vector [y,y,y,y,y])
+1 y 0 0 0+
|          |
|0 y 0 0 0|
|          |
|      3    |
|0 y x 0 0|
|          |
|      4    |
|0 y 0 x 0|
|          |
+1 y 1 1 1+
Type: Matrix Polynomial Integer

```

Use `copy` to make a copy of a matrix.

```

cdm := copy(dm)
+1 y 0 0 0+
|          |
|0 y 0 0 0|
|          |
|      3    |
|0 y x 0 0|
|          |
|      4    |
|0 y 0 x 0|
|          |

```

```
+1 y 1 1 1+
Type: Matrix Polynomial Integer
```

This is useful if you intend to modify a matrix destructively but want a copy of the original.

```
setelt(dm,4,1,1-x**7)
7
- x + 1
Type: Polynomial Integer
```

```
[dm,cdm]
+ 1 y 0 0 0+ +1 y 0 0 0+
| | | |
| 0 y 0 0 0| |0 y 0 0 0|
| | | |
| 3 | | 3 |
[| 0 y x 0 0| |0 y x 0 0|]
| | | |
| 7 4 | | 4 |
|- x + 1 y 0 x 0| |0 y 0 x 0|
| | | |
+ 1 y 1 1 1+ +1 y 1 1 1+
Type: List Matrix Polynomial Integer
```

Use `subMatrix` to extract part of an existing matrix. The syntax is `subMatrix(m, firstrow, lastrow, firstcol, lastcol)`.

```
subMatrix(dm,2,3,2,4)
+y 0 0+
| |
| 3 |
+y x 0+
Type: Matrix Polynomial Integer
```

To change a submatrix, use `setsubMatrix`.

```
d := diagonalMatrix [1.2,-1.3,1.4,-1.5]
+1.2 0.0 0.0 0.0 +
| |
|0.0 - 1.3 0.0 0.0 |
| |
|0.0 0.0 1.4 0.0 |
| |
+0.0 0.0 0.0 - 1.5+
Type: Matrix Float
```

If `e` is too big to fit where you specify, an error message is displayed. Use `subMatrix` to extract part of `e`, if necessary.

```
e := matrix [ [6.7,9.11],[-31.33,67.19] ]
+ 6.7    9.11 +
|          |
+- 31.33  67.19+
Type: Matrix Float
```

This changes the submatrix of d whose upper left corner is at the first row and second column and whose size is that of e.

```
setsubMatrix!(d,1,2,e)
+1.2    6.7    9.11    0.0 +
|          |          |
|0.0   - 31.33  67.19    0.0 |
|          |          |
|0.0    0.0    1.4    0.0 |
|          |          |
+0.0    0.0    0.0   - 1.5+
Type: Matrix Float
```

```
d
+1.2    6.7    9.11    0.0 +
|          |          |
|0.0   - 31.33  67.19    0.0 |
|          |          |
|0.0    0.0    1.4    0.0 |
|          |          |
+0.0    0.0    0.0   - 1.5+
Type: Matrix Float
```

Matrices can be joined either horizontally or vertically to make new matrices.

```
a := matrix [ [1/2,1/3,1/4],[1/5,1/6,1/7] ]
+1  1  1+
|- - -|
|2  3  4|
|    |
|1  1  1|
|- - -|
+5  6  7+
Type: Matrix Fraction Integer
```

```
b := matrix [ [3/5,3/7,3/11],[3/13,3/17,3/19] ]
+3  3  3+
|- - --|
|5  7  11|
|    |
| 3  3  3|
|-- -- --|
+13 17 19+
```

Type: Matrix Fraction Integer

Use `horizConcat` to append them side to side. The two matrices must have the same number of rows.

```
horizConcat(a,b)
+1  1  1  3  3  3+
|-  -  -  -  -  --|
|2  3  4  5  7  11|
|
|1  1  1  3  3  3|
|-  -  -  --  --  --|
+5  6  7  13 17 19+
```

Type: Matrix Fraction Integer

Use `vertConcat` to stack one upon the other. The two matrices must have the same number of columns.

```
vab := vertConcat(a,b)
+1  1  1  +
|-  -  -  |
|2  3  4  |
|
|1  1  1  |
|-  -  -  |
|5  6  7  |
|
|3  3  3  |
|-  -  --  |
|5  7  11|
|
| 3  3  3|
|--  --  --|
+13 17 19+
```

Type: Matrix Fraction Integer

The operation `transpose` is used to create a new matrix by reflection across the main diagonal.

```
transpose vab
+1  1  3  3+
|-  -  -  --|
|2  5  5  13|
|
|1  1  3  3|
|-  -  -  --|
|3  6  7  17|
|
|1  1  3  3|
|-  -  --  --|
```

```
+4  7  11  19+
```

```
Type: Matrix Fraction Integer
```

```
=====
Operations on Matrices
=====
```

Axiom provides both left and right scalar multiplication.

```
m := matrix [ [1,2],[3,4] ]
```

```
+1  2+
```

```
|  |
```

```
+3  4+
```

```
Type: Matrix Integer
```

```
4 * m * (-5)
```

```
+-- 20  - 40+
```

```
|  |
```

```
+-- 60  - 80+
```

```
Type: Matrix Integer
```

You can add, subtract, and multiply matrices provided, of course, that the matrices have compatible dimensions. If not, an error message is displayed.

```
n := matrix([ [1,0,-2],[-3,5,1] ])
```

```
+ 1  0  - 2+
```

```
|  |
```

```
+-- 3  5  1 +
```

```
Type: Matrix Integer
```

This following product is defined but $n * m$ is not.

```
m * n
```

```
+-- 5  10  0 +
```

```
|  |
```

```
+-- 9  20  - 2+
```

```
Type: Matrix Integer
```

The operations `nrows` and `ncols` return the number of rows and columns of a matrix. You can extract a row or a column of a matrix using the operations `row` and `column`. The object returned is a Vector.

Here is the third column of the matrix `n`.

```
vec := column(n,3)
```

```
[- 2,1]
```

```
Type: Vector Integer
```

You can multiply a matrix on the left by a "row vector" and on the right

by a "column vector".

```
vec * m
[1,0]
```

Type: Vector Integer

Of course, the dimensions of the vector and the matrix must be compatible or an error message is returned.

```
m * vec
[0,- 2]
```

Type: Vector Integer

The operation `inverse` computes the inverse of a matrix if the matrix is invertible, and returns "failed" if not.

This Hilbert matrix is invertible.

```
hilb := matrix([ [1/(i + j) for i in 1..3] for j in 1..3])
+1 1 1+
|- - -|
|2 3 4|
|   |
|1 1 1|
|- - -|
|3 4 5|
|   |
|1 1 1|
|- - -|
+4 5 6+
```

Type: Matrix Fraction Integer

```
inverse(hilb)
+ 72 - 240 180 +
|   |
|- 240 900 - 720|
|   |
+ 180 - 720 600 +
Type: Union(Matrix Fraction Integer,...)
```

This matrix is not invertible.

```
mm := matrix([ [1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16] ])
+1 2 3 4 +
|   |
|5 6 7 8|
|   |
|9 10 11 12|
|   |
+13 14 15 16+
```

Type: Matrix Integer

```
inverse(mm)
"failed"
```

Type: Union("failed",...)

The operation determinant computes the determinant of a matrix provided that the entries of the matrix belong to a CommutativeRing.

The above matrix mm is not invertible and, hence, must have determinant 0.

```
determinant(mm)
0
```

Type: NonNegativeInteger

The operation trace computes the trace of a square matrix.

```
trace(mm)
34
```

Type: PositiveInteger

The operation rank computes the rank of a matrix: the maximal number of linearly independent rows or columns.

```
rank(mm)
2
```

Type: PositiveInteger

The operation nullity computes the nullity of a matrix: the dimension of its null space.

```
nullity(mm)
2
```

Type: PositiveInteger

The operation nullSpace returns a list containing a basis for the null space of a matrix. Note that the nullity is the number of elements in a basis for the null space.

```
nullSpace(mm)
[[1,- 2,1,0],[2,- 3,0,1]]
```

Type: List Vector Integer

The operation rowEchelon returns the row echelon form of a matrix. It is easy to see that the rank of this matrix is two and that its nullity is also two.

```
rowEchelon(mm)
+1  2  3  4 +
|          |
```



```

|0  4  8 12|
|          |
|0  0  0  0|
|          |
+0  0  0  0 +

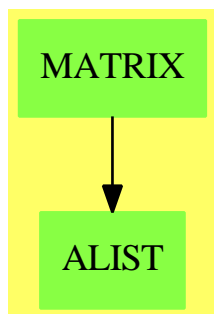
```

Type: Matrix Integer

See Also

- o)help OneDimensionalArray
- o)help TwoDimensionalArray
- o)help Vector
- o)help Permanent
- o)show Matrix

Matrix (MATRIX)



See

- ⇒ “IndexedMatrix” (IMATRIX) [10](#) on page [1238](#)
- ⇒ “RectangularMatrix” (RMATRIX) [19](#) on page [2245](#)
- ⇒ “SquareMatrix” (SQMATRIX) [20](#) on page [2547](#)

Exports:

antisymmetric?	any?	coerce	column	convert
copy	count	determinant	diagonal?	diagonalMatrix
elt	empty	empty?	eq?	eval
every?	exquo	fill!	hash	horizConcat
inverse	latex	less?	listOfLists	map
map!	matrix	maxColIndex	maxRowIndex	member?
members	minColIndex	minordet	minRowIndex	more?
ncols	new	nrows	nullSpace	nullity
parts	qelt	qsetelt!	rank	row
rowEchelon	sample	scalarMatrix	setColumn!	setRow!
setelt	setelt	setsubMatrix!	size?	square?
squareTop	subMatrix	swapColumns!	swapRows!	symmetric?
transpose	vertConcat	zero	#?	?**?
?/?	?=?	?~=?	?*?	?+?
-?	?-?			

— domain MATRIX Matrix —

```

)abbrev domain MATRIX Matrix
++ Author: Grabmeier, Gschnitzer, Williamson
++ Date Created: 1987
++ Date Last Updated: July 1990
++ Description:
++ \spadtype{Matrix} is a matrix domain where 1-based indexing is used
++ for both rows and columns.

Matrix(R): Exports == Implementation where
  R : Ring
  Row ==> Vector R
  Col ==> Vector R
  mnRow ==> 1
  mnCol ==> 1
  MATLIN ==> MatrixLinearAlgebraFunctions(R,Row,Col,$)
  MATSTOR ==> StorageEfficientMatrixOperations(R)

Exports ==> MatrixCategory(R,Row,Col) with
  diagonalMatrix: Vector R -> $
  ++ \spad{diagonalMatrix(v)} returns a diagonal matrix where the elements
  ++ of v appear on the diagonal.

  if R has ConvertibleTo InputForm then ConvertibleTo InputForm

  if R has Field then
    inverse: $ -> Union($,"failed")
    ++ \spad{inverse(m)} returns the inverse of the matrix m.
    ++ If the matrix is not invertible, "failed" is returned.
    ++ Error: if the matrix is not square.
--  matrix: Vector Vector R -> $
--  ++ \spad{matrix(v)} converts the vector of vectors v to a matrix, where

```

```
--      ++ the vector of vectors is viewed as a vector of the rows of the
--      ++ matrix
--      diagonalMatrix: Vector $ -> $
--      ++ \spad{diagonalMatrix([m1,...,mk])} creates a block diagonal matrix
--      ++ M with block matrices m1,...,mk down the diagonal,
--      ++ with 0 block matrices elsewhere.
--      vectorOfVectors: $ -> Vector Vector R
--      ++ \spad{vectorOfVectors(m)} returns the rows of the matrix m as a
--      ++ vector of vectors
```

Implementation ==>

```
InnerIndexedTwoDimensionalArray(R,mnRow,mnCol,Row,Col) add
  minr ==> minRowIndex
  maxr ==> maxRowIndex
  minc ==> minColIndex
  maxc ==> maxColIndex
  mini ==> minIndex
  maxi ==> maxIndex

  minRowIndex x == mnRow
  minColIndex x == mnCol

  swapRows_!(x,i1,i2) ==
    (i1 < minRowIndex(x)) or (i1 > maxRowIndex(x)) or _
    (i2 < minRowIndex(x)) or (i2 > maxRowIndex(x)) =>
      error "swapRows!: index out of range"
    i1 = i2 => x
    minRow := minRowIndex x
    xx := x pretend PrimitiveArray(PrimitiveArray(R))
    n1 := i1 - minRow; n2 := i2 - minRow
    row1 := qelt(xx,n1)
    qsetelt_!(xx,n1,qelt(xx,n2))
    qsetelt_!(xx,n2,row1)
    xx pretend $

  positivePower:($,Integer,NonNegativeInteger) -> $
  positivePower(x,n,nn) ==
--    one? n => x
    (n = 1) => x
    -- no need to allocate space for 3 additional matrices
    n = 2 => x * x
    n = 3 => x * x * x
    n = 4 => (y := x * x; y * y)
    a := new(nn,nn,0) pretend Matrix(R)
    b := new(nn,nn,0) pretend Matrix(R)
    c := new(nn,nn,0) pretend Matrix(R)
    xx := x pretend Matrix(R)
    power_!(a,b,c,xx,n :: NonNegativeInteger)$MATSTOR pretend $

  x:$ ** n:NonNegativeInteger ==
```

```

not((nn := nrows x) = ncols x) =>
  error "**: matrix must be square"
zero? n => scalarMatrix(nn,1)
positivePower(x,n,nn)

if R has commutative("*") then

  determinant x == determinant(x)$MATLIN
  minordet    x == minordet(x)$MATLIN

if R has EuclideanDomain then

  rowEchelon x == rowEchelon(x)$MATLIN

if R has IntegralDomain then

  rank      x == rank(x)$MATLIN
  nullity   x == nullity(x)$MATLIN
  nullSpace x == nullSpace(x)$MATLIN

if R has Field then

  inverse    x == inverse(x)$MATLIN

x:$ ** n:Integer ==
  nn := nrows x
  not(nn = ncols x) =>
    error "**: matrix must be square"
  zero? n => scalarMatrix(nn,1)
  positive? n => positivePower(x,n,nn)
  (xInv := inverse x) case "failed" =>
    error "**: matrix must be invertible"
  positivePower(xInv :: $,-n,nn)

-- matrix(v: Vector Vector R) ==
-- (rows := # v) = 0 => new(0,0,0)
-- -- error check: this is a top level function
-- cols := # v.mini(v)
-- for k in (mini(v) + 1)..maxi(v) repeat
--   cols ^= # v.k => error "matrix: rows of different lengths"
-- ans := new(rows,cols,0)
-- for i in minr(ans)..maxr(ans) for k in mini(v)..maxi(v) repeat
--   vv := v.k
--   for j in minc(ans)..maxc(ans) for l in mini(vv)..maxi(vv) repeat
--     ans(i,j) := vv.l
-- ans

diagonalMatrix(v: Vector R) ==
  n := #v; ans := zero(n,n)
  for i in minr(ans)..maxr(ans) for j in minc(ans)..maxc(ans) _

```

```

    for k in mini(v)..maxi(v) repeat qsetelt_!(ans,i,j,qelt(v,k))
ans

-- diagonalMatrix(vec: Vector $) ==
--   rows : NonNegativeInteger := 0
--   cols : NonNegativeInteger := 0
--   for r in mini(vec)..maxi(vec) repeat
--     mat := vec.r
--     rows := rows + nrows mat; cols := cols + ncols mat
--   ans := zero(rows,cols)
--   loR := minr ans; loC := minc ans
--   for r in mini(vec)..maxi(vec) repeat
--     mat := vec.r
--     hiR := loR + nrows(mat) - 1; hiC := loC + ncols(mat) - 1
--     for i in loR..hiR for k in minr(mat)..maxr(mat) repeat
--       for j in loC..hiC for l in minc(mat)..maxc(mat) repeat
--         ans(i,j) := mat(k,l)
--     loR := hiR + 1; loC := hiC + 1
--   ans

-- vectorOfVectors x ==
--   vv : Vector Vector R := new(nrows x,0)
--   cols := ncols x
--   for k in mini(vv)..maxi(vv) repeat
--     vv.k := new(cols,0)
--   for i in minr(x)..maxr(x) for k in mini(vv)..maxi(vv) repeat
--     v := vv.k
--     for j in minc(x)..maxc(x) for l in mini(v)..maxi(v) repeat
--       v.l := x(i,j)
--   vv

if R has ConvertibleTo InputForm then
  convert(x:$):InputForm ==
    convert [convert("matrix":Symbol)@InputForm,
      convert listOfLists x]$List(InputForm)



---



— MATRIX.dotabb —

"MATRIX" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MATRIX"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"MATRIX" -> "ALIST"



---



```

domain MODMON ModMonic

— ModMonic.input —

```

)set break resume
)sys rm -f ModMonic.output
)spool ModMonic.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ModMonic
--R ModMonic(R: Ring,Rep: UnivariatePolynomialCategory(R)) is a domain constructor
--R Abbreviation for ModMonic is MODMON
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for MODMON
--R
--R----- Operations -----
--R ?? : (%,R) -> %           ?? : (R,%) -> %
--R ?? : (%,%) -> %           ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R ?? : (%,%) -> %           ?-? : (%,%) -> %
--R -? : % -> %               ?=? : (%,%) -> Boolean
--R An : % -> Vector(R)       D : (%,(R -> R)) -> %
--R D : % -> %               D : (%,NonNegativeInteger) -> %
--R 1 : () -> %              UnVectorise : Vector(R) -> %
--R Vectorise : % -> Vector(R)  0 : () -> %
--R ?? : (%,PositiveInteger) -> % coefficients : % -> List(R)
--R coerce : Rep -> %         coerce : R -> %
--R coerce : Integer -> %     coerce : % -> OutputForm
--R degree : % -> NonNegativeInteger differentiate : % -> %
--R ?.? : (%,%) -> %         ?.? : (%,R) -> R
--R eval : (%,%,%) -> %      eval : (%,Equation(%)) -> %
--R ground : % -> R          ground? : % -> Boolean
--R hash : % -> SingleInteger init : () -> % if R has STEP
--R latex : % -> String      leadingCoefficient : % -> R
--R leadingMonomial : % -> %  lift : % -> Rep
--R map : ((R -> R),%) -> %   modulus : () -> Rep
--R monomial? : % -> Boolean  monomials : % -> List(%)
--R one? : % -> Boolean      pow : () -> PrimitiveArray(%)
--R pseudoRemainder : (%,%) -> % recip : % -> Union(%, "failed")
--R reduce : Rep -> %        reductum : % -> %
--R retract : % -> R         sample : () -> %
--R setPoly : Rep -> Rep     zero? : % -> Boolean
--R ?~=? : (%,%) -> Boolean
--R ?? : (Fraction(Integer),%) -> % if R has ALGEBRA(FRAC(INT))
--R ?? : (%,Fraction(Integer)) -> % if R has ALGEBRA(FRAC(INT))

```

```

--R ?? : (NonNegativeInteger,%) -> %
--R ***? : (%,NonNegativeInteger) -> %
--R ?/? : (%,R) -> % if R has FIELD
--R ?<? : (%,%) -> Boolean if R has ORDSET
--R ?<=? : (%,%) -> Boolean if R has ORDSET
--R ?>? : (%,%) -> Boolean if R has ORDSET
--R ?>=? : (%,%) -> Boolean if R has ORDSET
--R D : (%,(R -> R),NonNegativeInteger) -> %
--R D : (%,List(Symbol),List(NonNegativeInteger)) -> % if R has PDRING(SYMBOL)
--R D : (%,Symbol,NonNegativeInteger) -> % if R has PDRING(SYMBOL)
--R D : (%,List(Symbol)) -> % if R has PDRING(SYMBOL)
--R D : (%,Symbol) -> % if R has PDRING(SYMBOL)
--R D : (%,List(SingletonAsOrderedSet),List(NonNegativeInteger)) -> %
--R D : (%,SingletonAsOrderedSet,NonNegativeInteger) -> %
--R D : (%,List(SingletonAsOrderedSet)) -> %
--R D : (%,SingletonAsOrderedSet) -> %
--R ?? : (%,NonNegativeInteger) -> %
--R associates? : (%,%) -> Boolean if R has INTDOM
--R binomThmExpt : (%,%,NonNegativeInteger) -> % if R has COMRING
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if $ has CHARNZ and R has PFECAT or R has CHARNZ
--R coefficient : (%,List(SingletonAsOrderedSet),List(NonNegativeInteger)) -> %
--R coefficient : (%,SingletonAsOrderedSet,NonNegativeInteger) -> %
--R coefficient : (%,NonNegativeInteger) -> R
--R coerce : % -> % if R has INTDOM
--R coerce : Fraction(Integer) -> % if R has ALGEBRA(FRAC(INT)) or R has RETRACT(FRAC(INT))
--R coerce : SingletonAsOrderedSet -> %
--R composite : (Fraction(%),%) -> Union(Fraction(%), "failed") if R has INTDOM
--R composite : (%,%) -> Union(%, "failed") if R has INTDOM
--R computePowers : () -> PrimitiveArray(%)
--R conditionP : Matrix(%) -> Union(Vector(%), "failed") if $ has CHARNZ and R has PFECAT
--R content : (%,SingletonAsOrderedSet) -> % if R has GCDDOM
--R content : % -> R if R has GCDDOM
--R convert : % -> InputForm if SingletonAsOrderedSet has KONVERT(INFORM) and R has KONVERT(INFO
--R convert : % -> Pattern(Integer) if SingletonAsOrderedSet has KONVERT(PATTERN(INT)) and R has
--R convert : % -> Pattern(Float) if SingletonAsOrderedSet has KONVERT(PATTERN(FLOAT)) and R has
--R degree : (%,List(SingletonAsOrderedSet)) -> List(NonNegativeInteger)
--R degree : (%,SingletonAsOrderedSet) -> NonNegativeInteger
--R differentiate : (%,(R -> R),%) -> %
--R differentiate : (%,(R -> R)) -> %
--R differentiate : (%,(R -> R),NonNegativeInteger) -> %
--R differentiate : (%,List(Symbol),List(NonNegativeInteger)) -> % if R has PDRING(SYMBOL)
--R differentiate : (%,Symbol,NonNegativeInteger) -> % if R has PDRING(SYMBOL)
--R differentiate : (%,List(Symbol)) -> % if R has PDRING(SYMBOL)
--R differentiate : (%,Symbol) -> % if R has PDRING(SYMBOL)
--R differentiate : (%,NonNegativeInteger) -> %
--R differentiate : (%,List(SingletonAsOrderedSet),List(NonNegativeInteger)) -> %
--R differentiate : (%,SingletonAsOrderedSet,NonNegativeInteger) -> %
--R differentiate : (%,List(SingletonAsOrderedSet)) -> %
--R differentiate : (%,SingletonAsOrderedSet) -> %

```

```

--R discriminant : % -> R if R has COMRING
--R discriminant : (% , SingletonAsOrderedSet) -> % if R has COMRING
--R divide : (% , %) -> Record(quotient: %, remainder: %) if R has FIELD
--R divideExponents : (% , NonNegativeInteger) -> Union(%, "failed")
--R ?.? : (% , Fraction(%)) -> Fraction(%) if R has INTDOM
--R elt : (Fraction(%), R) -> R if R has FIELD
--R elt : (Fraction(%), Fraction(%)) -> Fraction(%) if R has INTDOM
--R enumerate : () -> List(%) if R has FINITE
--R euclideanSize : % -> NonNegativeInteger if R has FIELD
--R eval : (% , List(SingletonAsOrderedSet), List(%)) -> %
--R eval : (% , SingletonAsOrderedSet, %) -> %
--R eval : (% , List(SingletonAsOrderedSet), List(R)) -> %
--R eval : (% , SingletonAsOrderedSet, R) -> %
--R eval : (% , List(%), List(%)) -> %
--R eval : (% , List(Equation(%))) -> %
--R expressIdealMember : (List(%), %) -> Union(List(%), "failed") if R has FIELD
--R exquo : (% , %) -> Union(%, "failed") if R has INTDOM
--R exquo : (% , R) -> Union(%, "failed") if R has INTDOM
--R extendedEuclidean : (% , %) -> Record(coef1: %, coef2: %, generator: %) if R has FIELD
--R extendedEuclidean : (% , %, %) -> Union(Record(coef1: %, coef2: %), "failed") if R has FIELD
--R factor : % -> Factored(%) if R has PFECAT
--R factorPolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if R has PFECAT
--R factorSquareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if R has PFECAT
--R frobenius : % -> % if R has FFIELDC
--R gcd : (% , %) -> % if R has GCDDOM
--R gcd : List(%) -> % if R has GCDDOM
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%) if R has GCDDOM
--R index : PositiveInteger -> % if R has FINITE
--R integrate : % -> % if R has ALGEBRA(FRAC(INT))
--R isExpt : % -> Union(Record(var: SingletonAsOrderedSet, exponent: NonNegativeInteger), "failed")
--R isPlus : % -> Union(List(%), "failed")
--R isTimes : % -> Union(List(%), "failed")
--R karatsubaDivide : (% , NonNegativeInteger) -> Record(quotient: %, remainder: %)
--R lcm : (% , %) -> % if R has GCDDOM
--R lcm : List(%) -> % if R has GCDDOM
--R lookup : % -> PositiveInteger if R has FINITE
--R mainVariable : % -> Union(SingletonAsOrderedSet, "failed")
--R makeSUP : % -> SparseUnivariatePolynomial(R)
--R mapExponents : ((NonNegativeInteger -> NonNegativeInteger), %) -> %
--R max : (% , %) -> % if R has ORDSET
--R min : (% , %) -> % if R has ORDSET
--R minimumDegree : (% , List(SingletonAsOrderedSet)) -> List(NonNegativeInteger)
--R minimumDegree : (% , SingletonAsOrderedSet) -> NonNegativeInteger
--R minimumDegree : % -> NonNegativeInteger
--R monicDivide : (% , %) -> Record(quotient: %, remainder: %)
--R monicDivide : (% , %, SingletonAsOrderedSet) -> Record(quotient: %, remainder: %)
--R monomial : (% , List(SingletonAsOrderedSet), List(NonNegativeInteger)) -> %
--R monomial : (% , SingletonAsOrderedSet, NonNegativeInteger) -> %
--R monomial : (R, NonNegativeInteger) -> %
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed") if R has FIELD

```



```

--R multiplyExponents : (% , NonNegativeInteger) -> %
--R multivariate : (SparseUnivariatePolynomial(%), SingletonAsOrderedSet) -> %
--R multivariate : (SparseUnivariatePolynomial(R), SingletonAsOrderedSet) -> %
--R nextItem : % -> Union(% , "failed") if R has STEP
--R numberOfMonomials : % -> NonNegativeInteger
--R order : (% , %) -> NonNegativeInteger if R has INTDOM
--R patternMatch : (% , Pattern(Integer), PatternMatchResult(Integer, %)) -> PatternMatchResult(Integer, %)
--R patternMatch : (% , Pattern(Float), PatternMatchResult(Float, %)) -> PatternMatchResult(Float, %)
--R pomopo! : (% , R, NonNegativeInteger, %) -> %
--R prime? : % -> Boolean if R has PFECAT
--R primitiveMonomials : % -> List(%)
--R primitivePart : (% , SingletonAsOrderedSet) -> % if R has GCDDOM
--R primitivePart : % -> % if R has GCDDOM
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %) if R has FIELD
--R pseudoDivide : (% , %) -> Record(coef: R, quotient: %, remainder: %) if R has INTDOM
--R pseudoQuotient : (% , %) -> % if R has INTDOM
--R ?quo? : (% , %) -> % if R has FIELD
--R random : () -> % if R has FINITE
--R reducedSystem : Matrix(%) -> Matrix(R)
--R reducedSystem : (Matrix(%), Vector(%)) -> Record(mat: Matrix(R), vec: Vector(R))
--R reducedSystem : (Matrix(%), Vector(%)) -> Record(mat: Matrix(Integer), vec: Vector(Integer)) if R has LINEXP(INT)
--R reducedSystem : Matrix(%) -> Matrix(Integer) if R has LINEXP(INT)
--R ?rem? : (% , %) -> % if R has FIELD
--R resultant : (% , %) -> R if R has COMRING
--R resultant : (% , %, SingletonAsOrderedSet) -> % if R has COMRING
--R retract : % -> SingletonAsOrderedSet
--R retract : % -> Integer if R has RETRACT(INT)
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(SingletonAsOrderedSet, "failed")
--R retractIfCan : % -> Union(Integer, "failed") if R has RETRACT(INT)
--R retractIfCan : % -> Union(Fraction(Integer), "failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(R, "failed")
--R separate : (% , %) -> Record(primePart: %, commonPart: %) if R has GCDDOM
--R shiftLeft : (% , NonNegativeInteger) -> %
--R shiftRight : (% , NonNegativeInteger) -> %
--R size : () -> NonNegativeInteger if R has FINITE
--R sizeLess? : (% , %) -> Boolean if R has FIELD
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(%)), SparseUnivariatePolynomial(%)) -> %
--R squareFree : % -> Factored(%) if R has GCDDOM
--R squareFreePart : % -> % if R has GCDDOM
--R squareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if R has GCDDOM
--R subResultantGcd : (% , %) -> % if R has INTDOM
--R subtractIfCan : (% , %) -> Union(% , "failed")
--R totalDegree : (% , List(SingletonAsOrderedSet)) -> NonNegativeInteger
--R totalDegree : % -> NonNegativeInteger
--R unit? : % -> Boolean if R has INTDOM
--R unitCanonical : % -> % if R has INTDOM
--R unitNormal : % -> Record(unit: %, canonical: %, associate: %) if R has INTDOM
--R univariate : % -> SparseUnivariatePolynomial(R)
--R univariate : (% , SingletonAsOrderedSet) -> SparseUnivariatePolynomial(%)

```

```
--R unmakeSUP : SparseUnivariatePolynomial(R) -> %  
--R variables : % -> List(SingletonAsOrderedSet)  
--R vectorise : (% ,NonNegativeInteger) -> Vector(R)  
--R  
--E 1  
  
)spool  
)lisp (bye)
```

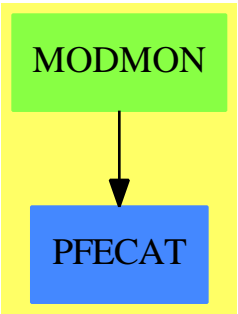
— ModMonic.help —

=====
ModMonic examples
=====

This package has not been documented

See Also:
o)show ModMonic

ModMonic (MODMON)



Exports:

0	1	An	associates?
binomThmExpt	characteristic	charthRoot	coefficient
coefficients	coerce	composite	computePowers
conditionP	content	convert	D
degree	differentiate	discriminant	divide
divideExponents	elt	euclideanSize	eval
expressIdealMember	exquo	extendedEuclidean	factor
factorPolynomial	factorSquareFreePolynomial	frobenius	gcd
gcdPolynomial	ground	ground?	hash
index	init	integrate	isExpt
isPlus	isTimes	karatsubaDivide	latex
lcm	leadingCoefficient	leadingMonomial	lift
lookup	mainVariable	makeSUP	map
mapExponents	max	min	minimumDegree
modulus	monicDivide	monomial	monomial?
monomials	multiEuclidean	multiplyExponents	multivariate
nextItem	numberOfMonomials	one?	order
patternMatch	pomopo!	pow	prime?
primitiveMonomials	primitivePart	principalIdeal	pseudoDivide
pseudoQuotient	pseudoRemainder	random	recip
reduce	reducedSystem	reductum	resultant
retract	retractIfCan	sample	separate
setPoly	shiftLeft	shiftRight	size
sizeLess?	solveLinearPolynomialEquation	squareFree	squareFreePart
squareFreePolynomial	subResultantGcd	subtractIfCan	totalDegree
unit?	unitCanonical	unitNormal	univariate
unmakeSUP	UnVectorise	variables	vectorise
Vectorise	zero?	?*?	?**?
?+?	?-?	-?	?=?
?^?	?.??	?~=?	?/?
?<?	?<=?	?>?	?>=?
?quo?	?rem?		

— domain MODMON ModMonic —

```
)abbrev domain MODMON ModMonic
++ Author: Mark Botch
++ Description:
++ This package has not been documented
-- following line prevents caching ModMonic
)bo PUSH('ModMonic, $mutableDomains)
```

```
ModMonic(R,Rep): C == T
where
  R: Ring
  Rep: UnivariatePolynomialCategory(R)
  C == UnivariatePolynomialCategory(R) with
```

```

--operations
setPoly : Rep -> Rep
  ++ setPoly(x) is not documented
modulus : -> Rep
  ++ modulus() is not documented
reduce: Rep -> %
  ++ reduce(x) is not documented
lift: % -> Rep --reduce lift = identity
  ++ lift(x) is not documented
coerce: Rep -> %
  ++ coerce(x) is not documented
Vectorise: % -> Vector(R)
  ++ Vectorise(x) is not documented
UnVectorise: Vector(R) -> %
  ++ UnVectorise(v) is not documented
An: % -> Vector(R)
  ++ An(x) is not documented
pow : -> PrimitiveArray(%)
  ++ pow() is not documented
computePowers : -> PrimitiveArray(%)
  ++ computePowers() is not documented
if R has FiniteFieldCategory then
  frobenius: % -> %
  ++ frobenius(x) is not documented
--LinearTransf: (% , Vector(R)) -> SquareMatrix<deg> R
--assertions
if R has Finite then Finite
T == add
--constants
m:Rep := monomial(1,1)$Rep --| degree(m) > 0 and LeadingCoef(m) = R$1
d := degree(m)$Rep
d1 := (d-1):NonNegativeInteger
twod := 2*d1+1
frobenius?:Boolean := R has FiniteFieldCategory
--VectorRep:= DirectProduct(d:NonNegativeInteger,R)
--declarations
x,y: %
p: Rep
d,n: Integer
e,k1,k2: NonNegativeInteger
c: R
--vect: Vector(R)
power:PrimitiveArray(%)
frobeniusPower:PrimitiveArray(%)
computeFrobeniusPowers : () -> PrimitiveArray(%)
--representations
--mutable m    --take this out??
--define
power := new(0,0)
frobeniusPower := new(0,0)

```

```

setPoly (mon : Rep) ==
  mon = $Rep m => mon
  oldm := m
  leadingCoefficient mon ^= 1 => error "polynomial must be monic"
  -- following copy code needed since FFPOLY can modify mon
  copymon:Rep:= 0
  while not zero? mon repeat
    copymon := monomial(leadingCoefficient mon, degree mon)$Rep + copymon
    mon := reductum mon
  m := copymon
  d := degree(m)$Rep
  d1 := (d-1)::NonNegativeInteger
  twod := 2*d1+1
  power := computePowers()
  if frobenius? then
    degree(oldm)>1 and not((oldm exquo$Rep m) case "failed") =>
      for i in 1..d1 repeat
        frobeniusPower(i) := reduce lift frobeniusPower(i)
      frobeniusPower := computeFrobeniusPowers()
  m
modulus == m
if R has Finite then
  size == d * size$R
  random == UnVectorise([random()$R for i in 0..d1])
0 == 0$Rep
1 == 1$Rep
c * x == c * $Rep x
n * x == (n::R) * $Rep x
coerce(c:R):% == monomial(c,0)$Rep
coerce(x:%):OutputForm == coerce(x)$Rep
coefficient(x,e):R == coefficient(x,e)$Rep
reductum(x) == reductum(x)$Rep
leadingCoefficient x == (leadingCoefficient x)$Rep
degree x == (degree x)$Rep
lift(x) == x pretend Rep
reduce(p) == (monicDivide(p,m)$Rep).remainder
coerce(p) == reduce(p)
x = y == x = $Rep y
x + y == x + $Rep y
- x == -$Rep x
x * y ==
  p := x * $Rep y
  ans:=0$Rep
  while (n:=degree p)>d1 repeat
    ans:=ans + leadingCoefficient(p)*power.(n-d)
    p := reductum p
  ans+p
Vectorise(x) == [coefficient(lift(x),i) for i in 0..d1]
UnVectorise(vect) ==
  reduce(+/[monomial(vect.(i+1),i) for i in 0..d1])

```

```

computePowers ==
  mat : PrimitiveArray(%) := new(d,0)
  mat.0 := reductum(-m)$Rep
  w: % := monomial$Rep (1,1)
  for i in 1..d1 repeat
    mat.i := w *$Rep mat.(i-1)
    if degree mat.i=d then
      mat.i := reductum mat.i + leadingCoefficient mat.i * mat.0
  mat
if frobenius? then
  computeFrobeniusPowers() ==
    mat : PrimitiveArray(%) := new(d,1)
    mat.1 := mult := monomial(1, size$R)$%
    for i in 2..d1 repeat
      mat.i := mult * mat.(i-1)
    mat

frobenius(a:%):% ==
  aq:% := 0
  while a^=0 repeat
    aq := aq + leadingCoefficient(a)*frobeniusPower(degree a)
    a := reductum a
  aq

pow == power
monomial(c,e)==
  if e<d then monomial(c,e)$Rep
  else
    if e<=twod then
      c * power.(e-d)
    else
      k1:=e quo twod
      k2 := (e-k1*twod)::NonNegativeInteger
      reduce((power.d1 **k1)*monomial(c,k2))
if R has Field then

  (x:% exquo y:%):Union(%, "failed") ==
    uv := extendedEuclidean(y, modulus(), x)$Rep
    uv case "failed" => "failed"
    return reduce(uv.coef1)

  recip(y:%):Union(%, "failed") == 1 exquo y
  divide(x:%, y:%) ==
    (q := (x exquo y)) case "failed" => error "not divisible"
    [q, 0]

-- An(MM) == Vectorise(-(reduce(reductum(m))::MM))
-- LinearTransf(vect,MM) ==
--   ans:= 0::SquareMatrix<d>(R)
--   for i in 1..d do setelt(ans,i,1,vect.i)

```

```
--      for j in 2..d do
--          setelt(ans,1,j, elt(ans,d,j-1) * An(MM).1)
--          for i in 2..d do
--              setelt(ans,i,j, elt(ans,i-1,j-1) + elt(ans,d,j-1) * An(MM).i)
--      ans
```

— MODMON.dotabb —

```
"MODMON" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MODMON"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MODMON" -> "PFECAT"
```

domain MODFIELD ModularField

— ModularField.input —

```
)set break resume
)sys rm -f ModularField.output
)spool ModularField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ModularField
--R
--R ModularField(R: CommutativeRing,Mod: AbelianMonoid,reduction: ((R,Mod) -> R),merge: ((Mod,Mo
--R Abbreviation for ModularField is MODFIELD
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for MODFIELD
--R
--R----- Operations -----
--R ?? : (Fraction(Integer),%) -> %      ?? : (%,Fraction(Integer)) -> %
--R ?? : (%,%) -> %                      ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %        ??? : (%,Integer) -> %
--R ??? : (%,PositiveInteger) -> %      ?+? : (%,%) -> %
--R ?-? : (%,%) -> %                     -? : % -> %
--R ?/? : (%,%) -> %                     ?=? : (%,%) -> Boolean
--R 1 : () -> %                          0 : () -> %
--R ?? : (%,Integer) -> %                ?? : (%,PositiveInteger) -> %
```

```

--R associates? : (%,% ) -> Boolean
--R coerce : Fraction(Integer) -> %
--R coerce : Integer -> %
--R factor : % -> Factored(%)
--R gcd : (%,% ) -> %
--R inv : % -> %
--R lcm : List(%) -> %
--R modulus : % -> Mod
--R prime? : % -> Boolean
--R recip : % -> Union(%, "failed")
--R ?rem? : (%,% ) -> %
--R sizeLess? : (%,% ) -> Boolean
--R squareFreePart : % -> %
--R unitCanonical : % -> %
--R ?~=? : (%,% ) -> Boolean
--R ?*? : (NonNegativeInteger,% ) -> %
--R ??? : (% ,NonNegativeInteger) -> %
--R ?^? : (% ,NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R divide : (%,% ) -> Record(quotient: %,remainder: %)
--R euclideanSize : % -> NonNegativeInteger
--R exQuo : (%,% ) -> Union(%, "failed")
--R expressIdealMember : (List(%),%) -> Union(List(%), "failed")
--R exquo : (%,% ) -> Union(%, "failed")
--R extendedEuclidean : (%,% ,%) -> Union(Record(coef1: %,coef2: %), "failed")
--R extendedEuclidean : (%,% ) -> Record(coef1: %,coef2: %,generator: %)
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R multiEuclidean : (List(%),%) -> Union(List(%), "failed")
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %)
--R subtractIfCan : (%,% ) -> Union(%, "failed")
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R
--E 1

)spool
)lisp (bye)

```

— ModularField.help —

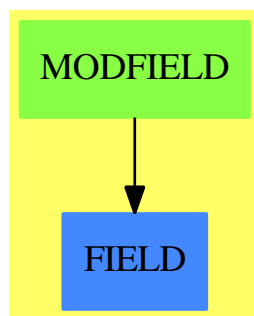
ModularField examples

These domains are used for the factorization and gcds of univariate polynomials over the integers in order to work modulo different primes.

See Also:

- o)show ModularField
- o)show ModularRing
- o)show EuclideanModularRing

ModularField (MODFIELD)



See

⇒ “ModularRing” (MODRING) [14](#) on page [1641](#)

⇒ “EuclideanModularRing” (EMR) [6](#) on page [701](#)

Exports:

0	1	associates?	characteristic	coerce
divide	euclideanSize	expressIdealMember	exquo	exQuo
extendedEuclidean	factor	gcd	gcdPolynomial	hash
inv	latex	lcm	modulus	multiEuclidean
one?	prime?	principalIdeal	recip	reduce
sample	sizeLess?	squareFree	squareFreePart	subtractIfCan
unit?	unitCanonical	unitNormal	zero?	?*?
?**?	?+?	?-?	-?	?/?
?=?	?^?	?~=?	?quo?	?rem?

— domain MODFIELD ModularField —

```

)abbrev domain MODFIELD ModularField
++ Author: Mark Botch
++ Description:
++ These domains are used for the factorization and gcds
++ of univariate polynomials over the integers in order to work modulo
++ different primes.
++ See \spadtype{ModularRing}, \spadtype{EuclideanModularRing}

```

```

ModularField(R,Mod,reduction:(R,Mod) -> R,

```

```

merge: (Mod, Mod) -> Union(Mod, "failed"),
exactQuo : (R, R, Mod) -> Union(R, "failed")) : C == T
where
R      : CommutativeRing
Mod    : AbelianMonoid

C == Field with
modulus: % -> Mod
++ modulus(x) is not documented
coerce: % -> R
++ coerce(x) is not documented
reduce: (R, Mod) -> %
++ reduce(r, m) is not documented
exQuo: (% , %) -> Union(%, "failed")
++ exQuo(x, y) is not documented

T == ModularRing(R, Mod, reduction, merge, exactQuo)

```

— MODFIELD.dotabb —

```

"MODFIELD" [color="#88FF44", href="bookvol10.3.pdf#nameddest=MODFIELD"]
"FIELD"    [color="#4488FF", href="bookvol10.2.pdf#nameddest=FIELD"]
"MODFIELD" -> "FIELD"

```

domain MODRING ModularRing

— ModularRing.input —

```

)set break resume
)sys rm -f ModularRing.output
)spool ModularRing.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ModularRing
--R ModularRing(R: CommutativeRing, Mod: AbelianMonoid, reduction: ((R, Mod) -> R), merge: ((Mod, Mod) -> Union(
--R Abbreviation for ModularRing is MODRING
--R This constructor is not exposed in this frame.

```

```

--R Issue )edit bookvol10.3.pamphlet to see algebra source code for MODRING
--R
--R----- Operations -----
--R ?? : (%,%) -> %                ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R +? : (%,%) -> %                ?-? : (%,%) -> %
--R -? : % -> %                    ?=? : (%,%) -> Boolean
--R 1 : () -> %                    0 : () -> %
--R ^? : (%,PositiveInteger) -> %   coerce : % -> R
--R coerce : Integer -> %           coerce : % -> OutputForm
--R hash : % -> SingleInteger       inv : % -> %
--R latex : % -> String             modulus : % -> Mod
--R one? : % -> Boolean             recip : % -> Union(%, "failed")
--R reduce : (R,Mod) -> %          sample : () -> %
--R zero? : % -> Boolean            ?~=? : (%,%) -> Boolean
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R ^? : (%,NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R exQuo : (%,%) -> Union(%, "failed")
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)

```

— ModularRing.help —

=====

ModularRing examples

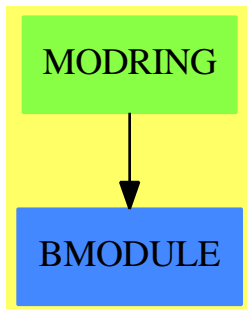
=====

These domains are used for the factorization and gcds
of univariate polynomials over the integers in order to work modulo
different primes.

See Also:

- o)show ModularRing
 - o)show EuclideanModularRing
 - o)show ModularField
-

ModularRing (MODRING)



See

⇒ “EuclideanModularRing” (EMR) [6](#) on page [701](#)

⇒ “ModularField” (MODFIELD) [14](#) on page [1638](#)

Exports:

0	1	characteristic	coerce	exQuo
hash	inv	latex	modulus	one?
recip	reduce	sample	subtractIfCan	zero?
?~=?	?*?	?**?	?^?	?+?
?-?	-?	?=?		

— domain MODRING ModularRing —

```

)abbrev domain MODRING ModularRing
++ Author: P.Gianni, B.Trager
++ Description:
++ These domains are used for the factorization and gcds
++ of univariate polynomials over the integers in order to work modulo
++ different primes.
++ See \spadtype{EuclideanModularRing} ,\spadtype{ModularField}

ModularRing(R,Mod,reduction:(R,Mod) -> R,
            merge:(Mod,Mod) -> Union(Mod,"failed"),
            exactQuo : (R,R,Mod) -> Union(R,"failed")) : C == T

where
  R      : CommutativeRing
  Mod    : AbelianMonoid

  C == Ring with
    modulus: % -> Mod
      ++ modulus(x) is not documented
    coerce: % -> R
      ++ coerce(x) is not documented
    reduce: (R,Mod) -> %
      ++ reduce(r,m) is not documented
    exQuo: (%,%)-> Union(%, "failed")

```

```

    ++ exQuo(x,y) is not documented
    recip: % -> Union(%, "failed")
    ++ recip(x) is not documented
    inv: % -> %
    ++ inv(x) is not documented

T == add
--representation
  Rep:= Record(val:R,module:Mod)
--declarations
  x,y: %

--define
  modulus(x)    == x.module
  coerce(x)     == x.val
  coerce(i:Integer):% == [i::R,0]$Rep
  i:Integer * x:% == (i::%)*x
  coerce(x):OutputForm == (x.val)::OutputForm
  reduce (a:R,m:Mod) == [reduction(a,m),m]$Rep

  characteristic():NonNegativeInteger == characteristic()$R
  0 == [0$R,0$Mod]$Rep
  1 == [1$R,0$Mod]$Rep
  zero? x == zero? x.val
--  one? x == one? x.val
  one? x == (x.val = 1)

  newmodulo(m1:Mod,m2:Mod) : Mod ==
    r:=merge(m1,m2)
    r case "failed" => error "incompatible moduli"
    r::Mod

  x=y ==
    x.val = y.val => true
    x.module = y.module => false
    (x-y).val = 0
  x+y == reduce((x.val +$R y.val),newmodulo(x.module,y.module))
  x-y == reduce((x.val -$R y.val),newmodulo(x.module,y.module))
  -x == reduce ((-$R x.val),x.module)
  x*y == reduce((x.val *$R y.val),newmodulo(x.module,y.module))

  exQuo(x,y) ==
    xm:=x.module
    if xm ^=$Mod y.module then xm:=newmodulo(xm,y.module)
    r:=exactQuo(x.val,y.val,xm)
    r case "failed"=> "failed"
    [r::R,xm]$Rep

--if R has EuclideanDomain then
  recip x ==

```

```

r:=exactQuo(1$R,x.val,x.modulo)
r case "failed" => "failed"
[r,x.modulo]$Rep

inv x ==
  if (u:=recip x) case "failed" then error("not invertible")
  else u::%

```

— MODRING.dotabb —

```

"MODRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MODRING"]
"BMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BMODULE"]
"MODRING" -> "BMODULE"

```

domain MODMONOM ModuleMonomial

— ModuleMonomial.input —

```

)set break resume
)sys rm -f ModuleMonomial.output
)spool ModuleMonomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ModuleMonomial
--R ModuleMonomial(IS: OrderedSet,E: SetCategory,ff: ((Record(index: IS,exponent: E),Record(index: IS,exponent: E)),Record(index: IS,exponent: E)))
--R Abbreviation for ModuleMonomial is MODMONOM
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for MODMONOM
--R
--R----- Operations -----
--R ?<? : (%,% ) -> Boolean          ?<=? : (%,% ) -> Boolean
--R ?=? : (%,% ) -> Boolean          ?>? : (%,% ) -> Boolean
--R ?>=? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R construct : (IS,E) -> %           exponent : % -> E
--R hash : % -> SingleInteger         index : % -> IS
--R latex : % -> String               max : (%,% ) -> %
--R min : (%,% ) -> %                 ?~? : (%,% ) -> Boolean

```

```

--R coerce : % -> Record(index: IS,exponent: E)
--R coerce : Record(index: IS,exponent: E) -> %
--R
--E 1

)spool
)lisp (bye)

```

— ModuleMonomial.help —

```

=====
ModuleMonomial examples
=====

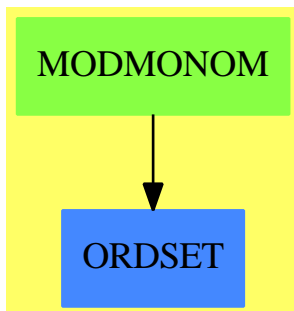
```

This package has no documentation

See Also:

o)show ModuleMonomial

ModuleMonomial (MODMONOM)



See

⇒ “GeneralModulePolynomial” (GMODPOL) 8 on page [1059](#)

Exports:

coerce	construct	exponent	hash	index
latex	max	min	?~=?	?<?
?<=?	?=?	?>?	?>=?	

— domain MODMONOM ModuleMonomial —

```

)abbrev domain MODMONOM ModuleMonomial
++ Author: Mark Botch
++ Description:
++ This package has no documentation

ModuleMonomial(IS: OrderedSet,
               E: SetCategory,
               ff:(MM, MM) -> Boolean): T == C where

  MM ==> Record(index:IS, exponent:E)

  T == OrderedSet with
    exponent: $ -> E
    ++ exponent(x) is not documented
    index: $ -> IS
    ++ index(x) is not documented
    coerce: MM -> $
    ++ coerce(x) is not documented
    coerce: $ -> MM
    ++ coerce(x) is not documented
    construct: (IS, E) -> $
    ++ construct(i,e) is not documented
  C == MM add
    Rep:= MM
    x:$ < y:$ == ff(x::Rep, y::Rep)
    exponent(x:$):E == x.exponent
    index(x:$): IS == x.index
    coerce(x:$):MM == x::Rep::MM
    coerce(x:MM):$ == x::Rep::$
    construct(i:IS, e:E):$ == [i, e]$MM::Rep::$

```

— MODMONOM.dotabb —

```

"MODMONOM" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MODMONOM"]
"ORDSET" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ORDSET"]
"MODMONOM" -> "ORDSET"

```

domain MODOP ModuleOperator

— ModuleOperator.input —


```

)set break resume
)sys rm -f ModuleOperator.output
)spool ModuleOperator.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ModuleOperator
--R
--R ModuleOperator(R: Ring,M: LeftModule(R)) is a domain constructor
--R Abbreviation for ModuleOperator is MODOP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for MODOP
--R
--R----- Operations -----
--R ???: (% ,%) -> %
--R ??? : (PositiveInteger,% ) -> %
--R ??? : (% ,PositiveInteger) -> %
--R ?-? : (% ,%) -> %
--R ?=? : (% ,%) -> Boolean
--R 0 : () -> %
--R coerce : BasicOperator -> %
--R coerce : Integer -> %
--R ?..? : (% ,M) -> M
--R hash : % -> SingleInteger
--R one? : % -> Boolean
--R recip : % -> Union(% ,"failed")
--R retract : % -> R
--R zero? : % -> Boolean
--R ???: (% ,R) -> % if R has COMRING
--R ???: (R,% ) -> % if R has COMRING
--R ???: (NonNegativeInteger,% ) -> %
--R ??? : (BasicOperator,Integer) -> %
--R ??? : (% ,NonNegativeInteger) -> %
--R ?^? : (% ,NonNegativeInteger) -> %
--R adjoint : (% ,%) -> % if R has COMRING
--R adjoint : % -> % if R has COMRING
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(% ,"failed") if R has CHARNZ
--R conjug : R -> R if R has COMRING
--R evaluateInverse : (% ,(M -> M)) -> %
--R makeop : (R,FreeGroup(BasicOperator)) -> %
--R retractIfCan : % -> Union(BasicOperator ,"failed")
--R retractIfCan : % -> Union(R ,"failed")
--R subtractIfCan : (% ,%) -> Union(% ,"failed")
--R
--E 1

)spool

```

)lisp (bye)

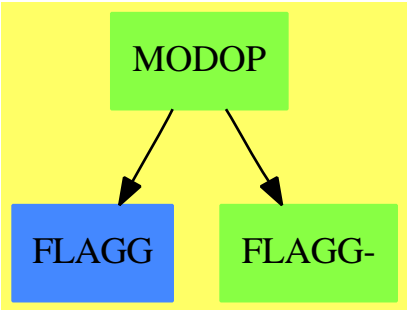
— ModuleOperator.help —

=====
ModuleOperator examples
=====

Algebra of ADDITIVE operators on a module.

See Also:
o)show ModuleOperator

ModuleOperator (MODOP)



See

⇒ “Operator” (OP) [16](#) on page [1804](#)

Exports:

0	1	adjoint	characteristic	charthRoot
coerce	conjug	evaluate	evaluateInverse	hash
latex	makeop	one?	opeval	recip
retract	retractIfCan	sample	subtractIfCan	zero?
?*?	?**?	?+?	?-?	-?
?=?	?^?	?..?	?~=?	

— domain MODOP ModuleOperator —

)abbrev domain MODOP ModuleOperator
++ Author: Manuel Bronstein
++ Date Created: 15 May 1990

```

++ Date Last Updated: 17 June 1993
++ Description:
++ Algebra of ADDITIVE operators on a module.

ModuleOperator(R: Ring, M:LeftModule(R)): Exports == Implementation where
  O    ==> OutputForm
  OP   ==> BasicOperator
  FG   ==> FreeGroup OP
  RM   ==> Record(coef:R, monom:FG)
  TERM ==> List RM
  FAB  ==> FreeAbelianGroup TERM
  OPADJ ==> "%opAdjoint"
  OPEVAL ==> "%opEval"
  INVEVAL ==> "%invEval"

Exports ==> Join(Ring, RetractableTo R, RetractableTo OP,
  Eltable(M, M)) with
  if R has CharacteristicZero then CharacteristicZero
  if R has CharacteristicNonZero then CharacteristicNonZero
  if R has CommutativeRing then
    Algebra(R)
    adjoint: $ -> $
      ++ adjoint(op) returns the adjoint of the operator \spad{op}.
    adjoint: ($, $) -> $
      ++ adjoint(op1, op2) sets the adjoint of op1 to be op2.
      ++ op1 must be a basic operator
    conjug : R -> R
      ++ conjug(x) should be local but conditional
    evaluate: ($, M -> M) -> $
      ++ evaluate(f, u +-> g u) attaches the map g to f.
      ++ f must be a basic operator
      ++ g MUST be additive, i.e. \spad{g(a + b) = g(a) + g(b)} for
      ++ any \spad{a}, \spad{b} in M.
      ++ This implies that \spad{g(n a) = n g(a)} for
      ++ any \spad{a} in M and integer \spad{n > 0}.
    evaluateInverse: ($, M -> M) -> $
      ++ evaluateInverse(x,f) is not documented
    "***": (OP, Integer) -> $
      ++ op**n is not documented
    "***": ($, Integer) -> $
      ++ op**n is not documented
    opeval : (OP, M) -> M
      ++ opeval should be local but conditional
    makeop : (R, FG) -> $
      ++ makeop should be local but conditional

Implementation ==> FAB add
import NoneFunctions1($)
import BasicOperatorFunctions1(M)

```

```

Rep := FAB

inv      : TERM -> $
termeval : (TERM, M) -> M
rmeval   : (RM, M) -> M
monomeval : (FG, M) -> M
opInvEval : (OP, M) -> M
mkop     : (R, FG) -> $
termprod0 : (Integer, TERM, TERM) -> $
termprod  : (Integer, TERM, TERM) -> TERM
termcopy  : TERM -> TERM
trm20     : (Integer, TERM) -> 0
term20    : TERM -> 0
rm20      : (R, FG) -> 0
nocopy    : OP -> $

1 == makeop(1, 1)
coerce(n:Integer):$ == n::R::$
coerce(r:R):$ == (zero? r => 0; makeop(r, 1))
coerce(op:OP):$ == nocopy copy op
nocopy(op:OP):$ == makeop(1, op::FG)
elt(x:$, r:M) == +/[t.exp * termeval(t.gen, r) for t in terms x]
rmeval(t, r) == t.coef * monomeval(t.monom, r)
termcopy t == [[rm.coef, rm.monom] for rm in t]
characteristic() == characteristic()$R
mkop(r, fg) == [[r, fg]$RM]$TERM :: $
evaluate(f, g) == nocopy setProperty(retract(f)@OP,OPEVAL,g pretend None)

if R has OrderedSet then
  makeop(r, fg) == (r >= 0 => mkop(r, fg); - mkop(-r, fg))
else makeop(r, fg) == mkop(r, fg)

inv(t:TERM):$ ==
  empty? t => 1
  c := first(t).coef
  m := first(t).monom
  inv(rest t) * makeop(1, inv m) * (recip(c)::R::$)

x:$ ** i:Integer ==
  i = 0 => 1
  i > 0 => expt(x,i pretend PositiveInteger)$RepeatedSquaring($
    (inv(retract(x)@TERM)) ** (-i))

evaluateInverse(f, g) ==
  nocopy setProperty(retract(f)@OP, INVEVAL, g pretend None)

coerce(x:$):0 ==
  zero? x => (0$R)::0
  reduce(_+, [trm20(t.exp, t.gen) for t in terms x])$List(0)

```

```

    trm20(c, t) ==
--      one? c => term20 t
      (c = 1) => term20 t
      c = -1 => - term20 t
      c::0 * term20 t

    term20 t ==
      reduce(*, [rm20(rm.coef, rm.monom) for rm in t])$List(0)

    rm20(c, m) ==
--      one? c => m::0
      (c = 1) => m::0
--      one? m => c::0
      (m = 1) => c::0
      c::0 * m::0

    x:$ * y:$ ==
      +/[ +/[termprod0(t.exp * s.exp, t.gen, s.gen) for s in terms y]
        for t in terms x]

    termprod0(n, x, y) ==
      n >= 0 => termprod(n, x, y)::$
      - (termprod(-n, x, y)::$)

    termprod(n, x, y) ==
      lc := first(xx := termcopy x)
      lc.coef := n * lc.coef
      rm := last xx
--      one?(first(y).coef) =>
      ((first(y).coef) = 1) =>
        rm.monom := rm.monom * first(y).monom
        concat_!(xx, termcopy rest y)
--      one?(rm.monom) =>
      ((rm.monom) = 1) =>
        rm.coef := rm.coef * first(y).coef
        rm.monom := first(y).monom
        concat_!(xx, termcopy rest y)
      concat_!(xx, termcopy y)

    if M has ExpressionSpace then
      opeval(op, r) ==
        (func := property(op, OPEVAL)) case "failed" => kernel(op, r)
        ((func::None) pretend (M -> M)) r
    else
      opeval(op, r) ==
        (func := property(op, OPEVAL)) case "failed" =>
          error "eval: operator has no evaluation function"
        ((func::None) pretend (M -> M)) r

```

```

opInvEval(op, r) ==
  (func := property(op, INVEVAL)) case "failed" =>
    error "eval: operator has no inverse evaluation function"
  ((func::None) pretend (M -> M)) r

termeval(t, r) ==
  for rm in reverse t repeat r := rmeval(rm, r)
  r

monomeval(m, r) ==
  for rec in reverse_! factors m repeat
    e := rec.exp
    g := rec.gen
    e > 0 =>
      for i in 1..e repeat r := opeval(g, r)
    e < 0 =>
      for i in 1..(-e) repeat r := opInvEval(g, r)
  r

recip x ==
  (r := retractIfCan(x)@Union(R, "failed")) case "failed" => "failed"
  (r1 := recip(r::R)) case "failed" => "failed"
  r1::R::$

retractIfCan(x:$):Union(R, "failed") ==
  (r := retractIfCan(x)@Union(TERM, "failed")) case "failed" => "failed"
  empty?(t := r::TERM) => 0$R
  empty? rest t =>
    rm := first t
--    one?(rm.monom) => rm.coef
    (rm.monom = 1) => rm.coef
    "failed"
    "failed"

retractIfCan(x:$):Union(OP, "failed") ==
  (r := retractIfCan(x)@Union(TERM, "failed")) case "failed" => "failed"
  empty?(t := r::TERM) => "failed"
  empty? rest t =>
    rm := first t
--    one?(rm.coef) => retractIfCan(rm.monom)
    (rm.coef = 1) => retractIfCan(rm.monom)
    "failed"
    "failed"

if R has CommutativeRing then
  termadj : TERM -> $
  rmadj   : RM -> $
  monomadj : FG -> $
  opadj    : OP -> $

```

```

r:R * x:$      == r::$ * x
x:$ * r:R      == x * (r::$)
adjoint x      == +/[t.exp * termadj(t.gen) for t in terms x]
rmadj t        == conjug(t.coef) * monomadj(t.monom)
adjoint(op, adj) == nocopy setProperty(retract(op)@OP, OPADJ, adj::None)

termadj t ==
  ans:$ := 1
  for rm in t repeat ans := rmadj(rm) * ans
  ans

monomadj m ==
  ans:$ := 1
  for rec in factors m repeat ans := (opadj(rec.gen) ** rec.exp) * ans
  ans

opadj op ==
  (adj := property(op, OPADJ)) case "failed" =>
    error "adjoint: operator does not have a defined adjoint"
  (adj::None) pretend $

if R has conjugate:R -> R then conjug r == conjugate r else conjug r == r

```

— MODOP.dotabb —

```

"MODOP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MODOP"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"MODOP" -> "FLAGG"
"MODOP" -> "FLAGG-"

```

domain MOEBIUS MoebiusTransform

— MoebiusTransform.input —

```

)set break resume
)sys rm -f MoebiusTransform.output
)spool MoebiusTransform.output
)set message test on
)set message auto off

```

```

)clear all

--S 1 of 1
)show MoebiusTransform
--R
--R MoebiusTransform(F: Field) is a domain constructor
--R Abbreviation for MoebiusTransform is MOEBIUS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for MOEBIUS
--R
--R----- Operations -----
--R ??? : (%,% ) -> %          ??? : (%,Integer) -> %
--R ??? : (%,PositiveInteger) -> %    ?/? : (%,% ) -> %
--R ?? : (%,% ) -> Boolean          1 : () -> %
--R ?? : (%,Integer) -> %          ?? : (%,PositiveInteger) -> %
--R coerce : % -> OutputForm      commutator : (%,% ) -> %
--R conjugate : (%,% ) -> %        eval : (% ,F) -> F
--R hash : % -> SingleInteger      inv : % -> %
--R latex : % -> String            moebius : (F,F,F,F) -> %
--R one? : % -> Boolean            recip : % -> %
--R recip : () -> %                recip : % -> Union(%, "failed")
--R sample : () -> %              scale : (% ,F) -> %
--R scale : F -> %                shift : (% ,F) -> %
--R shift : F -> %                ?~=? : (%,% ) -> Boolean
--R ??? : (% ,NonNegativeInteger) -> %
--R ?? : (% ,NonNegativeInteger) -> %
--R eval : (% ,OnePointCompletion(F)) -> OnePointCompletion(F)
--R
--E 1

)spool
)lisp (bye)

```

— MoebiusTransform.help —

```

=====
MoebiusTransform examples
=====

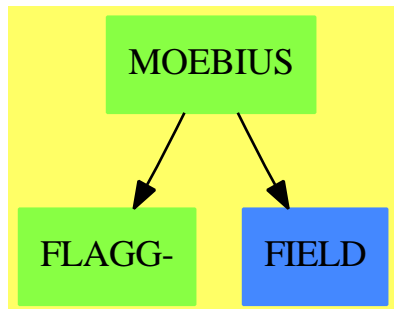
```

MoebiusTransform(F) is the domain of fractional linear (Moebius) transformations over F. This a domain of 2-by-2 matrices acting on P1(F).

See Also:

o)show MoebiusTransform

MoebiusTransform (MOEBIUS)



Exports:

1	coerce	commutator	conjugate	eval
hash	inv	latex	moebius	one?
recip	sample	scale	shift	?~=?
?**?	?^?	?*?	?/?	?=?
?^?				

— domain MOEBIUS MoebiusTransform —

```

)abbrev domain MOEBIUS MoebiusTransform
++ Author: Stephen "Say" Watt
++ Date Created: January 1987
++ Date Last Updated: 11 April 1990
++ Description:
++ MoebiusTransform(F) is the domain of fractional linear (Moebius)
++ transformations over F. This a domain of 2-by-2 matrices acting on P1(F).

```

```

MoebiusTransform(F): Exports == Implementation where

```

```

  F : Field
  OUT ==> OutputForm
  P1F ==> OnePointCompletion F          -- projective 1-space over F

```

```

Exports ==> Group with

```

```

moebius: (F,F,F,F) -> %
  ++ moebius(a,b,c,d) returns \spad{matrix [[a,b],[c,d]]}.
shift: F -> %
  ++ shift(k) returns \spad{matrix [[1,k],[0,1]]} representing the map
  ++ \spad{x -> x + k}.
scale: F -> %
  ++ scale(k) returns \spad{matrix [[k,0],[0,1]]} representing the map
  ++ \spad{x -> k * x}.
recip: () -> %
  ++ recip() returns \spad{matrix [[0,1],[1,0]]} representing the map
  ++ \spad{x -> 1 / x}.

```

```

shift: (%,F) -> %
  ++ shift(m,h) returns \spad{shift(h) * m}
  ++ (see shift from MoebiusTransform).
scale: (%,F) -> %
  ++ scale(m,h) returns \spad{scale(h) * m}
  ++ (see shift from MoebiusTransform).
recip: % -> %
  ++ recip(m) = recip() * m
eval: (%,F) -> F
  ++ eval(m,x) returns \spad{(a*x + b)/(c*x + d)}
  ++ where \spad{m = moebius(a,b,c,d)}
  ++ (see moebius from MoebiusTransform).
eval: (%,P1F) -> P1F
  ++ eval(m,x) returns \spad{(a*x + b)/(c*x + d)}
  ++ where \spad{m = moebius(a,b,c,d)}
  ++ (see moebius from MoebiusTransform).

Implementation ==> add

Rep := Record(a: F,b: F,c: F,d: F)

moebius(aa,bb,cc,dd) == [aa,bb,cc,dd]

a(t:%):F == t.a
b(t:%):F == t.b
c(t:%):F == t.c
d(t:%):F == t.d

1 == moebius(1,0,0,1)
t * s ==
  moebius(b(t)*c(s) + a(t)*a(s), b(t)*d(s) + a(t)*b(s), _
          d(t)*c(s) + c(t)*a(s), d(t)*d(s) + c(t)*b(s))
inv t == moebius(d(t),-b(t),-c(t),a(t))

shift f == moebius(1,f,0,1)
scale f == moebius(f,0,0,1)
recip() == moebius(0,1,1,0)

shift(t,f) == moebius(a(t) + f*c(t), b(t) + f*d(t), c(t), d(t))
scale(t,f) == moebius(f*a(t),f*b(t),c(t),d(t))
recip t == moebius(c(t),d(t),a(t),b(t))

eval(t:%,f:F) == (a(t)*f + b(t))/(c(t)*f + d(t))
eval(t:%,f:P1F) ==
  (ff := retractIfCan(f)@Union(F,"failed")) case "failed" =>
    (a(t)/c(t)) :: P1F
  zero?(den := c(t) * (fff := ff :: F) + d(t)) => infinity()
  ((a(t) * fff + b(t))/den) :: P1F

coerce t ==

```

```

var := "%x" :: OUT
num := (a(t) :: OUT) * var + (b(t) :: OUT)
den := (c(t) :: OUT) * var + (d(t) :: OUT)
rarrow(var,num/den)

proportional?: (List F,List F) -> Boolean
proportional?(list1,list2) ==
  empty? list1 => empty? list2
  empty? list2 => false
  zero? (x1 := first list1) =>
    (zero? first list2) and proportional?(rest list1,rest list2)
  zero? (x2 := first list2) => false
  map((f1:F):F +-> f1/x1, list1) = map((g1:F):F +-> g1/x2, list2)

t = s ==
  list1 : List F := [a(t),b(t),c(t),d(t)]
  list2 : List F := [a(s),b(s),c(s),d(s)]
  proportional?(list1,list2)

```

— MOEBIUS.dotabb —

```

"MOEBIUS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MOEBIUS"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"MOEBIUS" -> "FIELD"
"MOEBIUS" -> "FLAGG-"

```

domain MRING MonoidRing

— MonoidRing.input —

```

)set break resume
)sys rm -f MonoidRing.output
)spool MonoidRing.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MonoidRing

```

```

--R MonoidRing(R: Ring,M: Monoid) is a domain constructor
--R Abbreviation for MonoidRing is MRING
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for MRING
--R
--R----- Operations -----
--R ?? : (%,% ) -> %           ?? : (Integer,% ) -> %
--R ?? : (PositiveInteger,% ) -> %   ??? : (% ,PositiveInteger) -> %
--R ?? : (%,% ) -> %           ?-? : (%,% ) -> %
--R -? : % -> %               ?? : (%,% ) -> Boolean
--R 1 : () -> %               0 : () -> %
--R ?? : (% ,PositiveInteger) -> %   coefficient : (% ,M) -> R
--R coefficients : % -> List(R)       coerce : R -> %
--R coerce : M -> %                 coerce : Integer -> %
--R coerce : % -> OutputForm         hash : % -> SingleInteger
--R latex : % -> String              map : ((R -> R),%) -> %
--R monomial : (R,M) -> %            monomial? : % -> Boolean
--R monomials : % -> List(%)         one? : % -> Boolean
--R recip : % -> Union(%, "failed")   retract : % -> R
--R retract : % -> M                 sample : () -> %
--R zero? : % -> Boolean              ?~=? : (% ,%) -> Boolean
--R ?? : (% ,R) -> % if R has COMRING
--R ?? : (R,% ) -> % if R has COMRING
--R ?? : (NonNegativeInteger,% ) -> %
--R ??? : (% ,NonNegativeInteger) -> %
--R ?? : (% ,NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if R has CHARNZ
--R coerce : List(Record(coef: R, monom: M)) -> %
--R enumerate : () -> List(%) if M has FINITE and R has FINITE
--R index : PositiveInteger -> % if M has FINITE and R has FINITE
--R leadingCoefficient : % -> R if M has ORDSET
--R leadingMonomial : % -> M if M has ORDSET
--R lookup : % -> PositiveInteger if M has FINITE and R has FINITE
--R numberOfMonomials : % -> NonNegativeInteger
--R random : () -> % if M has FINITE and R has FINITE
--R reductum : % -> % if M has ORDSET
--R retractIfCan : % -> Union(R, "failed")
--R retractIfCan : % -> Union(M, "failed")
--R size : () -> NonNegativeInteger if M has FINITE and R has FINITE
--R subtractIfCan : (% ,%) -> Union(%, "failed")
--R terms : % -> List(Record(coef: R, monom: M))
--R
--E 1

)spool
)lisp (bye)

```

— MonoidRing.help —

=====

MonoidRing examples

=====

MonoidRing(R,M), implements the algebra of all maps from the monoid M to the commutative ring R with finite support.

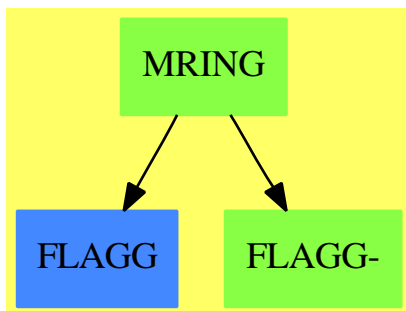
Multiplication of two maps f and g is defined to map an element c of M to the (convolution) sum over f(a)g(b) such that ab = c. Thus M can be identified with a canonical basis and the maps can also be considered as formal linear combinations of the elements in M. Scalar multiples of a basis element are called monomials. A prominent example is the class of polynomials where the monoid is a direct product of the natural numbers with pointwise addition. When M is FreeMonoid Symbol, one gets polynomials in infinitely many non-commuting variables. Another application area is representation theory of finite groups G, where modules over MonoidRing(R,G) are studied.

See Also:

o)show MonoidRing

—————>

MonoidRing (MRING)



Exports:

0	1	characteristic	charthRoot	coefficient
coefficients	coerce	hash	index	latex
leadingCoefficient	leadingMonomial	lookup	map	monomial
monomial?	monomials	numberOfMonomials	one?	random
recip	reductum	retract	retractIfCan	sample
size	subtractIfCan	terms	zero?	??
??*	?+?	?-?	-?	==?
?^?	?^=?			

— domain MRING MonoidRing —

```

)abbrev domain MRING MonoidRing
++ Authors: Stephan M. Watt; revised by Johannes Grabmeier
++ Date Created: January 1986
++ Date Last Updated: 14 December 1995, Mike Dewar
++ Description:
++ \spadtype{MonoidRing}(R,M), implements the algebra
++ of all maps from the monoid M to the commutative ring R with
++ finite support.
++ Multiplication of two maps f and g is defined
++ to map an element c of M to the (convolution) sum over f(a)g(b)
++ such that ab = c. Thus M can be identified with a canonical
++ basis and the maps can also be considered as formal linear combinations
++ of the elements in M. Scalar multiples of a basis element are called
++ monomials. A prominent example is the class of polynomials
++ where the monoid is a direct product of the natural numbers
++ with pointwise addition. When M is
++ \spadtype{FreeMonoid Symbol}, one gets polynomials
++ in infinitely many non-commuting variables. Another application
++ area is representation theory of finite groups G, where modules
++ over \spadtype{MonoidRing}(R,G) are studied.

MonoidRing(R: Ring, M: Monoid): MRcategory == MRdefinition where
  Term ==> Record(coef: R, monom: M)

MRcategory ==> Join(Ring, RetractableTo M, RetractableTo R) with
  monomial      : (R, M) -> %
  ++ monomial(r,m) creates a scalar multiple of the basis element m.
  coefficient : (% , M) -> R
  ++ coefficient(f,m) extracts the coefficient of m in f with respect
  ++ to the canonical basis M.
  coerce:      List Term -> %
  ++ coerce(lt) converts a list of terms and
  ++ coefficients to a member of the domain.
  terms        : % -> List Term
  ++ terms(f) gives the list of non-zero coefficients combined
  ++ with their corresponding basis element as records.
  ++ This is the internal representation.
  map          : (R -> R, %) -> %

```

```

    ++ map(fn,u) maps function fn onto the coefficients
    ++ of the non-zero monomials of u.
monomial?    : % -> Boolean
    ++ monomial?(f) tests if f is a single monomial.
coefficients: % -> List R
    ++ coefficients(f) lists all non-zero coefficients.
monomials: % -> List %
    ++ monomials(f) gives the list of all monomials whose
    ++ sum is f.
numberOfMonomials: % -> NonNegativeInteger
    ++ numberOfMonomials(f) is the number of non-zero coefficients
    ++ with respect to the canonical basis.
if R has CharacteristicZero then CharacteristicZero
if R has CharacteristicNonZero then CharacteristicNonZero
if R has CommutativeRing then Algebra(R)
if (R has Finite and M has Finite) then Finite
if M has OrderedSet then
  leadingMonomial : % -> M
    ++ leadingMonomial(f) gives the monomial of f whose
    ++ corresponding monoid element is the greatest
    ++ among all those with non-zero coefficients.
  leadingCoefficient: % -> R
    ++ leadingCoefficient(f) gives the coefficient of f, whose
    ++ corresponding monoid element is the greatest
    ++ among all those with non-zero coefficients.
  reductum          : % -> %
    ++ reductum(f) is f minus its leading monomial.

MRdefinition ==> add
  Ex ==> OutputForm
  Cf ==> coef
  Mn ==> monom

  Rep := List Term

  coerce(x: List Term): % == x :: %

  monomial(r:R, m:M) ==
    r = 0 => empty()
    [[r, m]]

  if (R has Finite and M has Finite) then
    size() == size()$R ** size()$M

  index k ==
    -- use p-adic decomposition of k
    -- coefficient of p**j determines coefficient of index(i+p)$M
    i:Integer := k rem size()
    p:Integer := size()$R
    n:Integer := size()$M

```

```

ans:% := 0
for j in 0.. while i > 0 repeat
  h := i rem p
  -- we use index(p) = 0$R
  if h ^= 0 then
    c : R := index(h :: PositiveInteger)$R
    m : M := index((j+n) :: PositiveInteger)$M
    --ans := ans + c *$% m
    ans := ans + monomial(c, m)$%
  i := i quo p
ans

lookup(z : %) : PositiveInteger ==
-- could be improved, if M has OrderedSet
-- z = index lookup z, n = lookup index n
-- use p-adic decomposition of k
-- coefficient of p**j determines coefficient of index(i+p)$M
zero?(z) => size()$% pretend PositiveInteger
liTe : List Term := terms z -- all non-zero coefficients
p : Integer := size()$R
n : Integer := size()$M
res : Integer := 0
for te in liTe repeat
  -- assume that lookup(p)$R = 0
  l:NonNegativeInteger:=lookup(te.Mn)$M
  ex : NonNegativeInteger := (n=l => 0;1)
  co : Integer := lookup(te.Cf)$R
  res := res + co * p ** ex
res pretend PositiveInteger

random() == index( (1+(random())$Integer rem size()$%) )_
pretend PositiveInteger)$%

0 == empty()
1 == [[1, 1]]
terms a == (copy a) pretend List(Term)
monomials a == [[t] for t in a]
coefficients a == [t.Cf for t in a]
coerce(m:M):% == [[1, m]]
coerce(r:R): % ==
-- coerce of ring
r = 0 => 0
[[r, 1]]
coerce(n:Integer): % ==
-- coerce of integers
n = 0 => 0
[[n::R, 1]]
- a == [[ -t.Cf, t.Mn] for t in a]
if R has noZeroDivisors
then

```



```

(r:R) * (a:%) ==
  r = 0 => 0
  [[r*t.Cf, t.Mn] for t in a]
else
  (r:R) * (a:%) ==
    r = 0 => 0
    [[rt, t.Mn] for t in a | (rt:=r*t.Cf) ^= 0]
if R has noZeroDivisors
  then
    (n:Integer) * (a:%) ==
      n = 0 => 0
      [[n*t.Cf, t.Mn] for t in a]
  else
    (n:Integer) * (a:%) ==
      n = 0 => 0
      [[nt, t.Mn] for t in a | (nt:=n*t.Cf) ^= 0]
map(f, a) == [[ft, t.Mn] for t in a | (ft:=f(t.Cf)) ^= 0]
numberOfMonomials a == #a

retractIfCan(a:%):Union(M, "failed") ==
--   one?(#a) and one?(a.first.Cf) => a.first.Mn
   ((#a) = 1) and ((a.first.Cf) = 1) => a.first.Mn
   "failed"

retractIfCan(a:%):Union(R, "failed") ==
--   one?(#a) and one?(a.first.Mn) => a.first.Cf
   ((#a) = 1) and ((a.first.Mn) = 1) => a.first.Cf
   "failed"

if R has noZeroDivisors then
  if M has Group then
    recip a ==
      lt := terms a
      #lt ^= 1 => "failed"
      (u := recip lt.first.Cf) case "failed" => "failed"
      --(u::R) * inv lt.first.Mn
      monomial((u::R), inv lt.first.Mn)%
    else
      recip a ==
        #a ^= 1 or a.first.Mn ^= 1 => "failed"
        (u := recip a.first.Cf) case "failed" => "failed"
        u::R::%

mkTerm(r:R, m:M):Ex ==
  r=1 => m::Ex
  r=0 or m=1 => r::Ex
  r::Ex * m::Ex

coerce(a:%):Ex ==
  empty? a => (0$Integer)::Ex

```

```

empty? rest a => mkTerm(a.first.Cf, a.first.Mn)
reduce(_+, [mkTerm(t.Cf, t.Mn) for t in a])$List(Ex)

if M has OrderedSet then -- we mean totally ordered
-- Terms are stored in decending order.
leadingCoefficient a == (empty? a => 0; a.first.Cf)
leadingMonomial a    == (empty? a => 1; a.first.Mn)
reductum a           == (empty? a => a; rest a)

a = b ==
  #a ^= #b => false
  for ta in a for tb in b repeat
    ta.Cf ^= tb.Cf or ta.Mn ^= tb.Mn => return false
  true

a + b ==
  c:% := empty()
  while not empty? a and not empty? b repeat
    ta := first a; tb := first b
    ra := rest a;  rb := rest b
    c :=
      ta.Mn > tb.Mn => (a := ra; concat_!(c, ta))
      ta.Mn < tb.Mn => (b := rb; concat_!(c, tb))
      a := ra; b := rb
      not zero?(r := ta.Cf+tb.Cf) =>
        concat_!(c, [r, ta.Mn])
    c
  concat_!(c, concat(a, b))

coefficient(a, m) ==
  for t in a repeat
    if t.Mn = m then return t.Cf
    if t.Mn < m then return 0
  0

if M has OrderedMonoid then

-- we use that multiplying an ordered list of monoid elements
-- by a single element respects the ordering

if R has noZeroDivisors then
  a:% * b:% ==
    +/[[[ta.Cf*tb.Cf, ta.Mn*tb.Mn]$Term
      for tb in b ] for ta in reverse a]
else
  a:% * b:% ==
    +/[[[r, ta.Mn*tb.Mn]$Term
      for tb in b | not zero?(r := ta.Cf*tb.Cf)]
      for ta in reverse a]

```

```

else -- M hasn't OrderedMonoid

-- we cannot assume that mutiplying an ordered list of
-- monoid elements by a single element respects the ordering:
-- we have to order and to collect equal terms
ge : (Term,Term) -> Boolean
ge(s,t) == t.Mn <= s.Mn

sortAndAdd : List Term -> List Term
sortAndAdd(liTe) == -- assume liTe not empty
  liTe := sort(ge,liTe)
  m : M := (first liTe).Mn
  cf : R := (first liTe).Cf
  res : List Term := []
  for te in rest liTe repeat
    if m = te.Mn then
      cf := cf + te.Cf
    else
      if not zero? cf then res := cons([cf,m]$Term, res)
      m := te.Mn
      cf := te.Cf
  if not zero? cf then res := cons([cf,m]$Term, res)
  reverse res

if R has noZeroDivisors then
  a:% * b:% ==
    zero? a => a
    zero? b => b -- avoid calling sortAndAdd with []
    +/[sortAndAdd [[ta.Cf*tb.Cf, ta.Mn*tb.Mn]$Term
      for tb in b ] for ta in reverse a]
else
  a:% * b:% ==
    zero? a => a
    zero? b => b -- avoid calling sortAndAdd with []
    +/[sortAndAdd [[r, ta.Mn*tb.Mn]$Term
      for tb in b | not zero?(r := ta.Cf*tb.Cf)]
      for ta in reverse a]

else -- M hasn't OrderedSet
  -- Terms are stored in random order.
  a = b ==
    #a ^= #b => false
    brace(a pretend List(Term)) =$Set(Term) brace(b pretend List(Term))

coefficient(a, m) ==
  for t in a repeat
    t.Mn = m => return t.Cf
0

```

```

addterm(Tabl: AssociationList(M,R), r:R, m:M):R ==
  (u := search(m, Tabl)) case "failed" => Tabl.m := r
  zero?(r := r + u::R) => (remove_!(m, Tabl); 0)
  Tabl.m := r

a + b ==
  Tabl := table()$AssociationList(M,R)
  for t in a repeat
    Tabl t.Mn := t.Cf
  for t in b repeat
    addterm(Tabl, t.Cf, t.Mn)
  [[Tabl m, m]$Term for m in keys Tabl]

a:% * b:% ==
  Tabl := table()$AssociationList(M,R)
  for ta in a repeat
    for tb in (b pretend List(Term)) repeat
      addterm(Tabl, ta.Cf*tb.Cf, ta.Mn*tb.Mn)
  [[Tabl.m, m]$Term for m in keys Tabl]

```

— MRING.dotabb —

```

"MRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MRING"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"MRING" -> "FLAGG"
"MRING" -> "FLAGG-"

```

domain MSET Multiset

— Multiset.input —

```

)set break resume
)sys rm -f Multiset.output
)spool Multiset.output
)set message test on
)set message auto off
)clear all
--S 1 of 14

```

```

s := multiset [1,2,3,4,5,4,3,2,3,4,5,6,7,4,10]
--R
--R
--R (1) {1,2: 2,3: 3,4: 4,2: 5,6,7,10}
--R
--R                                         Type: Multiset(PositiveInteger)
--E 1

--S 2 of 14
insert!(3,s)
--R
--R
--R (2) {1,2: 2,4: 3,4: 4,2: 5,6,7,10}
--R
--R                                         Type: Multiset(PositiveInteger)
--E 2

--S 3 of 14
remove!(3,s,1)
--R
--R
--R (3) {1,2: 2,3: 3,4: 4,2: 5,6,7,10}
--R
--R                                         Type: Multiset(PositiveInteger)
--E 3

--S 4 of 14
s
--R
--R
--R (4) {1,2: 2,3: 3,4: 4,2: 5,6,7,10}
--R
--R                                         Type: Multiset(PositiveInteger)
--E 4

--S 5 of 14
remove!(5,s)
--R
--R
--R (5) {1,2: 2,3: 3,4: 4,6,7,10}
--R
--R                                         Type: Multiset(PositiveInteger)
--E 5

--S 6 of 14
s
--R
--R
--R (6) {1,2: 2,3: 3,4: 4,6,7,10}
--R
--R                                         Type: Multiset(PositiveInteger)
--E 6

--S 7 of 14
count(5,s)
--R

```

```

--R
--R (7)  0
--R
--R                                         Type: NonNegativeInteger
--E 7

--S 8 of 14
t := multiset [2,2,2,-9]
--R
--R
--R (8)  {3: 2,- 9}
--R
--R                                         Type: Multiset(Integer)
--E 8

--S 9 of 14
U := union(s,t)
--R
--R
--R (9)  {1,5: 2,3: 3,4: 4,6,7,10,- 9}
--R
--R                                         Type: Multiset(Integer)
--E 9

--S 10 of 14
I := intersect(s,t)
--R
--R
--R (10) {5: 2}
--R
--R                                         Type: Multiset(Integer)
--E 10

--S 11 of 14
difference(s,t)
--R
--R
--R (11) {1,3: 3,4: 4,6,7,10}
--R
--R                                         Type: Multiset(Integer)
--E 11

--S 12 of 14
S := symmetricDifference(s,t)
--R
--R
--R (12) {1,3: 3,4: 4,6,7,10,- 9}
--R
--R                                         Type: Multiset(Integer)
--E 12

--S 13 of 14
(U = union(S,I))@Boolean
--R
--R
--R (13) true

```

```

--R                                                    Type: Boolean
--E 13

--S 14 of 14
t1 := multiset [1,2,2,3]; [t1 < t, t1 < s, t < s, t1 <= s]
--R
--R
--R (14) [false,true,false,true]
--R                                                    Type: List(Boolean)
--E 14
)spool
)lisp (bye)

```

— Multiset.help —

=====

Multiset examples

=====

The domain Multiset(R) is similar to Set(R) except that multiplicities (counts of duplications) are maintained and displayed. Use the operation multiset to create multisets from lists. All the standard operations from sets are available for multisets. An element with multiplicity greater than one has the multiplicity displayed first, then a colon, and then the element.

Create a multiset of integers.

```

s := multiset [1,2,3,4,5,4,3,2,3,4,5,6,7,4,10]
{1,2: 2,3: 3,4: 4,2: 5,6,7,10}
Type: Multiset PositiveInteger

```

The operation insert! adds an element to a multiset.

```

insert!(3,s)
{1,2: 2,4: 3,4: 4,2: 5,6,7,10}
Type: Multiset PositiveInteger

```

Use remove! to remove an element. If a third argument is present, it specifies how many instances to remove. Otherwise all instances of the element are removed. Display the resulting multiset.

```

remove!(3,s,1); s
{1,2: 2,3: 3,4: 4,2: 5,6,7,10}
Type: Multiset PositiveInteger

```

```
remove!(5,s); s
{1,2: 2,3: 3,4: 4,6,7,10}
Type: Multiset PositiveInteger
```

The operation count returns the number of copies of a given value.

```
count(5,s)
0
Type: NonNegativeInteger
```

A second multiset.

```
t := multiset [2,2,2,-9]
{3: 2,- 9}
Type: Multiset Integer
```

The union of two multisets is additive.

```
U := union(s,t)
{1,5: 2,3: 3,4: 4,6,7,10,- 9}
Type: Multiset Integer
```

The intersect operation gives the elements that are in common, with additive multiplicity.

```
I := intersect(s,t)
{5: 2}
Type: Multiset Integer
```

The difference of s and t consists of the elements that s has but t does not. Elements are regarded as indistinguishable, so that if s and t have any element in common, the difference does not contain that element.

```
difference(s,t)
{1,3: 3,4: 4,6,7,10}
Type: Multiset Integer
```

The symmetricDifference is the union of difference(s, t) and difference(t, s).

```
S := symmetricDifference(s,t)
{1,3: 3,4: 4,6,7,10,- 9}
Type: Multiset Integer
```

Check that the union of the symmetricDifference and the intersect equals the union of the elements.

```
(U = union(S,I))@Boolean
true
Type: Boolean
```


Check some inclusion relations.

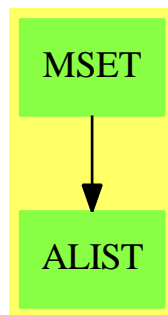
```
t1 := multiset [1,2,2,3]; [t1 < t, t1 < s, t < s, t1 <= s]
[false,true,false,true]
Type: List Boolean
```

See Also:

o)show Multiset

—————→

Multiset (MSET)



Exports:

any?	bag	brace	coerce	construct
convert	copy	count	dictionary	difference
duplicates	empty	empty?	eq?	eval
every?	extract!	find	hash	insert!
inspect	intersect	latex	less?	map
map!	members	member?	more?	multiset
parts	reduce	remove	remove!	removeDuplicates
sample	select	select!	set	size?
subset?	symmetricDifference	union	#?	?<?
?=?	?~=?			

— domain MSET Multiset —

```
)abbrev domain MSET Multiset
```

```
++ Author:Stephen M. Watt, William H. Burge, Richard D. Jenks, Frederic Lehouey
```

```
++ Date Created:NK
```

```
++ Date Last Updated: 14 June 1994
```

```
++ Description:
```

++ A multiset is a set with multiplicities.

```

Multiset(S: SetCategory): MultisetAggregate S with
  finiteAggregate
  shallowlyMutable
  multiset: () -> %
    ++ multiset()$D creates an empty multiset of domain D.
  multiset: S -> %
    ++ multiset(s) creates a multiset with singleton s.
  multiset: List S -> %
    ++ multiset(ls) creates a multiset with elements from \spad{ls}.
  members: % -> List S
    ++ members(ms) returns a list of the elements of \spad{ms}
    ++ without their multiplicity. See also \spadfun{parts}.
  remove: (S,%,Integer) -> %
    ++ remove(x,ms,number) removes at most \spad{number} copies of
    ++ element x if \spad{number} is positive, all of them if
    ++ \spad{number} equals zero, and all but at most \spad{-number} if
    ++ \spad{number} is negative.
  remove: ( S -> Boolean ,%,Integer) -> %
    ++ remove(p,ms,number) removes at most \spad{number} copies of
    ++ elements x such that \spad{p(x)} is \spadfun{true}
    ++ if \spad{number} is positive, all of them if
    ++ \spad{number} equals zero, and all but at most \spad{-number} if
    ++ \spad{number} is negative.
  remove_!: (S,%,Integer) -> %
    ++ remove!(x,ms,number) removes destructively at most \spad{number}
    ++ copies of element x if \spad{number} is positive, all
    ++ of them if \spad{number} equals zero, and all but at most
    ++ \spad{-number} if \spad{number} is negative.
  remove_!: ( S -> Boolean ,%,Integer) -> %
    ++ remove!(p,ms,number) removes destructively at most \spad{number}
    ++ copies of elements x such that \spad{p(x)} is
    ++ \spadfun{true} if \spad{number} is positive, all of them if
    ++ \spad{number} equals zero, and all but at most \spad{-number} if
    ++ \spad{number} is negative.

== add

Tbl ==> Table(S, Integer)
tbl ==> table$Tbl
Rep := Record(count: Integer, table: Tbl)

n: Integer
ms, m1, m2: %
t, t1, t2: Tbl
D ==> Record(entry: S, count: NonNegativeInteger)
K ==> Record(key: S, entry: Integer)

elt(t:Tbl, s:S):Integer ==

```

```

a := search(s,t)$Tbl
a case "failed" => 0
a::Integer

empty():% == [0,tbl()]
multiset():% == empty()
dictionary():% == empty() -- DictionaryOperations
set():% == empty()
brace():% == empty()

construct(l:List S):% ==
  t := tbl()
  n := 0
  for e in l repeat
    t.e := inc t.e
    n := inc n
  [n, t]
multiset(l:List S):% == construct l
bag(l:List S):% == construct l -- BagAggregate
dictionary(l:List S):% == construct l -- DictionaryOperations
set(l:List S):% == construct l
brace(l:List S):% == construct l

multiset(s:S):% == construct [s]

if S has ConvertibleTo InputForm then
  convert(ms:%):InputForm ==
    convert [convert("multiset":Symbol)@InputForm,
      convert(parts ms)@InputForm]

members(ms:%):List S == keys ms.table

coerce(ms:%):OutputForm ==
  l: List OutputForm := empty()
  t := ms.table
  colon := ": " :: OutputForm
  for e in keys t repeat
    ex := e::OutputForm
    n := t.e
    item :=
      n > 1 => hconcat [n :: OutputForm,colon, ex]
      ex
    l := cons(item,l)
  brace l

duplicates(ms:%):List D == -- MultiDictionary
ld : List D := empty()
t := ms.table
for e in keys t | (n := t.e) > 1 repeat
  ld := cons([e,n::NonNegativeInteger],ld)

```

```

ld

extract_!(ms:%):S ==                -- BagAggregate
  empty? ms => error "extract: Empty multiset"
  ms.count := dec ms.count
  t := ms.table
  e := inspect(t).key
  if (n := t.e) > 1 then t.e := dec n
  else remove_!(e,t)
  e

inspect(ms:%):S == inspect(ms.table).key -- BagAggregate

insert_!(e:S,ms:%):% ==              -- BagAggregate
  ms.count := inc ms.count
  ms.table.e := inc ms.table.e
  ms

member?(e:S,ms:%):Boolean == member?(e,keys ms.table)

empty?(ms:%):Boolean == ms.count = 0

#(ms:%):NonNegativeInteger == ms.count::NonNegativeInteger

count(e:S, ms:%):NonNegativeInteger == ms.table.e::NonNegativeInteger

remove_!(e:S, ms:%, max:Integer):% ==
  zero? max => remove_!(e,ms)
  t := ms.table
  if member?(e, keys t) then
    ((n := t.e) <= max) =>
      remove_!(e,t)
      ms.count := ms.count-n
  max > 0 =>
    t.e := n-max
    ms.count := ms.count-max
  (n := n+max) > 0 =>
    t.e := -max
    ms.count := ms.count-n
  ms

remove_!(p: S -> Boolean, ms:%, max:Integer):% ==
  zero? max => remove_!(p,ms)
  t := ms.table
  for e in keys t | p(e) repeat
    ((n := t.e) <= max) =>
      remove_!(e,t)
      ms.count := ms.count-n
  max > 0 =>
    t.e := n-max

```

```

        ms.count := ms.count-max
        (n := n+max) > 0 =>
            t.e := -max
        ms.count := ms.count-n
    ms

remove(e:S, ms:%, max:Integer):% == remove_!(e, copy ms, max)

remove(p: S -> Boolean, ms:%, max:Integer):% == remove_!(p, copy ms, max)

remove_!(e:S, ms:%):% == -- DictionaryOperations
    t := ms.table
    if member?(e, keys t) then
        ms.count := ms.count-t.e
        remove_!(e, t)
    ms

remove_!(p:S ->Boolean, ms:%):% == -- DictionaryOperations
    t := ms.table
    for e in keys t | p(e) repeat
        ms.count := ms.count-t.e
        remove_!(e, t)
    ms

select_!(p: S -> Boolean, ms:%):% == -- DictionaryOperations
    remove_!((s1:S):Boolean+>not p(s1), ms)

removeDuplicates_!(ms:%):% == -- MultiDictionary
    t := ms.table
    l := keys t
    for e in l repeat t.e := 1
    ms.count := #l
    ms

insert_!(e:S, ms:%, more:NonNegativeInteger):% == -- MultiDictionary
    ms.count := ms.count+more
    ms.table.e := ms.table.e+more
    ms

map_!(f: S->S, ms:%):% == -- HomogeneousAggregate
    t := ms.table
    t1 := tbl()
    for e in keys t repeat
        t1.f(e) := t.e
        remove_!(e, t)
    ms.table := t1
    ms

map(f: S -> S, ms:%):% == map_!(f, copy ms) -- HomogeneousAggregate

```

```

parts(m:%):List S ==
  l := empty()$List(S)
  t := m.table
  for e in keys t repeat
    for i in 1..t.e repeat
      l := cons(e,l)
  l

union(m1:%, m2:%):% ==
  t := tbl()
  t1:= m1.table
  t2:= m2.table
  for e in keys t1 repeat t.e := t1.e
  for e in keys t2 repeat t.e := t2.e + t.e
  [m1.count + m2.count, t]

intersect(m1:%, m2:%):% ==
--   if #m1 > #m2 then intersect(m2, m1)
  t := tbl()
  t1:= m1.table
  t2:= m2.table
  n := 0
  for e in keys t1 repeat
    m := min(t1.e,t2.e)
    m > 0 =>
      m := t1.e + t2.e
      t.e := m
      n := n + m
  [n, t]

difference(m1:%, m2:%):% ==
  t := tbl()
  t1:= m1.table
  t2:= m2.table
  n := 0
  for e in keys t1 repeat
    k1 := t1.e
    k2 := t2.e
    k1 > 0 and k2 = 0 =>
      t.e := k1
      n := n + k1
  n = 0 => empty()
  [n, t]

symmetricDifference(m1:%, m2:%):% ==
  union(difference(m1,m2), difference(m2,m1))

m1 = m2 ==
  m1.count ^= m2.count => false
  t1 := m1.table

```

```

t2 := m2.table
for e in keys t1 repeat
    t1.e ^= t2.e => return false
for e in keys t2 repeat
    t1.e ^= t2.e => return false
true

m1 < m2 ==
m1.count >= m2.count => false
t1 := m1.table
t2 := m2.table
for e in keys t1 repeat
    t1.e > t2.e => return false
m1.count < m2.count

subset?(m1:%, m2:%):Boolean ==
m1.count > m2.count => false
t1 := m1.table
t2 := m2.table
for e in keys t1 repeat t1.e > t2.e => return false
true

```

— MSET.dotabb —

```

"MSET" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MSET"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"MSET" -> "ALIST"

```

domain MPOLY MultivariatePolynomial

— MultivariatePolynomial.input —

```

)set break resume
)sys rm -f MultivariatePolynomial.output
)spool MultivariatePolynomial.output
)set message test on
)set message auto off
)clear all
--S 1 of 9
m : MPOLY([x,y],INT) := (x^2 - x*y^3 +3*y)^2

```

```

--R
--R
--R      4      3 3      6      2      4      2
--R  (1)  x  - 2y x  + (y  + 6y)x  - 6y x + 9y
--R                                          Type: MultivariatePolynomial([x,y],Integer)
--E 1

--S 2 of 9
m :: MPOLY([y,x],INT)
--R
--R
--R      2 6      4      3 3      2      2      4
--R  (2)  x y  - 6x y  - 2x y  + 9y  + 6x y + x
--R                                          Type: MultivariatePolynomial([y,x],Integer)
--E 2

--S 3 of 9
p : MPOLY([x,y],POLY INT)
--R
--R
--R                                          Type: Void
--E 3

--S 4 of 9
p :: POLY INT
--R
--R
--R  (4)  p
--R                                          Type: Polynomial(Integer)
--E 4

--S 5 of 9
% :: MPOLY([a,b],POLY INT)
--R
--R
--R  (5)  p
--R                                          Type: MultivariatePolynomial([a,b],Polynomial(Integer))
--E 5

--S 6 of 9
q : UP(x, FRAC MPOLY([y,z],INT))
--R
--R
--R                                          Type: Void
--E 6

--S 7 of 9
q := (x^2 - x*(z+1)/y + 2)^2
--R
--R
--R      2      2
--R      4      - 2z - 2  3  4y  + z  + 2z + 1  2  - 4z - 4

```



```

--R      (7)  x + ----- x + ----- x + ----- x + 4
--R              y              2              y
--R
--RType: UnivariatePolynomial(x,Fraction(MultivariatePolynomial([y,z],Integer)))
--E 7

```

```

--S 8 of 9
q :: UP(z, FRAC MPOLY([x,y],INT))
--R
--R
--R      (8)
--R      2      3      2      2 4      3      2      2      2
--R      x  x  - 2y x  + 2x  - 4y x  y x  - 2y x  + (4y  + 1)x  - 4y x  + 4y
--R      -- z  + ----- z + -----
--R      2      2      2
--R      y      y      y
--RType: UnivariatePolynomial(z,Fraction(MultivariatePolynomial([x,y],Integer)))
--E 8

```

```

--S 9 of 9
q :: MPOLY([x,z], FRAC UP(y,INT))
--R
--R
--R      2
--R      4      2      2 3      1 2      2      4y  + 1 2      4      4
--R      (9)  x  + (- - z - -)x  + (- - z  + - - z  + -----)x  + (- - z - -)x  + 4
--R              y      y      2      2      2      y      y
--R              y      y      y
--RType: MultivariatePolynomial([x,z],Fraction(UnivariatePolynomial(y,Integer)))
--E 9
)spool
)lisp (bye)

```

— MultivariatePolynomial.help —

=====

MultivariatePolynomial examples

=====

The domain constructor `MultivariatePolynomial` is similar to `Polynomial` except that it specifies the variables to be used. `Polynomial` are available for `MultivariatePolynomial`. The abbreviation for `MultivariatePolynomial` is `MPOLY`. The type expressions

```

MultivariatePolynomial([x,y],Integer)
MPOLY([x,y],INT)

```

refer to the domain of multivariate polynomials in the variables x and y where the coefficients are restricted to be integers. The first variable specified is the main variable and the display of the polynomial reflects this.

This polynomial appears with terms in descending powers of the variable x .

```
m : MPOLY([x,y],INT) := (x^2 - x*y^3 + 3*y)^2
      4      3 3      6      2      4      2
      x  - 2y x  + (y  + 6y)x  - 6y x + 9y
      Type: MultivariatePolynomial([x,y],Integer)
```

It is easy to see a different variable ordering by doing a conversion.

```
m :: MPOLY([y,x],INT)
      2 6      4      3 3      2      2      4
      x y  - 6x y  - 2x y  + 9y  + 6x y + x
      Type: MultivariatePolynomial([y,x],Integer)
```

You can use other, unspecified variables, by using `Polynomial` in the coefficient type of `MPOLY`.

```
p : MPOLY([x,y],POLY INT)
      Type: Void
```

Conversions can be used to re-express such polynomials in terms of the other variables. For example, you can first push all the variables into a polynomial with integer coefficients.

```
p :: POLY INT
p
      Type: Polynomial Integer
```

Now pull out the variables of interest.

```
% :: MPOLY([a,b],POLY INT)
p
      Type: MultivariatePolynomial([a,b],Polynomial Integer)
```

Restriction:

Axiom does not allow you to create types where `MultivariatePolynomial` is contained in the coefficient type of `Polynomial`. Therefore, `MPOLY([x,y],POLY INT)` is legal but `POLY MPOLY([x,y],INT)` is not.

Multivariate polynomials may be combined with univariate polynomials to create types with special structures.

```
q : UP(x, FRAC MPOLY([y,z],INT))
      Type: Void
```

This is a polynomial in x whose coefficients are quotients of polynomials in y and z .

$$(7) \quad q := (x^2 - x(z+1)/y + 2)^2$$

$$x^4 + \frac{-2z - 2}{y} x^3 + \frac{4y^2 + z^2 + 2z + 1}{y^2} x^2 + \frac{-4z - 4}{y} x + 4$$

Type: UnivariatePolynomial(x, Fraction MultivariatePolynomial([y,z], Integer))

Use conversions for structural rearrangements. z does not appear in a denominator and so it can be made the main variable.

$$q :: UP(z, FRAC MPOLY([x,y], INT))$$

$$x^2 - 2yx + 2x^2 - 4yx^2 + \frac{y^2 x^4 - 2y^3 x^3 + (4y^2 + 1)x^2 - 4yx + 4y^2}{y^2} z + \frac{y^2 x^4 - 2y^3 x^3 + (4y^2 + 1)x^2 - 4yx + 4y^2}{y^2} z^2$$

Type: UnivariatePolynomial(z, Fraction MultivariatePolynomial([x,y], Integer))

Or you can make a multivariate polynomial in x and z whose coefficients are fractions in polynomials in y .

$$q :: MPOLY([x,z], FRAC UP(y, INT))$$

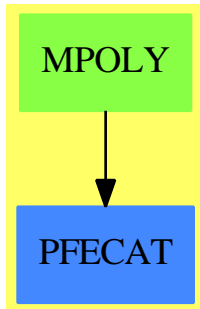
$$x^4 + (-\frac{z}{y} - \frac{z^2}{y})x^3 + (\frac{1}{y} - \frac{z}{y} + \frac{z^2}{y})x^2 + \frac{4y^2 + 1}{y^2}x + (-\frac{z}{y} - \frac{z^2}{y})x + 4$$

Type: MultivariatePolynomial([x,z], Fraction UnivariatePolynomial(y, Integer))

A conversion like $q :: MPOLY([x,y], FRAC UP(z, INT))$ is not possible in this example because y appears in the denominator of a fraction. As you can see, Axiom provides extraordinary flexibility in the manipulation and display of expressions via its conversion facility.

See Also:

- o)help DistributedMultivariatePolynomial
- o)help UnivariatePolynomial
- o)help Polynomial
- o)show MultivariatePolynomial

MultivariatePolynomial (MPOLY)**See**

- ⇒ “Polynomial” (POLY) [17](#) on page [2077](#)
- ⇒ “SparseMultivariatePolynomial” (SMP) [20](#) on page [2422](#)
- ⇒ “IndexedExponents” (INDE) [10](#) on page [1217](#)

Exports:

0	1	associates?
binomThmExpt	characteristic	charthRoot
coefficient	coefficients	coerce
conditionP	content	convert
D	degree	differentiate
discriminant	eval	exquo
factor	factorPolynomial	factorSquareFreePolynomial
gcd	gcdPolynomial	ground
ground?	hash	isExpt
isPlus	isTimes	latex
lcm	leadingCoefficient	leadingMonomial
mainVariable	map	mapExponents
max	min	minimumDegree
monicDivide	monomial	monomial?
monomials	multivariate	numberOfMonomials
one?	patternMatch	pomopo!
prime?	primitivePart	primitiveMonomials
recip	reducedSystem	reductum
resultant	retract	retractIfCan
sample	solveLinearPolynomialEquation	squareFree
squareFreePart	squareFreePolynomial	subtractIfCan
totalDegree	unit?	unitCanonical
unitNormal	univariate	variables
zero?	?*?	?**?
?+?	?-?	-?
?=?	?^?	?~=?
?/?	?<?	?<=?
?>?	?>=?	

— domain MPOLY MultivariatePolynomial —

```

)abbrev domain MPOLY MultivariatePolynomial
++ Author: Dave Barton, Barry Trager
++ Description:
++ This type is the basic representation of sparse recursive multivariate
++ polynomials whose variables are from a user specified list of symbols.
++ The ordering is specified by the position of the variable in the list.
++ The coefficient ring may be non commutative,
++ but the variables are assumed to commute.

MultivariatePolynomial(vl:List Symbol, R:Ring)
  == SparseMultivariatePolynomial(--SparseUnivariatePolynomial,
    R, OrderedVariableList vl)

```

— MPOLY.dotabb —

```
"MPOLY" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MPOLY"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MPOLY" -> "PFECAT"
```

domain MYEXPR MyExpression

— MyExpression.input —

```
)set break resume
)sys rm -f MyExpression.output
)spool MyExpression.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MyExpression
--R
--R MyExpression(q: Symbol,R: Join(Ring,OrderedSet,IntegralDomain)) is a domain constructor
--R Abbreviation for MyExpression is MYEXPR
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for MYEXPR
--R
--R----- Operations -----
--R ??? : (PositiveInteger,%) -> %      ??? : (Integer,%) -> %
--R ??? : (%,%) -> %                  ??? : (%,%) -> %
--R ??? : (%,PositiveInteger) -> %    ?+? : (%,%) -> %
--R -? : % -> %                       ?-? : (%,%) -> %
--R ?/? : (%,%) -> %                  ?<? : (%,%) -> Boolean
--R ?<=? : (%,%) -> Boolean           ?=? : (%,%) -> Boolean
--R ?>? : (%,%) -> Boolean           ?>=? : (%,%) -> Boolean
--R D : (%,Symbol) -> %               D : (%,List(Symbol)) -> %
--R 1 : () -> %                       0 : () -> %
--R ?? : (%,PositiveInteger) -> %    applyQuote : (Symbol,%,%) -> %
--R applyQuote : (Symbol,%) -> %      associates? : (%,%) -> Boolean
--R belong? : BasicOperator -> Boolean binomial : (%,%) -> %
--R box : List(%) -> %                box : % -> %
--R coerce : Integer -> %              coerce : % -> %
--R coerce : R -> %                   coerce : Symbol -> %
--R coerce : Kernel(%) -> %           coerce : % -> OutputForm
--R denominator : % -> %              differentiate : (%,Symbol) -> %
--R distribute : (%,%) -> %           distribute : % -> %
```

```

--R elt : (BasicOperator,%,%) -> %      elt : (BasicOperator,%) -> %
--R eval : (%,%,%) -> %                  eval : (%,Equation(%)) -> %
--R eval : (%,Kernel(%),%) -> %          factorial : % -> %
--R factorials : % -> %                  factorials : (%,Symbol) -> %
--R freeOf? : (%,Symbol) -> Boolean      freeOf? : (%,%) -> Boolean
--R ground : % -> R                      ground? : % -> Boolean
--R hash : % -> SingleInteger            height : % -> NonNegativeInteger
--R is? : (%,Symbol) -> Boolean           kernel : (BasicOperator,%) -> %
--R kernels : % -> List(Kernel(%))       latex : % -> String
--R max : (%,%) -> %                    min : (%,%) -> %
--R numerator : % -> %                  one? : % -> Boolean
--R paren : List(%) -> %                paren : % -> %
--R permutation : (%,%) -> %            product : (%,Symbol) -> %
--R recip : % -> Union(%, "failed")      retract : % -> Integer
--R retract : % -> R                    retract : % -> Symbol
--R retract : % -> Kernel(%)             sample : () -> %
--R subst : (%,Equation(%)) -> %         summation : (%,Symbol) -> %
--R tower : % -> List(Kernel(%))         unit? : % -> Boolean
--R unitCanonical : % -> %              variables : % -> List(Symbol)
--R zero? : % -> Boolean                 ~=? : (%,%) -> Boolean

--R ?? : (NonNegativeInteger,%) -> %
--R ?? : (%,R) -> % if R has COMRING
--R ?? : (R,%) -> % if R has COMRING
--R ?? : (%,Fraction(Integer)) -> % if R has INTDOM
--R ?? : (Fraction(Integer),%) -> % if R has INTDOM
--R ??? : (%,Integer) -> % if R has GROUP or R has INTDOM
--R ??? : (%,NonNegativeInteger) -> %
--R ?/? : (SparseMultivariatePolynomial(R,Kernel(%)),SparseMultivariatePolynomial(R,Kernel(%)))
--R D : (%,Symbol,NonNegativeInteger) -> %
--R D : (%,List(Symbol),List(NonNegativeInteger)) -> %
--R ?^? : (%,Integer) -> % if R has GROUP or R has INTDOM
--R ?^? : (%,NonNegativeInteger) -> %
--R applyQuote : (Symbol,List(%)) -> %
--R applyQuote : (Symbol,%,%,%,%) -> %
--R applyQuote : (Symbol,%,%,%) -> %
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if R has CHARNZ
--R coerce : Fraction(Integer) -> % if R has INTDOM or R has RETRACT(FRAC(INT))
--R coerce : Fraction(MyUnivariatePolynomial(q,R)) -> %
--R coerce : MyUnivariatePolynomial(q,R) -> %
--R coerce : Polynomial(R) -> % if R has RING
--R coerce : Fraction(Polynomial(R)) -> % if R has INTDOM
--R coerce : Fraction(Polynomial(Fraction(R))) -> % if R has INTDOM
--R coerce : Polynomial(Fraction(R)) -> % if R has INTDOM
--R coerce : Fraction(R) -> % if R has INTDOM
--R coerce : SparseMultivariatePolynomial(R,Kernel(%)) -> % if R has RING
--R commutator : (%,%) -> % if R has GROUP
--R conjugate : (%,%) -> % if R has GROUP
--R convert : % -> InputForm if R has KONVERT(INFORM)
--R convert : Factored(%) -> % if R has INTDOM

```

```
--R convert : % -> Pattern(Float) if R has KONVERT(FLOAT))
--R convert : % -> Pattern(Integer) if R has KONVERT(PATTERN(INT))
--R definingPolynomial : % -> % if $ has RING
--R denom : % -> SparseMultivariatePolynomial(R,Kernel(%)) if R has INTDOM
--R differentiate : (% ,List(Symbol)) -> %
--R differentiate : (% ,Symbol,NonNegativeInteger) -> %
--R differentiate : (% ,List(Symbol),List(NonNegativeInteger)) -> %
--R divide : (% ,%) -> Record(quotient: %,remainder: %) if R has INTDOM
--R elt : (BasicOperator,List(%)) -> %
--R elt : (BasicOperator,% ,% ,%) -> %
--R elt : (BasicOperator,% ,% ,%) -> %
--R euclideanSize : % -> NonNegativeInteger if R has INTDOM
--R eval : (% ,Symbol,NonNegativeInteger,(% -> %)) -> % if R has RING
--R eval : (% ,Symbol,NonNegativeInteger,(List(% ) -> %)) -> % if R has RING
--R eval : (% ,List(Symbol),List(NonNegativeInteger),List((List(% ) -> %))) -> % if R has RING
--R eval : (% ,List(Symbol),List(NonNegativeInteger),List((% -> %))) -> % if R has RING
--R eval : (% ,List(BasicOperator),List(% ),Symbol) -> % if R has KONVERT(INFORM)
--R eval : (% ,BasicOperator,% ,Symbol) -> % if R has KONVERT(INFORM)
--R eval : % -> % if R has KONVERT(INFORM)
--R eval : (% ,List(Symbol)) -> % if R has KONVERT(INFORM)
--R eval : (% ,Symbol) -> % if R has KONVERT(INFORM)
--R eval : (% ,BasicOperator,(% -> %)) -> %
--R eval : (% ,BasicOperator,(List(% ) -> %)) -> %
--R eval : (% ,List(BasicOperator),List((List(% ) -> %))) -> %
--R eval : (% ,List(BasicOperator),List((% -> %))) -> %
--R eval : (% ,Symbol,(% -> %)) -> %
--R eval : (% ,Symbol,(List(% ) -> %)) -> %
--R eval : (% ,List(Symbol),List((List(% ) -> %))) -> %
--R eval : (% ,List(Symbol),List((% -> %))) -> %
--R eval : (% ,List(% ),List(% )) -> %
--R eval : (% ,List(Equation(% ))) -> %
--R eval : (% ,List(Kernel(% )),List(% )) -> %
--R even? : % -> Boolean if $ has RETRACT(INT)
--R expressIdealMember : (List(% ),%) -> Union(List(% ),"failed") if R has INTDOM
--R exquo : (% ,%) -> Union(% ,"failed")
--R extendedEuclidean : (% ,%) -> Record(coef1: %,coef2: %,generator: %) if R has INTDOM
--R extendedEuclidean : (% ,% ,%) -> Union(Record(coef1: %,coef2: %),"failed") if R has INTDOM
--R factor : % -> Factored(% ) if R has INTDOM
--R gcd : (% ,%) -> % if R has INTDOM
--R gcd : List(% ) -> % if R has INTDOM
--R gcdPolynomial : (SparseUnivariatePolynomial(% ),SparseUnivariatePolynomial(% )) -> SparseUnivariatePolynomo
--R inv : % -> % if R has GROUP or R has INTDOM
--R is? : (% ,BasicOperator) -> Boolean
--R isExpt : (% ,Symbol) -> Union(Record(var: Kernel(% ),exponent: Integer),"failed") if R has RING
--R isExpt : (% ,BasicOperator) -> Union(Record(var: Kernel(% ),exponent: Integer),"failed") if R has RING
--R isExpt : % -> Union(Record(var: Kernel(% ),exponent: Integer),"failed") if R has SGROUP
--R isMult : % -> Union(Record(coef: Integer,var: Kernel(% )), "failed") if R has ABELSG
--R isPlus : % -> Union(List(% ), "failed") if R has ABELSG
--R isPower : % -> Union(Record(val: %,exponent: Integer), "failed") if R has RING
--R isTimes : % -> Union(List(% ), "failed") if R has SGROUP
```



```

--R kernel : (BasicOperator,List(%)) -> %
--R lcm : (%,% ) -> % if R has INTDOM
--R lcm : List(% ) -> % if R has INTDOM
--R mainKernel : % -> Union(Kernel(%),"failed")
--R map : ((% -> %),Kernel(%)) -> %
--R minPoly : Kernel(% ) -> SparseUnivariatePolynomial(% ) if $ has RING
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed") if R has INTDOM
--R numer : % -> SparseMultivariatePolynomial(R,Kernel(%)) if R has RING
--R odd? : % -> Boolean if $ has RETRACT(INT)
--R operator : BasicOperator -> BasicOperator
--R operators : % -> List(BasicOperator)
--R patternMatch : (% ,Pattern(Float),PatternMatchResult(Float,%)) -> PatternMatchResult(Float,% )
--R patternMatch : (% ,Pattern(Integer),PatternMatchResult(Integer,%)) -> PatternMatchResult(Integer,% )
--R prime? : % -> Boolean if R has INTDOM
--R principalIdeal : List(% ) -> Record(coef: List(%),generator: % ) if R has INTDOM
--R product : (% ,SegmentBinding(%)) -> %
--R ?quo? : (% ,%) -> % if R has INTDOM
--R reducedSystem : Matrix(% ) -> Matrix(Integer) if R has LINEXP(INT) and R has RING
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) if R has RING
--R reducedSystem : Matrix(% ) -> Matrix(R) if R has RING
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(R),vec: Vector(R)) if R has RING
--R ?rem? : (% ,%) -> % if R has INTDOM
--R retract : % -> Fraction(Integer) if R has INTDOM and R has RETRACT(INT) or R has RETRACT(FRACTION)
--R retract : % -> Fraction(MyUnivariatePolynomial(q,R))
--R retract : % -> MyUnivariatePolynomial(q,R)
--R retract : % -> Polynomial(R) if R has RING
--R retract : % -> Fraction(Polynomial(R)) if R has INTDOM
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if R has INTDOM and R has RETRACT(INT)
--R retractIfCan : % -> Union(Integer,"failed")
--R retractIfCan : % -> Union(MyUnivariatePolynomial(q,R),"failed")
--R retractIfCan : % -> Union(Polynomial(R),"failed") if R has RING
--R retractIfCan : % -> Union(Fraction(Polynomial(R)),"failed") if R has INTDOM
--R retractIfCan : % -> Union(R,"failed")
--R retractIfCan : % -> Union(Symbol,"failed")
--R retractIfCan : % -> Union(Kernel(%),"failed")
--R sizeLess? : (% ,%) -> Boolean if R has INTDOM
--R squareFree : % -> Factored(% ) if R has INTDOM
--R squareFreePart : % -> % if R has INTDOM
--R subst : (% ,List(Kernel(%)),List(%)) -> %
--R subst : (% ,List(Equation(%))) -> %
--R subtractIfCan : (% ,%) -> Union(%,"failed")
--R summation : (% ,SegmentBinding(%)) -> %
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R univariate : (% ,Kernel(%)) -> Fraction(SparseUnivariatePolynomial(%)) if R has INTDOM
--R
--E 1

)spool
)lisp (bye)

```

```

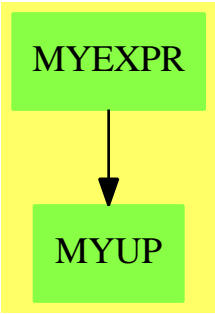
-----
-- MyExpression.help --
=====
MyExpression examples
=====

This domain has no description

See Also:
o )show MyExpression

-----
```

MyExpression (MYEXPR)



See
⇒ “MyUnivariatePolynomial” (MYUP) [14](#) on page [1694](#)
Exports:

0	1	applyQuote	associates?
belong?	binomial	box	characteristic
charthRoot	coerce	commutator	conjugate
convert	D	definingPolynomial	denom
denominator	differentiate	distribute	divide
elt	euclideanSize	eval	even?
expressIdealMember	exquo	extendedEuclidean	factor
factorial	factorials	freeOf?	gcd
gcdPolynomial	ground	ground?	hash
height	inv	is?	is?
isExpt	isMult	isPlus	isPower
isTimes	kernel	kernels	latex
lcm	mainKernel	map	max
min	minPoly	multiEuclidean	numer
numerator	odd?	one?	operator
operators	paren	patternMatch	permutation
prime?	principalIdeal	product	recip
reducedSystem	retract	retractIfCan	sample
sizeLess?	squareFree	squareFreePart	subst
subtractIfCan	summation	tower	unit?
unitCanonical	unitNormal	univariate	variables
zero?	?*	***?	?+?
-?	?-?	?/?	?<?
?<=?	?=?	?>?	?>=?
?^?	?~=?	?quo?	?rem?

— domain MYEXPR MyExpression —

```
)abbrev domain MYEXPR MyExpression
```

```
++ Author: Mark Botch
```

```
++ Description:
```

```
++ This domain has no description
```

```
MyExpression(q: Symbol, R): Exports == Implementation where
```

```
R: Join(Ring, OrderedSet, IntegralDomain)
```

```
UP ==> MyUnivariatePolynomial(q, R)
```

```
Exports == Join(FunctionSpace R, IntegralDomain,
  RetractableTo UP, RetractableTo Symbol,
  RetractableTo Integer, CombinatorialOpsCategory,
  PartialDifferentialRing Symbol) with
  _* : (%,%) -> %
  _/ : (%,%) -> %
  _*_ : (%,%) -> %
  numerator : % -> %
  denominator : % -> %
  ground? : % -> Boolean
```

```

coerce: Fraction UP -> %
retract: % -> Fraction UP

Implementation == Expression R add
Rep := Expression R

iunivariate(p: Polynomial R): UP ==
  poly: SparseUnivariatePolynomial(Polynomial R)
  := univariate(p, q)$(Polynomial R)
  map((z1:Polynomial R):R +-> retract(z1), poly)_
  $UnivariatePolynomialCategoryFunctions2(Polynomial R,
    SparseUnivariatePolynomial Polynomial R,
    R, UP)

retract(p: %): Fraction UP ==
  poly: Fraction Polynomial R := retract p
  upoly: UP := iunivariate numer poly
  vpoly: UP := iunivariate denom poly

  upoly / vpoly

retract(p: %): UP == iunivariate retract p

coerce(r: Fraction UP): % ==
  num: SparseUnivariatePolynomial R := makeSUP numer r
  den: SparseUnivariatePolynomial R := makeSUP denom r
  u: Polynomial R := multivariate(num, q)
  v: Polynomial R := multivariate(den, q)

  quot: Fraction Polynomial R := u/v

  quot::(Expression R)



---


— MYEXPR.dotabb —

"MYEXPR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MYEXPR"]
"MYUP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MYUP"]
"MYEXPR" -> "MYUP"

```

domain MYUP MyUnivariatePolynomial

— MyUnivariatePolynomial.input —

```

)set break resume
)sys rm -f MyUnivariatePolynomial.output
)spool MyUnivariatePolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MyUnivariatePolynomial
--R
--R MyUnivariatePolynomial(x: Symbol,R: Ring) is a domain constructor
--R Abbreviation for MyUnivariatePolynomial is MYUP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for MYUP
--R
--R----- Operations -----
--R ?? : (%,R) -> %           ?? : (R,%) -> %
--R ?? : (%,%) -> %           ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R ?+ : (%,%) -> %           ?- : (%,%) -> %
--R -? : % -> %               ?? : (%,%) -> Boolean
--R D : (%,(R -> R)) -> %       D : % -> %
--R D : (%,NonNegativeInteger) -> %   1 : () -> %
--R 0 : () -> %               ?^? : (%,PositiveInteger) -> %
--R coefficients : % -> List(R)       coerce : Symbol -> %
--R coerce : Polynomial(R) -> %       coerce : Variable(x) -> %
--R coerce : R -> %               coerce : Integer -> %
--R coerce : % -> OutputForm         degree : % -> NonNegativeInteger
--R differentiate : % -> %           ?.? : (%,%) -> %
--R ?.? : (%,R) -> R               eval : (%,%,%) -> %
--R eval : (%,Equation(%)) -> %       ground : % -> R
--R ground? : % -> Boolean           hash : % -> SingleInteger
--R init : () -> % if R has STEP      latex : % -> String
--R leadingCoefficient : % -> R       leadingMonomial : % -> %
--R map : ((R -> R),%) -> %          monomial? : % -> Boolean
--R monomials : % -> List(%)         one? : % -> Boolean
--R pseudoRemainder : (%,%) -> %     recip : % -> Union(%, "failed")
--R reductum : % -> %               retract : % -> Symbol
--R retract : % -> R                 sample : () -> %
--R zero? : % -> Boolean             ?~=? : (%,%) -> Boolean
--R ?? : (Fraction(Integer),%) -> % if R has ALGEBRA(FRAC(INT))
--R ?? : (%,Fraction(Integer)) -> % if R has ALGEBRA(FRAC(INT))
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R ?/? : (%,R) -> % if R has FIELD
--R ?<? : (%,%) -> Boolean if R has ORDSET
--R ?<=? : (%,%) -> Boolean if R has ORDSET
--R ?>? : (%,%) -> Boolean if R has ORDSET

```

```

--R ?>=? : (%,% ) -> Boolean if R has ORDSET
--R D : (%,(R -> R),NonNegativeInteger) -> %
--R D : (%,(List(Symbol),List(NonNegativeInteger))) -> % if R has PDRING(SYMBOL)
--R D : (%,(Symbol,NonNegativeInteger)) -> % if R has PDRING(SYMBOL)
--R D : (%,(List(Symbol))) -> % if R has PDRING(SYMBOL)
--R D : (%,(Symbol)) -> % if R has PDRING(SYMBOL)
--R D : (%,(List(SingletonAsOrderedSet),List(NonNegativeInteger))) -> %
--R D : (%,(SingletonAsOrderedSet,NonNegativeInteger)) -> %
--R D : (%,(List(SingletonAsOrderedSet))) -> %
--R D : (%,(SingletonAsOrderedSet)) -> %
--R ??? : (%,(NonNegativeInteger)) -> %
--R associates? : (%,% ) -> Boolean if R has INTDOM
--R binomThmExpt : (%,% ,NonNegativeInteger) -> % if R has COMRING
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%,"failed") if $ has CHARNZ and R has PFECAT or R has CHARNZ
--R coefficient : (%,(List(SingletonAsOrderedSet),List(NonNegativeInteger))) -> %
--R coefficient : (%,(SingletonAsOrderedSet,NonNegativeInteger)) -> %
--R coefficient : (%,(NonNegativeInteger)) -> R
--R coerce : % -> % if R has INTDOM
--R coerce : Fraction(Integer) -> % if R has ALGEBRA(FRAC(INT)) or R has RETRACT(FRAC(INT))
--R coerce : SingletonAsOrderedSet -> %
--R composite : (Fraction(%),%) -> Union(Fraction(%),"failed") if R has INTDOM
--R composite : (%,% ) -> Union(%,"failed") if R has INTDOM
--R conditionP : Matrix(%) -> Union(Vector(%),"failed") if $ has CHARNZ and R has PFECAT
--R content : (%,(SingletonAsOrderedSet)) -> % if R has GCDDOM
--R content : % -> R if R has GCDDOM
--R convert : % -> InputForm if SingletonAsOrderedSet has KONVERT(INFORM) and R has KONVERT(INFORM)
--R convert : % -> Pattern(Integer) if SingletonAsOrderedSet has KONVERT(PATTERN(INT)) and R has KONVERT(PA
--R convert : % -> Pattern(Float) if SingletonAsOrderedSet has KONVERT(PATTERN(FLOAT)) and R has KONVERT(PA
--R degree : (%,(List(SingletonAsOrderedSet))) -> List(NonNegativeInteger)
--R degree : (%,(SingletonAsOrderedSet)) -> NonNegativeInteger
--R differentiate : (%,(R -> R),%) -> %
--R differentiate : (%,(R -> R)) -> %
--R differentiate : (%,(R -> R),NonNegativeInteger) -> %
--R differentiate : (%,(List(Symbol),List(NonNegativeInteger))) -> % if R has PDRING(SYMBOL)
--R differentiate : (%,(Symbol,NonNegativeInteger)) -> % if R has PDRING(SYMBOL)
--R differentiate : (%,(List(Symbol))) -> % if R has PDRING(SYMBOL)
--R differentiate : (%,(Symbol)) -> % if R has PDRING(SYMBOL)
--R differentiate : (%,(NonNegativeInteger)) -> %
--R differentiate : (%,(List(SingletonAsOrderedSet),List(NonNegativeInteger))) -> %
--R differentiate : (%,(SingletonAsOrderedSet,NonNegativeInteger)) -> %
--R differentiate : (%,(List(SingletonAsOrderedSet))) -> %
--R differentiate : (%,(SingletonAsOrderedSet)) -> %
--R discriminant : % -> R if R has COMRING
--R discriminant : (%,(SingletonAsOrderedSet)) -> % if R has COMRING
--R divide : (%,% ) -> Record(quotient: %,remainder: %) if R has FIELD
--R divideExponents : (%,(NonNegativeInteger)) -> Union(%,"failed")
--R ?.? : (%,(Fraction(%))) -> Fraction(%) if R has INTDOM
--R elt : (Fraction(%),R) -> R if R has FIELD
--R elt : (Fraction(%),Fraction(%)) -> Fraction(%) if R has INTDOM

```

```

--R euclideanSize : % -> NonNegativeInteger if R has FIELD
--R eval : (% , List(SingletonAsOrderedSet), List(%)) -> %
--R eval : (% , SingletonAsOrderedSet, %) -> %
--R eval : (% , List(SingletonAsOrderedSet), List(R)) -> %
--R eval : (% , SingletonAsOrderedSet, R) -> %
--R eval : (% , List(%), List(%)) -> %
--R eval : (% , List(Equation(%))) -> %
--R expressIdealMember : (List(%), %) -> Union(List(%), "failed") if R has FIELD
--R exquo : (% , %) -> Union(% , "failed") if R has INTDOM
--R exquo : (% , R) -> Union(% , "failed") if R has INTDOM
--R extendedEuclidean : (% , %) -> Record(coef1: %, coef2: %, generator: %) if R has FIELD
--R extendedEuclidean : (% , %, %) -> Union(Record(coef1: %, coef2: %), "failed") if R has FIELD
--R factor : % -> Factored(%) if R has PFECAT
--R factorPolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%))
--R factorSquareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%))
--R fmeq : (% , NonNegativeInteger, R, %) -> %
--R gcd : (% , %) -> % if R has GCDDOM
--R gcd : List(%) -> % if R has GCDDOM
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R integrate : % -> % if R has ALGEBRA(FRAC(INT))
--R isExpt : % -> Union(Record(var: SingletonAsOrderedSet, exponent: NonNegativeInteger), "failed")
--R isPlus : % -> Union(List(%), "failed")
--R isTimes : % -> Union(List(%), "failed")
--R karatsubaDivide : (% , NonNegativeInteger) -> Record(quotient: %, remainder: %)
--R lcm : (% , %) -> % if R has GCDDOM
--R lcm : List(%) -> % if R has GCDDOM
--R mainVariable : % -> Union(SingletonAsOrderedSet, "failed")
--R makeSUP : % -> SparseUnivariatePolynomial(R)
--R mapExponents : ((NonNegativeInteger -> NonNegativeInteger), %) -> %
--R max : (% , %) -> % if R has ORDSET
--R min : (% , %) -> % if R has ORDSET
--R minimumDegree : (% , List(SingletonAsOrderedSet)) -> List(NonNegativeInteger)
--R minimumDegree : (% , SingletonAsOrderedSet) -> NonNegativeInteger
--R minimumDegree : % -> NonNegativeInteger
--R monicDivide : (% , %) -> Record(quotient: %, remainder: %)
--R monicDivide : (% , %, SingletonAsOrderedSet) -> Record(quotient: %, remainder: %)
--R monomial : (% , List(SingletonAsOrderedSet), List(NonNegativeInteger)) -> %
--R monomial : (% , SingletonAsOrderedSet, NonNegativeInteger) -> %
--R monomial : (R, NonNegativeInteger) -> %
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed") if R has FIELD
--R multiplyExponents : (% , NonNegativeInteger) -> %
--R multivariate : (SparseUnivariatePolynomial(%), SingletonAsOrderedSet) -> %
--R multivariate : (SparseUnivariatePolynomial(R), SingletonAsOrderedSet) -> %
--R nextItem : % -> Union(% , "failed") if R has STEP
--R numberOfMonomials : % -> NonNegativeInteger
--R order : (% , %) -> NonNegativeInteger if R has INTDOM
--R patternMatch : (% , Pattern(Integer), PatternMatchResult(Integer, %)) -> PatternMatchResult(Integer, %)
--R patternMatch : (% , Pattern(Float), PatternMatchResult(Float, %)) -> PatternMatchResult(Float, %)
--R pomopo! : (% , R, NonNegativeInteger, %) -> %
--R prime? : % -> Boolean if R has PFECAT

```

```

--R primitiveMonomials : % -> List(%)
--R primitivePart : (% , SingletonAsOrderedSet) -> % if R has GCDDOM
--R primitivePart : % -> % if R has GCDDOM
--R principalIdeal : List(%) -> Record(coef: List(%) , generator: %) if R has FIELD
--R pseudoDivide : (% , %) -> Record(coef: R , quotient: % , remainder: %) if R has INTDOM
--R pseudoQuotient : (% , %) -> % if R has INTDOM
--R ?quo? : (% , %) -> % if R has FIELD
--R reducedSystem : Matrix(%) -> Matrix(R)
--R reducedSystem : (Matrix(%) , Vector(%) ) -> Record(mat: Matrix(R) , vec: Vector(R))
--R reducedSystem : (Matrix(%) , Vector(%) ) -> Record(mat: Matrix(Integer) , vec: Vector(Integer)) if R has LIN
--R reducedSystem : Matrix(%) -> Matrix(Integer) if R has LINEXP(INT)
--R ?rem? : (% , %) -> % if R has FIELD
--R resultant : (% , %) -> R if R has COMRING
--R resultant : (% , % , SingletonAsOrderedSet) -> % if R has COMRING
--R retract : % -> SingletonAsOrderedSet
--R retract : % -> Integer if R has RETRACT(INT)
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(Symbol , "failed")
--R retractIfCan : % -> Union(SingletonAsOrderedSet , "failed")
--R retractIfCan : % -> Union(Integer , "failed") if R has RETRACT(INT)
--R retractIfCan : % -> Union(Fraction(Integer) , "failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(R , "failed")
--R separate : (% , %) -> Record(primePart: % , commonPart: %) if R has GCDDOM
--R shiftLeft : (% , NonNegativeInteger) -> %
--R shiftRight : (% , NonNegativeInteger) -> %
--R sizeLess? : (% , %) -> Boolean if R has FIELD
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(%)) , SparseUnivariatePolynomial(%)) -> U
--R squareFree : % -> Factored(%) if R has GCDDOM
--R squareFreePart : % -> % if R has GCDDOM
--R squareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if R has
--R subResultantGcd : (% , %) -> % if R has INTDOM
--R subtractIfCan : (% , %) -> Union(%, "failed")
--R totalDegree : (% , List(SingletonAsOrderedSet)) -> NonNegativeInteger
--R totalDegree : % -> NonNegativeInteger
--R unit? : % -> Boolean if R has INTDOM
--R unitCanonical : % -> % if R has INTDOM
--R unitNormal : % -> Record(unit: % , canonical: % , associate: %) if R has INTDOM
--R univariate : % -> SparseUnivariatePolynomial(R)
--R univariate : (% , SingletonAsOrderedSet) -> SparseUnivariatePolynomial(%)
--R unmakeSUP : SparseUnivariatePolynomial(R) -> %
--R variables : % -> List(SingletonAsOrderedSet)
--R vectorise : (% , NonNegativeInteger) -> Vector(R)
--R
--E 1

)spool
)lisp (bye)

```

— MyUnivariatePolynomial.help —

```
=====
MyUnivariatePolynomial examples
=====
```

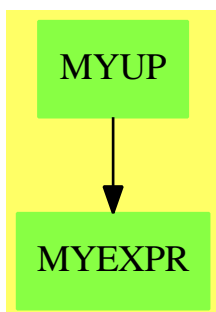
This domain has no description

See Also:

o)show MyUnivariatePolynomial

—————▶

MyUnivariatePolynomial (MYUP)



See

⇒ “MyExpression” (MYEXPR) [14](#) on page [1687](#)

Exports:

0	1	associates?
binomThmExpt	characteristic	charthRoot
coefficient	coefficients	coerce
composite	conditionP	content
convert	D	degree
differentiate	discriminant	divide
divideExponents	elt	euclideanSize
eval	expressIdealMember	exquo
extendedEuclidean	factor	factorPolynomial
factorSquareFreePolynomial	fmeqg	gcd
gcdPolynomial	ground	ground?
hash	init	integrate
isExpt	isPlus	isTimes
karatsubaDivide	latex	lcm
leadingCoefficient	leadingMonomial	mainVariable
makeSUP	map	mapExponents
max	min	minimumDegree
monicDivide	monomial	monomial?
monomials	multiEuclidean	multiplyExponents
multivariate	nextItem	numberOfMonomials
one?	order	patternMatch
pomopo!	prime?	primitiveMonomials
primitivePart	principalIdeal	pseudoDivide
pseudoQuotient	pseudoRemainder	recip
reducedSystem	reductum	resultant
retract	retractIfCan	sample
separate	shiftLeft	shiftRight
sizeLess?	solveLinearPolynomialEquation	squareFree
squareFreePart	squareFreePolynomial	subResultantGcd
subtractIfCan	totalDegree	unit?
unitCanonical	unitNormal	univariate
unmakeSUP	variables	vectorise
zero?	?*?	?**?
?+?	?-?	-?
?=?	?^?	?..?
?~=?	?/?	?<?
?<=?	?>?	?>=?
?^?	?quo?	?rem?

— domain MYUP MyUnivariatePolynomial —

```

)abbrev domain MYUP MyUnivariatePolynomial
++ Author: Mark Botch
++ Description:
++ This domain has no description

MyUnivariatePolynomial(x:Symbol, R:Ring):

```

```

UnivariatePolynomialCategory(R) with
  RetractableTo Symbol;
  coerce: Variable(x) -> %
    ++ coerce(x) converts the variable x to a univariate polynomial.
  fmecg: (%,NonNegativeInteger,R,%) -> %
    ++ fmecg(p1,e,r,p2) finds x : p1 - r * x**e * p2
  if R has univariate: (R, Symbol) -> SparseUnivariatePolynomial R
  then coerce: R -> %
  coerce: Polynomial R -> %
== SparseUnivariatePolynomial(R) add
Rep := SparseUnivariatePolynomial(R)
coerce(p: %):OutputForm == outputForm(p, outputForm x)
coerce(x: Symbol): % == monomial(1, 1)
coerce(v: Variable(x)): % == monomial(1, 1)
retract(p: %): Symbol ==
  retract(p)@SingletonAsOrderedSet
  x
if R has univariate: (R, Symbol) -> SparseUnivariatePolynomial R
then coerce(p: R): % == univariate(p, x)$R

coerce(p: Polynomial R): % ==
  poly: SparseUnivariatePolynomial(Polynomial R)
  := univariate(p, x)$(Polynomial R)
  map((z1:Polynomial R):R +-> retract(z1), poly)_
    $UnivariatePolynomialCategoryFunctions2(Polynomial R,
      SparseUnivariatePolynomial Polynomial R, R, %)

```

— MYUP.dotabb —

```

"MYUP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MYUP"]
"MYEXPR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=MYEXPR"]
"MYUP" -> "MYEXPR"

```

Chapter 15

Chapter N

domain NSDPS NeitherSparseOrDensePowerSeries

— NeitherSparseOrDensePowerSeries.input —

```
)set break resume
)sys rm -f NeitherSparseOrDensePowerSeries.output
)spool NeitherSparseOrDensePowerSeries.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NeitherSparseOrDensePowerSeries
--R
--R NeitherSparseOrDensePowerSeries(K: Field) is a domain constructor
--R Abbreviation for NeitherSparseOrDensePowerSeries is NSDPS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for NSDPS
--R
--R----- Operations -----
--R ??? : (%,K) -> %          ??? : (K,%) -> %
--R ??? : (Fraction(Integer),%) -> %    ??? : (%,Fraction(Integer)) -> %
--R ??? : (%,%) -> %          ??? : (Integer,%) -> %
--R ??? : (PositiveInteger,%) -> %      ??? : (%,Integer) -> %
--R ??? : (%,PositiveInteger) -> %      ?+? : (%,%) -> %
--R ?-? : (%,%) -> %          -? : % -> %
--R ?/? : (%,%) -> %          ?? : (%,%) -> Boolean
--R 1 : () -> %              0 : () -> %
--R ?? : (%,Integer) -> %      ?? : (%,PositiveInteger) -> %
--R associates? : (%,%) -> Boolean    center : % -> K
--R children : % -> List(%)          coefOfFirstNonZeroTerm : % -> K
```

```

--R coefficient : (%,Integer) -> K
--R coerce : % -> %
--R coerce : % -> OutputForm
--R concat : (%,% ) -> %
--R copy : % -> %
--R cycleTail : % -> %
--R degree : % -> Integer
--R delete : (%,Integer) -> %
--R ?.rest : (% ,rest) -> %
--R empty : () -> %
--R eq? : (%,% ) -> Boolean
--R explicitlyEmpty? : % -> Boolean
--R extend : (% ,Integer) -> %
--R filterUpTo : (% ,Integer) -> %
--R gcd : List(% ) -> %
--R hash : % -> SingleInteger
--R indices : % -> List(Integer)
--R inv : % -> %
--R lazy? : % -> Boolean
--R lcm : List(% ) -> %
--R leadingCoefficient : % -> K
--R leaf? : % -> Boolean
--R monomial : (K,Integer) -> %
--R nodes : % -> List(% )
--R order : % -> Integer
--R order : (% ,Integer) -> Integer
--R posExpnPart : % -> %
--R prime? : % -> Boolean
--R printInfo : Boolean -> Boolean
--R recip : % -> Union(% ,"failed")
--R ?rem? : (%,% ) -> %
--R removeZeroes : % -> %
--R rest : % -> %
--R sample : () -> %
--R series : (Integer,K,% ) -> %
--R sizeLess? : (%,% ) -> Boolean
--R squareFreePart : % -> %
--R truncate : (% ,Integer) -> %
--R unitCanonical : % -> %
--R zero? : % -> Boolean
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?*? : (NonNegativeInteger,% ) -> %
--R ?**? : (% ,NonNegativeInteger) -> %
--R ?/? : (% ,K) -> % if K has FIELD
--R D : (% ,List(Symbol),List(NonNegativeInteger)) -> % if K has *: (Integer,K) -> K and K has PD
--R D : (% ,Symbol,NonNegativeInteger) -> % if K has *: (Integer,K) -> K and K has PDRING(SYMBOL)
--R D : (% ,List(Symbol)) -> % if K has *: (Integer,K) -> K and K has PDRING(SYMBOL)
--R D : (% ,Symbol) -> % if K has *: (Integer,K) -> K and K has PDRING(SYMBOL)
--R D : (% ,NonNegativeInteger) -> % if K has *: (Integer,K) -> K
--R D : % -> % if K has *: (Integer,K) -> K

coerce : Fraction(Integer) -> %
coerce : Integer -> %
complete : % -> %
concat : List(% ) -> %
cycleEntry : % -> %
cyclic? : % -> Boolean
delay : (() -> % ) -> %
distance : (%,% ) -> Integer
?.? : (% ,Integer) -> K
empty? : % -> Boolean
explicitEntries? : % -> Boolean
explicitlyFinite? : % -> Boolean
factor : % -> Factored(% )
findCoef : (% ,Integer) -> K
gcd : (%,% ) -> %
index? : (Integer,% ) -> Boolean
insert : (%,% ,Integer) -> %
latex : % -> String
lazyEvaluate : % -> %
lcm : (%,% ) -> %
leadingMonomial : % -> %
map : ((K -> K),% ) -> %
monomial? : % -> Boolean
one? : % -> Boolean
order : % -> Integer
pole? : % -> Boolean
possiblyInfinite? : % -> Boolean
printInfo : () -> Boolean
?quo? : (%,% ) -> %
reductum : % -> %
removeFirstZeroes : % -> %
removeZeroes : (Integer,% ) -> %
rst : % -> %
sbt : (%,% ) -> %
shift : (% ,Integer) -> %
squareFree : % -> Factored(% )
tail : % -> %
unit? : % -> Boolean
variable : % -> Symbol
?~=? : (%,% ) -> Boolean

```

```

--R ?? : (% , NonNegativeInteger) -> %
--R any? : ((Record(k: Integer, c: K) -> Boolean), %) -> Boolean if $ has finiteAggregate
--R approximate : (% , Integer) -> K if K has **: (K, Integer) -> K and K has coerce: Symbol -> K
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(% , "failed") if K has CHARNZ
--R child? : (% , %) -> Boolean if Record(k: Integer, c: K) has SETCAT
--R coerce : % -> Stream(Record(k: Integer, c: K))
--R coerce : Stream(Record(k: Integer, c: K)) -> %
--R coerce : K -> % if K has COMRING
--R concat : (Record(k: Integer, c: K), %) -> %
--R concat : (% , Record(k: Integer, c: K)) -> %
--R concat! : (% , %) -> % if $ has shallowlyMutable
--R concat! : (% , Record(k: Integer, c: K)) -> % if $ has shallowlyMutable
--R construct : List(Record(k: Integer, c: K)) -> %
--R convert : % -> InputForm if Record(k: Integer, c: K) has KONVERT(INFORM)
--R count : ((Record(k: Integer, c: K) -> Boolean), %) -> NonNegativeInteger if $ has finiteAggregate
--R count : (Record(k: Integer, c: K), %) -> NonNegativeInteger if $ has finiteAggregate and Record(k: Integer, c: K) has finiteAggregate
--R cycleLength : % -> NonNegativeInteger
--R cycleSplit! : % -> % if $ has shallowlyMutable
--R delete : (% , UniversalSegment(Integer)) -> %
--R differentiate : (% , List(Symbol), List(NonNegativeInteger)) -> % if K has *: (Integer, K) -> K and K has PDRING(SYMBOL)
--R differentiate : (% , Symbol, NonNegativeInteger) -> % if K has *: (Integer, K) -> K and K has PDRING(SYMBOL)
--R differentiate : (% , List(Symbol)) -> % if K has *: (Integer, K) -> K and K has PDRING(SYMBOL)
--R differentiate : (% , Symbol) -> % if K has *: (Integer, K) -> K and K has PDRING(SYMBOL)
--R differentiate : (% , NonNegativeInteger) -> % if K has *: (Integer, K) -> K
--R differentiate : % -> % if K has *: (Integer, K) -> K
--R divide : (% , %) -> Record(quotient: % , remainder: %)
--R ?.value : (% , value) -> Record(k: Integer, c: K)
--R ?.first : (% , first) -> Record(k: Integer, c: K)
--R ?.last : (% , last) -> Record(k: Integer, c: K)
--R ?.? : (% , UniversalSegment(Integer)) -> %
--R ?.? : (% , Integer) -> Record(k: Integer, c: K)
--R elt : (% , Integer, Record(k: Integer, c: K)) -> Record(k: Integer, c: K)
--R ?.? : (% , %) -> % if Integer has SGROUP
--R entries : % -> List(Record(k: Integer, c: K))
--R entry? : (Record(k: Integer, c: K), %) -> Boolean if $ has finiteAggregate and Record(k: Integer, c: K) has finiteAggregate
--R euclideanSize : % -> NonNegativeInteger
--R eval : (% , List(Equation(Record(k: Integer, c: K)))) -> % if Record(k: Integer, c: K) has EVALAB(Record(k: Integer, c: K))
--R eval : (% , Equation(Record(k: Integer, c: K))) -> % if Record(k: Integer, c: K) has EVALAB(Record(k: Integer, c: K))
--R eval : (% , Record(k: Integer, c: K), Record(k: Integer, c: K)) -> % if Record(k: Integer, c: K) has EVALAB(Record(k: Integer, c: K))
--R eval : (% , List(Record(k: Integer, c: K)), List(Record(k: Integer, c: K))) -> % if Record(k: Integer, c: K) has EVALAB(Record(k: Integer, c: K))
--R eval : (% , K) -> Stream(K) if K has **: (K, Integer) -> K
--R every? : ((Record(k: Integer, c: K) -> Boolean), %) -> Boolean if $ has finiteAggregate
--R expressIdealMember : (List(% , %) -> Union(List(% , "failed"))
--R exquo : (% , %) -> Union(% , "failed")
--R extendedEuclidean : (% , %, %) -> Union(Record(coef1: % , coef2: % , "failed"))
--R extendedEuclidean : (% , %) -> Record(coef1: % , coef2: % , generator: %)
--R fill! : (% , Record(k: Integer, c: K)) -> % if $ has shallowlyMutable
--R find : ((Record(k: Integer, c: K) -> Boolean), %) -> Union(Record(k: Integer, c: K), "failed")
--R findTerm : (% , Integer) -> Record(k: Integer, c: K)

```

```

--R first : % -> Record(k: Integer,c: K)
--R first : (% ,NonNegativeInteger) -> %
--R first : % -> Record(k: Integer,c: K)
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUniva
--R insert : (Record(k: Integer,c: K),%,Integer) -> %
--R last : % -> Record(k: Integer,c: K)
--R last : (% ,NonNegativeInteger) -> %
--R leaves : % -> List(Record(k: Integer,c: K))
--R less? : (% ,NonNegativeInteger) -> Boolean
--R map : ((Record(k: Integer,c: K) -> Record(k: Integer,c: K)),%) -> %
--R map : (((Record(k: Integer,c: K),Record(k: Integer,c: K)) -> Record(k: Integer,c: K)),%,%) -> %
--R map! : ((Record(k: Integer,c: K) -> Record(k: Integer,c: K)),%) -> % if $ has shallowlyMutab
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (Record(k: Integer,c: K),%) -> Boolean if $ has finiteAggregate and Record(k: Inte
--R members : % -> List(Record(k: Integer,c: K)) if $ has finiteAggregate
--R minIndex : % -> Integer if Integer has ORDSET
--R monomial : (% ,SingletonAsOrderedSet,Integer) -> %
--R monomial : (% ,List(SingletonAsOrderedSet),List(Integer)) -> %
--R monomial2series : (List(%),List(NonNegativeInteger),Integer) -> %
--R more? : (% ,NonNegativeInteger) -> Boolean
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed")
--R multiplyExponents : (% ,PositiveInteger) -> %
--R new : (NonNegativeInteger,Record(k: Integer,c: K)) -> %
--R node? : (% ,%) -> Boolean if Record(k: Integer,c: K) has SETCAT
--R numberOfComputedEntries : % -> NonNegativeInteger
--R orderIfNegative : % -> Union(Integer,"failed")
--R parts : % -> List(Record(k: Integer,c: K)) if $ has finiteAggregate
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %)
--R qelt : (% ,Integer) -> Record(k: Integer,c: K)
--R qsetelt! : (% ,Integer,Record(k: Integer,c: K)) -> Record(k: Integer,c: K) if $ has shallowly
--R reduce : (((Record(k: Integer,c: K),Record(k: Integer,c: K)) -> Record(k: Integer,c: K)),%) -> %
--R reduce : (((Record(k: Integer,c: K),Record(k: Integer,c: K)) -> Record(k: Integer,c: K)),%,R
--R reduce : (((Record(k: Integer,c: K),Record(k: Integer,c: K)) -> Record(k: Integer,c: K)),%,R
--R remove : (Record(k: Integer,c: K),%) -> % if $ has finiteAggregate and Record(k: Integer,c:
--R remove : ((Record(k: Integer,c: K) -> Boolean),%) -> %
--R removeDuplicates : % -> % if $ has finiteAggregate and Record(k: Integer,c: K) has SETCAT
--R rest : (% ,NonNegativeInteger) -> %
--R second : % -> Record(k: Integer,c: K)
--R select : ((Record(k: Integer,c: K) -> Boolean),%) -> %
--R setchildren! : (% ,List(%)) -> % if $ has shallowlyMutable
--R setelt : (% ,value,Record(k: Integer,c: K)) -> Record(k: Integer,c: K) if $ has shallowlyMuta
--R setelt : (% ,first,Record(k: Integer,c: K)) -> Record(k: Integer,c: K) if $ has shallowlyMuta
--R setelt : (% ,rest,%) -> % if $ has shallowlyMutable
--R setelt : (% ,last,Record(k: Integer,c: K)) -> Record(k: Integer,c: K) if $ has shallowlyMutab
--R setelt : (% ,UniversalSegment(Integer),Record(k: Integer,c: K)) -> Record(k: Integer,c: K) if
--R setelt : (% ,Integer,Record(k: Integer,c: K)) -> Record(k: Integer,c: K) if $ has shallowlyMu
--R setfirst! : (% ,Record(k: Integer,c: K)) -> Record(k: Integer,c: K) if $ has shallowlyMutable
--R setlast! : (% ,Record(k: Integer,c: K)) -> Record(k: Integer,c: K) if $ has shallowlyMutable
--R setrest! : (% ,%) -> % if $ has shallowlyMutable
--R setvalue! : (% ,Record(k: Integer,c: K)) -> Record(k: Integer,c: K) if $ has shallowlyMutable

```

```

--R size? : (% , NonNegativeInteger) -> Boolean
--R split! : (% , Integer) -> % if $ has shallowlyMutable
--R subtractIfCan : (% , %) -> Union(% , "failed")
--R swap! : (% , Integer, Integer) -> Void if $ has shallowlyMutable
--R terms : % -> Stream(Record(k: Integer, c: K))
--R third : % -> Record(k: Integer, c: K)
--R truncate : (% , Integer, Integer) -> %
--R unitNormal : % -> Record(unit: % , canonical: % , associate: %)
--R value : % -> Record(k: Integer, c: K)
--R variables : % -> List(SingletonAsOrderedSet)
--R
--E 1

)spool
)lisp (bye)

```

— NeitherSparseOrDensePowerSeries.help —

```

=====
NeitherSparseOrDensePowerSeries examples
=====

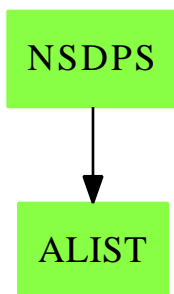
```

This domain is part of the PAFF package

See Also:

o)show NeitherSparseOrDensePowerSeries

NeitherSparseOrDensePowerSeries (NSDPS)



Exports:

0	1	#?
-?	***?	?*?
?+?	?-?	?..?
?.first	?.last	?.rest
?.value	?/?	?=?
?^?	?~=?	?quo?
?rem?	D	any?
approximate	associates?	center
characteristic	charthRoot	child?
children	coefOffFirstNonZeroTerm	coefficient
coerce	complete	concat
concat!	construct	convert
copy	count	cycleEntry
cycleLength	cycleSplit!	cycleTail
cyclic?	degree	delay
delete	differentiate	distance
divide	elt	empty
empty?	entries	entry?
eq?	euclideanSize	eval
every?	explicitEntries?	explicitlyEmpty?
explicitlyFinite?	expressIdealMember	exquo
extend	extendedEuclidean	factor
fill!	filterUpTo	find
findCoef	findTerm	first
frst	gcd	gcdPolynomial
hash	index?	indices
insert	inv	last
latex	lazy?	lazyEvaluate
lcm	leadingCoefficient	leadingMonomial
leaf?	leaves	less?
map	map!	maxIndex
member?	members	minIndex
monomial	monomial2series	monomial?
more?	multiEuclidean	multiplyExponents
new	node?	nodes
numberOfComputedEntries	one?	order
orderIfNegative	parts	pole?
posExpnPart	possiblyInfinite?	prime?
principalIdeal	printInfo	qelt
qsetelt!	recip	reduce
reductum	remove	removeDuplicates
removeFirstZeroes	removeZeroes	rest
rst	sample	sbt
second	select	series
setchildren!	setelt	setfirst!
setlast!	setrest!	setvalue!
shift	size?	sizeLess?
split!	squareFree	squareFreePart
subtractIfCan	swap!	tail
terms	third	truncate
unit?	unitCanonical	unitNormal
value	variable	variables
zero?		

— domain NSDPS NeitherSparseOrDensePowerSeries —

```

)abbrev domain NSDPS NeitherSparseOrDensePowerSeries
++ Authors: Gaetan Hache
++ Date Created: june 1996
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ This domain is part of the PAFF package
NeitherSparseOrDensePowerSeries(K):Exports == Implementation where
  K:Field

SI ==> SingleInteger
INT ==> Integer
TERM ==> Record(k:INT,c:K)
SER ==> Stream(TERM)
NNI ==> NonNegativeInteger

Exports ==> Join(LocalPowerSeriesCategory(K),LazyStreamAggregate(TERM)) with

  findTerm: (%,Integer) -> TERM

Implementation ==> SER add

Rep:=SER

var : Symbol := 't

multC: (K,INT,%) -> %

orderIfNegative(s:%)==
  zero?(s) => "failed"
  f:=fst(s)
  f.k >= 0 => "failed"
  zero?(f.c) => orderIfNegative(rest(s))
  f.k

posExpnPart(s)==
  zero?(s) => 0
  o:=order s
  (o >= 0) => s
  posExpnPart(rst s)

findTerm(s,n)==
  empty?(s) => [n,0]$TERM
  f:=fst(s)
  f.k > n => [n,0]$TERM
  f.k = n => f
  findTerm(rst(s),n)

```

```

findCoef(s,i)==findTerm(s,i).c

coerce(s:%):SER == s::Rep

coerce(s:SER):%==s

localVarForPrintInfo:Boolean:=false()

printInfo==localVarForPrintInfo

printInfo(flag)==localVarForPrintInfo:=flag

outTerm: TERM -> OutputForm

removeZeroes(s)== delay
  zero?(s) => 0
  f:=fst(s)
  zero?(f.c) => removeZeroes(rst(s))
  concat(f,removeZeroes(rst(s)))

inv(ra)==
  a:=removeFirstZeroes ra
  o:=-order(a)
  aa:=shift(a,o)
  aai:=recip aa
  aai case "failed" => _
    error "Big problem in inv function from CreateSeries"
  shift(aai,o)

iDiv: (%,%,K) -> %
iDiv(x,y,ry0) == delay
  empty? x => 0$%
  sx:TERM:=fst x
  c0:K:=ry0 * sx.c
  nT:TERM:=[sx.k, c0]
  tc0:%:=series(sx.k,c0,0$%)
  concat(nT,iDiv(rst x - tc0 * rst y,y,ry0))

recip x ==
  empty? x => "failed"
  rh1:TERM:=fst x
  ^zero?(rh1.k) => "failed"
  ic:K:= inv(rh1.c)
  delay
    concat([0,ic]$TERM,iDiv(- ic * rst x,x,ic))

removeFirstZeroes(s)==
  zero?(s) => 0
  f:=fst(s)
  zero?(f.c) => removeFirstZeroes(rst(s))

```

```

s

sbt(sa,sbb)== delay
  sb:=removeFirstZeroes(sbb)
  o:=order sb
  ^ (o > 0) => _
    error "Cannot substitute by a series of order less than 1 !!!!!"
  empty?(sa) or empty?(sb) => 0
  fa:TERM:=first(sa)
  fb:TERM:=first(sb)
  firstElem:TERM:=[fa.k*fb.k, fa.c*(fb.c**fa.k)]
  zero?(fa.c) => sbt(rst(sa),sb)
  concat(firstElem, rest((fa.c) * sb ** (fa.k)) + sbt(rst(sa),sb) )

coerce(s%):OutputForm==
  zero?(s) => "0" :: OutputForm
  count:SI:= _$streamCount$Lisp
  lstTerm:List TERM:=empty()
  rs%:= s
  for i in 1..count while ^empty?(rs) repeat
    fs:=first rs
    rs:=rst rs
    lstTerm:=concat(lstTerm,fs)
  listOfOutTerm:List OutputForm:=_
    [outTerm(t) for t in lstTerm | ^zero?(t.c) ]
  out:OutputForm:=
    if empty?(listOfOutTerm) then
      "0" :: OutputForm
    else
      reduce("+", listOfOutTerm)
  empty?(rs) => out
  out + ("..." :: OutputForm)

outTerm(t)==
  ee:=t.k
  cc:=t.c
  oe:OutputForm:=ee::OutputForm
  oc:OutputForm:=cc::OutputForm
  symb:OutputForm:= var :: OutputForm
  one?(cc) and one?(ee) => symb
  zero?(ee) => oc
  one?(cc) => symb ** oe
  one?(ee) => oc * symb
  oc * symb ** oe

removeZeroes(n,s)== delay
  n < 0 => s
  zero?(s) => 0
  f:=first(s)
  zero?(f.c) => removeZeroes(n-1, rst(s))

```

```

concat(f,removeZeroes(n-1, rst(s)))

order(s:%)==
  zero?(s) => error _
  "From order (PlaneCurveLocalPowerSeries): cannot compute the order of 0"
f:=fst(s)
zero?(f.c) => order(rest(s))
f.k

monomial2series(lpar,lexp,sh)==
  shift(reduce("*",[s**e for s in lpar for e in lexp]),sh)

coefOfFirstNonZeroTerm(s:%)==
  zero?(s) => error _
  "From order (PlaneCurveLocalPowerSeries): cannot find the coefOfFirstNonZeroTerm"
f:=fst(s)
zero?(f.c) => coefOfFirstNonZeroTerm(rest(s))
f.c

degreeOfTermLower?: (TERM,INT) -> Boolean
degreeOfTermLower?(t,n)== t.k < n

filterUpTo(s,n)==filterWhile(degreeOfTermLower?(#1,n),s)

series(exp,coef,s)==cons([exp,coef]$TERM,s)

a:% ** n:NNI == -- delay
  zero?(n) => 1
  expt(a,n :: PositiveInteger)$RepeatedSquaring(%)

0 == empty()

1 == construct([[0,1]$TERM])

zero?(a)==empty?(a::Rep)

shift(s,n)== delay
  zero?(s) => 0
  fs:=fst(s)
  es:=fs.k
  concat([es+n,fs.c]$TERM,shift(rest(s),n))

a:% + b:% == delay
  zero?(a) => b
  zero?(b) => a
  fa:=fst(a)
  fb:=fst(b)
  ea:=fa.k
  eb:=fb.k
  nc:K

```

```

ea = eb => concat([ea,fa.c+fb.c]$TERM,rest(a) + rest(b))
ea > eb => concat([eb,fb.c]$TERM,a + rest(b))
eb > ea => concat([ea,fa.c]$TERM,rest(a) + b)

- a:% == --delay
  multC( (-1) :: K , 0 , a)

a:% - b:% == --delay
a+(-b)

multC(coef,n,s)== delay
  zero?(coef) => 0
  zero?(s) => 0
  f:=first(s)
  concat([f.k+n,coef*f.c]$TERM,multC(coef,n,rest(s)))

coef:K * s:% == delay
  zero?(coef) => 0
  zero?(s) => 0
  f:=first(s)
  concat([f.k,coef*f.c]$TERM, coef *s% rest(s))

s:% * coef:K == coef * s

s1:% * s2:%== delay
  zero?(s1) or zero?(s2) => 0
  f1:TERM:=first(s1)
  f2:TERM:=first(s2)
  e1:INT:=f1.k; e2:INT:=f2.k
  c1:K:=f1.c;   c2:K:=f2.c
  concat([e1+e2,c1*c2]$TERM,_
    multC(c1,e1,rest(s2))+multC(c2,e2,rest(s1))+rest(s1)*rest(s2))

```

— NSDPS.dotabb —

```

"NSDPS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=NSDPS"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NSDPS" -> "ALIST"

```

domain NSMP NewSparseMultivariatePolynomial

Based on the **PseudoRemainderSequence** package, the domain constructor **NewSparseMultivariatePolynomial** extends the constructor **SparseMultivariatePolynomial**. It also provides some additional operations related to polynomial system solving by means of triangular sets.

— NewSparseMultivariatePolynomial.input —

```
)set break resume
)sys rm -f NewSparseMultivariatePolynomial.output
)spool NewSparseMultivariatePolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NewSparseMultivariatePolynomial
--R
--R NewSparseMultivariatePolynomial(R: Ring,VarSet: OrderedSet) is a domain constructor
--R Abbreviation for NewSparseMultivariatePolynomial is NSMP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for NSMP
--R
--R----- Operations -----
--R ?? : (%,R) -> %
--R ?? : (%,%) -> %
--R ?? : (PositiveInteger,%) -> %
--R ?+? : (%,%) -> %
--R -? : % -> %
--R D : (%,List(VarSet)) -> %
--R 1 : () -> %
--R ?? : (%,PositiveInteger) -> %
--R coerce : VarSet -> %
--R coerce : Integer -> %
--R deepestInitial : % -> %
--R differentiate : (%,VarSet) -> %
--R eval : (%,VarSet,R) -> %
--R eval : (%,Equation(%)) -> %
--R ground? : % -> Boolean
--R head : % -> %
--R headReduced? : (%,%) -> Boolean
--R init : % -> %
--R iteratedInitials : % -> List(%)
--R lazyPquo : (%,%,VarSet) -> %
--R lazyPrem : (%,%,VarSet) -> %
--R leadingCoefficient : % -> R
--R leastMonomial : % -> %
--R mainMonomial : % -> %
--R ?? : (R,%) -> %
--R ?? : (Integer,%) -> %
--R *** : (%,PositiveInteger) -> %
--R ?-? : (%,%) -> %
--R ?=? : (%,%) -> Boolean
--R D : (%,VarSet) -> %
--R 0 : () -> %
--R coefficients : % -> List(R)
--R coerce : R -> %
--R coerce : % -> OutputForm
--R deepestTail : % -> %
--R eval : (%,VarSet,%) -> %
--R eval : (%,%,%) -> %
--R ground : % -> R
--R hash : % -> SingleInteger
--R headReduce : (%,%) -> %
--R infRittWu? : (%,%) -> Boolean
--R initiallyReduce : (%,%) -> %
--R latex : % -> String
--R lazyPquo : (%,%) -> %
--R lazyPrem : (%,%) -> %
--R leadingMonomial : % -> %
--R mainCoefficients : % -> List(%)
--R mainMonomials : % -> List(%)
```



```

--R map : ((R -> R),%) -> %
--R monic? : % -> Boolean
--R monomial? : % -> Boolean
--R mvar : % -> VarSet
--R one? : % -> Boolean
--R pquo : (%,%) -> %
--R prem : (%,%) -> %
--R recip : % -> Union(%, "failed")
--R reductum : (%, VarSet) -> %
--R retract : % -> VarSet
--R sample : () -> %
--R tail : % -> %
--R zero? : % -> Boolean
--R ?? : (Fraction(Integer),%) -> % if R has ALGEBRA(FRAC(INT))
--R ?? : (%, Fraction(Integer)) -> % if R has ALGEBRA(FRAC(INT))
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%, NonNegativeInteger) -> %
--R ?/? : (%, R) -> % if R has FIELD
--R ?<? : (%,%) -> Boolean if R has ORDSET
--R ?<=? : (%,%) -> Boolean if R has ORDSET
--R ?>? : (%,%) -> Boolean if R has ORDSET
--R ?>=? : (%,%) -> Boolean if R has ORDSET
--R D : (%, List(VarSet), List(NonNegativeInteger)) -> %
--R D : (%, VarSet, NonNegativeInteger) -> %
--R LazardQuotient : (%,%, NonNegativeInteger) -> % if R has INTDOM
--R LazardQuotient2 : (%,%,%, NonNegativeInteger) -> % if R has INTDOM
--R RittWuCompare : (%,%) -> Union(Boolean, "failed")
--R ?? : (%, NonNegativeInteger) -> %
--R associates? : (%,%) -> Boolean if R has INTDOM
--R binomThmExpt : (%,%, NonNegativeInteger) -> % if R has COMRING
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if $ has CHARNZ and R has PFECAT or R has CHARNZ
--R coefficient : (%, List(VarSet), List(NonNegativeInteger)) -> %
--R coefficient : (%, VarSet, NonNegativeInteger) -> %
--R coefficient : (%, IndexedExponents(VarSet)) -> R
--R coerce : % -> % if R has INTDOM
--R coerce : Fraction(Integer) -> % if R has ALGEBRA(FRAC(INT)) or R has RETRACT(FRAC(INT))
--R coerce : SparseMultivariatePolynomial(R, VarSet) -> %
--R coerce : % -> SparseMultivariatePolynomial(R, VarSet)
--R coerce : % -> Polynomial(R) if VarSet has KONVERT(SYMBOL)
--R conditionP : Matrix(%) -> Union(Vector(%), "failed") if $ has CHARNZ and R has PFECAT
--R content : (%, VarSet) -> % if R has GCDDOM
--R content : % -> R if R has GCDDOM
--R convert : % -> Polynomial(R) if VarSet has KONVERT(SYMBOL)
--R convert : % -> String if R has RETRACT(INT) and VarSet has KONVERT(SYMBOL)
--R convert : Polynomial(R) -> % if VarSet has KONVERT(SYMBOL)
--R convert : Polynomial(Integer) -> % if R has ALGEBRA(INT) and VarSet has KONVERT(SYMBOL) and
--R convert : Polynomial(Fraction(Integer)) -> % if R has ALGEBRA(FRAC(INT)) and VarSet has KONV
--R convert : % -> InputForm if R has KONVERT(INFORM) and VarSet has KONVERT(INFORM)
--R convert : % -> Pattern(Integer) if R has KONVERT(PATTERN(INT)) and VarSet has KONVERT(PATTER

```

```

--R convert : % -> Pattern(Float) if R has KONVERT(PATTERN(FLOAT)) and VarSet has KONVERT(PATTERN(FLOAT))
--R degree : (% ,List(VarSet)) -> List(NonNegativeInteger)
--R degree : (% ,VarSet) -> NonNegativeInteger
--R degree : % -> IndexedExponents(VarSet)
--R differentiate : (% ,List(VarSet),List(NonNegativeInteger)) -> %
--R differentiate : (% ,VarSet,NonNegativeInteger) -> %
--R differentiate : (% ,List(VarSet)) -> %
--R discriminant : (% ,VarSet) -> % if R has COMRING
--R eval : (% ,List(VarSet),List(%)) -> %
--R eval : (% ,List(VarSet),List(R)) -> %
--R eval : (% ,List(%),List(%)) -> %
--R eval : (% ,List(Equation(%))) -> %
--R exactQuotient : (% ,%) -> % if R has INTDOM
--R exactQuotient : (% ,R) -> % if R has INTDOM
--R exactQuotient! : (% ,%) -> % if R has INTDOM
--R exactQuotient! : (% ,R) -> % if R has INTDOM
--R exquo : (% ,%) -> Union(%,"failed") if R has INTDOM
--R exquo : (% ,R) -> Union(%,"failed") if R has INTDOM
--R extendedSubResultantGcd : (% ,%) -> Record(gcd: %,coef1: %,coef2: %) if R has INTDOM
--R factor : % -> Factored(%) if R has PFECAT
--R factorPolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if R has PF
--R factorSquareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if
--R gcd : (% ,%) -> % if R has GCDDOM
--R gcd : List(%) -> % if R has GCDDOM
--R gcd : (R,%) -> R if R has GCDDOM
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynom
--R halfExtendedSubResultantGcd1 : (% ,%) -> Record(gcd: %,coef1: %) if R has INTDOM
--R halfExtendedSubResultantGcd2 : (% ,%) -> Record(gcd: %,coef2: %) if R has INTDOM
--R headReduced? : (% ,List(%)) -> Boolean
--R initiallyReduced? : (% ,List(%)) -> Boolean
--R initiallyReduced? : (% ,%) -> Boolean
--R isExpt : % -> Union(Record(var: VarSet,exponent: NonNegativeInteger),"failed")
--R isPlus : % -> Union(List(%),"failed")
--R isTimes : % -> Union(List(%),"failed")
--R lastSubResultant : (% ,%) -> % if R has INTDOM
--R lazyPremWithDefault : (% ,%,VarSet) -> Record(coef: %,gap: NonNegativeInteger,remainder: %)
--R lazyPremWithDefault : (% ,%) -> Record(coef: %,gap: NonNegativeInteger,remainder: %)
--R lazyPseudoDivide : (% ,%,VarSet) -> Record(coef: %,gap: NonNegativeInteger,quotient: %,remainder: %)
--R lazyPseudoDivide : (% ,%) -> Record(coef: %,gap: NonNegativeInteger,quotient: %,remainder: %)
--R lazyResidueClass : (% ,%) -> Record(polnum: %,polden: %,power: NonNegativeInteger)
--R lcm : (% ,%) -> % if R has GCDDOM
--R lcm : List(%) -> % if R has GCDDOM
--R leadingCoefficient : (% ,VarSet) -> %
--R mainContent : % -> % if R has GCDDOM
--R mainPrimitivePart : % -> % if R has GCDDOM
--R mainSquareFreePart : % -> % if R has GCDDOM
--R mainVariable : % -> Union(VarSet,"failed")
--R mapExponents : ((IndexedExponents(VarSet) -> IndexedExponents(VarSet)),%) -> %
--R max : (% ,%) -> % if R has ORDSET
--R min : (% ,%) -> % if R has ORDSET

```

```

--R minimumDegree : (% , List(VarSet)) -> List(NonNegativeInteger)
--R minimumDegree : (% , VarSet) -> NonNegativeInteger
--R minimumDegree : % -> IndexedExponents(VarSet)
--R monicDivide : (% , % , VarSet) -> Record(quotient: % , remainder: %)
--R monomial : (% , List(VarSet) , List(NonNegativeInteger)) -> %
--R monomial : (% , VarSet , NonNegativeInteger) -> %
--R monomial : (R , IndexedExponents(VarSet)) -> %
--R multivariate : (SparseUnivariatePolynomial(% ) , VarSet) -> %
--R multivariate : (SparseUnivariatePolynomial(R) , VarSet) -> %
--R nextsubResultant2 : (% , % , % , %) -> % if R has INTDOM
--R normalized? : (% , List(%)) -> Boolean
--R numberOfMonomials : % -> NonNegativeInteger
--R patternMatch : (% , Pattern(Integer) , PatternMatchResult(Integer , %)) -> PatternMatchResult(Integer , %)
--R patternMatch : (% , Pattern(Float) , PatternMatchResult(Float , %)) -> PatternMatchResult(Float , %)
--R pomopo! : (% , R , IndexedExponents(VarSet) , %) -> %
--R primPartElseUnitCanonical : % -> % if R has INTDOM
--R primPartElseUnitCanonical! : % -> % if R has INTDOM
--R prime? : % -> Boolean if R has PFECAT
--R primitiveMonomials : % -> List(% )
--R primitivePart : (% , VarSet) -> % if R has GCDDOM
--R primitivePart : % -> % if R has GCDDOM
--R primitivePart! : % -> % if R has GCDDOM
--R pseudoDivide : (% , %) -> Record(quotient: % , remainder: %)
--R reduced? : (% , List(%)) -> Boolean
--R reducedSystem : Matrix(% ) -> Matrix(R)
--R reducedSystem : (Matrix(% ) , Vector(%)) -> Record(mat: Matrix(R) , vec: Vector(R))
--R reducedSystem : (Matrix(% ) , Vector(%)) -> Record(mat: Matrix(Integer) , vec: Vector(Integer)) if R has LINEXP(INT)
--R reducedSystem : Matrix(% ) -> Matrix(Integer) if R has LINEXP(INT)
--R resultant : (% , %) -> % if R has INTDOM
--R resultant : (% , % , VarSet) -> % if R has COMRING
--R retract : % -> SparseMultivariatePolynomial(R , VarSet)
--R retract : Polynomial(R) -> % if VarSet has KONVERT(SYMBOL) and not(has(R , Algebra(Fraction(Integer) , VarSet)))
--R retract : Polynomial(Integer) -> % if R has ALGEBRA(INT) and VarSet has KONVERT(SYMBOL) and not(has(R , Algebra(Integer) , VarSet))
--R retract : Polynomial(Fraction(Integer)) -> % if R has ALGEBRA(FRAC(INT)) and VarSet has KONVERT(FRAC(INT))
--R retract : % -> Integer if R has RETRACT(INT)
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(SparseMultivariatePolynomial(R , VarSet) , "failed")
--R retractIfCan : Polynomial(R) -> Union(% , "failed") if VarSet has KONVERT(SYMBOL) and not(has(R , Algebra(Fraction(Integer) , VarSet)))
--R retractIfCan : Polynomial(Integer) -> Union(% , "failed") if R has ALGEBRA(INT) and VarSet has KONVERT(SYMBOL) and not(has(R , Algebra(Integer) , VarSet))
--R retractIfCan : Polynomial(Fraction(Integer)) -> Union(% , "failed") if R has ALGEBRA(FRAC(INT)) and VarSet has KONVERT(FRAC(INT))
--R retractIfCan : % -> Union(VarSet , "failed")
--R retractIfCan : % -> Union(Integer , "failed") if R has RETRACT(INT)
--R retractIfCan : % -> Union(Fraction(Integer) , "failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(R , "failed")
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(%)) , SparseUnivariatePolynomial(%)) -> List(SparseUnivariatePolynomial(%))
--R squareFree : % -> Factored(% ) if R has GCDDOM
--R squareFreePart : % -> % if R has GCDDOM
--R squareFreePolynomial : SparseUnivariatePolynomial(% ) -> Factored(SparseUnivariatePolynomial(%))
--R subResultantChain : (% , %) -> List(% ) if R has INTDOM
--R subResultantGcd : (% , %) -> % if R has INTDOM

```

```

--R subtractIfCan : (%,% ) -> Union(%, "failed")
--R totalDegree : (% , List(VarSet)) -> NonNegativeInteger
--R totalDegree : % -> NonNegativeInteger
--R unit? : % -> Boolean if R has INTDOM
--R unitCanonical : % -> % if R has INTDOM
--R unitNormal : % -> Record(unit: %, canonical: %, associate: %) if R has INTDOM
--R univariate : % -> SparseUnivariatePolynomial(R)
--R univariate : (% , VarSet) -> SparseUnivariatePolynomial(%)
--R
--E 1

)spool
)lisp (bye)

```

— NewSparseMultivariatePolynomial.help —

=====

NewSparseMultivariatePolynomial examples

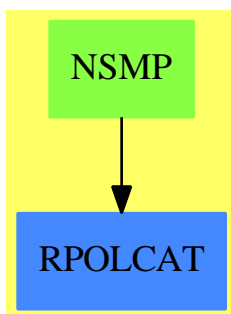
=====

A post-facto extension for SMP in order to speed up operations related to pseudo-division and gcd. This domain is based on the NSUP constructor which is itself a post-facto extension of the SUP constructor.

See Also:

o)show NewSparseMultivariatePolynomial

NewSparseMultivariatePolynomial (NSMP)



See

⇒ “NewSparseUnivariatePolynomial” (NSUP) [15](#) on page [1728](#)

Exports:

0	1	associates?
binomThmExpt	characteristic	charthRoot
coefficient	coefficients	coerce
conditionP	content	D
degree	deepestInitial	deepestTail
differentiate	discriminant	eval
exactQuotient	exactQuotient!	exquo
extendedSubResultantGcd	factor	factorPolynomial
factorSquareFreePolynomial	gcd	gcdPolynomial
halfExtendedSubResultantGcd1	ground	ground?
halfExtendedSubResultantGcd2	hash	head
headReduce	headReduced?	infRittWu?
init	initiallyReduce	initiallyReduced?
isExpt	isPlus	isTimes
iteratedInitials	lastSubResultant	latex
LazardQuotient	LazardQuotient2	lazyPremWithDefault
lazyPquo	lazyPrem	lazyPseudoDivide
lazyResidueClass	lcm	leadingCoefficient
leadingMonomial	leastMonomial	mainCoefficients
mainContent	mainMonomial	mainMonomials
mainPrimitivePart	mainSquareFreePart	mainVariable
map	mapExponents	max
mdeg	min	minimumDegree
monic?	monicDivide	monicModulo
monomial	monomial?	monomials
multivariate	mvar	nextsubResultant2
normalized?	numberOfMonomials	one?
patternMatch	popopo!	pquo
primPartElseUnitCanonical!	primPartElseUnitCanonical	prem
prime?	primitiveMonomials	primitivePart
primitivePart!	pseudoDivide	reducedSystem
resultant	retract	retractIfCan
RittWuCompare	quasiMonic?	recip
reduced?	reductum	retract
sample	solveLinearPolynomialEquation	squareFree
squareFreePart	squareFreePolynomial	subResultantChain
subResultantGcd	supRittWu?	subtractIfCan
tail	totalDegree	unit?
unitCanonical	unitNormal	univariate
univariate	variables	zero?
?*?	?**?	?+?
?-?	-?	?=?
?^?	?~=?	?/?
?<?	?<=?	?>?
?>=?		

— domain NSMP NewSparseMultivariatePolynomial —

```

)abbrev domain NSMP NewSparseMultivariatePolynomial
++ Author: Marc Moreno Maza
++ Date Created: 22/04/94
++ Date Last Updated: 14/12/1998
++ Description:
++ A post-facto extension for \axiomType{SMP} in order
++ to speed up operations related to pseudo-division and gcd.
++ This domain is based on the \axiomType{NSUP} constructor which is
++ itself a post-facto extension of the \axiomType{SUP} constructor.

NewSparseMultivariatePolynomial(R,VarSet) : Exports == Implementation where
  R:Ring
  VarSet:OrderedSet
  N ==> NonNegativeInteger
  Z ==> Integer
  SUP ==> NewSparseUnivariatePolynomial
  SMPR ==> SparseMultivariatePolynomial(R, VarSet)
  SUP2 ==> NewSparseUnivariatePolynomialFunctions2($,$)

Exports == Join(RecursivePolynomialCategory(R,IndexedExponents VarSet,_
  VarSet), CoercibleTo(SMPR),RetractableTo(SMPR))

Implementation == SparseMultivariatePolynomial(R, VarSet) add

  D := NewSparseUnivariatePolynomial($)
  VPoly:= Record(v:VarSet,ts:D)
  Rep:= Union(R,VPoly)

--local function
  PSimp: (D,VarSet) -> %

  PSimp(up,mv) ==
    if degree(up) = 0 then leadingCoefficient(up) else [mv,up]$VPoly

  coerce (p:$):SMPR ==
    p pretend SMPR

  coerce (p:SMPR):$ ==
    p pretend $

  retractIfCan (p:$) : Union(SMPR,"failed") ==
    (p pretend SMPR)::Union(SMPR,"failed")

  mvar p ==
    p case R => error" Error in mvar from NSMP : #1 has no variables."
    p.v

```

```

mdeg p ==
  p case R => 0$N
  degree(p.ts)$D

init p ==
  p case R => error" Error in init from NSMP : #1 has no variables."
  leadingCoefficient(p.ts)$D

head p ==
  p case R => p
  ([p.v,leadingMonomial(p.ts)$D]$VPoly)::Rep

tail p ==
  p case R => 0$$
  red := reductum(p.ts)$D
  ground?(red)$D => (ground(red)$D)::Rep
  ([p.v,red]$VPoly)::Rep

iteratedInitials p ==
  p case R => []
  p := leadingCoefficient(p.ts)$D
  cons(p,iteratedInitials(p))

localDeepestInitial (p : $) : $ ==
  p case R => p
  localDeepestInitial leadingCoefficient(p.ts)$D

deepestInitial p ==
  p case R =>
    error"Error in deepestInitial from NSMP : #1 has no variables."
  localDeepestInitial leadingCoefficient(p.ts)$D

mainMonomial p ==
  zero? p =>
    error"Error in mainMonomial from NSMP : the argument is zero"
  p case R => 1$$
  monomial(1$$,p.v,degree(p.ts)$D)

leastMonomial p ==
  zero? p =>
    error"Error in leastMonomial from NSMP : the argument is zero"
  p case R => 1$$
  monomial(1$$,p.v,minimumDegree(p.ts)$D)

mainCoefficients p ==
  zero? p =>
    error"Error in mainCoefficients from NSMP : the argument is zero"
  p case R => [p]
  coefficients(p.ts)$D

```

```

leadingCoefficient(p:$,x:VarSet):$ ==
  (p case R) => p
  p.v = x => leadingCoefficient(p.ts)$D
  zero? (d := degree(p,x)) => p
  coefficient(p,x,d)

localMonicModulo(a:$,b:$):$ ==
  -- b is assumed to have initial 1
  a case R => a
  a.v < b.v => a
  mM: $
  if a.v > b.v
  then
    m : D := map((a1:%):% +-> localMonicModulo(a1,b),a.ts)$SUP2
  else
    m : D := monicModulo(a.ts,b.ts)$D
  if ground?(m)$D
  then
    mM := (ground(m)$D)::Rep
  else
    mM := ([a.v,m]$VPoly)::Rep
  mM

monicModulo (a,b) ==
  b case R => error"Error in monicModulo from NSMP : #2 is constant"
  ib : $ := init(b)$
  not ground?(ib)$ =>
    error"Error in monicModulo from NSMP : #2 is not monic"
  mM : $
  --
  if not one?(ib)$
  if not ((ib) = 1)$
  then
    r : R := ground(ib)$
    rec : Union(R,"failed"):= recip(r)$R
    (rec case "failed") =>
      error"Error in monicModulo from NSMP : #2 is not monic"
    a case R => a
    a := (rec::R) * a
    b := (rec::R) * b
    mM := ib * localMonicModulo (a,b)
  else
    mM := localMonicModulo (a,b)
  mM

prem(a:$, b:$): $ ==
  -- with pseudoRemainder$NSUP
  b case R =>
    error "in prem$NSMP: ground? #2"
  db: N := degree(b.ts)$D
  lcb: $ := leadingCoefficient(b.ts)$D

```



```

test: Z := degree(a,b.v)::Z - db
delta: Z := max(test + 1$Z, 0$Z)
(a case R) or (a.v < b.v) => lcb ** (delta::N) * a
a.v = b.v =>
  r: D := pseudoRemainder(a.ts,b.ts)$D
  ground?(r) => return (ground(r)$D)::Rep
  ([a.v,r]$VPoly)::Rep
while not zero?(a) and not negative?(test) repeat
  term := monomial(leadingCoefficient(a,b.v),b.v,test::N)
  a := lcb * a - term * b
  delta := delta - 1$Z
  test := degree(a,b.v)::Z - db
lcb ** (delta::N) * a

pquo (a:$, b:$) : $ ==
  cPS := lazyPseudoDivide (a,b)
  c := (cPS.coef) ** (cPS.gap)
  c * cPS.quotient

pseudoDivide(a:$, b:$): Record (quotient : $, remainder : $) ==
  -- from RPOLCAT
  cPS := lazyPseudoDivide(a,b)
  c := (cPS.coef) ** (cPS.gap)
  [c * cPS.quotient, c * cPS.remainder]

lazyPrem(a:$, b:$): $ ==
  -- with lazyPseudoRemainder$NSUP
  -- Uses leadingCoefficient: ($, V) -> $
  b case R =>
    error "in lazyPrem$NSMP: ground? #2"
  (a case R) or (a.v < b.v) => a
  a.v = b.v => PSimp(lazyPseudoRemainder(a.ts,b.ts)$D,a.v)
  db: N := degree(b.ts)$D
  lcb: $ := leadingCoefficient(b.ts)$D
  test: Z := degree(a,b.v)::Z - db
  while not zero?(a) and not negative?(test) repeat
    term := monomial(leadingCoefficient(a,b.v),b.v,test::N)
    a := lcb * a - term * b
    test := degree(a,b.v)::Z - db
  a

lazyPquo (a:$, b:$) : $ ==
  -- with lazyPseudoQuotient$NSUP
  b case R =>
    error " in lazyPquo$NSMP: #2 is conctant"
  (a case R) or (a.v < b.v) => 0
  a.v = b.v => PSimp(lazyPseudoQuotient(a.ts,b.ts)$D,a.v)
  db: N := degree(b.ts)$D
  lcb: $ := leadingCoefficient(b.ts)$D
  test: Z := degree(a,b.v)::Z - db

```

```

q := 0$$
test: Z := degree(a,b.v)::Z - db
while not zero?(a) and not negative?(test) repeat
  term := monomial(leadingCoefficient(a,b.v),b.v,test::N)
  a := lcb * a - term * b
  q := lcb * q + term
  test := degree(a,b.v)::Z - db
q

lazyPseudoDivide(a:$, b:$): Record(coef:$, gap: N,quotient:$, remainder:$) ==
-- with lazyPseudoDivide$NSUP
b case R =>
  error " in lazyPseudoDivide$NSMP: #2 is constant"
(a case R) or (a.v < b.v) => [1$$,0$N,0$$,a]
a.v = b.v =>
  cgqr := lazyPseudoDivide(a.ts,b.ts)
  [cgqr.coef, cgqr.gap, PSimp(cgqr.quotient,a.v), PSimp(cgqr.remainder,a.v)]
db: N := degree(b.ts)$D
lcb: $ := leadingCoefficient(b.ts)$D
test: Z := degree(a,b.v)::Z - db
q := 0$$
delta: Z := max(test + 1$Z, 0$Z)
while not zero?(a) and not negative?(test) repeat
  term := monomial(leadingCoefficient(a,b.v),b.v,test::N)
  a := lcb * a - term * b
  q := lcb * q + term
  delta := delta - 1$Z
  test := degree(a,b.v)::Z - db
[lcb, (delta::N), q, a]

lazyResidueClass(a:$, b:$): Record(polnum:$, polden:$, power:N) ==
-- with lazyResidueClass$NSUP
b case R =>
  error " in lazyResidueClass$NSMP: #2 is constant"
lcb: $ := leadingCoefficient(b.ts)$D
(a case R) or (a.v < b.v) => [a,lcb,0]
a.v = b.v =>
  lrc := lazyResidueClass(a.ts,b.ts)$D
  [PSimp(lrc.polnum,a.v), lrc.polden, lrc.power]
db: N := degree(b.ts)$D
test: Z := degree(a,b.v)::Z - db
pow: N := 0
while not zero?(a) and not negative?(test) repeat
  term := monomial(leadingCoefficient(a,b.v),b.v,test::N)
  a := lcb * a - term * b
  pow := pow + 1
  test := degree(a,b.v)::Z - db
[a, lcb, pow]

if R has IntegralDomain

```

```

then

packD := PseudoRemainderSequence($,D)

exactQuo(x:$, y:$):$ ==
  ex: Union($,"failed") := x exquo$$ y
  (ex case $) => ex::$
  error "in exactQuotient$NSMP: bad args"

LazardQuotient(x:$, y:$, n: N):$ ==
  zero?(n) => error("LazardQuotient$NSMP : n = 0")
  one?(n) => x
  (n = 1) => x
  a: N := 1
  while n >= (b := 2*a) repeat a := b
  c: $ := x
  n := (n - a)::N
  repeat
    one?(a) => return c
    (a = 1) => return c
    a := a quo 2
    c := exactQuo(c*c,y)
    if n >= a then ( c := exactQuo(c*x,y) ; n := (n - a)::N )

LazardQuotient2(p:$, a:$, b:$, n: N) ==
  zero?(n) => error " in LazardQuotient2$NSMP: bad #4"
  one?(n) => p
  (n = 1) => p
  c: $ := LazardQuotient(a,b,(n-1)::N)
  exactQuo(c*p,b)

next_subResultant2(p:$, q:$, z:$, s:$) ==
  PSimp(next_sousResultant2(p.ts,q.ts,z.ts,s)$packD,p.v)

subResultantGcd(a:$, b:$): $ ==
  (a case R) or (b case R) =>
    error "subResultantGcd$NSMP: one arg is constant"
  a.v ~= b.v =>
    error "subResultantGcd$NSMP: mvar(#1) ~= mvar(#2)"
  PSimp(subResultantGcd(a.ts,b.ts),a.v)

halfExtendedSubResultantGcd1(a:$,b:$): Record (gcd: $, coef1: $) ==
  (a case R) or (b case R) =>
    error "halfExtendedSubResultantGcd1$NSMP: one arg is constant"
  a.v ~= b.v =>
    error "halfExtendedSubResultantGcd1$NSMP: mvar(#1) ~= mvar(#2)"
  hesrg := halfExtendedSubResultantGcd1(a.ts,b.ts)$D
  [PSimp(hesrg.gcd,a.v), PSimp(hesrg.coef1,a.v)]

halfExtendedSubResultantGcd2(a:$,b:$): Record (gcd: $, coef2: $) ==

```

```

(a case R) or (b case R) =>
  error "halfExtendedSubResultantGcd2$NSMP: one arg is constant"
a.v ~= b.v =>
  error "halfExtendedSubResultantGcd2$NSMP: mvar(#1) ~= mvar(#2)"
hesrg := halfExtendedSubResultantGcd2(a.ts,b.ts)$D
[PSimp(hesrg.gcd,a.v), PSimp(hesrg.coef2,a.v)]

extendedSubResultantGcd(a:$,b:$): Record (gcd: $, coef1: $, coef2: $) ==
(a case R) or (b case R) =>
  error "extendedSubResultantGcd$NSMP: one arg is constant"
a.v ~= b.v =>
  error "extendedSubResultantGcd$NSMP: mvar(#1) ~= mvar(#2)"
esrg := extendedSubResultantGcd(a.ts,b.ts)$D
[PSimp(esrg.gcd,a.v),PSimp(esrg.coef1,a.v),PSimp(esrg.coef2,a.v)]

resultant(a:$, b:$): $ ==
(a case R) or (b case R) =>
  error "resultant$NSMP: one arg is constant"
a.v ~= b.v =>
  error "resultant$NSMP: mvar(#1) ~= mvar(#2)"
resultant(a.ts,b.ts)$D

subResultantChain(a:$, b:$): List $ ==
(a case R) or (b case R) =>
  error "subResultantChain$NSMP: one arg is constant"
a.v ~= b.v =>
  error "subResultantChain$NSMP: mvar(#1) ~= mvar(#2)"
[PSimp(up,a.v) for up in subResultantsChain(a.ts,b.ts)]

lastSubResultant(a:$, b:$): $ ==
(a case R) or (b case R) =>
  error "lastSubResultant$NSMP: one arg is constant"
a.v ~= b.v =>
  error "lastSubResultant$NSMP: mvar(#1) ~= mvar(#2)"
PSimp(lastSubResultant(a.ts,b.ts),a.v)

if R has EuclideanDomain
then

  exactQuotient (a:$,b:R) ==
--
  one? b => a
  (b = 1) => a
  a case R => (a::R quo$R b)::R
  ([a.v, map((a1:R):% +-> exactQuotient(a1,b),a.ts)$SUP2]$VPoly)::Rep

  exactQuotient! (a:$,b:R) ==
--
  one? b => a
  (b = 1) => a
  a case R => (a::R quo$R b)::R
  a.ts := map((a1:R):% +-> exactQuotient!(a1,b),a.ts)$SUP2

```

```

      a
    else
      exactQuotient (a:$,b:R) ==
--      one? b => a
      (b = 1) => a
      a case R => ((a::R exquo$R b)::R)::R
      ([a.v, map((a1:%):% +-> exactQuotient(a1,b),a.ts)$SUP2]$VPoly)::Rep

      exactQuotient! (a:$,b:R) ==
--      one? b => a
      (b = 1) => a
      a case R => ((a::R exquo$R b)::R)::R
      a.ts := map((a1:%):% +-> exactQuotient!(a1,b),a.ts)$SUP2
      a

    if R has GcdDomain
    then

      localGcd(r:R,p:$):R ==
      p case R => gcd(r,p::R)$R
      gcd(r,content(p))$R

      gcd(r:R,p:$):R ==
--      one? r => r
      (r = 1) => r
      zero? p => r
      localGcd(r,p)

      content p ==
      p case R => p
      up : D := p.ts
      r := 0$R
--      while (not zero? up) and (not one? r) repeat
      while (not zero? up) and (not (r = 1)) repeat
      r := localGcd(r,leadingCoefficient(up))
      up := reductum up
      r

      primitivePart! p ==
      zero? p => p
      p case R => 1$$
      cp := content(p)
      p.ts :=
      unitCanonical(map((a1:%):% +-> exactQuotient!(a1,cp),p.ts)$SUP2)$D
      p

```

— NSMP.dotabb —

```
"NSMP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=NSMP"]
"RPOLCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RPOLCAT"]
"NSMP" -> "RPOLCAT"
```

domain NSUP NewSparseUnivariatePolynomial

Based on the **PseudoRemainderSequence** package, the domain constructor **NewSparseUnivariatePolynomial** extends the constructor **SparseUnivariatePolynomial**.

— NewSparseUnivariatePolynomial.input —

```
)set break resume
)sys rm -f NewSparseUnivariatePolynomial.output
)spool NewSparseUnivariatePolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NewSparseUnivariatePolynomial
--R
--R NewSparseUnivariatePolynomial(R: Ring) is a domain constructor
--R Abbreviation for NewSparseUnivariatePolynomial is NSUP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for NSUP
--R
--R----- Operations -----
--R ?? : (%,R) -> %           ?? : (R,%) -> %
--R ?? : (%,%) -> %           ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R ?? : (%,%) -> %           ?-? : (%,%) -> %
--R -? : % -> %               ?=? : (%,%) -> Boolean
--R D : (%,(R -> R)) -> %       D : % -> %
--R D : (%,NonNegativeInteger) -> %   1 : () -> %
--R 0 : () -> %               ?? : (%,PositiveInteger) -> %
--R coefficients : % -> List(R)   coerce : R -> %
--R coerce : Integer -> %         coerce : % -> OutputForm
--R degree : % -> NonNegativeInteger   differentiate : % -> %
--R .? : (%,%) -> %             ?.? : (%,R) -> R
--R eval : (%,%,%) -> %         eval : (%,Equation(%)) -> %
--R ground : % -> R             ground? : % -> Boolean
```

```

--R hash : % -> SingleInteger
--R latex : % -> String
--R leadingCoefficient : % -> R
--R map : ((R -> R),%) -> %
--R monomial? : % -> Boolean
--R one? : % -> Boolean
--R recip : % -> Union(%, "failed")
--R retract : % -> R
--R zero? : % -> Boolean
--R ?? : (Fraction(Integer),%) -> % if R has ALGEBRA(FRAC(INT))
--R ?? : (%, Fraction(Integer)) -> % if R has ALGEBRA(FRAC(INT))
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%, NonNegativeInteger) -> %
--R ?/? : (%, R) -> % if R has FIELD
--R ?<? : (%,%) -> Boolean if R has ORDSET
--R ?<=? : (%,%) -> Boolean if R has ORDSET
--R ?>? : (%,%) -> Boolean if R has ORDSET
--R ?>=? : (%,%) -> Boolean if R has ORDSET
--R D : (%, (R -> R), NonNegativeInteger) -> %
--R D : (%, List(Symbol), List(NonNegativeInteger)) -> % if R has PDRING(SYMBOL)
--R D : (%, Symbol, NonNegativeInteger) -> % if R has PDRING(SYMBOL)
--R D : (%, List(Symbol)) -> % if R has PDRING(SYMBOL)
--R D : (%, Symbol) -> % if R has PDRING(SYMBOL)
--R D : (%, List(SingletonAsOrderedSet), List(NonNegativeInteger)) -> %
--R D : (%, SingletonAsOrderedSet, NonNegativeInteger) -> %
--R D : (%, List(SingletonAsOrderedSet)) -> %
--R D : (%, SingletonAsOrderedSet) -> %
--R ?? : (%, NonNegativeInteger) -> %
--R associates? : (%,%) -> Boolean if R has INTDOM
--R binomThmExpt : (%,%, NonNegativeInteger) -> % if R has COMRING
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if $ has CHARNZ and R has PFECAT or R has CHARNZ
--R coefficient : (%, List(SingletonAsOrderedSet), List(NonNegativeInteger)) -> %
--R coefficient : (%, SingletonAsOrderedSet, NonNegativeInteger) -> %
--R coefficient : (%, NonNegativeInteger) -> R
--R coerce : % -> % if R has INTDOM
--R coerce : Fraction(Integer) -> % if R has ALGEBRA(FRAC(INT)) or R has RETRACT(FRAC(INT))
--R coerce : SparseUnivariatePolynomial(R) -> %
--R coerce : % -> SparseUnivariatePolynomial(R)
--R coerce : SingletonAsOrderedSet -> %
--R composite : (Fraction(%),%) -> Union(Fraction(%), "failed") if R has INTDOM
--R composite : (%,%) -> Union(%, "failed") if R has INTDOM
--R conditionP : Matrix(%) -> Union(Vector(%), "failed") if $ has CHARNZ and R has PFECAT
--R content : (%, SingletonAsOrderedSet) -> % if R has GCDDOM
--R content : % -> R if R has GCDDOM
--R convert : % -> InputForm if SingletonAsOrderedSet has KONVERT(INFORM) and R has KONVERT(INFO
--R convert : % -> Pattern(Integer) if SingletonAsOrderedSet has KONVERT(PATTERN(INT)) and R has
--R convert : % -> Pattern(Float) if SingletonAsOrderedSet has KONVERT(PATTERN(FLOAT)) and R has
--R degree : (%, List(SingletonAsOrderedSet)) -> List(NonNegativeInteger)
--R degree : (%, SingletonAsOrderedSet) -> NonNegativeInteger
init : () -> % if R has STEP
lazyPseudoQuotient : (%,%) -> %
leadingMonomial : % -> %
monicModulo : (%,%) -> %
monomials : % -> List(%)
pseudoRemainder : (%,%) -> %
reductum : % -> %
sample : () -> %
?~=? : (%,%) -> Boolean

```

```

--R differentiate : (%,(R -> R),%) -> %
--R differentiate : (%,(R -> R)) -> %
--R differentiate : (%,(R -> R),NonNegativeInteger) -> %
--R differentiate : (%,List(Symbol),List(NonNegativeInteger)) -> % if R has PDRING(SYMBOL)
--R differentiate : (%,Symbol,NonNegativeInteger) -> % if R has PDRING(SYMBOL)
--R differentiate : (%,List(Symbol)) -> % if R has PDRING(SYMBOL)
--R differentiate : (%,Symbol) -> % if R has PDRING(SYMBOL)
--R differentiate : (%,NonNegativeInteger) -> %
--R differentiate : (%,List(SingletonAsOrderedSet),List(NonNegativeInteger)) -> %
--R differentiate : (%,SingletonAsOrderedSet,NonNegativeInteger) -> %
--R differentiate : (%,List(SingletonAsOrderedSet)) -> %
--R differentiate : (%,SingletonAsOrderedSet) -> %
--R discriminant : % -> R if R has COMRING
--R discriminant : (%,SingletonAsOrderedSet) -> % if R has COMRING
--R divide : (%,%) -> Record(quotient: %,remainder: %) if R has FIELD
--R divideExponents : (%,NonNegativeInteger) -> Union(%, "failed")
--R ?.? : (%,Fraction(%)) -> Fraction(%) if R has INTDOM
--R elt : (Fraction(%),R) -> R if R has FIELD
--R elt : (Fraction(%),Fraction(%)) -> Fraction(%) if R has INTDOM
--R euclideanSize : % -> NonNegativeInteger if R has FIELD
--R eval : (%,List(SingletonAsOrderedSet),List(%)) -> %
--R eval : (%,SingletonAsOrderedSet,%) -> %
--R eval : (%,List(SingletonAsOrderedSet),List(R)) -> %
--R eval : (%,SingletonAsOrderedSet,R) -> %
--R eval : (%,List(%),List(%)) -> %
--R eval : (%,List(Equation(%))) -> %
--R expressIdealMember : (List(%),%) -> Union(List(%), "failed") if R has FIELD
--R exquo : (%,%) -> Union(%, "failed") if R has INTDOM
--R exquo : (%,R) -> Union(%, "failed") if R has INTDOM
--R extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %) if R has FIELD
--R extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %), "failed") if R has FIELD
--R extendedResultant : (%,%) -> Record(resultant: R,coef1: %,coef2: %) if R has INTDOM
--R extendedSubResultantGcd : (%,%) -> Record(gcd: %,coef1: %,coef2: %) if R has INTDOM
--R factor : % -> Factored(%) if R has PFECAT
--R factorPolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if R has PF
--R factorSquareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if
--R fmecg : (%,NonNegativeInteger,R,%) -> %
--R gcd : (%,%) -> % if R has GCDDOM
--R gcd : List(%) -> % if R has GCDDOM
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynom
--R halfExtendedResultant1 : (%,%) -> Record(resultant: R,coef1: %) if R has INTDOM
--R halfExtendedResultant2 : (%,%) -> Record(resultant: R,coef2: %) if R has INTDOM
--R halfExtendedSubResultantGcd1 : (%,%) -> Record(gcd: %,coef1: %) if R has INTDOM
--R halfExtendedSubResultantGcd2 : (%,%) -> Record(gcd: %,coef2: %) if R has INTDOM
--R integrate : % -> % if R has ALGEBRA(FRAC(INT))
--R isExpt : % -> Union(Record(var: SingletonAsOrderedSet,exponent: NonNegativeInteger), "failed")
--R isPlus : % -> Union(List(%), "failed")
--R isTimes : % -> Union(List(%), "failed")
--R karatsubaDivide : (%,NonNegativeInteger) -> Record(quotient: %,remainder: %)
--R lastSubResultant : (%,%) -> % if R has INTDOM

```



```

--R lazyPseudoDivide : (%,%) -> Record(coef: R,gap: NonNegativeInteger,quotient: %,remainder: %)
--R lazyPseudoRemainder : (%,%) -> %
--R lazyResidueClass : (%,%) -> Record(polnum: %,polden: R,power: NonNegativeInteger)
--R lcm : (%,%) -> % if R has GCDDOM
--R lcm : List(%) -> % if R has GCDDOM
--R mainVariable : % -> Union(SingletonAsOrderedSet,"failed")
--R makeSUP : % -> SparseUnivariatePolynomial(R)
--R mapExponents : ((NonNegativeInteger -> NonNegativeInteger),%) -> %
--R max : (%,%) -> % if R has ORDSET
--R min : (%,%) -> % if R has ORDSET
--R minimumDegree : (%,List(SingletonAsOrderedSet)) -> List(NonNegativeInteger)
--R minimumDegree : (%,SingletonAsOrderedSet) -> NonNegativeInteger
--R minimumDegree : % -> NonNegativeInteger
--R monicDivide : (%,%) -> Record(quotient: %,remainder: %)
--R monicDivide : (%,%,SingletonAsOrderedSet) -> Record(quotient: %,remainder: %)
--R monomial : (%,List(SingletonAsOrderedSet),List(NonNegativeInteger)) -> %
--R monomial : (%,SingletonAsOrderedSet,NonNegativeInteger) -> %
--R monomial : (R,NonNegativeInteger) -> %
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed") if R has FIELD
--R multiplyExponents : (%,NonNegativeInteger) -> %
--R multivariate : (SparseUnivariatePolynomial(%),SingletonAsOrderedSet) -> %
--R multivariate : (SparseUnivariatePolynomial(R),SingletonAsOrderedSet) -> %
--R nextItem : % -> Union(%,"failed") if R has STEP
--R numberOfMonomials : % -> NonNegativeInteger
--R order : (%,%) -> NonNegativeInteger if R has INTDOM
--R patternMatch : (%,Pattern(Integer),PatternMatchResult(Integer,%)) -> PatternMatchResult(Integer,%)
--R patternMatch : (%,Pattern(Float),PatternMatchResult(Float,%)) -> PatternMatchResult(Float,%)
--R pomopo! : (%,R,NonNegativeInteger,%) -> %
--R prime? : % -> Boolean if R has PFECAT
--R primitiveMonomials : % -> List(%)
--R primitivePart : (%,SingletonAsOrderedSet) -> % if R has GCDDOM
--R primitivePart : % -> % if R has GCDDOM
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %) if R has FIELD
--R pseudoDivide : (%,%) -> Record(coef: R,quotient: %,remainder: %) if R has INTDOM
--R pseudoQuotient : (%,%) -> % if R has INTDOM
--R ?quo? : (%,%) -> % if R has FIELD
--R reducedSystem : Matrix(%) -> Matrix(R)
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(R),vec: Vector(R))
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) if R has INTDOM
--R reducedSystem : Matrix(%) -> Matrix(Integer) if R has LINEXP(INT)
--R ?rem? : (%,%) -> % if R has FIELD
--R resultant : (%,%) -> R if R has COMRING
--R resultant : (%,%,SingletonAsOrderedSet) -> % if R has COMRING
--R retract : % -> SparseUnivariatePolynomial(R)
--R retract : % -> SingletonAsOrderedSet
--R retract : % -> Integer if R has RETRACT(INT)
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(SparseUnivariatePolynomial(R),"failed")
--R retractIfCan : % -> Union(SingletonAsOrderedSet,"failed")
--R retractIfCan : % -> Union(Integer,"failed") if R has RETRACT(INT)

```

```

--R retractIfCan : % -> Union(Fraction(Integer),"failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(R,"failed")
--R separate : (%,% ) -> Record(primePart: %,commonPart: %) if R has GCDDOM
--R shiftLeft : (% ,NonNegativeInteger) -> %
--R shiftRight : (% ,NonNegativeInteger) -> %
--R sizeLess? : (%,% ) -> Boolean if R has FIELD
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(%)),SparseUnivariatePolynomial(%)) -> %
--R squareFree : % -> Factored(%) if R has GCDDOM
--R squareFreePart : % -> % if R has GCDDOM
--R squareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if R has GCDDOM
--R subResultantGcd : (%,% ) -> % if R has INTDOM
--R subResultantsChain : (%,% ) -> List(%) if R has INTDOM
--R subtractIfCan : (%,% ) -> Union(%, "failed")
--R totalDegree : (% ,List(SingletonAsOrderedSet)) -> NonNegativeInteger
--R totalDegree : % -> NonNegativeInteger
--R unit? : % -> Boolean if R has INTDOM
--R unitCanonical : % -> % if R has INTDOM
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %) if R has INTDOM
--R univariate : % -> SparseUnivariatePolynomial(R)
--R univariate : (% ,SingletonAsOrderedSet) -> SparseUnivariatePolynomial(%)
--R unmakeSUP : SparseUnivariatePolynomial(R) -> %
--R variables : % -> List(SingletonAsOrderedSet)
--R vectorise : (% ,NonNegativeInteger) -> Vector(R)
--R
--E 1

```

```

)spool
)lisp (bye)

```

— NewSparseUnivariatePolynomial.help —

```

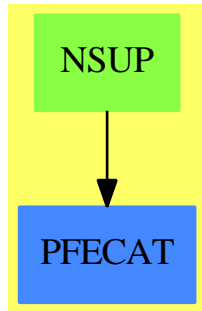
=====
NewSparseUnivariatePolynomial examples
=====

```

A post-facto extension for SUP in order to speed up operations related to pseudo-division and gcd for both SUP and, consequently, NSMP.

See Also:

```
o )show NewSparseUnivariatePolynomial
```

NewSparseUnivariatePolynomial (NSUP)

See

⇒ “NewSparseMultivariatePolynomial” (NSMP) [15](#) on page [1713](#)

Exports:

0	1	associates?
binomThmExpt	characteristic	charthRoot
coefficient	coefficients	coerce
composite	conditionP	content
convert	D	degree
differentiate	discriminant	divide
divideExponents	elt	euclideanSize
eval	expressIdealMember	exquo
extendedEuclidean	extendedResultant	extendedSubResultantGcd
factor	factorPolynomial	factorSquareFreePolynomial
fmeeg	gcd	gcdPolynomial
ground	ground?	halfExtendedResultant1
halfExtendedResultant2	halfExtendedSubResultantGcd1	halfExtendedSubResultantGcd2
hash	init	integrate
isExpt	isPlus	isTimes
karatsubaDivide	lastSubResultant	latex
lazyPseudoDivide	lazyPseudoQuotient	lazyPseudoRemainder
lazyResidueClass	lcm	leadingCoefficient
leadingMonomial	mainVariable	makeSUP
map	mapExponents	max
min	minimumDegree	monicDivide
monicModulo	monomial	monomial?
monomials	multiEuclidean	multiplyExponents
multivariate	nextItem	numberOfMonomials
one?	order	patternMatch
popopo!	prime?	primitiveMonomials
primitivePart	principalIdeal	pseudoDivide
pseudoQuotient	pseudoRemainder	recip
reducedSystem	reductum	resultant
retract	retractIfCan	sample
separate	shiftLeft	shiftRight
sizeLess?	solveLinearPolynomialEquation	squareFree
squareFreePart	squareFreePolynomial	subResultantGcd
subResultantsChain	subtractIfCan	totalDegree
totalDegree	unit?	unitCanonical
unitNormal	univariate	unmakeSUP
variables	vectorise	zero?
?*?	?**?	?+?
?-?	-?	?=?
?^?	?.	?~=?
?/?	?<?	?<=?
?>?	?>=?	?quo?
?rem?		

```

)abbrev domain NSUP NewSparseUnivariatePolynomial
++ Author: Marc Moreno Maza
++ Date Created: 23/07/98
++ Date Last Updated: 14/12/98
++ Description:
++ A post-facto extension for \axiomType{SUP} in order
++ to speed up operations related to pseudo-division and gcd for
++ both \axiomType{SUP} and, consequently, \axiomType{NSMP}.

NewSparseUnivariatePolynomial(R): Exports == Implementation where

R:Ring
NNI ==> NonNegativeInteger
SUPR ==> SparseUnivariatePolynomial R

Exports == Join(UnivariatePolynomialCategory(R),
CoercibleTo(SUPR),RetractableTo(SUPR)) with
fmcg : (%,NNI,R,%) -> %
++ \axiom{fmcg(p1,e,r,p2)} returns \axiom{p1 - r * x**e * p2}
++ where \axiom{x} is \axiom{monomial(1,1)}
monicModulo : ($, $) -> $
++ \axiom{monicModulo(a,b)} returns \axiom{r} such that \axiom{r} is
++ reduced w.r.t. \axiom{b} and \axiom{b} divides \axiom{a - r}
++ where \axiom{b} is monic.
lazyResidueClass: ($,$) -> Record(polnum:$, polden:R, power:NNI)
++ \axiom{lazyResidueClass(a,b)} returns \axiom{[r,c,n]} such that
++ \axiom{r} is reduced w.r.t. \axiom{b} and \axiom{b} divides
++ \axiom{c^n * a - r} where \axiom{c} is \axiom{leadingCoefficient(b)}
++ and \axiom{n} is as small as possible with the previous properties.
lazyPseudoRemainder: ($,$) -> $
++ \axiom{lazyPseudoRemainder(a,b)} returns \axiom{r} if
++ \axiom{lazyResidueClass(a,b)} returns \axiom{[r,c,n]}.
++ This lazy pseudo-remainder is computed by means of the
++ fmcg from NewSparseUnivariatePolynomial operation.
lazyPseudoDivide: ($,$) -> Record(coef:R,gap:NNI,quotient:$,remainder:$)
++ \axiom{lazyPseudoDivide(a,b)} returns \axiom{[c,g,q,r]} such that
++ \axiom{c^n * a = q*b + r} and \axiom{lazyResidueClass(a,b)} returns
++ \axiom{[r,c,n]} where
++ \axiom{n + g = max(0, degree(b) - degree(a) + 1)}.
lazyPseudoQuotient: ($,$) -> $
++ \axiom{lazyPseudoQuotient(a,b)} returns \axiom{q} if
++ \axiom{lazyPseudoDivide(a,b)} returns \axiom{[c,g,q,r]}
if R has IntegralDomain
then
subResultantsChain: ($, $) -> List $
++ \axiom{subResultantsChain(a,b)} returns the list of the non-zero
++ sub-resultants of \axiom{a} and \axiom{b} sorted by increasing
++ degree.
lastSubResultant: ($, $) -> $
++ \axiom{lastSubResultant(a,b)} returns \axiom{resultant(a,b)}

```

```

++ if \axiom{a} and \axiom{b} has no non-trivial gcd
++ in \axiom{R(-1) P}
++ otherwise the non-zero sub-resultant with smallest index.
extendedSubResultantGcd: ($, $) -> Record(gcd: $, coef1: $, coef2: $)
++ \axiom{extendedSubResultantGcd(a,b)} returns \axiom{[g,ca, cb]}
++ such that \axiom{g} is a gcd of \axiom{a} and \axiom{b} in
++ \axiom{R(-1) P} and \axiom{g = ca * a + cb * b}
halfExtendedSubResultantGcd1: ($, $) -> Record(gcd: $, coef1: $)
++ \axiom{halfExtendedSubResultantGcd1(a,b)} returns \axiom{[g,ca]}
++ such that \axiom{extendedSubResultantGcd(a,b)} returns
++ \axiom{[g,ca, cb]}
halfExtendedSubResultantGcd2: ($, $) -> Record(gcd: $, coef2: $)
++ \axiom{halfExtendedSubResultantGcd2(a,b)} returns \axiom{[g,cb]}
++ such that \axiom{extendedSubResultantGcd(a,b)} returns
++ \axiom{[g,ca, cb]}
extendedResultant: ($, $) -> Record(resultant: R, coef1: $, coef2: $)
++ \axiom{extendedResultant(a,b)} returns \axiom{[r,ca,cb]} such that
++ \axiom{r} is the resultant of \axiom{a} and \axiom{b} and
++ \axiom{r = ca * a + cb * b}
halfExtendedResultant1: ($, $) -> Record(resultant: R, coef1: $)
++ \axiom{halfExtendedResultant1(a,b)} returns \axiom{[r,ca]}
++ such that \axiom{extendedResultant(a,b)} returns
++ \axiom{[r,ca, cb]}
halfExtendedResultant2: ($, $) -> Record(resultant: R, coef2: $)
++ \axiom{halfExtendedResultant2(a,b)} returns \axiom{[r,ca]} such
++ that \axiom{extendedResultant(a,b)} returns \axiom{[r,ca, cb]}

```

Implementation == SparseUnivariatePolynomial(R) add

```

Term == Record(k:NonNegativeInteger,c:R)
Rep ==> List Term

rep(s:$):Rep == s pretend Rep
per(l:Rep):$ == l pretend $

coerce (p:$):SUPR ==
  p pretend SUPR

coerce (p:SUPR):$ ==
  p pretend $

retractIfCan (p:$) : Union(SUPR,"failed") ==
  (p pretend SUPR)::Union(SUPR,"failed")

monicModulo(x,y) ==
  zero? y =>
    error "in monicModulo$NSUP: division by 0"
  ground? y =>
    error "in monicModulo$NSUP: ground? #2"
  yy := rep y

```

```

--      not one? (yy.first.c) =>
not ((yy.first.c) = 1) =>
    error "in monicModulo$NSUP: not monic #2"
xx := rep x; empty? xx => x
e := yy.first.k; y := per(yy.rest)
-- while (not empty? xx) repeat
repeat
    if (u:=subtractIfCan(xx.first.k,e)) case "failed" then break
    xx:= rep fmeCG(per rest(xx), u, xx.first.c, y)
    if empty? xx then break
per xx

lazyResidueClass(x,y) ==
zero? y =>
    error "in lazyResidueClass$NSUP: division by 0"
ground? y =>
    error "in lazyResidueClass$NSUP: ground? #2"
yy := rep y; co := yy.first.c; xx: Rep := rep x
empty? xx => [x, co, 0]
pow: NNI := 0; e := yy.first.k; y := per(yy.rest);
repeat
    if (u:=subtractIfCan(xx.first.k,e)) case "failed" then break
    xx:= rep fmeCG(co * per rest(xx), u, xx.first.c, y)
    pow := pow + 1
    if empty? xx then break
[per xx, co, pow]

lazyPseudoRemainder(x,y) ==
zero? y =>
    error "in lazyPseudoRemainder$NSUP: division by 0"
ground? y =>
    error "in lazyPseudoRemainder$NSUP: ground? #2"
ground? x => x
yy := rep y; co := yy.first.c
--      one? co => monicModulo(x,y)
      (co = 1) => monicModulo(x,y)
      (co = -1) => - monicModulo(-x,-y)
xx:= rep x; e := yy.first.k; y := per(yy.rest)
repeat
    if (u:=subtractIfCan(xx.first.k,e)) case "failed" then break
    xx:= rep fmeCG(co * per rest(xx), u, xx.first.c, y)
    if empty? xx then break
per xx

lazyPseudoDivide(x,y) ==
zero? y =>
    error "in lazyPseudoDivide$NSUP: division by 0"
ground? y =>
    error "in lazyPseudoDivide$NSUP: ground? #2"
yy := rep y; e := yy.first.k;

```

```

xx: Rep := rep x; co := yy.first.c
(empty? xx) or (xx.first.k < e) => [co,0,0,x]
pow: NNI := subtractIfCan(xx.first.k,e)::NNI + 1
qq: Rep := []; y := per(yy.rest)
repeat
  if (u:=subtractIfCan(xx.first.k,e)) case "failed" then break
  qq := cons([u::NNI, xx.first.c]$Term, rep (co * per qq))
  xx := rep fmecg(co * per rest(xx), u, xx.first.c, y)
  pow := subtractIfCan(pow,1)::NNI
  if empty? xx then break
[co, pow, per reverse qq, per xx]

lazyPseudoQuotient(x,y) ==
  zero? y =>
    error "in lazyPseudoQuotient$NSUP: division by 0"
  ground? y =>
    error "in lazyPseudoQuotient$NSUP: ground? #2"
  yy := rep y; e := yy.first.k; xx: Rep := rep x
  (empty? xx) or (xx.first.k < e) => 0
  qq: Rep := []; co := yy.first.c; y := per(yy.rest)
  repeat
    if (u:=subtractIfCan(xx.first.k,e)) case "failed" then break
    qq := cons([u::NNI, xx.first.c]$Term, rep (co * per qq))
    xx := rep fmecg(co * per rest(xx), u, xx.first.c, y)
    if empty? xx then break
  per reverse qq

if R has IntegralDomain
then
  pack ==> PseudoRemainderSequence(R, %)

  subResultantGcd(p1,p2) == subResultantGcd(p1,p2)$pack

  subResultantsChain(p1,p2) == chainSubResultants(p1,p2)$pack

  lastSubResultant(p1,p2) == lastSubResultant(p1,p2)$pack

  resultant(p1,p2) == resultant(p1,p2)$pack

  extendedResultant(p1,p2) ==
    re: Record(coef1: $, coef2: $, resultant: R) := resultantEuclidean(p1,p2)$pack
    [re.resultant, re.coef1, re.coef2]

  halfExtendedResultant1(p1:$, p2: $): Record(resultant: R, coef1: $) ==
    re: Record(coef1: $, resultant: R) := semiResultantEuclidean1(p1, p2)$pack
    [re.resultant, re.coef1]

  halfExtendedResultant2(p1:$, p2: $): Record(resultant: R, coef2: $) ==
    re: Record(coef2: $, resultant: R) := semiResultantEuclidean2(p1, p2)$pack
    [re.resultant, re.coef2]

```



```

extendedSubResultantGcd(p1,p2) ==
  re: Record(coef1: $, coef2: $, gcd: $) := subResultantGcdEuclidean(p1,p2)$pack
  [re.gcd, re.coef1, re.coef2]

halfExtendedSubResultantGcd1(p1:$, p2: $): Record(gcd: $, coef1: $) ==
  re: Record(coef1: $, gcd: $) := semiSubResultantGcdEuclidean1(p1, p2)$pack
  [re.gcd, re.coef1]

halfExtendedSubResultantGcd2(p1:$, p2: $): Record(gcd: $, coef2: $) ==
  re: Record(coef2: $, gcd: $) := semiSubResultantGcdEuclidean2(p1, p2)$pack
  [re.gcd, re.coef2]

pseudoDivide(x,y) ==
  zero? y =>
    error "in pseudoDivide$NSUP: division by 0"
  ground? y =>
    error "in pseudoDivide$NSUP: ground? #2"
  yy := rep y; e := yy.first.k
  xx: Rep := rep x; co := yy.first.c
  (empty? xx) or (xx.first.k < e) => [co,0,x]
  pow: NNI := subtractIfCan(xx.first.k,e)::NNI + 1
  qq: Rep := []; y := per(yy.rest)
  repeat
    if (u:=subtractIfCan(xx.first.k,e)) case "failed" then break
    qq := cons([u::NNI, xx.first.c]$Term, rep (co * per qq))
    xx := rep fmech(co * per rest(xx), u, xx.first.c, y)
    pow := subtractIfCan(pow,1)::NNI
    if empty? xx then break
  zero? pow => [co, per reverse qq, per xx]
  default: R := co ** pow
  q := default * (per reverse qq)
  x := default * (per xx)
  [co, q, x]

pseudoQuotient(x,y) ==
  zero? y =>
    error "in pseudoDivide$NSUP: division by 0"
  ground? y =>
    error "in pseudoDivide$NSUP: ground? #2"
  yy := rep y; e := yy.first.k; xx: Rep := rep x
  (empty? xx) or (xx.first.k < e) => 0
  pow: NNI := subtractIfCan(xx.first.k,e)::NNI + 1
  qq: Rep := []; co := yy.first.c; y := per(yy.rest)
  repeat
    if (u:=subtractIfCan(xx.first.k,e)) case "failed" then break
    qq := cons([u::NNI, xx.first.c]$Term, rep (co * per qq))
    xx := rep fmech(co * per rest(xx), u, xx.first.c, y)
    pow := subtractIfCan(pow,1)::NNI
    if empty? xx then break

```

```
zero? pow => per reverse qq
(co ** pow) * (per reverse qq)
```

— NSUP.dotabb —

```
"NSUP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=NSUP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"NSUP" -> "PFECAT"
```

domain NONE None

— None.input —

```
)set break resume
)sys rm -f None.output
)spool None.output
)set message test on
)set message auto off
)clear all
--S 1 of 3
[ ]
--R
--R
--R (1) []
--R
--E 1
```

Type: List(None)

```
--S 2 of 3
[ ] :: List Float
--R
--R
--R (2) []
--R
--E 2
```

Type: List(Float)

```
--S 3 of 3
[ ]$List(NonNegativeInteger)
--R
--R
--R (3) []
```

```
--R                                         Type: List(NonNegativeInteger)
--E 3
)spool
)lisp (bye)
```

— None.help —

```
=====
None examples
=====
```

The None domain is not very useful for interactive work but it is provided nevertheless for completeness of the Axiom type system.

Probably the only place you will ever see it is if you enter an empty list with no type information.

```
[ ]
[]
                                         Type: List None
```

Such an empty list can be converted into an empty list of any other type.

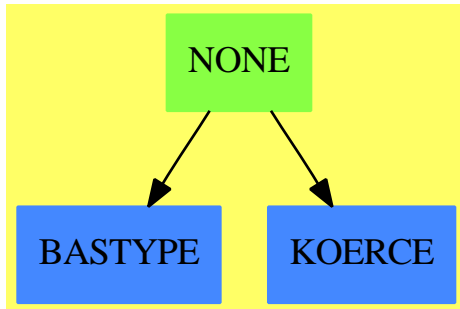
```
[ ] :: List Float
[]
                                         Type: List Float
```

If you wish to produce an empty list of a particular type directly, such as List NonNegativeInteger, do it this way.

```
[ ]$List(NonNegativeInteger)
[]
                                         Type: List NonNegativeInteger
```

See Also:
o)show None

None (NONE)



See

⇒ “Any” (ANY) [2](#) on page [50](#)

Exports:

coerce hash latex $==?$ $\sim=?$

— domain NONE None —

```

)abbrev domain NONE None
++ Author: Mark Botch
++ Description:
++ \spadtype{None} implements a type with no objects. It is mainly
++ used in technical situations where such a thing is needed (e.g.
++ the interpreter and some of the internal \spadtype{Expression} code).

```

```

None():SetCategory == add
  coerce(none:%):OutputForm == "NONE" :: OutputForm
  x:% = y:% == EQ(x,y)$Lisp

```

— NONE.dotabb —

```

"NONE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=NONE"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"NONE" -> "BASTYPE"
"NONE" -> "KOERCE"

```

domain NNI NonNegativeInteger

— NonNegativeInteger.input —

```
)set break resume
)sys rm -f NonNegativeInteger.output
)spool NonNegativeInteger.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NonNegativeInteger
--R NonNegativeInteger is a domain constructor
--R Abbreviation for NonNegativeInteger is NNI
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for NNI
--R
--R----- Operations -----
--R ?? : (%,%) -> %                ?? : (PositiveInteger,%) -> %
--R ??? : (%,PositiveInteger) -> %  ?+? : (%,%) -> %
--R ?<? : (%,%) -> Boolean          ?<=? : (%,%) -> Boolean
--R ?=? : (%,%) -> Boolean          ?>? : (%,%) -> Boolean
--R ?>=? : (%,%) -> Boolean         1 : () -> %
--R 0 : () -> %                    ?^? : (%,PositiveInteger) -> %
--R coerce : % -> OutputForm       gcd : (%,%) -> %
--R hash : % -> SingleInteger       latex : % -> String
--R max : (%,%) -> %                min : (%,%) -> %
--R one? : % -> Boolean              ?quo? : (%,%) -> %
--R random : % -> %                  recip : % -> Union(%, "failed")
--R ?rem? : (%,%) -> %               sample : () -> %
--R shift : (%,Integer) -> %         sup : (%,%) -> %
--R zero? : % -> Boolean             ?~=? : (%,%) -> Boolean
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R ?^? : (%,NonNegativeInteger) -> %
--R divide : (%,%) -> Record(quotient: %,remainder: %)
--R exquo : (%,%) -> Union(%, "failed")
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)
```

— NonNegativeInteger.help —

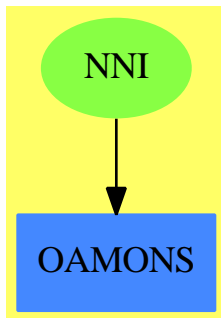
```
=====
NonNegativeInteger examples
=====
```

NonNegativeInteger provides functions for non-negative integers.

See Also:

o)show NonNegativeInteger

NonNegativeInteger (NNI)



See

⇒ “Integer” (INT) [10](#) on page [1361](#)

⇒ “PositiveInteger” (PI) [17](#) on page [2100](#)

⇒ “RomanNumeral” (ROMAN) [19](#) on page [2326](#)

Exports:

0	1	coerce	divide	exquo
gcd	hash	latex	max	min
one?	random	recip	sample	shift
subtractIfCan	sup	zero?	?~=?	?*?
?**?	?^?	?+?	?<?	?<=?
?=?	?>?	?>=?	?quo?	?rem?

— domain NNI NonNegativeInteger —

```
)abbrev domain NNI NonNegativeInteger
```

```
++ Author: Mark Botch
```

```
++ Description:
```

```
++ \spadtype{NonNegativeInteger} provides functions for non-negative integers.
```

```
NonNegativeInteger: Join(OrderedAbelianMonoidSup, Monoid) with
    _quo : (%,% ) -> %
```

```

++ a quo b returns the quotient of \spad{a} and b, forgetting
++ the remainder.
_rem : (%,% ) -> %
++ a rem b returns the remainder of \spad{a} and b.
gcd : (%,% ) -> %
++ gcd(a,b) computes the greatest common divisor of two
++ non negative integers \spad{a} and b.
divide: (%,% ) -> Record(quotient:%,remainder:%)
++ divide(a,b) returns a record containing both
++ remainder and quotient.
_exquo: (%,% ) -> Union(%, "failed")
++ exquo(a,b) returns the quotient of \spad{a} and b, or "failed"
++ if b is zero or \spad{a} rem b is zero.
shift: (% , Integer) -> %
++ shift(a,i) shift \spad{a} by i bits.
random : % -> %
++ random(n) returns a random integer from 0 to \spad{n-1}.
commutative("*")
++ commutative("*") means multiplication is commutative,
++ that is, \spad{x*y = y*x}.

== SubDomain(Integer,#1 >= 0) add
x,y:%
sup(x,y) == MAX(x,y)$Lisp
shift(x:%, n:Integer):% == ASH(x,n)$Lisp
subtractIfCan(x, y) ==
  c:Integer := (x pretend Integer) - (y pretend Integer)
  c < 0 => "failed"
  c pretend %

-----

-- NNI.dotabb --

"NNI" [color="#88FF44",href="bookvol10.3.pdf#nameddest=NNI",shape=ellipse]
"OAMONS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAMONS"]
"NNI" -> "OAMONS"

-----

```

domain NOTTING NottinghamGroup

— NottinghamGroup.input —

)set break resume

```
)sys rm -f NottinghamGroup.output
)spool NottinghamGroup.output
)set message test on
)set message auto off
)clear all
```

```
--S 1 of 8
```

```
x:=monomial(1,1)$UFPS PF 1783
```

```
--R
```

```
--R
```

```
--R (1) x
```

```
--R                                         Type: UnivariateFormalPowerSeries(PrimeField(1783))
```

```
--E 1
```

```
--S 2 of 8
```

```
s:=retract(sin x)$NOTTING PF 1783
```

```
--R
```

```
--R
```

```
--R          3      5      7      9      11
--R (2) x + 297x + 1679x + 427x + 316x + 0(x )
```

```
--R                                         Type: NottinghamGroup(PrimeField(1783))
```

```
--E 2
```

```
--S 3 of 8
```

```
s^2
```

```
--R
```

```
--R
```

```
--R          3      5      7      9      11
--R (3) x + 594x + 535x + 1166x + 1379x + 0(x )
```

```
--R                                         Type: NottinghamGroup(PrimeField(1783))
```

```
--E 3
```

```
--S 4 of 8
```

```
s^-1
```

```
--R
```

```
--R
```

```
--R          3      5      7      9      11
--R (4) x + 1486x + 847x + 207x + 1701x + 0(x )
```

```
--R                                         Type: NottinghamGroup(PrimeField(1783))
```

```
--E 4
```

```
--S 5 of 8
```

```
s^-1*s
```

```
--R
```

```
--R
```

```
--R          11
--R (5) x + 0(x )
```

```
--R                                         Type: NottinghamGroup(PrimeField(1783))
```

```
--E 5
```



```

--S 6 of 8
s*s^-1
--R
--R
--R      11
--R (6)  x + 0(x )
--R
--R                                          Type: NottinghamGroup(PrimeField(1783))
--E 6

--S 7 of 8
sample()$NOTTING(PF(1783))
--R
--R (7)  x
--R
--R                                          Type: NottinghamGroup(PrimeField(1783))
--E 7

--S 8 of 8
)show NottinghamGroup
--R
--R NottinghamGroup(F: FiniteFieldCategory) is a domain constructor
--R Abbreviation for NottinghamGroup is NOTTING
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for NOTTING
--R
--R----- Operations -----
--R ???: (% ,%) -> %                ??? : (% ,Integer) -> %
--R ??? : (% ,PositiveInteger) -> %  ?/? : (% ,%) -> %
--R ?? : (% ,%) -> Boolean          1 : () -> %
--R ?? : (% ,Integer) -> %          ?^? : (% ,PositiveInteger) -> %
--R coerce : % -> OutputForm        commutator : (% ,%) -> %
--R conjugate : (% ,%) -> %          hash : % -> SingleInteger
--R inv : % -> %                     latex : % -> String
--R one? : % -> Boolean              recip : % -> Union(% ,"failed")
--R sample : () -> %                 ?~? : (% ,%) -> Boolean
--R ??? : (% ,NonNegativeInteger) -> %
--R ?^? : (% ,NonNegativeInteger) -> %
--R retract : UnivariateFormalPowerSeries(F) -> %
--R
--E 8

)spool
)lisp (bye)

```

— NottinghamGroup.help —

=====

NottinghamGroup examples

=====

n

If F is a finite field with p elements, then we may form the group of all formal power series $\{t(1+a_1t+a_2t^2+\dots)\}$ where $u(0)=0$ and $u(0)=1$ and a_i is an element of F .

i

The group operation is substitution (composition).
This is called the Nottingham Group.

The Nottingham Group is the projective limit of finite p -groups.
Every finite p -group can be embedded in the Nottingham Group.

$x := \text{monomial}(1,1)$ \$UFPS PF 1783
x

$s := \text{retract}(\sin x)$ \$NOTTING PF 1783

$$x^3 + 297x^5 + 1679x^7 + 427x^9 + 316x^{11} + O(x^{12})$$

s^2

$$x^3 + 594x^5 + 535x^7 + 1166x^9 + 1379x^{11} + O(x^{12})$$

s^{-1}

$$x^3 + 1486x^5 + 847x^7 + 207x^9 + 1701x^{11} + O(x^{12})$$

$s^{-1}s$

$$x^{11} + O(x^{12})$$

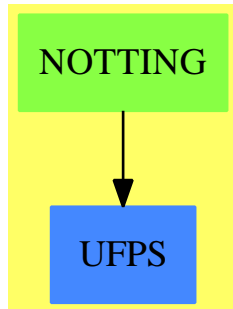
$s*s^{-1}$

$$x^{11} + O(x^{12})$$

See Also:

o)show NottinghamGroup
o)show UnivariateFormalPowerSeries

NottinghamGroup (NOTTING)



Exports:

1	coerce	commutator	conjugate	hash
inv	latex	one?	recip	sample
$\wedge =$	retract	*	**	/
=	\wedge			

— domain NOTTING NottinghamGroup —

```
)abbrev domain NOTTING NottinghamGroup
++ Author: Mark Botch
++ Description:
++ This is an implmenetation of the Nottingham Group

NottinghamGroup(F:FiniteFieldCategory): Group with
  retract: UnivariateFormalPowerSeries F -> %
  == add
  Rep:=UnivariateFormalPowerSeries F

  coerce f == coerce(f::Rep)$UnivariateFormalPowerSeries(F)

  retract f ==
    if zero? coefficient(f,0) and one? coefficient(f,1)
    then f::Rep
    else error"The leading term must be x"

  1 == monomial(1,1)

  f*g == f.g

  inv f == revert f
```

— NOTTING.dotabb —

— NumericalIntegrationProblem.input —

NumericalIntegrationProblem examples

=====

NumericalIntegrationProblem is a domain for the representation of
Numerical Integration problems for use by ANNA.

The representation is a Union of two record types - one for integration of
a function of one variable:

```
Record(var:Symbol,
       fn:Expression DoubleFloat,
       range:Segment OrderedCompletion DoubleFloat,
       abserr:DoubleFloat,
       relerr:DoubleFloat)
```

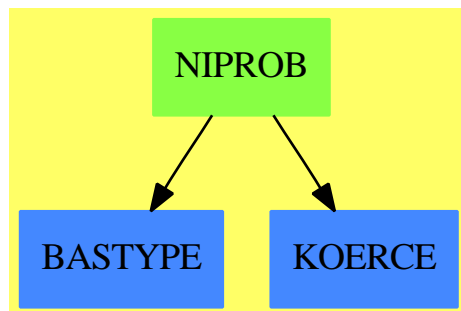
and one for multivariate integration:

```
Record(fn:Expression DoubleFloat,
       range:List Segment OrderedCompletion DoubleFloat,
       abserr:DoubleFloat,
       relerr:DoubleFloat)
```

See Also:

```
o )show NumericalIntegrationProblem
```

NumericalIntegrationProblem (NIPROB)



Exports:

```
coerce hash latex retract ?=? ?~=?
```

— domain NIPROB NumericalIntegrationProblem —

```
)abbrev domain NIPROB NumericalIntegrationProblem
```

```

++ Author: Brian Dupee
++ Date Created: December 1997
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{NumericalIntegrationProblem} is a \axiom{domain}
++ for the representation of Numerical Integration problems for use
++ by ANNA.
++
++ The representation is a Union of two record types - one for integration of
++ a function of one variable:
++
++ \axiomType{Record}(var:\axiomType{Symbol},\br
++ fn:\axiomType{Expression DoubleFloat},\br
++ range:\axiomType{Segment OrderedCompletion DoubleFloat},\br
++ abserr:\axiomType{DoubleFloat},\br
++ relerr:\axiomType{DoubleFloat},)
++
++ and one for multivariate integration:
++
++ \axiomType{Record}(fn:\axiomType{Expression DoubleFloat},\br
++ range:\axiomType{List Segment OrderedCompletion DoubleFloat},\br
++ abserr:\axiomType{DoubleFloat},\br
++ relerr:\axiomType{DoubleFloat},).
++
NumericalIntegrationProblem(): EE == II where

EDFA ==> Expression DoubleFloat
SOC DFA ==> Segment OrderedCompletion DoubleFloat
DFA ==> DoubleFloat
NIAA ==> Record(var:Symbol,fn:EDFA,range:SOC DFA,abserr:DFA,relerr:DFA)
MDNIAA ==> Record(fn:EDFA,range:List SOC DFA,abserr:DFA,relerr:DFA)

EE ==> SetCategory with
  coerce: NIAA -> %
    ++ coerce(x) is not documented
  coerce: MDNIAA -> %
    ++ coerce(x) is not documented
  coerce: Union(nia:NIAA,mdnia:MDNIAA) -> %
    ++ coerce(x) is not documented
  coerce: % -> OutputForm
    ++ coerce(x) is not documented
  retract: % -> Union(nia:NIAA,mdnia:MDNIAA)
    ++ retract(x) is not documented

II ==> add
  Rep := Union(nia:NIAA,mdnia:MDNIAA)

  coerce(s:NIAA) == [s]
  coerce(s:MDNIAA) == [s]

```

```

coerce(s:Union(nia:NIAA,mdnia:MDNIAA)) == s
coerce(x:%):OutputForm ==
  (x) case nia => (x.nia)::OutputForm
  (x.mdnia)::OutputForm
retract(x:%):Union(nia:NIAA,mdnia:MDNIAA) ==
  (x) case nia => [x.nia]
  [x.mdnia]

```

— NIPROB.dotabb —

```

"NIPROB" [color="#88FF44",href="bookvol10.3.pdf#nameddest=NIPROB"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"NIPROB" -> "BASTYPE"
"NIPROB" -> "KOERCE"

```

domain ODEPROB NumericalODEProblem

— NumericalODEProblem.input —

```

)set break resume
)sys rm -f NumericalODEProblem.output
)spool NumericalODEProblem.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NumericalODEProblem
--R
--R NumericalODEProblem is a domain constructor
--R Abbreviation for NumericalODEProblem is ODEPROB
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ODEPROB
--R
--R----- Operations -----
--R ?? : (% ,%) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger      latex : % -> String
--R ?~=? : (% ,%) -> Boolean
--R coerce : Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(DoubleFloat)),yin

```

```
--R retract : % -> Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(DoubleFloat)),yinit: L.
--R
--E 1
```

```
)spool
)lisp (bye)
```

— NumericalODEProblem.help —

```
=====
NumericalODEProblem examples
=====
```

NumericalODEProblem is a domain for the representation of Numerical ODE problems for use by ANNA.

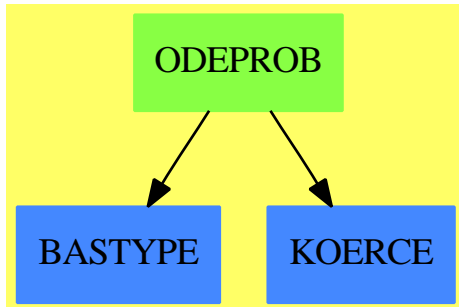
The representation is of type:

```
Record(xinit:DoubleFloat,
      xend:DoubleFloat,
      fn:Vector Expression DoubleFloat,
      yinit:List DoubleFloat,
      intvals:List DoubleFloat,
      g:Expression DoubleFloat,
      abserr:DoubleFloat,
      relerr:DoubleFloat)
```

See Also:

```
o )show NumericalODEProblem
```

NumericalODEProblem (ODEPROB)



Exports:

coerce hash latex retract ==? ?~=?

— domain ODEPROB NumericalODEProblem —

```

)abbrev domain ODEPROB NumericalODEProblem
++ Author: Brian Dupee
++ Date Created: December 1997
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{NumericalODEProblem} is a \axiom{domain}
++ for the representation of Numerical ODE problems for use
++ by ANNA.
++
++ The representation is of type:
++
++ \axiomType{Record}(xinit:\axiomType{DoubleFloat},\br
++ xend:\axiomType{DoubleFloat},\br
++ fn:\axiomType{Vector Expression DoubleFloat},\br
++ yinit:\axiomType{List DoubleFloat},intvals:\axiomType{List DoubleFloat},\br
++ g:\axiomType{Expression DoubleFloat},abserr:\axiomType{DoubleFloat},\br
++ relerr:\axiomType{DoubleFloat})
++
NumericalODEProblem(): EE == II where

DFB ==> DoubleFloat
VEDFB ==> Vector Expression DoubleFloat
LDFB ==> List DoubleFloat
EDFB ==> Expression DoubleFloat
ODEAB ==> Record(xinit:DFB,xend:DFB,fn:VEDFB,yinit:LDFB,intvals:LDFB,g:EDFB,abserr:DFB,relerr:D

EE ==> SetCategory with
  coerce: ODEAB -> %
    ++ coerce(x) is not documented
  
```

```

coerce: % -> OutputForm
++ coerce(x) is not documented
retract: % -> ODEAB
++ retract(x) is not documented

II ==> add
Rep := ODEAB

coerce(s:ODEAB) == s
coerce(x:%):OutputForm ==
  (retract(x)):OutputForm
retract(x:%):ODEAB == x :: Rep

-----

— ODEPROB.dotabb —

"ODEPROB" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ODEPROB"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"ODEPROB" -> "BASTYPE"
"ODEPROB" -> "KOERCE"

-----

```

domain OPTPROB NumericalOptimizationProblem

```

— NumericalOptimizationProblem.input —

)set break resume
)sys rm -f NumericalOptimizationProblem.output
)spool NumericalOptimizationProblem.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NumericalOptimizationProblem
--R
--R NumericalOptimizationProblem is a domain constructor
--R Abbreviation for NumericalOptimizationProblem is OPTPROB
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for OPTPROB
--R

```

```

--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger        latex : % -> String
--R ?~=? : (%,% ) -> Boolean
--R coerce : Union(noa: Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: List(Ord
--R coerce : Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat)) -> %
--R coerce : Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: List(OrderedCompleti
--R retract : % -> Union(noa: Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: Lis
--R
--E 1

```

```

)spool
)lisp (bye)

```

— NumericalOptimizationProblem.help —

```

=====
NumericalOptimizationProblem examples
=====

```

NumericalOptimizationProblem is a domain for the representation of
Numerical Optimization problems for use by ANNA.

The representation is a Union of two record types - one for optimization of
a single function of one or more variables:

```

Record(fn:Expression DoubleFloat,
       init:List DoubleFloat,
       lb:List OrderedCompletion DoubleFloat,
       cf:List Expression DoubleFloat,
       ub:List OrderedCompletion DoubleFloat)

```

and one for least-squares problems i.e. optimization of a set of
observations of a data set:

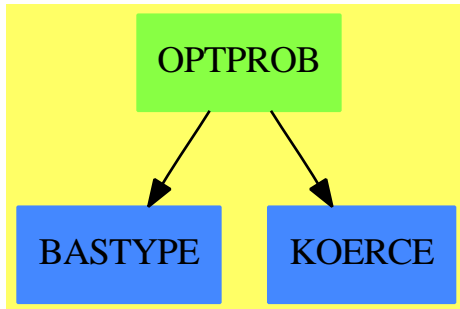
```

Record(lfn:List Expression DoubleFloat,
       init:List DoubleFloat).

```

See Also:
o)show NumericalOptimizationProblem

NumericalOptimizationProblem (OPTPROB)

**Exports:**

coerce hash latex retract ?? ?~=?

— domain OPTPROB NumericalOptimizationProblem —

```

)abbrev domain OPTPROB NumericalOptimizationProblem
++ Author: Brian Dupee
++ Date Created: December 1997
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{NumericalOptimizationProblem} is a \axiom{domain}
++ for the representation of Numerical Optimization problems for use
++ by ANNA.
++
++ The representation is a Union of two record types - one for otimization of
++ a single function of one or more variables:
++
++ \axiomType{Record}(\br
++ fn:\axiomType{Expression DoubleFloat},\br
++ init:\axiomType{List DoubleFloat},\br
++ lb:\axiomType{List OrderedCompletion DoubleFloat},\br
++ cf:\axiomType{List Expression DoubleFloat},\br
++ ub:\axiomType{List OrderedCompletion DoubleFloat})
++
++ and one for least-squares problems i.e. optimization of a set of
++ observations of a data set:
++
++ \axiomType{Record}(lfn:\axiomType{List Expression DoubleFloat},\br
++ init:\axiomType{List DoubleFloat}).
  
```

NumericalOptimizationProblem(): EE == II where

```

LDFD    ==> List DoubleFloat
LEDFD   ==> List Expression DoubleFloat
LSAD    ==> Record(lfn:LEDFD, init:LDFD)
  
```

```

UNOALSAD ==> Union(noa:NOAD,lsa:LSAD)
EDFD      ==> Expression DoubleFloat
LOCDFD    ==> List OrderedCompletion DoubleFloat
NOAD       ==> Record(fn:EDFD, init:LDFD, lb:LOCDFD, cf:LEDFD, ub:LOCDFD)

```

```

EE ==> SetCategory with
  coerce: NOAD -> %
    ++ coerce(x) is not documented
  coerce: LSAD -> %
    ++ coerce(x) is not documented
  coerce: UNOALSAD -> %
    ++ coerce(x) is not documented
  coerce: % -> OutputForm
    ++ coerce(x) is not documented
  retract: % -> UNOALSAD
    ++ retract(x) is not documented

```

```

II ==> add
  Rep := UNOALSAD

  coerce(s:NOAD) == [s]
  coerce(s:LSAD) == [s]
  coerce(x:UNOALSAD) == x
  coerce(x:%):OutputForm ==
    (x) case noa => (x.noa)::OutputForm
    (x.lsa)::OutputForm
  retract(x:%):UNOALSAD ==
    (x) case noa => [x.noa]
    [x.lsa]

```

— OPTPROB.dotabb —

```

"OPTPROB" [color="#88FF44",href="bookvol10.3.pdf#nameddest=OPTPROB"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"OPTPROB" -> "BASTYPE"
"OPTPROB" -> "KOERCE"

```

domain PDEPROB NumericalPDEProblem

— NumericalPDEProblem.input —

```

)set break resume
)sys rm -f NumericalPDEProblem.output
)spool NumericalPDEProblem.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NumericalPDEProblem
--R
--R NumericalPDEProblem is a domain constructor
--R Abbreviation for NumericalPDEProblem is PDEPROB
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PDEPROB
--R
--R----- Operations -----
--R ?? : (% ,%) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger      latex : % -> String
--R ?~=? : (% ,%) -> Boolean
--R coerce : Record(pde: List(Expression(DoubleFloat)),constraints: List(Record(start: DoubleFloat,finish: DoubleFloat)))
--R retract : % -> Record(pde: List(Expression(DoubleFloat)),constraints: List(Record(start: DoubleFloat,finish: DoubleFloat)))
--R
--E 1

)spool
)lisp (bye)

```

— NumericalPDEProblem.help —

=====

NumericalPDEProblem examples

=====

NumericalPDEProblem is a domain for the representation of Numerical PDE problems for use by ANNA.

The representation is of type:

```

Record(pde:List Expression DoubleFloat,
      constraints:List PDEC,
      f:List List Expression DoubleFloat,
      st:String,
      tol:DoubleFloat)

```

where PDEC is of type:

```

Record(start:DoubleFloat,

```

```

finish:DoubleFloat,
grid:NonNegativeInteger,
boundaryType:Integer,
dStart:Matrix DoubleFloat,
dFinish:Matrix DoubleFloat)

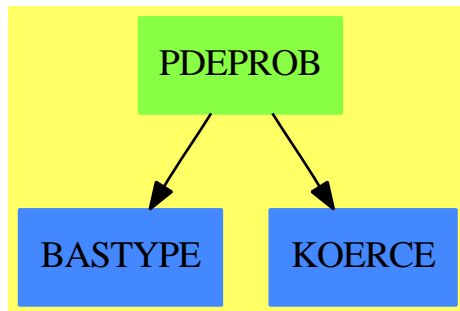
```

See Also:

o)show NumericalPDEProblem

—————→

NumericalPDEProblem (PDEPROB)



Exports:

coerce hash latex retract ?=? ?~=?

— domain PDEPROB NumericalPDEProblem —

```

)abbrev domain PDEPROB NumericalPDEProblem
++ Author: Brian Dupee
++ Date Created: December 1997
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{NumericalPDEProblem} is a \axiom{domain}
++ for the representation of Numerical PDE problems for use
++ by ANNA.
++
++ The representation is of type:
++
++ \axiomType{Record}(pde:\axiomType{List Expression DoubleFloat}, \br
++ constraints:\axiomType{List PDEC}, \br
++ f:\axiomType{List List Expression DoubleFloat},\br
++ st:\axiomType{String},\br
++ tol:\axiomType{DoubleFloat})

```

```

++
++ where \axiomType{PDEC} is of type:
++
++ \axiomType{Record}(start:\axiomType{DoubleFloat}, \br
++ finish:\axiomType{DoubleFloat},\br
++ grid:\axiomType{NonNegativeInteger},\br
++ boundaryType:\axiomType{Integer},\br
++ dStart:\axiomType{Matrix DoubleFloat}, \br
++ dFinish:\axiomType{Matrix DoubleFloat})

NumericalPDEProblem(): EE == II where

DFC ==> DoubleFloat
NNIC ==> NonNegativeInteger
INTC ==> Integer
MDFC ==> Matrix DoubleFloat
PDECC ==> Record(start:DFC, finish:DFC, grid:NNIC, boundaryType:INTC,
                 dStart:MDFC, dFinish:MDFC)
LEDFC ==> List Expression DoubleFloat
PDEBC ==> Record(pde:LEDFC, constraints:List PDECC, f:List LEDFC,
                 st:String, tol:DFC)

EE ==> SetCategory with
  coerce: PDEBC -> %
    ++ coerce(x) is not documented
  coerce: % -> OutputForm
    ++ coerce(x) is not documented
  retract: % -> PDEBC
    ++ retract(x) is not documented

II ==> add
  Rep := PDEBC

  coerce(s:PDEBC) == s
  coerce(x:%):OutputForm ==
    (retract(x)):OutputForm
  retract(x:%):PDEBC == x :: Rep

```

— PDEPROB.dotabb —

```

"PDEPROB" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PDEPROB"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"PDEPROB" -> "BASTYPE"
"PDEPROB" -> "KOERCE"

```

Chapter 16

Chapter O

domain OCT Octonion

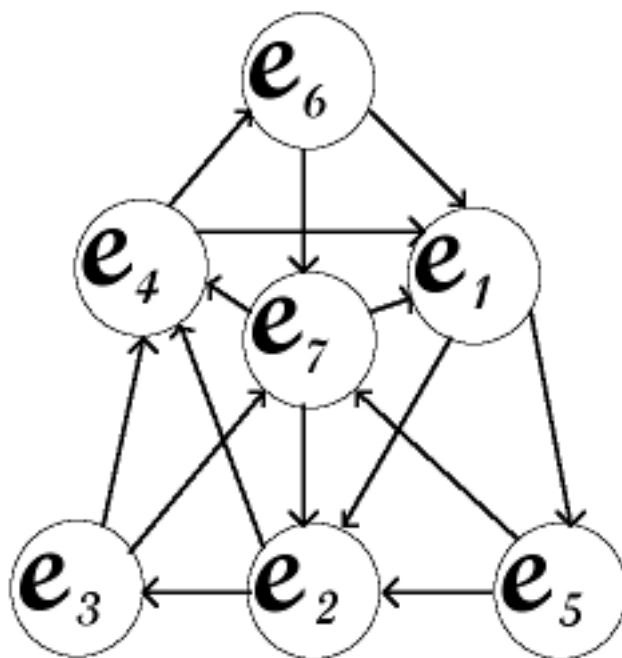
The octonions have the following multiplication table:

1	i	j	k	E	I	J	K
i	-1	k	$-j$	I	$-E$	$-K$	J
j	$-k$	-1	i	J	K	$-E$	$-I$
k	j	$-i$	-1	K	$-J$	I	$-E$
E	$-I$	$-J$	$-K$	-1	i	j	k
I	E	$-K$	J	$-i$	-1	$-k$	j
J	K	E	$-I$	$-j$	k	-1	$-i$
K	$-J$	I	E	$-k$	$-j$	i	-1

There are 3 different kinds of associativity. An algebra is

- **power-associative** if the subalgebra generated by any one element is associative. That is, given any element e then $e * (e * e) = (e * e) * e$.
- **alternative** if the subalgebra generated by any two elements is associative. That is, given any two elements, a and b then $a * (a * b) = (a * a) * b$ and $a * (b * a) = (a * b) * a$ and $b * (a * a) = (b * a) * a$.
- **associative** if the subalgebra generated by any three elements is associative. That is, given any three elements a , b , and c then $a * (b * c) = (a * b) * c$.

The Octonions are power-associative and alternative but not associative, since $I * (J * K) \neq (I * J) * K$.



— Octonion.input —

```

)set break resume
)sys rm -f Octonion.output
)spool Octonion.output
)set message test on
)set message auto off
)clear all
--S 1 of 15
oci1 := octon(1,2,3,4,5,6,7,8)
--R
--R
--R (1) 1 + 2i + 3j + 4k + 5E + 6I + 7J + 8K
--R
--R                                          Type: Octonion(Integer)
--E 1

--S 2 of 15
oci2 := octon(7,2,3,-4,5,6,-7,0)
--R
--R
--R (2) 7 + 2i + 3j - 4k + 5E + 6I - 7J
--R
--R                                          Type: Octonion(Integer)
--E 2

--S 3 of 15
oci3 := octon(quatern(-7,-12,3,-10), quatern(5,6,9,0))

```

```

--R
--R
--R (3)  - 7 - 12i + 3j - 10k + 5E + 6I + 9J
--R                                          Type: Octonion(Integer)
--E 3

--S 4 of 15
(oci1 * oci2) * oci3 - oci1 * (oci2 * oci3)
--R
--R
--R (4)  2696i - 2928j - 4072k + 16E - 1192I + 832J + 2616K
--R                                          Type: Octonion(Integer)
--E 4

--S 5 of 15
[real oci1, imagi oci1, imagj oci1, imagk oci1, _
 imagE oci1, imagI oci1, imagJ oci1, imagK oci1]
--R
--R
--R (5)  [1,2,3,4,5,6,7,8]
--R                                          Type: List(PositiveInteger)
--E 5

--S 6 of 15
q : Quaternion Polynomial Integer := quatern(q1, qi, qj, qk)
--R
--R
--R (6)  q1 + qi i + qj j + qk k
--R                                          Type: Quaternion(Polynomial(Integer))
--E 6

--S 7 of 15
E : Octonion Polynomial Integer:= octon(0,0,0,0,1,0,0,0)
--R
--R
--R (7)  E
--R                                          Type: Octonion(Polynomial(Integer))
--E 7

--S 8 of 15
q * E
--R
--R
--R (8)  q1 E + qi I + qj J + qk K
--R                                          Type: Octonion(Polynomial(Integer))
--E 8

--S 9 of 15
E * q
--R

```

```

--R
--R (9)  $q_1 E - q_i I - q_j J - q_k K$ 
--R
--R                                         Type: Octonion(Polynomial(Integer))
--E 9

--S 10 of 15
q * 1$(Octonion Polynomial Integer)
--R
--R
--R (10)  $q_1 + q_i i + q_j j + q_k k$ 
--R
--R                                         Type: Octonion(Polynomial(Integer))
--E 10

--S 11 of 15
1$(Octonion Polynomial Integer) * q
--R
--R
--R (11)  $q_1 + q_i i + q_j j + q_k k$ 
--R
--R                                         Type: Octonion(Polynomial(Integer))
--E 11

--S 12 of 15
o : Octonion Polynomial Integer := octon(o1, oi, oj, ok, oE, oI, oJ, oK)
--R
--R
--R (12)  $o_1 + o_i i + o_j j + o_k k + o_E E + o_I I + o_J J + o_K K$ 
--R
--R                                         Type: Octonion(Polynomial(Integer))
--E 12

--S 13 of 15
norm o
--R
--R
--R (13)  $ok^2 + oj^2 + oi^2 + oK^2 + oJ^2 + oI^2 + oE^2 + o1^2$ 
--R
--R                                         Type: Polynomial(Integer)
--E 13

--S 14 of 15
p : Octonion Polynomial Integer := octon(p1, pi, pj, pk, pE, pI, pJ, pK)
--R
--R
--R (14)  $p_1 + p_i i + p_j j + p_k k + p_E E + p_I I + p_J J + p_K K$ 
--R
--R                                         Type: Octonion(Polynomial(Integer))
--E 14

--S 15 of 15
norm(o*p)-norm(p)*norm(o)
--R
--R

```

```

--R (15) 0
--R                                         Type: Polynomial(Integer)
--E 15
)spool
)lisp (bye)

```

— Octonion.help —

=====

Octonion examples

=====

The Octonions, also called the Cayley-Dixon algebra, defined over a commutative ring are an eight-dimensional non-associative algebra. Their construction from quaternions is similar to the construction of quaternions from complex numbers.

As Octonion creates an eight-dimensional algebra, you have to give eight components to construct an octonion.

```

oci1 := octon(1,2,3,4,5,6,7,8)
      1 + 2i + 3j + 4k + 5E + 6I + 7J + 8K
                        Type: Octonion Integer

```

```

oci2 := octon(7,2,3,-4,5,6,-7,0)
      7 + 2i + 3j - 4k + 5E + 6I - 7J
                        Type: Octonion Integer

```

Or you can use two quaternions to create an octonion.

```

oci3 := octon(quatern(-7,-12,3,-10), quatern(5,6,9,0))
      - 7 - 12i + 3j - 10k + 5E + 6I + 9J
                        Type: Octonion Integer

```

You can easily demonstrate the non-associativity of multiplication.

```

(oci1 * oci2) * oci3 - oci1 * (oci2 * oci3)
      2696i - 2928j - 4072k + 16E - 1192I + 832J + 2616K
                        Type: Octonion Integer

```

As with the quaternions, we have a real part, the imaginary parts i, j, k, and four additional imaginary parts E, I, J and K. These parts correspond to the canonical basis (1,i,j,k,E,I,J,K).

For each basis element there is a component operation to extract the coefficient of the basis element for a given octonion.

```
[real oci1, imagi oci1, imagj oci1, imagk oci1, _
  imagE oci1, imagI oci1, imagJ oci1, imagK oci1]
[1,2,3,4,5,6,7,8]
```

Type: List PositiveInteger

A basis with respect to the quaternions is given by (1,E). However, you might ask, what then are the commuting rules? To answer this, we create some generic elements.

We do this in Axiom by simply changing the ground ring from Integer to Polynomial Integer.

```
q : Quaternion Polynomial Integer := quatern(q1, qi, qj, qk)
q1 + qi i + qj j + qk k
```

Type: Quaternion Polynomial Integer

```
E : Octonion Polynomial Integer := octon(0,0,0,0,1,0,0,0)
E
```

Type: Octonion Polynomial Integer

Note that quaternions are automatically converted to octonions in the obvious way.

```
q * E
q1 E + qi I + qj J + qk K
```

Type: Octonion Polynomial Integer

```
E * q
q1 E - qi I - qj J - qk K
```

Type: Octonion Polynomial Integer

```
q * 1$(Octonion Polynomial Integer)
q1 + qi i + qj j + qk k
```

Type: Octonion Polynomial Integer

```
1$(Octonion Polynomial Integer) * q
q1 + qi i + qj j + qk k
```

Type: Octonion Polynomial Integer

Finally, we check that the norm, defined as the sum of the squares of the coefficients, is a multiplicative map.

```
o : Octonion Polynomial Integer := octon(o1, oi, oj, ok, oE, oI, oJ, oK)
o1 + oi i + oj j + ok k + oE E + oI I + oJ J + oK K
```

Type: Octonion Polynomial Integer

```
norm o
      2      2      2      2      2      2      2      2
ok  + oj  + oi  + oK  + oJ  + oI  + oE  + o1
```

Type: Polynomial Integer

```
p : Octonion Polynomial Integer := octon(p1, pi, pj, pk, pE, pI, pJ, pK)
p1 + pi i + pj j + pk k + pE E + pI I + pJ J + pK K
Type: Octonion Polynomial Integer
```

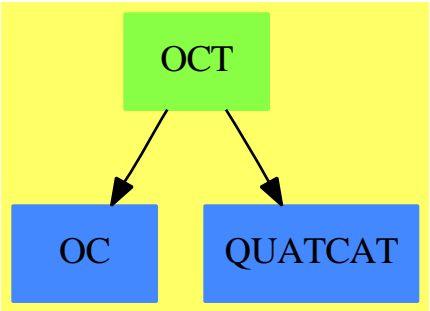
Since the result is 0, the norm is multiplicative.

```
norm(o*p)-norm(p)*norm(o)
0
Type: Polynomial Integer
```

See Also:
o)help Quaternion
o)show Octonion



Octonion (OCT)



Exports:

0	1	abs	characteristic	charthRoot
coerce	conjugate	convert	eval	hash
imagE	imagI	imagJ	imagK	imagi
imagj	imagk	index	inv	latex
lookup	map	max	min	norm
octon	one?	random	rational	rational?
rationalIfCan	real	recip	retract	retractIfCan
sample	size	subtractIfCan	zero?	?*?
?**?	?+?	?-?	-?	?=?
?^?	?~=?	?<?	?<=?	?>?
?>=?	?..?			


```

)abbrev domain OCT Octonion
++ Author: R. Wisbauer, J. Grabmeier
++ Date Created: 05 September 1990
++ Date Last Updated: 20 September 1990
++ References: e.g. I.L Kantor, A.S. Solodovnikov:
++   Hypercomplex Numbers, Springer Verlag Heidelberg, 1989,
++   ISBN 0-387-96980-2
++ Description:
++ Octonion implements octonions (Cayley-Dixon algebra) over a
++ commutative ring, an eight-dimensional non-associative
++ algebra, doubling the quaternions in the same way as doubling
++ the complex numbers to get the quaternions
++ the main constructor function is octon which takes 8
++ arguments: the real part, the i imaginary part, the j
++ imaginary part, the k imaginary part, (as with quaternions)
++ and in addition the imaginary parts E, I, J, K.

--boot $noSubsumption := true
Octonion(R:CommutativeRing): export == impl where

  QR ==> Quaternion R

  export ==> Join(OctonionCategory R, FullyRetractableTo QR) with
    octon: (QR,QR) -> %
      ++ octon(qe,qE) constructs an octonion from two quaternions
      ++ using the relation 0 = Q + QE.
  impl ==> add
    Rep := Record(e: QR,E: QR)

    0 == [0,0]
    1 == [1,0]

    a,b,c,d,f,g,h,i : R
    p,q : QR
    x,y : %

    real x == real (x.e)
    imagi x == imagI (x.e)
    imagj x == imagJ (x.e)
    imagk x == imagK (x.e)
    imagE x == real (x.E)
    imagI x == imagI (x.E)
    imagJ x == imagJ (x.E)
    imagK x == imagK (x.E)
    octon(a,b,c,d,f,g,h,i) == [quatern(a,b,c,d)$QR,quatern(f,g,h,i)$QR]
    octon(p,q) == [p,q]
    coerce(q) == [q,0$QR]
    retract(x):QR ==
      not(zero? imagE x and zero? imagI x and zero? imagJ x and zero? imagK x)=>
        error "Cannot retract octonion to quaternion."

```

```

    quatern(real x, imagi x,imagj x, imagk x)$QR
    retractIfCan(x):Union(QR,"failed") ==
      not(zero? imagE x and zero? imagI x and zero? imagJ x and zero? imagK x)=>
        "failed"
    quatern(real x, imagi x,imagj x, imagk x)$QR
    x * y == [x.e*y.e-(conjugate y.E)*x.E, y.E*x.e + x.E*(conjugate y.e)]

```

— OCT.dotabb —

```

"OCT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=OCT"]
"OC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OC"]
"QUATCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=QUATCAT"]
"OCT" -> "OC"
"OCT" -> "QUATCAT"

```

domain ODEIFTBL ODEIntensityFunctionsTable

— ODEIntensityFunctionsTable.input —

```

)set break resume
)sys rm -f ODEIntensityFunctionsTable.output
)spool ODEIntensityFunctionsTable.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ODEIntensityFunctionsTable
--R
--R ODEIntensityFunctionsTable is a domain constructor
--R Abbreviation for ODEIntensityFunctionsTable is ODEIFTBL
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ODEIFTBL
--R
--R----- Operations -----
--R clearTheIFTable : () -> Void          showTheIFTable : () -> %
--R iFTable : List(Record(key: Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(DoubleFloat)),yinit: Vector(Expression(DoubleFloat))),yend: Vector(Expression(DoubleFloat))),yfn: Vector(Expression(DoubleFloat)))
--R insert! : Record(key: Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(DoubleFloat))),yinit: Vector(Expression(DoubleFloat))),yfn: Vector(Expression(DoubleFloat)))
--R keys : % -> List(Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(DoubleFloat))),yinit: Vector(Expression(DoubleFloat))),yfn: Vector(Expression(DoubleFloat)))
--R showIntensityFunctions : Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(DoubleFloat))),yinit: Vector(Expression(DoubleFloat))),yfn: Vector(Expression(DoubleFloat)))

```

```
--R
--E 1

)spool
)lisp (bye)
```

— ODEIntensityFunctionsTable.help —

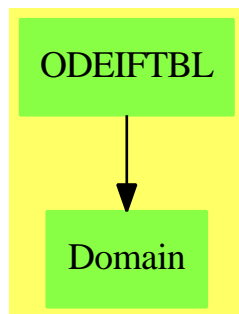
```
=====
ODEIntensityFunctionsTable examples
=====
```

ODEIntensityFunctionsTable() provides a dynamic table and a set of functions to store details found out about sets of ODE's.

See Also:

o)show ODEIntensityFunctionsTable

ODEIntensityFunctionsTable (ODEIFTBL)



Exports:

clearTheIFTable iFTable insert! keys showIntensityFunctions showTheIFTable

— domain ODEIFTBL ODEIntensityFunctionsTable —

```
)abbrev domain ODEIFTBL ODEIntensityFunctionsTable
++ Author: Brian Dupee
++ Date Created: May 1994
++ Date Last Updated: January 1996
```

```

++ Description:
++ \axiom{ODEIntensityFunctionsTable()} provides a dynamic table and a set of
++ functions to store details found out about sets of ODE's.

ODEIntensityFunctionsTable(): E == I where
  LEDF ==> List Expression DoubleFloat
  LEEDF ==> List Equation Expression DoubleFloat
  EEDF ==> Equation Expression DoubleFloat
  VEDF ==> Vector Expression DoubleFloat
  MEDF ==> Matrix Expression DoubleFloat
  MDF ==> Matrix DoubleFloat
  EDF ==> Expression DoubleFloat
  DF ==> DoubleFloat
  F ==> Float
  INT ==> Integer
  CDF ==> Complex DoubleFloat
  LDF ==> List DoubleFloat
  LF ==> List Float
  S ==> Symbol
  LS ==> List Symbol
  MFI ==> Matrix Fraction Integer
  LFI ==> List Fraction Integer
  FI ==> Fraction Integer
  ODEA ==> Record(xinit:DF,xend:DF,fn:VEDF,yinit:LDF,intvals:LDF,g:EDF,abserr:DF,relerr:DF)
  ON ==> Record(additions:INT,multiplications:INT,exponentiations:INT,functionCalls:INT)
  RVE ==> Record(val:EDF,exponent:INT)
  RSS ==> Record(stiffnessFactor:F,stabilityFactor:F)
  ATT ==> Record(stiffness:F,stability:F,expense:F,accuracy:F,intermediateResults:F)
  ROA ==> Record(key:ODEA,entry:ATT)

E ==> with
  showTheIFTTable:() -> $
    ++ showTheIFTTable() returns the current table of intensity functions.
  clearTheIFTTable : () -> Void
    ++ clearTheIFTTable() clears the current table of intensity functions.
  keys : $ -> List(ODEA)
    ++ keys(tab) returns the list of keys of f
  iFTTable: List Record(key:ODEA,entry:ATT) -> $
    ++ iFTTable(l) creates an intensity-functions table from the elements
    ++ of l.
  insert!:Record(key:ODEA,entry:ATT) -> $
    ++ insert!(r) inserts an entry r into theIFTTable
  showIntensityFunctions:ODEA -> Union(ATT,"failed")
    ++ showIntensityFunctions(k) returns the entries in the
    ++ table of intensity functions k.

I ==> add
  Rep := Table(ODEA,ATT)
  import Rep

```

```

theIFTable:$ := empty()$Rep

showTheIFTable():$ ==
  theIFTable

clearTheIFTable():Void ==
  theIFTable := empty()$Rep
  void()$Void

iFTable(l>List Record(key:ODEA,entry:ATT)):$ ==
  theIFTable := table(l)$Rep

insert!(r:Record(key:ODEA,entry:ATT)):$ ==
  insert!(r,theIFTable)$Rep

keys(t:$):List ODEA ==
  keys(t)$Rep

showIntensityFunctions(k:ODEA):Union(ATT,"failed") ==
  search(k,theIFTable)$Rep

```

— ODEIFTBL.dotabb —

```

"ODEIFTBL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ODEIFTBL"]
"Domain" [color="#88FF44"]
"ODEIFTBL" -> "Domain"

```

domain ARRAY1 OneDimensionalArray

— OneDimensionalArray.input —

```

)set break resume
)sys rm -f OneDimensionalArray.output
)spool OneDimensionalArray.output
)set message test on
)set message auto off
)clear all
--S 1 of 9
oneDimensionalArray [i**2 for i in 1..10]
--R

```

```

--R
--R (1) [1,4,9,16,25,36,49,64,81,100]
--R                                         Type: OneDimensionalArray(PositiveInteger)
--E 1

--S 2 of 9
a : ARRAY1 INT := new(10,0)
--R
--R
--R (2) [0,0,0,0,0,0,0,0,0,0]
--R                                         Type: OneDimensionalArray(Integer)
--E 2

--S 3 of 9
for i in 1..10 repeat a.i := i; a
--R
--R
--R (3) [1,2,3,4,5,6,7,8,9,10]
--R                                         Type: OneDimensionalArray(Integer)
--E 3

--S 4 of 9
map!(i +-> i ** 2,a); a
--R
--R
--R (4) [1,4,9,16,25,36,49,64,81,100]
--R                                         Type: OneDimensionalArray(Integer)
--E 4

--S 5 of 9
reverse! a
--R
--R
--R (5) [100,81,64,49,36,25,16,9,4,1]
--R                                         Type: OneDimensionalArray(Integer)
--E 5

--S 6 of 9
swap!(a,4,5); a
--R
--R
--R (6) [100,81,64,36,49,25,16,9,4,1]
--R                                         Type: OneDimensionalArray(Integer)
--E 6

--S 7 of 9
sort! a
--R
--R
--R (7) [1,4,9,16,25,36,49,64,81,100]

```

```

--R                                                    Type: OneDimensionalArray(Integer)
--E 7

--S 8 of 9
b := a(6..10)
--R
--R
--R (8) [36,49,64,81,100]
--R                                                    Type: OneDimensionalArray(Integer)
--E 8

--S 9 of 9
copyInto!(a,b,1)
--R
--R
--R (9) [36,49,64,81,100,36,49,64,81,100]
--R                                                    Type: OneDimensionalArray(Integer)
--E 9
)spool
)lisp (bye)

```

— OneDimensionalArray.help —

=====

OneDimensionalArray examples

=====

The OneDimensionalArray domain is used for storing data in a one-dimensional indexed data structure. Such an array is a homogeneous data structure in that all the entries of the array must belong to the same Axiom domain. Each array has a fixed length specified by the user and arrays are not extensible. The indexing of one-dimensional arrays is one-based. This means that the "first" element of an array is given the index 1.

To create a one-dimensional array, apply the operation `oneDimensionalArray` to a list.

```

oneDimensionalArray [i**2 for i in 1..10]
[1,4,9,16,25,36,49,64,81,100]
Type: OneDimensionalArray PositiveInteger

```

Another approach is to first create `a`, a one-dimensional array of 10 0's. `OneDimensionalArray` has the convenient abbreviation `ARRAY1`.

```

a : ARRAY1 INT := new(10,0)
[0,0,0,0,0,0,0,0,0,0]

```

Type: OneDimensionalArray Integer

Set each i-th element to i, then display the result.

```
for i in 1..10 repeat a.i := i; a
[1,2,3,4,5,6,7,8,9,10]
```

Type: OneDimensionalArray Integer

Square each element by mapping the function $i \mapsto i^2$ onto each element.

```
map!(i ↦ i ** 2,a); a
[1,4,9,16,25,36,49,64,81,100]
```

Type: OneDimensionalArray Integer

Reverse the elements in place.

```
reverse! a
[100,81,64,49,36,25,16,9,4,1]
```

Type: OneDimensionalArray Integer

Swap the 4th and 5th element.

```
swap!(a,4,5); a
[100,81,64,36,49,25,16,9,4,1]
```

Type: OneDimensionalArray Integer

Sort the elements in place.

```
sort! a
[1,4,9,16,25,36,49,64,81,100]
```

Type: OneDimensionalArray Integer

Create a new one-dimensional array b containing the last 5 elements of a.

```
b := a(6..10)
[36,49,64,81,100]
```

Type: OneDimensionalArray Integer

Replace the first 5 elements of a with those of b.

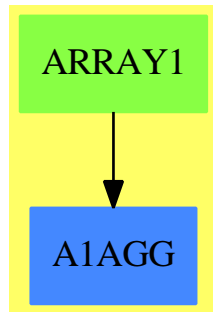
```
copyInto!(a,b,1)
[36,49,64,81,100,36,49,64,81,100]
```

Type: OneDimensionalArray Integer

See Also:

- o)help Vector
- o)help FlexibleArray
- o)show OneDimensionalArray

OneDimensionalArray (ARRAY1)



See

- ⇒ “PrimitiveArray” (PRIMARR) [17](#) on page [2109](#)
- ⇒ “Tuple” (TUPLE) [21](#) on page [2752](#)
- ⇒ “IndexedFlexibleArray” (IFARRAY) [10](#) on page [1222](#)
- ⇒ “FlexibleArray” (FARRAY) [7](#) on page [885](#)
- ⇒ “IndexedOneDimensionalArray” (IARRAY1) [10](#) on page [1243](#)

Exports:

any?	coerce	concat	construct	convert
copy	copyInto!	count	delete	elt
empty	empty?	entries	entry?	eq?
eval	every?	fill!	find	first
hash	index?	indices	insert	latex
less?	map	map!	max	maxIndex
member?	members	merge	min	minIndex
more?	new	oneDimensionalArray	parts	position
qelt	qsetelt!	reduce	remove	removeDuplicates
reverse	reverse!	sample	select	setelt
size?	sort	sort!	sorted?	swap!
#?	?<?	?<=?	?=?	?>?
?>=?	?~=?	?..?		

— domain ARRAY1 OneDimensionalArray —

```
)abbrev domain ARRAY1 OneDimensionalArray
++ Author: Mark Botch
++ Description:
++ This is the domain of 1-based one dimensional arrays
```

```
OneDimensionalArray(S:Type): Exports == Implementation where
```

```

ARRAYMININDEX ==> 1      -- if you want to change this, be my guest
Exports == OneDimensionalArrayAggregate S with
  oneDimensionalArray: List S -> %
    ++ oneDimensionalArray(l) creates an array from a list of elements l
    ++
    ++X oneDimensionalArray [i**2 for i in 1..10]

  oneDimensionalArray: (NonNegativeInteger, S) -> %
    ++ oneDimensionalArray(n,s) creates an array from n copies of element s
    ++
    ++X oneDimensionalArray(10,0.0)

Implementation == IndexedOneDimensionalArray(S, ARRAYMININDEX) add
  oneDimensionalArray(u) ==
    n := #u
    n = 0 => empty()
    a := new(n, first u)
    for i in 2..n for x in rest u repeat a.i := x
    a
  oneDimensionalArray(n,s) == new(n,s)

```

— ARRAY1.dotabb —

```

"ARRAY1" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ARRAY1"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"ARRAY1" -> "A1AGG"

```

domain ONECOMP OnePointCompletion

— OnePointCompletion.input —

```

)set break resume
)sys rm -f OnePointCompletion.output
)spool OnePointCompletion.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OnePointCompletion

```

```

--R
--R OnePointCompletion(R: SetCategory) is a domain constructor
--R Abbreviation for OnePointCompletion is ONECOMP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ONECOMP
--R
--R----- Operations -----
--R ? : % -> % if R has ABELGRP          ?=? : (%,%) -> Boolean
--R 1 : () -> % if R has ORDRING          0 : () -> % if R has ABELGRP
--R coerce : R -> %                      coerce : % -> OutputForm
--R finite? : % -> Boolean                hash : % -> SingleInteger
--R infinite? : % -> Boolean              infinity : () -> %
--R latex : % -> String                  retract : % -> R
--R ?=? : (%,%) -> Boolean
--R ?? : (PositiveInteger,%) -> % if R has ABELGRP
--R ?? : (NonNegativeInteger,%) -> % if R has ABELGRP
--R ?? : (Integer,%) -> % if R has ABELGRP
--R ?? : (%,%) -> % if R has ORDRING
--R ??? : (%,NonNegativeInteger) -> % if R has ORDRING
--R ??? : (%,PositiveInteger) -> % if R has ORDRING
--R ?+? : (%,%) -> % if R has ABELGRP
--R ?-? : (%,%) -> % if R has ABELGRP
--R ?<? : (%,%) -> Boolean if R has ORDRING
--R ?<=? : (%,%) -> Boolean if R has ORDRING
--R ?>? : (%,%) -> Boolean if R has ORDRING
--R ?>=? : (%,%) -> Boolean if R has ORDRING
--R ?? : (%,NonNegativeInteger) -> % if R has ORDRING
--R ?? : (%,PositiveInteger) -> % if R has ORDRING
--R abs : % -> % if R has ORDRING
--R characteristic : () -> NonNegativeInteger if R has ORDRING
--R coerce : Integer -> % if R has ORDRING or R has RETRACT(INT)
--R coerce : Fraction(Integer) -> % if R has RETRACT(FRAC(INT))
--R max : (%,%) -> % if R has ORDRING
--R min : (%,%) -> % if R has ORDRING
--R negative? : % -> Boolean if R has ORDRING
--R one? : % -> Boolean if R has ORDRING
--R positive? : % -> Boolean if R has ORDRING
--R rational : % -> Fraction(Integer) if R has INS
--R rational? : % -> Boolean if R has INS
--R rationalIfCan : % -> Union(Fraction(Integer),"failed") if R has INS
--R recip : % -> Union(%, "failed") if R has ORDRING
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retract : % -> Integer if R has RETRACT(INT)
--R retractIfCan : % -> Union(R, "failed")
--R retractIfCan : % -> Union(Fraction(Integer), "failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(Integer, "failed") if R has RETRACT(INT)
--R sample : () -> % if R has ABELGRP
--R sign : % -> Integer if R has ORDRING
--R subtractIfCan : (%,%) -> Union(%, "failed") if R has ABELGRP
--R zero? : % -> Boolean if R has ABELGRP

```

```
--R
```

```
--E 1
```

```
)spool
```

```
)lisp (bye)
```

— OnePointCompletion.help —

=====

OnePointCompletion examples

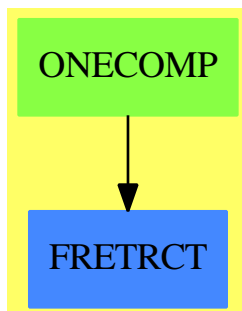
=====

Completion with infinity. Adjunction of a complex infinity to a set.

See Also:

o)show OnePointCompletion

OnePointCompletion (ONECOMP)



See

⇒ “OrderedCompletion” (ORDCOMP) [16](#) on page [1809](#)

Exports:

0	1	abs	characteristic	coerce
coerce	finite?	hash	infinite?	infinity
latex	max	min	negative?	one?
positive?	rational	rational?	rationalIfCan	recip
retract	retractIfCan	sample	sign	subtractIfCan
zero?	-?	?=?	?~=?	?*?
?**?	?+?	?-?	?<?	?<=?
?>?	?>=?	?^?		

— domain ONECOMP OnePointCompletion —

```

)abbrev domain ONECOMP OnePointCompletion
++ Author: Manuel Bronstein
++ Date Created: 4 Oct 1989
++ Date Last Updated: 1 Nov 1989
++ Description:
++ Completion with infinity.
++ Adjunction of a complex infinity to a set.

OnePointCompletion(R:SetCategory): Exports == Implementation where
  B ==> Boolean

Exports ==> Join(SetCategory, FullyRetractableTo R) with
  infinity : () -> %
    ++ infinity() returns infinity.
  finite? : % -> B
    ++ finite?(x) tests if x is finite.
  infinite?: % -> B
    ++ infinite?(x) tests if x is infinite.
  if R has AbelianGroup then AbelianGroup
  if R has OrderedRing then OrderedRing
  if R has IntegerNumberSystem then
    rational?: % -> Boolean
      ++ rational?(x) tests if x is a finite rational number.
    rational : % -> Fraction Integer
      ++ rational(x) returns x as a finite rational number.
      ++ Error: if x is not a rational number.
    rationalIfCan: % -> Union(Fraction Integer, "failed")
      ++ rationalIfCan(x) returns x as a finite rational number if
      ++ it is one, "failed" otherwise.

Implementation ==> add
  Rep := Union(R, "infinity")

  coerce(r:R):% == r
  retract(x:%):R == (x case R => x::R; error "Not finite")
  finite? x == x case R
  infinite? x == x case "infinity"
  infinity() == "infinity"
  retractIfCan(x:%):Union(R, "failed") == (x case R => x::R; "failed")

  coerce(x:%):OutputForm ==
    x case "infinity" => "infinity"::OutputForm
    x::R::OutputForm

  x = y ==
    x case "infinity" => y case "infinity"
    y case "infinity" => false

```

```

x::R = y::R

if R has AbelianGroup then
  0 == 0$R

  n:Integer * x:% ==
    x case "infinity" =>
      zero? n => error "Undefined product"
      infinity()
    n * x::R

  - x ==
    x case "infinity" => error "Undefined inverse"
    - (x::R)

  x + y ==
    x case "infinity" => x
    y case "infinity" => y
    x::R + y::R

if R has OrderedRing then
  fininf: R -> %

  1 == 1$R
  characteristic() == characteristic()$R

  fininf r ==
    zero? r => error "Undefined product"
    infinity()

  x:% * y:% ==
    x case "infinity" =>
      y case "infinity" => y
      fininf(y::R)
    y case "infinity" => fininf(x::R)
    x::R * y::R

  recip x ==
    x case "infinity" => 0
    zero?(x::R) => infinity()
    (u := recip(x::R)) case "failed" => "failed"
    u::R::%

  x < y ==
    x case "infinity" => false      -- do not change the order
    y case "infinity" => true       -- of those two tests
    x::R < y::R

if R has IntegerNumberSystem then
  rational? x == finite? x

```

```

rational x == rational(retract(x)@R)

rationalIfCan x ==
  (r:= retractIfCan(x)@Union(R,"failed")) case "failed" =>"failed"
  rational(r::R)

```

— ONECOMP.dotabb —

```

"ONECOMP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ONECOMP"]
"FRETRCT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRETRCT"]
"ONECOMP" -> "FRETRCT"

```

domain OMCONN OpenMathConnection

— OpenMathConnection.input —

```

)set break resume
)sys rm -f OpenMathConnection.output
)spool OpenMathConnection.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OpenMathConnection
--R OpenMathConnection is a domain constructor
--R Abbreviation for OpenMathConnection is OMCONN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for OMCONN
--R
--R----- Operations -----
--R OMcloseConn : % -> Void          OMmakeConn : SingleInteger -> %
--R OMbindTCP : (% ,SingleInteger) -> Boolean
--R OMconnInDevice : % -> OpenMathDevice
--R OMconnOutDevice : % -> OpenMathDevice
--R OMconnectTCP : (% ,String,SingleInteger) -> Boolean
--R
--E 1

)spool
)lisp (bye)

```

— OpenMathConnection.help —

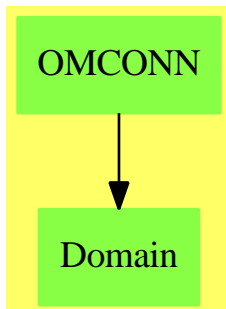
```
=====
OpenMathConnection examples
=====
```

OpenMathConnection provides low-level functions
for handling connections to and from OpenMathDevice's.

See Also:

- o)show OpenMathConnection
- o)show OpenMathDevice

OpenMathConnection (OMCONN)



See

⇒ “OpenMathEncoding” (OMENC) [16](#) on page [1789](#)

⇒ “OpenMathDevice” (OMDEV) [16](#) on page [1784](#)

Exports:

```
OMbindTCP      OMcloseConn  OMconnectTCP  OMconnInDevice
OMconnOutDevice OMmakeConn
```

— domain OMCONN OpenMathConnection —

```
)abbrev domain OMCONN OpenMathConnection
++ Author: Vilya Harvey
++ Description:
++ \spadtype{OpenMathConnection} provides low-level functions
++ for handling connections to and from \spadtype{OpenMathDevice}s.
```



```

OpenMathConnection(): with
  OMmakeConn    : SingleInteger -> % ++ \spad{OMmakeConn}
  OMcloseConn   : % -> Void ++ \spad{OMcloseConn}
  OMconnInDevice: %-> OpenMathDevice ++ \spad{OMconnInDevice:}
  OMconnOutDevice: %-> OpenMathDevice ++ \spad{OMconnOutDevice:}
  OMconnectTCP  : (%, String, SingleInteger) -> Boolean ++ \spad{OMconnectTCP}
  OMbindTCP     : (%, SingleInteger) -> Boolean ++ \spad{OMbindTCP}
== add
  OMmakeConn(timeout: SingleInteger): % == OM_-MAKECONN(timeout)$Lisp
  OMcloseConn(conn: %): Void == OM_-CLOSECONN(conn)$Lisp

  OMconnInDevice(conn: %): OpenMathDevice ==
    OM_-GETCONNINDEV(conn)$Lisp
  OMconnOutDevice(conn: %): OpenMathDevice ==
    OM_-GETCONNOUTDEV(conn)$Lisp

  OMconnectTCP(conn: %, host: String, port: SingleInteger): Boolean ==
    OM_-CONNECTTCP(conn, host, port)$Lisp
  OMbindTCP(conn: %, port: SingleInteger): Boolean ==
    OM_-BINDTCP(conn, port)$Lisp

```

— OMCONN.dotabb —

```

"OMCONN" [color="#88FF44",href="bookvol10.3.pdf#nameddest=OMCONN"]
"Domain" [color="#88FF44"]
"OMCONN" -> "Domain"

```

domain OMDEV OpenMathDevice

— OpenMathDevice.input —

```

)set break resume
)sys rm -f OpenMathDevice.output
)spool OpenMathDevice.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OpenMathDevice

```

```

--R OpenMathDevice is a domain constructor
--R Abbreviation for OpenMathDevice is OMDEV
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for OMDEV
--R
--R----- Operations -----
--R OMclose : % -> Void          OMgetApp : % -> Void
--R OMgetAtp : % -> Void          OMgetAttr : % -> Void
--R OMgetBVar : % -> Void         OMgetBind : % -> Void
--R OMgetEndApp : % -> Void       OMgetEndAtp : % -> Void
--R OMgetEndAttr : % -> Void      OMgetEndBVar : % -> Void
--R OMgetEndBind : % -> Void      OMgetEndError : % -> Void
--R OMgetEndObject : % -> Void    OMgetError : % -> Void
--R OMgetFloat : % -> DoubleFloat OMgetInteger : % -> Integer
--R OMgetObject : % -> Void       OMgetString : % -> String
--R OMgetType : % -> Symbol       OMgetVariable : % -> Symbol
--R OMputApp : % -> Void          OMputAtp : % -> Void
--R OMputAttr : % -> Void         OMputBVar : % -> Void
--R OMputBind : % -> Void         OMputEndApp : % -> Void
--R OMputEndAtp : % -> Void       OMputEndAttr : % -> Void
--R OMputEndBVar : % -> Void      OMputEndBind : % -> Void
--R OMputEndError : % -> Void     OMputEndObject : % -> Void
--R OMputError : % -> Void        OMputObject : % -> Void
--R OMputString : (%,String) -> Void
--R OMgetSymbol : % -> Record(cd: String,name: String)
--R OMopenFile : (String,String,OpenMathEncoding) -> %
--R OMopenString : (String,OpenMathEncoding) -> %
--R OMputFloat : (%,DoubleFloat) -> Void
--R OMputInteger : (%,Integer) -> Void
--R OMputSymbol : (%,String,String) -> Void
--R OMputVariable : (%,Symbol) -> Void
--R OMsetEncoding : (%,OpenMathEncoding) -> Void
--R
--E 1

)spool
)lisp (bye)

```

— OpenMathDevice.help —

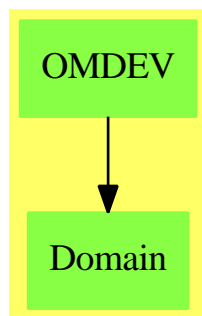
OpenMathDevice examples

OpenMathDevice provides support for reading and writing openMath objects to files, strings etc. It also provides access to low-level operations from within the interpreter.

See Also:

- o `)show OpenMathDevice`
- o `)show OpenMathConnection`

OpenMathDevice (OMDEV)



See

⇒ “OpenMathEncoding” (OMENC) [16](#) on page [1789](#)

⇒ “OpenMathConnection” (OMCONN) [16](#) on page [1781](#)

Exports:

OMclose	OMgetApp	OMgetAtp	OMgetAttr
OMgetBVar	OMgetBind	OMgetEndApp	OMgetEndAtp
OMgetEndAttr	OMgetEndBVar	OMgetEndBind	OMgetEndError
OMgetEndObject	OMgetError	OMgetFloat	OMgetInteger
OMgetObject	OMgetString	OMgetType	OMgetVariable
OMputApp	OMputAtp	OMputAttr	OMputBVar
OMputBind	OMputEndApp	OMputEndAtp	OMputEndAttr
OMputEndBVar	OMputEndBind	OMputEndError	OMputEndObject
OMputError	OMputObject	OMputString	OMgetSymbol
OMopenFile	OMopenString	OMputFloat	OMputInteger
OMputSymbol	OMputVariable	OMsetEncoding	

— domain OMDEV OpenMathDevice —

```

)abbrev domain OMDEV OpenMathDevice
++ Author: Vilya Harvey
++ Description:
++ \spadtype{OpenMathDevice} provides support for reading
++ and writing openMath objects to files, strings etc. It also provides
++ access to low-level operations from within the interpreter.
  
```

```

OpenMathDevice(): with
  OMopenFile      : (String, String, OpenMathEncoding) -> %
  ++ OMopenFile(f,mode,enc) opens file \axiom{f} for reading or writing
  ++ OpenMath objects (depending on \axiom{mode} which can be "r", "w"
  ++ or "a" for read, write and append respectively), in the encoding
  ++ \axiom{enc}.
  OMopenString    : (String, OpenMathEncoding) -> %
  ++ OMopenString(s,mode) opens the string \axiom{s} for reading or writing
  ++ OpenMath objects in encoding \axiom{enc}.
  OMclose         : % -> Void
  ++ OMclose(dev) closes \axiom{dev}, flushing output if necessary.
  OMsetEncoding   : (% , OpenMathEncoding) -> Void
  ++ OMsetEncoding(dev,enc) sets the encoding used for reading or writing
  ++ OpenMath objects to or from \axiom{dev} to \axiom{enc}.
  OMputApp        : % -> Void
  ++ OMputApp(dev) writes a begin application token to \axiom{dev}.
  OMputAtp        : % -> Void
  ++ OMputAtp(dev) writes a begin attribute pair token to \axiom{dev}.
  OMputAttr       : % -> Void
  ++ OMputAttr(dev) writes a begin attribute token to \axiom{dev}.
  OMputBind       : % -> Void
  ++ OMputBind(dev) writes a begin binder token to \axiom{dev}.
  OMputBVar       : % -> Void
  ++ OMputBVar(dev) writes a begin bound variable list token to \axiom{dev}.
  OMputError      : % -> Void
  ++ OMputError(dev) writes a begin error token to \axiom{dev}.
  OMputObject     : % -> Void
  ++ OMputObject(dev) writes a begin object token to \axiom{dev}.
  OMputEndApp     : % -> Void
  ++ OMputEndApp(dev) writes an end application token to \axiom{dev}.
  OMputEndAtp     : % -> Void
  ++ OMputEndAtp(dev) writes an end attribute pair token to \axiom{dev}.
  OMputEndAttr    : % -> Void
  ++ OMputEndAttr(dev) writes an end attribute token to \axiom{dev}.
  OMputEndBind    : % -> Void
  ++ OMputEndBind(dev) writes an end binder token to \axiom{dev}.
  OMputEndBVar    : % -> Void
  ++ OMputEndBVar(dev) writes an end bound variable list token to \axiom{dev}.
  OMputEndError   : % -> Void
  ++ OMputEndError(dev) writes an end error token to \axiom{dev}.
  OMputEndObject  : % -> Void
  ++ OMputEndObject(dev) writes an end object token to \axiom{dev}.
  OMputInteger    : (% , Integer) -> Void
  ++ OMputInteger(dev,i) writes the integer \axiom{i} to \axiom{dev}.
  OMputFloat      : (% , DoubleFloat) -> Void
  ++ OMputFloat(dev,i) writes the float \axiom{i} to \axiom{dev}.
  OMputVariable   : (% , Symbol) -> Void
  ++ OMputVariable(dev,i) writes the variable \axiom{i} to \axiom{dev}.
  OMputString     : (% , String) -> Void
  ++ OMputString(dev,i) writes the string \axiom{i} to \axiom{dev}.

```

```

OMputSymbol    : (% , String, String) -> Void
++ OMputSymbol(dev,cd,s) writes the symbol \axiom{s} from CD \axiom{cd}
++ to \axiom{dev}.

OMgetApp       : % -> Void
++ OMgetApp(dev) reads a begin application token from \axiom{dev}.
OMgetAtp       : % -> Void
++ OMgetAtp(dev) reads a begin attribute pair token from \axiom{dev}.
OMgetAttr      : % -> Void
++ OMgetAttr(dev) reads a begin attribute token from \axiom{dev}.
OMgetBind      : % -> Void
++ OMgetBind(dev) reads a begin binder token from \axiom{dev}.
OMgetBVar      : % -> Void
++ OMgetBVar(dev) reads a begin bound variable list token from \axiom{dev}.
OMgetError     : % -> Void
++ OMgetError(dev) reads a begin error token from \axiom{dev}.
OMgetObject    : % -> Void
++ OMgetObject(dev) reads a begin object token from \axiom{dev}.
OMgetEndApp    : % -> Void
++ OMgetEndApp(dev) reads an end application token from \axiom{dev}.
OMgetEndAtp    : % -> Void
++ OMgetEndAtp(dev) reads an end attribute pair token from \axiom{dev}.
OMgetEndAttr   : % -> Void
++ OMgetEndAttr(dev) reads an end attribute token from \axiom{dev}.
OMgetEndBind   : % -> Void
++ OMgetEndBind(dev) reads an end binder token from \axiom{dev}.
OMgetEndBVar   : % -> Void
++ OMgetEndBVar(dev) reads an end bound variable list token from \axiom{dev}.
OMgetEndError  : % -> Void
++ OMgetEndError(dev) reads an end error token from \axiom{dev}.
OMgetEndObject : % -> Void
++ OMgetEndObject(dev) reads an end object token from \axiom{dev}.
OMgetInteger   : % -> Integer
++ OMgetInteger(dev) reads an integer from \axiom{dev}.
OMgetFloat     : % -> DoubleFloat
++ OMgetFloat(dev) reads a float from \axiom{dev}.
OMgetVariable  : % -> Symbol
++ OMgetVariable(dev) reads a variable from \axiom{dev}.
OMgetString    : % -> String
++ OMgetString(dev) reads a string from \axiom{dev}.
OMgetSymbol    : % -> Record(cd:String, name:String)
++ OMgetSymbol(dev) reads a symbol from \axiom{dev}.

OMgetType      : % -> Symbol
++ OMgetType(dev) returns the type of the next object on \axiom{dev}.
== add
OMopenFile(fname: String, fmode: String, enc: OpenMathEncoding): % ==
  OM_-OPENFILEDEV(fname, fmode, enc)$Lisp
OMopenString(str: String, enc: OpenMathEncoding): % ==
  OM_-OPENSTRINGDEV(str, enc)$Lisp

```

```

OMclose(dev: %): Void ==
  OM_-CLOSEDEV(dev)$Lisp
OMsetEncoding(dev: %, enc: OpenMathEncoding): Void ==
  OM_-SETDEVENCODING(dev, enc)$Lisp

OMputApp(dev: %): Void == OM_-PUTAPP(dev)$Lisp
OMputAtp(dev: %): Void == OM_-PUTATP(dev)$Lisp
OMputAttr(dev: %): Void == OM_-PUTATTR(dev)$Lisp
OMputBind(dev: %): Void == OM_-PUTBIND(dev)$Lisp
OMputBVar(dev: %): Void == OM_-PUTBVAR(dev)$Lisp
OMputError(dev: %): Void == OM_-PUTERROR(dev)$Lisp
OMputObject(dev: %): Void == OM_-PUTOBJECT(dev)$Lisp
OMputEndApp(dev: %): Void == OM_-PUTENDAPP(dev)$Lisp
OMputEndAtp(dev: %): Void == OM_-PUTENDATP(dev)$Lisp
OMputEndAttr(dev: %): Void == OM_-PUTENDATTR(dev)$Lisp
OMputEndBind(dev: %): Void == OM_-PUTENDBIND(dev)$Lisp
OMputEndBVar(dev: %): Void == OM_-PUTENDBVAR(dev)$Lisp
OMputEndError(dev: %): Void == OM_-PUTENDERROR(dev)$Lisp
OMputEndObject(dev: %): Void == OM_-PUTENDOBJECT(dev)$Lisp
OMputInteger(dev: %, i: Integer): Void == OM_-PUTINT(dev, i)$Lisp
OMputFloat(dev: %, f: DoubleFloat): Void == OM_-PUTFLOAT(dev, f)$Lisp
--OMputByteArray(dev: %, b: Array Byte): Void == OM_-PUTBYTEARRAY(dev, b)$Lisp
OMputVariable(dev: %, v: Symbol): Void == OM_-PUTVAR(dev, v)$Lisp
OMputString(dev: %, s: String): Void == OM_-PUTSTRING(dev, s)$Lisp
OMputSymbol(dev: %, cd: String, nm: String): Void == OM_-PUTSYMBOL(dev, cd, nm)$Lisp

OMgetApp(dev: %): Void == OM_-GETAPP(dev)$Lisp
OMgetAtp(dev: %): Void == OM_-GETATP(dev)$Lisp
OMgetAttr(dev: %): Void == OM_-GETATTR(dev)$Lisp
OMgetBind(dev: %): Void == OM_-GETBIND(dev)$Lisp
OMgetBVar(dev: %): Void == OM_-GETBVAR(dev)$Lisp
OMgetError(dev: %): Void == OM_-GETERROR(dev)$Lisp
OMgetObject(dev: %): Void == OM_-GETOBJECT(dev)$Lisp
OMgetEndApp(dev: %): Void == OM_-GETENDAPP(dev)$Lisp
OMgetEndAtp(dev: %): Void == OM_-GETENDATP(dev)$Lisp
OMgetEndAttr(dev: %): Void == OM_-GETENDATTR(dev)$Lisp
OMgetEndBind(dev: %): Void == OM_-GETENDBIND(dev)$Lisp
OMgetEndBVar(dev: %): Void == OM_-GETENDBVAR(dev)$Lisp
OMgetEndError(dev: %): Void == OM_-GETENDERROR(dev)$Lisp
OMgetEndObject(dev: %): Void == OM_-GETENDOBJECT(dev)$Lisp
OMgetInteger(dev: %): Integer == OM_-GETINT(dev)$Lisp
OMgetFloat(dev: %): DoubleFloat == OM_-GETFLOAT(dev)$Lisp
--OMgetByteArray(dev: %): Array Byte == OM_-GETBYTEARRAY(dev)$Lisp
OMgetVariable(dev: %): Symbol == OM_-GETVAR(dev)$Lisp
OMgetString(dev: %): String == OM_-GETSTRING(dev)$Lisp
OMgetSymbol(dev: %): Record(cd:String, name:String) == OM_-GETSYMBOL(dev)$Lisp

OMgetType(dev: %): Symbol == OM_-GETTYPE(dev)$Lisp

```

— OMDEV.dotabb —

```
"OMDEV" [color="#88FF44",href="bookvol10.3.pdf#nameddest=OMDEV"]
"Domain" [color="#88FF44"]
"OMDEV" -> "Domain"
```

domain OMENC OpenMathEncoding

— OpenMathEncoding.input —

```
)set break resume
)sys rm -f OpenMathEncoding.output
)spool OpenMathEncoding.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OpenMathEncoding
--R OpenMathEncoding is a domain constructor
--R Abbreviation for OpenMathEncoding is OMENC
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for OMENC
--R
--R----- Operations -----
--R ?=? : (%,%) -> Boolean          OMencodingBinary : () -> %
--R OMencodingSGML : () -> %       OMencodingUnknown : () -> %
--R OMencodingXML : () -> %       coerce : % -> OutputForm
--R hash : % -> SingleInteger      latex : % -> String
--R ?~=? : (%,%) -> Boolean
--R
--E 1

)spool
)lisp (bye)
```

— OpenMathEncoding.help —

=====

OpenMathEncoding examples

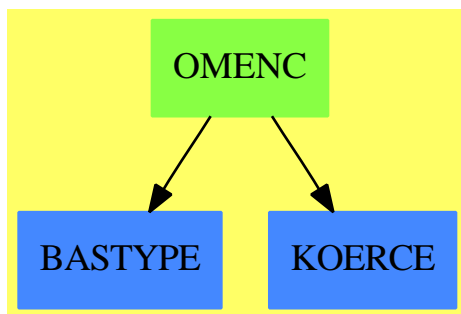
=====

OpenMathEncoding is the set of valid OpenMath encodings.

See Also:

o)show OpenMathEncoding

OpenMathEncoding (OMENC)



See

⇒ “OpenMathDevice” (OMDEV) [16](#) on page [1784](#)

⇒ “OpenMathConnection” (OMCONN) [16](#) on page [1781](#)

Exports:

coerce	hash	latex	OMencodingBinary	OMencodingSGML
OMencodingUnknown	OMencodingXML	?=?	?~=?	

— domain OMENC OpenMathEncoding —

```

)abbrev domain OMENC OpenMathEncoding
++ Author: Vilya Harvey
++ Description:
++ \spadtype{OpenMathEncoding} is the set of valid OpenMath encodings.
  
```

```

OpenMathEncoding(): SetCategory with
  OMencodingUnknown : () -> %
  ++ OMencodingUnknown() is the constant for unknown encoding types. If this
  ++ is used on an input device, the encoding will be autodetected.
  ++ It is invalid to use it on an output device.
  OMencodingXML      : () -> %
  ++ OMencodingXML() is the constant for the OpenMath XML encoding.
  OMencodingSGML     : () -> %
  
```



```

++ OMencodingSGML() is the constant for the deprecated OpenMath SGML encoding.
OMencodingBinary : () -> %
++ OMencodingBinary() is the constant for the OpenMath binary encoding.
== add
Rep := SingleInteger

=(u,v) == (u=v)$Rep

import Rep

coerce(u) ==
  u::Rep = 0$Rep => "Unknown"::OutputForm
  u::Rep = 1$Rep => "Binary"::OutputForm
  u::Rep = 2::Rep => "XML"::OutputForm
  u::Rep = 3::Rep => "SGML"::OutputForm
  error "Bogus OpenMath Encoding Type"

OMencodingUnknown(): % == 0::Rep
OMencodingBinary(): % == 1::Rep
OMencodingXML(): % == 2::Rep
OMencodingSGML(): % == 3::Rep

```

— OMENC.dotabb —

```

"OMENC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=OMENC"]
"BATYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BATYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"OMENC" -> "BATYPER"
"OMENC" -> "KOERCE"

```

domain OMERR OpenMathError

— OpenMathError.input —

```

)set break resume
)sys rm -f OpenMathError.output
)spool OpenMathError.output
)set message test on
)set message auto off
)clear all

```

```

--S 1 of 1
)show OpenMathError
--R
--R OpenMathError is a domain constructor
--R Abbreviation for OpenMathError is OMERR
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for OMERR
--R
--R----- Operations -----
--R ?? : (% ,%) -> Boolean          coerce : % -> OutputForm
--R errorInfo : % -> List(Symbol)   hash : % -> SingleInteger
--R latex : % -> String             ?~=? : (% ,%) -> Boolean
--R errorKind : % -> OpenMathErrorKind
--R omError : (OpenMathErrorKind,List(Symbol)) -> %
--R
--E 1

)spool
)lisp (bye)

```

— OpenMathError.help —

=====

OpenMathError examples

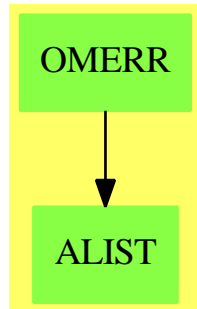
=====

OpenMathError is the domain of OpenMath errors.

See Also:

o)show OpenMathError

OpenMathError (OMERR)



See

⇒ “OpenMathErrorKind” (OMERRK) [16](#) on page [1794](#)

Exports:

```

coerce      errorInfo  errorKind  hash    latex
omError     ??         ? =?

```

— domain OMERR OpenMathError —

```

)abbrev domain OMERR OpenMathError
++ Author: Vilya Harvey
++ Description:
++ \spadtype{OpenMathError} is the domain of OpenMath errors.

OpenMathError() : SetCategory with
  errorKind : % -> OpenMathErrorKind
  ++ errorKind(u) returns the type of error which u represents.
  errorInfo : % -> List Symbol
  ++ errorInfo(u) returns information about the error u.
  omError    : (OpenMathErrorKind, List Symbol) -> %
  ++ omError(k,l) creates an instance of OpenMathError.
== add
Rep := Record(err:OpenMathErrorKind, info:List Symbol)

import List String

coerce(e:%):OutputForm ==
  OMParseError? e.err => message "Error parsing OpenMath object"
  infoSize := #(e.info)
  OMUnknownCD? e.err =>
--      not one? infoSize => error "Malformed info list in OMUnknownCD"
--      not (infoSize = 1) => error "Malformed info list in OMUnknownCD"
--      message concat("Cannot handle CD ",string first e.info)
  OMUnknownSymbol? e.err =>
--      not 2=infoSize => error "Malformed info list in OMUnknownSymbol"
--      message concat ["Cannot handle Symbol ",

```

```

        string e.info.2, " from CD ", string e.info.1]
OMReadError? e.err =>
  message "OpenMath read error"
  error "Malformed OpenMath Error"

omError(e:OpenMathErrorKind,i>List Symbol):% == [e,i]$Rep

errorKind(e:%):OpenMathErrorKind == e.err
errorInfo(e:%):List Symbol == e.info

```

— OMERR.dotabb —

```

"OMERR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=OMERR"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"OMERR" -> "ALIST"

```

domain OMERRK OpenMathErrorKind

— OpenMathErrorKind.input —

```

)set break resume
)sys rm -f OpenMathErrorKind.output
)spool OpenMathErrorKind.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OpenMathErrorKind
--R OpenMathErrorKind is a domain constructor
--R Abbreviation for OpenMathErrorKind is OMERRK
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for OMERRK
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean      OMParseError? : % -> Boolean
--R OMReadError? : % -> Boolean  OMUnknownCD? : % -> Boolean
--R OMUnknownSymbol? : % -> Boolean  coerce : Symbol -> %
--R coerce : % -> OutputForm      hash : % -> SingleInteger
--R latex : % -> String           ?~=? : (%,% ) -> Boolean

```

```
--R
--E 1
```

```
)spool
)lisp (bye)
```

— OpenMathErrorKind.help —

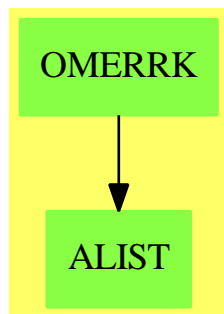
```
=====
OpenMathErrorKind examples
=====
```

OpenMathErrorKind represents different kinds of OpenMath errors:
specifically parse errors, unknown CD or symbol errors, and read errors.

See Also:

o)show OpenMathErrorKind

OpenMathErrorKind (OMERRK)



See

⇒ “OpenMathError” (OMERR) [16](#) on page [1792](#)

Exports:

OMParseError? OMReadError? OMUnknownCD? OMUnknownSymbol? coerce hash latex

— domain OMERRK OpenMathErrorKind —

```
)abbrev domain OMERRK OpenMathErrorKind
++ Author: Vilya Harvey
```

```

++ Description:
++ \spadtype{OpenMathErrorKind} represents different kinds
++ of OpenMath errors: specifically parse errors, unknown CD or symbol
++ errors, and read errors.

OpenMathErrorKind() : SetCategory with
  coerce          : Symbol -> %
  ++ coerce(u) creates an OpenMath error object of an appropriate type if
  ++ \axiom{u} is one of \axiom{OMParseError}, \axiom{OMReadError},
  ++ \axiom{OMUnknownCD} or \axiom{OMUnknownSymbol}, otherwise it
  ++ raises a runtime error.
  OMParseError?   : % -> Boolean
  ++ OMParseError?(u) tests whether u is an OpenMath parsing error.
  OMUnknownCD?    : % -> Boolean
  ++ OMUnknownCD?(u) tests whether u is an OpenMath unknown CD error.
  OMUnknownSymbol? : % -> Boolean
  ++ OMUnknownSymbol?(u) tests whether u is an OpenMath unknown symbol error.
  OMReadError?    : % -> Boolean
  ++ OMReadError?(u) tests whether u is an OpenMath read error.
== add
Rep := Union(parseError:"OMParseError", unknownCD:"OMUnknownCD",
             unknownSymbol:"OMUnknownSymbol",readError:"OMReadError")

OMParseError?(u) == (u case parseError)$Rep
OMUnknownCD?(u) == (u case unknownCD)$Rep
OMUnknownSymbol?(u) == (u case unknownSymbol)$Rep
OMReadError?(u) == (u case readError)$Rep

coerce(s:Symbol):% ==
  s = OMParseError    => ["OMParseError"]$Rep
  s = OMUnknownCD     => ["OMUnknownCD"]$Rep
  s = OMUnknownSymbol => ["OMUnknownSymbol"]$Rep
  s = OMReadError     => ["OMReadError"]$Rep
  error concat(string s, " is not a valid OpenMathErrorKind.")

a = b == (a=b)$Rep

coerce(e:%):OutputForm == coerce(e)$Rep

```

— OMERRK.dotabb —

"OMERRK" [color="#88FF44",href="bookvol10.3.pdf#nameddest=OMERRK"]
 "ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
 "OMERRK" -> "ALIST"

— Operator.input —

```
)set break resume
)sys rm -f Operator.output
)spool Operator.output
)set message test on
)set message auto off
)clear all
--S 1 of 21
R := SQMATRIX(2, INT)
--R 
--R 
--R   (1) SquareMatrix(2,Integer)
--R                                     Type: Domain
--E 1

--S 2 of 21
t := operator("tilde") :: OP(R)
--R 
--R 
--R   (2) tilde
--R                                     Type: Operator(SquareMatrix(2,Integer))
--E 2

--S 3 of 21
)set expose add constructor Operator
--R 
--I   Operator is now explicitly exposed in frame frame0
--E 3

--S 4 of 21
evaluate(t, m +-> transpose m)
--R 
--R 
--R   (3) tilde
--R                                     Type: Operator(SquareMatrix(2,Integer))
--E 4

--S 5 of 21
s : R := matrix [ [0, 1], [1, 0] ]
--R 
--R 
--R      +0  1+
--R   (4) |    |
--R      +1  0+
```

--E 5

--S 6 of 21

rho := t * s

--R

--R

--R +0 1+

--R (5) tilde| |

--R +1 0+

--R

Type: Operator(SquareMatrix(2,Integer))

--E 6

--S 7 of 21

z := rho**4 - 1

--R

--R

--R +0 1+ +0 1+ +0 1+ +0 1+

--R (6) - 1 + tilde| |tilde| |tilde| |tilde| |

--R +1 0+ +1 0+ +1 0+ +1 0+

--R

Type: Operator(SquareMatrix(2,Integer))

--E 7

--S 8 of 21

m:R := matrix [[1, 2], [3, 4]]

--R

--R

--R +1 2+

--R (7) | |

--R +3 4+

--R

Type: SquareMatrix(2,Integer)

--E 8

--S 9 of 21

z m

--R

--R

--R +0 0+

--R (8) | |

--R +0 0+

--R

Type: SquareMatrix(2,Integer)

--E 9

--S 10 of 21

rho m

--R

--R

--R +3 1+

--R (9) | |

--R +4 2+

--R

Type: SquareMatrix(2,Integer)

--E 10

--S 11 of 21

rho rho m

--R

--R

--R +4 3+

--R (10) | |

--R +2 1+

--R

Type: SquareMatrix(2,Integer)

--E 11

--S 12 of 21

(rho^3) m

--R

--R

--R +2 4+

--R (11) | |

--R +1 3+

--R

Type: SquareMatrix(2,Integer)

--E 12

--S 13 of 21

b := t * s - s * t

--R

--R

--R +0 1+ +0 1+

--R (12) - | |tilde + tilde| |

--R +1 0+ +1 0+

--R

Type: Operator(SquareMatrix(2,Integer))

--E 13

--S 14 of 21

b m

--R

--R

--R +1 - 3+

--R (13) | |

--R +3 - 1+

--R

Type: SquareMatrix(2,Integer)

--E 14

--S 15 of 21

L n ==

n = 0 => 1

n = 1 => x

(2*n-1)/n * x * L(n-1) - (n-1)/n * L(n-2)

--R

--R

Type: Void

--E 15

```

--S 16 of 21
dx := operator("D") :: OP(POLY FRAC INT)
--R
--R
--R (15) D
--R
--R                                         Type: Operator(Polynomial(Fraction(Integer)))
--E 16

--S 17 of 21
evaluate(dx, p +-> D(p, 'x))
--R
--R
--R (16) D
--R
--R                                         Type: Operator(Polynomial(Fraction(Integer)))
--E 17

--S 18 of 21
E n == (1 - x**2) * dx**2 - 2 * x * dx + n*(n+1)
--R
--R
--R                                         Type: Void
--E 18

--S 19 of 21
L 15
--R
--R Compiling function L with type Integer -> Polynomial(Fraction(
--R Integer))
--R Compiling function L as a recurrence relation.
--R
--R (18)
--R      9694845 15 35102025 13 50702925 11 37182145 9 14549535 7
--R      ----- x - ----- x + ----- x - ----- x + ----- x
--R      2048      2048      2048      2048      2048
--R +
--R      2909907 5 255255 3 6435
--R      - ----- x + ----- x - ----- x
--R      2048      2048      2048
--R
--R                                         Type: Polynomial(Fraction(Integer))
--E 19

--S 20 of 21
E 15
--R
--R Compiling function E with type PositiveInteger -> Operator(
--R Polynomial(Fraction(Integer)))
--R
--R
--R      2      2
--R (19) 240 - 2x D - (x - 1)D
--R
--R                                         Type: Operator(Polynomial(Fraction(Integer)))

```

```

--E 20

--S 21 of 21
(E 15)(L 15)
--R
--R
--R (20) 0
--R
--R                                         Type: Polynomial(Fraction(Integer))
--E 21
)spool
)lisp (bye)

```

— Operator.help —

=====

Operator examples

=====

Given any ring R, the ring of the Integer-linear operators over R is called `Operator(R)`. To create an operator over R, first create a basic operator using the operation operator, and then convert it to `Operator(R)` for the R you want.

We choose R to be the two by two matrices over the integers.

```

R := SQMATRIX(2, INT)
   SquareMatrix(2,Integer)
                                     Type: Domain

```

Create the operator tilde on R.

```

t := operator("tilde") :: OP(R)
   tilde
                                     Type: Operator SquareMatrix(2,Integer)

```

Since `Operator` is unexposed we must either package-call operations from it, or expose it explicitly. For convenience we will do the latter.

Expose `Operator`.

```

)set expose add constructor Operator

```

To attach an evaluation function (from R to R) to an operator over R, use `evaluate(op, f)` where `op` is an operator over R and `f` is a function `R -> R`. This needs to be done only once when the operator is defined. Note that `f` must be Integer-linear (that is, $f(ax+y) = a f(x) + f(y)$ for any integer `a`, and any `x` and `y` in R).

```
evaluate(t, m +> transpose m)
tilde
```

Type: Operator SquareMatrix(2,Integer)

```
s : R := matrix [ [0, 1], [1, 0] ]  
+0  1+  
|    |  
+1  0+
```

Type: SquareMatrix(2,Integer)

```

rho := t * s
      +0 1+
      tilde| |
      +1 0+
Type: Operator SquareMatrix(2,Integer)

```

```

z := rho**4 - 1
      +0 1+      +0 1+      +0 1+      +0 1+
- 1 + tilde| |tilde| |tilde| |tilde| |
      +1 0+      +1 0+      +1 0+      +1 0+
Type: Operator SquareMatrix(2,Integer)

```

```
m:=R := matrix [ [1, 2], [3, 4] ]  
      +1  2+  
      |   |  
      +3  4+  
  
Type: SquareMatrix(2,Integer)
```

```

z m
+0  0+
|    |
+0  0+

Type: SquareMatrix(2,Integer)

```

As you have probably guessed by now, ρ acts on matrices by rotating the elements clockwise.

```
rho m
  +3  1+
  |   |
  +4  2+
                                     Type: SquareMatrix(2,Integer)
```

```
rho rho m
  +4  3+
  |   |
  +2  1+
                                     Type: SquareMatrix(2,Integer)
```

```
(rho^3) m
  +2  4+
  |   |
  +1  3+
                                     Type: SquareMatrix(2,Integer)
```

Do the swapping of rows and transposition commute? We can check by computing their bracket.

```
b := t * s - s * t
      +0  1+          +0  1+
      - |   |tilde + tilde|   |
      +1  0+          +1  0+
                                     Type: Operator SquareMatrix(2,Integer)
```

Now apply it to m .

```
b m
  +1  - 3+
  |       |
  +3  - 1+
                                     Type: SquareMatrix(2,Integer)
```

Next we demonstrate how to define a differential operator on a polynomial ring.

This is the recursive definition of the n -th Legendre polynomial.

```
L n ==
  n = 0 => 1
  n = 1 => x
  (2*n-1)/n * x * L(n-1) - (n-1)/n * L(n-2)
                                     Type: Void
```

Create the differential operator d/dx on polynomials in x over the

rational numbers.

```
dx := operator("D") :: OP(POLY FRAC INT)
D
Type: Operator Polynomial Fraction Integer
```

Now attach the map to it.

```
evaluate(dx, p +-> D(p, 'x))
D
Type: Operator Polynomial Fraction Integer
```

This is the differential equation satisfied by the n-th Legendre polynomial.

```
E n == (1 - x**2) * dx**2 - 2 * x * dx + n*(n+1)
Type: Void
```

Now we verify this for n = 15. Here is the polynomial.

```
L 15
 9694845 15 35102025 13 50702925 11 37182145 9 14549535 7
----- x - ----- x + ----- x - ----- x + ----- x
 2048      2048      2048      2048      2048
+
 2909907 5 255255 3 6435
- ----- x + ----- x - ----- x
 2048      2048      2048
Type: Polynomial Fraction Integer
```

Here is the operator.

```
E 15
240 - 2x D - (x - 1)D
Type: Operator Polynomial Fraction Integer
```

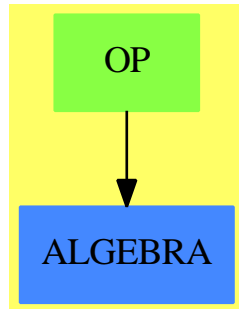
Here is the evaluation.

```
(E 15)(L 15)
0
Type: Polynomial Fraction Integer
```

See Also:

o)show Operator

Operator (OP)



See

⇒ “ModuleOperator” (MODOP) [14](#) on page [1647](#)

Exports:

0	1	adjoint	characteristic	charthRoot
coerce	conjug	evaluate	evaluateInverse	makeop
hash	latex	one?	opeval	recip
retract	retractIfCan	sample	subtractIfCan	zero?
?~=?	?*?	?**?	?^?	?..?
?+?	?-?	-?	?=?	

— domain OP Operator —

```
)abbrev domain OP Operator
++ Author: Manuel Bronstein
++ Date Created: 15 May 1990
++ Date Last Updated: 12 February 1993
++ Description:
++ Algebra of ADDITIVE operators over a ring.
```

```
Operator(R: Ring) == ModuleOperator(R,R)
```

—————

— OP.dotabb —

```
"OP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=OP"]
"ALGEBRA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ALGEBRA"]
"OP" -> "ALGEBRA"
```

—————

domain OMLO OppositeMonogenicLinearOperator

— OppositeMonogenicLinearOperator.input —

```

)set break resume
)sys rm -f OppositeMonogenicLinearOperator.output
)spool OppositeMonogenicLinearOperator.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OppositeMonogenicLinearOperator
--R
--R OppositeMonogenicLinearOperator(P: MonogenicLinearOperator(R),R: Ring) is a domain constructor
--R Abbreviation for OppositeMonogenicLinearOperator is OMLO
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for OMLO
--R
--R----- Operations -----
--R ??? : (R,%) -> %          ??? : (%,R) -> %
--R ??? : (%,%) -> %          ??? : (Integer,%) -> %
--R ??? : (PositiveInteger,%) -> %    ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %          ?-? : (%,%) -> %
--R -? : % -> %              ?=? : (%,%) -> Boolean
--R D : % -> % if P has DIFRING      1 : () -> %
--R 0 : () -> %                ?? : (%,PositiveInteger) -> %
--R coerce : Integer -> %          coerce : % -> OutputForm
--R degree : % -> NonNegativeInteger hash : % -> SingleInteger
--R latex : % -> String            leadingCoefficient : % -> R
--R one? : % -> Boolean            op : P -> %
--R po : % -> P                  recip : % -> Union(%, "failed")
--R reductum : % -> %            sample : () -> %
--R zero? : % -> Boolean          ?~=? : (%,%) -> Boolean
--R ??? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R D : (%,NonNegativeInteger) -> % if P has DIFRING
--R ?? : (%,NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R coefficient : (%,NonNegativeInteger) -> R
--R coerce : R -> % if R has COMRING
--R differentiate : % -> % if P has DIFRING
--R differentiate : (%,NonNegativeInteger) -> % if P has DIFRING
--R minimumDegree : % -> NonNegativeInteger
--R monomial : (R,NonNegativeInteger) -> %
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R
--E 1

```



```
)spool
)lisp (bye)
```

— OppositeMonogenicLinearOperator.help —

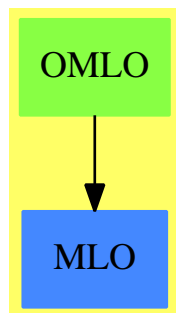
```
=====
OppositeMonogenicLinearOperator examples
=====
```

This constructor creates the MonogenicLinearOperator domain which is "opposite" in the ring sense to P. That is, as sets $P = \mathbb{Q}$ but $a * b$ in \mathbb{Q} is equal to $b * a$ in P.

See Also:

```
o )show OppositeMonogenicLinearOperator
```

OppositeMonogenicLinearOperator (OMLO)



See

⇒ “OrdinaryDifferentialRing” (ODR) [16](#) on page [1858](#)

⇒ “DirectProductModule” (DPMO) [5](#) on page [557](#)

⇒ “DirectProductMatrixModule” (DPMM) [5](#) on page [552](#)

Exports:

0	1	characteristic	coefficient	coerce
D	degree	differentiate	hash	latex
leadingCoefficient	minimumDegree	monomial	one?	op
po	recip	reductum	sample	subtractIfCan
zero?	?^?	?~=?	?*?	?**?
?+?	?-?	-?	?=?	

— domain OMLO OppositeMonogenicLinearOperator —

```

)abbrev domain OMLO OppositeMonogenicLinearOperator
++ Author: Stephen M. Watt
++ Date Created: 1986
++ Date Last Updated: May 30, 1991
++ Description:
++ This constructor creates the \spadtype{MonogenicLinearOperator} domain
++ which is ‘‘opposite’’ in the ring sense to P.
++ That is, as sets \spad{P = $} but \spad{a * b} in \spad{$} is equal to
++ \spad{b * a} in P.

```

```

OppositeMonogenicLinearOperator(P, R): OPRcat == OPRdef where
  P: MonogenicLinearOperator(R)
  R: Ring

```

```

  OPRcat == MonogenicLinearOperator(R) with
    if P has DifferentialRing then DifferentialRing
    op: P -> $ ++ op(p) creates a value in $ equal to p in P.
    po: $ -> P ++ po(q) creates a value in P equal to q in $.

```

```

  OPRdef == P add
    Rep := P
    x, y: $
    a: P
    op a == a: $
    po x == x: P
    x*y == (y:P) *$P (x:P)
    coerce(x): OutputForm == prefix(op::OutputForm, [coerce(x:P)$P])

```

— OMLO.dotabb —

```

"OMLO" [color="#88FF44",href="bookvol10.3.pdf#nameddest=OMLO"]
"ML0" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ML0"]
"OMLO" -> "ML0"

```

domain ORDCOMP OrderedCompletion

— OrderedCompletion.input —

```

)set break resume
)sys rm -f OrderedCompletion.output
)spool OrderedCompletion.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OrderedCompletion
--R
--R OrderedCompletion(R: SetCategory) is a domain constructor
--R Abbreviation for OrderedCompletion is ORDCOMP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ORDCOMP
--R
--R----- Operations -----
--R ? : % -> % if R has ABELGRP           ?? : (%,%) -> Boolean
--R 1 : () -> % if R has ORDRING           0 : () -> % if R has ABELGRP
--R coerce : R -> %                       coerce : % -> OutputForm
--R finite? : % -> Boolean                 hash : % -> SingleInteger
--R infinite? : % -> Boolean               latex : % -> String
--R minusInfinity : () -> %               plusInfinity : () -> %
--R retract : % -> R                      whatInfinity : % -> SingleInteger
--R ~=? : (%,%) -> Boolean
--R ??*? : (PositiveInteger,%) -> % if R has ABELGRP
--R ??*? : (NonNegativeInteger,%) -> % if R has ABELGRP
--R ??*? : (Integer,%) -> % if R has ABELGRP
--R ??*? : (%,%) -> % if R has ORDRING
--R ???*? : (%,NonNegativeInteger) -> % if R has ORDRING
--R ???*? : (%,PositiveInteger) -> % if R has ORDRING
--R ?+? : (%,%) -> % if R has ABELGRP
--R ?-? : (%,%) -> % if R has ABELGRP
--R ?<? : (%,%) -> Boolean if R has ORDRING
--R ?<=? : (%,%) -> Boolean if R has ORDRING
--R ?>? : (%,%) -> Boolean if R has ORDRING
--R ?>=? : (%,%) -> Boolean if R has ORDRING
--R ?^? : (%,NonNegativeInteger) -> % if R has ORDRING
--R ?^? : (%,PositiveInteger) -> % if R has ORDRING
--R abs : % -> % if R has ORDRING
--R characteristic : () -> NonNegativeInteger if R has ORDRING
--R coerce : Integer -> % if R has ORDRING or R has RETRACT(INT)
--R coerce : Fraction(Integer) -> % if R has RETRACT(FRAC(INT))
--R max : (%,%) -> % if R has ORDRING
--R min : (%,%) -> % if R has ORDRING
--R negative? : % -> Boolean if R has ORDRING
--R one? : % -> Boolean if R has ORDRING
--R positive? : % -> Boolean if R has ORDRING
--R rational : % -> Fraction(Integer) if R has INS
--R rational? : % -> Boolean if R has INS
--R rationalIfCan : % -> Union(Fraction(Integer),"failed") if R has INS

```

```

--R recip : % -> Union(%, "failed") if R has ORDRING
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retract : % -> Integer if R has RETRACT(INT)
--R retractIfCan : % -> Union(R, "failed")
--R retractIfCan : % -> Union(Fraction(Integer), "failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(Integer, "failed") if R has RETRACT(INT)
--R sample : () -> % if R has ABELGRP
--R sign : % -> Integer if R has ORDRING
--R subtractIfCan : (%, %) -> Union(%, "failed") if R has ABELGRP
--R zero? : % -> Boolean if R has ABELGRP
--R
--E 1

)spool
)lisp (bye)

```

— OrderedCompletion.help —

=====

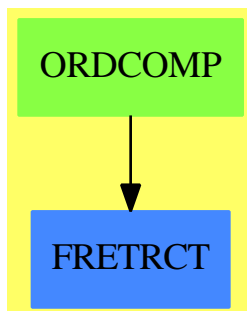
OrderedCompletion examples

=====

Completion with + and - infinity.
Adjunction of two real infinites quantities to a set.

See Also:
o)show OrderedCompletion

OrderedCompletion (ORDCOMP)



See

\Rightarrow “OnePointCompletion” (ONECOMP) 16 on page 1777

Exports:

0	1	abs	characteristic	coerce
finite?	hash	infinite?	latex	max
min	minusInfinity	negative?	one?	plusInfinity
positive?	rational	rational?	rationalIfCan	recip
retract	retractIfCan	sample	sign	subtractIfCan
whatInfinity	zero?	?~=?	?*?	?**?
?+?	?-?	?<?	?<=?	?>?
?>=?	?^?	-?	?=?	

— domain ORDCOMP OrderedCompletion —

```

)abbrev domain ORDCOMP OrderedCompletion
++ Author: Manuel Bronstein
++ Date Created: 4 Oct 1989
++ Date Last Updated: 1 Nov 1989
++ Description:
++ Completion with + and - infinity.
++ Adjunction of two real infinities quantities to a set.

OrderedCompletion(R:SetCategory): Exports == Implementation where
  B ==> Boolean

Exports ==> Join(SetCategory, FullyRetractableTo R) with
  plusInfinity : () -> %      ++ plusInfinity() returns +infinity.
  minusInfinity: () -> %      ++ minusInfinity() returns -infinity.
  finite?      : % -> B
    ++ finite?(x) tests if x is finite.
  infinite?    : % -> B
    ++ infinite?(x) tests if x is +infinity or -infinity,
  whatInfinity : % -> SingleInteger
    ++ whatInfinity(x) returns 0 if x is finite,
    ++ 1 if x is +infinity, and -1 if x is -infinity.
  if R has AbelianGroup then AbelianGroup
  if R has OrderedRing then OrderedRing
  if R has IntegerNumberSystem then
    rational?: % -> Boolean
      ++ rational?(x) tests if x is a finite rational number.
    rational : % -> Fraction Integer
      ++ rational(x) returns x as a finite rational number.
      ++ Error: if x cannot be so converted.
    rationalIfCan: % -> Union(Fraction Integer, "failed")
      ++ rationalIfCan(x) returns x as a finite rational number if
      ++ it is one and "failed" otherwise.

Implementation ==> add
  Rep := Union(fin:R, inf:B) -- true = +infinity, false = -infinity

```

```

coerce(r:R):%          == [r]
retract(x:%):R         == (x case fin => x.fin; error "Not finite")
finite? x              == x case fin
infinite? x            == x case inf
plusInfinity()         == [true]
minusInfinity()        == [false]

retractIfCan(x:%):Union(R, "failed") ==
  x case fin => x.fin
  "failed"

coerce(x:%):OutputForm ==
  x case fin => (x.fin)::OutputForm
  e := "infinity"::OutputForm
  x.inf => empty() + e
  - e

whatInfinity x ==
  x case fin => 0
  x.inf => 1
  -1

x = y ==
  x case inf =>
    y case inf => not xor(x.inf, y.inf)
    false
  y case inf => false
  x.fin = y.fin

if R has AbelianGroup then
  0 == [0$R]

n:Integer * x:% ==
  x case inf =>
    n > 0 => x
    n < 0 => [not(x.inf)]
    error "Undefined product"
  [n * x.fin]

- x ==
  x case inf => [not(x.inf)]
  [- (x.fin)]

x + y ==
  x case inf =>
    y case fin => x
    xor(x.inf, y.inf) => error "Undefined sum"
  x
  y case inf => y

```

```

    [x.fin + y.fin]

if R has OrderedRing then
  fininf: (B, R) -> %

  1 == [1$R]
  characteristic() == characteristic()$R

  fininf(b, r) ==
    r > 0 => [b]
    r < 0 => [not b]
    error "Undefined product"

  x:% * y:% ==
    x case inf =>
      y case inf =>
        xor(x.inf, y.inf) => minusInfinity()
        plusInfinity()
        fininf(x.inf, y.fin)
      y case inf => fininf(y.inf, x.fin)
    [x.fin * y.fin]

  recip x ==
    x case inf => 0
    (u := recip(x.fin)) case "failed" => "failed"
    [u::R]

  x < y ==
    x case inf =>
      y case inf =>
        xor(x.inf, y.inf) => y.inf
        false
      not(x.inf)
    y case inf => y.inf
    x.fin < y.fin

if R has IntegerNumberSystem then
  rational? x == finite? x
  rational x == rational(retract(x)@R)

  rationalIfCan x ==
    (r:= retractIfCan(x)@Union(R,"failed")) case "failed" =>"failed"
    rational(r::R)

```

```
"ORDCOMP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ORDCOMP"]
"FRETRCT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRETRCT"]
"ORDCOMP" -> "FRETRCT"
```

domain ODP OrderedDirectProduct

— OrderedDirectProduct.input —

```
)set break resume
)sys rm -f OrderedDirectProduct.output
)spool OrderedDirectProduct.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OrderedDirectProduct
--R OrderedDirectProduct(dim: NonNegativeInteger,S: OrderedAbelianMonoidSup,f: ((Vector(S),Vector(S)) -> Bo
--R Abbreviation for OrderedDirectProduct is ODP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ODP
--R
--R----- Operations -----
--R -? : % -> % if S has RING          1 : () -> % if S has MONOID
--R 0 : () -> % if S has CABMON        coerce : % -> Vector(S)
--R copy : % -> %                      directProduct : Vector(S) -> %
--R ?.? : (%,Integer) -> S             elt : (%,Integer,S) -> S
--R empty : () -> %                   empty? : % -> Boolean
--R entries : % -> List(S)             eq? : (%,%) -> Boolean
--R index? : (Integer,%) -> Boolean     indices : % -> List(Integer)
--R map : ((S -> S),%) -> %            qelt : (%,Integer) -> S
--R sample : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?? : (PositiveInteger,%) -> % if S has ABELSG
--R ?? : (NonNegativeInteger,%) -> % if S has CABMON
--R ?? : (S,%) -> % if S has MONOID
--R ?? : (%,S) -> % if S has MONOID
--R ?? : (%,%) -> % if S has MONOID
--R ?? : (Integer,%) -> % if S has RING
--R ??? : (%,PositiveInteger) -> % if S has MONOID
--R ??? : (%,NonNegativeInteger) -> % if S has MONOID
--R ?+? : (%,%) -> % if S has ABELSG
--R ?-? : (%,%) -> % if S has RING
--R ?/? : (%,S) -> % if S has FIELD
```



```

--R ?<? : (%,%) -> Boolean if S has OAMONS or S has ORDRING
--R ?<=? : (%,%) -> Boolean if S has OAMONS or S has ORDRING
--R ?=? : (%,%) -> Boolean if S has SETCAT
--R ?>? : (%,%) -> Boolean if S has OAMONS or S has ORDRING
--R ?>=? : (%,%) -> Boolean if S has OAMONS or S has ORDRING
--R D : (%,(S -> S)) -> % if S has RING
--R D : (%,(S -> S),NonNegativeInteger) -> % if S has RING
--R D : (%,List(Symbol),List(NonNegativeInteger)) -> % if S has PDRING(SYMBOL) and S has RING
--R D : (%,Symbol,NonNegativeInteger) -> % if S has PDRING(SYMBOL) and S has RING
--R D : (%,List(Symbol)) -> % if S has PDRING(SYMBOL) and S has RING
--R D : (%,Symbol) -> % if S has PDRING(SYMBOL) and S has RING
--R D : (%,NonNegativeInteger) -> % if S has DIFRING and S has RING
--R D : % -> % if S has DIFRING and S has RING
--R ?? : (%,PositiveInteger) -> % if S has MONOID
--R ?? : (%,NonNegativeInteger) -> % if S has MONOID
--R abs : % -> % if S has ORDRING
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R characteristic : () -> NonNegativeInteger if S has RING
--R coerce : S -> % if S has SETCAT
--R coerce : Fraction(Integer) -> % if S has RETRACT(FRAC(INT)) and S has SETCAT
--R coerce : Integer -> % if S has RETRACT(INT) and S has SETCAT or S has RING
--R coerce : % -> OutputForm if S has SETCAT
--R count : (S,%) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R differentiate : (%,(S -> S)) -> % if S has RING
--R differentiate : (%,(S -> S),NonNegativeInteger) -> % if S has RING
--R differentiate : (%,List(Symbol),List(NonNegativeInteger)) -> % if S has PDRING(SYMBOL) and S
--R differentiate : (%,Symbol,NonNegativeInteger) -> % if S has PDRING(SYMBOL) and S has RING
--R differentiate : (%,List(Symbol)) -> % if S has PDRING(SYMBOL) and S has RING
--R differentiate : (%,Symbol) -> % if S has PDRING(SYMBOL) and S has RING
--R differentiate : (%,NonNegativeInteger) -> % if S has DIFRING and S has RING
--R differentiate : % -> % if S has DIFRING and S has RING
--R dimension : () -> CardinalNumber if S has FIELD
--R dot : (%,%) -> S if S has RING
--R entry? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R enumerate : () -> List(%) if S has FINITE
--R eval : (%,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (%,S) -> % if $ has shallowlyMutable
--R first : % -> S if Integer has ORDSET
--R hash : % -> SingleInteger if S has SETCAT
--R index : PositiveInteger -> % if S has FINITE
--R latex : % -> String if S has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R lookup : % -> PositiveInteger if S has FINITE
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R max : (%,%) -> % if S has OAMONS or S has ORDRING

```

```

--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R members : % -> List(S) if $ has finiteAggregate
--R min : (%,%) -> % if S has OAMONS or S has ORDRING
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (%,NonNegativeInteger) -> Boolean
--R negative? : % -> Boolean if S has ORDRING
--R one? : % -> Boolean if S has MONOID
--R parts : % -> List(S) if $ has finiteAggregate
--R positive? : % -> Boolean if S has ORDRING
--R qsetelt! : (%,Integer,S) -> S if $ has shallowlyMutable
--R random : () -> % if S has FINITE
--R recip : % -> Union(%, "failed") if S has MONOID
--R reducedSystem : Matrix(%) -> Matrix(S) if S has RING
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(S),vec: Vector(S)) if S has RING
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) if S has LIN
--R reducedSystem : Matrix(%) -> Matrix(Integer) if S has LINEP(INT) and S has RING
--R retract : % -> S if S has SETCAT
--R retract : % -> Fraction(Integer) if S has RETRACT(FRAC(INT)) and S has SETCAT
--R retract : % -> Integer if S has RETRACT(INT) and S has SETCAT
--R retractIfCan : % -> Union(S, "failed") if S has SETCAT
--R retractIfCan : % -> Union(Fraction(Integer), "failed") if S has RETRACT(FRAC(INT)) and S has SETCAT
--R retractIfCan : % -> Union(Integer, "failed") if S has RETRACT(INT) and S has SETCAT
--R setelt : (%,Integer,S) -> S if $ has shallowlyMutable
--R sign : % -> Integer if S has ORDRING
--R size : () -> NonNegativeInteger if S has FINITE
--R size? : (%,NonNegativeInteger) -> Boolean
--R subtractIfCan : (%,%) -> Union(%, "failed") if S has CABMON
--R sup : (%,%) -> % if S has OAMONS
--R swap! : (%,Integer,Integer) -> Void if $ has shallowlyMutable
--R unitVector : PositiveInteger -> % if S has RING
--R zero? : % -> Boolean if S has CABMON
--R ~=?: (%,%) -> Boolean if S has SETCAT
--R
--E 1

)spool
)lisp (bye)

```

— OrderedDirectProduct.help —

```

=====
OrderedDirectProduct examples
=====

```

This type represents the finite direct or cartesian product of an underlying ordered component type. The ordering on the type is determined

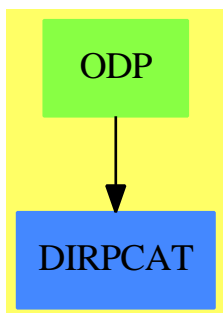
by its third argument which represents the less than function on vectors. This type is a suitable third argument for `GeneralDistributedMultivariatePolynomial`.

See Also:

o `)show OrderedDirectProduct`

—————→

OrderedDirectProduct (ODP)



See

⇒ “HomogeneousDirectProduct” (HDP) 9 on page 1172

⇒ “SplitHomogeneousDirectProduct” (SHDP) 20 on page 2508

Exports:

0	1	abs	any?	characteristic
coerce	copy	count	D	differentiate
dimension	directProduct	dot	elt	empty
empty?	entries	entry?	eq?	eval
every?	fill!	first	hash	index
index?	indices	latex	less?	lookup
map	map!	max	maxIndex	member?
members	min	minIndex	more?	negative?
one?	parts	positive?	qelt	qsetelt!
random	recip	reducedSystem	retract	retractIfCan
sample	setelt	sign	size	size?
subtractIfCan	sup	swap!	unitVector	zero?
#?	?*?	?**?	?+?	?-?
?/?	?<?	?<=?	?=?	?>?
?>=?	?^?	?~=?	~?	..?

— domain ODP OrderedDirectProduct —

`)abbrev domain ODP OrderedDirectProduct`

```

-- all direct product category domains must be compiled
-- without subsumption, set SourceLevelSubset to EQUAL
--)bo $noSubsumption := true

++ Author: Mark Botch
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors: Vector, DirectProduct
++ Also See: HomogeneousDirectProduct, SplitHomogeneousDirectProduct
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This type represents the finite direct or cartesian product of an
++ underlying ordered component type. The ordering on the type is determined
++ by its third argument which represents the less than function on
++ vectors. This type is a suitable third argument for
++ \spadtype{GeneralDistributedMultivariatePolynomial}.

```

```

OrderedDirectProduct(dim:NonNegativeInteger,
                     S:OrderedAbelianMonoidSup,
                     f:(Vector(S),Vector(S))->Boolean):T
    == C where
T == DirectProductCategory(dim,S)
C == DirectProduct(dim,S) add
    Rep:=Vector(S)
    x:% < y:% == f(x::Rep,y::Rep)

```

— ODP.dotabb —

```

"ODP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ODP"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"ODP" -> "DIRPCAT"

```

domain OF MONOID OrderedFreeMonoid

— OrderedFreeMonoid.input —

```

)set break resume

```

```

)sys rm -f OrderedFreeMonoid.output
)spool OrderedFreeMonoid.output
)set message test on
)set message auto off
)clear all

--S 1 of 24
m1:=(x*y*y*z)$OFMONOID(Symbol)
--R
--R
--R      2
--R  (1)  x y z
--R
--R                                          Type: OrderedFreeMonoid(Symbol)
--E 1

--S 2 of 24
m2:=(x*y)$OFMONOID(Symbol)
--R
--R
--R  (2)  x y
--R
--R                                          Type: OrderedFreeMonoid(Symbol)
--E 2

--S 3 of 24
lquo(m1,m2)
--R
--R
--R  (3)  y z
--R
--R                                          Type: Union(OrderedFreeMonoid(Symbol),...)
--E 3

--S 4 of 24
m3:=(y*y)$OFMONOID(Symbol)
--R
--R
--R      2
--R  (4)  y
--R
--R                                          Type: OrderedFreeMonoid(Symbol)
--E 4

--S 5 of 24
divide(m1,m2)
--R
--R
--R  (5)  [lm= y z,rm= "failed"]
--R                                          Type: Union(Record(lm: Union(OrderedFreeMonoid(Symbol),"failed"),...)
--E 5

--S 6 of 24
divide(m1,m3)

```

```

--R
--R
--R (6) [lm= "failed",rm= "failed"]
--IType: Union(Record(lm: Union(OrderedFreeMonoid(Symbol),"failed"),...
--E 6

--S 7 of 24
m4:=(y^3)$OFMONOID(Symbol)
--R
--R
--R      3
--R (7)  y
--R
--R                                         Type: OrderedFreeMonoid(Symbol)
--E 7

--S 8 of 24
divide(m1,m4)
--R
--R
--R (8) [lm= "failed",rm= "failed"]
--IType: Union(Record(lm: Union(OrderedFreeMonoid(Symbol),"failed"),...
--E 8

)set function compile on

-- Build the non-commutative algebra h=k[x,y] and then make computations
-- in h using some predefined rules for x and y. For example, giving
--   x*y*x=y*x*y
--   x*x=a*x+b
--   y*y=a*y+b
-- where a and b are generic elements of k.
-- Then reduce the polynomials in x and y according to the previous rules.
-- That is, given
--   (x+y)^2   ( = x^2+x*y+y*x+y^2)
-- should reduce to
--   a*(x+y)+2*b+x*y+y*x

-- Generic elements of k (OVAR is OrderedVariableList)

--S 9 of 24
C ==> OVAR [a,b]
--R
--R
--R                                         Type: Void
--E 9

-- Commutative Field: k = Q[a,b]
-- Q = Fraction Integer
-- SMP = SparseMultivariatePolynomials

```

```

--S 10 of 24
K ==> SMP(FRAC INT,C)
--R
--R
--R                                          Type: Void
--E 10

-- Non-commutative variables

--S 11 of 24
V ==> OVAR [x,y]
--R
--R
--R                                          Type: Void
--E 11

-- Non-commuative algebra k=k[x,y]
-- XDPOLY XDistributedPolynomial

--S 12 of 24
H ==> XDPOLY(V,K)
--R
--R
--R                                          Type: Void
--E 12

-- Free Monoid

--S 13 of 24
M ==> OFMONOID V
--R
--R
--R                                          Type: Void
--E 13

-- Substitution rules are applied to words from the monoid over the
-- variables and return polynomials

--S 14 of 24
subs(w:M):H ==
  -- x*y*x = y*x*y
  n1:=lquo(w,(x::V*y::V*x::V)$M)$M
  n1 case "failed" => monom(w,1)$H
  -- x*x = a*x+b
  n2:=lquo(w,(x::V^2)$M)$M
  n2 case "failed" => monom(w,1)$H
  -- y*y = a*y+b
  n3:=lquo(w,(y::V^2)$M)$M
  n3 case "failed" => monom(w,1)$H
  monom(n3,1)$H * (a::K*y::V+b::K)$M * monom(n3,1)$H
  monom(n2,1)$H * (a::K*x::V+b::K)$H * monom(n2,1)$H
  monom(n1,1)$H * (y::V*x::V*y::V)$H * monom(n1,1)$H
--R
--R  Function declaration subs : OrderedFreeMonoid(OrderedVariableList([x

```

```

--R      ,y])) -> XDistributedPolynomial(OrderedVariableList([x,y]),
--R      SparseMultivariatePolynomial(Fraction(Integer),
--R      OrderedVariableList([a,b]))) has been added to workspace.
--R
--R                                                    Type: Void
--E 14

-- Apply rules to a term. Keep coefficients
--S 15 of 24
newterm(x:Record(k:M,c:K)):H == x.c*subs(x,k)
--R
--R      Function declaration newterm : Record(k: OrderedFreeMonoid(
--R      OrderedVariableList([x,y])),c: SparseMultivariatePolynomial(
--R      Fraction(Integer),OrderedVariableList([a,b]))) ->
--R      XDistributedPolynomial(OrderedVariableList([x,y]),
--R      SparseMultivariatePolynomial(Fraction(Integer),
--R      OrderedVariableList([a,b]))) has been added to workspace.
--R
--R                                                    Type: Void
--E 15

-- Reconstruct the polynomial term by term

--S 16 of 24
newpoly(t:H):H == reduce(+,map(newterm,listOfTerms(t)))
--R
--R      Function declaration newpoly : XDistributedPolynomial(
--R      OrderedVariableList([x,y]),SparseMultivariatePolynomial(Fraction(
--R      Integer),OrderedVariableList([a,b]))) -> XDistributedPolynomial(
--R      OrderedVariableList([x,y]),SparseMultivariatePolynomial(Fraction(
--R      Integer),OrderedVariableList([a,b]))) has been added to
--R      workspace.
--R
--R                                                    Type: Void
--E 16

-- Example calculations

--S 17 of 24
p1:(x::V+y::V)$H^2
--R
--R
--RDaly Bug
--R      Category, domain or package constructor ^ is not available.
--E 17

--S 18 of 24
newpoly(p1)
--R
--R      Compiling function newpoly with type XDistributedPolynomial(
--R      OrderedVariableList([x,y]),SparseMultivariatePolynomial(Fraction(
--R      Integer),OrderedVariableList([a,b]))) -> XDistributedPolynomial(
--R      OrderedVariableList([x,y]),SparseMultivariatePolynomial(Fraction(

```



```

--R      Integer),OrderedVariableList([a,b]))
--R  There are no library operations named subs
--R    Use HyperDoc Browse or issue
--R                                     )what op subs
--R    to learn if there is any operation containing " subs " in its
--R    name.
--R  Cannot find a definition or applicable library operation named subs
--R    with argument type(s)
--RRecord(k: OrderedFreeMonoid(OrderedVariableList([x,y])),c: SparseMultivariatePolynomial(Fraction(Integer),
--R                                     Variable(k)
--R
--R    Perhaps you should use "@" to indicate the required return type,
--R    or "$" to specify which version of the function you need.
--R  AXIOM will attempt to step through and interpret the code.
--R  Compiling function newterm with type Record(k: OrderedFreeMonoid(
--R    OrderedVariableList([x,y])),c: SparseMultivariatePolynomial(
--R    Fraction(Integer),OrderedVariableList([a,b])) ->
--R    XDistributedPolynomial(OrderedVariableList([x,y]),
--R    SparseMultivariatePolynomial(Fraction(Integer),
--R    OrderedVariableList([a,b])))
--R  There are no library operations named subs
--R    Use HyperDoc Browse or issue
--R                                     )what op subs
--R    to learn if there is any operation containing " subs " in its
--R    name.
--R
--RDaly Bug
--R  Cannot find a definition or applicable library operation named subs
--R    with argument type(s)
--RRecord(k: OrderedFreeMonoid(OrderedVariableList([x,y])),c: SparseMultivariatePolynomial(Fraction(Integer),
--R                                     Variable(k)
--R
--R    Perhaps you should use "@" to indicate the required return type,
--R    or "$" to specify which version of the function you need.
--E 18

--S 19 of 24
p2:=(x::V+y::V)$H^3
--R
--R
--R      3      2      2      2      2      3
--R  (17)  y  + y x + y x y + y x  + x y  + x y x + x y + x
--RType: XDistributedPolynomial(OrderedVariableList([x,y]),...)
--E 19

--S 20 of 24
pNew:=newpoly(p2)
--R
--R  There are no library operations named subs
--R    Use HyperDoc Browse or issue

```

```

--R                                     )what op subs
--R      to learn if there is any operation containing " subs " in its
--R      name.
--R      Cannot find a definition or applicable library operation named subs
--R      with argument type(s)
--R Record(k: OrderedFreeMonoid(OrderedVariableList([x,y])),c: SparseMultivariatePolynomial(Fraction(Integer)
--R                                     Variable(k)
--R
--R      Perhaps you should use "@" to indicate the required return type,
--R      or "$" to specify which version of the function you need.
--R      AXIOM will attempt to step through and interpret the code.
--R      Compiling function newterm with type Record(k: OrderedFreeMonoid(
--R      OrderedVariableList([x,y])),c: SparseMultivariatePolynomial(
--R      Fraction(Integer),OrderedVariableList([a,b]))) ->
--R      XDistributedPolynomial(OrderedVariableList([x,y]),
--R      SparseMultivariatePolynomial(Fraction(Integer),
--R      OrderedVariableList([a,b])))
--R      There are no library operations named subs
--R      Use HyperDoc Browse or issue
--R                                     )what op subs
--R      to learn if there is any operation containing " subs " in its
--R      name.
--R
--R Daly Bug
--R      Cannot find a definition or applicable library operation named subs
--R      with argument type(s)
--R Record(k: OrderedFreeMonoid(OrderedVariableList([x,y])),c: SparseMultivariatePolynomial(Fraction(Integer)
--R                                     Variable(k)
--R
--R      Perhaps you should use "@" to indicate the required return type,
--R      or "$" to specify which version of the function you need.
--E 20

-- But the rules should be applied more than once
--S 21 of 24
while pNew ~= p2 repeat
  p2 := pNew
  pNew := newpoly(p2)
--R
--R      There are no library operations named subs
--R      Use HyperDoc Browse or issue
--R                                     )what op subs
--R      to learn if there is any operation containing " subs " in its
--R      name.
--R      Cannot find a definition or applicable library operation named subs
--R      with argument type(s)
--R Record(k: OrderedFreeMonoid(OrderedVariableList([x,y])),c: SparseMultivariatePolynomial(Fraction(Integer)
--R                                     Variable(k)
--R
--R      Perhaps you should use "@" to indicate the required return type,

```


--S 24 of 24

reduce(p2)

--R

--R Compiling function newpoly with type XDistributedPolynomial(
--R OrderedVariableList([x,y]),SparseMultivariatePolynomial(Fraction(
--R Integer),OrderedVariableList([a,b]))) -> XDistributedPolynomial(
--R OrderedVariableList([x,y]),SparseMultivariatePolynomial(Fraction(
--R Integer),OrderedVariableList([a,b])))

--R Compiling function reduce with type XDistributedPolynomial(
--R OrderedVariableList([x,y]),SparseMultivariatePolynomial(Fraction(
--R Integer),OrderedVariableList([a,b]))) -> XDistributedPolynomial(
--R OrderedVariableList([x,y]),SparseMultivariatePolynomial(Fraction(
--R Integer),OrderedVariableList([a,b])))

--R There are no library operations named subs

--R Use HyperDoc Browse or issue

--R)what op subs

--R to learn if there is any operation containing " subs " in its
--R name.

--R Cannot find a definition or applicable library operation named subs

--R with argument type(s)

--R Record(k: OrderedFreeMonoid(OrderedVariableList([x,y])),c: SparseMultivariatePolynomial(Fraction(Integer),
--R Variable(k))

--R

--R Perhaps you should use "@" to indicate the required return type,
--R or "\$" to specify which version of the function you need.

--R AXIOM will attempt to step through and interpret the code.

--R Compiling function newterm with type Record(k: OrderedFreeMonoid(
--R OrderedVariableList([x,y])),c: SparseMultivariatePolynomial(
--R Fraction(Integer),OrderedVariableList([a,b]))) ->
--R XDistributedPolynomial(OrderedVariableList([x,y]),
--R SparseMultivariatePolynomial(Fraction(Integer),
--R OrderedVariableList([a,b])))

--R There are no library operations named subs

--R Use HyperDoc Browse or issue

--R)what op subs

--R to learn if there is any operation containing " subs " in its
--R name.

--R

--R Daly Bug

--R Cannot find a definition or applicable library operation named subs

--R with argument type(s)

--R Record(k: OrderedFreeMonoid(OrderedVariableList([x,y])),c: SparseMultivariatePolynomial(Fraction(Integer),
--R Variable(k))

--R

--R Perhaps you should use "@" to indicate the required return type,
--R or "\$" to specify which version of the function you need.

--E 24

)spool

```
)lisp (bye)
```

— OrderedFreeMonoid.help —

```
=====
OrderedFreeMonoid examples
=====
```

The free monoid on a set S is the monoid of finite products of the form $\text{reduce}(*, [s_i ** n_i])$ where the s_i 's are in S , and the n_i 's are non-negative integers. The multiplication is not commutative. For two elements x and y the relation $x < y$ holds if either $\text{length}(x) < \text{length}(y)$ holds or if these lengths are equal and if x is smaller than y w.r.t. the lexicographical ordering induced by S . This domain inherits implementation from `FreeMonoid`.

```
m1:=(x*y*y*z)$OFMONOID(Symbol)
```

```
m2:=(x*y)$OFMONOID(Symbol)
```

```
lquo(m1,m2)
```

```
m3:=(y*y)$OFMONOID(Symbol)
```

```
div(m1,m2)
```

```
div(m1,m3)
```

```
m4:=(y^3)$OFMONOID(Symbol)
```

```
div(m1,m4)
```

Build the non-commutative algebra $h=k[x,y]$ and then make computations in h using some predefined rules for x and y . For example, giving

```
x*y*x=y*x*y
x*x=a*x+b
y*y=a*y+b
```

where a and b are generic elements of k .

Then reduce the polynomials in x and y according to the previous rules. That is, given

```
(x+y)^2  ( = x^2+x*y+y*x+y^2)
```

should reduce to

```
a*(x+y)+2*b+x*y+y*x
```

We can reduce the clutter of the work by defining macros for the types of the domains we need to create. So first we create those macros.

We create generic elements of k . First we define a macro for the domain of ordered variables (OVAR is OrderedVariableList)

```
C ==> OVAR [a,b]
```

Next we define a macro for the commutative field domain $k = \mathbb{Q}[a,b]$ where \mathbb{Q} is Fraction(Integer) and SMP is SparseMultivariatePolynomials

```
K ==> SMP(FRAC INT,C)
```

Now we need some non-commutative variables so we create a macro for that.

```
V ==> OVAR [x,y]
```

And now we need to define the non-commutative algebra $k=k[x,y]$. We use the domain XDistributedPolynomial (XDPOLY) as a macro.

```
H ==> XDPOLY(V,K)
```

The non-commutative variables are in an Ordered Free Monoid

```
M ==> OFMONOID V
```

We have three rules to apply. So we create a function that examines one term. Substitution rules are applied to words from the monoid over the variables and return polynomials. If any rule matches we construct the substitution, create a new monomial term and return it.

```
subs(w:M):H ==
  -- x*y*x = y*x*y
  n1:=lquo(w,(x::V*y::V*x::V)$M)$M
  n1 case "failed" => monom(w,1)$H
    -- x*x = a*x+b
    n2:=lquo(w,(x::V^2)$M)$M
    n2 case "failed" => monom(w,1)$H
      -- y*y = a*y+b
      n3:=lquo(w,(y::V^2)$M)$M
      n3 case "failed" => monom(w,1)$H
        monom(n3,1)$H * (a::K*y::V+b::K)$M * monom(n3,1)$H
        monom(n2,1)$H * (a::K*x::V+b::K)$H * monom(n2,1)$H
        monom(n1,1)$H * (y::V*x::V*y::V)$H * monom(n1,1)$H
```

We apply these rules to a term, remembering the coefficient

```
newterm(x:Record(k:M,c:K)):H == x.c*subs(x,k)
```

And now we create a function to reconstruct the polynomial term by term.

```
newpoly(t:H):H == reduce(+,map(newterm,listOfTerms(t)))
```

For example,

```
p1:(x::V+y::V)$H^2
newpoly(p1)
p2:=(x::V+y::V)$H^3
pNew:=newpoly(p2)
```

But the rules should be applied more than once so we create a function to iterate the rules until nothing changes.

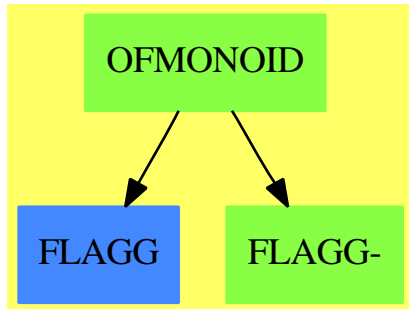
```
reduce(p:H):H ==
  p2 := newpoly(p)
  p3 := newpoly(p2)
  while p3 ~= p2 repeat
    p2 := p3
    p3 := newpoly(p2)
  p3
```

And we can see it work

```
reduce(p2)
```

See Also:

```
o )show OrderedFreeMonoid
```

OrderedFreeMonoid (OFMONOID)**Exports:**

1	coerce	factors	first	hash
hclf	hcrf	latex	length	lexico
lquo	max	min	mirror	nthExpon
nthFactor	one?	overlap	recip	rest
retract	retractIfCan	rquo	sample	size
varList	??	??*	?<?	?<=?
?=?	?>?	?>=?	?^?	?~=?
?div?				

— domain OFMONOID OrderedFreeMonoid —

```

)abbrev domain OFMONOID OrderedFreeMonoid
++ Author: Michel Petitot petitot@lifl.fr
++ Date Created: 91
++ Date Last Updated: 7 Juillet 92
++ Fix History: compilation v 2.1 le 13 dec 98
++ Description:
++ The free monoid on a set \spad{S} is the monoid of finite products of
++ the form \spad{reduce(*,[si ** ni])} where the si's are in S, and the ni's
++ are non-negative integers. The multiplication is not commutative.
++ For two elements \spad{x} and \spad{y} the relation \spad{x < y}
++ holds if either \spad{length(x) < length(y)} holds or if these lengths
++ are equal and if \spad{x} is smaller than \spad{y} w.r.t. the
++ lexicographical ordering induced by \spad{S}.
++ This domain inherits implementation from \spadtype{FreeMonoid}.

OrderedFreeMonoid(S: OrderedSet): OFMcategory == OFMdefinition where
  NNI ==> NonNegativeInteger
  REC ==> Record(gen:S, exp:NNI)
  RESULT ==> Union(Record(lm:Union(%, "failed"), rm:Union(%, "failed")), "failed")

OFMcategory == Join(OrderedMonoid, RetractableTo S) with
  "(": (S, %) -> %
  ++ \spad{s*x} returns the product of \spad{x} by \spad{s} on the left.

```



```

++
++X m1:=(x*y*y*z)$OFMONOID(Symbol)
++X x*m1
"*":    (% , S) -> %
++ \spad{x*s} returns the product of \spad{x} by \spad{s} on the right.
++
++X m1:=(y**3)$OFMONOID(Symbol)
++X m1*x
"**":   (S, NNI) -> %
++ \spad{s**n} returns the product of \spad{s} by itself \spad{n} times.
++
++X m1:=(y**3)$OFMONOID(Symbol)
first: % -> S
++ \spad{first(x)} returns the first letter of \spad{x}.
++
++X m1:=(x*y*y*z)$OFMONOID(Symbol)
++X first m1
rest: % -> %
++ \spad{rest(x)} returns \spad{x} except the first letter.
++
++X m1:=(x*y*y*z)$OFMONOID(Symbol)
++X rest m1
mirror: % -> %
++ \spad{mirror(x)} returns the reversed word of \spad{x}.
++
++X m1:=(x*y*y*z)$OFMONOID(Symbol)
++X mirror m1
lexico: (% ,%) -> Boolean
++ \spad{lexico(x,y)} returns \spad{true}
++ iff \spad{x} is smaller than \spad{y}
++ w.r.t. the pure lexicographical ordering induced by \spad{S}.
++
++X m1:=(x*y*y*z)$OFMONOID(Symbol)
++X m2:=(x*y)$OFMONOID(Symbol)
++X lexico(m1,m2)
++X lexico(m2,m1)
hclf:   (% , %) -> %
++ \spad{hclf(x, y)} returns the highest common left factor
++ of \spad{x} and \spad{y},
++ that is the largest \spad{d} such that \spad{x = d a}
++ and \spad{y = d b}.
++
++X m1:=(x*y*z)$OFMONOID(Symbol)
++X m2:=(x*y)$OFMONOID(Symbol)
++X hclf(m1,m2)
hcrf:   (% , %) -> %
++ \spad{hcrf(x, y)} returns the highest common right
++ factor of \spad{x} and \spad{y},
++ that is the largest \spad{d} such that \spad{x = a d}
++ and \spad{y = b d}.

```

```

++
++X m1:=(x*y*z)$OFMONOID(Symbol)
++X m2:=(y*z)$OFMONOID(Symbol)
++X hcrf(m1,m2)
lquo:  (% , %) -> Union(% , "failed")
++ \spad{lquo(x, y)} returns the exact left quotient of \spad{x}
++ by \spad{y} that is \spad{q} such that \spad{x = y * q},
++ "failed" if \spad{x} is not of the form \spad{y * q}.
++
++X m1:=(x*y*y*z)$OFMONOID(Symbol)
++X m2:=(x*y)$OFMONOID(Symbol)
++X lquo(m1,m2)
rquo:  (% , %) -> Union(% , "failed")
++ \spad{rquo(x, y)} returns the exact right quotient of \spad{x}
++ by \spad{y} that is \spad{q} such that \spad{x = q * y},
++ "failed" if \spad{x} is not of the form \spad{q * y}.
++
++X m1:=(q*y^3)$OFMONOID(Symbol)
++X m2:=(y^2)$OFMONOID(Symbol)
++X lquo(m1,m2)
lquo:  (% , S) -> Union(% , "failed")
++ \spad{lquo(x, s)} returns the exact left quotient of \spad{x}
++ by \spad{s}.
++
++X m1:=(x*y*y*z)$OFMONOID(Symbol)
++X lquo(m1,x)
rquo:  (% , S) -> Union(% , "failed")
++ \spad{rquo(x, s)} returns the exact right quotient
++ of \spad{x} by \spad{s}.
++
++X m1:=(x*y)$OFMONOID(Symbol)
++X div(m1,y)
divide: (% , %) -> RESULT
++ \spad{divide(x,y)} returns the left and right exact quotients of
++ \spad{x} by \spad{y}, that is \spad{[l,r]} such that \spad{x = l*y*r}.
++ "failed" is returned iff \spad{x} is not of the form \spad{l * y * r}.
++
++X m1:=(x*y*y*z)$OFMONOID(Symbol)
++X m2:=(x*y)$OFMONOID(Symbol)
++X divide(m1,m2)
overlap: (% , %) -> Record(lm: % , mm: % , rm: %)
++ \spad{overlap(x, y)} returns \spad{[l, m, r]} such that
++ \spad{x = l * m} and \spad{y = m * r} hold and such that
++ \spad{l} and \spad{r} have no overlap,
++ that is \spad{overlap(l, r) = [l, 1, r]}.
++
++X m1:=(x*y*y*z)$OFMONOID(Symbol)
++X m2:=(x*y)$OFMONOID(Symbol)
++X overlap(m1,m2)
size:  % -> NNI

```

```

++ \spad{size(x)} returns the number of monomials in \spad{x}.
++
++X m1:=(x*y*y*z)$OFMONOID(Symbol)
++X size(m1,2)
nthExpon: (% , Integer) -> NNI
++ \spad{nthExpon(x, n)} returns the exponent of the
++ \spad{n-th} monomial of \spad{x}.
++
++X m1:=(x*y*y*z)$OFMONOID(Symbol)
++X nthExpon(m1,2)
nthFactor: (% , Integer) -> S
++ \spad{nthFactor(x, n)} returns the factor of the \spad{n-th}
++ monomial of \spad{x}.
++
++X m1:=(x*y*y*z)$OFMONOID(Symbol)
++X nthFactor(m1,2)
factors: % -> List REC
++ \spad{factors(a1\^e1,...,an\^en)} returns
++ \spad{[[a1, e1],...,[an, en]]}.
++
++X m1:=(x*y*y*z)$OFMONOID(Symbol)
++X factors m1
length: % -> NNI
++ \spad{length(x)} returns the length of \spad{x}.
++
++X m1:=(x*y*y*z)$OFMONOID(Symbol)
++X length m1
varList: % -> List S
++ \spad{varList(x)} returns the list of variables of \spad{x}.
++
++X m1:=(x*y*y*z)$OFMONOID(Symbol)
++X varList m1

OFMdefinition == FreeMonoid(S) add
Rep := ListMonoidOps(S, NNI, 1)

-- definitions
lquo(w:%, l:S) ==
  x: List REC := listOfMonoms(w)$Rep
  null x      => "failed"
  fx: REC := first x
  fx.gen ^ = 1 => "failed"
  fx.exp = 1  => makeMulti rest(x)
  makeMulti [[fx.gen, (fx.exp - 1)::NNI ]$REC, :rest x]

rquo(w:%, l:S) ==
  u:% := reverse w
  (r := lquo (u,l)) case "failed" => "failed"
  reverse_! (r::%)

```

```

divide(left:%,right:%) ==
  a:=lquo(left,right)
  b:=rquo(left,right)
  [a,b]

length x == reduce("+", [f.exp for f in listOfMonoms x], 0)

varList x ==
  le: List S := [t.gen for t in listOfMonoms x]
  sort_! removeDuplicates(le)

first w ==
  x: List REC := listOfMonoms w
  null x => error "empty word !!!"
  x.first.gen

rest w ==
  x: List REC := listOfMonoms w
  null x => error "empty word !!!"
  fx: REC := first x
  fx.exp = 1 => makeMulti rest x
  makeMulti [[fx.gen, (fx.exp - 1)::NNI]$REC, :rest x]

lexico(a,b) ==      -- ordre lexicographique
  la := listOfMonoms a
  lb := listOfMonoms b
  while (not null la) and (not null lb) repeat
    la.first.gen > lb.first.gen => return false
    la.first.gen < lb.first.gen => return true
    if la.first.exp = lb.first.exp then
      la:=rest la
      lb:=rest lb
    else if la.first.exp > lb.first.exp then
      la:=concat([la.first.gen,
        (la.first.exp - lb.first.exp)::NNI], rest lb)
      lb:=rest lb
    else
      lb:=concat([lb.first.gen,
        (lb.first.exp-la.first.exp)::NNI], rest la)
      la:=rest la
  empty? la and not empty? lb

a < b ==      -- ordre lexicographique par longueur
  la:NNI := length a; lb:NNI := length b
  la = lb => lexico(a,b)
  la < lb

mirror x == reverse(x)$Rep

```

```
"OFMONOID" [color="#88FF44",href="bookvol10.3.pdf#nameddest=OFMONOID"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"OFMONOID" -> "FLAGG"
"OFMONOID" -> "FLAGG-"
```

— OrderedVariableList.input —

```

)set break resume
)sys rm -f OrderedVariableList.output
)spool OrderedVariableList.output
)set message test on
)set message auto off
)clear all
--S 1 of 5
ls:List Symbol:=['x','a','z']
--R
--R
--R (1) [x,a,z]
--R
--R Type: List(Symbol)
--E 1

--S 2 of 5
Z:=OVAR ls
--R
--R
--R (2) OrderedVariableList([x,a,z])
--R
--R Type: Domain
--E 2

--S 3 of 5
size()$Z
--R
--R
--R (3) 3
--R
--R Type: NonNegativeInteger
--E 3

```

```

--S 4 of 5
lv:=[index(i::PI)$Z for i in 1..size()$Z]
--R
--I   Compiling function G1408 with type Integer -> Boolean
--I   Compiling function G1572 with type NonNegativeInteger -> Boolean
--R
--R   (4)  [x,a,z]
--R
--R                                          Type: List(OrderedVariableList([x,a,z]))
--E 4

--S 5 of 5
sorted?(>,lv)
--R
--R
--R   (5)  true
--R
--R                                          Type: Boolean
--E 5
)spool
)lisp (bye)

```

— OrderedVariableList.help —

=====

OrderedVariableList examples

=====

The domain OrderedVariableList provides symbols which are restricted to a particular list and have a definite ordering. Those two features are specified by a List Symbol object that is the argument to the domain.

This is a sample ordering of three symbols.

```

ls:List Symbol:=['x','a','z']
    [x,a,z]
                                     Type: List Symbol

```

Let's build the domain

```

Z:=OVAR ls
    OrderedVariableList [x,a,z]
                                     Type: Domain

```

How many variables does it have?

```

size()$Z

```

```
3
```

```
Type: NonNegativeInteger
```

They are (in the imposed order)

```
lv:=[index(i::PI)$Z for i in 1..size()$Z]
[x,a,z]
```

```
Type: List OrderedVariableList [x,a,z]
```

Check that the ordering is right

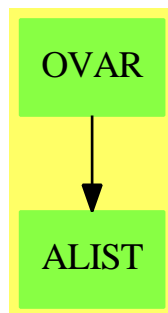
```
sorted?(>,lv)
true
```

```
Type: Boolean
```

See Also:

```
o )show OrderedVariableList
```

OrderedVariableList (OVAR)



Exports:

coerce	convert	hash	index	latex
lookup	max	min	random	size
variable	?~=?	?<?	?<=?	?=?
?>?	?>=?			

— domain OVAR OrderedVariableList —

```
)abbrev domain OVAR OrderedVariableList
++ Author: Mark Botch
++ Description:
++ This domain implements ordered variables
```

```

OrderedVariableList(VariableList:List Symbol):
  Join(OrderedFinite, ConvertibleTo Symbol, ConvertibleTo InputForm,
    ConvertibleTo Pattern Float, ConvertibleTo Pattern Integer) with
    variable: Symbol -> Union(%,"failed")
    ++ variable(s) returns a member of the variable set or failed
== add
  VariableList := removeDuplicates VariableList
  Rep := PositiveInteger
  s1,s2:%
  convert(s1):Symbol == VariableList.((s1::Rep)::PositiveInteger)
  coerce(s1):OutputForm == (convert(s1)@Symbol)::OutputForm
  convert(s1):InputForm == convert(convert(s1)@Symbol)
  convert(s1):Pattern(Integer) == convert(convert(s1)@Symbol)
  convert(s1):Pattern(Float) == convert(convert(s1)@Symbol)
  index i == i::%
  lookup j == j :: Rep
  size () == #VariableList
  variable(exp:Symbol) ==
    for i in 1.. for exp2 in VariableList repeat
      if exp=exp2 then return i::PositiveInteger::%
    "failed"
  s1 < s2 == s2 <$Rep s1
  s1 = s2 == s1 =$Rep s2
  latex(x:%):String == latex(convert(x)@Symbol)

```

— OVAR.dotabb —

```

"OVAR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=OVAR"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"OVAR" -> "ALIST"

```

domain ODPOL OrderlyDifferentialPolynomial

— OrderlyDifferentialPolynomial.input —

```

)set break resume
)sys rm -f OrderlyDifferentialPolynomial.output
)spool OrderlyDifferentialPolynomial.output
)set message test on

```



```

)set message auto off
)clear all
--S 1 of 36
dpol:= ODPOL(FRAC INT)
--R
--R
--R (1) OrderlyDifferentialPolynomial(Fraction(Integer))
--R
--R                                          Type: Domain
--E 1

--S 2 of 36
w := makeVariable('w)$dpol
--R
--R
--R (2) theMap(DPOLCAT-;makeVariable;AM;17!0,0)
--RType: (NonNegativeInteger -> OrderlyDifferentialPolynomial(Fraction(Integer)))
--E 2

--S 3 of 36
z := makeVariable('z)$dpol
--R
--R
--R (3) theMap(DPOLCAT-;makeVariable;AM;17!0,0)
--RType: (NonNegativeInteger -> OrderlyDifferentialPolynomial(Fraction(Integer)))
--E 3

--S 4 of 36
w.5
--R
--R
--R (4) w
--R      5
--R
--R                                          Type: OrderlyDifferentialPolynomial(Fraction(Integer))
--E 4

--S 5 of 36
w 0
--R
--R
--R (5) w
--R
--R                                          Type: OrderlyDifferentialPolynomial(Fraction(Integer))
--E 5

--S 6 of 36
[z.i for i in 1..5]
--R
--R
--R (6) [z ,z ,z ,z ,z ]
--R      1 2 3 4 5
--R
--R                                          Type: List(OrderlyDifferentialPolynomial(Fraction(Integer)))

```

--E 6

--S 7 of 36

f:= w.4 - w.1 * w.1 * z.3

--R

--R

--R

--R (7) $w^2 - w z^3$

--R Type: OrderlyDifferentialPolynomial(Fraction(Integer))

--E 7

--S 8 of 36

g:=(z.1)**3 * (z.2)**2 - w.2

--R

--R

--R

--R (8) $z^3 z^2 - w^2$

--R Type: OrderlyDifferentialPolynomial(Fraction(Integer))

--E 8

--S 9 of 36

D(f)

--R

--R

--R

--R (9) $w^2 - w z^2 - 2w w z^3$

--R Type: OrderlyDifferentialPolynomial(Fraction(Integer))

--E 9

--S 10 of 36

D(f,4)

--R

--R

--R (10)

--R $w^2 - w z^2 - 8w w z^2 + (-12w w - 12w^2)z^2 - 2w z^3 w$

--R +

--R $(-8w w - 24w w)z^2 - 8w z^3 w - 6w z^3$

--R Type: OrderlyDifferentialPolynomial(Fraction(Integer))

--E 10

--S 11 of 36

df:=makeVariable(f)\$dpol

--R

```

--R
--R (11) theMap(DPOLCAT-,makeVariable;AM;17!0,0)
--RType: (NonNegativeInteger -> OrderlyDifferentialPolynomial(Fraction(Integer)))
--E 11

--S 12 of 36
df.4
--R
--R
--R (12)
--R      2      2
--R      w - w z - 8w w z + (- 12w w - 12w )z - 2w z w
--R      8      1 7      1 2 6      1 3      2 5      1 3 5
--R      +
--R      2
--R      (- 8w w - 24w w )z - 8w z w - 6w z
--R      1 4      2 3 4      2 3 4      3 3
--R
--R      Type: OrderlyDifferentialPolynomial(Fraction(Integer))
--E 12

--S 13 of 36
order(g)
--R
--R
--R (13) 2
--R
--R      Type: PositiveInteger
--E 13

--S 14 of 36
order(g, 'w)
--R
--R
--R (14) 2
--R
--R      Type: PositiveInteger
--E 14

--S 15 of 36
differentialVariables(g)
--R
--R
--R (15) [z,w]
--R
--R      Type: List(Symbol)
--E 15

--S 16 of 36
degree(g)
--R
--R
--R      2 3
--R (16) z z

```

```

--R      2 1
--R      Type: IndexedExponents(OrderyDifferentialVariable(Symbol))
--E 16

--S 17 of 36
degree(g, 'w)
--R
--R
--R      (17)  1
--R
--R      Type: PositiveInteger
--E 17

--S 18 of 36
weights(g)
--R
--R
--R      (18)  [7,2]
--R
--R      Type: List(NonNegativeInteger)
--E 18

--S 19 of 36
weights(g,'w)
--R
--R
--R      (19)  [2]
--R
--R      Type: List(NonNegativeInteger)
--E 19

--S 20 of 36
weight(g)
--R
--R
--R      (20)  7
--R
--R      Type: PositiveInteger
--E 20

--S 21 of 36
isobaric?(g)
--R
--R
--R      (21)  false
--R
--R      Type: Boolean
--E 21

--S 22 of 36
eval(g,['w::Symbol],[f])
--R
--R
--R      2      2      3 2
--R      (22)  - w + w z + 4w w z + (2w w + 2w )z + z z

```

```

--R          6      1 5      1 2 4      1 3      2 3      1 2
--R                                     Type: OrderlyDifferentialPolynomial(Fraction(Integer))
--E 22

```

```

--S 23 of 36
eval(g,variables(w.0),[f])
--R
--R
--R          3 2
--R (23)  z z - w
--R      1 2 2
--R                                     Type: OrderlyDifferentialPolynomial(Fraction(Integer))
--E 23

```

```

--S 24 of 36
monomials(g)
--R
--R
--R          3 2
--R (24)  [z z , - w ]
--R      1 2 2
--R                                     Type: List(OrderlyDifferentialPolynomial(Fraction(Integer)))
--E 24

```

```

--S 25 of 36
variables(g)
--R
--R
--R (25)  [z , w , z ]
--R      2 2 1
--R                                     Type: List(OrderlyDifferentialVariable(Symbol))
--E 25

```

```

--S 26 of 36
gcd(f,g)
--R
--R
--R (26)  1
--R                                     Type: OrderlyDifferentialPolynomial(Fraction(Integer))
--E 26

```

```

--S 27 of 36
groebner([f,g])
--R
--R
--R          2      3 2
--R (27)  [w - w z , z z - w ]
--R      4      1 3 1 2 2
--R                                     Type: List(OrderlyDifferentialPolynomial(Fraction(Integer)))
--E 27

```

```

--S 28 of 36
lg:=leader(g)
--R
--R
--R (28)  z
--R      2
--R
--R                                          Type: OrderlyDifferentialVariable(Symbol)
--E 28

--S 29 of 36
sg:=separant(g)
--R
--R
--R      3
--R (29)  2z  z
--R      1  2
--R
--R                                          Type: OrderlyDifferentialPolynomial(Fraction(Integer))
--E 29

--S 30 of 36
ig:=initial(g)
--R
--R
--R      3
--R (30)  z
--R      1
--R
--R                                          Type: OrderlyDifferentialPolynomial(Fraction(Integer))
--E 30

--S 31 of 36
g1:=D g
--R
--R
--R      3      2 3
--R (31)  2z  z z - w + 3z  z
--R      1 2 3 3      1 2
--R
--R                                          Type: OrderlyDifferentialPolynomial(Fraction(Integer))
--E 31

--S 32 of 36
lg1:= leader g1
--R
--R
--R (32)  z
--R      3
--R
--R                                          Type: OrderlyDifferentialVariable(Symbol)
--E 32

--S 33 of 36

```

```

pdf:=D(f, lg1)
--R
--R
--R      2
--R (33)  - w
--R      1
--R
--R                                     Type: OrderlyDifferentialPolynomial(Fraction(Integer))
--E 33

--S 34 of 36
prf:=sg * f- pdf * g1
--R
--R
--R      3      2      2 2 3
--R (34)  2z z w - w w + 3w z z
--R      1 2 4 1 3 1 1 2
--R
--R                                     Type: OrderlyDifferentialPolynomial(Fraction(Integer))
--E 34

--S 35 of 36
lcf:=leadingCoefficient univariate(prf, lg)
--R
--R
--R      2 2
--R (35)  3w z
--R      1 1
--R
--R                                     Type: OrderlyDifferentialPolynomial(Fraction(Integer))
--E 35

--S 36 of 36
ig * prf - lcf * g * lg
--R
--R
--R      6      2 3      2 2
--R (36)  2z z w - w z w + 3w z w z
--R      1 2 4 1 1 3 1 1 2 2
--R
--R                                     Type: OrderlyDifferentialPolynomial(Fraction(Integer))
--E 36
)spool
)lisp (bye)

```

— OrderlyDifferentialPolynomial.help —

```

=====
OrderlyDifferentialPolynomial examples
=====

```

Many systems of differential equations may be transformed to equivalent systems of ordinary differential equations where the equations are expressed polynomially in terms of the unknown functions. In Axiom, the domain constructors `OrderlyDifferentialPolynomial` (abbreviated `ODPOL`) and `SequentialDifferentialPolynomial` (abbreviation `SDPOL`) implement two domains of ordinary differential polynomials over any differential ring. In the simplest case, this differential ring is usually either the ring of integers, or the field of rational numbers. However, Axiom can handle ordinary differential polynomials over a field of rational functions in a single indeterminate.

The two domains `ODPOL` and `SDPOL` are almost identical, the only difference being the choice of a different ranking, which is an ordering of the derivatives of the indeterminates. The first domain uses an orderly ranking, that is, derivatives of higher order are ranked higher, and derivatives of the same order are ranked alphabetically. The second domain uses a sequential ranking, where derivatives are ordered first alphabetically by the differential indeterminates, and then by order. A more general domain constructor, `DifferentialSparseMultivariatePolynomial` (abbreviation `DSMP`) allows both a user-provided list of differential indeterminates as well as a user-defined ranking. We shall illustrate `ODPOL(FRAC INT)`, which constructs a domain of ordinary differential polynomials in an arbitrary number of differential indeterminates with rational numbers as coefficients.

```
dpol:= ODPOL(FRAC INT)
      OrderlyDifferentialPolynomial Fraction Integer
      Type: Domain
```

A differential indeterminate `w` may be viewed as an infinite sequence of algebraic indeterminates, which are the derivatives of `w`. To facilitate referencing these, Axiom provides the operation `makeVariable` to convert an element of type `Symbol` to a map from the natural numbers to the differential polynomial ring.

```
w := makeVariable('w)$dpol
      theMap(DPOLCAT-;makeVariable;AM;17!0,0)
Type: (NonNegativeInteger -> OrderlyDifferentialPolynomial Fraction Integer)

z := makeVariable('z)$dpol
      theMap(DPOLCAT-;makeVariable;AM;17!0,0)
Type: (NonNegativeInteger -> OrderlyDifferentialPolynomial Fraction Integer)
```

The fifth derivative of `w` can be obtained by applying the map `w` to the number 5. Note that the order of differentiation is given as a subscript (except when the order is 0).

```
w.5
w
5
```


Type: OrderlyDifferentialPolynomial Fraction Integer

w 0
w

Type: OrderlyDifferentialPolynomial Fraction Integer

The first five derivatives of z can be generated by a list.

[z.i for i in 1..5]
[z ,z ,z ,z ,z]
1 2 3 4 5

Type: List OrderlyDifferentialPolynomial Fraction Integer

The usual arithmetic can be used to form a differential polynomial from the derivatives.

f:= w.4 - w.1 * w.1 * z.3
2
w - w z
4 1 3

Type: OrderlyDifferentialPolynomial Fraction Integer

g:=(z.1)**3 * (z.2)**2 - w.2
3 2
z z - w
1 2 2

Type: OrderlyDifferentialPolynomial Fraction Integer

The operation D computes the derivative of any differential polynomial.

D(f)
2
w - w z - 2w w z
5 1 4 1 2 3

Type: OrderlyDifferentialPolynomial Fraction Integer

The same operation can compute higher derivatives, like the fourth derivative.

D(f,4)
2 2
w - w z - 8w w z + (- 12w w - 12w)z - 2w z w
8 1 7 1 2 6 1 3 2 5 1 3 5
+
2
(- 8w w - 24w w)z - 8w z w - 6w z
1 4 2 3 4 2 3 4 3 3

Type: OrderlyDifferentialPolynomial Fraction Integer

The operation makeVariable creates a map to facilitate referencing the derivatives of f, similar to the map w.

```
df:=makeVariable(f)$dpol
  theMap(DPOLCAT-;makeVariable;AM;17!0,0)
Type: (NonNegativeInteger -> OrderlyDifferentialPolynomial Fraction Integer)
```

The fourth derivative of f may be referenced easily.

```
df.4
      2
      w  - w  z  - 8w w z  + (- 12w w  - 12w  )z  - 2w z w
      8    1  7    1 2 6    1 3    2    5    1 3 5
+
      2
      (- 8w w  - 24w w )z  - 8w z w  - 6w  z
      1 4    2 3 4    2 3 4    3 3
Type: OrderlyDifferentialPolynomial Fraction Integer
```

The operation order returns the order of a differential polynomial, or the order in a specified differential indeterminate.

```
order(g)
2
Type: PositiveInteger
```

```
order(g, 'w)
2
Type: PositiveInteger
```

The operation differentialVariables returns a list of differential indeterminates occurring in a differential polynomial.

```
differentialVariables(g)
[z,w]
Type: List Symbol
```

The operation degree returns the degree, or the degree in the differential indeterminate specified.

```
degree(g)
  2 3
  z z
  2 1
Type: IndexedExponents OrderlyDifferentialVariable Symbol
```

```
degree(g, 'w)
1
Type: PositiveInteger
```

The operation weights returns a list of weights of differential monomials appearing in differential polynomial, or a list of weights in a specified

differential indeterminate.

```
weights(g)
[7,2]
Type: List NonNegativeInteger
```

```
weights(g,'w')
[2]
Type: List NonNegativeInteger
```

The operation `weight` returns the maximum weight of all differential monomials appearing in the differential polynomial.

```
weight(g)
7
Type: PositiveInteger
```

A differential polynomial is isobaric if the weights of all differential monomials appearing in it are equal.

```
isobaric?(g)
false
Type: Boolean
```

To substitute differentially, use `eval`. Note that we must coerce `'w` to `Symbol`, since in `ODPOL`, differential indeterminates belong to the domain `Symbol`. Compare this result to the next, which substitutes algebraically (no substitution is done since `w.0` does not appear in `g`).

```
eval(g,['w::Symbol],[f])
      2          2          3 2
- w + w z + 4w w z + (2w w + 2w )z + z z
  6   1 5   1 2 4   1 3   2 3   1 2
Type: OrderlyDifferentialPolynomial Fraction Integer
```

```
eval(g,variables(w.0),[f])
      3 2
z z - w
1 2   2
Type: OrderlyDifferentialPolynomial Fraction Integer
```

Since `OrderlyDifferentialPolynomial` belongs to `PolynomialCategory`, all the operations defined in the latter category, or in packages for the latter category, are available.

```
monomials(g)
      3 2
[z z , - w ]
  1 2   2
Type: List OrderlyDifferentialPolynomial Fraction Integer
```

```

variables(g)
  [z ,w ,z ]
    2 2 1
Type: List OrderlyDifferentialVariable Symbol

gcd(f,g)
  1
Type: OrderlyDifferentialPolynomial Fraction Integer

groebner([f,g])
      2      3 2
[w - w z ,z z - w ]
  4      1 3 1 2      2
Type: List OrderlyDifferentialPolynomial Fraction Integer

```

The next three operations are essential for elimination procedures in differential polynomial rings. The operation leader returns the leader of a differential polynomial, which is the highest ranked derivative of the differential indeterminates that occurs.

```

lg:=leader(g)
  z
  2
Type: OrderlyDifferentialVariable Symbol

```

The operation separant returns the separant of a differential polynomial, which is the partial derivative with respect to the leader.

```

sg:=separant(g)
  3
  2z z
  1 2
Type: OrderlyDifferentialPolynomial Fraction Integer

```

The operation initial returns the initial, which is the leading coefficient when the given differential polynomial is expressed as a polynomial in the leader.

```

ig:=initial(g)
  3
  z
  1
Type: OrderlyDifferentialPolynomial Fraction Integer

```

Using these three operations, it is possible to reduce f modulo the differential ideal generated by g . The general scheme is to first reduce the order, then reduce the degree in the leader. First, eliminate $z.3$ using the derivative of g .

```

g1 := D g
      3          2 3
    2z z z - w + 3z z
      1 2 3      1 2
Type: OrderlyDifferentialPolynomial Fraction Integer

```

Find its leader.

```

lg1:= leader g1
      z
      3
Type: OrderlyDifferentialVariable Symbol

```

Differentiate f partially with respect to this leader.

```

pdf:=D(f, lg1)
      2
    - w
      1
Type: OrderlyDifferentialPolynomial Fraction Integer

```

Compute the partial remainder of f with respect to g.

```

prf:=sg * f- pdf * g1
      3          2          2 2 3
    2z z w - w w + 3w z z
      1 2 4      1 3      1 1 2
Type: OrderlyDifferentialPolynomial Fraction Integer

```

Note that high powers of lg still appear in prf. Compute the leading coefficient of prf as a polynomial in the leader of g.

```

lcf:=leadingCoefficient univariate(prf, lg)
      2 2
    3w z
      1 1
Type: OrderlyDifferentialPolynomial Fraction Integer

```

Finally, continue eliminating the high powers of lg appearing in prf to obtain the (pseudo) remainder of f modulo g and its derivatives.

```

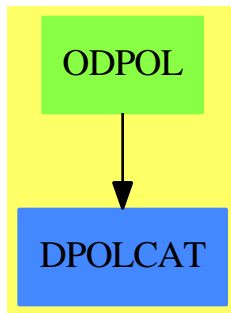
ig * prf - lcf * g * lg
      6          2 3          2 2
    2z z w - w z w + 3w z w z
      1 2 4      1 1 3      1 1 2 2
Type: OrderlyDifferentialPolynomial Fraction Integer

```

See Also:

```
o )show OrderlyDifferentialPolynomial
```

OrderlyDifferentialPolynomial (ODPOL)



See

- ⇒ “OrderlyDifferentialVariable” (ODVAR) [16](#) on page [1855](#)
- ⇒ “SequentialDifferentialVariable” (SDVAR) [20](#) on page [2389](#)
- ⇒ “DifferentialSparseMultivariatePolynomial” (DSMP) [5](#) on page [541](#)
- ⇒ “SequentialDifferentialPolynomial” (SDPOL) [20](#) on page [2385](#)

Exports:

0	1	associates?
binomThmExpt	characteristic	charthRoot
coefficient	coefficients	coerce
conditionP	content	D
degree	differentialVariables	differentiate
discriminant	eval	exquo
factor	factorPolynomial	factorSquareFreePolynomial
gcd	gcdPolynomial	ground
ground?	hash	initial
isExpt	isobaric?	isPlus
isTimes	latex	lcm
leader	leadingCoefficient	leadingMonomial
mainVariable	map	mapExponents
max	min	minimumDegree
monicDivide	monomial	monomial?
monomials	multivariate	numberOfMonomials
one?	order	patternMatch
pomopo!	prime?	primitiveMonomials
primitivePart	recip	reducedSystem
reductum	resultant	retract
retractIfCan	sample	separant
solveLinearPolynomialEquation	squareFree	squareFreePart
squareFreePolynomial	subtractIfCan	totalDegree
totalDegree	unit?	unitCanonical
unitNormal	univariate	univariate
variables	weight	weights
zero?	?*?	?**?
?+?	?-?	-?
?=?	?^?	?~=?
?/?	?<?	?<=?
?>?	?>=?	

— domain ODPOL OrderlyDifferentialPolynomial —

```

)abbrev domain ODPOL OrderlyDifferentialPolynomial
++ Author: William Sit
++ Date Created: 24 September, 1991
++ Date Last Updated: 7 February, 1992
++ References: Kolchin, E.R. "Differential Algebra and Algebraic Groups"
++ (Academic Press, 1973).
++ Description:
++ \spadtype{OrderlyDifferentialPolynomial} implements
++ an ordinary differential polynomial ring in arbitrary number
++ of differential indeterminates, with coefficients in a
++ ring. The ranking on the differential indeterminate is orderly.
++ This is analogous to the domain \spadtype{Polynomial}.

```

```

OrderlyDifferentialPolynomial(R):
  Exports == Implementation where
  R: Ring
  S ==> Symbol
  V ==> OrderlyDifferentialVariable S
  E ==> IndexedExponents(V)
  SMP ==> SparseMultivariatePolynomial(R, S)
  Exports ==> Join(DifferentialPolynomialCategory(R,S,V,E),
                   RetractableTo SMP)

Implementation ==> DifferentialSparseMultivariatePolynomial(R,S,V)

```

— ODPOL.dotabb —

```

"ODPOL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ODPOL"]
"DPOLCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DPOLCAT"]
"ODPOL" -> "DPOLCAT"

```

domain ODVAR OrderlyDifferentialVariable

— OrderlyDifferentialVariable.input —

```

)set break resume
)sys rm -f OrderlyDifferentialVariable.output
)spool OrderlyDifferentialVariable.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OrderlyDifferentialVariable
--R
--R OrderlyDifferentialVariable(S: OrderedSet) is a domain constructor
--R Abbreviation for OrderlyDifferentialVariable is ODVAR
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ODVAR
--R
--R----- Operations -----
--R ?<? : (%,% ) -> Boolean          ?<=? : (%,% ) -> Boolean
--R ?=? : (%,% ) -> Boolean          ?>? : (%,% ) -> Boolean

```



```

--R ?>=? : (% , %) -> Boolean
--R coerce : % -> OutputForm
--R hash : % -> SingleInteger
--R max : (% , %) -> %
--R order : % -> NonNegativeInteger
--R variable : % -> S
--R ?^=? : (% , %) -> Boolean
--R differentiate : (% , NonNegativeInteger) -> %
--R makeVariable : (S , NonNegativeInteger) -> %
--R retractIfCan : % -> Union(S , "failed")
--R
--E 1

)spool
)lisp (bye)

```

— OrderlyDifferentialVariable.help —

```

=====
OrderlyDifferentialVariable examples
=====

```

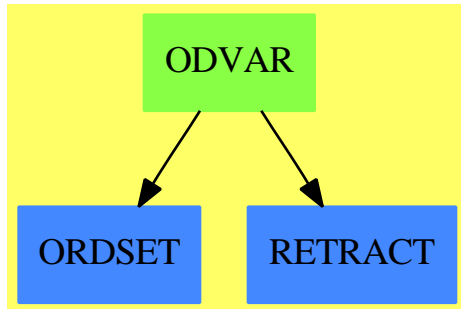
OrderlyDifferentialVariable adds a commonly used orderly ranking to the set of derivatives of an ordered list of differential indeterminates. An orderly ranking is a ranking $<$ of the derivatives with the property that for two derivatives u and v , $u < v$ if the order of u is less than that of v .

This domain belongs to DifferentialVariableCategory. It defines weight to be just order, and it defines an orderly ranking $<$ on derivatives u via the lexicographic order on the pair $(\text{order}(u), \text{variable}(u))$.

See Also:

o)show OrderlyDifferentialVariable

OrderlyDifferentialVariable (ODVAR)



See

- ⇒ “SequentialDifferentialVariable” (SDVAR) [20](#) on page [2389](#)
- ⇒ “DifferentialSparseMultivariatePolynomial” (DSMP) [5](#) on page [541](#)
- ⇒ “OrderlyDifferentialPolynomial” (ODPOL) [16](#) on page [1851](#)
- ⇒ “SequentialDifferentialPolynomial” (SDPOL) [20](#) on page [2385](#)

Exports:

coerce	differentiate	hash	latex	makeVariable
max	min	order	retract	retractIfCan
variable	weight	?~=?	?<?	?<=?
?=?	?>?	?>=?		

— domain ODVAR OrderlyDifferentialVariable —

```

)abbrev domain ODVAR OrderlyDifferentialVariable
++ Author: William Sit
++ Date Created: 19 July 1990
++ Date Last Updated: 13 September 1991
++ References: Kolchin, E.R. "Differential Algebra and Algebraic Groups"
++ (Academic Press, 1973).
++ Description:
++ \spadtype{OrderlyDifferentialVariable} adds a commonly used orderly
++ ranking to the set of derivatives of an ordered list of differential
++ indeterminates. An orderly ranking is a ranking \spadfun{<} of the
++ derivatives with the property that for two derivatives u and v,
++ u \spadfun{<} v if the \spadfun{order} of u is less than that of v.
++ This domain belongs to \spadtype{DifferentialVariableCategory}. It
++ defines \spadfun{weight} to be just \spadfun{order}, and it
++ defines an orderly ranking \spadfun{<} on derivatives u via the
++ lexicographic order on the pair
++ (\spadfun{order}(u), \spadfun{variable}(u)).

OrderlyDifferentialVariable(S:OrderedSet):DifferentialVariableCategory(S)
== add
  Rep := Record(var:S, ord:NonNegativeInteger)
  makeVariable(s,n) == [s, n]
  
```

```
variable v      == v.var
order v        == v.ord
```

— ODVAR.dotabb —

```
"ODVAR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ODVAR"]
"ORDSET" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ORDSET"]
"RETRACT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RETRACT"]
"ODVAR" -> "ORDSET"
"ODVAR" -> "RETRACT"
```

domain ODR OrdinaryDifferentialRing

— OrdinaryDifferentialRing.input —

```
)set break resume
)sys rm -f OrdinaryDifferentialRing.output
)spool OrdinaryDifferentialRing.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OrdinaryDifferentialRing
--R
--R OrdinaryDifferentialRing(Kernels: SetCategory,R: PartialDifferentialRing(Kernels),var: Kerne
--R Abbreviation for OrdinaryDifferentialRing is ODR
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ODR
--R
--R----- Operations -----
--R ?? : (%,%) -> %                ?? : (%,%) -> %
--R ?? : (Integer,%) -> %          ?? : (PositiveInteger,%) -> %
--R ??? : (%,PositiveInteger) -> % ?+? : (%,%) -> %
--R ?-? : (%,%) -> %              -? : % -> %
--R ?? : (%,%) -> Boolean          D : % -> %
--R D : (%,NonNegativeInteger) -> % 1 : () -> %
--R 0 : () -> %                    ?? : (%,PositiveInteger) -> %
--R coerce : % -> R                coerce : R -> %
--R coerce : Integer -> %          coerce : % -> OutputForm
```

```

--R differentiate : % -> %                                hash : % -> SingleInteger
--R inv : % -> % if R has FIELD                            latex : % -> String
--R one? : % -> Boolean                                    recip : % -> Union(%,"failed")
--R sample : () -> %                                       zero? : % -> Boolean
--R ?~=? : (%,% ) -> Boolean
--R ?*? : (% ,Fraction(Integer)) -> % if R has FIELD
--R ?*? : (Fraction(Integer),%) -> % if R has FIELD
--R ?*? : (NonNegativeInteger,%) -> %
--R ?**? : (% ,Integer) -> % if R has FIELD
--R ?**? : (% ,NonNegativeInteger) -> %
--R ?/? : (%,% ) -> % if R has FIELD
--R ?^? : (% ,Integer) -> % if R has FIELD
--R ?^? : (% ,NonNegativeInteger) -> %
--R associates? : (%,% ) -> Boolean if R has FIELD
--R characteristic : () -> NonNegativeInteger
--R coerce : % -> % if R has FIELD
--R coerce : Fraction(Integer) -> % if R has FIELD
--R differentiate : (% ,NonNegativeInteger) -> %
--R divide : (%,% ) -> Record(quotient: %,remainder: %) if R has FIELD
--R euclideanSize : % -> NonNegativeInteger if R has FIELD
--R expressIdealMember : (List(%),%) -> Union(List(%),"failed") if R has FIELD
--R exquo : (%,% ) -> Union(%,"failed") if R has FIELD
--R extendedEuclidean : (%,% ) -> Record(coef1: %,coef2: %,generator: %) if R has FIELD
--R extendedEuclidean : (%,% ,%) -> Union(Record(coef1: %,coef2: %),"failed") if R has FIELD
--R factor : % -> Factored(%) if R has FIELD
--R gcd : (%,% ) -> % if R has FIELD
--R gcd : List(%) -> % if R has FIELD
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R lcm : (%,% ) -> % if R has FIELD
--R lcm : List(%) -> % if R has FIELD
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed") if R has FIELD
--R prime? : % -> Boolean if R has FIELD
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %) if R has FIELD
--R ?quo? : (%,% ) -> % if R has FIELD
--R ?rem? : (%,% ) -> % if R has FIELD
--R sizeLess? : (%,% ) -> Boolean if R has FIELD
--R squareFree : % -> Factored(%) if R has FIELD
--R squareFreePart : % -> % if R has FIELD
--R subtractIfCan : (%,% ) -> Union(%,"failed")
--R unit? : % -> Boolean if R has FIELD
--R unitCanonical : % -> % if R has FIELD
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %) if R has FIELD
--R
--E 1

)spool
)lisp (bye)

```

— OrdinaryDifferentialRing.help —

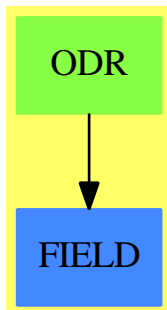
```
=====
OrdinaryDifferentialRing examples
=====
```

This constructor produces an ordinary differential ring from a partial differential ring by specifying a variable.

See Also:

o)show OrdinaryDifferentialRing

OrdinaryDifferentialRing (ODR)



See

⇒ “OppositeMonogenicLinearOperator” (OMLO) [16](#) on page [1806](#)

⇒ “DirectProductModule” (DPMO) [5](#) on page [557](#)

⇒ “DirectProductMatrixModule” (DPMM) [5](#) on page [552](#)

Exports:

0	1	associates?	characteristic
coerce	D	differentiate	divide
euclideanSize	expressIdealMember	exquo	extendedEuclidean
factor	gcd	gcdPolynomial	hash
inv	latex	lcm	multiEuclidean
one?	prime?	principalIdeal	recip
sample	sizeLess?	squareFree	squareFreePart
subtractIfCan	unit?	unitCanonical	unitNormal
zero?	?*?	?**?	?+?
?-?	-?	?=?	?^?
?~=?	?/?	?quo?	?rem?

— domain ODR OrdinaryDifferentialRing —

```

)abbrev domain ODR OrdinaryDifferentialRing
++ Author: Stephen M. Watt
++ Date Created: 1986
++ Date Last Updated: June 3, 1991
++ Description:
++ This constructor produces an ordinary differential ring from
++ a partial differential ring by specifying a variable.

OrdinaryDifferentialRing(Kernels,R,var): DRcategory == DRcapsule where
  Kernels:SetCategory
  R: PartialDifferentialRing(Kernels)
  var : Kernels
  DRcategory == Join(BiModule($,$), DifferentialRing) with
    if R has Field then Field
    coerce: R -> $
      ++ coerce(r) views r as a value in the ordinary differential ring.
    coerce: $ -> R
      ++ coerce(p) views p as a value in the partial differential ring.
  DRcapsule == R add
    n: Integer
    Rep := R
    coerce(u:R):$ == u::Rep::$
    coerce(p:$):R == p::Rep::R
    differentiate p == differentiate(p, var)

    if R has Field then
      p / q == ((p::R) / $R (q::R))::$
      p ** n == ((p::R) ** $R n)::R
      inv(p) == (inv(p::R)$R)::R

```

— ODR.dotabb —

```

"ODR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ODR"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"ODR" -> "FIELD"

```

domain OWP OrdinaryWeightedPolynomials

— OrdinaryWeightedPolynomials.input —

```

)set break resume
)sys rm -f OrdinaryWeightedPolynomials.output
)spool OrdinaryWeightedPolynomials.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OrdinaryWeightedPolynomials
--R
--R OrdinaryWeightedPolynomials(R: Ring,v1: List(Symbol),w1: List(NonNegativeInteger),wtlevel: N
--R Abbreviation for OrdinaryWeightedPolynomials is OWP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for OWP
--R
--R----- Operations -----
--R ?? : (%,%) -> %                ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %  ??? : (%,PositiveInteger) -> %
--R ?? : (%,%) -> %                ?-? : (%,%) -> %
--R -? : % -> %                    ?=? : (%,%) -> Boolean
--R 1 : () -> %                    0 : () -> %
--R ?? : (%,PositiveInteger) -> %  coerce : Polynomial(R) -> %
--R coerce : % -> Polynomial(R)    coerce : Integer -> %
--R coerce : % -> OutputForm       hash : % -> SingleInteger
--R latex : % -> String            one? : % -> Boolean
--R recip : % -> Union(%, "failed") sample : () -> %
--R zero? : % -> Boolean           ?~=? : (%,%) -> Boolean
--R ?? : (%,R) -> % if R has COMRING
--R ?? : (R,%) -> % if R has COMRING
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R ?? : (%,%) -> Union(%, "failed") if R has FIELD
--R ?^? : (%,NonNegativeInteger) -> %
--R changeWeightLevel : NonNegativeInteger -> Void
--R characteristic : () -> NonNegativeInteger
--R coerce : R -> % if R has COMRING
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)

```

— OrdinaryWeightedPolynomials.help —

```
=====
```

OrdinaryWeightedPolynomials examples

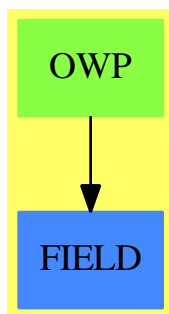
```
=====
```

This domain represents truncated weighted polynomials over the "Polynomial" type. The variables must be specified, as must the weights. The representation is sparse in the sense that only non-zero terms are represented.

See Also:

o)show OrdinaryWeightedPolynomials

OrdinaryWeightedPolynomials (OWP)



Exports:

0	1	changeWeightLevel	characteristic	coerce
hash	latex	one?	recip	sample
subtractIfCan	zero?	?*?	?**?	?+?
?-?	-?	?=?	?^?	?~=?
?/?				

— domain OWP OrdinaryWeightedPolynomials —

```

)abbrev domain OWP OrdinaryWeightedPolynomials
++ Author: James Davenport
++ Date Created: 17 April 1992
++ Date Last Updated: 12 July 1992
++ Description:
++ This domain represents truncated weighted polynomials over the
++ "Polynomial" type. The variables must be
++ specified, as must the weights.
++ The representation is sparse
++ in the sense that only non-zero terms are represented.
  
```



```

OrdinaryWeightedPolynomials(R:Ring,
                             vl:List Symbol, wl:List NonNegativeInteger,
                             wtlevel:NonNegativeInteger):

  Ring with
    if R has CommutativeRing then Algebra(R)
    coerce: $ -> Polynomial(R)
    ++ coerce(p) converts back into a Polynomial(R), ignoring weights
    coerce: Polynomial(R) -> $
    ++ coerce(p) coerces a Polynomial(R) into Weighted form,
    ++ applying weights and ignoring terms
    if R has Field then "/": ($,$) -> Union($,"failed")
    ++ x/y division (only works if minimum weight
    ++ of divisor is zero, and if R is a Field)
    changeWeightLevel: NonNegativeInteger -> Void
    ++ changeWeightLevel(n) This changes the weight level to the
    ++ new value given:
    ++ NB: previously calculated terms are not affected
  == WeightedPolynomials(R,Symbol,IndexedExponents(Symbol),
                        Polynomial(R),
                        vl,wl,wtlevel)

```

— OWP.dotabb —

```

"OWP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=OWP"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"OWP" -> "FIELD"

```

domain OSI OrdSetInts

— OrdSetInts.input —

```

)set break resume
)sys rm -f OrdSetInts.output
)spool OrdSetInts.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OrdSetInts

```

```

--R OrdSetInts is a domain constructor
--R Abbreviation for OrdSetInts is OSI
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for OSI
--R
--R----- Operations -----
--R ?<? : (%,% ) -> Boolean      ?<=? : (%,% ) -> Boolean
--R ?=? : (%,% ) -> Boolean      ?>? : (%,% ) -> Boolean
--R ?>=? : (%,% ) -> Boolean     coerce : Integer -> %
--R coerce : % -> OutputForm     hash : % -> SingleInteger
--R latex : % -> String          max : (%,% ) -> %
--R min : (%,% ) -> %           value : % -> Integer
--R ?^=? : (%,% ) -> Boolean
--R
--E 1

```

```

)spool
)lisp (bye)

```

— OrdSetInts.help —

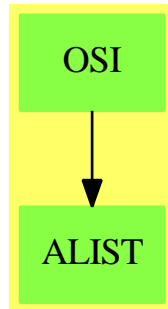
=====

OrdSetInts examples

=====

A domain used in order to take the free R-module on the Integers I.
This is actually the forgetful functor from OrderedRings
to OrderedSets applied to I

See Also:
o)show OrdSetInts

OrdSetInts (OSI)**See**⇒ “Commutator” (COMM) [4](#) on page [411](#)⇒ “FreeNilpotentLie” (FNLA) [7](#) on page [1027](#)**Exports:**

coerce	hash	latex	max	min
value	?~=?	?<?	?<=?	?=?
?>?	?>=?			

— domain OSI OrdSetInts —

```

)abbrev domain OSI OrdSetInts
++ Author: Larry Lambe
++ Date created: 14 August 1988
++ Date Last Updated: 11 March 1991
++ Description:
++ A domain used in order to take the free R-module on the
++ Integers I. This is actually the forgetful functor from OrderedRings
++ to OrderedSets applied to I

```

OrdSetInts: Export == Implement where

```

I ==> Integer
L ==> List
O ==> OutputForm

```

Export == OrderedSet with

```

coerce : Integer -> %
  ++ coerce(i) returns the element corresponding to i
value   : % -> I
  ++ value(x) returns the integer associated with x

```

Implement == add

```

Rep := Integer
x,y: %

```

```

x = y == x =$Rep y

```

```

x < y == x <$Rep y

coerce(i:Integer):% == i

value(x) == x:Rep

coerce(x):0 ==
  sub(e::Symbol::0, coerce(x)$Rep)$0

```

— OSI.dotabb —

```

"OSI" [color="#88FF44",href="bookvol10.3.pdf#nameddest=OSI"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"OSI" -> "ALIST"

```

domain OUTFORM OutputForm

— OutputForm.input —

```

)set break resume
)sys rm -f OutputForm.output
)spool OutputForm.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OutputForm
--R
--R OutputForm is a domain constructor
--R Abbreviation for OutputForm is OUTFORM
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for OUTFORM
--R
--R----- Operations -----
--R ??? : (%,%) -> %          ??? : (%,%) -> %
--R ?+? : (%,%) -> %          -? : % -> %
--R ?-? : (%,%) -> %          ?/? : (%,%) -> %
--R ?<? : (%,%) -> %          ?<=? : (%,%) -> %
--R ?=? : (%,%) -> %          ?=? : (%,%) -> Boolean

```

```

--R ?>? : (% , %) -> %
--R ?SEGMENT : % -> %
--R ?^=? : (% , %) -> %
--R assign : (% , %) -> %
--R blankSeparate : List(%) -> %
--R brace : List(%) -> %
--R bracket : List(%) -> %
--R center : % -> %
--R coerce : % -> OutputForm
--R ?div? : (% , %) -> %
--R ?.? : (% , List(%)) -> %
--R exquo : (% , %) -> %
--R hconcat : List(%) -> %
--R height : () -> Integer
--R hspace : Integer -> %
--R infix : (% , List(%)) -> %
--R int : (% , % , %) -> %
--R int : % -> %
--R latex : % -> String
--R left : (% , Integer) -> %
--R message : String -> %
--R not? : % -> %
--R outputForm : DoubleFloat -> %
--R outputForm : Symbol -> %
--R over : (% , %) -> %
--R overlable : (% , %) -> %
--R paren : % -> %
--R postfix : (% , %) -> %
--R presub : (% , %) -> %
--R prime : % -> %
--R prod : (% , % , %) -> %
--R prod : % -> %
--R quote : % -> %
--R ?rem? : (% , %) -> %
--R right : (% , Integer) -> %
--R root : % -> %
--R scripts : (% , List(%)) -> %
--R slash : (% , %) -> %
--R sub : (% , %) -> %
--R sum : (% , % , %) -> %
--R sum : % -> %
--R superHeight : % -> Integer
--R vconcat : List(%) -> %
--R vspace : Integer -> %
--R width : % -> Integer
--R ?~=? : (% , %) -> Boolean
--R differentiate : (% , NonNegativeInteger) -> %
--R dot : (% , NonNegativeInteger) -> %
--R prime : (% , NonNegativeInteger) -> %
--R
?>=? : (% , %) -> %
?..? : (% , %) -> %
?and? : (% , %) -> %
binomial : (% , %) -> %
box : % -> %
brace : % -> %
bracket : % -> %
center : (% , Integer) -> %
commaSeparate : List(%) -> %
dot : % -> %
empty : () -> %
hash : % -> SingleInteger
hconcat : (% , %) -> %
height : % -> Integer
infix : (% , % , %) -> %
infix? : % -> Boolean
int : (% , %) -> %
label : (% , %) -> %
left : % -> %
matrix : List(List(%)) -> %
messagePrint : String -> Void
?or? : (% , %) -> %
outputForm : String -> %
outputForm : Integer -> %
overbar : % -> %
paren : List(%) -> %
pile : List(%) -> %
prefix : (% , List(%)) -> %
presuper : (% , %) -> %
print : % -> Void
prod : (% , %) -> %
?quo? : (% , %) -> %
rarrow : (% , %) -> %
right : % -> %
root : (% , %) -> %
rspace : (Integer , Integer) -> %
semicolonSeparate : List(%) -> %
string : % -> %
subHeight : % -> Integer
sum : (% , %) -> %
super : (% , %) -> %
supersub : (% , List(%)) -> %
vconcat : (% , %) -> %
width : () -> Integer
zag : (% , %) -> %

```

--E 1

)spool
)lisp (bye)

— **OutputForm.help** —

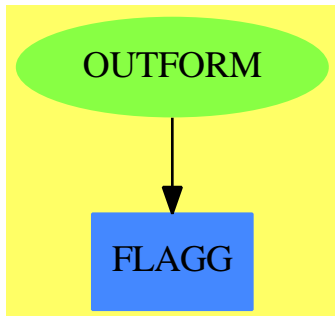
=====
OutputForm examples
=====

This domain is used to create and manipulate mathematical expressions for output. It is intended to provide an insulating layer between the expression rendering software (e.g.FORTRAN, TeX, or Script) and the output coercions in the various domains.

See Also:

o)show OutputForm

OutputForm (OUTFORM)



Exports:

assign	binomial	blankSeparate	box	brace
bracket	center	coerce	commaSeparate	differentiate
dot	empty	exquo	hash	hconcat
height	hspace	infix	infix?	int
label	latex	left	matrix	message
messagePrint	not?	outputForm	over	overbar
overlabel	paren	pile	postfix	prefix
presub	presuper	prime	print	prod
quote	rarrow	right	root	rspace
scripts	semicolonSeparate	slash	string	sub
subHeight	sum	super	superHeight	supersub
vconcat	vspace	width	zag	?*?
?**?	?+?	-?	?-?	?/?
?<?	?<=?	?=?	?>?	?>=?
?SEGMENT	?..?	?^=?	?and?	?div?
?..?	?or?	?quo?	?rem?	?~=?

— domain **OUTFORM** **OutputForm** —

```

)abbrev domain OUTFORM OutputForm
++ Author: SMW March/88
++ Description:
++ This domain is used to create and manipulate mathematical expressions
++ for output. It is intended to provide an insulating layer between
++ the expression rendering software (e.g.FORTRAN, TeX, or Script) and
++ the output coercions in the various domains.

OutputForm(): SetCategory with
  --% Printing
  print : $ -> Void
    ++ print(u) prints the form u.
  message: String -> $
    ++ message(s) creates an form with no string quotes
    ++ from string s.
  messagePrint: String -> Void
    ++ messagePrint(s) prints s without string quotes. Note:
    ++ \spad{messagePrint(s)} is equivalent to \spad{print message(s)}.
  --% Creation of atomic forms
  outputForm: Integer -> $
    ++ outputForm(n) creates an form for integer n.
  outputForm: Symbol -> $
    ++ outputForm(s) creates an form for symbol s.
  outputForm: String -> $
    ++ outputForm(s) creates an form for string s.
  outputForm: DoubleFloat -> $
    ++ outputForm(sf) creates an form for small float sf.
  empty : () -> $
    ++ empty() creates an empty form.

```

```

--% Sizings
width: $ -> Integer
    ++ width(f) returns the width of form f (an integer).
height: $ -> Integer
    ++ height(f) returns the height of form f (an integer).
width: -> Integer
    ++ width() returns the width of the display area (an integer).
height: -> Integer
    ++ height() returns the height of the display area (an integer).
subHeight: $ -> Integer
    ++ subHeight(f) returns the height of form f below the base line.
superHeight: $ -> Integer
    ++ superHeight(f) returns the height of form f above the base line.
--% Space manipulations
hspace: Integer -> $ ++ hspace(n) creates white space of width n.
vspace: Integer -> $ ++ vspace(n) creates white space of height n.
rspace: (Integer,Integer) -> $
    ++ rspace(n,m) creates rectangular white space, n wide by m high.
--% Area adjustments
left: ($,Integer) -> $
    ++ left(f,n) left-justifies form f within space of width n.
right: ($,Integer) -> $
    ++ right(f,n) right-justifies form f within space of width n.
center: ($,Integer) -> $
    ++ center(f,n) centers form f within space of width n.
left: $ -> $
    ++ left(f) left-justifies form f in total space.
right: $ -> $
    ++ right(f) right-justifies form f in total space.
center: $ -> $
    ++ center(f) centers form f in total space.

--% Area manipulations
hconcat: ($,$) -> $
    ++ hconcat(f,g) horizontally concatenate forms f and g.
vconcat: ($,$) -> $
    ++ vconcat(f,g) vertically concatenates forms f and g.
hconcat: List $ -> $
    ++ hconcat(u) horizontally concatenates all forms in list u.
vconcat: List $ -> $
    ++ vconcat(u) vertically concatenates all forms in list u.

--% Application formers
prefix: ($, List $) -> $
    ++ prefix(f,l) creates a form depicting the n-ary prefix
    ++ application of f to a tuple of arguments given by list l.
infix: ($, List $) -> $
    ++ infix(f,l) creates a form depicting the n-ary application
    ++ of infix operation f to a tuple of arguments l.
infix: ($, $, $) -> $

```



```

++ infix(op, a, b) creates a form which prints as: a op b.
postfix: ($, $) -> $
++ postfix(op, a) creates a form which prints as: a op.
infix?: $ -> Boolean
++ infix?(op) returns true if op is an infix operator,
++ and false otherwise.
elt: ($, List $) -> $
++ elt(op,l) creates a form for application of op
++ to list of arguments l.

--% Special forms
string: $ -> $
++ string(f) creates f with string quotes.
label: ($, $) -> $
++ label(n,f) gives form f an equation label n.
box: $ -> $
++ box(f) encloses f in a box.
matrix: List List $ -> $
++ matrix(llf) makes llf (a list of lists of forms) into
++ a form which displays as a matrix.
zag: ($, $) -> $
++ zag(f,g) creates a form for the continued fraction form for f over g.
root: $ -> $
++ root(f) creates a form for the square root of form f.
root: ($, $) -> $
++ root(f,n) creates a form for the nth root of form f.
over: ($, $) -> $
++ over(f,g) creates a form for the vertical fraction of f over g.
slash: ($, $) -> $
++ slash(f,g) creates a form for the horizontal fraction of f over g.
assign: ($, $) -> $
++ assign(f,g) creates a form for the assignment \spad{f := g}.
rarrow: ($, $) -> $
++ rarrow(f,g) creates a form for the mapping \spad{f -> g}.
differentiate: ($, NonNegativeInteger) -> $
++ differentiate(f,n) creates a form for the nth derivative of f,
++ e.g. \spad{f'}, \spad{f''}, \spad{f'''},
++ "f super \spad{iv}".
binomial: ($, $) -> $
++ binomial(n,m) creates a form for the binomial coefficient of n and m.

--% Scripts
sub: ($, $) -> $
++ sub(f,n) creates a form for f subscripted by n.
super: ($, $) -> $
++ super(f,n) creates a form for f superscripted by n.
presub: ($, $) -> $
++ presub(f,n) creates a form for f presubscripted by n.
presuper: ($, $) -> $
++ presuper(f,n) creates a form for f presuperscripted by n.

```

```

scripts: ($, List $) -> $
  ++ \spad{scripts}(f, [sub, super, presuper, presub])}
  ++ creates a form for f with scripts on all 4 corners.
supersub: ($, List $) -> $
  ++ supersub(a, [sub1,super1,sub2,super2,...])
  ++ creates a form with each subscript aligned
  ++ under each superscript.

--% Diacritical marks
quote: $ -> $
  ++ quote(f) creates the form f with a prefix quote.
dot: $ -> $
  ++ dot(f) creates the form with a one dot overhead.
dot: ($, NonNegativeInteger) -> $
  ++ dot(f,n) creates the form f with n dots overhead.
prime: $ -> $
  ++ prime(f) creates the form f followed by a suffix prime (single quote).
prime: ($, NonNegativeInteger) -> $
  ++ prime(f,n) creates the form f followed by n primes.
overbar: $ -> $
  ++ overbar(f) creates the form f with an overbar.
overlabel: ($, $) -> $
  ++ overlabel(x,f) creates the form f with "x overbar" over the top.

--% Plexes
sum: ($) -> $
  ++ sum(expr) creates the form prefixing expr by a capital sigma.
sum: ($, $) -> $
  ++ sum(expr,lowerlimit) creates the form prefixing expr by
  ++ a capital sigma with a lowerlimit.
sum: ($, $, $) -> $
  ++ sum(expr,lowerlimit,upperlimit) creates the form prefixing expr by
  ++ a capital sigma with both a lowerlimit and upperlimit.
prod: ($) -> $
  ++ prod(expr) creates the form prefixing expr by a capital pi.
prod: ($, $) -> $
  ++ prod(expr,lowerlimit) creates the form prefixing expr by
  ++ a capital pi with a lowerlimit.
prod: ($, $, $) -> $
  ++ prod(expr,lowerlimit,upperlimit) creates the form prefixing expr by
  ++ a capital pi with both a lowerlimit and upperlimit.
int: ($) -> $
  ++ int(expr) creates the form prefixing expr with an integral sign.
int: ($, $) -> $
  ++ int(expr,lowerlimit) creates the form prefixing expr by an
  ++ integral sign with a lowerlimit.
int: ($, $, $) -> $
  ++ int(expr,lowerlimit,upperlimit) creates the form prefixing expr by
  ++ an integral sign with both a lowerlimit and upperlimit.

```

```

--% Matchfix forms
brace:  $ -> $
  ++ brace(f) creates the form enclosing f in braces (curly brackets).
brace:  List $ -> $
  ++ brace(lf) creates the form separating the elements of lf
  ++ by commas and encloses the result in curly brackets.
bracket: $ -> $
  ++ bracket(f) creates the form enclosing f in square brackets.
bracket: List $ -> $
  ++ bracket(lf) creates the form separating the elements of lf
  ++ by commas and encloses the result in square brackets.
paren:  $ -> $
  ++ paren(f) creates the form enclosing f in parentheses.
paren:  List $ -> $
  ++ paren(lf) creates the form separating the elements of lf
  ++ by commas and encloses the result in parentheses.

--% Separators for aggregates
pile:   List $ -> $
  ++ pile(l) creates the form consisting of the elements of l which
  ++ displays as a pile, i.e. the elements begin on a new line and
  ++ are indented right to the same margin.

commaSeparate: List $ -> $
  ++ commaSeparate(l) creates the form separating the elements of l
  ++ by commas.
semicolonSeparate: List $ -> $
  ++ semicolonSeparate(l) creates the form separating the elements of l
  ++ by semicolons.
blankSeparate: List $ -> $
  ++ blankSeparate(l) creates the form separating the elements of l
  ++ by blanks.

--% Specific applications
"=":    ($, $) -> $
  ++ f = g creates the equivalent infix form.
"^=":   ($, $) -> $
  ++ f ^= g creates the equivalent infix form.
"<":    ($, $) -> $
  ++ f < g creates the equivalent infix form.
">":    ($, $) -> $
  ++ f > g creates the equivalent infix form.
"<=":   ($, $) -> $
  ++ f <= g creates the equivalent infix form.
">=":   ($, $) -> $
  ++ f >= g creates the equivalent infix form.
"+":    ($, $) -> $
  ++ f + g creates the equivalent infix form.
"-":    ($, $) -> $
  ++ f - g creates the equivalent infix form.
"_-":   ($)    -> $

```

```

    ++ - f creates the equivalent prefix form.
    "*":    ($, $) -> $
    ++ f * g creates the equivalent infix form.
    "/":    ($, $) -> $
    ++ f / g creates the equivalent infix form.
    "**":    ($, $) -> $
    ++ f ** g creates the equivalent infix form.
    "div":   ($, $) -> $
    ++ f div g creates the equivalent infix form.
    "rem":   ($, $) -> $
    ++ f rem g creates the equivalent infix form.
    "quo":   ($, $) -> $
    ++ f quo g creates the equivalent infix form.
    "exquo": ($, $) -> $
    ++ exquo(f,g) creates the equivalent infix form.
    "and":   ($, $) -> $
    ++ f and g creates the equivalent infix form.
    "or":    ($, $) -> $
    ++ f or g creates the equivalent infix form.
    "not":   ($)   -> $
    ++ not f creates the equivalent prefix form.
    SEGMENT: ($,$) -> $
    ++ SEGMENT(x,y) creates the infix form: \spad{x..y}.
    SEGMENT: ($)   -> $
    ++ SEGMENT(x) creates the prefix form: \spad{x..}.

== add
import NumberFormats

-- Todo:
--   program forms, greek letters
--   infix, prefix, postfix, matchfix support in OUT BOOT
--   labove rabove, corresponding overs.
--   better super script, overmark, undermark
--   bug in product, paren blankSeparate []
--   uniformize integrals, products, etc as plexes.

cons ==> CONS$Lisp
car  ==> CAR$Lisp
cdr  ==> CDR$Lisp

Rep := List $

a, b: $
l: List $
s: String
e: Symbol
n: Integer
nn:NonNegativeInteger

```

```

sform:    String  -> $
eform:    Symbol  -> $
iform:    Integer -> $

print x          == mathprint(x)$Lisp
message s        == (empty? s => empty(); s pretend $)
messagePrint s   == print message s
(a:$ = b:$):Boolean == EQUAL(a, b)$Lisp
(a:$ = b:$):$     == [sform "=", a, b]
coerce(a):OutputForm == a pretend OutputForm
outputForm n     == n pretend $
outputForm e     == e pretend $
outputForm(f:DoubleFloat) == f pretend $
sform s          == s pretend $
eform e          == e pretend $
iform n          == n pretend $

outputForm s ==
  sform concat(quote())$Character, concat(s, quote())$Character))

width(a) == outformWidth(a)$Lisp
height(a) == height(a)$Lisp
subHeight(a) == subspan(a)$Lisp
superHeight(a) == superspan(a)$Lisp
height() == 20
width() == 66

center(a,w) == hconcat(hspace((w - width(a)) quo 2),a)
left(a,w) == hconcat(a,hspace((w - width(a))))
right(a,w) == hconcat(hspace(w - width(a)),a)
center(a) == center(a,width())
left(a) == left(a,width())
right(a) == right(a,width())

vspace(n) ==
  n = 0 => empty()
  vconcat(sform " ",vspace(n - 1))

hspace(n) ==
  n = 0 => empty()
  sform(fillerSpaces(n)$Lisp)

rspace(n, m) ==
  n = 0 or m = 0 => empty()
  vconcat(hspace n, rspace(n, m - 1))

matrix ll ==
  lv:$ := [LIST2VEC$Lisp l for l in ll]
  CONS(eform MATRIX, LIST2VEC$Lisp lv)$Lisp

```

```

pile l          == cons(eform SC, l)
commaSeparate l == cons(eform AGGLST, l)
semicolonSeparate l == cons(eform AGGSET, l)
blankSeparate l ==
  c:=eform CONCATB
  l1:=$:=[]
  for u in reverse l repeat
    if EQCAR(u,c)$Lisp
      then l1:=[:cdr u,:l1]
      else l1:=[:u,:l1]
  cons(c, l1)

brace a        == [eform BRACE, a]
brace l        == brace commaSeparate l
bracket a      == [eform BRACKET, a]
bracket l      == bracket commaSeparate l
paren a        == [eform PAREN, a]
paren l        == paren commaSeparate l

sub (a,b)      == [eform SUB, a, b]
super (a, b)   == [eform SUPERSUB,a,sform " ",b]
presub(a,b)    == [eform SUPERSUB,a,sform " ",sform " ",sform " ",b]
presuper(a, b) == [eform SUPERSUB,a,sform " ",sform " ",b]
scripts (a, l) ==
  null l => a
  null rest l => sub(a, first l)
  cons(eform SUPERSUB, cons(a, l))
supersub(a, l) ==
  if odd? (#l) then l := append(l, [empty()])
  cons(eform ALTSUPERSUB, cons(a, l))

hconcat(a,b)   == [eform CONCAT, a, b]
hconcat l      == cons(eform CONCAT, l)
vconcat(a,b)   == [eform VCONCAT, a, b]
vconcat l      == cons(eform VCONCAT, l)

a ^= b        == [sform "^=", a, b]
a < b         == [sform "<", a, b]
a > b         == [sform ">", a, b]
a <= b        == [sform "<=", a, b]
a >= b        == [sform ">=", a, b]

a + b         == [sform "+", a, b]
a - b         == [sform "-", a, b]
- a           == [sform "-", a]
a * b         == [sform "*", a, b]
a / b         == [sform "/", a, b]
a ** b        == [sform "**", a, b]
a div b       == [sform "div", a, b]
a rem b       == [sform "rem", a, b]

```

```

a quo b      == [sform "quo",  a, b]
a exquo b    == [sform "exquo", a, b]
a and b      == [sform "and",  a, b]
a or b       == [sform "or",   a, b]
not a        == [sform "not",  a]
SEGMENT(a,b) == [eform SEGMENT, a, b]
SEGMENT(a)   == [eform SEGMENT, a]
binomial(a,b)==[eform BINOMIAL, a, b]

empty() == [eform NOTHING]

infix? a ==
  e:$ :=
    IDENTP$Lisp a => a
    STRINGP$Lisp a => INTERN$Lisp a
    return false
  if GET(e,QUOTE(INFIXOP$Lisp)$Lisp)$Lisp then true else false

elt(a, l) ==
  cons(a, l)
prefix(a,l) ==
  not infix? a => cons(a, l)
  hconcat(a, paren commaSeparate l)
infix(a, l) ==
  null l => empty()
  null rest l => first l
  infix? a => cons(a, l)
  hconcat [first l, a, infix(a, rest l)]
infix(a,b,c) ==
  infix? a => [a, b, c]
  hconcat [b, a, c]
postfix(a, b) ==
  hconcat(b, a)

string a == [eform STRING, a]
quote a == [eform QUOTE, a]
overbar a == [eform OVERBAR, a]
dot a == super(a, sform ".")
prime a == super(a, sform ",")
dot(a,nn) == (s := new(nn, char "."); super(a, sform s))
prime(a,nn) == (s := new(nn, char ","); super(a, sform s))

overlabel(a,b) == [eform OVERLABEL, a, b]
box a == [eform BOX, a]
zag(a,b) == [eform ZAG, a, b]
root a == [eform ROOT, a]
root(a,b) == [eform ROOT, a, b]
over(a,b) == [eform OVER, a, b]
slash(a,b) == [eform SLASH, a, b]
assign(a,b) == [eform LET, a, b]

```

```

label(a,b) == [eform EQUATNUM, a, b]
rarrow(a,b)== [eform TAG, a, b]
differentiate(a, nn)==
  zero? nn => a
  nn < 4 => prime(a, nn)
  r := FormatRoman(nn::PositiveInteger)
  s := lowerCase(r::String)
  super(a, paren sform s)

sum(a)      == [eform SIGMA, empty(), a]
sum(a,b)    == [eform SIGMA, b, a]
sum(a,b,c)  == [eform SIGMA2, b, c, a]
prod(a)     == [eform PI, empty(), a]
prod(a,b)   == [eform PI, b, a]
prod(a,b,c) == [eform PI2, b, c, a]
int(a)      == [eform INTSIGN,empty(), empty(), a]
int(a,b)    == [eform INTSIGN,b, empty(), a]
int(a,b,c)  == [eform INTSIGN,b, c, a]

```

— OUTFORM.dotabb —

```

"OUTFORM" [color="#88FF44",href="bookvol10.3.pdf#nameddest=OUTFORM",
           shape=ellipse]
"FLAGG"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"OUTFORM" -> "FLAGG"

```

Chapter 17

Chapter P

domain PADIC PAdicInteger

— PAdicInteger.input —

```
)set break resume
)sys rm -f PAdicInteger.output
)spool PAdicInteger.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PAdicInteger
--R
--R PAdicInteger(p: Integer) is a domain constructor
--R Abbreviation for PAdicInteger is PADIC
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PADIC
--R
--R----- Operations -----
--R ??? : (%,%) -> %           ??? : (Integer,%) -> %
--R ??? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %           ?-? : (%,%) -> %
--R -? : % -> %               ?=? : (%,%) -> Boolean
--R 1 : () -> %               0 : () -> %
--R ?? : (%,PositiveInteger) -> %   associates? : (%,%) -> Boolean
--R coerce : % -> %             coerce : Integer -> %
--R coerce : % -> OutputForm     complete : % -> %
--R digits : % -> Stream(Integer) extend : (%,Integer) -> %
--R gcd : List(%) -> %          gcd : (%,%) -> %
--R hash : % -> SingleInteger    latex : % -> String
```

```

--R lcm : List(%) -> %
--R moduloP : % -> Integer
--R one? : % -> Boolean
--R ?quo? : (%,%) -> %
--R recip : % -> Union(%, "failed")
--R sample : () -> %
--R sqrt : (%, Integer) -> %
--R unitCanonical : % -> %
--R ?^=? : (%,%) -> Boolean
--R ?*? : (NonNegativeInteger, %) -> %
--R ***? : (%, NonNegativeInteger) -> %
--R ?^? : (%, NonNegativeInteger) -> %
--R approximate : (%, Integer) -> Integer
--R characteristic : () -> NonNegativeInteger
--R divide : (%,%) -> Record(quotient: %, remainder: %)
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(%), %) -> Union(List(%), "failed")
--R exquo : (%,%) -> Union(%, "failed")
--R extendedEuclidean : (%,%,%) -> Union(Record(coef1: %, coef2: %), "failed")
--R extendedEuclidean : (%,%) -> Record(coef1: %, coef2: %, generator: %)
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed")
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %)
--R root : (SparseUnivariatePolynomial(Integer), Integer) -> %
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R unitNormal : % -> Record(unit: %, canonical: %, associate: %)
--R
--E 1

)spool
)lisp (bye)

```

— PAdicInteger.help —

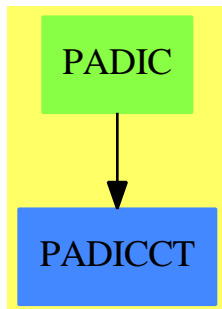
=====

PAdicInteger examples

=====

Stream-based implementation of Zp: p-adic numbers are represented as
 $\text{sum}(i = 0.., a[i] * p^i),$
 where the $a[i]$ lie in $0, 1, \dots, (p - 1).$

See Also:
 o)show PAdicInteger

PAdicInteger (PADIC)**See**

- ⇒ “InnerPAdicInteger” (IPADIC) [10](#) on page [1294](#)
- ⇒ “BalancedPAdicInteger” (BPADIC) [3](#) on page [258](#)
- ⇒ “PAdicRationalConstructor” (PADICRC) [17](#) on page [1890](#)
- ⇒ “PAdicRational” (PADICRAT) [17](#) on page [1885](#)
- ⇒ “BalancedPAdicRational” (BPADICRT) [3](#) on page [262](#)

Exports:

0	1	approximate	associates?
characteristic	coerce	complete	digits
divide	euclideanSize	expressIdealMember	exquo
extend	extendedEuclidean	gcd	gcdPolynomial
hash	latex	lcm	lcm
moduloP	modulus	multiEuclidean	one?
order	principalIdeal	quotientByP	recip
root	sample	sizeLess?	sqrt
subtractIfCan	unit?	unitCanonical	unitNormal
zero?	?*?	?**?	?+?
?-?	-?	?=?	?^?
?~=?	?quo?	?rem?	

— domain PADIC PAdicInteger —

```

)abbrev domain PADIC PAdicInteger
++ Author: Clifton J. Williamson
++ Date Created: 20 August 1989
++ Date Last Updated: 15 May 1990
++ Description:
++ Stream-based implementation of Zp: p-adic numbers are represented as
++ sum(i = 0.., a[i] * p^i), where the a[i] lie in 0,1,...,(p - 1).

```

```

PAdicInteger(p:Integer) == InnerPAdicInteger(p,true$Boolean)

```

— PADIC.dotabb —

```
"PADIC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PADIC"]
"PADICCT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PADICCT"]
"PADIC" -> "PADICCT"
```

domain PADICRAT PAdicRational

— PAdicRational.input —

```
)set break resume
)sys rm -f PAdicRational.output
)spool PAdicRational.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PAdicRational
--R
--R PAdicRational(p: Integer) is a domain constructor
--R Abbreviation for PAdicRational is PADICRAT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PADICRAT
--R
--R----- Operations -----
--R ???: (%,PAdicInteger(p)) -> %      ???: (PAdicInteger(p),%) -> %
--R ???: (Fraction(Integer),%) -> %    ???: (%,Fraction(Integer)) -> %
--R ???: (%,%) -> %                    ???: (Integer,%) -> %
--R ???: (PositiveInteger,%) -> %      ???: (%,Integer) -> %
--R ???: (%,PositiveInteger) -> %      ???: (%,%) -> %
--R ???: (%,%) -> %                    -?: % -> %
--R ???: (%,%) -> %                    ???: (%,%) -> Boolean
--R 1 : () -> %                        0 : () -> %
--R ???: (%,Integer) -> %              ???: (%,PositiveInteger) -> %
--R associates? : (%,%) -> Boolean    coerce : PAdicInteger(p) -> %
--R coerce : Fraction(Integer) -> %   coerce : % -> %
--R coerce : Integer -> %              coerce : % -> OutputForm
--R denom : % -> PAdicInteger(p)      denominator : % -> %
--R factor : % -> Factored(%)          gcd : List(%) -> %
--R gcd : (%,%) -> %                  hash : % -> SingleInteger
--R inv : % -> %                       latex : % -> String
--R lcm : List(%) -> %                 lcm : (%,%) -> %
```

```

--R numer : % -> PAdicInteger(p)          numerator : % -> %
--R one? : % -> Boolean                    prime? : % -> Boolean
--R ?quo? : (%,% ) -> %                    recip : % -> Union(%, "failed")
--R ?rem? : (%,% ) -> %                    removeZeroes : (Integer,% ) -> %
--R removeZeroes : % -> %                  retract : % -> PAdicInteger(p)
--R sample : () -> %                       sizeLess? : (%,% ) -> Boolean
--R squareFree : % -> Factored(%)          squareFreePart : % -> %
--R unit? : % -> Boolean                    unitCanonical : % -> %
--R zero? : % -> Boolean                    ?~=?: (%,% ) -> Boolean
--R ??? : (NonNegativeInteger,% ) -> %
--R ??? : (% , NonNegativeInteger) -> %
--R ?/? : (PAdicInteger(p), PAdicInteger(p)) -> %
--R ?<? : (% , %) -> Boolean if PAdicInteger(p) has ORDSET
--R ?<=? : (% , %) -> Boolean if PAdicInteger(p) has ORDSET
--R ?>? : (% , %) -> Boolean if PAdicInteger(p) has ORDSET
--R ?>=? : (% , %) -> Boolean if PAdicInteger(p) has ORDSET
--R D : (% , (PAdicInteger(p) -> PAdicInteger(p))) -> %
--R D : (% , (PAdicInteger(p) -> PAdicInteger(p)), NonNegativeInteger) -> %
--R D : (% , List(Symbol), List(NonNegativeInteger)) -> % if PAdicInteger(p) has PDRING(SYMBOL)
--R D : (% , Symbol, NonNegativeInteger) -> % if PAdicInteger(p) has PDRING(SYMBOL)
--R D : (% , List(Symbol)) -> % if PAdicInteger(p) has PDRING(SYMBOL)
--R D : (% , Symbol) -> % if PAdicInteger(p) has PDRING(SYMBOL)
--R D : (% , NonNegativeInteger) -> % if PAdicInteger(p) has DIFRING
--R D : % -> % if PAdicInteger(p) has DIFRING
--R ?^? : (% , NonNegativeInteger) -> %
--R abs : % -> % if PAdicInteger(p) has OINTDOM
--R approximate : (% , Integer) -> Fraction(Integer)
--R ceiling : % -> PAdicInteger(p) if PAdicInteger(p) has INS
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if $ has CHARNZ and PAdicInteger(p) has PFECAT or PAdicInteger(p) has
--R coerce : Symbol -> % if PAdicInteger(p) has RETRACT(SYMBOL)
--R conditionP : Matrix(%) -> Union(Vector(%, "failed") if $ has CHARNZ and PAdicInteger(p) has PFECAT
--R continuedFraction : % -> ContinuedFraction(Fraction(Integer))
--R convert : % -> DoubleFloat if PAdicInteger(p) has REAL
--R convert : % -> Float if PAdicInteger(p) has REAL
--R convert : % -> InputForm if PAdicInteger(p) has KONVERT(INFORM)
--R convert : % -> Pattern(Float) if PAdicInteger(p) has KONVERT(PATTERN(FLOAT))
--R convert : % -> Pattern(Integer) if PAdicInteger(p) has KONVERT(PATTERN(INT))
--R differentiate : (% , (PAdicInteger(p) -> PAdicInteger(p))) -> %
--R differentiate : (% , (PAdicInteger(p) -> PAdicInteger(p)), NonNegativeInteger) -> %
--R differentiate : (% , List(Symbol), List(NonNegativeInteger)) -> % if PAdicInteger(p) has PDRING(SYMBOL)
--R differentiate : (% , Symbol, NonNegativeInteger) -> % if PAdicInteger(p) has PDRING(SYMBOL)
--R differentiate : (% , List(Symbol)) -> % if PAdicInteger(p) has PDRING(SYMBOL)
--R differentiate : (% , Symbol) -> % if PAdicInteger(p) has PDRING(SYMBOL)
--R differentiate : (% , NonNegativeInteger) -> % if PAdicInteger(p) has DIFRING
--R differentiate : % -> % if PAdicInteger(p) has DIFRING
--R divide : (% , %) -> Record(quotient: %, remainder: %)
--R ?.: (% , PAdicInteger(p)) -> % if PAdicInteger(p) has ELTAB(PADIC(p), PADIC(p))
--R euclideanSize : % -> NonNegativeInteger
--R eval : (% , Symbol, PAdicInteger(p)) -> % if PAdicInteger(p) has IEVALAB(SYMBOL, PADIC(p))

```

```

--R eval : (%,List(Symbol),List(PAdicInteger(p))) -> % if PAdicInteger(p) has IEVALAB(SYMBOL,PAD
--R eval : (%,List(Equation(PAdicInteger(p)))) -> % if PAdicInteger(p) has EVALAB(PADIC(p))
--R eval : (%,Equation(PAdicInteger(p))) -> % if PAdicInteger(p) has EVALAB(PADIC(p))
--R eval : (%,PAdicInteger(p),PAdicInteger(p)) -> % if PAdicInteger(p) has EVALAB(PADIC(p))
--R eval : (%,List(PAdicInteger(p)),List(PAdicInteger(p))) -> % if PAdicInteger(p) has EVALAB(PA
--R expressIdealMember : (List(%),%) -> Union(List(%),"failed")
--R exquo : (%,%) -> Union(%,"failed")
--R extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %),"failed")
--R extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %)
--R factorPolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%))
--R factorSquareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynom
--R floor : % -> PAdicInteger(p) if PAdicInteger(p) has INS
--R fractionPart : % -> % if PAdicInteger(p) has EUCDOM
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUniva
--R init : () -> % if PAdicInteger(p) has STEP
--R map : ((PAdicInteger(p) -> PAdicInteger(p)),%) -> %
--R max : (%,%) -> % if PAdicInteger(p) has ORDSET
--R min : (%,%) -> % if PAdicInteger(p) has ORDSET
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed")
--R negative? : % -> Boolean if PAdicInteger(p) has OINTDOM
--R nextItem : % -> Union(%,"failed") if PAdicInteger(p) has STEP
--R patternMatch : (%,Pattern(Float),PatternMatchResult(Float,%)) -> PatternMatchResult(Float,%)
--R patternMatch : (%,Pattern(Integer),PatternMatchResult(Integer,%)) -> PatternMatchResult(Inte
--R positive? : % -> Boolean if PAdicInteger(p) has OINTDOM
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %)
--R random : () -> % if PAdicInteger(p) has INS
--R reducedSystem : Matrix(%) -> Matrix(PAdicInteger(p))
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(PAdicInteger(p)),vec: Vector(PAD
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) i
--R reducedSystem : Matrix(%) -> Matrix(Integer) if PAdicInteger(p) has LINEXP(INT)
--R retract : % -> Integer if PAdicInteger(p) has RETRACT(INT)
--R retract : % -> Fraction(Integer) if PAdicInteger(p) has RETRACT(INT)
--R retract : % -> Symbol if PAdicInteger(p) has RETRACT(SYMBOL)
--R retractIfCan : % -> Union(Integer,"failed") if PAdicInteger(p) has RETRACT(INT)
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if PAdicInteger(p) has RETRACT(INT)
--R retractIfCan : % -> Union(Symbol,"failed") if PAdicInteger(p) has RETRACT(SYMBOL)
--R retractIfCan : % -> Union(PAdicInteger(p),"failed")
--R sign : % -> Integer if PAdicInteger(p) has OINTDOM
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(%)),SparseUnivariatePolynom
--R squareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(
--R subtractIfCan : (%,%) -> Union(%,"failed")
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R wholePart : % -> PAdicInteger(p) if PAdicInteger(p) has EUCDOM
--R
--E 1

)spool
)lisp (bye)

```

— PAdicRational.help —

===== PAdicRational examples =====

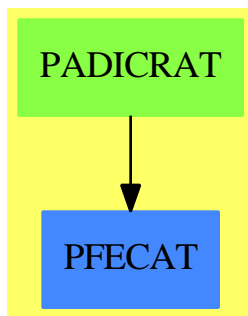
Stream-based implementation of \mathbb{Q}_p : numbers are represented as
 $\text{sum}(i = k.., a[i] * p^i)$
 where the $a[i]$ lie in $0, 1, \dots, (p - 1)$.

See Also:

o)show PAdicRational

—————

PAdicRational (PADICRAT)



See

- ⇒ “InnerPAdicInteger” (IPADIC) [10](#) on page [1294](#)
- ⇒ “PAdicInteger” (PADIC) [17](#) on page [1881](#)
- ⇒ “BalancedPAdicInteger” (BPADIC) [3](#) on page [258](#)
- ⇒ “PAdicRationalConstructor” (PADICRC) [17](#) on page [1890](#)
- ⇒ “BalancedPAdicRational” (BPADICRT) [3](#) on page [262](#)

Exports:

0	1	abs
approximate	associates?	ceiling
characteristic	charthRoot	coerce
conditionP	continuedFraction	convert
D	denom	denominator
differentiate	divide	euclideanSize
eval	expressIdealMember	exquo
extendedEuclidean	factor	factorPolynomial
factorSquareFreePolynomial	floor	fractionPart
gcd	gcdPolynomial	hash
init	inv	latex
lcm	map	max
min	multiEuclidean	negative?
nextItem	numer	numerator
one?	patternMatch	positive?
prime?	principalIdeal	random
recip	reducedSystem	removeZeroes
retract	retractIfCan	sample
sign	sizeLess?	solveLinearPolynomialEquation
squareFree	squareFreePart	squareFreePolynomial
subtractIfCan	unit?	unitCanonical
unitNormal	wholePart	zero?
?*?	?**?	?+?
?-?	~?	?/?
?=?	?^?	?~=?
?<?	?<=?	?>?
?>=?	?..?	?quo?
?rem?		

— domain PADICRAT PAdicRational —

```

)abbrev domain PADICRAT PAdicRational
++ Author: Clifton J. Williamson
++ Date Created: 15 May 1990
++ Date Last Updated: 15 May 1990
++ Description:
++ Stream-based implementation of Qp: numbers are represented as
++ sum(i = k.., a[i] * p^i) where the a[i] lie in 0,1,...,(p - 1).

PAdicRational(p:Integer) == PAdicRationalConstructor(p,PAdicInteger p)

```

— PADICRAT.dotabb —

"PADICRAT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PADICRAT"]

"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
 "PADICRAT" -> "PFECAT"

domain PADICRC PAdicRationalConstructor

— PAdicRationalConstructor.input —

```
)set break resume
)sys rm -f PAdicRationalConstructor.output
)spool PAdicRationalConstructor.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PAdicRationalConstructor
--R
--R PAdicRationalConstructor(p: Integer,PADIC: PAdicIntegerCategory(p)) is a domain constructor
--R Abbreviation for PAdicRationalConstructor is PADICRC
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PADICRC
--R
--R----- Operations -----
--R ?? : (%,PADIC) -> %          ?? : (PADIC,%) -> %
--R ?? : (Fraction(Integer),%) -> %  ?? : (%,Fraction(Integer)) -> %
--R ?? : (%,%) -> %             ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %    ??? : (%,Integer) -> %
--R ??? : (%,PositiveInteger) -> %  ?+? : (%,%) -> %
--R ?-? : (%,%) -> %             -? : % -> %
--R ?/? : (PADIC,PADIC) -> %        ?/? : (%,%) -> %
--R ?=? : (%,%) -> Boolean          D : (%,(PADIC -> PADIC)) -> %
--R 1 : () -> %                     0 : () -> %
--R ?? : (%,Integer) -> %          ?? : (%,PositiveInteger) -> %
--R associates? : (%,%) -> Boolean  coerce : PADIC -> %
--R coerce : Fraction(Integer) -> %  coerce : % -> %
--R coerce : Integer -> %           coerce : % -> OutputForm
--R denom : % -> PADIC              denominator : % -> %
--R factor : % -> Factored(%)       gcd : List(%) -> %
--R gcd : (%,%) -> %                hash : % -> SingleInteger
--R inv : % -> %                     latex : % -> String
--R lcm : List(%) -> %              lcm : (%,%) -> %
--R map : ((PADIC -> PADIC),%) -> %  numer : % -> PADIC
--R numerator : % -> %              one? : % -> Boolean
--R prime? : % -> Boolean            ?quo? : (%,%) -> %
```

```

--R recip : % -> Union(%, "failed")
--R removeZeroes : (Integer, %) -> %
--R retract : % -> PADIC
--R sizeLess? : (%, %) -> Boolean
--R squareFreePart : % -> %
--R unitCanonical : % -> %
--R ?^=? : (%, %) -> Boolean
--R ?*? : (NonNegativeInteger, %) -> %
--R ?**? : (%, NonNegativeInteger) -> %
--R ?<? : (%, %) -> Boolean if PADIC has ORDSET
--R ?<=? : (%, %) -> Boolean if PADIC has ORDSET
--R ?>? : (%, %) -> Boolean if PADIC has ORDSET
--R ?>=? : (%, %) -> Boolean if PADIC has ORDSET
--R D : (%, (PADIC -> PADIC), NonNegativeInteger) -> %
--R D : (%, List(Symbol), List(NonNegativeInteger)) -> % if PADIC has PDRING(SYMBOL)
--R D : (%, Symbol, NonNegativeInteger) -> % if PADIC has PDRING(SYMBOL)
--R D : (%, List(Symbol)) -> % if PADIC has PDRING(SYMBOL)
--R D : (%, Symbol) -> % if PADIC has PDRING(SYMBOL)
--R D : (%, NonNegativeInteger) -> % if PADIC has DIFRING
--R D : % -> % if PADIC has DIFRING
--R ?^? : (%, NonNegativeInteger) -> %
--R abs : % -> % if PADIC has OINTDOM
--R approximate : (%, Integer) -> Fraction(Integer)
--R ceiling : % -> PADIC if PADIC has INS
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if $ has CHARNZ and PADIC has PFECAT or PADIC has CHARNZ
--R coerce : Symbol -> % if PADIC has RETRACT(SYMBOL)
--R conditionP : Matrix(%) -> Union(Vector(%), "failed") if $ has CHARNZ and PADIC has PFECAT
--R continuedFraction : % -> ContinuedFraction(Fraction(Integer))
--R convert : % -> DoubleFloat if PADIC has REAL
--R convert : % -> Float if PADIC has REAL
--R convert : % -> InputForm if PADIC has KONVERT(INFORM)
--R convert : % -> Pattern(Float) if PADIC has KONVERT(PATTERN(FLOAT))
--R convert : % -> Pattern(Integer) if PADIC has KONVERT(PATTERN(INT))
--R differentiate : (%, (PADIC -> PADIC)) -> %
--R differentiate : (%, (PADIC -> PADIC), NonNegativeInteger) -> %
--R differentiate : (%, List(Symbol), List(NonNegativeInteger)) -> % if PADIC has PDRING(SYMBOL)
--R differentiate : (%, Symbol, NonNegativeInteger) -> % if PADIC has PDRING(SYMBOL)
--R differentiate : (%, List(Symbol)) -> % if PADIC has PDRING(SYMBOL)
--R differentiate : (%, Symbol) -> % if PADIC has PDRING(SYMBOL)
--R differentiate : (%, NonNegativeInteger) -> % if PADIC has DIFRING
--R differentiate : % -> % if PADIC has DIFRING
--R divide : (%, %) -> Record(quotient: %, remainder: %)
--R ?.? : (%, PADIC) -> % if PADIC has ELTAB(PADIC, PADIC)
--R euclideanSize : % -> NonNegativeInteger
--R eval : (%, Symbol, PADIC) -> % if PADIC has IEVALAB(SYMBOL, PADIC)
--R eval : (%, List(Symbol), List(PADIC)) -> % if PADIC has IEVALAB(SYMBOL, PADIC)
--R eval : (%, List(Equation(PADIC))) -> % if PADIC has EVALAB(PADIC)
--R eval : (%, Equation(PADIC)) -> % if PADIC has EVALAB(PADIC)
--R eval : (%, PADIC, PADIC) -> % if PADIC has EVALAB(PADIC)

```

```

--R eval : (% , List(PADIC) , List(PADIC)) -> % if PADIC has EVALAB(PADIC)
--R expressIdealMember : (List(%),%) -> Union(List(%), "failed")
--R exquo : (% , %) -> Union(% , "failed")
--R extendedEuclidean : (% , % , %) -> Union(Record(coef1: % , coef2: %), "failed")
--R extendedEuclidean : (% , %) -> Record(coef1: % , coef2: % , generator: %)
--R factorPolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if PADIC has
--R factorSquareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if
--R floor : % -> PADIC if PADIC has INS
--R fractionPart : % -> % if PADIC has EUCDOM
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R init : () -> % if PADIC has STEP
--R max : (% , %) -> % if PADIC has ORDSET
--R min : (% , %) -> % if PADIC has ORDSET
--R multiEuclidean : (List(%),%) -> Union(List(%), "failed")
--R negative? : % -> Boolean if PADIC has OINTDOM
--R nextItem : % -> Union(% , "failed") if PADIC has STEP
--R patternMatch : (% , Pattern(Float) , PatternMatchResult(Float,%)) -> PatternMatchResult(Float,% ) if PADIC has
--R patternMatch : (% , Pattern(Integer) , PatternMatchResult(Integer,%)) -> PatternMatchResult(Integer,% ) if PADIC has
--R positive? : % -> Boolean if PADIC has OINTDOM
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %)
--R random : () -> % if PADIC has INS
--R reducedSystem : Matrix(%) -> Matrix(PADIC)
--R reducedSystem : (Matrix(%), Vector(%)) -> Record(mat: Matrix(PADIC), vec: Vector(PADIC))
--R reducedSystem : (Matrix(%), Vector(%)) -> Record(mat: Matrix(Integer), vec: Vector(Integer)) if PADIC has
--R reducedSystem : Matrix(%) -> Matrix(Integer) if PADIC has LINEXP(INT)
--R retract : % -> Integer if PADIC has RETRACT(INT)
--R retract : % -> Fraction(Integer) if PADIC has RETRACT(INT)
--R retract : % -> Symbol if PADIC has RETRACT(SYMBOL)
--R retractIfCan : % -> Union(Integer, "failed") if PADIC has RETRACT(INT)
--R retractIfCan : % -> Union(Fraction(Integer), "failed") if PADIC has RETRACT(INT)
--R retractIfCan : % -> Union(Symbol, "failed") if PADIC has RETRACT(SYMBOL)
--R retractIfCan : % -> Union(PADIC, "failed")
--R sign : % -> Integer if PADIC has OINTDOM
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(%)), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R squareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if PADIC has
--R subtractIfCan : (% , %) -> Union(% , "failed")
--R unitNormal : % -> Record(unit: % , canonical: % , associate: %)
--R wholePart : % -> PADIC if PADIC has EUCDOM
--R
--E 1

)spool
)lisp (bye)

```

PAdicRationalConstructor examples

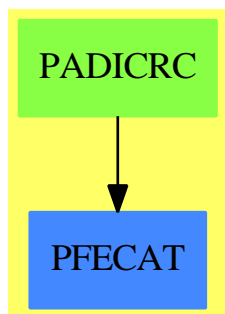
=====

This is the category of stream-based representations of \mathbb{Q}_p .

See Also:

o)show PAdicRationalConstructor

PAdicRationalConstructor (PADICRC)



See

- ⇒ “InnerPAdicInteger” (IPADIC) [10](#) on page [1294](#)
- ⇒ “PAdicInteger” (PADIC) [17](#) on page [1881](#)
- ⇒ “BalancedPAdicInteger” (BPADIC) [3](#) on page [258](#)
- ⇒ “PAdicRational” (PADICRAT) [17](#) on page [1885](#)
- ⇒ “BalancedPAdicRational” (BPADICRT) [3](#) on page [262](#)

Exports:

0	1	abs
approximate	associates?	ceiling
characteristic	charthRoot	coerce
conditionP	continuedFraction	convert
D	denom	denominator
differentiate	divide	euclideanSize
eval	expressIdealMember	exquo
extendedEuclidean	factor	factorPolynomial
factorSquareFreePolynomial	floor	fractionPart
gcd	gcdPolynomial	gcd
hash	init	inv
latex	lcm	map
max	min	multiEuclidean
negative?	nextItem	numer
numerator	one?	patternMatch
positive?	prime?	principalIdeal
random	recip	reducedSystem
removeZeroes	retract	retractIfCan
sample	sign	sizeLess?
solveLinearPolynomialEquation	squareFree	squareFreePart
squareFreePolynomial	subtractIfCan	unit?
unitCanonical	unitNormal	wholePart
zero?	?*?	?**?
?+?	?-?	-?
?/?	?=?	?^?
?~=?	?<?	?<=?
?>?	?>=?	?..?
?quo?	?rem?	

— domain PADICRC PAdicRationalConstructor —

```

)abbrev domain PADICRC PAdicRationalConstructor
++ Author: Clifton J. Williamson
++ Date Created: 10 May 1990
++ Date Last Updated: 10 May 1990
++ Description:
++ This is the category of stream-based representations of Qp.

PAdicRationalConstructor(p,PADIC): Exports == Implementation where
p      : Integer
PADIC  : PAdicIntegerCategory p
CF     ==> ContinuedFraction
I      ==> Integer
NNI    ==> NonNegativeInteger
OUT    ==> OutputForm
L      ==> List
RN     ==> Fraction Integer
ST     ==> Stream

```

```

Exports ==> QuotientFieldCategory(PADIC) with
  approximate: (% , I) -> RN
    ++ \spad{approximate(x,n)} returns a rational number y such that
    ++ \spad{y = x (mod p^n)}.
  continuedFraction: % -> CF RN
    ++ \spad{continuedFraction(x)} converts the p-adic rational number x
    ++ to a continued fraction.
  removeZeroes: % -> %
    ++ \spad{removeZeroes(x)} removes leading zeroes from the
    ++ representation of the p-adic rational \spad{x}.
    ++ A p-adic rational is represented by (1) an exponent and
    ++ (2) a p-adic integer which may have leading zero digits.
    ++ When the p-adic integer has a leading zero digit, a 'leading zero'
    ++ is removed from the p-adic rational as follows:
    ++ the number is rewritten by increasing the exponent by 1 and
    ++ dividing the p-adic integer by p.
    ++ Note: \spad{removeZeroes(f)} removes all leading zeroes from f.
  removeZeroes: (I,% ) -> %
    ++ \spad{removeZeroes(n,x)} removes up to n leading zeroes from
    ++ the p-adic rational \spad{x}.

Implementation ==> add

  PEXPR := p :: OUT

--% representation

  Rep := Record(expon:I,pint:PADIC)

  getExpon: % -> I
  getZp    : % -> PADIC
  makeQp   : (I,PADIC) -> %

  getExpon x    == x.expon
  getZp x       == x.pint
  makeQp(r,int) == [r,int]

--% creation

  0 == makeQp(0,0)
  1 == makeQp(0,1)

  coerce(x:I)      == x :: PADIC :: %
  coerce(r:RN)     == (numer(r) :: %)/(denom(r) :: %)
  coerce(x:PADIC) == makeQp(0,x)

--% normalizations

  removeZeroes x ==

```

```

empty? digits(xx := getZp x) => 0
zero? moduloP xx =>
  removeZeroes makeQp(getExpon x + 1,quotientByP xx)
x

removeZeroes(n,x) ==
  n <= 0 => x
  empty? digits(xx := getZp x) => 0
  zero? moduloP xx =>
    removeZeroes(n - 1,makeQp(getExpon x + 1,quotientByP xx))
  x

--% arithmetic

x = y ==
  EQ(x,y)$Lisp => true
  n := getExpon(x) - getExpon(y)
  n >= 0 =>
    (p**(n :: NNI) * getZp(x)) = getZp(y)
    (p**((- n) :: NNI) * getZp(y)) = getZp(x)

x + y ==
  n := getExpon(x) - getExpon(y)
  n >= 0 =>
    makeQp(getExpon y,getZp(y) + p**(n :: NNI) * getZp(x))
    makeQp(getExpon x,getZp(x) + p**((-n) :: NNI) * getZp(y))

-x == makeQp(getExpon x,-getZp(x))

x - y ==
  n := getExpon(x) - getExpon(y)
  n >= 0 =>
    makeQp(getExpon y,p**(n :: NNI) * getZp(x) - getZp(y))
    makeQp(getExpon x,getZp(x) - p**((-n) :: NNI) * getZp(y))

n:I * x:% == makeQp(getExpon x,n * getZp x)
x:% * y:% == makeQp(getExpon x + getExpon y,getZp x * getZp y)

x:% ** n:I ==
  zero? n => 1
  positive? n => expt(x,n :: PositiveInteger)$RepeatedSquaring(%)
  inv expt(x,(-n) :: PositiveInteger)$RepeatedSquaring(%)

recip x ==
  x := removeZeroes(1000,x)
  zero? moduloP(xx := getZp x) => "failed"
  (inv := recip xx) case "failed" => "failed"
  makeQp(- getExpon x,inv :: PADIC)

inv x ==

```



```

(inv := recip x) case "failed" => error "inv: no inverse"
inv :: %

x:% / y:% == x * inv y
x:PADIC / y:PADIC == (x :: %) / (y :: %)
x:PADIC * y:% == makeQp(getExpon y, x * getZp y)

approximate(x,n) ==
  k := getExpon x
  (p :: RN) ** k * approximate(getZp x, n - k)

cfStream: % -> Stream RN
cfStream x == delay
--   zero? x => empty()
   invx := inv x; x0 := approximate(invx,1)
   concat(x0, cfStream(invx - (x0 :: %)))

continuedFraction x ==
  x0 := approximate(x,1)
  reducedContinuedFraction(x0, cfStream(x - (x0 :: %)))

termOutput:(I,I) -> OUT
termOutput(k,c) ==
  k = 0 => c :: OUT
  mon := (k = 1 => PEXPR; PEXPR ** (k :: OUT))
  c = 1 => mon
  c = -1 => -mon
  (c :: OUT) * mon

showAll?:() -> Boolean
-- check a global Lisp variable
showAll?() == true

coerce(x:%):OUT ==
  x := removeZeroes(_$streamCount$Lisp,x)
  m := getExpon x; zp := getZp x
  uu := digits zp
  l : L OUT := empty()
  empty? uu => 0 :: OUT
  n : NNI ; count : NNI := _$streamCount$Lisp
  for n in 0..count while not empty? uu repeat
    if frst(uu) ^= 0 then
      l := concat(termOutput((n :: I) + m, frst(uu)), l)
      uu := rst uu
  if showAll?() then
    for n in (count + 1).. while explicitEntries? uu and _
      not eq?(uu, rst uu) repeat
      if frst(uu) ^= 0 then
        l := concat(termOutput((n::I) + m, frst(uu)), l)
        uu := rst uu

```

```

l :=
  explicitlyEmpty? uu => 1
  eq?(uu,rst uu) and first uu = 0 => 1
  concat(prefix("0" :: OUT,[PEXPR ** ((n :: I) + m) :: OUT]),1)
empty? l => 0 :: OUT
reduce("+",reverse! l)

```

— PADICRC.dotabb —

```

"PADICRC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PADICRC"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PADICRC" -> "PFECAT"

```

domain PALETTE Palette

— Palette.input —

```

)set break resume
)sys rm -f Palette.output
)spool Palette.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Palette
--R Palette is a domain constructor
--R Abbreviation for Palette is PALETTE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PALETTE
--R
--R----- Operations -----
--R ?=? : (%,%) -> Boolean          bright : Color -> %
--R coerce : Color -> %            coerce : % -> OutputForm
--R dark : Color -> %              dim : Color -> %
--R hash : % -> SingleInteger      hue : % -> Color
--R latex : % -> String            light : Color -> %
--R pastel : Color -> %            shade : % -> Integer
--R ?~=? : (%,%) -> Boolean
--R

```

```
--E 1
```

```
)spool
)lisp (bye)
```

— **Palette.help** —

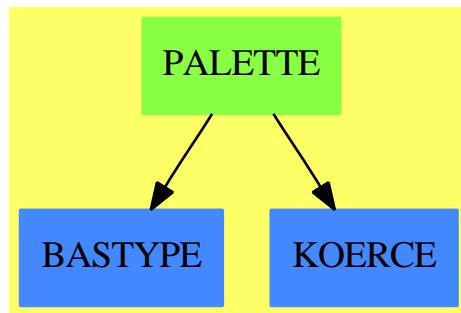
```
=====
Palette examples
=====
```

This domain describes four groups of color shades (palettes).

See Also:

o)show Palette

Palette (PALETTE)



See

⇒ “Color” (COLOR) [4](#) on page [407](#)

Exports:

bright	coerce	dark	dim	hash
hue	latex	light	pastel	shade
?~=?	?=?			

— **domain PALETTE Palette** —

```
)abbrev domain PALETTE Palette
++ Author: Jim Wen
++ Date Created: May 10th 1989
```

```

++ Date Last Updated: Jan 19th 1990
++ Description:
++ This domain describes four groups of color shades (palettes).

```

```

Palette(): Exports == Implementation where
I      ==> Integer
C      ==> Color
SHADE  ==> ["Dark","Dim","Bright","Pastel","Light"]

```

```

Exports ==> SetCategory with
dark   : C -> %
  ++ dark(c) sets the shade of the indicated hue of c to it's lowest value.
dim    : C -> %
  ++ dim(c) sets the shade of a hue, c,  above dark, but below bright.
bright : C -> %
  ++ bright(c) sets the shade of a hue, c, above dim, but below pastel.
pastel : C -> %
  ++ pastel(c) sets the shade of a hue, c,  above bright, but below light.
light  : C -> %
  ++ light(c) sets the shade of a hue, c,  to it's highest value.
hue    : % -> C
  ++ hue(p) returns the hue field of the indicated palette p.
shade  : % -> I
  ++ shade(p) returns the shade index of the indicated palette p.
coerce : C -> %
  ++ coerce(c) sets the average shade for the palette to that of the
  ++ indicated color c.

```

```

Implementation ==> add
Rep := Record(shadeField:I, hueField:C)

dark   c == [1,c]
dim    c == [2,c]
bright c == [3,c]
pastel c == [4,c]
light  c == [5,c]
hue    p == p.hueField
shade  p == p.shadeField
sample() == bright(sample())
coerce(c:Color):% == bright c
coerce(p:%):OutputForm ==
  hconcat ["[,coerce(p.hueField),"] from the ",_
           SHADE.(p.shadeField)," palette"]

```

```

"PALETTE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PALETTE"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"PALETTE" -> "BASTYPE"
"PALETTE" -> "KOERCE"

```

domain PARPCURV ParametricPlaneCurve

— ParametricPlaneCurve.input —

```

)set break resume
)sys rm -f ParametricPlaneCurve.output
)spool ParametricPlaneCurve.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ParametricPlaneCurve
--R
--R ParametricPlaneCurve(ComponentFunction: Type) is a domain constructor
--R Abbreviation for ParametricPlaneCurve is PARPCURV
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PARPCURV
--R
--R----- Operations -----
--R coordinate : (%,NonNegativeInteger) -> ComponentFunction
--R curve : (ComponentFunction,ComponentFunction) -> %
--R
--E 1

)spool
)lisp (bye)

```

— ParametricPlaneCurve.help —

```

=====
ParametricPlaneCurve examples
=====

ParametricPlaneCurve is used for plotting parametric plane

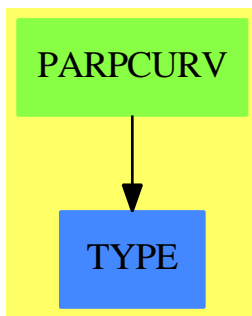
```

curves in the affine plane.

See Also:

o)show ParametricPlaneCurve

ParametricPlaneCurve (PARPCURV)



See

⇒ “ParametricSpaceCurve” (PARSCURV) [17](#) on page [1901](#)

⇒ “ParametricSurface” (PARSURF) [17](#) on page [1904](#)

Exports:

coordinate curve

— domain PARPCURV ParametricPlaneCurve —

```

)abbrev domain PARPCURV ParametricPlaneCurve
++ Author: Clifton J. Williamson
++ Date Created: 24 May 1990
++ Date Last Updated: 24 May 1990
++ Description:
++ ParametricPlaneCurve is used for plotting parametric plane
++ curves in the affine plane.

ParametricPlaneCurve(ComponentFunction): Exports == Implementation where
  ComponentFunction : Type
  NNI                 ==> NonNegativeInteger

Exports ==> with
  curve: (ComponentFunction,ComponentFunction) -> %
    ++ curve(c1,c2) creates a plane curve from 2 component functions \spad{c1}
    ++ and \spad{c2}.
  coordinate: (% ,NNI) -> ComponentFunction
  
```

```

++ coordinate(c,i) returns a coordinate function for c using 1-based
++ indexing according to i. This indicates what the function for the
++ coordinate component i of the plane curve is.

Implementation ==> add

Rep := Record(xCoord:ComponentFunction,yCoord:ComponentFunction)

curve(x,y) == [x,y]
coordinate(c,n) ==
  n = 1 => c.xCoord
  n = 2 => c.yCoord
  error "coordinate: index out of bounds"



---


— PARPCURV.dotabb —

"PARPCURV" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PARPCURV"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"PARPCURV" -> "TYPE"



---



```

domain PARSCURV ParametricSpaceCurve

```

— ParametricSpaceCurve.input —

)set break resume
)sys rm -f ParametricSpaceCurve.output
)spool ParametricSpaceCurve.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ParametricSpaceCurve
--R
--R ParametricSpaceCurve(ComponentFunction: Type) is a domain constructor
--R Abbreviation for ParametricSpaceCurve is PARSCURV
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PARSCURV
--R
--R----- Operations -----

```

```

--R coordinate : (%,NonNegativeInteger) -> ComponentFunction
--R curve : (ComponentFunction,ComponentFunction,ComponentFunction) -> %
--R
--E 1

)spool
)lisp (bye)

```

— ParametricSpaceCurve.help —

=====

ParametricSpaceCurve examples

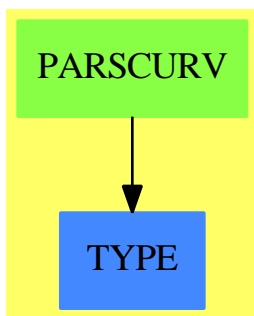
=====

ParametricSpaceCurve is used for plotting parametric space curves in affine 3-space.

See Also:

o)show ParametricSpaceCurve

ParametricSpaceCurve (PARSCURV)



See

⇒ “ParametricPlaneCurve” (PARPCURV) [17](#) on page [1899](#)

⇒ “ParametricSurface” (PARSURF) [17](#) on page [1904](#)

Exports:

coordinate curve

— domain PARSCURV ParametricSpaceCurve —


```

)abbrev domain PARSCURV ParametricSpaceCurve
++ Author: Clifton J. Williamson
++ Date Created: 24 May 1990
++ Date Last Updated: 24 May 1990
++ Description:
++ ParametricSpaceCurve is used for plotting parametric space
++ curves in affine 3-space.

ParametricSpaceCurve(ComponentFunction): Exports == Implementation where
  ComponentFunction : Type
  NNI                ==> NonNegativeInteger

Exports ==> with
  curve: (ComponentFunction,ComponentFunction,ComponentFunction) -> %
    ++ curve(c1,c2,c3) creates a space curve from 3 component functions
    ++ \spad{c1}, \spad{c2}, and \spad{c3}.
  coordinate: (% ,NNI) -> ComponentFunction
    ++ coordinate(c,i) returns a coordinate function of c using 1-based
    ++ indexing according to i. This indicates what the function for the
    ++ coordinate component, i, of the space curve is.

Implementation ==> add

Rep := Record(xCoord:ComponentFunction,
              yCoord:ComponentFunction,
              zCoord:ComponentFunction)

curve(x,y,z) == [x,y,z]
coordinate(c,n) ==
  n = 1 => c.xCoord
  n = 2 => c.yCoord
  n = 3 => c.zCoord
  error "coordinate: index out of bounds"

-----

— PARSCURV.dotabb —

"PARSCURV" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PARSCURV"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"PARSCURV" -> "TYPE"

-----

```

domain PARSURF ParametricSurface**— ParametricSurface.input —**

```

)set break resume
)sys rm -f ParametricSurface.output
)spool ParametricSurface.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ParametricSurface
--R
--R ParametricSurface(ComponentFunction: Type) is a domain constructor
--R Abbreviation for ParametricSurface is PARSURF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PARSURF
--R
--R----- Operations -----
--R coordinate : (%,NonNegativeInteger) -> ComponentFunction
--R surface : (ComponentFunction,ComponentFunction,ComponentFunction) -> %
--R
--E 1

)spool
)lisp (bye)

```

— ParametricSurface.help —

```

=====
ParametricSurface examples
=====

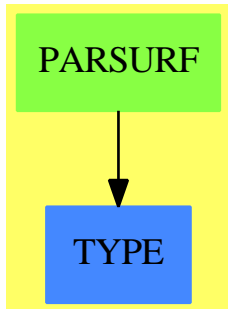
```

ParametricSurface is used for plotting parametric surfaces in affine 3-space.

See Also:

o)show ParametricSurface

ParametricSurface (PARSURF)



See

⇒ “ParametricPlaneCurve” (PARPCURV) [17](#) on page [1899](#)

⇒ “ParametricSpaceCurve” (PARSCURV) [17](#) on page [1901](#)

Exports:

coordinate surface

— domain **PARSURF** ParametricSurface —

```

)abbrev domain PARSURF ParametricSurface
++ Author: Clifton J. Williamson
++ Date Created: 24 May 1990
++ Date Last Updated: 24 May 1990
++ Description:
++ ParametricSurface is used for plotting parametric surfaces in
++ affine 3-space.

ParametricSurface(ComponentFunction): Exports == Implementation where
  ComponentFunction : Type
  NNI                 ==> NonNegativeInteger

Exports ==> with
  surface: (ComponentFunction,ComponentFunction,ComponentFunction) -> %
    ++ surface(c1,c2,c3) creates a surface from 3 parametric component
    ++ functions \spad{c1}, \spad{c2}, and \spad{c3}.
  coordinate: (% ,NNI) -> ComponentFunction
    ++ coordinate(s,i) returns a coordinate function of s using 1-based
    ++ indexing according to i. This indicates what the function for the
    ++ coordinate component, i, of the surface is.

Implementation ==> add

Rep := Record(xCoord:ComponentFunction,
              yCoord:ComponentFunction,
              zCoord:ComponentFunction)
  
```

```

surface(x,y,z) == [x,y,z]
coordinate(c,n) ==
  n = 1 => c.xCoord
  n = 2 => c.yCoord
  n = 3 => c.zCoord
  error "coordinate: index out of bounds"

```

— PARSURF.dotabb —

```

"PARSURF" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PARSURF"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"PARSURF" -> "TYPE"

```

domain PFR PartialFraction

— PartialFraction.input —

```

)set break resume
)sys rm -f PartialFraction.output
)spool PartialFraction.output
)set message test on
)set message auto off
)clear all

```

```
--S 1 of 22
```

```
partialFraction(1,factorial 10)
```

```
--R
```

```
--R
```

```
--R      159   23   12   1
```

```
--R  (1) --- - -- - -- + -
```

```
--R      8    4    2    7
```

```
--R      2    3    5
```

```
--R
```

Type: PartialFraction(Integer)

```
--E 1
```

```
--S 2 of 22
```

```
f := padicFraction(%)
```

```
--R
```

```
--R
```

```

--R      1   1   1   1   1   1   2   1   2   2   2   1
--R  (2)  - + - + - + - + - + - - - - - - - - - + -
--R      2   4   5   6   7   8   2   3   4   5   2   7
--R      2   2   2   2   2   3   3   3   5
--R                                          Type: PartialFraction(Integer)
--E 2

```

```

--S 3 of 22
compactFraction(f)
--R
--R
--R      159   23   12   1
--R  (3)  ---- - - - - - + -
--R      8     4     2   7
--R      2     3     5
--R                                          Type: PartialFraction(Integer)
--E 3

```

```

--S 4 of 22
numberOfFractionalTerms(f)
--R
--R
--R  (4)  12
--R                                          Type: PositiveInteger
--E 4

```

```

--S 5 of 22
nthFractionalTerm(f,3)
--R
--R
--R      1
--R  (5)  --
--R      5
--R      2
--R                                          Type: PartialFraction(Integer)
--E 5

```

```

--S 6 of 22
partialFraction(1,- 13 + 14 * %i)
--R
--R
--R      1       4
--R  (6)  - ---- + ----
--R      1 + 2%i  3 + 8%i
--R                                          Type: PartialFraction(Complex(Integer))
--E 6

```

```

--S 7 of 22
% :: Fraction Complex Integer
--R

```

```

--R
--R      %i
--R  (7)  - ----
--R      14 + 13%i
--R
--R                                          Type: Fraction(Complex(Integer))
--E 7

```

```

--S 8 of 22
u : FR UP(x, FRAC INT) := reduce(*,[primeFactor(x+i,i) for i in 1..4])
--R
--R
--R      2      3      4
--R  (8)  (x + 1)(x + 2)(x + 3)(x + 4)
--R
--R                                          Type: Factored(UnivariatePolynomial(x,Fraction(Integer)))
--E 8

```

```

--S 9 of 22
partialFraction(1,u)
--R
--R
--R  (9)
--R      1      1      7      17 2      139      607 3      10115 2      391      44179
--R      --- - x + --- - -- x - 12x - --- --- x + --- x + --- x + ---
--R      648      4      16      8      8      324      432      4      324
--R      ----- + ----- + ----- + -----
--R      x + 1      2      3      4
--R      (x + 2)      (x + 3)      (x + 4)
--R
--R                                          Type: PartialFraction(UnivariatePolynomial(x,Fraction(Integer)))
--E 9

```

```

--S 10 of 22
padicFraction %
--R
--R
--R  (10)
--R      1      1      1      17      3      1      607      403
--R      --- - - - - -- -- - - - - -
--R      648      4      16      8      4      2      324      432
--R      ----- + ----- - ----- - ----- + ----- - ----- + -----
--R      x + 1      x + 2      2      x + 3      2      3      x + 4      2
--R      (x + 2)      (x + 3)      (x + 3)      (x + 3)      (x + 4)
--R
--R  +
--R      13      1
--R      -- --
--R      36      12
--R      ----- + -----
--R      3      4
--R      (x + 4)      (x + 4)
--R
--R                                          Type: PartialFraction(UnivariatePolynomial(x,Fraction(Integer)))
--E 10

```

```

--S 11 of 22
fraction:=Fraction(Polynomial(Integer))
--R
--R
--R (11) Fraction(Polynomial(Integer))
--R
--R                                          Type: Domain
--E 11

--S 12 of 22
up:=UnivariatePolynomial(y,fraction)
--R
--R
--R (12) UnivariatePolynomial(y,Fraction(Polynomial(Integer)))
--R
--R                                          Type: Domain
--E 12

--S 13 of 22
pfup:=PartialFraction(up)
--R
--R
--R (13)
--R PartialFraction(UnivariatePolynomial(y,Fraction(Polynomial(Integer))))
--R
--R                                          Type: Domain
--E 13

--S 14 of 22
a:=x+1/(y+1)
--R
--R
--R
--R      x y + x + 1
--R (14)  -----
--R      y + 1
--R
--R                                          Type: Fraction(Polynomial(Integer))
--E 14

--S 15 of 22
b:=partialFraction(a,y)$PartialFractionPackage(Integer)
--R
--R
--R
--R      1
--R (15)  x + ----
--R      y + 1
--R
--R Type: PartialFraction(UnivariatePolynomial(y,Fraction(Polynomial(Integer))))
--E 15

--S 16 of 22
c:=b::pfup
--R
--R

```

[illegible]


```

--E 21

--S 22 of 22
sin(cw)*cos(ccx)+sin(ccx)*cos(cw)
--R
--R
--R
--R      1      1
--R      (22)  cos(-----)sin(x) + cos(x)sin(-----)
--R             y + 1      y + 1
--R
--R                                          Type: Expression(Integer)
--E 22

)spool
)lisp (bye)

```

— PartialFraction.help —

=====
PartialFraction examples
=====

A partial fraction is a decomposition of a quotient into a sum of quotients where the denominators of the summands are powers of primes. Most people first encounter partial fractions when they are learning integral calculus. For a technical discussion of partial fractions, see, for example, Lang's Algebra. For example, the rational number $1/6$ is decomposed into $1/2-1/3$. You can compute partial fractions of quotients of objects from domains belonging to the category `EuclideanDomain`. For example, `Integer`, `Complex Integer`, and `UnivariatePolynomial(x, Fraction Integer)` all belong to `EuclideanDomain`. In the examples following, we demonstrate how to decompose quotients of each of these kinds of object into partial fractions.

It is necessary that we know how to factor the denominator when we want to compute a partial fraction. Although the interpreter can often do this automatically, it may be necessary for you to include a call to `factor`. In these examples, it is not necessary to factor the denominators explicitly.

The main operation for computing partial fractions is called `partialFraction` and we use this to compute a decomposition of $1/10!$. The first argument to `partialFraction` is the numerator of the quotient and the second argument is the factored denominator.

```

partialFraction(1,factorial 10)
159   23   12   1
--- - -- - -- + -

```

$$\frac{2^8}{2} + \frac{3^4}{3} + \frac{5^2}{5} + \frac{7}{7}$$

Type: PartialFraction Integer

Since the denominators are powers of primes, it may be possible to expand the numerators further with respect to those primes. Use the operation `padicFraction` to do this.

```
f := padicFraction(%)
```

$$-\frac{1}{2} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^6} + \frac{1}{2^7} + \frac{1}{2^8} - \frac{2}{2^2} - \frac{1}{2^3} - \frac{2}{2^4} - \frac{2}{2^5} - \frac{2}{2^6} + \frac{1}{2^7}$$

Type: PartialFraction Integer

The operation `compactFraction` returns an expanded fraction into the usual form. The compacted version is used internally for computational efficiency.

```
compactFraction(f)
```

$$-\frac{159}{2} - \frac{23}{3} - \frac{12}{5} + \frac{1}{7}$$

Type: PartialFraction Integer

You can add, subtract, multiply and divide partial fractions. In addition, you can extract the parts of the decomposition. `numberOfFractionalTerms` computes the number of terms in the fractional part. This does not include the whole part of the fraction, which you get by calling `wholePart`. In this example, the whole part is just 0.

```
numberOfFractionalTerms(f)
```

12

Type: PositiveInteger

The operation `nthFractionalTerm` returns the individual terms in the decomposition. Notice that the object returned is a partial fraction itself. `firstNumerator` and `firstDenominator` extract the numerator and denominator of the first term of the fraction.

```
nthFractionalTerm(f,3)
```

$$-\frac{1}{2^5}$$

Type: PartialFraction Integer

Given two gaussian integers, you can decompose their quotient into a partial fraction.

```
partialFraction(1,- 13 + 14 * %i)
      1      4
    - ---- + ----
      1 + 2%i  3 + 8%i
Type: PartialFraction Complex Integer
```

To convert back to a quotient, simply use a conversion.

```
% :: Fraction Complex Integer
      %i
    - ----
      14 + 13%i
Type: Fraction Complex Integer
```

To conclude this section, we compute the decomposition of

$$\frac{1}{(x+1)(x+2)^2(x+3)^3(x+4)^4}$$

The polynomials in this object have type
UnivariatePolynomial(x, Fraction Integer).

We use the primeFactor operation to create the denominator in factored form directly.

```
u : FR UP(x, FRAC INT) := reduce(*,[primeFactor(x+i,i) for i in 1..4])
      2      3      4
    (x + 1)(x + 2) (x + 3) (x + 4)
Type: Factored UnivariatePolynomial(x,Fraction Integer)
```

These are the compact and expanded partial fractions for the quotient.

```
partialFraction(1,u)
      1      1      7      17 2      139 607 3      10115 2      391      44179
    --- - x + -- - -- x - 12x - --- --- x + ---- x + --- x + ----
    648 4      16      8      8      324      432      4      324
    ---- + ---- + ---- + ----
    x + 1      2      3      4
      (x + 2)      (x + 3)      (x + 4)
Type: PartialFraction UnivariatePolynomial(x,Fraction Integer)

padicFraction %
      1      1      1      17      3      1      607      403
    --- - - -- -- - - - - - - - - -
    648 4      16      8      4      2      324      432
    ---- + ---- - ---- - ---- + ---- - ---- + ---- + ----
    x + 1 x + 2      2 x + 3      2      3 x + 4      2
```

```
See Also:
o )help Factored
o )help Complex
o )help FullPartialFractionExpansionXmpPage
o )show PartialFraction
```



0	1	associates?	characteristic
coerce	compactFraction	divide	euclideanSize
expressIdealMember	exquo	extendedEuclidean	extendedEuclidean
factor	firstDenom	firstNumer	gcd
gcdPolynomial	hash	inv	latex
lcm	multiEuclidean	nthFractionalTerm	numberOfFractionalTerms
one?	padicallyExpand	padicFraction	partialFraction
prime?	principalIdeal	recip	sample
sizeLess?	squareFree	squareFreePart	subtractIfCan
unit?	unitCanonical	unitNormal	wholePart
zero?	?*?	?**?	?+?
?-?	-?	?/?	?=?
?^?	?~=?	?quo?	?rem?

— domain PFR PartialFraction —

```

)abbrev domain PFR PartialFraction
++ Author: Robert S. Sutor
++ Date Created: 1986
++ Change History: 05/20/91 BMT Converted to the new library
++ Description:
++ The domain \spadtype{PartialFraction} implements partial fractions
++ over a euclidean domain \spad{R}. This requirement on the
++ argument domain allows us to normalize the fractions. Of
++ particular interest are the 2 forms for these fractions. The
++ ‘compact’ form has only one fractional term per prime in the
++ denominator, while the ‘p-adic’ form expands each numerator
++ p-adically via the prime p in the denominator. For computational
++ efficiency, the compact form is used, though the p-adic form may
++ be gotten by calling the function padicFraction}. For a
++ general euclidean domain, it is not known how to factor the
++ denominator. Thus the function partialFraction takes as its
++ second argument an element of \spadtype{Factored(R)}.

PartialFraction(R: EuclideanDomain): Cat == Capsule where
  FRR ==> Factored R
  SUPR ==> SparseUnivariatePolynomial R

  Cat == Join(Field, Algebra R) with
    coerce: % -> Fraction R
    ++ coerce(p) sums up the components of the partial fraction and
    ++ returns a single fraction.
    ++
    ++X a:=(13/74)::PFR(INT)
    ++X a::FRAC(INT)

  coerce: Fraction FRR -> %
    ++ coerce(f) takes a fraction with numerator and denominator in
    ++ factored form and creates a partial fraction. It is

```

```

++ necessary for the parts to be factored because it is not
++ known in general how to factor elements of \spad{R} and
++ this is needed to decompose into partial fractions.
++
++X (13/74)::PFR(INT)

compactFraction: % -> %
++ compactFraction(p) normalizes the partial fraction \spad{p}
++ to the compact representation. In this form, the partial
++ fraction has only one fractional term per prime in the
++ denominator.
++
++X a:=partialFraction(1,factorial 10)
++X b:=padicFraction(a)
++X compactFraction(b)

firstDenom: % -> FRR
++ firstDenom(p) extracts the denominator of the first fractional
++ term. This returns 1 if there is no fractional part (use
++ wholePart from PartialFraction to get the whole part).
++
++X a:=partialFraction(1,factorial 10)
++X firstDenom(a)

firstNumer: % -> R
++ firstNumer(p) extracts the numerator of the first fractional
++ term. This returns 0 if there is no fractional part (use
++ wholePart from PartialFraction to get the whole part).
++
++X a:=partialFraction(1,factorial 10)
++X firstNumer(a)

nthFractionalTerm: (%,Integer) -> %
++ nthFractionalTerm(p,n) extracts the nth fractional term from
++ the partial fraction \spad{p}. This returns 0 if the index
++ \spad{n} is out of range.
++
++X a:=partialFraction(1,factorial 10)
++X b:=padicFraction(a)
++X nthFractionalTerm(b,3)

numberOfFractionalTerms: % -> Integer
++ numberOfFractionalTerms(p) computes the number of fractional
++ terms in \spad{p}. This returns 0 if there is no fractional
++ part.
++
++X a:=partialFraction(1,factorial 10)
++X b:=padicFraction(a)
++X numberOfFractionalTerms(b)

```

```

padicallyExpand: (R,R) -> SUPR
++ padicallyExpand(p,x) is a utility function that expands
++ the second argument \spad{x} 'p-adically' in
++ the first.

padicFraction: % -> %
++ padicFraction(q) expands the fraction p-adically in the primes
++ \spad{p} in the denominator of \spad{q}. For example,
++ \spad{padicFraction(3/(2**2)) = 1/2 + 1/(2**2)}.
++ Use compactFraction from PartialFraction to
++ return to compact form.
++
++X a:=partialFraction(1,factorial 10)
++X padicFraction(a)

partialFraction: (R, FRR) -> %
++ partialFraction(numer,denom) is the main function for
++ constructing partial fractions. The second argument is the
++ denominator and should be factored.
++
++X partialFraction(1,factorial 10)

wholePart: % -> R
++ wholePart(p) extracts the whole part of the partial fraction
++ \spad{p}.
++
++X a:=(74/13)::PFR(INT)
++X wholePart(a)

Capsule == add

-- some constructor assignments and macros

Ex      ==> OutputForm
fTerm   ==> Record(num: R, den: FRR)           -- den should have
                                                -- unit = 1 and only
                                                -- 1 factor

LfTerm  ==> List Record(num: R, den: FRR)
QR      ==> Record(quotient: R, remainder: R)

Rep     := Record(whole:R, fract: LfTerm)

-- private function signatures

copypf: % -> %
LessThan: (fTerm, fTerm) -> Boolean
multiplyFracTerms: (fTerm, fTerm) -> %
normalizeFracTerm: fTerm -> %
partialFractionNormalized: (R, FRR) -> %

```

```

-- declarations

a,b: %
n: Integer
r: R

-- private function definitions

copypf(a: %): % == [a.whole,copy a.fract]$$%

LessThan(s: fTerm, t: fTerm) ==
  -- have to wait until FR has < operation
  if (GGREATERP(s.den,t.den)$Lisp : Boolean) then false
  else true

multiplyFracTerms(s : fTerm, t : fTerm) ==
  nthFactor(s.den,1) = nthFactor(t.den,1) =>
    normalizeFracTerm([s.num * t.num, s.den * t.den]$fTerm) : Rep
  i : Union(Record(coef1: R, coef2: R),"failed")
  coefs : Record(coef1: R, coef2: R)
  i := extendedEuclidean(expand t.den, expand s.den,s.num * t.num)
  i case "failed" => error "PartialFraction: not in ideal"
  coefs := (i :: Record(coef1: R, coef2: R))
  c : % := copypf 0$$%
  d : %
  if coefs.coef2 ~= 0$R then
    c := normalizeFracTerm ([coefs.coef2, t.den]$fTerm)
  if coefs.coef1 ~= 0$R then
    d := normalizeFracTerm ([coefs.coef1, s.den]$fTerm)
    c.whole := c.whole + d.whole
    not (null d.fract) => c.fract := append(d.fract,c.fract)
  c

normalizeFracTerm(s : fTerm) ==
  -- makes sure num is "less than" den, whole may be non-zero
  qr : QR := divide(s.num, (expand s.den))
  qr.remainder = 0$R => [qr.quotient, nil()$LfTerm]
  -- now verify num and den are coprime
  f : R := nthFactor(s.den,1)
  nexpon : Integer := nthExponent(s.den,1)
  expon : Integer := 0
  q : QR := divide(qr.remainder, f)
  while (q.remainder = 0$R) and (expon < nexpon) repeat
    expon := expon + 1
    qr.remainder := q.quotient
    q := divide(qr.remainder,f)
  expon = 0 => [qr.quotient,[[qr.remainder, s.den]$fTerm]$LfTerm]
  expon = nexpon => (qr.quotient + qr.remainder) :: %
  [qr.quotient,[[qr.remainder, nilFactor(f,nexpon-expon)]$fTerm]$LfTerm]

```



```

partialFractionNormalized(nm: R, dn : FRR) ==
  -- assume unit dn = 1
  nm = 0$R => 0$%
  dn = 1$FRR => nm :: %
  qr : QR := divide(nm, expand dn)
  c : % := [0$R, [[qr.remainder,
    nilFactor(nthFactor(dn,1), nthExponent(dn,1))]]$fTerm]$LfTerm]
  d : %
  for i in 2..numberOfFactors(dn) repeat
    d :=
      [0$R, [[1$R, nilFactor(nthFactor(dn,i), nthExponent(dn,i))]]$fTerm]$LfTerm]
    c := c * d
  (qr.quotient :: %) + c

-- public function definitions

padicFraction(a : %) ==
  b: % := compactFraction a
  null b.fract => b
  l : LfTerm := nil
  s : fTerm
  f : R
  e,d: Integer
  for s in b.fract repeat
    e := nthExponent(s.den,1)
    e = 1 => l := cons(s,l)
    f := nthFactor(s.den,1)
    d := degree(sp := padicallyExpand(f,s.num))
    while (sp ^= 0$SUPR) repeat
      l := cons([leadingCoefficient sp, nilFactor(f,e-d)]$fTerm, l)
      d := degree(sp := reductum sp)
  [b.whole, sort(LessThan,l)]$%

compactFraction(a : %) ==
  -- only one power for each distinct denom will remain
  2 > # a.fract => a
  af : LfTerm := reverse a.fract
  bf : LfTerm := nil
  bw : R := a.whole
  b : %
  s : fTerm := [(first af).num, (first af).den]$fTerm
  f : R := nthFactor(s.den,1)
  e : Integer := nthExponent(s.den,1)
  t : fTerm
  for t in rest af repeat
    f = nthFactor(t.den,1) =>
      s.num := s.num + (t.num *
        (f **$R ((e - nthExponent(t.den,1)) : NonNegativeInteger)))
  b := normalizeFracTerm s
  bw := bw + b.whole

```

```

    if not (null b.fract) then bf := cons(first b.fract,bf)
    s := [t.num, t.den]$fTerm
    f := nthFactor(s.den,1)
    e := nthExponent(s.den,1)
    b := normalizeFracTerm s
    [bw + b.whole,append(b.fract,bf)]$%

0 == [0$R, nil()$LfTerm]
1 == [1$R, nil()$LfTerm]
characteristic() == characteristic()$R

coerce(r): % == [r, nil()$LfTerm]
coerce(n): % == [(n :: R), nil()$LfTerm]
coerce(a): Fraction R ==
  q : Fraction R := (a.whole :: Fraction R)
  s : fTerm
  for s in a.fract repeat
    q := q + (s.num / (expand s.den))
  q
coerce(q: Fraction FRR): % ==
  u : R := (recip unit denom q):: R
  r1 : R := u * expand numer q
  partialFractionNormalized(r1, u * denom q)

a exquo b ==
  b = 0$% => "failed"
  b = 1$% => a
  br : Fraction R := inv (b :: Fraction R)
  a * partialFraction(numer br,(denom br) :: FRR)
recip a == (1$% exquo a)

firstDenom a ==          -- denominator of 1st fractional term
  null a.fract => 1$FRR
  (first a.fract).den
firstNumer a ==          -- numerator of 1st fractional term
  null a.fract => 0$R
  (first a.fract).num
numberOfFractionalTerms a == # a.fract
nthFractionalTerm(a,n) ==
  l : LfTerm := a.fract
  (n < 1) or (n > # l) => 0$%
  [0$R,[l.n]$LfTerm]$$
wholePart a == a.whole

partialFraction(nm: R, dn : FRR) ==
  nm = 0$R => 0$%
  -- move inv unit of den to numerator
  u : R := unit dn
  u := (recip u) :: R
  partialFractionNormalized(u * nm,u * dn)

```

```

padicallyExpand(p : R, r : R) ==
  -- expands r as a sum of powers of p, with coefficients
  -- r = HornerEval(padicallyExpand(p,r),p)
  qr : QR := divide(r, p)
  qr.quotient = 0$R => qr.remainder :: SUPR
  (qr.remainder :: SUPR) + monomial(1$R,1$NonNegativeInteger)$SUPR *
    padicallyExpand(p,qr.quotient)

a = b ==
  a.whole ^= b.whole => false -- must verify this
  (null a.fract) =>
    null b.fract => a.whole = b.whole
    false
  null b.fract => false
  -- oh, no! following is temporary
  (a :: Fraction R) = (b :: Fraction R)

- a ==
  s: fTerm
  l: LfTerm := nil
  for s in reverse a.fract repeat l := cons([- s.num,s.den]$fTerm,l)
  [- a.whole,l]

r * a ==
  r = 0$R => 0$%
  r = 1$R => a
  b : % := (r * a.whole) :: %
  c : %
  s : fTerm
  for s in reverse a.fract repeat
    c := normalizeFracTerm [r * s.num, s.den]$fTerm
    b.whole := b.whole + c.whole
    not (null c.fract) => b.fract := append(c.fract, b.fract)
  b

n * a == (n :: R) * a

a + b ==
  compactFraction
  [a.whole + b.whole,
   sort(LessThan,append(a.fract,copy b.fract))]]$%

a * b ==
  null a.fract => a.whole * b
  null b.fract => b.whole * a
  af : % := [0$R, a.fract]$% -- a - a.whole
  c : % := (a.whole * b) + (b.whole * af)
  s,t : fTerm
  for s in a.fract repeat

```

```

      for t in b.fract repeat
        c := c + multiplyFracTerms(s,t)
      c

coerce(a): Ex ==
  null a.fract => a.whole :: Ex
  s : fTerm
  l : List Ex
  if a.whole = 0 then l := nil else l := [a.whole :: Ex]
  for s in a.fract repeat
    s.den = 1$FRR => l := cons(s.num :: Ex, l)
    l := cons(s.num :: Ex / s.den :: Ex, l)
  # l = 1 => first l
  reduce("+", reverse l)

```

— PFR.dotabb —

```

"PFR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PFR"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"PFR" -> "FIELD"
"PFR" -> "FLAGG-"
"PFR" -> "FLAGG"

```

domain PARTITION Partition

— Partition.input —

```

)set break resume
)sys rm -f Partition.output
)spool Partition.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Partition
--R
--R Partition is a domain constructor

```

```

--R Abbreviation for Partition is PRITITION
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PRITITION
--R
--R----- Operations -----
--R ?? : (PositiveInteger,%) -> %      ?+? : (%,%) -> %
--R ?<? : (%,%) -> Boolean             ?<=? : (%,%) -> Boolean
--R ?=? : (%,%) -> Boolean             ?>? : (%,%) -> Boolean
--R ?>=? : (%,%) -> Boolean            0 : () -> %
--R coerce : % -> List(Integer)         coerce : % -> OutputForm
--R conjugate : % -> %                  convert : % -> List(Integer)
--R hash : % -> SingleInteger           latex : % -> String
--R max : (%,%) -> %                   min : (%,%) -> %
--R partition : List(Integer) -> %      pdct : % -> Integer
--R sample : () -> %                   zero? : % -> Boolean
--R ?~=? : (%,%) -> Boolean
--R ?? : (NonNegativeInteger,%) -> %
--R powers : List(Integer) -> List(List(Integer))
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R
--E 1

```

```

)spool
)lisp (bye)

```

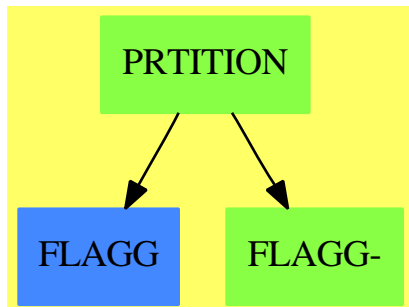
— Partition.help —

=====
Partition examples
=====

Domain for partitions of positive integers. Partition is an
OrderedCancellationAbelianMonoid which is used as the basis for
symmetric polynomial representation of the sums of powers in
SymmetricPolynomial. Thus, (5 2 2 1) will represent $s_5 * s_2^2 * s_1$.

See Also:
o)show Partition

Partition (PRTITION)



See

⇒ “SymmetricPolynomial” (SYMPOLY) [20](#) on page [2654](#)

Exports:

0	coerce	conjugate	convert	hash
latex	max	min	partition	pdct
powers	sample	subtractIfCan	zero?	?~=?
?*?	?+?	?<?	?<=?	?=?
?>?	?>=?			

— domain PRTITION Partition —

```

)abbrev domain PRTITION Partition
++ Author: William H. Burge
++ Date Created: 29 October 1987
++ Date Last Updated: 23 Sept 1991
++ Description:
++ Domain for partitive of positive integers
++ Partition is an OrderedCancellationAbelianMonoid which is used
++ as the basis for symmetric polynomial representation of the
++ sums of powers in SymmetricPolynomial. Thus, \spad{(5 2 2 1)} will
++ represent \spad{s5 * s2**2 * s1}.

```

Partition: Exports == Implementation where

```

L ==> List
I ==> Integer
OUT ==> OutputForm
NNI ==> NonNegativeInteger
UN ==> Union(%, "failed")

```

```

Exports ==> Join(OrderedCancellationAbelianMonoid,
                  ConvertibleTo List Integer) with
partition: L I -> %
++ partition(li) converts a list of integers li to a partition
powers: L I -> L L I
++ powers(li) returns a list of 2-element lists. For each 2-element

```

```

++ list, the first element is an entry of li and the second
++ element is the multiplicity with which the first element
++ occurs in li. There is a 2-element list for each value
++ occurring in l.
pdct: % -> I
++ \spad{pdct(a1**n1 a2**n2 ...)} returns
++ \spad{n1! * a1**n1 * n2! * a2**n2 * ...}.
++ This function is used in the package \spadtype{CycleIndicators}.
conjugate: % -> %
++ conjugate(p) returns the conjugate partition of a partition p
coerce: % -> List Integer
++ coerce(p) coerces a partition into a list of integers

Implementation ==> add

import PartitionsAndPermutations

Rep := List Integer
0 == nil()

coerce (s:%) == s pretend List Integer
convert x == copy(x pretend L I)

partition list == sort((i1:Integer,i2:Integer):Boolean +-> i2 < i1,list)

x < y ==
  empty? x => not empty? y
  empty? y => false
  first x = first y => rest x < rest y
  first x < first y

x = y ==
  EQUAL(x,y)$Lisp
-- empty? x => empty? y
-- empty? y => false
-- first x = first y => rest x = rest y
-- false

x + y ==
  empty? x => y
  empty? y => x
  first x > first y => concat(first x,rest(x) + y)
  concat(first y,x + rest(y))
n:NNI * x:% == (zero? n => 0; x + (subtractIfCan(n,1) :: NNI) * x)

dp: (I,%) -> %
dp(i,x) ==
  empty? x => 0
  first x = i => rest x
  concat(first x,dp(i,rest x))

```

```

remv: (I,%) -> UN
remv(i,x) == (member?(i,x) => dp(i,x); "failed")

subtractIfCan(x, y) ==
  empty? x =>
    empty? y => 0
    "failed"
  empty? y => x
  (aa := remv(first y,x)) case "failed" => "failed"
  subtractIfCan((aa :: %), rest y)

li1 : L I --!! 'bite' won't compile without this
bite: (I,L I) -> L I
bite(i,li) ==
  empty? li => concat(0,nil())
  first li = i =>
    li1 := bite(i,rest li)
    concat(first(li1) + 1,rest li1)
  concat(0,li)

li : L I --!! 'powers' won't compile without this
powers l ==
  empty? l => nil()
  li := bite(first l,rest l)
  concat([first l,first(li) + 1],powers(rest li))

conjugate x == conjugate(x pretend Rep)$PartitionsAndPermutations

mkterm: (I,I) -> OUT
mkterm(i1,i2) ==
  i2 = 1 => (i1 :: OUT) ** (" " :: OUT)
  (i1 :: OUT) ** (i2 :: OUT)

mkexp1: L L I -> L OUT
mkexp1 lli ==
  empty? lli => nil()
  li := first lli
  empty?(rest lli) and second(li) = 1 =>
    concat(first(li) :: OUT,nil())
  concat(mkterm(first li,second li),mkexp1(rest lli))

coerce(x:%):OUT ==
  empty? (x pretend Rep) => coerce(x pretend Rep)$Rep
  paren(reduce("*",mkexp1(powers(x pretend Rep))))

pdct x ==
  */[factorial(second a) * (first(a) ** (second(a) pretend NNI))
    for a in powers(x pretend Rep)]

```

— PRTITION.dotabb —

```
"PRTITION" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PRTITION"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"PRTITION" -> "FLAGG-"
"PRTITION" -> "FLAGG"
```

domain PATTERN Pattern

— Pattern.input —

```
)set break resume
)sys rm -f Pattern.output
)spool Pattern.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Pattern
--R
--R Pattern(R: SetCategory) is a domain constructor
--R Abbreviation for Pattern is PATTERN
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PATTERN
--R
--R----- Operations -----
--R ??? : (%,%) -> %          ??? : (%,%) -> %
--R ?+? : (%,%) -> %          ?/? : (%,%) -> %
--R ?? : (%,%) -> Boolean     1 : () -> %
--R 0 : () -> %              addBadValue : (%,Any) -> %
--R coerce : Symbol -> %      coerce : R -> %
--R coerce : % -> OutputForm  constant? : % -> Boolean
--R convert : List(%) -> %    copy : % -> %
--R depth : % -> NonNegativeInteger generic? : % -> Boolean
--R getBadValues : % -> List(Any) hasPredicate? : % -> Boolean
--R hasTopPredicate? : % -> Boolean hash : % -> SingleInteger
--R inR? : % -> Boolean        latex : % -> String
--R multiple? : % -> Boolean   optional? : % -> Boolean
--R predicates : % -> List(Any) quoted? : % -> Boolean
```

```

--R resetBadValues : % -> %                retract : % -> Symbol
--R retract : % -> R                      symbol? : % -> Boolean
--R variables : % -> List(%)              ?~=? : (%,% ) -> Boolean
--R ?**? : (% ,NonNegativeInteger) -> %
--R elt : (BasicOperator,List(%)) -> %
--R isExpt : % -> Union(Record(val: %,exponent: NonNegativeInteger),"failed")
--R isList : % -> Union(List(%),"failed")
--R isOp : % -> Union(Record(op: BasicOperator,arg: List(%)),"failed")
--R isOp : (% ,BasicOperator) -> Union(List(%),"failed")
--R isPlus : % -> Union(List(%),"failed")
--R isPower : % -> Union(Record(val: %,exponent: %),"failed")
--R isQuotient : % -> Union(Record(num: %,den: %),"failed")
--R isTimes : % -> Union(List(%),"failed")
--R optpair : List(%) -> Union(List(%),"failed")
--R patternVariable : (Symbol,Boolean,Boolean,Boolean) -> %
--R retractIfCan : % -> Union(Symbol,"failed")
--R retractIfCan : % -> Union(R,"failed")
--R setPredicates : (% ,List(Any)) -> %
--R setTopPredicate : (% ,List(Symbol),Any) -> %
--R topPredicate : % -> Record(var: List(Symbol),pred: Any)
--R withPredicates : (% ,List(Any)) -> %
--R
--E 1

```

```

)spool
)lisp (bye)

```

— Pattern.help —

=====

Pattern examples

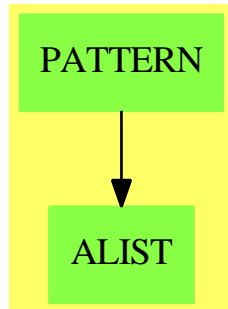
=====

Patterns for use by the pattern matcher.

See Also:

o)show Pattern

Pattern (PATTERN)



Exports:

0	1	addBadValue	coerce	constant?
convert	copy	depth	elt	generic?
getBadValues	hasPredicate?	hasTopPredicate?	hash	inR?
isExpt	isList	isOp	isOp	isPlus
isPower	isQuotient	isTimes	latex	multiple?
optional?	optpair	patternVariable	predicates	quoted?
resetBadValues	retract	retractIfCan	setPredicates	setTopPredicate
symbol?	topPredicate	variables	withPredicates	?*?
?**?	?+?	?/?	?=?	?~=?

— domain PATTERN Pattern —

```

)abbrev domain PATTERN Pattern
++ Author: Manuel Bronstein
++ Date Created: 10 Nov 1988
++ Date Last Updated: 20 June 1991
++ Description:
++ Patterns for use by the pattern matcher.
-- Not exposed.
-- Patterns are optimized for quick answers to structural questions.

```

```

Pattern(R:SetCategory): Exports == Implementation where
B  ==> Boolean
SI ==> SingleInteger
Z  ==> Integer
SY ==> Symbol
O  ==> OutputForm
BOP ==> BasicOperator
QOT ==> Record(num:%, den:%)
REC ==> Record(val:%, exponent:NonNegativeInteger)
RSY ==> Record(tag:SI, val: SY, pred:List Any, bad:List Any)
KER ==> Record(tag:SI, op:BOP, arg:List %)
PAT ==> Union(ret:R, ker: KER, exp:REC, qot: QOT, sym:RSY)

```

```

-- the following MUST be the name of the formal exponentiation operator
POWER ==> "%power"::Symbol

-- the 4 SYM_ constants must be disting powers of 2 (bitwise arithmetic)
SYM_GENERIC ==> 1::SI
SYM_MULTIPLE ==> 2::SI
SYM_OPTIONAL ==> 4::SI

PAT_PLUS ==> 1::SI
PAT_TIMES ==> 2::SI
PAT_LIST ==> 3::SI
PAT_ZERO ==> 4::SI
PAT_ONE ==> 5::SI
PAT_EXPT ==> 6::SI

Exports ==> Join(SetCategory, RetractableTo R, RetractableTo SY) with
0          : constant -> %          ++ 0
1          : constant -> %          ++ 1
isPlus     : % -> Union(List %, "failed")
  ++ isPlus(p) returns \spad{[a1,...,an]} if \spad{n > 1}
  ++ and \spad{p = a1 + ... + an},
  ++ and "failed" otherwise.
isTimes    : % -> Union(List %, "failed")
  ++ isTimes(p) returns \spad{[a1,...,an]} if \spad{n > 1} and
  ++ \spad{p = a1 * ... * an}, and
  ++ "failed" otherwise.
isOp       : (% , BOP) -> Union(List %, "failed")
  ++ isOp(p, op) returns \spad{[a1,...,an]} if \spad{p = op(a1,...,an)}, and
  ++ "failed" otherwise.
isOp       : % -> Union(Record(op:BOP, arg:List %), "failed")
  ++ isOp(p) returns \spad{[op, [a1,...,an]]} if
  ++ \spad{p = op(a1,...,an)}, and
  ++ "failed" otherwise;
isExpt     : % -> Union(REC, "failed")
  ++ isExpt(p) returns \spad{[q, n]} if \spad{n > 0} and \spad{p = q ** n},
  ++ and "failed" otherwise.
isQuotient : % -> Union(QOT, "failed")
  ++ isQuotient(p) returns \spad{[a, b]} if \spad{p = a / b}, and
  ++ "failed" otherwise.
isList     : % -> Union(List %, "failed")
  ++ isList(p) returns \spad{[a1,...,an]} if \spad{p = [a1,...,an]},
  ++ "failed" otherwise;
isPower    : % -> Union(Record(val:%, exponent:%), "failed")
  ++ isPower(p) returns \spad{[a, b]} if \spad{p = a ** b}, and
  ++ "failed" otherwise.
elt        : (BOP, List %) -> %
  ++ \spad{elt(op, [a1,...,an])} returns \spad{op(a1,...,an)}.
"+"        : (% , %) -> %
  ++ \spad{a + b} returns the pattern \spad{a + b}.
"*"        : (% , %) -> %

```

```

++ \spad{a * b} returns the pattern \spad{a * b}.
"*)" : (% , NonNegativeInteger) -> %
++ \spad{a ** n} returns the pattern \spad{a ** n}.
"*)" : (% , %) -> %
++ \spad{a ** b} returns the pattern \spad{a ** b}.
"/" : (% , %) -> %
++ \spad{a / b} returns the pattern \spad{a / b}.
depth : % -> NonNegativeInteger
++ depth(p) returns the nesting level of p.
convert : List % -> %
++ \spad{convert([a1,...,an])} returns the pattern \spad{[a1,...,an]}.
copy : % -> %
++ copy(p) returns a recursive copy of p.
inR? : % -> B
++ inR?(p) tests if p is an atom (i.e. an element of R).
quoted? : % -> B
++ quoted?(p) tests if p is of the form 's for a symbol s.
symbol? : % -> B
++ symbol?(p) tests if p is a symbol.
constant? : % -> B
++ constant?(p) tests if p contains no matching variables.
generic? : % -> B
++ generic?(p) tests if p is a single matching variable.
multiple? : % -> B
++ multiple?(p) tests if p is a single matching variable
++ allowing list matching or multiple term matching in a
++ sum or product.
optional? : % -> B
++ optional?(p) tests if p is a single matching variable
++ which can match an identity.
hasPredicate?: % -> B
++ hasPredicate?(p) tests if p has predicates attached to it.
predicates : % -> List Any
++ predicates(p) returns \spad{[p1,...,pn]} such that the predicate
++ attached to p is p1 and ... and pn.
setPredicates: (% , List Any) -> %
++ \spad{setPredicates(p, [p1,...,pn])} attaches the predicate
++ p1 and ... and pn to p.
withPredicates:(% , List Any) -> %
++ \spad{withPredicates(p, [p1,...,pn])} makes a copy of p and attaches
++ the predicate p1 and ... and pn to the copy, which is
++ returned.
patternVariable: (SY, B, B, B) -> %
++ patternVariable(x, c?, o?, m?) creates a pattern variable x,
++ which is constant if \spad{c? = true}, optional if \spad{o? = true},
++ and multiple if \spad{m? = true}.
setTopPredicate: (% , List SY, Any) -> %
++ \spad{setTopPredicate(x, [a1,...,an], f)} returns x with
++ the top-level predicate set to \spad{f(a1,...,an)}.
topPredicate: % -> Record(var:List SY, pred:Any)

```

```

    ++ topPredicate(x) returns \spad{[[a1,...,an], f]} where the top-level
    ++ predicate of x is \spad{f(a1,...,an)}.
    ++ Note: n is 0 if x has no top-level
    ++ predicate.
hasTopPredicate?: % -> B
    ++ hasTopPredicate?(p) tests if p has a top-level predicate.
resetBadValues: % -> %
    ++ resetBadValues(p) initializes the list of "bad values" for p
    ++ to \spad{[]}.
    ++ Note: p is not allowed to match any of its "bad values".
addBadValue: (% , Any) -> %
    ++ addBadValue(p, v) adds v to the list of "bad values" for p.
    ++ Note: p is not allowed to match any of its "bad values".
getBadValues: % -> List Any
    ++ getBadValues(p) returns the list of "bad values" for p.
    ++ Note: p is not allowed to match any of its "bad values".
variables: % -> List %
    ++ variables(p) returns the list of matching variables
    ++ appearing in p.
optpair: List % -> Union(List %, "failed")
    ++ optpair(l) returns l has the form \spad{[a, b]} and
    ++ a is optional, and
    ++ "failed" otherwise;

Implementation ==> add
Rep := Record(cons?: B, pat:PAT, lev: NonNegativeInteger,
              topvar: List SY, toppred: Any)

dummy:BOP := operator(new())$Symbol
nopred    := coerce(0$Integer)$AnyFunctions1(Integer)

mkPat      : (B, PAT, NonNegativeInteger) -> %
mkrsy      : (SY, B, B, B) -> RSY
SYM20      : RSY -> 0
PAT20      : PAT -> 0
patcopy    : PAT -> PAT
bitSet?    : (SI , SI) -> B
pateq?     : (PAT, PAT) -> B
LPAT20     : ((0, 0) -> 0, List %) -> 0
taggedElt  : (SI, List %) -> %
isTaggedOp : (% , SI) -> Union(List %, "failed")
incmax     : List % -> NonNegativeInteger

coerce(r:R):% == mkPat(true, [r], 0)
mkPat(c, p, l) == [c, p, l, empty(), nopred]
hasTopPredicate? x == not empty?(x.topvar)
topPredicate x == [x.topvar, x.toppred]
setTopPredicate(x, l, f) == (x.topvar := l; x.toppred := f; x)
constant? p == p.cons?
depth p == p.lev

```

```

inR? p          == p.pat case ret
symbol? p       == p.pat case sym
isPlus p        == isTaggedOp(p, PAT_PLUS)
isTimes p       == isTaggedOp(p, PAT_TIMES)
isList p        == isTaggedOp(p, PAT_LIST)
isExpt p        == (p.pat case exp => p.pat.exp; "failed")
isQuotient p    == (p.pat case qot => p.pat.qot; "failed")
hasPredicate? p == not empty? predicates p
quoted? p       == symbol? p and zero?(p.pat.sym.tag)
generic? p      == symbol? p and bitSet?(p.pat.sym.tag, SYM_GENERIC)
multiple? p     == symbol? p and bitSet?(p.pat.sym.tag, SYM_MULTIPLE)
optional? p     == symbol? p and bitSet?(p.pat.sym.tag, SYM_OPTIONAL)
bitSet?(a, b)   == And(a, b) ^= 0
coerce(p:%):0   == PAT20(p.pat)
p1:% ** p2:%    == taggedElt(PAT_EXPT, [p1, p2])
LPAT20(f, l)    == reduce(f, [x::0 for x in l])$List(0)
retract(p:%):R == (inR? p => p.pat.ret; error "Not retractable")
convert(l:List %):% == taggedElt(PAT_LIST, l)
retractIfCan(p:%):Union(R,"failed") == (inR? p => p.pat.ret;"failed")
withPredicates(p, l) == setPredicates(copy p, l)
coerce(sy:SY):% == patternVariable(sy, false, false, false)
copy p == [constant? p, patcopy(p.pat), p.lev, p.topvar, p.toppred]

-- returns [a, b] if #l = 2 and optional? a, "failed" otherwise
optpair l ==
  empty? rest rest l =>
    b := first rest l
    optional?(a := first l) => l
    optional? b => reverse l
    "failed"
    "failed"

incmax l ==
  1 + reduce("max", [p.lev for p in l], 0)$List(NonNegativeInteger)

p1 = p2 ==
  (p1.cons? = p2.cons?) and (p1.lev = p2.lev) and
  (p1.topvar = p2.topvar) and
  ((EQ(p1.toppred, p2.toppred)$Lisp) pretend B) and
  pateq?(p1.pat, p2.pat)

isPower p ==
  (u := isTaggedOp(p, PAT_EXPT)) case "failed" => "failed"
  [first(u::List(%)), second(u::List(%))]

taggedElt(n, l) ==
  mkPat(every?(constant?, l), [[n, dummy, l]$KER], incmax l)

elt(o, l) ==
  is?(o, POWER) and #l = 2 => first(l) ** last(l)

```

```

mkPat(every?(constant?, 1), [[0, o, 1]$KER], incmax 1)

isOp p ==
  (p.pat case ker) and zero?(p.pat.ker.tag) =>
    [p.pat.ker.op, p.pat.ker.arg]
  "failed"

isTaggedOp(p,t) ==
  (p.pat case ker) and (p.pat.ker.tag = t) => p.pat.ker.arg
  "failed"

if R has Monoid then
  1 == 1::R::%
else
  1 == taggedElt(PAT_ONE, empty())

if R has AbelianMonoid then
  0 == 0::R::%
else
  0 == taggedElt(PAT_ZERO, empty())

p:% ** n:NonNegativeInteger ==
  p = 0 and n > 0 => 0
  p = 1 or zero? n => 1
--  one? n => p
  (n = 1) => p
  mkPat(constant? p, [[p, n]$REC], 1 + (p.lev))

p1 / p2 ==
  p2 = 1 => p1
  mkPat(constant? p1 and constant? p2, [[p1, p2]$QOT],
    1 + max(p1.lev, p2.lev))

p1 + p2 ==
  p1 = 0 => p2
  p2 = 0 => p1
  (u1 := isPlus p1) case List(%) =>
    (u2 := isPlus p2) case List(%) =>
      taggedElt(PAT_PLUS, concat(u1::List %, u2::List %))
      taggedElt(PAT_PLUS, concat(u1::List %, p2))
  (u2 := isPlus p2) case List(%) =>
    taggedElt(PAT_PLUS, concat(p1, u2::List %))
  taggedElt(PAT_PLUS, [p1, p2])

p1 * p2 ==
  p1 = 0 or p2 = 0 => 0
  p1 = 1 => p2
  p2 = 1 => p1
  (u1 := isTimes p1) case List(%) =>
    (u2 := isTimes p2) case List(%) =>

```



```

    taggedElt(PAT_TIMES, concat(u1::List %, u2::List %))
    taggedElt(PAT_TIMES, concat(u1::List %, p2))
(u2 := isTimes p2) case List(%) =>
    taggedElt(PAT_TIMES, concat(p1, u2::List %))
taggedElt(PAT_TIMES, [p1, p2])

isOp(p, o) ==
    (p.pat case ker) and zero?(p.pat.ker.tag) and (p.pat.ker.op =o) =>
        p.pat.ker.arg
    "failed"

predicates p ==
    symbol? p => p.pat.sym.pred
    empty()

setPredicates(p, l) ==
    generic? p => (p.pat.sym.pred := l; p)
    error "Can only attach predicates to generic symbol"

resetBadValues p ==
    generic? p => (p.pat.sym.bad := empty()$List(Any); p)
    error "Can only attach bad values to generic symbol"

addBadValue(p, a) ==
    generic? p =>
        if not member?(a, p.pat.sym.bad) then
            p.pat.sym.bad := concat(a, p.pat.sym.bad)
        p
    error "Can only attach bad values to generic symbol"

getBadValues p ==
    generic? p => p.pat.sym.bad
    error "Not a generic symbol"

SYM20 p ==
    sy := (p.val)::0
    empty?(p.pred) => sy
    paren infix(" | " :: 0, sy,
        reduce("and", [sub("f"::0, i::0) for i in 1..#(p.pred)])$List(0))

variables p ==
    constant? p => empty()
    generic? p => [p]
    q := p.pat
    q case ret => empty()
    q case exp => variables(q.exp.val)
    q case qot => concat_!(variables(q.qot.num), variables(q.qot.den))
    q case ker => concat [variables r for r in q.ker.arg]
    empty()

```

```

PAT20 p ==
  p case ret => (p.ret)::0
  p case sym => SYM20(p.sym)
  p case exp => (p.exp.val)::0 ** (p.exp.exponent)::0
  p case qot => (p.qot.num)::0 / (p.qot.den)::0
  p.ker.tag = PAT_PLUS => LPAT20("+", p.ker.arg)
  p.ker.tag = PAT_TIMES => LPAT20("*", p.ker.arg)
  p.ker.tag = PAT_LIST => (p.ker.arg)::0
  p.ker.tag = PAT_ZERO => 0::Integer::0
  p.ker.tag = PAT_ONE => 1::Integer::0
  l := [x::0 for x in p.ker.arg]$List(0)
  (u:=display(p.ker.op)) case "failed" =>prefix(name(p.ker.op)::0,l)
  (u::(List 0 -> 0)) 1

patcopy p ==
  p case ret => [p.ret]
  p case sym =>
    [[p.sym.tag, p.sym.val, copy(p.sym.pred), copy(p.sym.bad)]$RSY]
  p case ker=>[[p.ker.tag,p.ker.op,[copy x for x in p.ker.arg]]$KER]
  p case qot => [[copy(p.qot.num), copy(p.qot.den)]$QOT]
  [[copy(p.exp.val), p.exp.exponent]$REC]

pateq?(p1, p2) ==
  p1 case ret => (p2 case ret) and (p1.ret = p2.ret)
  p1 case qot =>
    (p2 case qot) and (p1.qot.num = p2.qot.num)
    and (p1.qot.den = p2.qot.den)
  p1 case sym =>
    (p2 case sym) and (p1.sym.val = p2.sym.val)
    and {p1.sym.pred} = $Set(Any) {p2.sym.pred}
    and {p1.sym.bad} = $Set(Any) {p2.sym.bad}
  p1 case ker =>
    (p2 case ker) and (p1.ker.tag = p2.ker.tag)
    and (p1.ker.op = p2.ker.op) and (p1.ker.arg = p2.ker.arg)
  (p2 case exp) and (p1.exp.exponent = p2.exp.exponent)
  and (p1.exp.val = p2.exp.val)

retractIfCan(p:%):Union(SY, "failed") ==
  symbol? p => p.pat.sym.val
  "failed"

mkrsy(t, c?, o?, m?) ==
  c? => [0, t, empty(), empty()]
  mlt := (m? => SYM_MULTIPLE; 0)
  opt := (o? => SYM_OPTIONAL; 0)
  [Or(Or(SYM_GENERIC, mlt), opt), t, empty(), empty()]

patternVariable(sy, c?, o?, m?) ==
  rsy := mkrsy(sy, c?, o?, m?)
  mkPat(zero?(rsy.tag), [rsy], 0)

```

— PATTERN.dotabb —

```
"PATTERN" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PATTERN"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"PATTERN" -> "ALIST"
```

domain PATLRES PatternMatchListResult

— PatternMatchListResult.input —

```
)set break resume
)sys rm -f PatternMatchListResult.output
)spool PatternMatchListResult.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternMatchListResult
--R
--R PatternMatchListResult(R: SetCategory,S: SetCategory,L: ListAggregate(S)) is a domain const
--R Abbreviation for PatternMatchListResult is PATLRES
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PATLRES
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean               coerce : % -> OutputForm
--R failed : () -> %                      failed? : % -> Boolean
--R hash : % -> SingleInteger             latex : % -> String
--R new : () -> %                        ?~=? : (%,% ) -> Boolean
--R atoms : % -> PatternMatchResult(R,S)
--R lists : % -> PatternMatchResult(R,L)
--R makeResult : (PatternMatchResult(R,S),PatternMatchResult(R,L)) -> %
--R
--E 1

)spool
)lisp (bye)
```

— PatternMatchListResult.help —

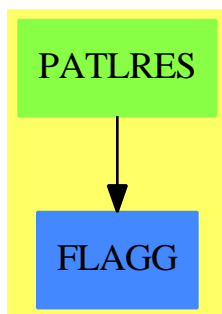
```
=====
PatternMatchListResult examples
=====
```

A PatternMatchListResult is an object internally returned by the pattern matcher when matching on lists. It is either a failed match, or a pair of PatternMatchResult, one for atoms (elements of the list), and one for lists.

See Also:

o)show PatternMatchListResult

PatternMatchListResult (PATLRES)



See

⇒ “PatternMatchResult” (PATRES) [17](#) on page [1940](#)

Exports:

atoms	coerce	failed	failed?	hash
latex	lists	makeResult	new	?=?
?~=?				

— domain PATLRES PatternMatchListResult —

```
)abbrev domain PATLRES PatternMatchListResult
++ Author: Manuel Bronstein
++ Date Created: 4 Dec 1989
++ Date Last Updated: 4 Dec 1989
++ Description:
```

```

++ A PatternMatchListResult is an object internally returned by the
++ pattern matcher when matching on lists.
++ It is either a failed match, or a pair of PatternMatchResult,
++ one for atoms (elements of the list), and one for lists.
-- not exported

PatternMatchListResult(R:SetCategory, S:SetCategory, L:ListAggregate S):
  SetCategory with
    failed?   : % -> Boolean
    ++ failed?(r) tests if r is a failed match.
    failed    : () -> %
    ++ failed() returns a failed match.
    new       : () -> %
    ++ new() returns a new empty match result.
    makeResult: (PatternMatchResult(R,S), PatternMatchResult(R,L)) -> %
    ++ makeResult(r1,r2) makes the combined result [r1,r2].
    atoms     : % -> PatternMatchResult(R, S)
    ++ atoms(r) returns the list of matches that match atoms
    ++ (elements of the lists).
    lists     : % -> PatternMatchResult(R, L)
    ++ lists(r) returns the list of matches that match lists.
== add
Rep := Record(a:PatternMatchResult(R, S), l:PatternMatchResult(R, L))

new()           == [new(), new()]
atoms r         == r.a
lists r         == r.l
failed()        == [failed(), failed()]
failed? r       == failed?(atoms r)
x = y           == (atoms x = atoms y) and (lists x = lists y)

makeResult(r1, r2) ==
  failed? r1 or failed? r2 => failed()
  [r1, r2]

coerce(r:%):OutputForm ==
  failed? r => atoms(r)::OutputForm
  RecordPrint(r, Rep)$Lisp

```

— PATLRES.dotabb —

```

"PATLRES" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PATLRES"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PATLRES" -> "FLAGG"

```

domain PATRES PatternMatchResult

— PatternMatchResult.input —

```

)set break resume
)sys rm -f PatternMatchResult.output
)spool PatternMatchResult.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternMatchResult
--R
--R PatternMatchResult(R: SetCategory,S: SetCategory) is a domain constructor
--R Abbreviation for PatternMatchResult is PATRES
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PATRES
--R
--R----- Operations -----
--R ?? : (%,% ) -> Boolean          addMatch : (Pattern(R),S,% ) -> %
--R coerce : % -> OutputForm        failed : () -> %
--R failed? : % -> Boolean          hash : % -> SingleInteger
--R latex : % -> String             new : () -> %
--R union : (%,% ) -> %            ~=? : (%,% ) -> Boolean
--R addMatchRestricted : (Pattern(R),S,%,S) -> %
--R construct : List(Record(key: Symbol,entry: S)) -> %
--R destruct : % -> List(Record(key: Symbol,entry: S))
--R getMatch : (Pattern(R),%) -> Union(S,"failed")
--R insertMatch : (Pattern(R),S,% ) -> %
--R satisfy? : (%,Pattern(R)) -> Union(Boolean,"failed")
--R
--E 1

)spool
)lisp (bye)

```

— PatternMatchResult.help —

```

=====
PatternMatchResult examples
=====

```

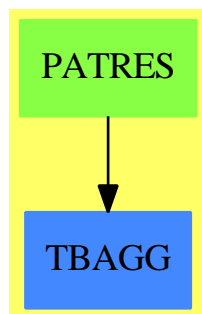
A PatternMatchResult is an object internally returned by the pattern matcher; It is either a failed match, or a list of matches of the form (var, expr) meaning that the variable var

matches the expression `expr`.

See Also:

o `)show PatternMatchResult`

PatternMatchResult (PATRES)



See

⇒ “PatternMatchListResult” (PATLRES) [17](#) on page [1937](#)

Exports:

<code>addMatch</code>	<code>addMatchRestricted</code>	<code>coerce</code>	<code>construct</code>	<code>destruct</code>
<code>failed</code>	<code>failed?</code>	<code>getMatch</code>	<code>hash</code>	<code>insertMatch</code>
<code>latex</code>	<code>new</code>	<code>satisfy?</code>	<code>union</code>	<code>?=?</code>
<code>?~=?</code>				

— domain PATRES PatternMatchResult —

```

)abbrev domain PATRES PatternMatchResult
++ Author: Manuel Bronstein
++ Date Created: 28 Nov 1989
++ Date Last Updated: 5 Jul 1990
++ Description:
++ A PatternMatchResult is an object internally returned by the
++ pattern matcher; It is either a failed match, or a list of
++ matches of the form (var, expr) meaning that the variable var
++ matches the expression expr.
-- not exported

PatternMatchResult(R:SetCategory, S:SetCategory): SetCategory with
  failed?          : % -> Boolean
  ++ failed?(r) tests if r is a failed match.
  failed          : () -> %
  ++ failed() returns a failed match.

```

```

new          : () -> %
  ++ new() returns a new empty match result.
union        : (%, %) -> %
  ++ union(a, b) makes the set-union of two match results.
getMatch     : (Pattern R, %) -> Union(S, "failed")
  ++ getMatch(var, r) returns the expression that var matches
  ++ in the result r, and "failed" if var is not matched in r.
addMatch     : (Pattern R, S, %) -> %
  ++ addMatch(var, expr, r) adds the match (var, expr) in r,
  ++ provided that expr satisfies the predicates attached to var,
  ++ and that var is not matched to another expression already.
insertMatch  : (Pattern R, S, %) -> %
  ++ insertMatch(var, expr, r) adds the match (var, expr) in r,
  ++ without checking predicates or previous matches for var.
addMatchRestricted: (Pattern R, S, %, S) -> %
  ++ addMatchRestricted(var, expr, r, val) adds the match
  ++ (var, expr) in r,
  ++ provided that expr satisfies the predicates attached to var,
  ++ that var is not matched to another expression already,
  ++ and that either var is an optional pattern variable or that
  ++ expr is not equal to val (usually an identity).
destruct     : % -> List Record(key:Symbol, entry:S)
  ++ destruct(r) returns the list of matches (var, expr) in r.
  ++ Error: if r is a failed match.
construct    : List Record(key:Symbol, entry:S) -> %
  ++ construct([v1,e1],...,[vn,en]) returns the match result
  ++ containing the matches (v1,e1),...,(vn,en).
satisfy?     : (%, Pattern R) -> Union(Boolean, "failed")
  ++ satisfy?(r, p) returns true if the matches satisfy the
  ++ top-level predicate of p, false if they don't, and "failed"
  ++ if not enough variables of p are matched in r to decide.

== add
LR ==> AssociationList(Symbol, S)

import PatternFunctions1(R, S)

Rep := Union(LR, "failed")

new()          == empty()
failed()       == "failed"
failed? x      == x case "failed"
insertMatch(p, x, l) == concat([retract p, x], l::LR)
construct l    == construct(l)$LR
destruct l     == entries(l::LR)$LR

-- returns "failed" if not all the variables of the pred. are matched
satisfy?(r, p) ==
  failed? r => false
  lr := r::LR

```



```

lv := [if (u := search(v, lr)) case "failed" then return "failed"
      else u::S for v in topPredicate(p).var]$List(S)

satisfy?(lv, p)

union(x, y) ==
  failed? x or failed? y => failed()
  removeDuplicates concat(x::LR, y::LR)

x = y ==
  failed? x => failed? y
  failed? y => false
  x::LR = $LR y::LR

coerce(x:%):OutputForm ==
  failed? x => "Does not match":OutputForm
  destruct(x)::OutputForm

addMatchRestricted(p, x, l, ident) ==
  (not optional? p) and (x = ident) => failed()
  addMatch(p, x, l)

addMatch(p, x, l) ==
  failed?(l) or not(satisfy?(x, p)) => failed()
  al := l::LR
  sy := retract(p)@Symbol
  (r := search(sy, al)) case "failed" => insertMatch(p, x, l)
  r::S = x => l
  failed()

getMatch(p, l) ==
  failed? l => "failed"
  search(retract(p)@Symbol, l::LR)

```

— PATRES.dotabb —

```

"PATRES" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PATRES"]
"TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]
"PATRES" -> "TBAGG"

```

domain PENDTREE PendantTree

— PendantTree.input —

```

)set break resume
)sys rm -f PendantTree.output
)spool PendantTree.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PendantTree
--R
--R PendantTree(S: SetCategory) is a domain constructor
--R Abbreviation for PendantTree is PENDTREE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PENDTREE
--R
--R----- Operations -----
--R children : % -> List(%)          coerce : % -> Tree(S)
--R copy : % -> %                    cyclic? : % -> Boolean
--R distance : (%,%) -> Integer      ?.right : (%,right) -> %
--R ?.left : (%,left) -> %          ?.value : (%,value) -> S
--R empty : () -> %                 empty? : % -> Boolean
--R eq? : (%,%) -> Boolean          leaf? : % -> Boolean
--R leaves : % -> List(S)          left : % -> %
--R map : ((S -> S),%) -> %         nodes : % -> List(%)
--R ptree : (%,%) -> %             ptree : S -> %
--R right : % -> %                 sample : () -> %
--R value : % -> S
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ==? : (%,%) -> Boolean if S has SETCAT
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R child? : (%,%) -> Boolean if S has SETCAT
--R coerce : % -> OutputForm if S has SETCAT
--R count : (S,%) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R eval : (%,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R hash : % -> SingleInteger if S has SETCAT
--R latex : % -> String if S has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R member? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R members : % -> List(S) if $ has finiteAggregate
--R more? : (%,NonNegativeInteger) -> Boolean
--R node? : (%,%) -> Boolean if S has SETCAT
--R parts : % -> List(S) if $ has finiteAggregate

```

```

--R setchildren! : (% , List(%)) -> % if $ has shallowlyMutable
--R setelt : (% , right, %) -> % if $ has shallowlyMutable
--R setelt : (% , left, %) -> % if $ has shallowlyMutable
--R setelt : (% , value, S) -> S if $ has shallowlyMutable
--R setleft! : (% , %) -> % if $ has shallowlyMutable
--R setright! : (% , %) -> % if $ has shallowlyMutable
--R setvalue! : (% , S) -> S if $ has shallowlyMutable
--R size? : (% , NonNegativeInteger) -> Boolean
--R ?~=? : (% , %) -> Boolean if S has SETCAT
--R
--E 1

)spool
)lisp (bye)

```

— PendantTree.help —

```

=====
PendantTree examples
=====

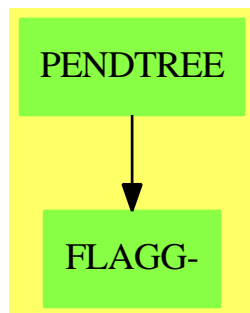
```

A PendantTree(S) is either a leaf? and is an S or has
a left and a right both PendantTree(S)'s

See Also:
o)show PendantTree

A PendantTree(S) is either a leaf? and is an S or has a left and a right both PendantTree(S)'s

PendantTree (PENDTREE)



See

⇒ “Tree” (TREE) [21](#) on page [2740](#)
 ⇒ “BinaryTree” (BTREE) [3](#) on page [310](#)
 ⇒ “BinarySearchTree” (BSTREE) [3](#) on page [302](#)
 ⇒ “BinaryTournament” (BTourn) [3](#) on page [306](#)
 ⇒ “BalancedBinaryTree” (BBTREE) [3](#) on page [252](#)

Exports:

any?	child?	children	coerce	copy
count	cyclic?	distance	empty	empty?
eq?	eval	every?	hash	latex
leaf?	leaves	left	less?	map
map!	member?	members	more?	node?
nodes	parts	ptree	right	sample
setchildren!	setelt	setleft!	setright!	setvalue!
size?	value	#?	?=?	?~=?
?.right	?.left	?.value		

— domain PENDTREE PendantTree —

```

)abbrev domain PENDTREE PendantTree
++ Author: Mark Botch
++ Description:
++ A PendantTree(S) is either a leaf? and is an S or has
++ a left and a right both PendantTree(S)'s

PendantTree(S: SetCategory): T == C where
T == BinaryRecursiveAggregate(S) with
  ptree : S->%
    ++ ptree(s) is a leaf? pendant tree
    ++
    ++X t1:=ptree([1,2,3])

  ptree:(%, %)->%
    ++ ptree(x,y) is not documented
    ++
    ++X t1:=ptree([1,2,3])
    ++X ptree(t1,ptree([1,2,3]))

  coerce:%->Tree S
    ++ coerce(x) is not documented
    ++
    ++X t1:=ptree([1,2,3])
    ++X t2:=ptree(t1,ptree([1,2,3]))
    ++X t2::Tree List PositiveInteger

C == add
  Rep := Tree S
  import Tree S

```

```

coerce (t:%):Tree S == t pretend Tree S
ptree(n) == tree(n,[])$Rep pretend %
ptree(l,r) == tree(value(r:Rep)$Rep,cons(l,children(r:Rep)$Rep)):%
leaf? t == empty?(children(t)$Rep)
t1=t2 == (t1:Rep) = (t2:Rep)
left b ==
  leaf? b => error "ptree:no left"
  first(children(b)$Rep)
right b ==
  leaf? b => error "ptree:no right"
  tree(value(b)$Rep,rest (children(b)$Rep))
value b ==
  leaf? b => value(b)$Rep
  error "the pendant tree has no value"
coerce(b:%): OutputForm ==
  leaf? b => value(b)$Rep :: OutputForm
  paren blankSeparate [left b::OutputForm,right b ::OutputForm]

```

— PENDTREE.dotabb —

```

"PENDTREE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PENDTREE"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"PENDTREE" -> "FLAGG-"

```

domain PERM Permutation

— Permutation.input —

```

)set break resume
)sys rm -f Permutation.output
)spool Permutation.output
)set message test on
)set message auto off
)clear all
--S 1 of 8
p := coercePreimagesImages([[1,2,3],[1,2,3]])
--R
--R
--R (1)  1
--R
Type: Permutation(PositiveInteger)

```

[illegible]

[illegible]

```
q := coercePreimagesImages([[0,1,2,3],[1,0]])$PERM ZMOD 4
(1 0)
Type: Permutation IntegerMod 4

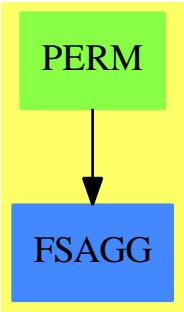
fixedPoints(p*q)
{2,0}
Type: Set IntegerMod 4

even?(p*q)
false
Type: Boolean

See Also:
o )show Permutation
```



Permutation (PERM)



Exports:		
1	coerce	coerceImages
coerceListOfPairs	coercePreimagesImages	commutator
conjugate	cycle	cyclePartition
cycles	degree	eval
even?	fixedPoints	hash
inv	latex	listRepresentation
max	min	movedPoints
numberOfCycles	odd?	one?
orbit	order	recip
sample	sign	sort
?*?	?**?	?/?
?<?	?=?	?^?
?..?	?~=?	?<=?
?>?	?>=?	

— domain PERM Permutation —

```

)abbrev domain PERM Permutation
++ Authors: Johannes Grabmeier, Holger Gollan, Martin Rubey
++ Date Created: 19 May 1989
++ Date Last Updated: 2 June 2006
++ Reference: G. James/A. Kerber: The Representation Theory of the Symmetric
++ Group. Encycl. of Math. and its Appl., Vol. 16., Cambridge
++ Description:
++ Permutation(S) implements the group of all bijections
++ on a set S, which move only a finite number of points.
++ A permutation is considered as a map from S into S. In particular
++ multiplication is defined as composition of maps:\br
++  $\pi_1 * \pi_2 = \pi_1 \circ \pi_2$ .\br
++ The internal representation of permutations are two lists
++ of equal length representing preimages and images.

Permutation(S:SetCategory): public == private where

B      ==> Boolean
PI     ==> PositiveInteger
I      ==> Integer
L      ==> List
NNI    ==> NonNegativeInteger
V      ==> Vector
PT     ==> Partition
OUTFORM ==> OutputForm
RECCYPE ==> Record(cycl: L L S, permut: %)
RECPRIM ==> Record(preimage: L S, image : L S)

public ==> PermutationCategory S with

listRepresentation: %                -> RECPRIM
++ listRepresentation(p) produces a representation rep of
++ the permutation p as a list of preimages and images, i.e
++ p maps (rep.preimage).k to (rep.image).k for all
++ indices k. Elements of \spad{S} not in (rep.preimage).k
++ are fixed points, and these are the only fixed points of the
++ permutation.
coercePreimagesImages : List List S -> %
++ coercePreimagesImages(lls) coerces the representation lls
++ of a permutation as a list of preimages and images to a permutation.
++ We assume that both preimage and image do not contain repetitions.
++
++X p := coercePreimagesImages([[1,2,3],[1,2,3]])
++X q := coercePreimagesImages([[0,1,2,3],[3,0,2,1]])$PERM ZMOD 4
coerce      : List List S -> %
++ coerce(lls) coerces a list of cycles lls to a
++ permutation, each cycle being a list with no

```

```

++ repetitions, is coerced to the permutation, which maps
++ ls.i to ls.i+1, indices modulo the length of the list,
++ then these permutations are multiplied.
++ Error: if repetitions occur in one cycle.
coerce      : List S      -> %
++ coerce(ls) coerces a cycle ls, i.e. a list with not
++ repetitions to a permutation, which maps ls.i to
++ ls.i+1, indices modulo the length of the list.
++ Error: if repetitions occur.
coerceListOfPairs : List List S      -> %
++ coerceListOfPairs(lls) coerces a list of pairs lls to a
++ permutation.
++ Error: if not consistent, i.e. the set of the first elements
++ coincides with the set of second elements.
--coerce      : %      -> OUTPUTFORM
++ coerce(p) generates output of the permutation p with domain
++ OutputForm.
degree      : %      -> NonNegativeInteger
++ degree(p) returns the number of points moved by the
++ permutation p.
movedPoints : %      -> Set S
++ movedPoints(p) returns the set of points moved by the permutation p.
++
++X p := coercePreimagesImages([[1,2,3],[1,2,3]])
++X movedPoints p
cyclePartition : %      -> Partition
++ cyclePartition(p) returns the cycle structure of a permutation
++ p including cycles of length 1 only if S is finite.
order      : %      -> NonNegativeInteger
++ order(p) returns the order of a permutation p as a group element.
numberOfCycles : %      -> NonNegativeInteger
++ numberOfCycles(p) returns the number of non-trivial cycles of
++ the permutation p.
sign      : %      -> Integer
++ sign(p) returns the signum of the permutation p, +1 or -1.
even?     : %      -> Boolean
++ even?(p) returns true if and only if p is an even permutation,
++ i.e. sign(p) is 1.
++
++X p := coercePreimagesImages([[1,2,3],[1,2,3]])
++X even? p
odd?      : %      -> Boolean
++ odd?(p) returns true if and only if p is an odd permutation
++ i.e. sign(p) is -1.
sort      : L %      -> L %
++ sort(lp) sorts a list of permutations lp according to
++ cycle structure first according to length of cycles,
++ second, if S has \spadtype{Finite} or S has
++ \spadtype{OrderedSet} according to lexicographical order of
++ entries in cycles of equal length.

```

```

if S has Finite then
  fixedPoints : % -> Set S
  ++ fixedPoints(p) returns the points fixed by the permutation p.
  ++X p := coercePreimagesImages([[0,1,2,3],[3,0,2,1]])$PERM ZMOD 4
  ++X fixedPoints p
if S has IntegerNumberSystem or S has Finite then
  coerceImages : L S -> %
  ++ coerceImages(ls) coerces the list ls to a permutation
  ++ whose image is given by ls and the preimage is fixed
  ++ to be [1,...,n].
  ++ Note: {coerceImages(ls)=coercePreimagesImages([1,...,n],ls)}.
  ++ We assume that both preimage and image do not contain repetitions.

private ==> add

-- representation of the object:

Rep := V L S

-- import of domains and packages

import OutputForm
import Vector List S

-- variables

p,q : %
exp : I

-- local functions first, signatures:

smaller? : (S,S) -> B
rotateCycle: L S -> L S
coerceCycle: L L S -> %
smallerCycle?: (L S, L S) -> B
shorterCycle?: (L S, L S) -> B
permord:(RECCYPE,RECCYPE) -> B
coerceToCycle:(%,B) -> L L S
duplicates?: L S -> B

smaller?(a:S, b:S): B ==
  S has OrderedSet => a <$S b
  S has Finite => lookup a < lookup b
  false

rotateCycle(cyc: L S): L S ==
  -- smallest element is put in first place
  -- doesn't change cycle if underlying set
  -- is not ordered or not finite.
  min:S := first cyc

```

```

minpos:I := 1          -- 1 = minIndex cyc
for i in 2..maxIndex cyc repeat
  if smaller?(cyc.i,min) then
    min := cyc.i
    minpos := i
--  one? minpos => cyc
(minpos = 1) => cyc
concat(last(cyc,((#cyc-minpos+1)::NNI)),first(cyc,(minpos-1)::NNI))

coerceCycle(lls : L L S): % ==
perm : % := 1
for lists in reverse lls repeat
  perm := cycle lists * perm
perm

smallerCycle?(cyca: L S, cycb: L S): B ==
#cyca ^= #cycb =>
#cyca < #cycb
for i in cyca for j in cycb repeat
  i ^= j => return smaller?(i, j)
false

shorterCycle?(cyca: L S, cycb: L S): B ==
#cyca < #cycb

permord(pa: RECCYPE, pb : RECCYPE): B ==
for i in pa.cycl for j in pb.cycl repeat
  i ^= j => return smallerCycle?(i, j)
#pa.cycl < #pb.cycl

coerceToCycle(p: %, doSorting?: B): L L S ==
preim := p.1
im := p.2
cycles := nil()$(L L S)
while not null preim repeat
  -- start next cycle
  firstEltInCycle: S := first preim
  nextCycle : L S := list firstEltInCycle
  preim := rest preim
  nextEltInCycle := first im
  im := rest im
  while nextEltInCycle ^= firstEltInCycle repeat
    nextCycle := cons(nextEltInCycle, nextCycle)
    i := position(nextEltInCycle, preim)
    preim := delete(preim,i)
    nextEltInCycle := im.i
    im := delete(im,i)
  nextCycle := reverse nextCycle
  -- check on 1-cycles, we don't list these
  if not null rest nextCycle then

```

```

    if doSorting? and (S has OrderedSet or S has Finite) then
      -- put smallest element in cycle first:
      nextCycle := rotateCycle nextCycle
      cycles := cons(nextCycle, cycles)
    not doSorting? => cycles
  -- sort cycles
  S has OrderedSet or S has Finite =>
    sort(smallerCycle?,cycles)$(L L S)
  sort(shorterCycle?,cycles)$(L L S)

duplicates? (ls : L S ): B ==
  x := copy ls
  while not null x repeat
    member? (first x ,rest x) => return true
    x := rest x
  false

-- now the exported functions

listRepresentation p ==
  s : RECPRI := [p.1,p.2]

coercePreimagesImages preImageAndImage ==
  preImage: List S := []
  image: List S := []
  for i in preImageAndImage.1
    for pi in preImageAndImage.2 repeat
      if i ~= pi then
        preImage := cons(i, preImage)
        image := cons(pi, image)

  [preImage, image]

movedPoints p == construct p.1

degree p == #movedPoints p

p = q ==
  #(preimp := p.1) ^= #(preimq := q.1) => false
  for i in 1..maxIndex preimp repeat
    pos := position(preimp.i, preimq)
    pos = 0 => return false
    (p.2).i ^= (q.2).pos => return false
  true

orbit(p ,el) ==
  -- start with a 1-element list:
  out : Set S := brace list el
  el2 := eval(p, el)
  while el2 ^= el repeat

```

```

-- be carefull: insert adds one element
-- as side effect to out
insert_!(el2, out)
el2 := eval(p, el2)
out

cyclePartition p ==
  partition([#c for c in coerceToCycle(p, false)])$Partition

order p ==
  ord: I := lcm removeDuplicates convert cyclePartition p
  ord::NNI

sign(p) ==
  even? p => 1
  - 1

even?(p) == even?((#(p.1) - numberOfCycles p)
  -- see the book of James and Kerber on symmetric groups
  -- for this formula.

odd?(p) == odd?((#(p.1) - numberOfCycles p)

pa < pb ==
  pacyc:= coerceToCycle(pa,true)
  pbcyc:= coerceToCycle(pb,true)
  for i in pacyc for j in pbcyc repeat
    i ^= j => return smallerCycle? ( i, j )
  maxIndex pacyc < maxIndex pbcyc

coerce(lls : L L S): % == coerceCycle lls

coerce(ls : L S): % == cycle ls

sort(inList : L %): L % ==
  not (S has OrderedSet or S has Finite) => inList
  ownList: L RECCYPE := nil()$(L RECCYPE)
  for sigma in inList repeat
    ownList :=
      cons([coerceToCycle(sigma,true),sigma]::RECCYPE, ownList)
  ownList := sort(permod, ownList)$(L RECCYPE)
  outList := nil()$(L %)
  for rec in ownList repeat
    outList := cons(rec.permut, outList)
  reverse outList

coerce (p: %): OUTFORM ==
  cycles: L L S := coerceToCycle(p,true)
  outfmL : L OUTFORM := nil()
  for cycle in cycles repeat

```

```

    outcycL: L OUTFORM := nil()
    for elt in cycle repeat
        outcycL := cons(elt :: OUTFORM, outcycL)
    outfmL := cons(paren blankSeparate reverse outcycL, outfmL)
-- The identity element will be output as 1:
null outfmL => outputForm(1@Integer)
-- represent a single cycle in the form (a b c d)
-- and not in the form ((a b c d)):
null rest outfmL => first outfmL
hconcat reverse outfmL

cycles(vs ) == coerceCycle vs

cycle(ls) ==
#ls < 2 => 1
duplicates? ls => error "cycle: the input contains duplicates"
[ls, append(rest ls, list first ls)]

coerceListOfPairs(loP) ==
preim := nil()$(L S)
im := nil()$(L S)
for pair in loP repeat
    if first pair ^= second pair then
        preim := cons(first pair, preim)
        im := cons(second pair, im)
duplicates?(preim) or duplicates?(im) or brace(preim)$(Set S) _
    ^= brace(im)$(Set S) =>
    error "coerceListOfPairs: the input cannot be interpreted as a permutation"
[preim, im]

q * p ==
-- use vectors for efficiency??
preimOfp : V S := construct p.1
imOfp : V S := construct p.2
preimOfq := q.1
imOfq := q.2
preimOfqp := nil()$(L S)
imOfqp := nil()$(L S)
-- 1 = minIndex preimOfp
for i in 1..(maxIndex preimOfp) repeat
    -- find index of image of p.i in q if it exists
    j := position(imOfp.i, preimOfq)
    if j = 0 then
        -- it does not exist
        preimOfqp := cons(preimOfp.i, preimOfqp)
        imOfqp := cons(imOfp.i, imOfqp)
    else
        -- it exists
        el := imOfq.j
        -- if the composition fixes the element, we don't

```

```

-- have to do anything
if el ~= preimOfp.i then
  preimOfqp := cons(preimOfp.i, preimOfqp)
  imOfqp := cons(el, imOfqp)
-- we drop the parts of q which have to do with p
preimOfq := delete(preimOfq, j)
imOfq := delete(imOfq, j)
[append(preimOfqp, preimOfq), append(imOfqp, imOfq)]

1 == new(2,empty())$Rep

inv p == [p.2, p.1]

eval(p, el) ==
  pos := position(el, p.1)
  pos = 0 => el
  (p.2).pos

elt(p, el) == eval(p, el)

numberOfCycles p == #coerceToCycle(p, false)

if S has IntegerNumberSystem then

  coerceImages (image) ==
    preImage : L S := [i::S for i in 1..maxIndex image]
    coercePreimagesImages [preImage,image]

if S has Finite then

  coerceImages (image) ==
    preImage : L S := [index(i::PI)::S for i in 1..maxIndex image]
    coercePreimagesImages [preImage,image]

fixedPoints ( p ) == complement movedPoints p

cyclePartition p ==
  pt := partition([#c for c in coerceToCycle(p, false)])$Partition
  pt +$PT conjugate(partition([#fixedPoints(p)])$PT)$PT

```

— PERM.dotabb —

```

"PERM" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PERM"]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"PERM" -> "FSAGG"

```


domain PERMGRP PermutationGroup

— PermutationGroup.input —

```

)set break resume
)sys rm -f PermutationGroup.output
)spool PermutationGroup.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PermutationGroup
--R
--R PermutationGroup(S: SetCategory) is a domain constructor
--R Abbreviation for PermutationGroup is PERMGRP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PERMGRP
--R
--R----- Operations -----
--R ?<? : (%,% ) -> Boolean          ?<=? : (%,% ) -> Boolean
--R ?=? : (%,% ) -> Boolean          base : % -> List(S)
--R coerce : % -> OutputForm         degree : % -> NonNegativeInteger
--R hash : % -> SingleInteger        latex : % -> String
--R movedPoints : % -> Set(S)        orbit : (% ,Set(S)) -> Set(Set(S))
--R orbit : (% ,S) -> Set(S)         orbits : % -> Set(Set(S))
--R order : % -> NonNegativeInteger  random : % -> Permutation(S)
--R ?~=? : (%,% ) -> Boolean
--R coerce : List(Permutation(S)) -> %
--R coerce : % -> List(Permutation(S))
--R ?.? : (% ,NonNegativeInteger) -> Permutation(S)
--R generators : % -> List(Permutation(S))
--R initializeGroupForWordProblem : (% ,Integer,Integer) -> Void
--R initializeGroupForWordProblem : % -> Void
--R member? : (Permutation(S),%) -> Boolean
--R orbit : (% ,List(S)) -> Set(List(S))
--R permutationGroup : List(Permutation(S)) -> %
--R random : (% ,Integer) -> Permutation(S)
--R strongGenerators : % -> List(Permutation(S))
--R wordInGenerators : (Permutation(S),%) -> List(NonNegativeInteger)
--R wordInStrongGenerators : (Permutation(S),%) -> List(NonNegativeInteger)
--R wordsForStrongGenerators : % -> List(List(NonNegativeInteger))
--R

```

```
--E 1
```

```
)spool
)lisp (bye)
```

— **PermutationGroup.help** —

```
=====
PermutationGroup examples
=====
```

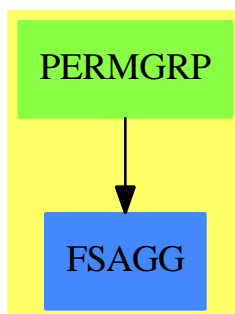
PermutationGroup implements permutation groups acting on a set S , i.e. all subgroups of the symmetric group of S , represented as a list of permutations (generators). Note that therefore the objects are not members of the Axiom category Group.

Using the idea of base and strong generators by Sims, basic routines and algorithms are implemented so that the word problem for permutation groups can be solved.

See Also:

```
o )show PermutationGroup
```

PermutationGroup (PERMGRP)



Exports:

base	coerce	degree
hash	latex	movedPoints
orbit	orbits	order
random	generators	initializeGroupForWordProblem
member?	permutationGroup	random
strongGenerators	wordInGenerators	wordInStrongGenerators
wordsForStrongGenerators	?~=?	?..?
?<?	?<=?	?=?

— domain PERMGRP PermutationGroup —

```

)abbrev domain PERMGRP PermutationGroup
++ Authors: G. Schneider, H. Gollan, J. Grabmeier
++ Date Created: 13 February 1987
++ Date Last Updated: 24 May 1991
++ References:
++   C. Sims: Determining the conjugacy classes of a permutation group,
++   in Computers in Algebra and Number Theory, SIAM-AMS Proc., Vol. 4,
++   Amer. Math. Soc., Providence, R. I., 1971, pp. 191-195
++ Description:
++ PermutationGroup implements permutation groups acting
++ on a set S, i.e. all subgroups of the symmetric group of S,
++ represented as a list of permutations (generators). Note that
++ therefore the objects are not members of the Axiom category
++ \spadtype{Group}.
++ Using the idea of base and strong generators by Sims,
++ basic routines and algorithms
++ are implemented so that the word problem for
++ permutation groups can be solved.
---+ Note: we plan to implement lattice operations on the subgroup
---+ lattice in a later release

PermutationGroup(S:SetCategory): public == private where

L    ==> List
PERM ==> Permutation
FSET ==> Set
I    ==> Integer
NNI  ==> NonNegativeInteger
V    ==> Vector
B    ==> Boolean
OUT  ==> OutputForm
SYM  ==> Symbol
REC  ==> Record ( orb : L NNI , svc : V I )
REC2 ==> Record(order:NNI,sgset:L V NNI,_,
                gpbase:L NNI,orbs:L REC,mp:L S,wd:L L NNI)
REC3 ==> Record(elt:V NNI,lst:L NNI)
REC4 ==> Record(bool:B,lst:L NNI)

```

```

public ==> SetCategory with

coerce          : %          -> L PERM S
  ++ coerce(gp) returns the generators of the group gp.
generators      : %          -> L PERM S
  ++ generators(gp) returns the generators of the group gp.
elt             : (% ,NNI)    -> PERM S
  ++ elt(gp,i) returns the i-th generator of the group gp.
random          : (% ,I)      -> PERM S
  ++ random(gp,i) returns a random product of maximal i generators
  ++ of the group gp.
random          : %          -> PERM S
  ++ random(gp) returns a random product of maximal 20 generators
  ++ of the group gp.
  ++ Note: random(gp)=random(gp,20).
order           : %          -> NNI
  ++ order(gp) returns the order of the group gp.
degree          : %          -> NNI
  ++ degree(gp) returns the number of points moved by all permutations
  ++ of the group gp.
base            : %          -> L S
  ++ base(gp) returns a base for the group gp.
strongGenerators : %          -> L PERM S
  ++ strongGenerators(gp) returns strong generators for
  ++ the group gp.
wordsForStrongGenerators : %          -> L L NNI
  ++ wordsForStrongGenerators(gp) returns the words for the strong
  ++ generators of the group gp in the original generators of
  ++ gp, represented by their indices in the list, given by
  ++ generators.
coerce          : L PERM S -> %
  ++ coerce(ls) coerces a list of permutations ls to the group
  ++ generated by this list.
permutationGroup : L PERM S -> %
  ++ permutationGroup(ls) coerces a list of permutations ls to
  ++ the group generated by this list.
orbit           : (% ,S)      -> FSET S
  ++ orbit(gp,el) returns the orbit of the element el under the
  ++ group gp, i.e. the set of all points gained by applying
  ++ each group element to el.
orbits          : %          -> FSET FSET S
  ++ orbits(gp) returns the orbits of the group gp, i.e.
  ++ it partitions the (finite) of all moved points.
orbit           : (% ,FSET S)-> FSET FSET S
  ++ orbit(gp,els) returns the orbit of the unordered
  ++ set els under the group gp.
orbit           : (% ,L S)    -> FSET L S
  ++ orbit(gp,ls) returns the orbit of the ordered
  ++ list ls under the group gp.
  ++ Note: return type is L L S temporarily because FSET L S has an error.

```

```

-- (GILT DAS NOCH?)
member?      : (PERM S, %)-> B
  ++ member?(pp, gp) answers the question, whether the
  ++ permutation pp is in the group gp or not.
wordInStrongGenerators : (PERM S, %)-> L NNI
  ++ wordInStrongGenerators(p, gp) returns the word for the
  ++ permutation p in the strong generators of the group gp,
  ++ represented by the indices of the list, given by strongGenerators.
wordInGenerators : (PERM S, %)-> L NNI
  ++ wordInGenerators(p, gp) returns the word for the permutation p
  ++ in the original generators of the group gp,
  ++ represented by the indices of the list, given by generators.
movedPoints  : %      -> FSET S
  ++ movedPoints(gp) returns the points moved by the group gp.
"<"          : (% , %)  -> B
  ++ gp1 < gp2 returns true if and only if gp1
  ++ is a proper subgroup of gp2.
"<="         : (% , %)  -> B
  ++ gp1 <= gp2 returns true if and only if gp1
  ++ is a subgroup of gp2.
  ++ Note: because of a bug in the parser you have to call this
  ++ function explicitly by gp1 <= $(PERMGRP S) gp2.
-- (GILT DAS NOCH?)
initializeGroupForWordProblem : %      -> Void
  ++ initializeGroupForWordProblem(gp) initializes the group gp
  ++ for the word problem.
  ++ Notes: it calls the other function of this name with parameters
  ++ 0 and 1: initializeGroupForWordProblem(gp, 0, 1).
  ++ Notes: (1) be careful: invoking this routine will destroy the
  ++ possibly information about your group (but will recompute it again)
  ++ (2) users need not call this function normally for the solution of
  ++ the word problem.
initializeGroupForWordProblem : (% , I, I) -> Void
  ++ initializeGroupForWordProblem(gp, m, n) initializes the group
  ++ gp for the word problem.
  ++ Notes: (1) with a small integer you get shorter words, but the
  ++ routine takes longer than the standard routine for longer words.
  ++ (2) be careful: invoking this routine will destroy the possibly stored
  ++ information about your group (but will recompute it again).
  ++ (3) users need not call this function normally for the solution of
  ++ the word problem.

private ==> add

-- representation of the object:

Rep := Record ( gens : L PERM S , information : REC2 )

-- import of domains and packages

```

```

import Permutation S
import OutputForm
import Symbol
import Void

--first the local variables

sgs          : L V NNI      := []
baseOfGroup  : L NNI        := []
sizeOfGroup  : NNI          := 1
degree       : NNI          := 0
gporb        : L REC        := []
out          : L L V NNI    := []
outword      : L L L NNI    := []
wordlist     : L L NNI      := []
basePoint    : NNI          := 0
newBasePoint : B            := true
supp         : L S          := []
ord          : NNI          := 1
wordProblem  : B            := true

--local functions first, signatures:

shortenWord:(L NNI, %)->L NNI
times:(V NNI, V NNI)->V NNI
strip:(V NNI,REC,L V NNI,L L NNI)->REC3
orbitInternal:(%,L S )->L L S
inv: V NNI->V NNI
ranelt:(L V NNI,L L NNI, I)->REC3
testIdentity:V NNI->B
pointList: %->L S
orbitWithSvc:(L V NNI ,NNI )->REC
cosetRep:(NNI ,REC ,L V NNI )->REC3
bsgs1:(L V NNI,NNI,L L NNI,I,%,I)->NNI
computeOrbits: I->L NNI
reduceGenerators: I->Void
bsgs:(%, I, I)->NNI
initialize: %->FSET PERM S
knownGroup?: %->Void
subgroup:(%, %)->B
memberInternal:(PERM S, %, B)->REC4

--local functions first, implementations:

shortenWord ( lw : L NNI , gp : % ) : L NNI ==
  -- tries to shorten a word in the generators by removing identities
  gpgens : L PERM S := coerce gp
  orderList : L NNI := [ order gen for gen in gpgens ]
  newlw : L NNI := copy lw
  for i in 1.. maxIndex orderList repeat

```

```

    if orderList.i = 1 then
      while member?(i,newlw) repeat
        -- removing the trivial element
        pos := position(i,newlw)
        newlw := delete(newlw,pos)
    flag : B := true
    while flag repeat
      actualLength : NNI := (maxIndex newlw) pretend NNI
      pointer := actualLength
      test := newlw.pointer
      anzahl : NNI := 1
      flag := false
      while pointer > 1 repeat
        pointer := ( pointer - 1 )::NNI
        if newlw.pointer ^= test then
          -- don't get a trivial element, try next
          test := newlw.pointer
          anzahl := 1
        else
          anzahl := anzahl + 1
          if anzahl = orderList.test then
            -- we have an identity, so remove it
            for i in (pointer+anzahl)..actualLength repeat
              newlw.(i-anzahl) := newlw.i
            newlw := first(newlw, (actualLength - anzahl) :: NNI)
            flag := true
            pointer := 1
      newlw

times ( p : V NNI , q : V NNI ) : V NNI ==
  -- internal multiplication of permutations
  [ qelt(p,qelt(q,i)) for i in 1..degree ]

strip(element:V NNI,orbit:REC,group:L V NNI,words:L L NNI) : REC3 ==
  -- strip an element into the stabilizer
  actelt      := element
  schreierVector := orbit.svc
  point       := orbit.orb.1
  outlist     := nil()$(L NNI)
  entryLessZero : B := false
  while ^entryLessZero repeat
    entry := schreierVector.(actelt.point)
    entryLessZero := (entry < 0)
    if ^entryLessZero then
      actelt := times(group.entry, actelt)
      if wordProblem then outlist := append ( words.(entry::NNI) , outlist )
  [ actelt , reverse outlist ]

orbitInternal ( gp : % , startList : L S ) : L L S ==
  orbitList : L L S := [ startList ]

```

```

pos : I := 1
while not zero? pos repeat
  gpset : L PERM S := gp.gens
  for gen in gpset repeat
    newList := nil()$(L S)
    workList := orbitList.pos
    for j in #workList..1 by -1 repeat
      newList := cons ( eval ( gen , workList.j ) , newList )
    if ~member?( newList , orbitList ) then
      orbitList := cons ( newList , orbitList )
      pos := pos + 1
  pos := pos - 1
reverse orbitList

inv ( p : V NNI ) : V NNI ==
  -- internal inverse of a permutation
  q : V NNI := new(degree,0)$(V NNI)
  for i in 1..degree repeat q.(qelt(p,i)) := i
  q

ranelt ( group : L V NNI , word : L L NNI , maxLoops : I ) : REC3 ==
  -- generate a "random" element
  numberOfGenerators := # group
  randomInteger : I := 1 + (random())$Integer rem numberOfGenerators
  randomElement : V NNI := group.randomInteger
  words := nil()$(L NNI)
  if wordProblem then words := word.(randomInteger::NNI)
  if maxLoops > 0 then
    numberOfLoops : I := 1 + (random())$Integer rem maxLoops
  else
    numberOfLoops : I := maxLoops
  while numberOfLoops > 0 repeat
    randomInteger : I := 1 + (random())$Integer rem numberOfGenerators
    randomElement := times ( group.randomInteger , randomElement )
    if wordProblem then words := append ( word.(randomInteger::NNI) , words )
    numberOfLoops := numberOfLoops - 1
  [ randomElement , words ]

testIdentity ( p : V NNI ) : B ==
  -- internal test for identity
  for i in 1..degree repeat qelt(p,i) ^= i => return false
  true

pointList(group : %) : L S ==
  support : FSET S := brace() -- empty set !!
  for perm in group.gens repeat
    support := union(support, movedPoints perm)
  parts support

orbitWithSvc ( group : L V NNI , point : NNI ) : REC ==

```



```

-- compute orbit with Schreier vector, "-2" means not in the orbit,
-- "-1" means starting point, the PI correspond to generators
newGroup := nil()$(L V NNI)
for el in group repeat
  newGroup := cons ( inv el , newGroup )
newGroup      := reverse newGroup
orbit         : L NNI := [ point ]
schreierVector : V I   := new ( degree , -2 )
schreierVector.point := -1
position : I := 1
while not zero? position repeat
  for i in 1..#newGroup repeat
    newPoint := orbit.position
    newPoint := newGroup.i.newPoint
    if ^ member? ( newPoint , orbit ) then
      orbit      := cons ( newPoint , orbit )
      position   := position + 1
      schreierVector.newPoint := i
    position := position - 1
  [ reverse orbit , schreierVector ]

cosetRep ( point : NNI , o : REC , group : L V NNI ) : REC3 ==
  ppt      := point
  xelt : V NNI := [ n for n in 1..degree ]
  word    := nil()$(L NNI)
  orb     := o.orb
  osvc    := o.svc
  while degree > 0 repeat
    p := osvc.ppt
    p < 0 => return [ xelt , word ]
    x := group.p
    xelt := times ( x , xelt )
    if wordProblem then word := append ( wordlist.p , word )
    ppt := x.ppt

bsgs1 (group:L V NNI,number1:NNI,words:L L NNI,maxLoops:I,gp:%,diff:I)_
: NNI ==
-- try to get a good approximation for the strong generators and base
for i in number1..degree repeat
  ort := orbitWithSvc ( group , i )
  k   := ort.orb
  k1  := # k
  if k1 ^= 1 then leave
  gpsgs := nil()$(L V NNI)
  words2 := nil()$(L L NNI)
  gplength : NNI := #group
  for jj in 1..gplength repeat if (group.jj).i ^= i then leave
  for k in 1..gplength repeat
    el2 := group.k
    if el2.i ^= i then

```

```

    gpsgs := cons ( el2 , gpsgs )
    if wordProblem then words2 := cons ( words.k , words2 )
else
    gpsgs := cons ( times ( group.jj , el2 ) , gpsgs )
    if wordProblem _
        then words2 := cons ( append ( words.jj , words.k ) , words2 )
group2 := nil()$(L V NNI)
words3 := nil()$(L L NNI)
j : I := 15
while j > 0 repeat
    -- find generators for the stabilizer
    ran := ranelt ( group , words , maxLoops )
    str := strip ( ran.el , ort , group , words )
    el2 := str.el
    if ^ testIdentity el2 then
        if ^ member?(el2,group2) then
            group2 := cons ( el2 , group2 )
            if wordProblem then
                help : L NNI := append ( reverse str.lst , ran.lst )
                help      := shortenWord ( help , gp )
                words3     := cons ( help , words3 )
            j := j - 2
        j := j - 1
    -- this is for word length control
    if wordProblem then maxLoops := maxLoops - diff
    if ( null group2 ) or ( maxLoops < 0 ) then
        sizeOfGroup := k1
        baseOfGroup := [ i ]
        out          := [ gpsgs ]
        outword      := [ words2 ]
        return sizeOfGroup
    k2 := bsgs1 ( group2 , i + 1 , words3 , maxLoops , gp , diff )
    sizeOfGroup := k1 * k2
    out          := append ( out , [ gpsgs ] )
    outword      := append ( outword , [ words2 ] )
    baseOfGroup := cons ( i , baseOfGroup )
    sizeOfGroup

computeOrbits ( kkk : I ) : L NNI ==
    -- compute the orbits for the stabilizers
    sgs := nil()
    orbitLength := nil()$(L NNI)
    gporb := nil()
    for i in 1..#baseOfGroup repeat
        sgs := append ( sgs , out.i )
        pt  := #baseOfGroup - i + 1
        obs := orbitWithSvc ( sgs , baseOfGroup.pt )
        orbitLength := cons ( #obs.orb , orbitLength )
        gporb := cons ( obs , gporb )
    gporb := reverse gporb

```

```

reverse orbitLength

reduceGenerators ( kkk : I ) : Void ==
-- try to reduce number of strong generators
orbitLength := computeOrbits ( kkk )
sgs         := nil()
wordlist    := nil()
for i in 1..(kkk-1) repeat
  sgs := append ( sgs , out.i )
  if wordProblem then wordlist := append ( wordlist , outword.i )
removedGenerator := false
baseLength : NNI := #baseOfGroup
for nnn in kkk..(baseLength-1) repeat
  sgs := append ( sgs , out.nnn )
  if wordProblem then wordlist := append ( wordlist , outword.nnn )
  pt := baseLength - nnn + 1
  obs := orbitWithSvc ( sgs , baseOfGroup.pt )
  i := 1
  while not ( i > # out.nnn ) repeat
    pos := position ( out.nnn.i , sgs )
    sgs2 := delete(sgs, pos)
    obs2 := orbitWithSvc ( sgs2 , baseOfGroup.pt )
    if # obs2.orb = orbitLength.nnn then
      test := true
      for j in (nnn+1)..(baseLength-1) repeat
        pt2 := baseLength - j + 1
        sgs2 := append ( sgs2 , out.j )
        obs2 := orbitWithSvc ( sgs2 , baseOfGroup.pt2 )
        if # obs2.orb ^= orbitLength.j then
          test := false
          leave
      if test then
        removedGenerator := true
        sgs := delete (sgs, pos)
        if wordProblem then wordlist := delete(wordlist, pos)
        out.nnn := delete (out.nnn, i)
        if wordProblem then _
          outword.nnn := delete(outword.nnn, i )
      else
        i := i + 1
    else
      i := i + 1
  if removedGenerator then orbitLength := computeOrbits ( kkk )
void()

bsgs ( group : % ,maxLoops : I , diff : I ) : NNI ==
-- the MOST IMPORTANT part of the package
supp := pointList group
degree := # supp

```

```

if degree = 0 then
  sizeOfGroup := 1
  sgs         := [ [ 0 ] ]
  baseOfGroup := nil()
  gporb       := nil()
  return sizeOfGroup
newGroup := nil()$(L V NNI)
gp       : L PERM S := group.gens
words := nil()$(L L NNI)
for ggg in 1..#gp repeat
  q := new(degree,0)$(V NNI)
  for i in 1..degree repeat
    newEl := eval ( gp.ggg , supp.i )
    pos2  := position ( newEl , supp )
    q.i   := pos2 pretend NNI
  newGroup := cons ( q , newGroup )
  if wordProblem then words := cons(list ggg, words)
if maxLoops < 1 then
  -- try to get the (approximate) base length
  if zero? (# ((group.information).gpbase)) then
    wordProblem := false
    k           := bsgs1 ( newGroup , 1 , words , 20 , group , 0 )
    wordProblem := true
    maxLoops    := (# baseOfGroup) - 1
  else
    maxLoops    := (# ((group.information).gpbase)) - 1
k           := bsgs1 ( newGroup , 1 , words , maxLoops , group , diff )
kkk : I := 1
newGroup := reverse newGroup
noAnswer : B := true
while noAnswer repeat
  reduceGenerators kkk
-- *** Here is former "bsgs2" *** --
-- test whether we have a base and a strong generating set
sgs := nil()
wordlist := nil()
for i in 1..(kkk-1) repeat
  sgs := append ( sgs , out.i )
  if wordProblem then wordlist := append ( wordlist , outword.i )
noresult : B := true
for i in kkk..#baseOfGroup while noresult repeat
  sgs := append ( sgs , out.i )
  if wordProblem then wordlist := append ( wordlist , outword.i )
gporbi := gporb.i
for pt in gporbi.orb while noresult repeat
  ppp := cosetRep ( pt , gporbi , sgs )
  y1  := inv ppp.elc
  word3 := ppp.lst
  for jjj in 1..#sgs while noresult repeat
    word := nil()$(L NNI)

```

```

z          := times ( sgs.jjj , y1 )
if wordProblem then word := append ( wordlist.jjj , word )
ppp        := cosetRep ( (sgs.jjj).pt , gporbi , sgs )
z          := times ( ppp.elc , z )
if wordProblem then word := append ( ppp.lst , word )
newBasePoint := false
for j in (i-1)..1 by -1 while noresult repeat
  s := gporb.j.svc
  p := gporb.j.orb.1
  while ( degree > 0 ) and noresult repeat
    entry := s.(z.p)
    if entry < 0 then
      if entry = -1 then leave
      basePoint := j::NNI
      noresult := false
    else
      ee := sgs.entry
      z := times ( ee , z )
      if wordProblem then word := append ( wordlist.entry , word )
  if noresult then
    basePoint := 1
    newBasePoint := true
    noresult := testIdentity z
noAnswer := not (testIdentity z)
if noAnswer then
  -- we have missed something
  word2 := nil()$(L NNI)
  if wordProblem then
    for wd in word3 repeat
      ttt := newGroup.wd
      while not (testIdentity ttt) repeat
        word2 := cons ( wd , word2 )
        ttt := times ( ttt , newGroup.wd )
      word := append ( word , word2 )
      word := shortenWord ( word , group )
  if newBasePoint then
    for i in 1..degree repeat
      if z.i ^= i then
        baseOfGroup := append ( baseOfGroup , [ i ] )
        leave
    out := cons (list z , out )
    if wordProblem then outword := cons (list word , outword )
  else
    out.basePoint := cons ( z , out.basePoint )
    if wordProblem then outword.basePoint := cons(word ,outword.basePoint )
  kkk := basePoint
sizeOfGroup := 1
for j in 1..#baseOfGroup repeat
  sizeOfGroup := sizeOfGroup * # gporb.j.orb
sizeOfGroup

```

```

initialize ( group : % ) : FSET PERM S ==
  group2 := brace()$(FSET PERM S)
  gp : L PERM S := group.gens
  for gen in gp repeat
    if degree gen > 0 then insert_!(gen, group2)
  group2

knownGroup? (gp : %) : Void ==
  -- do we know the group already?
  result := gp.information
  if result.order = 0 then
    wordProblem      := false
    ord              := bsgs ( gp , 20 , 0 )
    result            := [ ord , sgs , baseOfGroup , gporb , supp , [] ]
    gp.information    := result
  else
    ord              := result.order
    sgs              := result.sgset
    baseOfGroup      := result.gpbasis
    gporb            := result.orbs
    supp             := result.mp
    wordlist         := result.wd
  void

subgroup ( gp1 : % , gp2 : % ) : B ==
  gpset1 := initialize gp1
  gpset2 := initialize gp2
  empty? difference (gpset1, gpset2) => true
  for el in parts gpset1 repeat
    not member? (el, gp2) => return false
  true

memberInternal ( p : PERM S , gp : % , flag : B ) : REC4 ==
  -- internal membership testing
  supp      := pointList gp
  outlist   := nil()$(L NNI)
  mP : L S := parts movedPoints p
  for x in mP repeat
    not member? (x, supp) => return [ false , nil()$(L NNI) ]
  if flag then
    member? ( p , gp.gens ) => return [ true , nil()$(L NNI) ]
    knownGroup? gp
  else
    result := gp.information
    if #(result.wd) = 0 then
      initializeGroupForWordProblem gp
    else
      ord := result.order

```

```

sgs          := result.sgset
baseOfGroup  := result.gpbases
gporb        := result.orbs
supp         := result.mp
wordlist     := result.wd
degree := # supp
pp := new(degree,0)$(V NNI)
for i in 1..degree repeat
  el := eval ( p , supp.i )
  pos := position ( el , supp )
  pp.i := pos::NNI
words := nil()$(L L NNI)
if wordProblem then
  for i in 1..#sgs repeat
    lw : L NNI := [ (#sgs - i + 1)::NNI ]
    words := cons ( lw , words )
  for i in #baseOfGroup..1 by -1 repeat
    str := strip ( pp , gporb.i , sgs , words )
    pp := str.elc
    if wordProblem then outlist := append ( outlist , str.lst )
  [ testIdentity pp , reverse outlist ]

--now the exported functions

coerce ( gp : % ) : L PERM S == gp.gens
generators ( gp : % ) : L PERM S == gp.gens

strongGenerators ( group ) ==
  knownGroup? group
  degree := # supp
  strongGens := nil()$(L PERM S)
  for i in sgs repeat
    pairs := nil()$(L L S)
    for j in 1..degree repeat
      pairs := cons ( [ supp.j , supp.(i.j) ] , pairs )
    strongGens := cons ( coerceListOfPairs pairs , strongGens )
  reverse strongGens

elt ( gp , i ) == (gp.gens).i

movedPoints ( gp ) == brace pointList gp

random ( group , maximalNumberOfFactors ) ==
  maximalNumberOfFactors < 1 => 1$(PERM S)
  gp : L PERM S := group.gens
  numberOfGenerators := # gp
  randomInteger : I := 1 + (random()$Integer rem numberOfGenerators)
  randomElement := gp.randomInteger
  numberOfLoops : I := 1 + (random()$Integer rem maximalNumberOfFactors)
  while numberOfLoops > 0 repeat

```

```

    randomInteger : I := 1 + (random()$Integer rem numberOfGenerators)
    randomElement := gp.randomInteger * randomElement
    numberOfLoops := numberOfLoops - 1
    randomElement

random ( group ) == random ( group , 20 )

order ( group ) ==
    knownGroup? group
    ord

degree ( group ) == # pointList group

base ( group ) ==
    knownGroup? group
    groupBase := nil()$(L S)
    for i in baseOfGroup repeat
        groupBase := cons ( supp.i , groupBase )
    reverse groupBase

wordsForStrongGenerators ( group ) ==
    knownGroup? group
    wordlist

coerce ( gp : L PERM S ) : % ==
    result : REC2 := [ 0 , [] , [] , [] , [] , [] ]
    group      := [ gp , result ]

permutationGroup ( gp : L PERM S ) : % ==
    result : REC2 := [ 0 , [] , [] , [] , [] , [] ]
    group      := [ gp , result ]

coerce(group: %) : OUT ==
    outList := nil()$(L OUT)
    gp : L PERM S := group.gens
    for i in (maxIndex gp)..1 by -1 repeat
        outList := cons(coerce gp.i, outList)
    postfix(outputForm(">":SYM),postfix(commaSeparate outList,outputForm("<":SYM)))

orbit ( gp : % , el : S ) : FSET S ==
    elList : L S := [ el ]
    outList      := orbitInternal ( gp , elList )
    outSet       := brace()$(FSET S)
    for i in 1..#outList repeat
        insert_! ( outList.i.1 , outSet )
    outSet

orbits ( gp ) ==
    spp      := movedPoints gp
    orbits := nil()$(L FSET S)

```



```

while cardinality spp > 0 repeat
  el      := extract_! spp
  orbitSet := orbit ( gp , el )
  orbits   := cons ( orbitSet , orbits )
  spp      := difference ( spp , orbitSet )
brace orbits

member? (p, gp) ==
  wordProblem := false
  mi := memberInternal ( p , gp , true )
  mi.bool

wordInStrongGenerators (p, gp) ==
  mi := memberInternal ( inv p , gp , false )
  not mi.bool => error "p is not an element of gp"
  mi.lst

wordInGenerators (p, gp) ==
  lll : L NNI := wordInStrongGenerators (p, gp)
  outlist := nil()$(L NNI)
  for wd in lll repeat
    outlist := append ( outlist , wordlist.wd )
  shortenWord ( outlist , gp )

gp1 < gp2 ==
  not empty? difference ( movedPoints gp1 , movedPoints gp2 ) => false
  not subgroup ( gp1 , gp2 ) => false
  order gp1 = order gp2 => false
  true

gp1 <= gp2 ==
  not empty? difference ( movedPoints gp1 , movedPoints gp2 ) => false
  subgroup ( gp1 , gp2 )

gp1 = gp2 ==
  movedPoints gp1 ^= movedPoints gp2 => false
  if #(gp1.gens) <= #(gp2.gens) then
    not subgroup ( gp1 , gp2 ) => return false
  else
    not subgroup ( gp2 , gp1 ) => return false
  order gp1 = order gp2 => true
  false

orbit ( gp : % , startSet : FSET S ) : FSET FSET S ==
  startList : L S := parts startSet
  outList      := orbitInternal ( gp , startList )
  outSet       := brace()$(FSET FSET S)
  for i in 1..#outList repeat
    newSet : FSET S := brace outList.i
    insert_! ( newSet , outSet )

```

```

outSet

orbit ( gp : % , startList : L S ) : FSET L S ==
  brace orbitInternal(gp, startList)

initializeGroupForWordProblem ( gp , maxLoops , diff ) ==
  wordProblem      := true
  ord              := bsgs ( gp , maxLoops , diff )
  gp.information := [ ord , sgs , baseOfGroup , gporb , supp , wordlist ]
  void

initializeGroupForWordProblem ( gp ) == initializeGroupForWordProblem ( gp , 0 , 1 )

```

— PERMGRP.dotabb —

```

"PERMGRP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PERMGRP"]
"FSAGG"   [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"PERMGRP" -> "FSAGG"

```

domain HACKPI Pi

What is the purpose of the HACKPI domain?

HACKPI is a hack provided for the benefit of the axiom interpreter. As a mathematical type, it is the simple transcendental extension $\mathbb{Q}(\pi)$ of the rational numbers. This type allows interactive users to use the name '%pi' without a type both where a numerical value is expected [as in `draw(sin x,x=-%pi..%pi)`] or when the exact symbolic value is meant. The interpreter defaults a typeless %pi to HACKPI and then uses the various conversions to cast it further as required by the context.

One could argue that it is unfair to single %pi out from other constants, but it occurs frequently enough in school examples (specially for graphs) so it was worth a special hack. In a non-interactive environment (library), HACKPI would not exist.

(Manuel Bronstein)

— Pi.input —

```

)set break resume
)sys rm -f Pi.output
)spool Pi.output

```

```

)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Pi
--R
--R Pi is a domain constructor
--R Abbreviation for Pi is HACKPI
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for HACKPI
--R
--R----- Operations -----
--R ?? : (Fraction(Integer),%) -> %      ?? : (%,Fraction(Integer)) -> %
--R ?? : (%,%) -> %                      ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %        ??? : (%,Integer) -> %
--R ??? : (%,PositiveInteger) -> %      ?+? : (%,%) -> %
--R ?-? : (%,%) -> %                    -? : % -> %
--R ?/? : (%,%) -> %                    ?=? : (%,%) -> Boolean
--R 1 : () -> %                         0 : () -> %
--R ?? : (%,Integer) -> %               ?? : (%,PositiveInteger) -> %
--R associates? : (%,%) -> Boolean      coerce : % -> Float
--R coerce : % -> DoubleFloat           coerce : Fraction(Integer) -> %
--R coerce : % -> %                     coerce : Integer -> %
--R coerce : % -> OutputForm            convert : % -> InputForm
--R convert : % -> DoubleFloat          convert : % -> Float
--R factor : % -> Factored(%)           gcd : List(%) -> %
--R gcd : (%,%) -> %                   hash : % -> SingleInteger
--R inv : % -> %                        latex : % -> String
--R lcm : List(%) -> %                  lcm : (%,%) -> %
--R one? : % -> Boolean                 pi : () -> %
--R prime? : % -> Boolean               ?quo? : (%,%) -> %
--R recip : % -> Union(%, "failed")     ?rem? : (%,%) -> %
--R retract : % -> Fraction(Integer)    retract : % -> Integer
--R sample : () -> %                   sizeLess? : (%,%) -> Boolean
--R squareFree : % -> Factored(%)       squareFreePart : % -> %
--R unit? : % -> Boolean                 unitCanonical : % -> %
--R zero? : % -> Boolean                 ?^=? : (%,%) -> Boolean
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R ?? : (%,NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R convert : % -> Fraction(SparseUnivariatePolynomial(Integer))
--R divide : (%,%) -> Record(quotient: %, remainder: %)
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(%),%) -> Union(List(%), "failed")
--R exquo : (%,%) -> Union(%, "failed")
--R extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %), "failed")
--R extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %)
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUniva

```

```

--R multiEuclidean : (List(%),%) -> Union(List(%),"failed")
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %)
--R retractIfCan : % -> Union(Fraction(Integer),"failed")
--R retractIfCan : % -> Union(Integer,"failed")
--R subtractIfCan : (%,% ) -> Union(%,"failed")
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R
--E 1

```

```

)spool
)lisp (bye)

```

— Pi.help —

=====

Pi examples

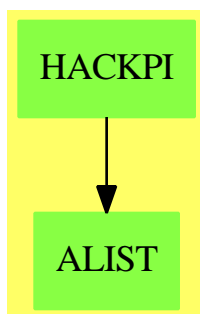
=====

Symbolic fractions in %pi with integer coefficients.

The point for using Pi as the default domain for those fractions is that Pi is coercible to the float types, and not Expression.

See Also:
o)show Pi

Pi (HACKPI)



See

⇒ “Expression” (EXPR) [6](#) on page [723](#)

Exports:

0	1	associates?	characteristic	coerce
convert	divide	euclideanSize	expressIdealMember	exquo
extendedEuclidean	extendedEuclidean	factor	gcd	gcdPolynomial
hash	inv	latex	lcm	multiEuclidean
one?	pi	prime?	principalIdeal	recip
retract	retractIfCan	retractIfCan	sample	sizeLess?
squareFree	squareFreePart	subtractIfCan	unit?	unitCanonical
unitNormal	zero?	?*?	?**?	?+?
?-?	-?	?/?	?=?	?^?
?^=?	?quo?	?rem?		

— domain HACKPI Pi —

```

)abbrev domain HACKPI Pi
++ Author: Manuel Bronstein
++ Date Created: 21 Feb 1990
++ Date Last Updated: 12 Mai 1992
++ Description:
++ Symbolic fractions in %pi with integer coefficients;
++ The point for using Pi as the default domain for those fractions
++ is that Pi is coercible to the float types, and not Expression.

Pi(): Exports == Implementation where
  PZ ==> Polynomial Integer
  UP ==> SparseUnivariatePolynomial Integer
  RF ==> Fraction UP

Exports ==> Join(Field, CharacteristicZero, RetractableTo Integer,
  RetractableTo Fraction Integer, RealConstant,
  CoercibleTo DoubleFloat, CoercibleTo Float,
  ConvertibleTo RF, ConvertibleTo InputForm) with
  pi: () -> % ++ pi() returns the symbolic %pi.
Implementation ==> RF add
  Rep := RF

  sympi := "%pi"::Symbol

  p2sf: UP -> DoubleFloat
  p2f : UP -> Float
  p2o : UP -> OutputForm
  p2i : UP -> InputForm
  p2p: UP -> PZ

  pi() == (monomial(1, 1)$UP :: RF) pretend %
  convert(x:%):RF == x pretend RF
  convert(x:%):Float == x::Float
  convert(x:%):DoubleFloat == x::DoubleFloat
  coerce(x:%):DoubleFloat == p2sf( numer x) / p2sf(denom x)

```

```

coerce(x:):Float      == p2f( numer x) / p2f(denom x)
p2o p                 == outputForm(p, sympi::OutputForm)
p2i p                 == convert p2p p

p2p p ==
  ans:PZ := 0
  while p ^= 0 repeat
    ans := ans + monomial(leadingCoefficient(p)::PZ, sympi, degree p)
    p := reductum p
  ans

coerce(x:):OutputForm ==
  (r := retractIfCan(x)@Union(UP, "failed")) case UP => p2o(r::UP)
  p2o( numer x) / p2o(denom x)

convert(x:):InputForm ==
  (r := retractIfCan(x)@Union(UP, "failed")) case UP => p2i(r::UP)
  p2i( numer x) / p2i(denom x)

p2sf p ==
  map((x:Integer):DoubleFloat+>x::DoubleFloat, p)_
  $SparseUnivariatePolynomialFunctions2(Integer, DoubleFloat)
  (pi())$DoubleFloat

p2f p ==
  map((x:Integer):Float+>x::Float,p)_
  $SparseUnivariatePolynomialFunctions2(Integer, Float)
  (pi())$Float

```

— HACKPI.dotabb —

```

"HACKPI" [color="#88FF44",href="bookvol10.3.pdf#nameddest=HACKPI"]
"ALIST"  [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"HACKPI" -> "ALIST"

```

domain ACLOT PlaneAlgebraicCurvePlot

— PlaneAlgebraicCurvePlot.input —

```

)set break resume

```

```

)sys rm -f PlaneAlgebraicCurvePlot.output
)spool PlaneAlgebraicCurvePlot.output
)set message test on
)set message auto off
)clear all

--S 1 of 5
sketch:=makeSketch(x+y,x,y,-1/2..1/2,-1/2..1/2)$ACPLOT
--R
--R (1) ACPLLOT
--R      1      1      1      1
--R      y + x = 0,  - - <= x <= -,  - - <= y <= -
--R      2      2      2      2
--R      [0.5,- 0.5]
--R      [- 0.5,0.5]
--R
--R                                          Type: PlaneAlgebraicCurvePlot
--E 1

--S 2 of 5
refined:=refine(sketch,0.1)
--R
--R (2) ACPLLOT
--R      1      1      1      1
--R      y + x = 0,  - - <= x <= -,  - - <= y <= -
--R      2      2      2      2
--R      [0.5,- 0.5]
--R      [0.49600000000000083,- 0.49600000000000083]
--R      [0.49200000000000083,- 0.49200000000000083]
--R      [0.48800000000000082,- 0.48800000000000082]
--R      [0.48400000000000082,- 0.48400000000000082]
--R      [0.48000000000000081,- 0.48000000000000081]
--R      [0.47600000000000081,- 0.47600000000000081]
--R      [0.47200000000000081,- 0.47200000000000081]
--R      [0.4680000000000008,- 0.4680000000000008]
--R      [0.4640000000000008,- 0.4640000000000008]
--R      [0.4600000000000008,- 0.4600000000000008]
--R      [0.45600000000000079,- 0.45600000000000079]
--R      [0.45200000000000079,- 0.45200000000000079]
--R      [0.44800000000000079,- 0.44800000000000079]
--R      [0.44400000000000078,- 0.44400000000000078]
--R      [0.44000000000000078,- 0.44000000000000078]
--R      [0.43600000000000078,- 0.43600000000000078]
--R      [0.43200000000000077,- 0.43200000000000077]
--R      [0.42800000000000077,- 0.42800000000000077]
--R      [0.42400000000000077,- 0.42400000000000077]
--R      [0.42000000000000076,- 0.42000000000000076]
--R      [0.41600000000000076,- 0.41600000000000076]
--R      [0.41200000000000075,- 0.41200000000000075]
--R      [0.40800000000000075,- 0.40800000000000075]
--R      [0.40400000000000075,- 0.40400000000000075]

```

```
--R      [0.40000000000000074,- 0.40000000000000074]
--R      [0.39600000000000074,- 0.39600000000000074]
--R      [0.39200000000000074,- 0.39200000000000074]
--R      [0.38800000000000073,- 0.38800000000000073]
--R      [0.38400000000000073,- 0.38400000000000073]
--R      [0.38000000000000073,- 0.38000000000000073]
--R      [0.37600000000000072,- 0.37600000000000072]
--R      [0.37200000000000072,- 0.37200000000000072]
--R      [0.36800000000000072,- 0.36800000000000072]
--R      [0.36400000000000071,- 0.36400000000000071]
--R      [0.36000000000000071,- 0.36000000000000071]
--R      [0.3560000000000007,- 0.3560000000000007]
--R      [0.3520000000000007,- 0.3520000000000007]
--R      [0.3480000000000007,- 0.3480000000000007]
--R      [0.34400000000000069,- 0.34400000000000069]
--R      [0.34000000000000069,- 0.34000000000000069]
--R      [0.33600000000000069,- 0.33600000000000069]
--R      [0.33200000000000068,- 0.33200000000000068]
--R      [0.32800000000000068,- 0.32800000000000068]
--R      [0.32400000000000068,- 0.32400000000000068]
--R      [0.32000000000000067,- 0.32000000000000067]
--R      [0.31600000000000067,- 0.31600000000000067]
--R      [0.31200000000000067,- 0.31200000000000067]
--R      [0.30800000000000066,- 0.30800000000000066]
--R      [0.30400000000000066,- 0.30400000000000066]
--R      [0.30000000000000066,- 0.30000000000000066]
--R      [0.29600000000000065,- 0.29600000000000065]
--R      [0.29200000000000065,- 0.29200000000000065]
--R      [0.28800000000000064,- 0.28800000000000064]
--R      [0.28400000000000064,- 0.28400000000000064]
--R      [0.28000000000000064,- 0.28000000000000064]
--R      [0.27600000000000063,- 0.27600000000000063]
--R      [0.27200000000000063,- 0.27200000000000063]
--R      [0.26800000000000063,- 0.26800000000000063]
--R      [0.26400000000000062,- 0.26400000000000062]
--R      [0.26000000000000062,- 0.26000000000000062]
--R      [0.25600000000000062,- 0.25600000000000062]
--R      [0.25200000000000061,- 0.25200000000000061]
--R      [0.24800000000000061,- 0.24800000000000061]
--R      [0.24400000000000061,- 0.24400000000000061]
--R      [0.2400000000000006,- 0.2400000000000006]
--R      [0.2360000000000006,- 0.2360000000000006]
--R      [0.23200000000000059,- 0.23200000000000059]
--R      [0.22800000000000059,- 0.22800000000000059]
--R      [0.22400000000000059,- 0.22400000000000059]
--R      [0.22000000000000058,- 0.22000000000000058]
--R      [0.21600000000000058,- 0.21600000000000058]
--R      [0.21200000000000058,- 0.21200000000000058]
--R      [0.20800000000000057,- 0.20800000000000057]
--R      [0.20400000000000057,- 0.20400000000000057]
```



```

--R      [0.20000000000000057,- 0.20000000000000057]
--R      [0.19600000000000056,- 0.19600000000000056]
--R      [0.19200000000000056,- 0.19200000000000056]
--R      [0.18800000000000056,- 0.18800000000000056]
--R      [0.18400000000000055,- 0.18400000000000055]
--R      [0.18000000000000055,- 0.18000000000000055]
--R      [0.17600000000000054,- 0.17600000000000054]
--R      [0.17200000000000054,- 0.17200000000000054]
--R      [0.16800000000000054,- 0.16800000000000054]
--R      [0.16400000000000053,- 0.16400000000000053]
--R      [0.16000000000000053,- 0.16000000000000053]
--R      [0.15600000000000053,- 0.15600000000000053]
--R      [0.15200000000000052,- 0.15200000000000052]
--R      [0.14800000000000052,- 0.14800000000000052]
--R      [0.14400000000000052,- 0.14400000000000052]
--R      [0.14000000000000051,- 0.14000000000000051]
--R      [0.13600000000000051,- 0.13600000000000051]
--R      [0.13200000000000051,- 0.13200000000000051]
--R      [0.1280000000000005,- 0.1280000000000005]
--R      [0.1240000000000005,- 0.1240000000000005]
--R      [0.1200000000000005,- 0.1200000000000005]
--R      [0.11600000000000049,- 0.11600000000000049]
--R      [0.11200000000000049,- 0.11200000000000049]
--R      [0.10800000000000048,- 0.10800000000000048]
--R      [0.10400000000000048,- 0.10400000000000048]
--R      [0.10000000000000048,- 0.10000000000000048]
--R      [9.6000000000000474E-2,- 9.6000000000000474E-2]
--R      [9.200000000000047E-2,- 9.200000000000047E-2]
--R      [8.8000000000000467E-2,- 8.8000000000000467E-2]
--R      [8.4000000000000463E-2,- 8.4000000000000463E-2]
--R      [8.000000000000046E-2,- 8.000000000000046E-2]
--R      [7.6000000000000456E-2,- 7.6000000000000456E-2]
--R      [7.2000000000000453E-2,- 7.2000000000000453E-2]
--R      [6.8000000000000449E-2,- 6.8000000000000449E-2]
--R      [6.4000000000000445E-2,- 6.4000000000000445E-2]
--R      [6.0000000000000442E-2,- 6.0000000000000442E-2]
--R      [5.6000000000000438E-2,- 5.6000000000000438E-2]
--R      [5.2000000000000435E-2,- 5.2000000000000435E-2]
--R      [4.8000000000000431E-2,- 4.8000000000000431E-2]
--R      [4.4000000000000428E-2,- 4.4000000000000428E-2]
--R      [4.0000000000000424E-2,- 4.0000000000000424E-2]
--R      [3.6000000000000421E-2,- 3.6000000000000421E-2]
--R      [3.2000000000000417E-2,- 3.2000000000000417E-2]
--R      [2.8000000000000417E-2,- 2.8000000000000417E-2]
--R      [2.4000000000000417E-2,- 2.4000000000000417E-2]
--R      [2.0000000000000417E-2,- 2.0000000000000417E-2]
--R      [1.6000000000000417E-2,- 1.6000000000000417E-2]
--R      [1.2000000000000417E-2,- 1.2000000000000417E-2]
--R      [8.000000000004165E-3,- 8.000000000004165E-3]
--R      [4.000000000004164E-3,- 4.000000000004164E-3]

```

```
--R      [4.163336342344337E-16,- 4.163336342344337E-16]
--R      [- 3.999999999995837E-3,3.999999999995837E-3]
--R      [- 7.999999999995838E-3,7.999999999995838E-3]
--R      [- 1.19999999999584E-2,1.19999999999584E-2]
--R      [- 1.59999999999584E-2,1.59999999999584E-2]
--R      [- 1.99999999999584E-2,1.99999999999584E-2]
--R      [- 2.39999999999584E-2,2.39999999999584E-2]
--R      [- 2.79999999999584E-2,2.79999999999584E-2]
--R      [- 3.19999999999584E-2,3.19999999999584E-2]
--R      [- 3.59999999999588E-2,3.59999999999588E-2]
--R      [- 3.99999999999591E-2,3.99999999999591E-2]
--R      [- 4.39999999999595E-2,4.39999999999595E-2]
--R      [- 4.79999999999599E-2,4.79999999999599E-2]
--R      [- 5.19999999999602E-2,5.19999999999602E-2]
--R      [- 5.59999999999606E-2,5.59999999999606E-2]
--R      [- 5.99999999999609E-2,5.99999999999609E-2]
--R      [- 6.39999999999613E-2,6.39999999999613E-2]
--R      [- 6.79999999999616E-2,6.79999999999616E-2]
--R      [- 7.1999999999962E-2,7.1999999999962E-2]
--R      [- 7.59999999999623E-2,7.59999999999623E-2]
--R      [- 7.99999999999627E-2,7.99999999999627E-2]
--R      [- 8.39999999999631E-2,8.39999999999631E-2]
--R      [- 8.79999999999634E-2,8.79999999999634E-2]
--R      [- 9.19999999999638E-2,9.19999999999638E-2]
--R      [- 9.59999999999641E-2,9.59999999999641E-2]
--R      [- 9.99999999999645E-2,9.99999999999645E-2]
--R      [- 0.103999999999965,0.103999999999965]
--R      [- 0.107999999999965,0.107999999999965]
--R      [- 0.111999999999966,0.111999999999966]
--R      [- 0.115999999999966,0.115999999999966]
--R      [- 0.119999999999966,0.119999999999966]
--R      [- 0.123999999999967,0.123999999999967]
--R      [- 0.127999999999967,0.127999999999967]
--R      [- 0.131999999999967,0.131999999999967]
--R      [- 0.135999999999968,0.135999999999968]
--R      [- 0.139999999999968,0.139999999999968]
--R      [- 0.143999999999968,0.143999999999968]
--R      [- 0.147999999999969,0.147999999999969]
--R      [- 0.151999999999969,0.151999999999969]
--R      [- 0.155999999999969,0.155999999999969]
--R      [- 0.15999999999997,0.15999999999997]
--R      [- 0.16399999999997,0.16399999999997]
--R      [- 0.167999999999971,0.167999999999971]
--R      [- 0.171999999999971,0.171999999999971]
--R      [- 0.175999999999971,0.175999999999971]
--R      [- 0.179999999999972,0.179999999999972]
--R      [- 0.183999999999972,0.183999999999972]
--R      [- 0.187999999999972,0.187999999999972]
--R      [- 0.191999999999973,0.191999999999973]
--R      [- 0.195999999999973,0.195999999999973]
```

```
--R      [- 0.1999999999999973,0.1999999999999973]
--R      [- 0.2039999999999974,0.2039999999999974]
--R      [- 0.2079999999999974,0.2079999999999974]
--R      [- 0.2119999999999974,0.2119999999999974]
--R      [- 0.2159999999999975,0.2159999999999975]
--R      [- 0.2199999999999975,0.2199999999999975]
--R      [- 0.2239999999999975,0.2239999999999975]
--R      [- 0.2279999999999976,0.2279999999999976]
--R      [- 0.2319999999999976,0.2319999999999976]
--R      [- 0.2359999999999977,0.2359999999999977]
--R      [- 0.2399999999999977,0.2399999999999977]
--R      [- 0.2439999999999977,0.2439999999999977]
--R      [- 0.2479999999999978,0.2479999999999978]
--R      [- 0.2519999999999978,0.2519999999999978]
--R      [- 0.2559999999999978,0.2559999999999978]
--R      [- 0.2599999999999979,0.2599999999999979]
--R      [- 0.2639999999999979,0.2639999999999979]
--R      [- 0.2679999999999979,0.2679999999999979]
--R      [- 0.271999999999998,0.271999999999998]
--R      [- 0.275999999999998,0.275999999999998]
--R      [- 0.279999999999998,0.279999999999998]
--R      [- 0.2839999999999981,0.2839999999999981]
--R      [- 0.2879999999999981,0.2879999999999981]
--R      [- 0.2919999999999982,0.2919999999999982]
--R      [- 0.2959999999999982,0.2959999999999982]
--R      [- 0.2999999999999982,0.2999999999999982]
--R      [- 0.3039999999999983,0.3039999999999983]
--R      [- 0.3079999999999983,0.3079999999999983]
--R      [- 0.3119999999999983,0.3119999999999983]
--R      [- 0.3159999999999984,0.3159999999999984]
--R      [- 0.3199999999999984,0.3199999999999984]
--R      [- 0.3239999999999984,0.3239999999999984]
--R      [- 0.3279999999999985,0.3279999999999985]
--R      [- 0.3319999999999985,0.3319999999999985]
--R      [- 0.3359999999999985,0.3359999999999985]
--R      [- 0.3399999999999986,0.3399999999999986]
--R      [- 0.3439999999999986,0.3439999999999986]
--R      [- 0.3479999999999986,0.3479999999999986]
--R      [- 0.3519999999999987,0.3519999999999987]
--R      [- 0.3559999999999987,0.3559999999999987]
--R      [- 0.3599999999999988,0.3599999999999988]
--R      [- 0.3639999999999988,0.3639999999999988]
--R      [- 0.3679999999999988,0.3679999999999988]
--R      [- 0.3719999999999989,0.3719999999999989]
--R      [- 0.3759999999999989,0.3759999999999989]
--R      [- 0.3799999999999989,0.3799999999999989]
--R      [- 0.383999999999999,0.383999999999999]
--R      [- 0.387999999999999,0.387999999999999]
--R      [- 0.391999999999999,0.391999999999999]
--R      [- 0.3959999999999991,0.3959999999999991]
```

```

--R      [- 0.39999999999999991,0.39999999999999991]
--R      [- 0.40399999999999991,0.40399999999999991]
--R      [- 0.40799999999999992,0.40799999999999992]
--R      [- 0.41199999999999992,0.41199999999999992]
--R      [- 0.41599999999999993,0.41599999999999993]
--R      [- 0.41999999999999993,0.41999999999999993]
--R      [- 0.42399999999999993,0.42399999999999993]
--R      [- 0.42799999999999994,0.42799999999999994]
--R      [- 0.43199999999999994,0.43199999999999994]
--R      [- 0.43599999999999994,0.43599999999999994]
--R      [- 0.43999999999999995,0.43999999999999995]
--R      [- 0.44399999999999995,0.44399999999999995]
--R      [- 0.44799999999999995,0.44799999999999995]
--R      [- 0.45199999999999996,0.45199999999999996]
--R      [- 0.45599999999999996,0.45599999999999996]
--R      [- 0.45999999999999996,0.45999999999999996]
--R      [- 0.46399999999999997,0.46399999999999997]
--R      [- 0.46799999999999997,0.46799999999999997]
--R      [- 0.47199999999999998,0.47199999999999998]
--R      [- 0.47599999999999998,0.47599999999999998]
--R      [- 0.47999999999999998,0.47999999999999998]
--R      [- 0.48399999999999999,0.48399999999999999]
--R      [- 0.48799999999999999,0.48799999999999999]
--R      [- 0.49199999999999999,0.49199999999999999]
--R      [- 0.496,0.496]
--R      [- 0.5,0.5]
--R
--R                                          Type: PlaneAlgebraicCurvePlot
--E 2

--S 3 of 5
listBranches(sketch)
--R
--R      (3)  [[0.5,- 0.5],[- 0.5,0.5]]
--R
--R                                          Type: List(List(Point(DoubleFloat)))
--E 3

--S 4 of 5
listBranches(refined)
--R
--R      (4)
--R      [
--R      [0.5,- 0.5], [0.496000000000000083,- 0.496000000000000083],
--R      [0.492000000000000083,- 0.492000000000000083],
--R      [0.488000000000000082,- 0.488000000000000082],
--R      [0.484000000000000082,- 0.484000000000000082],
--R      [0.480000000000000081,- 0.480000000000000081],
--R      [0.476000000000000081,- 0.476000000000000081],
--R      [0.472000000000000081,- 0.472000000000000081],
--R      [0.46800000000000008,- 0.46800000000000008],
--R      [0.46400000000000008,- 0.46400000000000008],
--R      ]

```

```
--R [0.4600000000000008,- 0.4600000000000008],
--R [0.45600000000000079,- 0.45600000000000079],
--R [0.45200000000000079,- 0.45200000000000079],
--R [0.44800000000000079,- 0.44800000000000079],
--R [0.44400000000000078,- 0.44400000000000078],
--R [0.44000000000000078,- 0.44000000000000078],
--R [0.43600000000000078,- 0.43600000000000078],
--R [0.43200000000000077,- 0.43200000000000077],
--R [0.42800000000000077,- 0.42800000000000077],
--R [0.42400000000000077,- 0.42400000000000077],
--R [0.42000000000000076,- 0.42000000000000076],
--R [0.41600000000000076,- 0.41600000000000076],
--R [0.41200000000000075,- 0.41200000000000075],
--R [0.40800000000000075,- 0.40800000000000075],
--R [0.40400000000000075,- 0.40400000000000075],
--R [0.40000000000000074,- 0.40000000000000074],
--R [0.39600000000000074,- 0.39600000000000074],
--R [0.39200000000000074,- 0.39200000000000074],
--R [0.38800000000000073,- 0.38800000000000073],
--R [0.38400000000000073,- 0.38400000000000073],
--R [0.38000000000000073,- 0.38000000000000073],
--R [0.37600000000000072,- 0.37600000000000072],
--R [0.37200000000000072,- 0.37200000000000072],
--R [0.36800000000000072,- 0.36800000000000072],
--R [0.36400000000000071,- 0.36400000000000071],
--R [0.36000000000000071,- 0.36000000000000071],
--R [0.3560000000000007,- 0.3560000000000007],
--R [0.3520000000000007,- 0.3520000000000007],
--R [0.3480000000000007,- 0.3480000000000007],
--R [0.34400000000000069,- 0.34400000000000069],
--R [0.34000000000000069,- 0.34000000000000069],
--R [0.33600000000000069,- 0.33600000000000069],
--R [0.33200000000000068,- 0.33200000000000068],
--R [0.32800000000000068,- 0.32800000000000068],
--R [0.32400000000000068,- 0.32400000000000068],
--R [0.32000000000000067,- 0.32000000000000067],
--R [0.31600000000000067,- 0.31600000000000067],
--R [0.31200000000000067,- 0.31200000000000067],
--R [0.30800000000000066,- 0.30800000000000066],
--R [0.30400000000000066,- 0.30400000000000066],
--R [0.30000000000000066,- 0.30000000000000066],
--R [0.29600000000000065,- 0.29600000000000065],
--R [0.29200000000000065,- 0.29200000000000065],
--R [0.28800000000000064,- 0.28800000000000064],
--R [0.28400000000000064,- 0.28400000000000064],
--R [0.28000000000000064,- 0.28000000000000064],
--R [0.27600000000000063,- 0.27600000000000063],
--R [0.27200000000000063,- 0.27200000000000063],
--R [0.26800000000000063,- 0.26800000000000063],
--R [0.26400000000000062,- 0.26400000000000062],
```

```

--R      [0.260000000000000062,- 0.260000000000000062],
--R      [0.256000000000000062,- 0.256000000000000062],
--R      [0.252000000000000061,- 0.252000000000000061],
--R      [0.248000000000000061,- 0.248000000000000061],
--R      [0.244000000000000061,- 0.244000000000000061],
--R      [0.24000000000000006,- 0.24000000000000006],
--R      [0.23600000000000006,- 0.23600000000000006],
--R      [0.232000000000000059,- 0.232000000000000059],
--R      [0.228000000000000059,- 0.228000000000000059],
--R      [0.224000000000000059,- 0.224000000000000059],
--R      [0.220000000000000058,- 0.220000000000000058],
--R      [0.216000000000000058,- 0.216000000000000058],
--R      [0.212000000000000058,- 0.212000000000000058],
--R      [0.208000000000000057,- 0.208000000000000057],
--R      [0.204000000000000057,- 0.204000000000000057],
--R      [0.200000000000000057,- 0.200000000000000057],
--R      [0.196000000000000056,- 0.196000000000000056],
--R      [0.192000000000000056,- 0.192000000000000056],
--R      [0.188000000000000056,- 0.188000000000000056],
--R      [0.184000000000000055,- 0.184000000000000055],
--R      [0.180000000000000055,- 0.180000000000000055],
--R      [0.176000000000000054,- 0.176000000000000054],
--R      [0.172000000000000054,- 0.172000000000000054],
--R      [0.168000000000000054,- 0.168000000000000054],
--R      [0.164000000000000053,- 0.164000000000000053],
--R      [0.160000000000000053,- 0.160000000000000053],
--R      [0.156000000000000053,- 0.156000000000000053],
--R      [0.152000000000000052,- 0.152000000000000052],
--R      [0.148000000000000052,- 0.148000000000000052],
--R      [0.144000000000000052,- 0.144000000000000052],
--R      [0.140000000000000051,- 0.140000000000000051],
--R      [0.136000000000000051,- 0.136000000000000051],
--R      [0.132000000000000051,- 0.132000000000000051],
--R      [0.12800000000000005,- 0.12800000000000005],
--R      [0.12400000000000005,- 0.12400000000000005],
--R      [0.12000000000000005,- 0.12000000000000005],
--R      [0.116000000000000049,- 0.116000000000000049],
--R      [0.112000000000000049,- 0.112000000000000049],
--R      [0.108000000000000048,- 0.108000000000000048],
--R      [0.104000000000000048,- 0.104000000000000048],
--R      [0.100000000000000048,- 0.100000000000000048],
--R      [9.6000000000000474E-2,- 9.6000000000000474E-2],
--R      [9.200000000000047E-2,- 9.200000000000047E-2],
--R      [8.8000000000000467E-2,- 8.8000000000000467E-2],
--R      [8.4000000000000463E-2,- 8.4000000000000463E-2],
--R      [8.000000000000046E-2,- 8.000000000000046E-2],
--R      [7.6000000000000456E-2,- 7.6000000000000456E-2],
--R      [7.2000000000000453E-2,- 7.2000000000000453E-2],
--R      [6.8000000000000449E-2,- 6.8000000000000449E-2],
--R      [6.4000000000000445E-2,- 6.4000000000000445E-2],

```

```

--R      [6.0000000000000442E-2,- 6.0000000000000442E-2] ,
--R      [5.6000000000000438E-2,- 5.6000000000000438E-2] ,
--R      [5.2000000000000435E-2,- 5.2000000000000435E-2] ,
--R      [4.8000000000000431E-2,- 4.8000000000000431E-2] ,
--R      [4.4000000000000428E-2,- 4.4000000000000428E-2] ,
--R      [4.0000000000000424E-2,- 4.0000000000000424E-2] ,
--R      [3.6000000000000421E-2,- 3.6000000000000421E-2] ,
--R      [3.2000000000000417E-2,- 3.2000000000000417E-2] ,
--R      [2.8000000000000417E-2,- 2.8000000000000417E-2] ,
--R      [2.4000000000000417E-2,- 2.4000000000000417E-2] ,
--R      [2.0000000000000417E-2,- 2.0000000000000417E-2] ,
--R      [1.6000000000000417E-2,- 1.6000000000000417E-2] ,
--R      [1.2000000000000417E-2,- 1.2000000000000417E-2] ,
--R      [8.00000000000004165E-3,- 8.00000000000004165E-3] ,
--R      [4.00000000000004164E-3,- 4.00000000000004164E-3] ,
--R      [4.163336342344337E-16,- 4.163336342344337E-16] ,
--R      [- 3.9999999999995837E-3,3.9999999999995837E-3] ,
--R      [- 7.9999999999995838E-3,7.9999999999995838E-3] ,
--R      [- 1.199999999999584E-2,1.199999999999584E-2] ,
--R      [- 1.599999999999584E-2,1.599999999999584E-2] ,
--R      [- 1.999999999999584E-2,1.999999999999584E-2] ,
--R      [- 2.399999999999584E-2,2.399999999999584E-2] ,
--R      [- 2.799999999999584E-2,2.799999999999584E-2] ,
--R      [- 3.199999999999584E-2,3.199999999999584E-2] ,
--R      [- 3.599999999999588E-2,3.599999999999588E-2] ,
--R      [- 3.999999999999591E-2,3.999999999999591E-2] ,
--R      [- 4.399999999999595E-2,4.399999999999595E-2] ,
--R      [- 4.799999999999599E-2,4.799999999999599E-2] ,
--R      [- 5.199999999999602E-2,5.199999999999602E-2] ,
--R      [- 5.599999999999606E-2,5.599999999999606E-2] ,
--R      [- 5.999999999999609E-2,5.999999999999609E-2] ,
--R      [- 6.399999999999613E-2,6.399999999999613E-2] ,
--R      [- 6.799999999999616E-2,6.799999999999616E-2] ,
--R      [- 7.19999999999962E-2,7.19999999999962E-2] ,
--R      [- 7.599999999999623E-2,7.599999999999623E-2] ,
--R      [- 7.999999999999627E-2,7.999999999999627E-2] ,
--R      [- 8.399999999999631E-2,8.399999999999631E-2] ,
--R      [- 8.799999999999634E-2,8.799999999999634E-2] ,
--R      [- 9.199999999999638E-2,9.199999999999638E-2] ,
--R      [- 9.599999999999641E-2,9.599999999999641E-2] ,
--R      [- 9.999999999999645E-2,9.999999999999645E-2] ,
--R      [- 0.1039999999999965,0.1039999999999965] ,
--R      [- 0.1079999999999965,0.1079999999999965] ,
--R      [- 0.1119999999999966,0.1119999999999966] ,
--R      [- 0.1159999999999966,0.1159999999999966] ,
--R      [- 0.1199999999999966,0.1199999999999966] ,
--R      [- 0.1239999999999967,0.1239999999999967] ,
--R      [- 0.1279999999999967,0.1279999999999967] ,
--R      [- 0.1319999999999967,0.1319999999999967] ,
--R      [- 0.1359999999999968,0.1359999999999968] ,

```

```
--R      [- 0.1399999999999968,0.1399999999999968],
--R      [- 0.1439999999999968,0.1439999999999968],
--R      [- 0.1479999999999969,0.1479999999999969],
--R      [- 0.1519999999999969,0.1519999999999969],
--R      [- 0.1559999999999969,0.1559999999999969],
--R      [- 0.159999999999997,0.159999999999997],
--R      [- 0.163999999999997,0.163999999999997],
--R      [- 0.1679999999999971,0.1679999999999971],
--R      [- 0.1719999999999971,0.1719999999999971],
--R      [- 0.1759999999999971,0.1759999999999971],
--R      [- 0.1799999999999972,0.1799999999999972],
--R      [- 0.1839999999999972,0.1839999999999972],
--R      [- 0.1879999999999972,0.1879999999999972],
--R      [- 0.1919999999999973,0.1919999999999973],
--R      [- 0.1959999999999973,0.1959999999999973],
--R      [- 0.1999999999999973,0.1999999999999973],
--R      [- 0.2039999999999974,0.2039999999999974],
--R      [- 0.2079999999999974,0.2079999999999974],
--R      [- 0.2119999999999974,0.2119999999999974],
--R      [- 0.2159999999999975,0.2159999999999975],
--R      [- 0.2199999999999975,0.2199999999999975],
--R      [- 0.2239999999999975,0.2239999999999975],
--R      [- 0.2279999999999976,0.2279999999999976],
--R      [- 0.2319999999999976,0.2319999999999976],
--R      [- 0.2359999999999977,0.2359999999999977],
--R      [- 0.2399999999999977,0.2399999999999977],
--R      [- 0.2439999999999977,0.2439999999999977],
--R      [- 0.2479999999999978,0.2479999999999978],
--R      [- 0.2519999999999978,0.2519999999999978],
--R      [- 0.2559999999999978,0.2559999999999978],
--R      [- 0.2599999999999979,0.2599999999999979],
--R      [- 0.2639999999999979,0.2639999999999979],
--R      [- 0.2679999999999979,0.2679999999999979],
--R      [- 0.271999999999998,0.271999999999998],
--R      [- 0.275999999999998,0.275999999999998],
--R      [- 0.279999999999998,0.279999999999998],
--R      [- 0.2839999999999981,0.2839999999999981],
--R      [- 0.2879999999999981,0.2879999999999981],
--R      [- 0.2919999999999982,0.2919999999999982],
--R      [- 0.2959999999999982,0.2959999999999982],
--R      [- 0.2999999999999982,0.2999999999999982],
--R      [- 0.3039999999999983,0.3039999999999983],
--R      [- 0.3079999999999983,0.3079999999999983],
--R      [- 0.3119999999999983,0.3119999999999983],
--R      [- 0.3159999999999984,0.3159999999999984],
--R      [- 0.3199999999999984,0.3199999999999984],
--R      [- 0.3239999999999984,0.3239999999999984],
--R      [- 0.3279999999999985,0.3279999999999985],
--R      [- 0.3319999999999985,0.3319999999999985],
--R      [- 0.3359999999999985,0.3359999999999985],
```



```

--R      [- 0.3399999999999986,0.3399999999999986],
--R      [- 0.3439999999999986,0.3439999999999986],
--R      [- 0.3479999999999986,0.3479999999999986],
--R      [- 0.3519999999999987,0.3519999999999987],
--R      [- 0.3559999999999987,0.3559999999999987],
--R      [- 0.3599999999999988,0.3599999999999988],
--R      [- 0.3639999999999988,0.3639999999999988],
--R      [- 0.3679999999999988,0.3679999999999988],
--R      [- 0.3719999999999989,0.3719999999999989],
--R      [- 0.3759999999999989,0.3759999999999989],
--R      [- 0.3799999999999989,0.3799999999999989],
--R      [- 0.383999999999999,0.383999999999999],
--R      [- 0.387999999999999,0.387999999999999],
--R      [- 0.391999999999999,0.391999999999999],
--R      [- 0.3959999999999991,0.3959999999999991],
--R      [- 0.3999999999999991,0.3999999999999991],
--R      [- 0.4039999999999991,0.4039999999999991],
--R      [- 0.4079999999999992,0.4079999999999992],
--R      [- 0.4119999999999992,0.4119999999999992],
--R      [- 0.4159999999999993,0.4159999999999993],
--R      [- 0.4199999999999993,0.4199999999999993],
--R      [- 0.4239999999999993,0.4239999999999993],
--R      [- 0.4279999999999994,0.4279999999999994],
--R      [- 0.4319999999999994,0.4319999999999994],
--R      [- 0.4359999999999994,0.4359999999999994],
--R      [- 0.4399999999999995,0.4399999999999995],
--R      [- 0.4439999999999995,0.4439999999999995],
--R      [- 0.4479999999999995,0.4479999999999995],
--R      [- 0.4519999999999996,0.4519999999999996],
--R      [- 0.4559999999999996,0.4559999999999996],
--R      [- 0.4599999999999996,0.4599999999999996],
--R      [- 0.4639999999999997,0.4639999999999997],
--R      [- 0.4679999999999997,0.4679999999999997],
--R      [- 0.4719999999999998,0.4719999999999998],
--R      [- 0.4759999999999998,0.4759999999999998],
--R      [- 0.4799999999999998,0.4799999999999998],
--R      [- 0.4839999999999999,0.4839999999999999],
--R      [- 0.4879999999999999,0.4879999999999999],
--R      [- 0.4919999999999999,0.4919999999999999], [- 0.496,0.496],
--R      [- 0.5,0.5]]
--R      ]
--R
--R                                          Type: List(List(Point(DoubleFloat)))
--E 4

```

```

--S 5 of 5
)show ACPLLOT
--R
--R PlaneAlgebraicCurvePlot is a domain constructor
--R Abbreviation for PlaneAlgebraicCurvePlot is ACPLLOT
--R This constructor is not exposed in this frame.

```

```

--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ACPLLOT
--R
--R----- Operations -----
--R coerce : % -> OutputForm          refine : (% , DoubleFloat) -> %
--R listBranches : % -> List(List(Point(DoubleFloat)))
--R makeSketch : (Polynomial(Integer), Symbol, Symbol, Segment(Fraction(Integer)), Segment(Fraction(Integer)))
--R xRange : % -> Segment(DoubleFloat)
--R yRange : % -> Segment(DoubleFloat)
--R
--E 5

```

```

)spool
)lisp (bye)

```

— PlaneAlgebraicCurvePlot.help —

```

=====
PlaneAlgebraicCurvePlot examples
=====

```

Plot a NON-SINGULAR plane algebraic curve $p(x,y) = 0$.

```
sketch:=makeSketch(x+y,x,y,-1/2..1/2,-1/2..1/2)$ACPLLOT
```

```

          ACPLLOT
          1      1      1      1
y + x = 0,  - - <= x <= -,  - - <= y <= -
          2      2      2      2
          [0.5,- 0.5]
          [- 0.5,0.5]

```

```
refined:=refine(sketch,0.1)
```

```

          ACPLLOT
          1      1      1      1
y + x = 0,  - - <= x <= -,  - - <= y <= -
          2      2      2      2
          [0.5,- 0.5]
[0.496000000000000083,- 0.496000000000000083]
[0.492000000000000083,- 0.492000000000000083]
[0.488000000000000082,- 0.488000000000000082]
[0.484000000000000082,- 0.484000000000000082]
...
[- 0.48399999999999999,0.48399999999999999]
[- 0.48799999999999999,0.48799999999999999]

```

```

[- 0.4919999999999999,0.4919999999999999]
      [- 0.496,0.496]
      [- 0.5,0.5]

listBranches(sketch)

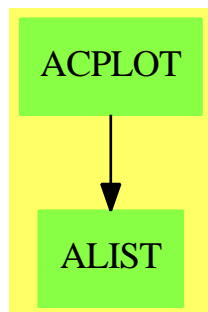
[[[0.5,- 0.5],[- 0.5,0.5]]]

listBranches(refined)

[
  [[0.5,- 0.5], [0.496000000000000083,- 0.496000000000000083],
   [0.492000000000000083,- 0.492000000000000083],
   [0.488000000000000082,- 0.488000000000000082],
   ...
  [- 0.4839999999999999,0.4839999999999999],
  [- 0.4879999999999999,0.4879999999999999],
  [- 0.4919999999999999,0.4919999999999999], [- 0.496,0.496],

```

PlaneAlgebraicCurvePlot (ACPLOT)



Exports:

coerce listBranches makeSketch refine xRange yRange

— domain ACPLOT PlaneAlgebraicCurvePlot —

```

)abbrev domain ACPLOT PlaneAlgebraicCurvePlot
++ Author: Clifton J. Williamson and Timothy Daly
++ Date Created: Fall 1988
++ Date Last Updated: 27 April 1990

```

```

++ Description:
++ Plot a NON-SINGULAR plane algebraic curve  $p(x,y) = 0$ .

PlaneAlgebraicCurvePlot(): PlottablePlaneCurveCategory _
with

makeSketch:(Polynomial Integer,Symbol,Symbol,Segment Fraction Integer,_
              Segment Fraction Integer) -> %
    ++ makeSketch(p,x,y,a..b,c..d) creates an ACLOT of the
    ++ curve \spad{p = 0} in the region  $a \leq x \leq b$ ,  $c \leq y \leq d$ .
    ++ More specifically, 'makeSketch' plots a non-singular algebraic curve
    ++ \spad{p = 0} in an rectangular region  $xMin \leq x \leq xMax$ ,
    ++  $yMin \leq y \leq yMax$ . The user inputs
    ++ \spad{makeSketch(p,x,y,xMin..xMax,yMin..yMax)}.
    ++ Here p is a polynomial in the variables x and y with
    ++ integer coefficients (p belongs to the domain
    ++ \spad{Polynomial Integer}). The case
    ++ where p is a polynomial in only one of the variables is
    ++ allowed. The variables x and y are input to specify the
    ++ the coordinate axes. The horizontal axis is the x-axis and
    ++ the vertical axis is the y-axis. The rational numbers
    ++ xMin,...,yMax specify the boundaries of the region in
    ++ which the curve is to be plotted.
    ++
    ++X makeSketch(x+y,x,y,-1/2..1/2,-1/2..1/2)$ACLOT

refine:(%,DoubleFloat) -> %
    ++ refine(p,x) is not documented
    ++
    ++X sketch:=makeSketch(x+y,x,y,-1/2..1/2,-1/2..1/2)$ACLOT
    ++X refined:=refine(sketch,0.1)

== add

import PointPackage DoubleFloat
import Plot
import RealSolvePackage

BoundaryPts ==> Record(left: List Point DoubleFloat,_
                       right: List Point DoubleFloat,_
                       bottom: List Point DoubleFloat,_
                       top: List Point DoubleFloat)

NewPtInfo ==> Record(newPt: Point DoubleFloat,_
                    type: String)

Corners ==> Record(minXVal: DoubleFloat,_
                  maxXVal: DoubleFloat,_
                  minYVal: DoubleFloat,_
                  maxYVal: DoubleFloat)

```

```

kinte      ==> solve$RealSolvePackage()

rsolve     ==> realSolve$RealSolvePackage()

singValBetween?:(DoubleFloat,DoubleFloat,List DoubleFloat) -> Boolean

segmentInfo:(DoubleFloat -> DoubleFloat,DoubleFloat,DoubleFloat,_,
              List DoubleFloat,List DoubleFloat,List DoubleFloat,_,
              DoubleFloat,DoubleFloat) -> _
  Record(seg:Segment DoubleFloat,_,
         left: DoubleFloat,_,
         lowerVals: List DoubleFloat,_,
         upperVals:List DoubleFloat)

swapCoords:Point DoubleFloat -> Point DoubleFloat

samePlottedPt?:(Point DoubleFloat,Point DoubleFloat) -> Boolean

findPtOnList:(Point DoubleFloat,List Point DoubleFloat) -> _
  Union(Point DoubleFloat,"failed")

makeCorners:(DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat) -> Corners

getXMin: Corners -> DoubleFloat

getXMax: Corners -> DoubleFloat

getYMin: Corners -> DoubleFloat

getYMax: Corners -> DoubleFloat

SFPolyToUPoly:Polynomial DoubleFloat -> _
  SparseUnivariatePolynomial DoubleFloat

RNPolyToUPoly:Polynomial Fraction Integer -> _
  SparseUnivariatePolynomial Fraction Integer

coerceCoefsToSFs:Polynomial Integer -> Polynomial DoubleFloat

coerceCoefsToRNs:Polynomial Integer -> Polynomial Fraction Integer

RNtoSF:Fraction Integer -> DoubleFloat

RNtoNF:Fraction Integer -> Float

SFtoNF:DoubleFloat -> Float

listPtsOnHorizBdry:(Polynomial Fraction Integer,Symbol,Fraction Integer,_,
                   Float,Float) -> _

```

```

List Point DoubleFloat

listPtsOnVertBdry:(Polynomial Fraction Integer,Symbol,Fraction Integer,_
                  Float,Float) -> _
List Point DoubleFloat

listPtsInRect:(List List Float,Float,Float,Float,Float) -> _
List Point DoubleFloat

ptsSuchThat?:(List List Float,List Float -> Boolean) -> Boolean

inRect?:(List Float,Float,Float,Float,Float) -> Boolean

onHorzSeg?:(List Float,Float,Float,Float) -> Boolean

onVertSeg?:(List Float,Float,Float,Float) -> Boolean

newX:(List List Float,List List Float,Float,Float,Float,Fraction Integer,_
      Fraction Integer) -> Fraction Integer

newY:(List List Float,List List Float,Float,Float,Float,_
      Fraction Integer,Fraction Integer) -> Fraction Integer

makeOneVarSketch:(Polynomial Integer,Symbol,Symbol,Fraction Integer,_
                  Fraction Integer,Fraction Integer,Fraction Integer,_
                  Symbol) -> %

makeLineSketch:(Polynomial Integer,Symbol,Symbol,Fraction Integer,_
                Fraction Integer,Fraction Integer,Fraction Integer) -> %

makeRatFcnSketch:(Polynomial Integer,Symbol,Symbol,Fraction Integer,_
                  Fraction Integer,Fraction Integer,Fraction Integer,_
                  Symbol) -> %

makeGeneralSketch:(Polynomial Integer,Symbol,Symbol,Fraction Integer,_
                   Fraction Integer,Fraction Integer,Fraction Integer) -> %

traceBranches:(Polynomial DoubleFloat,Polynomial DoubleFloat,_
               Polynomial DoubleFloat,Symbol,Symbol,Corners,DoubleFloat,_
               DoubleFloat,PositiveInteger, List Point DoubleFloat,_
               BoundaryPts) -> List List Point DoubleFloat

dummyFirstPt:(Point DoubleFloat,Polynomial DoubleFloat,_
              Polynomial DoubleFloat,Symbol,Symbol,List Point DoubleFloat,_
              List Point DoubleFloat,List Point DoubleFloat,_
              List Point DoubleFloat) -> Point DoubleFloat

listPtsOnSegment:(Polynomial DoubleFloat,Polynomial DoubleFloat,_
                  Polynomial DoubleFloat,Symbol,Symbol,Point DoubleFloat,_
                  Point DoubleFloat,Corners, DoubleFloat,DoubleFloat,_

```

```

    PositiveInteger,List Point DoubleFloat,_
    List Point DoubleFloat) -> List List Point DoubleFloat

listPtsOnLoop:(Polynomial DoubleFloat,Polynomial DoubleFloat,_
    Polynomial DoubleFloat,Symbol,Symbol,Point DoubleFloat,_
    Corners, DoubleFloat,DoubleFloat,PositiveInteger,_
    List Point DoubleFloat,List Point DoubleFloat) -> _
    List List Point DoubleFloat

computeNextPt:(Polynomial DoubleFloat,Polynomial DoubleFloat,_
    Polynomial DoubleFloat,Symbol,Symbol,Point DoubleFloat,_
    Point DoubleFloat,Corners, DoubleFloat,DoubleFloat,_
    PositiveInteger,List Point DoubleFloat,_
    List Point DoubleFloat) -> NewPtInfo

newtonApprox:(SparseUnivariatePolynomial DoubleFloat, DoubleFloat, _
    DoubleFloat, PositiveInteger) -> Union(DoubleFloat, "failed")

--% representation

Rep := Record(poly      : Polynomial Integer,_
    xVar      : Symbol,_
    yVar      : Symbol,_
    minXVal   : Fraction Integer,_
    maxXVal   : Fraction Integer,_
    minYVal   : Fraction Integer,_
    maxYVal   : Fraction Integer,_
    bdryPts   : BoundaryPts,_
    hTanPts   : List Point DoubleFloat,_
    vTanPts   : List Point DoubleFloat,_
    branches  : List List Point DoubleFloat)

--% global constants

EPSILON : Float := .000001 -- precision to which realSolve finds roots
PLOTERR : DoubleFloat := float(1,-3,10)
    -- maximum allowable difference in each coordinate when
    -- determining if 2 plotted points are equal

--% global flags

NADA    : String := "nothing in particular"
BDRY    : String := "boundary point"
CRIT    : String := "critical point"
BOTTOM  : String := "bottom"
TOP     : String := "top"

--% hacks

NFtoSF: Float -> DoubleFloat

```

```

NFtoSF x == 0 + convert(x)$Float

--% points
makePt: (DoubleFloat,DoubleFloat) -> Point DoubleFloat
makePt(xx,yy) == point(1 : List DoubleFloat := [xx,yy])

swapCoords(pt) == makePt(yCoord pt,xCoord pt)

samePlottedPt?(p0,p1) ==
  -- determines if p1 lies in a square with side 2 PLOTERR
  -- centered at p0
  x0 := xCoord p0; y0 := yCoord p0
  x1 := xCoord p1; y1 := yCoord p1
  (abs(x1-x0) < PLOTERR) and (abs(y1-y0) < PLOTERR)

findPtOnList(pt,pointList) ==
  for point in pointList repeat
    samePlottedPt?(pt,point) => return point
  "failed"

--% corners

makeCorners(xMinSF,xMaxSF,yMinSF,yMaxSF) ==
  [xMinSF,xMaxSF,yMinSF,yMaxSF]

getXMin(corners) == corners.minXVal
getXMax(corners) == corners.maxXVal
getYMin(corners) == corners.minYVal
getYMax(corners) == corners.maxYVal

--% coercions

SFPolyToUPoly(p) ==
  -- 'p' is of type Polynomial, but has only one variable
  zero? p => 0
  monomial(leadingCoefficient p,totalDegree p) +
    SFPolyToUPoly(reductum p)

RNPolyToUPoly(p) ==
  -- 'p' is of type Polynomial, but has only one variable
  zero? p => 0
  monomial(leadingCoefficient p,totalDegree p) +
    RNPolyToUPoly(reductum p)

coerceCoefsToSFs(p) ==
  -- coefficients of 'p' are coerced to be DoubleFloat's
  map(coerce,p)$PolynomialFunctions2(Integer,DoubleFloat)

coerceCoefsToRNs(p) ==
  -- coefficients of 'p' are coerced to be DoubleFloat's

```



```

map(coerce,p)$PolynomialFunctions2(Integer,Fraction Integer)

RNtoSF(r) == coerce(r)@DoubleFloat
RNtoNF(r) == coerce(r)@Float
SFtoNF(x) == convert(x)@Float

--% computation of special points

listPtsOnHorizBdry(pRN,y,y0,xMinNF,xMaxNF) ==
-- strict inequality here: corners on vertical boundary
pointList : List Point DoubleFloat := nil()
ySF := RNtoSF(y0)
f := eval(pRN,y,y0)
roots : List Float := kinte(f,EPSILON)
for root in roots repeat
  if (xMinNF < root) and (root < xMaxNF) then
    pointList := cons(makePt(NFtoSF root, ySF), pointList)
pointList

listPtsOnVertBdry(pRN,x,x0,yMinNF,yMaxNF) ==
pointList : List Point DoubleFloat := nil()
xSF := RNtoSF(x0)
f := eval(pRN,x,x0)
roots : List Float := kinte(f,EPSILON)
for root in roots repeat
  if (yMinNF <= root) and (root <= yMaxNF) then
    pointList := cons(makePt(xSF, NFtoSF root), pointList)
pointList

listPtsInRect(points,xMin,xMax,yMin,yMax) ==
pointList : List Point DoubleFloat := nil()
for point in points repeat
  xx := first point; yy := second point
  if (xMin<=xx) and (xx<=xMax) and (yMin<=yy) and (yy<=yMax) then
    pointList := cons(makePt(NFtoSF xx,NFtoSF yy),pointList)
pointList

ptsSuchThat?(points,pred) ==
for point in points repeat
  if pred point then return true
false

inRect?(point,xMinNF,xMaxNF,yMinNF,yMaxNF) ==
xx := first point; yy := second point
xMinNF <= xx and xx <= xMaxNF and yMinNF <= yy and yy <= yMaxNF

onHorzSeg?(point,xMinNF,xMaxNF,yNF) ==
xx := first point; yy := second point
yy = yNF and xMinNF <= xx and xx <= xMaxNF

```

```

onVertSeg?(point,yMinNF,yMaxNF,xNF) ==
  xx := first point; yy := second point
  xx = xNF and yMinNF <= yy and yy <= yMaxNF

newX(vtanPts,singPts,yMinNF,yMaxNF,xNF,xRN,horizInc) ==
  xNewNF := xNF + RNtoNF horizInc
  xRtNF := max(xNF,xNewNF); xLftNF := min(xNF,xNewNF)
-- ptsSuchThat?(singPts,inRect?(#1,xLftNF,xRtNF,yMinNF,yMaxNF)) =>
  foo : List Float -> Boolean := x +-> inRect?(x,xLftNF,xRtNF,yMinNF,yMaxNF)
  ptsSuchThat?(singPts,foo) =>
    newX(vtanPts,singPts,yMinNF,yMaxNF,xNF,xRN,_
      horizInc/2::(Fraction Integer))
-- ptsSuchThat?(vtanPts,onVertSeg?(#1,yMinNF,yMaxNF,xNewNF)) =>
  goo : List Float -> Boolean := x +-> onVertSeg?(x,yMinNF,yMaxNF,xNewNF)
  ptsSuchThat?(vtanPts,goo) =>
    newX(vtanPts,singPts,yMinNF,yMaxNF,xNF,xRN,_
      horizInc/2::(Fraction Integer))
  xRN + horizInc

newY(htanPts,singPts,xMinNF,xMaxNF,yNF,yRN,vertInc) ==
  yNewNF := yNF + RNtoNF vertInc
  yTopNF := max(yNF,yNewNF); yBotNF := min(yNF,yNewNF)
-- ptsSuchThat?(singPts,inRect?(#1,xMinNF,xMaxNF,yBotNF,yTopNF)) =>
  foo : List Float -> Boolean := x +-> inRect?(x,xMinNF,xMaxNF,yBotNF,yTopNF)
  ptsSuchThat?(singPts,foo) =>
    newY(htanPts,singPts,xMinNF,xMaxNF,yNF,yRN,_
      vertInc/2::(Fraction Integer))
-- ptsSuchThat?(htanPts,onHorzSeg?(#1,xMinNF,xMaxNF,yNewNF)) =>
  goo : List Float -> Boolean := x +-> onHorzSeg?(x,xMinNF,xMaxNF,yNewNF)
  ptsSuchThat?(htanPts,goo) =>
    newY(htanPts,singPts,xMinNF,xMaxNF,yNF,yRN,_
      vertInc/2::(Fraction Integer))
  yRN + vertInc

--% creation of sketches

makeSketch(p,x,y,xRange,yRange) ==
  xMin := lo xRange; xMax := hi xRange
  yMin := lo yRange; yMax := hi yRange
  -- test input for consistency
  xMax <= xMin =>
    error "makeSketch: bad range for first variable"
  yMax <= yMin =>
    error "makeSketch: bad range for second variable"
  varList := variables p
  # varList > 2 =>
    error "makeSketch: polynomial in more than 2 variables"
  # varList = 0 =>
    error "makeSketch: constant polynomial"
  -- polynomial in 1 variable

```

```

# varList = 1 =>
  (not member?(x,varList)) and (not member?(y,varList)) =>
    error "makeSketch: bad variables"
  makeOneVarSketch(p,x,y,xMin,xMax,yMin,yMax,first varList)
-- polynomial in 2 variables
(not member?(x,varList)) or (not member?(y,varList)) =>
  error "makeSketch: bad variables"
totalDegree p = 1 =>
  makeLineSketch(p,x,y,xMin,xMax,yMin,yMax)
-- polynomial is linear in one variable
-- y is a rational function of x
degree(p,y) = 1 =>
  makeRatFcnSketch(p,x,y,xMin,xMax,yMin,yMax,y)
-- x is a rational function of y
degree(p,x) = 1 =>
  makeRatFcnSketch(p,x,y,xMin,xMax,yMin,yMax,x)
-- the general case
makeGeneralSketch(p,x,y,xMin,xMax,yMin,yMax)

--% special cases

makeOneVarSketch(p,x,y,xMin,xMax,yMin,yMax,var) ==
-- the case where 'p' is a polynomial in only one variable
-- the graph consists of horizontal or vertical lines
if var = x then
  minVal := RNtoNF xMin
  maxVal := RNtoNF xMax
else
  minVal := RNtoNF yMin
  maxVal := RNtoNF yMax
lf : List Point DoubleFloat := nil()
rt : List Point DoubleFloat := nil()
bt : List Point DoubleFloat := nil()
tp : List Point DoubleFloat := nil()
htans : List Point DoubleFloat := nil()
vtans : List Point DoubleFloat := nil()
bran : List List Point DoubleFloat := nil()
roots := kinte(p,EPSILON)
sketchRoots : List DoubleFloat := nil()
for root in roots repeat
  if (minVal <= root) and (root <= maxVal) then
    sketchRoots := cons(NFtoSF root,sketchRoots)
null sketchRoots =>
  [p,x,y,xMin,xMax,yMin,yMax,[lf,rt,bt,tp],htans,vtans,bran]
if var = x then
  yMinSF := RNtoSF yMin; yMaxSF := RNtoSF yMax
  for rootSF in sketchRoots repeat
    tp := cons(pt1 := makePt(rootSF,yMaxSF),tp)
    bt := cons(pt2 := makePt(rootSF,yMinSF),bt)
    branch : List Point DoubleFloat := [pt1,pt2]

```

```

        bran := cons(branch, bran)
    else
        xMinSF := RNtoSF xMin; xMaxSF := RNtoSF xMax
        for rootSF in sketchRoots repeat
            rt := cons(pt1 := makePt(xMaxSF, rootSF), rt)
            lf := cons(pt2 := makePt(xMinSF, rootSF), lf)
            branch : List Point DoubleFloat := [pt1, pt2]
            bran := cons(branch, bran)
        [p, x, y, xMin, xMax, yMin, yMax, [lf, rt, bt, tp], htans, vtans, bran]

makeLineSketch(p, x, y, xMin, xMax, yMin, yMax) ==
-- the case where  $p(x, y) = a x + b y + c$  with  $a \neq 0, b \neq 0$ 
-- this is a line which is neither vertical nor horizontal
xMinSF := RNtoSF xMin; xMaxSF := RNtoSF xMax
yMinSF := RNtoSF yMin; yMaxSF := RNtoSF yMax
-- determine the coefficients a, b, and c
a := ground(coefficient(p, x, 1)) :: DoubleFloat
b := ground(coefficient(p, y, 1)) :: DoubleFloat
c := ground(coefficient(coefficient(p, x, 0), y, 0)) :: DoubleFloat
lf : List Point DoubleFloat := nil()
rt : List Point DoubleFloat := nil()
bt : List Point DoubleFloat := nil()
tp : List Point DoubleFloat := nil()
htans : List Point DoubleFloat := nil()
vtans : List Point DoubleFloat := nil()
branch : List Point DoubleFloat := nil()
bran : List List Point DoubleFloat := nil()
-- compute x coordinate of point on line with y = yMin
xBottom := (- b*yMinSF - c)/a
-- compute x coordinate of point on line with y = yMax
xTop := (- b*yMaxSF - c)/a
-- compute y coordinate of point on line with x = xMin
yLeft := (- a*xMinSF - c)/b
-- compute y coordinate of point on line with x = xMax
yRight := (- a*xMaxSF - c)/b
-- determine which of the above 4 points are in the region
-- to be plotted and list them as a branch
if (xMinSF < xBottom) and (xBottom < xMaxSF) then
    bt := cons(pt := makePt(xBottom, yMinSF), bt)
    branch := cons(pt, branch)
if (xMinSF < xTop) and (xTop < xMaxSF) then
    tp := cons(pt := makePt(xTop, yMaxSF), tp)
    branch := cons(pt, branch)
if (yMinSF <= yLeft) and (yLeft <= yMaxSF) then
    lf := cons(pt := makePt(xMinSF, yLeft), lf)
    branch := cons(pt, branch)
if (yMinSF <= yRight) and (yRight <= yMaxSF) then
    rt := cons(pt := makePt(xMaxSF, yRight), rt)
    branch := cons(pt, branch)
bran := cons(branch, bran)

```

```

[p,x,y,xMin,xMax,yMin,yMax,[lf,rt,bt,tp],htans,vtans,bran]

singValBetween?(xCurrent,xNext,xSingList) ==
  for xVal in xSingList repeat
    (xCurrent < xVal) and (xVal < xNext) => return true
  false

segmentInfo(f,lo,hi,botList,topList,singList,minSF,maxSF) ==
  repeat
    -- 'current' is the smallest element of 'topList' and 'botList'
    -- 'currentFrom' records the list from which it was taken
    if null topList then
      if null botList then
        return [segment(lo,hi),hi,nil(),nil()]
      else
        current := first botList
        botList := rest botList
        currentFrom := BOTTOM
    else
      if null botList then
        current := first topList
        topList := rest topList
        currentFrom := TOP
      else
        bot := first botList
        top := first topList
        if bot < top then
          current := bot
          botList := rest botList
          currentFrom := BOTTOM
        else
          current := top
          topList := rest topList
          currentFrom := TOP
    -- 'nxt' is the next smallest element of 'topList'
    -- and 'botList'
    -- 'nextFrom' records the list from which it was taken
    if null topList then
      if null botList then
        return [segment(lo,hi),hi,nil(),nil()]
      else
        nxt := first botList
        botList := rest botList
        nextFrom := BOTTOM
    else
      if null botList then
        nxt := first topList
        topList := rest topList
        nextFrom := TOP
      else

```

```

    bot := first botList
    top := first topList
    if bot < top then
        nxt := bot
        botList := rest botList
        nextFrom := BOTTOM
    else
        nxt := top
        topList := rest topList
        nextFrom := TOP
    if currentFrom = nextFrom then
        if singValBetween?(current,nxt,singList) then
            return [segment(lo,current),nxt,botList,topList]
        else
            val := f((nxt - current)/2::DoubleFloat)
            if (val <= minSF) or (val >= maxSF) then
                return [segment(lo,current),nxt,botList,topList]
            else
                if singValBetween?(current,nxt,singList) then
                    return [segment(lo,current),nxt,botList,topList]

makeRatFcnSketch(p,x,y,xMin,xMax,yMin,yMax,depVar) ==
-- the case where p(x,y) is linear in x or y
-- Thus, one variable is a rational function of the other.
-- Therefore, we may use the 2-dimensional function plotting
-- package. The only problem is determining the intervals on
-- on which the function is to be plotted.
--!! corners: e.g. upper left corner is on graph with y' > 0
factoredP := p :: (Factored Polynomial Integer)
numberOfFactors(factoredP) > 1 =>
    error "reducible polynomial" --!! sketch each factor
dpdx := differentiate(p,x)
dpdy := differentiate(p,y)
pRN := coerceCoefsToRNs p
xMinSF := RNtoSF xMin; xMaxSF := RNtoSF xMax
yMinSF := RNtoSF yMin; yMaxSF := RNtoSF yMax
xMinNF := RNtoNF xMin; xMaxNF := RNtoNF xMax
yMinNF := RNtoNF yMin; yMaxNF := RNtoNF yMax
-- 'p' is of degree 1 in the variable 'depVar'.
-- Thus, 'depVar' is a rational function of the other variable.
num := -coefficient(p,depVar,0)
den := coefficient(p,depVar,1)
numUPolySF := SFPolyToUPoly(coerceCoefsToSFs(num))
denUPolySF := SFPolyToUPoly(coerceCoefsToSFs(den))
-- this is the rational function
f : DoubleFloat -> DoubleFloat := s +-> elt(numUPolySF,s)/elt(denUPolySF,s)
-- values of the dependent and independent variables
if depVar = x then
    indVarMin := yMin; indVarMax := yMax
    indVarMinNF := yMinNF; indVarMaxNF := yMaxNF

```

```

indVarMinSF := yMinSF; indVarMaxSF := yMaxSF
depVarMin   := xMin;  depVarMax   := xMax
depVarMinSF := xMinSF; depVarMaxSF := xMaxSF
else
  indVarMin   := xMin;  indVarMax   := xMax
  indVarMinNF := xMinNF; indVarMaxNF := xMaxNF
  indVarMinSF := xMinSF; indVarMaxSF := xMaxSF
  depVarMin   := yMin;  depVarMax   := yMax
  depVarMinSF := yMinSF; depVarMaxSF := yMaxSF
-- Create lists of critical points.
htanPts := rsolve([p,dpx],[x,y],EPSILON)
vtanPts := rsolve([p,dpdy],[x,y],EPSILON)
htans := listPtsInRect(htanPts,xMinNF,xMaxNF,yMinNF,yMaxNF)
vtans := listPtsInRect(vtanPts,xMinNF,xMaxNF,yMinNF,yMaxNF)
-- Create lists which will contain boundary points.
lf : List Point DoubleFloat := nil()
rt : List Point DoubleFloat := nil()
bt : List Point DoubleFloat := nil()
tp : List Point DoubleFloat := nil()
-- Determine values of the independent variable at the which
-- the rational function has a pole as well as the values of
-- the independent variable for which there is a point on the
-- upper or lower boundary.
singList : List DoubleFloat :=
  roots : List Float := kinte(den,EPSILON)
  outList : List DoubleFloat := nil()
  for root in roots repeat
    if (indVarMinNF < root) and (root < indVarMaxNF) then
      outList := cons(NFtoSF root,outList)
  sort((x,y) +-> x < y, outList)
topList : List DoubleFloat :=
  roots : List Float := kinte(eval(pRN,depVar,depVarMax),EPSILON)
  outList : List DoubleFloat := nil()
  for root in roots repeat
    if (indVarMinNF < root) and (root < indVarMaxNF) then
      outList := cons(NFtoSF root,outList)
  sort((x,y) +-> x < y, outList)
botList : List DoubleFloat :=
  roots : List Float := kinte(eval(pRN,depVar,depVarMin),EPSILON)
  outList : List DoubleFloat := nil()
  for root in roots repeat
    if (indVarMinNF < root) and (root < indVarMaxNF) then
      outList := cons(NFtoSF root,outList)
  sort((x,y) +-> x < y, outList)
-- We wish to determine if the graph has points on the 'left'
-- and 'right' boundaries, so we compute the value of the
-- rational function at the lefthand and righthand values of
-- the dependent variable. If the function has a singularity
-- on the left or right boundary, then 'leftVal' or 'rightVal'
-- is given a dummy value which will convince the program that

```

```

-- there is no point on the left or right boundary.
denUPolyRN := RNPolyToUPoly(coerceCoefsToRNs(den))
if elt(denUPolyRN,indVarMin) = 0$(Fraction Integer) then
  leftVal := depVarMinSF - (abs(depVarMinSF) + 1$DoubleFloat)
else
  leftVal := f(indVarMinSF)
if elt(denUPolyRN,indVarMax) = 0$(Fraction Integer) then
  rightVal := depVarMinSF - (abs(depVarMinSF) + 1$DoubleFloat)
else
  rightVal := f(indVarMaxSF)
-- Now put boundary points on the appropriate lists.
if depVar = x then
  if (xMinSF < leftVal) and (leftVal < xMaxSF) then
    bt := cons(makePt(leftVal,yMinSF),bt)
  if (xMinSF < rightVal) and (rightVal < xMaxSF) then
    tp := cons(makePt(rightVal,yMaxSF),tp)
  for val in botList repeat
    lf := cons(makePt(xMinSF,val),lf)
  for val in topList repeat
    rt := cons(makePt(xMaxSF,val),rt)
else
  if (yMinSF < leftVal) and (leftVal < yMaxSF) then
    lf := cons(makePt(xMinSF,leftVal),lf)
  if (yMinSF < rightVal) and (rightVal < yMaxSF) then
    rt := cons(makePt(xMaxSF,rightVal),rt)
  for val in botList repeat
    bt := cons(makePt(val,yMinSF),bt)
  for val in topList repeat
    tp := cons(makePt(val,yMaxSF),tp)
bran : List List Point DoubleFloat := nil()
-- Determine segments on which the rational function is to
-- be plotted.
if (depVarMinSF < leftVal) and (leftVal < depVarMaxSF) then
  lo := indVarMinSF
else
  if null topList then
    if null botList then
      return [p,x,y,xMin,xMax,yMin,yMax,[lf,rt,bt,tp],_
              htans,vtans,bran]
    else
      lo := first botList
      botList := rest botList
  else
    if null botList then
      lo := first topList
      topList := rest topList
    else
      bot := first botList
      top := first topList
      if bot < top then

```



```

        lo := bot
        botList := rest botList
    else
        lo := top
        topList := rest topList
hi := 0$DoubleFloat -- @#%~&* compiler
if (depVarMinSF < rightVal) and (rightVal < depVarMaxSF) then
    hi := indVarMaxSF
else
    if null topList then
        if null botList then
            error "makeRatFcnSketch: plot domain"
        else
            hi := last botList
            botList := remove(hi,botList)
    else
        if null botList then
            hi := last topList
            topList := remove(hi,topList)
        else
            bot := last botList
            top := last topList
            if bot > top then
                hi := bot
                botList := remove(hi,botList)
            else
                hi := top
                topList := remove(hi,topList)
if (depVar = x) then
    (minSF := xMinSF; maxSF := xMaxSF)
else
    (minSF := yMinSF; maxSF := yMaxSF)
segList : List Segment DoubleFloat := nil()
repeat
    segInfo := segmentInfo(f,lo,hi,botList,topList,singList,_,
                           minSF,maxSF)
    segList := cons(segInfo.seg,segList)
    lo := segInfo.left
    botList := segInfo.lowerVals
    topList := segInfo.upperVals
    if lo = hi then break
for segment in segList repeat
    RFPlot : Plot := plot(f,segment)
    curve := first(listBranches(RFPlot))
    if depVar = y then
        bran := cons(curve,bran)
    else
        bran := cons(map(swapCoords,curve),bran)
[p,x,y,xMin,xMax,yMin,yMax,[lf,rt,bt,tp],htans,vtans,bran]

```

--% the general case

```

makeGeneralSketch(pol,x,y,xMin,xMax,yMin,yMax) ==
  --!! corners of region should not be on curve
  --!! enlarge region if necessary
  factoredPol := pol :: (Factored Polynomial Integer)
  numberOfFactors(factoredPol) > 1 =>
    error "reducible polynomial" --!! sketch each factor
  p := nthFactor(factoredPol,1)
  dpdx := differentiate(p,x); dpdy := differentiate(p,y)
  xMinNF := RNtoNF xMin; xMaxNF := RNtoNF xMax
  yMinNF := RNtoNF yMin; yMaxNF := RNtoNF yMax
  -- compute singular points; error if singularities in region
  singPts := rsolve([p,dpdx,dpdy],[x,y],EPSILON)
-- ptsSuchThat?(singPts,inRect?(#1,xMinNF,xMaxNF,yMinNF,yMaxNF)) =>
  foo : List Float -> Boolean := s +-> inRect?(s,xMinNF,xMaxNF,yMinNF,yMaxNF)
  ptsSuchThat?(singPts,foo) =>
    error "singular pts in region of sketch"
  -- compute critical points
  htanPts := rsolve([p,dpdx],[x,y],EPSILON)
  vtanPts := rsolve([p,dpdy],[x,y],EPSILON)
  critPts := append(htanPts,vtanPts)
  -- if there are critical points on the boundary, then enlarge
  -- the region, but be sure that the new region does not contain
  -- any singular points
  hInc : Fraction Integer := (1/20) * (xMax - xMin)
  vInc : Fraction Integer := (1/20) * (yMax - yMin)
-- if ptsSuchThat?(critPts,onVertSeg?(#1,yMinNF,yMaxNF,xMinNF)) then
  foo : List Float -> Boolean := s +-> onVertSeg?(s,yMinNF,yMaxNF,xMinNF)
  if ptsSuchThat?(critPts,foo) then
    xMin := newX(critPts,singPts,yMinNF,yMaxNF,xMinNF,xMin,-hInc)
    xMinNF := RNtoNF xMin
-- if ptsSuchThat?(critPts,onVertSeg?(#1,yMinNF,yMaxNF,xMaxNF)) then
  foo : List Float -> Boolean := s +-> onVertSeg?(s,yMinNF,yMaxNF,xMaxNF)
  if ptsSuchThat?(critPts,foo) then
    xMax := newX(critPts,singPts,yMinNF,yMaxNF,xMaxNF,xMax,hInc)
    xMaxNF := RNtoNF xMax
-- if ptsSuchThat?(critPts,onHorzSeg?(#1,xMinNF,xMaxNF,yMinNF)) then
  foo : List Float -> Boolean := s +-> onHorzSeg?(s,xMinNF,xMaxNF,yMinNF)
  if ptsSuchThat?(critPts,foo) then
    yMin := newY(critPts,singPts,xMinNF,xMaxNF,yMinNF,yMin,-vInc)
    yMinNF := RNtoNF yMin
-- if ptsSuchThat?(critPts,onHorzSeg?(#1,xMinNF,xMaxNF,yMaxNF)) then
  foo : List Float -> Boolean := s +-> onHorzSeg?(s,xMinNF,xMaxNF,yMaxNF)
  if ptsSuchThat?(critPts,foo) then
    yMax := newY(critPts,singPts,xMinNF,xMaxNF,yMaxNF,yMax,vInc)
    yMaxNF := RNtoNF yMax
  htans := listPtsInRect(htanPts,xMinNF,xMaxNF,yMinNF,yMaxNF)
  vtans := listPtsInRect(vtanPts,xMinNF,xMaxNF,yMinNF,yMaxNF)
  crits := append(htans,vtans)

```

```

-- conversions to DoubleFloats
xMinSF := RNtoSF xMin; xMaxSF := RNtoSF xMax
yMinSF := RNtoSF yMin; yMaxSF := RNtoSF yMax
corners := makeCorners(xMinSF,xMaxSF,yMinSF,yMaxSF)
pSF := coerceCoefsToSFs p
dpdxSF := coerceCoefsToSFs dpdx
dpdySF := coerceCoefsToSFs dpdy
delta := min((xMaxSF - xMinSF)/25,(yMaxSF - yMinSF)/25)
err := min(delta/100,PLOTERR/100)
bound : PositiveInteger := 10
-- compute points on the boundary
pRN := coerceCoefsToRNs(p)
lf : List Point DoubleFloat :=
  listPtsOnVertBdry(pRN,x,xMin,yMinNF,yMaxNF)
rt : List Point DoubleFloat :=
  listPtsOnVertBdry(pRN,x,xMax,yMinNF,yMaxNF)
bt : List Point DoubleFloat :=
  listPtsOnHorizBdry(pRN,y,yMin,xMinNF,xMaxNF)
tp : List Point DoubleFloat :=
  listPtsOnHorizBdry(pRN,y,yMax,xMinNF,xMaxNF)
bdPts : BoundaryPts := [lf,rt,bt,tp]
bran := traceBranches(pSF,dpdxSF,dpdySF,x,y,corners,delta,err,_
  bound,crits,bdPts)
[p,x,y,xMin,xMax,yMin,yMax,bdPts,htans,vtans,bran]

refine(plot,stepFraction) ==
  p := plot.poly; x := plot.xVar; y := plot.yVar
  dpdx := differentiate(p,x); dpdy := differentiate(p,y)
  pSF := coerceCoefsToSFs p
  dpdxSF := coerceCoefsToSFs dpdx
  dpdySF := coerceCoefsToSFs dpdy
  xMin := plot.minXVal; xMax := plot.maxXVal
  yMin := plot.minYVal; yMax := plot.maxYVal
  xMinSF := RNtoSF xMin; xMaxSF := RNtoSF xMax
  yMinSF := RNtoSF yMin; yMaxSF := RNtoSF yMax
  corners := makeCorners(xMinSF,xMaxSF,yMinSF,yMaxSF)
  pSF := coerceCoefsToSFs p
  dpdxSF := coerceCoefsToSFs dpdx
  dpdySF := coerceCoefsToSFs dpdy
  delta :=
    stepFraction * min((xMaxSF - xMinSF)/25,(yMaxSF - yMinSF)/25)
  err := min(delta/100,PLOTERR/100)
  bound : PositiveInteger := 10
  crits := append(plot.hTanPts,plot.vTanPts)
  bdPts := plot.bdryPts
  bran := traceBranches(pSF,dpdxSF,dpdySF,x,y,corners,delta,err,_
    bound,crits,bdPts)
  htans := plot.hTanPts; vtans := plot.vTanPts
  [p,x,y,xMin,xMax,yMin,yMax,bdPts,htans,vtans,bran]

```

```

traceBranches(pSF,dpdxSF,dpdySF,x,y,cornerRadius,delta,err,bound,_
               crits,bdPts) ==
  -- for boundary points, trace curve from boundary to boundary
  -- add the branch to the list of branches
  -- update list of boundary points by deleting first and last
  -- points on this branch
  -- update list of critical points by deleting any critical
  -- points which were plotted
  lf := bdPts.left; rt := bdPts.right
  tp := bdPts.top ; bt := bdPts.bottom
  bdry := append(append(lf,rt),append(bt,tp))
  bran : List List Point DoubleFloat := nil()
  while not null bdry repeat
    pt := first bdry
    p0 := dummyFirstPt(pt,dpdxSF,dpdySF,x,y,lf,rt,bt,tp)
    segInfo := listPtsOnSegment(pSF,dpdxSF,dpdySF,x,y,p0,pt,_
                               corners,delta,err,bound,crits,bdry)
    bran := cons(first segInfo,bran)
    crits := second segInfo
    bdry := third segInfo
  -- trace loops beginning and ending with critical points
  -- add the branch to the list of branches
  -- update list of critical points by deleting any critical
  -- points which were plotted
  while not null crits repeat
    pt := first crits
    segInfo := listPtsOnLoop(pSF,dpdxSF,dpdySF,x,y,pt,_
                            corners,delta,err,bound,crits,bdry)
    bran := cons(first segInfo,bran)
    crits := second segInfo
  bran

dummyFirstPt(p1,dpdxSF,dpdySF,x,y,lf,rt,bt,tp) ==
  -- The function 'computeNextPt' requires 2 points, p0 and p1.
  -- When computing the second point on a branch which starts
  -- on the boundary, we use the boundary point as p1 and the
  -- 'dummy' point returned by this function as p0.
  x1 := xCoord p1; y1 := yCoord p1
  zero := 0$DoubleFloat; one := 1$DoubleFloat
  px := ground(eval(dpdxSF,[x,y],[x1,y1]))
  py := ground(eval(dpdySF,[x,y],[x1,y1]))
  if px * py < zero then -- positive slope at p1
    member?(p1,lf) or member?(p1,bt) =>
      makePt(x1 - one,y1 - one)
      makePt(x1 + one,y1 + one)
  else
    member?(p1,lf) or member?(p1,tp) =>
      makePt(x1 - one,y1 + one)
      makePt(x1 + one,y1 - one)

```

```

listPtsOnSegment(pSF,dpdxSF,dpdySF,x,y,p0,p1,cornerRadius,_
                  delta,err,bound,crits,bdry) ==
-- p1 is a boundary point; p0 is a 'dummy' point
bdry := remove(p1,bdry)
pointList : List Point DoubleFloat := [p1]
ptInfo := computeNextPt(pSF,dpdxSF,dpdySF,x,y,p0,p1,cornerRadius,_
                        delta,err,bound,crits,bdry)

p2 := ptInfo.newPt
ptInfo.type = BDRY =>
  bdry := remove(p2,bdry)
  pointList := cons(p2,pointList)
  [pointList,crits,bdry]
if ptInfo.type = CRIT then crits := remove(p2,crits)
pointList := cons(p2,pointList)
repeat
  pt0 := second pointList; pt1 := first pointList
  ptInfo := computeNextPt(pSF,dpdxSF,dpdySF,x,y,pt0,pt1,cornerRadius,_
                          delta,err,bound,crits,bdry)

  p2 := ptInfo.newPt
  ptInfo.type = BDRY =>
    bdry := remove(p2,bdry)
    pointList := cons(p2,pointList)
    return [pointList,crits,bdry]
  if ptInfo.type = CRIT then crits := remove(p2,crits)
  pointList := cons(p2,pointList)
--!! delete next line (compiler bug)
[pointList,crits,bdry]

listPtsOnLoop(pSF,dpdxSF,dpdySF,x,y,p1,cornerRadius,_
              delta,err,bound,crits,bdry) ==
x1 := xCoord p1; y1 := yCoord p1
px := ground(eval(dpdxSF,[x,y],[x1,y1]))
py := ground(eval(dpdySF,[x,y],[x1,y1]))
p0 := makePt(x1 - 1$DoubleFloat,y1 - 1$DoubleFloat)
pointList : List Point DoubleFloat := [p1]
ptInfo := computeNextPt(pSF,dpdxSF,dpdySF,x,y,p0,p1,cornerRadius,_
                        delta,err,bound,crits,bdry)

p2 := ptInfo.newPt
ptInfo.type = BDRY =>
  error "boundary reached while on loop"
if ptInfo.type = CRIT then
  p1 = p2 =>
    error "first and second points on loop are identical"
  crits := remove(p2,crits)
pointList := cons(p2,pointList)
repeat
  pt0 := second pointList; pt1 := first pointList
  ptInfo := computeNextPt(pSF,dpdxSF,dpdySF,x,y,pt0,pt1,cornerRadius,_

```

```

                                delta,err,bound,crits,bdry)

p2 := ptInfo.newPt
ptInfo.type = BDRY =>
  error "boundary reached while on loop"
if ptInfo.type = CRIT then
  crits := remove(p2,crits)
  p1 = p2 =>
    pointList := cons(p2,pointList)
    return [pointList,crits,bdry]
  pointList := cons(p2,pointList)
--!! delete next line (compiler bug)
[pointList,crits,bdry]

computeNextPt(pSF,dpdxSF,dpdySF,x,y,p0,p1,cornerRadius,
                                delta,err,bound,crits,bdry) ==
-- p0=(x0,y0) and p1=(x1,y1) are the last two points on the curve.
-- The function computes the next point on the curve.
-- The function determines if the next point is a critical point
-- or a boundary point.
-- The function returns a record of the form
-- Record(newPt:Point DoubleFloat,type:String).
-- If the new point is a boundary point, then 'type' is
-- "boundary point" and 'newPt' is a boundary point to be
-- deleted from the list of boundary points yet to be plotted.
-- Similarly, if the new point is a critical point, then 'type' is
-- "critical point" and 'newPt' is a critical point to be
-- deleted from the list of critical points yet to be plotted.
-- If the new point is neither a critical point nor a boundary
-- point, then 'type' is "nothing in particular".
xMinSF := getXMin corners; xMaxSF := getXMax corners
yMinSF := getYMin corners; yMaxSF := getYMax corners
x0 := xCoord p0; y0 := yCoord p0
x1 := xCoord p1; y1 := yCoord p1
px := ground(eval(dpdxSF,[x,y],[x1,y1]))
py := ground(eval(dpdySF,[x,y],[x1,y1]))
-- let m be the slope of the tangent line at p1
-- if |m| < 1, we will increment the x-coordinate by delta
-- (indicated by 'incVar = x'), find an approximate
-- y-coordinate using the tangent line, then find the actual
-- y-coordinate using a Newton iteration
if abs(py) > abs(px) then
  incVar0 := incVar := x
  deltaX := (if x1 > x0 then delta else -delta)
  x2Approx := x1 + deltaX
  y2Approx := y1 + (-px/py)*deltaX
-- if |m| >= 1, we interchange the roles of the x- and y-
-- coordinates
else
  incVar0 := incVar := y
  deltaY := (if y1 > y0 then delta else -delta)

```

```

    x2Approx := x1 + (-py/px)*deltaY
    y2Approx := y1 + deltaY
    lookingFor := NADA
    -- See if (x2Approx,y2Approx) is out of bounds.
    -- If so, find where the line segment connecting (x1,y1) and
    -- (x2Approx,y2Approx) intersects the boundary and use this
    -- point as (x2Approx,y2Approx).
    -- If the resulting point is on the left or right boundary,
    -- we will now consider x as the 'incremented variable' and we
    -- will compute the y-coordinate using a Newton iteration.
    -- Similarly, if the point is on the top or bottom boundary,
    -- we will consider y as the 'incremented variable' and we
    -- will compute the x-coordinate using a Newton iteration.
    if x2Approx >= xMaxSF then
        incVar := x
        lookingFor := BDRY
        x2Approx := xMaxSF
        y2Approx := y1 + (-px/py)*(x2Approx - x1)
    else
        if x2Approx <= xMinSF then
            incVar := x
            lookingFor := BDRY
            x2Approx := xMinSF
            y2Approx := y1 + (-px/py)*(x2Approx - x1)
        if y2Approx >= yMaxSF then
            incVar := y
            lookingFor := BDRY
            y2Approx := yMaxSF
            x2Approx := x1 + (-py/px)*(y2Approx - y1)
        else
            if y2Approx <= yMinSF then
                incVar := y
                lookingFor := BDRY
                y2Approx := yMinSF
                x2Approx := x1 + (-py/px)*(y2Approx - y1)
            -- set xLo = min(x1,x2Approx), xHi = max(x1,x2Approx)
            -- set yLo = min(y1,y2Approx), yHi = max(y1,y2Approx)
            if x1 < x2Approx then
                xLo := x1
                xHi := x2Approx
            else
                xLo := x2Approx
                xHi := x1
            if y1 < y2Approx then
                yLo := y1
                yHi := y2Approx
            else
                yLo := y2Approx
                yHi := y1
            -- check for critical points (x*,y*) with x* between

```

```

-- x1 and x2Approx or y* between y1 and y2Approx
-- store values of x2Approx and y2Approx
x2Approxx := x2Approx
y2Approxx := y2Approx
-- xPointList will contain all critical points (x*,y*)
-- with x* between x1 and x2Approx
xPointList : List Point DoubleFloat := nil()
-- yPointList will contain all critical points (x*,y*)
-- with y* between y1 and y2Approx
yPointList : List Point DoubleFloat := nil()
for pt in crits repeat
  xx := xCoord pt; yy := yCoord pt
  -- if x1 = x2Approx, then p1 is a point with horizontal
  -- tangent line
  -- in this case, we don't want critical points with
  -- x-coordinate x1
  if xx = x2Approx and not (xx = x1) then
    if min(abs(yy-yLo),abs(yy-yHi)) < delta then
      xPointList := cons(pt,xPointList)
  if ((xLo < xx) and (xx < xHi)) then
    if min(abs(yy-yLo),abs(yy-yHi)) < delta then
      xPointList := cons(pt,nil())
      x2Approx := xx
      if xx < x1 then xLo := xx else xHi := xx
  -- if y1 = y2Approx, then p1 is a point with vertical
  -- tangent line
  -- in this case, we don't want critical points with
  -- y-coordinate y1
  if yy = y2Approx and not (yy = y1) then
    yPointList := cons(pt,yPointList)
  if ((yLo < yy) and (yy < yHi)) then
    if min(abs(xx-xLo),abs(xx-xHi)) < delta then
      yPointList := cons(pt,nil())
      y2Approx := yy
      if yy < y1 then yLo := yy else yHi := yy
  -- points in both xPointList and yPointList
  if (not null xPointList) and (not null yPointList) then
    xPointList = yPointList =>
      -- this implies that the lists have only one point
      incVar := incVar0
      if incVar = x then
        y2Approx := y1 + (-px/py)*(x2Approx - x1)
      else
        x2Approx := x1 + (-py/px)*(y2Approx - y1)
      lookingFor := CRIT      -- proceed
  incVar0 = x =>
  -- first try Newton iteration with 'y' as incremented variable
  x2Temp := x1 + (-py/px)*(y2Approx - y1)
  f := SFPolyToUPoly(eval(pSF,y,y2Approx))
  x2New := newtonApprox(f,x2Temp,err,bound)

```



```

x2New case "failed" =>
  y2Approx := y1 + (-px/py)*(x2Approx - x1)
  incVar := x
  lookingFor := CRIT      -- proceed
y2Temp := y1 + (-px/py)*(x2Approx - x1)
f := SFPolyToUPoly(eval(pSF,x,x2Approx))
y2New := newtonApprox(f,y2Temp,err,bound)
y2New case "failed" =>
  return computeNextPt(pSF,dpdxSF,dpdySF,x,y,p0,p1,cornerRadius,
    abs((x2Approx-x1)/2),err,bound,crits,bdry)
pt1 := makePt(x2Approx,y2New :: DoubleFloat)
pt2 := makePt(x2New :: DoubleFloat,y2Approx)
critPt1 := findPtOnList(pt1,crits)
critPt2 := findPtOnList(pt2,crits)
(critPt1 case "failed") and (critPt2 case "failed") =>
  abs(x2Approx - x1) > abs(x2Temp - x1) =>
    return [pt1,NADA]
  return [pt2,NADA]
(critPt1 case "failed") =>
  return [critPt2:: (Point DoubleFloat),CRIT]
(critPt2 case "failed") =>
  return [critPt1:: (Point DoubleFloat),CRIT]
abs(x2Approx - x1) > abs(x2Temp - x1) =>
  return [critPt2:: (Point DoubleFloat),CRIT]
return [critPt1:: (Point DoubleFloat),CRIT]
y2Temp := y1 + (-px/py)*(x2Approx - x1)
f := SFPolyToUPoly(eval(pSF,x,x2Approx))
y2New := newtonApprox(f,y2Temp,err,bound)
y2New case "failed" =>
  x2Approx := x1 + (-py/px)*(y2Approx - y1)
  incVar := y
  lookingFor := CRIT      -- proceed
x2Temp := x1 + (-py/px)*(y2Approx - y1)
f := SFPolyToUPoly(eval(pSF,y,y2Approx))
x2New := newtonApprox(f,x2Temp,err,bound)
x2New case "failed" =>
  return computeNextPt(pSF,dpdxSF,dpdySF,x,y,p0,p1,cornerRadius,
    abs((y2Approx-y1)/2),err,bound,crits,bdry)
pt1 := makePt(x2Approx,y2New :: DoubleFloat)
pt2 := makePt(x2New :: DoubleFloat,y2Approx)
critPt1 := findPtOnList(pt1,crits)
critPt2 := findPtOnList(pt2,crits)
(critPt1 case "failed") and (critPt2 case "failed") =>
  abs(y2Approx - y1) > abs(y2Temp - y1) =>
    return [pt2,NADA]
  return [pt1,NADA]
(critPt1 case "failed") =>
  return [critPt2:: (Point DoubleFloat),CRIT]
(critPt2 case "failed") =>
  return [critPt1:: (Point DoubleFloat),CRIT]

```

```

    abs(y2Approx - y1) > abs(y2Temp - y1) =>
        return [critPt1::(Point DoubleFloat),CRIT]
    return [critPt2::(Point DoubleFloat),CRIT]
if (not null xPointList) and (null yPointList) then
    y2Approx := y1 + (-px/py)*(x2Approx - x1)
    incVar0 = x =>
        incVar := x
        lookingFor := CRIT          -- proceed
    f := SFPolyToUPoly(eval(pSF,x,x2Approx))
    y2New := newtonApprox(f,y2Approx,err,bound)
    y2New case "failed" =>
        x2Approx := x2Approxx
        y2Approx := y2Approxx      -- proceed
    pt := makePt(x2Approx,y2New::DoubleFloat)
    critPt := findPtOnList(pt,crits)
    critPt case "failed" =>
        return [pt,NADA]
    return [critPt :: (Point DoubleFloat),CRIT]
if (null xPointList) and (not null yPointList) then
    x2Approx := x1 + (-py/px)*(y2Approx - y1)
    incVar0 = y =>
        incVar := y
        lookingFor := CRIT          -- proceed
    f := SFPolyToUPoly(eval(pSF,y,y2Approx))
    x2New := newtonApprox(f,x2Approx,err,bound)
    x2New case "failed" =>
        x2Approx := x2Approxx
        y2Approx := y2Approxx      -- proceed
    pt := makePt(x2New::DoubleFloat,y2Approx)
    critPt := findPtOnList(pt,crits)
    critPt case "failed" =>
        return [pt,NADA]
    return [critPt :: (Point DoubleFloat),CRIT]
if incVar = x then
    x2 := x2Approx
    f := SFPolyToUPoly(eval(pSF,x,x2))
    y2New := newtonApprox(f,y2Approx,err,bound)
    y2New case "failed" =>
        return computeNextPt(pSF,dpdxSF,dpdySF,x,y,p0,p1,cornerRadius,
                               abs((x2-x1)/2),err,bound,crits,bdry)
    y2 := y2New :: DoubleFloat
else
    y2 := y2Approx
    f := SFPolyToUPoly(eval(pSF,y,y2))
    x2New := newtonApprox(f,x2Approx,err,bound)
    x2New case "failed" =>
        return computeNextPt(pSF,dpdxSF,dpdySF,x,y,p0,p1,cornerRadius,
                               abs((y2-y1)/2),err,bound,crits,bdry)
    x2 := x2New :: DoubleFloat
pt := makePt(x2,y2)

```

```

--!! check that 'pt' is not out of bounds
-- check if you've gotten a critical or boundary point
lookingFor = NADA =>
  [pt,lookingFor]
lookingFor = BDRY =>
  bdryPt := findPtOnList(pt,bdry)
  bdryPt case "failed" =>
    error "couldn't find boundary point"
  [bdryPt :: (Point DoubleFloat),BDRY]
critPt := findPtOnList(pt,crits)
critPt case "failed" =>
  [pt,NADA]
[critPt :: (Point DoubleFloat),CRIT]

--% Newton iterations

newtonApprox(f,a0,err,bound) ==
-- Newton iteration to approximate a root of the polynomial 'f'
-- using an initial approximation of 'a0'
-- Newton iteration terminates when consecutive approximations
-- are within 'err' of each other
-- returns "failed" if this has not been achieved after 'bound'
-- iterations
Df := differentiate f
oldApprox := a0
newApprox := a0 - elt(f,a0)/elt(Df,a0)
i : PositiveInteger := 1
while abs(newApprox - oldApprox) > err repeat
  i = bound => return "failed"
  oldApprox := newApprox
  newApprox := oldApprox - elt(f,oldApprox)/elt(Df,oldApprox)
  i := i+1
newApprox

--% graphics output

listBranches(acplot) == acplot.branches

--% terminal output

coerce(acplot:%) ==
pp := acplot.poly :: OutputForm
xx := acplot.xVar :: OutputForm
yy := acplot.yVar :: OutputForm
xLo := acplot.minXVal :: OutputForm
xHi := acplot.maxXVal :: OutputForm
yLo := acplot.minYVal :: OutputForm
yHi := acplot.maxYVal :: OutputForm
zip := message(" = 0")
com := message(", ")

```

```

les := message(" <= ")
l : List OutputForm :=
  [pp,zip,com,xLo,les,xx,les,xHi,com,yLo,les,yy,les,yHi]
f : List OutputForm := nil()
for branch in acplot.branches repeat
  ll : List OutputForm := [p :: OutputForm for p in branch]
  f := cons(vconcat ll,f)
ff := vconcat(hconcat l,vconcat f)
vconcat(message "ACPLOT",ff)

```

— ACPLOT.dotabb —

```

"ACPLOT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ACPLOT"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ACPLOT" -> "ALIST"

```

domain PLACES Places

— Places.input —

```

)set break resume
)sys rm -f Places.output
)spool Places.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Places
--R
--R Places(K: Field) is a domain constructor
--R Abbreviation for Places is PLACES
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PLACES
--R
--R----- Operations -----
--R ?? : (Integer,%) -> Divisor(%)      ?+? : (%,%) -> Divisor(%)
--R -? : % -> Divisor(%)                ?-? : (%,%) -> Divisor(%)
--R ?=? : (%,%) -> Boolean               coerce : % -> OutputForm
--R create : Symbol -> %                 create : List(K) -> %

```

```

--R degree : % -> PositiveInteger      ?? : (%,Integer) -> K
--R foundPlaces : () -> List(%)        hash : % -> SingleInteger
--R itsALeaf! : % -> Void              latex : % -> String
--R leaf? : % -> Boolean               reduce : List(%) -> Divisor(%)
--R ?~=? : (%,% ) -> Boolean
--R ?+? : (% ,Divisor(%)) -> Divisor(%)
--R ?+? : (Divisor(%),%) -> Divisor(%)
--R ?-? : (% ,Divisor(%)) -> Divisor(%)
--R ?-? : (Divisor(%),%) -> Divisor(%)
--R localParam : % -> List(NeitherSparseOrDensePowerSeries(K))
--R setDegree! : (% ,PositiveInteger) -> Void
--R setFoundPlacesToEmpty : () -> List(%)
--R setParam! : (% ,List(NeitherSparseOrDensePowerSeries(K))) -> Void
--R
--E 1

)spool
)lisp (bye)

```

— Places.help —

=====

Places examples

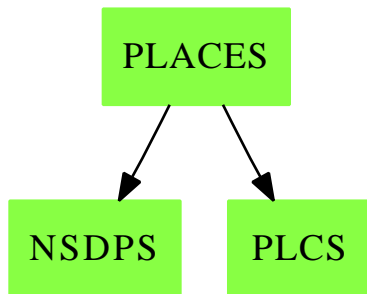
=====

The following is part of the PAFF package

See Also:

- o)show Places

Places (PLACES)

**Exports:**

-?	?*?	?+?	?-?	??
?=?	?~=?	coerce	create	degree
foundPlaces	hash	itsALeaf!	latex	leaf?
localParam	reduce	setDegree!	setFoundPlacesToEmpty	setParam!

— domain **PLACES** Places —

```

)abbrev domain PLACES Places
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
Places(K):Exports == Implementation where
  K:Field
  PCS ==> NeitherSparseOrDensePowerSeries(K)

  Exports ==> PlacesCategory(K,PCS)

  Implementation ==> Plcs(K,PCS)

```

— **PLACES.dotabb** —

```

"PLACES" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PLACES"];
"NSDPS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=NSDPS"];
"PLCS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PLCS"]
"PLACES" -> "NSDPS"
"PLACES" -> "PLCS"

```

domain PLACESPS PlacesOverPseudoAlgebraicClosureOfFiniteField

— PlacesOverPseudoAlgebraicClosureOfFiniteField.input —

```

)set break resume
)sys rm -f PlacesOverPseudoAlgebraicClosureOfFiniteField.output
)spool PlacesOverPseudoAlgebraicClosureOfFiniteField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PlacesOverPseudoAlgebraicClosureOfFiniteField
--R
--R PlacesOverPseudoAlgebraicClosureOfFiniteField(K: FiniteFieldCategory) is a domain construct
--R Abbreviation for PlacesOverPseudoAlgebraicClosureOfFiniteField is PLACESPS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PLACESPS
--R
--R----- Operations -----
--R ?? : (Integer,%) -> Divisor(%)      ?? : (%,%) -> Divisor(%)
--R -? : % -> Divisor(%)                ?-? : (%,%) -> Divisor(%)
--R ?? : (%,%) -> Boolean               coerce : % -> OutputForm
--R create : Symbol -> %                 degree : % -> PositiveInteger
--R foundPlaces : () -> List(%)          hash : % -> SingleInteger
--R itsALeaf! : % -> Void                latex : % -> String
--R leaf? : % -> Boolean                 reduce : List(%) -> Divisor(%)
--R ~=? : (%,%) -> Boolean
--R ?? : (%,Divisor(%)) -> Divisor(%)
--R ?+? : (Divisor(%),%) -> Divisor(%)
--R ?-? : (%,Divisor(%)) -> Divisor(%)
--R ?~? : (Divisor(%),%) -> Divisor(%)
--R create : List(PseudoAlgebraicClosureOfFiniteField(K)) -> %
--R ?.? : (%,Integer) -> PseudoAlgebraicClosureOfFiniteField(K)
--R localParam : % -> List(NeitherSparseOrDensePowerSeries(PseudoAlgebraicClosureOfFiniteField(K))
--R setDegree! : (%,PositiveInteger) -> Void
--R setFoundPlacesToEmpty : () -> List(%)
--R setParam! : (%,List(NeitherSparseOrDensePowerSeries(PseudoAlgebraicClosureOfFiniteField(K)))
--R
--E 1

)spool
)lisp (bye)

```

— PlacesOverPseudoAlgebraicClosureOfFiniteField.help —

```
=====
PlacesOverPseudoAlgebraicClosureOfFiniteField examples
=====
```

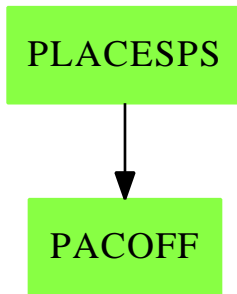
The following is part of the PAFF package

See Also:

```
o )show PlacesOverPseudoAlgebraicClosureOfFiniteField
```

—————

PlacesOverPseudoAlgebraicClosureOfFiniteField (PLACESPS)



Exports:

-?	?*?	?+?	?-?	?..?
?=?	?~=?	coerce	create	degree
foundPlaces	hash	itsALeaf!	latex	leaf?
localParam	reduce	setDegree!	setFoundPlacesToEmpty	setParam!

— domain PLACESPS PlacesOverPseudoAlgebraicClosureOfFiniteField —

```
)abbrev domain PLACESPS PlacesOverPseudoAlgebraicClosureOfFiniteField
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
PlacesOverPseudoAlgebraicClosureOfFiniteField(K):Exports
== Implementation where
```



```

K:FiniteFieldCategory
KK ==> PseudoAlgebraicClosureOfFiniteField(K)
PCS ==> NeitherSparseOrDensePowerSeries(KK)

Exports ==> PlacesCategory(KK,PCS)

Implementation ==> Plcs(KK,PCS)

```

— PLACESPS.dotabb —

```

"PLACESPS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PLACESPS"];
"PACOFF" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PACOFF"]
"PLACESPS" -> "PACOFF"

```

domain PLCS Plcs

— Plcs.input —

```

)set break resume
)sys rm -f Plcs.output
)spool Plcs.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Plcs
--R
--R Plcs(K: Field,PCS: LocalPowerSeriesCategory(K)) is a domain constructor
--R Abbreviation for Plcs is PLCS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PLCS
--R
--R----- Operations -----
--R ?? : (Integer,%) -> Divisor(%)      ?? : (%,%) -> Divisor(%)
--R -? : % -> Divisor(%)                ?-? : (%,%) -> Divisor(%)
--R ?? : (%,%) -> Boolean                coerce : % -> OutputForm
--R create : Symbol -> %                  create : List(K) -> %
--R degree : % -> PositiveInteger         ?.? : (%,Integer) -> K
--R foundPlaces : () -> List(%)           hash : % -> SingleInteger
--R itsALeaf! : % -> Void                  latex : % -> String

```

```

--R leaf? : % -> Boolean                                localParam : % -> List(PCS)
--R reduce : List(%) -> Divisor(%)                      setParam! : (%,List(PCS)) -> Void
--R ?~=? : (%,%) -> Boolean
--R ?+? : (%,Divisor(%)) -> Divisor(%)
--R ?+? : (Divisor(%),%) -> Divisor(%)
--R ?-? : (%,Divisor(%)) -> Divisor(%)
--R ?-? : (Divisor(%),%) -> Divisor(%)
--R setDegree! : (%,PositiveInteger) -> Void
--R setFoundPlacesToEmpty : () -> List(%)
--R
--E 1

)spool
)lisp (bye)

```

—————
 — Plcs.help —

=====

Plcs examples

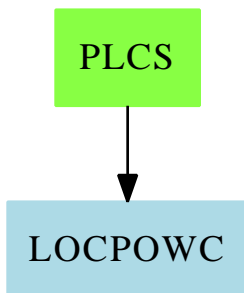
=====

The following is part of the PAFF package

See Also:
 o)show Plcs

—————

Plcs (PLCS)



Exports:

— domain PLCS Plcs —

```

)abbrev domain PLCS Plcs
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
Plcs(K:Field,PCS:LocalPowerSeriesCategory(K)):Exports == Implementation where

  nameOfPlace ==> Union( List(K), Symbol )

  rec ==> Record(theName:nameOfPlace,_
                 locPar: List PCS,_
                 deg: PositiveInteger,_
                 isALeaf:Boolean,_
                 inName:Symbol,_
                 actualSet:Symbol)

  Exports ==> PlacesCategory(K,PCS)

  Implementation ==>  add

    Rep:= rec

    setOfPlacesName:Symbol:=new(ActualSetOfPlacesName)$Symbol

    a:% + b:% == (a:: Divisor(%)) +$Divisor(%) (b::Divisor(%))

    a:% - b:% == (a:: Divisor(%)) -$Divisor(%) (b::Divisor(%))

    n:Integer * b:% == n *$Divisor(%) (b :: Divisor(%))

    reduce(lp)==
      lpd:List Divisor(%) := [p :: Divisor(%) for p in lp]
      reduce("+", lpd, 0$Divisor(%))

    d:Divisor(%) + b:% == d + (b::Divisor(%))

    a:% + d:Divisor(%) == (a::Divisor(%)) + d

    d:Divisor(%) - b:% == d - (b::Divisor(%))

    a:% - d:Divisor(%) == (a::Divisor(%)) - d

    -a:% == - ( a::Divisor(%))

    outName: nameOfPlace -> OutputForm

```

```

outName(pt)==
  pt case Symbol => pt :: OutputForm
  dd:OutputForm:= ":" :: OutputForm
  llout:List(OutputForm):=[ hconcat(dd, a::OutputForm) for a in rest pt]
  lout:= cons( (first pt)::OutputForm , llout)
  out:= hconcat lout
  bracket(out)

coerce(pt:%):OutputForm ==
  nn:OutputForm:= outName(pt.theName)
  ee:OutputForm:= degree(pt) :: OutputForm
  nn ** ee

a:% = b:% ==
  ~(a.actualSet =$Symbol b.actualSet) =>
    a:String:=
      "From Places Domain: Cannot use old places with new places."
      " You have declared two different package PAFF or PAFFFF with the "
      "same arguments. This is not allowed because in that case the two "
      "packages used the same domain to represent the set of places. "
      "Two packages having the same arguments should be used in "
      "different frame"
    error a
    a.inName =$Symbol b.inName

elt(pl,n)==
  pt:= (pl :: Rep).theName
  pt case Symbol => _
    error "From Places domain : cannot return the coordinates of a leaf"
  elt(pt,n)$List(K)

leaf?(pl)==pl.isALeaf

itsALeaf_!(pl)==
  pl.isALeaf := true()
  void()

listOfFoundPlaces:List %:=[]

foundPlaces()==listOfFoundPlaces

setFoundPlacesToEmpty()==
  tmp:=copy listOfFoundPlaces
  listOfFoundPlaces:=[]
  setOfPlacesName:Symbol:=new(ActualSetOfPlacesName)$Symbol
  tmp

findInExistOnes: % -> %
findInExistOnes(pt)==
  ll:=listOfFoundPlaces

```

```

found:Boolean:=false()
fpl:%
while ^found and ^empty?(ll) repeat
  fpl:= first ll
  -- remember: the "=" test is on done on the symbolic name
  found:= pt.theName = fpl.theName
  ll:=rest ll
^found =>
  listOfFoundPlaces:=cons(pt,listOfFoundPlaces)
  pt
fpl

create(pt>List(K)):%==
  newName:=new(SIMPLE)$Symbol
  newPt:%=[pt,[],1,false(),newName,setOfPlacesName]$rec
  findInExistOnes(newPt)

create(pt:Symbol):%==
  newPt:%=[pt,[],1,false(),pt,setOfPlacesName]$rec
  findInExistOnes(newPt)

setDegree_!(pt,d)==
  pt.deg := d
  void()

setParam_!(pt,ls)==
  pt.locPar:=ls
  void()

localParam(pt)==pt.locPar

degree(pl)==pl.deg

```

— PLCS.dotabb —

```

"PLCS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PLCS"];
"LOCP0WC" [color=lightblue,href="bookvol10.2.pdf#nameddest=LOCP0WC"];
"PLCS" -> "LOCP0WC"

```

domain PLOT Plot

— Plot.input —

```

)set break resume
)sys rm -f Plot.output
)spool Plot.output
)set message test on
)set message auto off
)clear all
--S 1 of 2
fp:=(t:DFLOAT):DFLOAT +-> sin(t)
--R
--R (1) theMap(Closure)
--R
--R                                          Type: (DoubleFloat -> DoubleFloat)
--E 1

--S 2 of 2
plot(fp,-1.0..1.0)$PLOT
--R
--R
--R (2) PLOT(x = (- 1.)..1.    y = (- 0.8414709848078965)..0.8414709848078965)
--R      [- 1.,- 0.8414709848078965]
--R      [- 0.9583333333333337,- 0.81823456433427133]
--R      [- 0.91666666666666674,- 0.79357780324894212]
--R      [- 0.87500000000000011,- 0.76754350223602708]
--R      [- 0.83333333333333348,- 0.74017685319603721]
--R      [- 0.79166666666666685,- 0.7115253607990657]
--R      [- 0.75000000000000022,- 0.68163876002333434]
--R      [- 0.70833333333333359,- 0.65056892982223602]
--R      [- 0.66666666666666696,- 0.61836980306973721]
--R      [- 0.62500000000000033,- 0.58509727294046243]
--R      [- 0.5833333333333337,- 0.55080909588697013]
--R      [- 0.54166666666666707,- 0.51556479138264011]
--R      [- 0.50000000000000044,- 0.47942553860420339]
--R      [- 0.45833333333333376,- 0.44245407023325911]
--R      [- 0.41666666666666707,- 0.40471456356112506]
--R      [- 0.37500000000000039,- 0.3662725290860479]
--R      [- 0.3333333333333337,- 0.3271946967961526]
--R      [- 0.29166666666666702,- 0.28754890033552849]
--R      [- 0.25000000000000033,- 0.24740395925452324]
--R      [- 0.20833333333333368,- 0.20682955954864138]
--R      [- 0.16666666666666702,- 0.16589613269341538]
--R      [- 0.12500000000000036,- 0.12467473338522805]
--R      [- 8.3333333333333703E-2,- 8.3236916200310623E-2]
--R      [- 4.1666666666667039E-2,- 4.1654611386019461E-2]
--R      [- 3.7470027081099033E-16,- 3.7470027081099033E-16]
--R      [4.166666666666629E-2,4.1654611386018711E-2]

```

```

--R      [8.3333333333332954E-2,8.3236916200309874E-2]
--R      [0.12499999999999961,0.1246747333852273]
--R      [0.16666666666666627,0.16589613269341463]
--R      [0.20833333333333293,0.20682955954864066]
--R      [0.24999999999999958,0.24740395925452252]
--R      [0.29166666666666624,0.28754890033552777]
--R      [0.33333333333333293,0.32719469679615187]
--R      [0.37499999999999961,0.36627252908604718]
--R      [0.4166666666666663,0.4047145635611244]
--R      [0.45833333333333298,0.44245407023325839]
--R      [0.49999999999999967,0.47942553860420273]
--R      [0.5416666666666663,0.51556479138263944]
--R      [0.58333333333333293,0.55080909588696947]
--R      [0.62499999999999956,0.58509727294046177]
--R      [0.66666666666666619,0.61836980306973666]
--R      [0.70833333333333282,0.65056892982223535]
--R      [0.74999999999999944,0.6816387600233379]
--R      [0.79166666666666607,0.71152536079906514]
--R      [0.8333333333333327,0.74017685319603665]
--R      [0.87499999999999933,0.76754350223602663]
--R      [0.91666666666666596,0.79357780324894167]
--R      [0.95833333333333259,0.81823456433427078]
--R      [1.,0.8414709848078965]
--R
--R
--E 2
)spool
)lisp (bye)

```

Type: Plot

— Plot.help —

=====

Plot examples

=====

The Plot (PLOT) domain supports plotting of functions defined over a real number system. Plot is limited to 2 dimensional plots.

The function plot: (F -> F,R) -> % plots the function f(x) on the interval a..b. So we need to define a function that maps from DoubleFloat to DoubleFloat:

```
fp:=(t:DFLOAT):DFLOAT +-> sin(t)
```

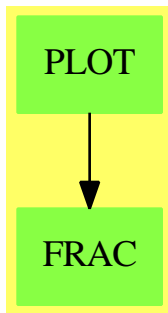
and then feed it to the plot function with a Segment DoubleFloat

```
plot(fp,-1.0..1.0)$PLOT
```

See Also:

o)show Plot

Plot (PLOT)



Exports:

adaptive?	coerce	debug	listBranches	maxPoints
minPoints	numFunEvals	parametric?	plot	plotPolar
pointPlot	refine	screenResolution	setAdaptive	setMaxPoints
setMinPoints	setScreenResolution	tRange	xRange	yRange
zoom				

— domain PLOT Plot —

```

)abbrev domain PLOT Plot
++ Author: Michael Monagan (revised by Clifton J. Williamson)
++ Date Created: Jan 1988
++ Date Last Updated: 30 Nov 1990 by Jonathan Steinbach
++ Description:
++ The Plot domain supports plotting of functions defined over a
++ real number system. A real number system is a model for the real
++ numbers and as such may be an approximation. For example
++ floating point numbers and infinite continued fractions.
++ The facilities at this point are limited to 2-dimensional plots
++ or either a single function or a parametric function.
  
```

```

Plot(): Exports == Implementation where
  
```

```

B ==> Boolean
F ==> DoubleFloat
I ==> Integer
L ==> List
N ==> NonNegativeInteger
  
```



```

OUT ==> OutputForm
P  ==> Point F
RN ==> Fraction Integer
S  ==> String
SEG ==> Segment
R  ==> Segment F
C  ==> Record(source: F -> P, ranges: L R, knots: L F, points: L P)

Exports ==> PlottablePlaneCurveCategory with

--% function plots

plot: (F -> F,R) -> %
++ plot(f,a..b) plots the function \spad{f(x)}
++ on the interval \spad{[a,b]}.
++
++X fp:=(t:DFLOAT):DFLOAT +-> sin(t)
++X plot(fp,-1.0..1.0)$PLOT

plot: (F -> F,R,R) -> %
++ plot(f,a..b,c..d) plots the function \spad{f(x)} on the interval
++ \spad{[a,b]}; y-range of \spad{[c,d]} is noted in Plot object.

--% multiple function plots

plot: (L(F -> F),R) -> %
++ plot([f1,...,fm],a..b) plots the functions \spad{y = f1(x)},...,
++ \spad{y = fm(x)} on the interval \spad{a..b}.
plot: (L(F -> F),R,R) -> %
++ plot([f1,...,fm],a..b,c..d) plots the functions \spad{y = f1(x)},...,
++ \spad{y = fm(x)} on the interval \spad{a..b}; y-range of \spad{[c,d]} is
++ noted in Plot object.

--% parametric plots

plot: (F -> F,F -> F,R) -> %
++ plot(f,g,a..b) plots the parametric curve \spad{x = f(t)},
++ \spad{y = g(t)} as t ranges over the interval \spad{[a,b]}.
plot: (F -> F,F -> F,R,R,R) -> %
++ plot(f,g,a..b,c..d,e..f) plots the parametric curve \spad{x = f(t)},
++ \spad{y = g(t)} as t ranges over the interval \spad{[a,b]}; x-range
++ of \spad{[c,d]} and y-range of \spad{[e,f]} are noted in Plot object.

--% parametric plots

pointPlot: (F -> P,R) -> %
++ pointPlot(t +-> (f(t),g(t)),a..b) plots the parametric curve
++ \spad{x = f(t)}, \spad{y = g(t)} as t ranges over the interval
++ \spad{[a,b]}.
pointPlot: (F -> P,R,R,R) -> %

```

```

++ pointPlot(t --> (f(t),g(t)),a..b,c..d,e..f) plots the parametric
++ curve \spad{x = f(t)}, \spad{y = g(t)} as t ranges over the interval
++ \spad{[a,b]}; x-range of \spad{[c,d]} and y-range of \spad{[e,f]}
++ are noted in Plot object.

--% polar plots

plotPolar: (F -> F,R) -> %
++ plotPolar(f,a..b) plots the polar curve \spad{r = f(theta)} as
++ theta ranges over the interval \spad{[a,b]}; this is the same as
++ the parametric curve \spad{x = f(t)*cos(t)}, \spad{y = f(t)*sin(t)}.

plotPolar: (F -> F) -> %
++ plotPolar(f) plots the polar curve \spad{r = f(theta)} as theta
++ ranges over the interval \spad{[0,2*pi]}; this is the same as
++ the parametric curve \spad{x = f(t)*cos(t)}, \spad{y = f(t)*sin(t)}.

plot: (% ,R) -> %          -- change the range
++ plot(x,r) is not documented
parametric?: % -> B
++ parametric? determines whether it is a parametric plot?

zoom: (% ,R) -> %
++ zoom(x,r) is not documented
zoom: (% ,R,R) -> %
++ zoom(x,r,s) is not documented
refine: (% ,R) -> %
++ refine(x,r) is not documented
refine: % -> %
++ refine(p) performs a refinement on the plot p

tRange: % -> R
++ tRange(p) returns the range of the parameter in a parametric plot p

minPoints: () -> I
++ minPoints() returns the minimum number of points in a plot
setMinPoints: I -> I
++ setMinPoints(i) sets the minimum number of points in a plot to i
maxPoints: () -> I
++ maxPoints() returns the maximum number of points in a plot
setMaxPoints: I -> I
++ setMaxPoints(i) sets the maximum number of points in a plot to i
screenResolution: () -> I
++ screenResolution() returns the screen resolution
setScreenResolution: I -> I
++ setScreenResolution(i) sets the screen resolution to i
adaptive?: () -> B
++ adaptive?() determines whether plotting be done adaptively
setAdaptive: B -> B
++ setAdaptive(true) turns adaptive plotting on

```

```

    ++ \spad{setAdaptive(false)} turns adaptive plotting off
numFunEvals: () -> I
    ++ numFunEvals() returns the number of points computed
debug: B -> B
    ++ debug(true) turns debug mode on
    ++ \spad{debug(false)} turns debug mode off

Implementation ==> add
    import PointPackage(DoubleFloat)

--% local functions

checkRange      : R -> R
    -- checks that left-hand endpoint is less than right-hand endpoint
intersect       : (R,R) -> R
    -- intersection of two intervals
union           : (R,R) -> R
    -- union of two intervals
join            : (L C,I) -> R
parametricRange: % -> R
select          : (L P,P -> F,(F,F) -> F) -> F
rangeRefine     : (C,R) -> C
adaptivePlot    : (C,R,R,R,I) -> C
basicPlot       : (F -> P,R) -> C
basicRefine     : (C,R) -> C
pt              : (F,F) -> P
Fnan?           : F -> Boolean
Pnan?           : P -> Boolean

--% representation

Rep := Record( parametric: B, _
               display: L R, _
               bounds: L R, _
               axisLabels: L S, _
               functions: L C )

--% global constants

ADAPTIVE: B := true
MINPOINTS: I := 49
MAXPOINTS: I := 1000
NUMFUN EVALS: I := 0
SCREENRES: I := 500
ANGLEBOUND: F := cos inv (4::F)
DEBUG: B := false

Fnan?(x) == x ~= x
Pnan?(x) == any?(Fnan?,x)

```

```

--% graphics output

listBranches plot ==
  outList : L L P := nil()
  for curve in plot.functions repeat
    -- curve is C
    newl:L P:=nil()
    for p in curve.points repeat
      if not Pnan? p then newl:=cons(p,newl)
      else if not empty? newl then
        outList := concat(newl:=reverse! newl,outList)
        newl:=nil()
    if not empty? newl then outList := concat(newl:=reverse! newl,outList)
--    print(outList::OutputForm)
  outList

checkRange r == (lo r > hi r => error "ranges cannot be negative"; r)
intersect(s,t) == checkRange (max(lo s,lo t) .. min(hi s,hi t))
union(s,t) == min(lo s,lo t) .. max(hi s,hi t)
join(l,i) ==
  rr := first l
  u : R :=
    i = 0 => first(rr.ranges)
    i = 1 => second(rr.ranges)
    third(rr.ranges)
  for r in rest l repeat
    i = 0 => u := union(u,first(r.ranges))
    i = 1 => u := union(u,second(r.ranges))
    u := union(u,third(r.ranges))
  u
parametricRange r == first(r.bounds)

minPoints() == MINPOINTS
setMinPoints n ==
  if n < 3 then error "three points minimum required"
  if MAXPOINTS < n then MAXPOINTS := n
  MINPOINTS := n
maxPoints() == MAXPOINTS
setMaxPoints n ==
  if n < 3 then error "three points minimum required"
  if MINPOINTS > n then MINPOINTS := n
  MAXPOINTS := n
screenResolution() == SCREENRES
setScreenResolution n ==
  if n < 2 then error "buy a new terminal"
  SCREENRES := n
adaptive?() == ADAPTIVE
setAdaptive b == ADAPTIVE := b
parametric? p == p.parametric

```

```

numFunEvals() == NUMFUNEVALS
debug b == DEBUG := b

xRange plot == second plot.bounds
yRange plot == third plot.bounds
tRange plot == first plot.bounds

select(l,f,g) ==
  m := f first l
  if Fnan? m then m := 0
  for p in rest l repeat
    n := m
    m := g(m, f p)
    if Fnan? m then m := n
  m

rangeRefine(curve,nRange) ==
  checkRange nRange; l := lo nRange; h := hi nRange
  t := curve.knots; p := curve.points; f := curve.source
  while not null t and first t < l repeat
    (t := rest t; p := rest p)
  c: L F := nil(); q: L P := nil()
  while not null t and (first t) <= h repeat
    c := concat(first t,c); q := concat(first p,q)
    t := rest t; p := rest p
  if null c then return basicPlot(f,nRange)
  if first c < h then
    c := concat(h,c)
    q := concat(f h,q)
    NUMFUNEVALS := NUMFUNEVALS + 1
  t := c := reverse_! c; p := q := reverse_! q
  s := (h-l)/(minPoints()::F-1)
  if (first t) ^= l then
    t := c := concat(l,c)
    p := q := concat(f l,p)
    NUMFUNEVALS := NUMFUNEVALS + 1
  while not null rest t repeat
    n := wholePart((second(t) - first(t))/s)
    d := (second(t) - first(t))/((n+1)::F)
    for i in 1..n repeat
      t.rest := concat(first(t) + d,rest t)
      p.rest := concat(f second t,rest p)
      NUMFUNEVALS := NUMFUNEVALS + 1
    t := rest t; p := rest p
  t := rest t
  p := rest p
  xRange := select(q,xCoord,min) .. select(q,xCoord,max)
  yRange := select(q,yCoord,min) .. select(q,yCoord,max)
  [ f, [nRange,xRange,yRange], c, q]

```

```

adaptivePlot(curve,tRange,xRange,yRange,pixelfraction) ==
  xDiff := hi xRange - lo xRange
  yDiff := hi yRange - lo yRange
  xDiff = 0 or yDiff = 0 => curve
  l := lo tRange; h := hi tRange
  (tDiff := h-l) = 0 => curve
--   if (EQL(yDiff, _$NaNvalue$Lisp)$Lisp) then yDiff := 1::F
  t := curve.knots
  #t < 3 => curve
  p := curve.points; f := curve.source
  minLength:F := 4::F/500::F
  maxLength:F := 1::F/6::F
  tLimit := tDiff/(pixelfraction*500)::F
  while not null t and first t < l repeat (t := rest t; p := rest p)
  #t < 3 => curve
  headert := t; headerp := p

-- jitter the input points
--   while not null rest rest t repeat
--     t0 := second(t); t1 := third(t)
--     jitter := (random())$I)::F
--     jitter := sin (jitter)
--     val := t0 + jitter * (t1-t0)/10::F
--     t.2 := val; p.2 := f val
--     t := rest t; p := rest p
--     t := headert; p := headerp

st := t; sp := p
todot : L L F := nil()
todop : L L P := nil()
while not null rest rest st repeat
  todot := concat_!(todot, st)
  todop := concat_!(todop, sp)
  st := rest st; sp := rest sp
st := headert; sp := headerp
todo1 := todot; todo2 := todop
n : I := 0
while not null todo1 repeat
  st := first(todo1)
  t0 := first(st); t1 := second(st); t2 := third(st)
  if t2 > h then leave
  t2 - t0 < tLimit =>
    todo1 := rest todo1
    todo2 := rest todo2
    if not null todo1 then (t := first(todo1); p := first(todo2))
  sp := first(todo2)
  x0 := xCoord first(sp); y0 := yCoord first(sp)
  x1 := xCoord second(sp); y1 := yCoord second(sp)
  x2 := xCoord third(sp); y2 := yCoord third(sp)
  a1 := (x1-x0)/xDiff; b1 := (y1-y0)/yDiff

```

```

a2 := (x2-x1)/xDiff; b2 := (y2-y1)/yDiff
s1 := sqrt(a1**2+b1**2); s2 := sqrt(a2**2+b2**2)
dp := a1*a2+b1*b2

s1 < maxLength and s2 < maxLength and _
(s1 = 0::F or s2 = 0::F or
 s1 < minLength and s2 < minLength or _
 dp/s1/s2 > ANGLEBOUND) =>
  todo1 := rest todo1
  todo2 := rest todo2
  if not null todo1 then (t := first(todo1); p := first(todo2))
if n > MAXPOINTS then leave else n := n + 1
st := rest t
if not null rest rest st then
  tm := (t0+t1)/2::F
  tj := tm
  t.rest := concat(tj,rest t)
  p.rest := concat(f tj, rest p)
  todo1 := concat_!(todo1, t)
  todo2 := concat_!(todo2, p)
  t := rest t; p := rest p
  todo1 := concat_!(todo1, t)
  todo2 := concat_!(todo2, p)
  t := rest t; p := rest p
  todo1 := rest todo1; todo2 := rest todo2

  tm := (t1+t2)/2::F
  tj := tm
  t.rest := concat(tj, rest t)
  p.rest := concat(f tj, rest p)
  todo1 := concat_!(todo1, t)
  todo2 := concat_!(todo2, p)
  t := rest t; p := rest p
  todo1 := concat_!(todo1, t)
  todo2 := concat_!(todo2, p)
  todo1 := rest todo1
  todo2 := rest todo2
  if not null todo1 then (t := first(todo1); p := first(todo2))
else
  tm := (t0+t1)/2::F
  tj := tm
  t.rest := concat(tj,rest t)
  p.rest := concat(f tj, rest p)
  todo1 := concat_!(todo1, t)
  todo2 := concat_!(todo2, p)
  t := rest t; p := rest p
  todo1 := concat_!(todo1, t)
  todo2 := concat_!(todo2, p)
  t := rest t; p := rest p

```

```

    tm := (t1+t2)/2::F
    tj := tm
    t.rest := concat(tj, rest t)
    p.rest := concat(f tj, rest p)
    todo1 := concat_!(todo1, t)
    todo2 := concat_!(todo2, p)
    todo1 := rest todo1
    todo2 := rest todo2
    if not null todo1 then (t := first(todo1); p := first(todo2))
n > 0 =>
    NUMFUNEVALS := NUMFUNEVALS + n
    t := curve.knots; p := curve.points
    xRange := select(p,xCoord,min) .. select(p,xCoord,max)
    yRange := select(p,yCoord,min) .. select(p,yCoord,max)
    [ curve.source, [tRange,xRange,yRange], t, p ]
curve

basicPlot(f,tRange) ==
    checkRange tRange
    l := lo tRange
    h := hi tRange
    t : L F := list l
    p : L P := list f l
    s := (h-l)/(minPoints()-1)::F
    for i in 2..minPoints()-1 repeat
        l := l+s
        t := concat(l,t)
        p := concat(f l,p)
    t := reverse_! concat(h,t)
    p := reverse_! concat(f h,p)
--    print(p::OutputForm)
    xRange : R := select(p,xCoord,min) .. select(p,xCoord,max)
    yRange : R := select(p,yCoord,min) .. select(p,yCoord,max)
    [ f, [tRange,xRange,yRange], t, p ]

zoom(p,xRange) ==
    [p.parametric, [xRange,third(p.display)], p.bounds, _
    p.axisLabels, p.functions]
zoom(p,xRange,yRange) ==
    [p.parametric, [xRange,yRange], p.bounds, _
    p.axisLabels, p.functions]

basicRefine(curve,nRange) ==
    tRange:R := first curve.ranges
    -- curve := copy$C curve -- Yet another compiler bug
    curve: C := [curve.source,curve.ranges,curve.knots,curve.points]
    t := curve.knots := copy curve.knots
    p := curve.points := copy curve.points
    l := lo nRange; h := hi nRange
    f := curve.source

```



```

while not null rest t and first t < h repeat
  second(t) < 1 => (t := rest t; p := rest p)
  -- insert new point between t.0 and t.1
  tm : F := (first(t) + second(t))/2::F
  -- if DEBUG then output$0 (tm::E)
  pm := f tm
  NUMFUNEVALS := NUMFUNEVALS + 1
  t.rest := concat(tm,rest t); t := rest rest t
  p.rest := concat(pm,rest p); p := rest rest p
t := curve.knots; p := curve.points
xRange := select(p,xCoord,min) .. select(p,xCoord,max)
yRange := select(p,yCoord,min) .. select(p,yCoord,max)
[ curve.source, [tRange,xRange,yRange], t, p ]

refine p == refine(p,parametricRange p)
refine(p,nRange) ==
  NUMFUNEVALS := 0
  tRange := parametricRange p
  nRange := intersect(tRange,nRange)
  curves: L C := [basicRefine(c,nRange) for c in p.functions]
  xRange := join(curves,1); yRange := join(curves,2)
  if adaptive? then
    tlimit := if parametric? p then 8 else 1
    curves := [adaptivePlot(c,nRange,xRange,yRange, _
      tlimit) for c in curves]
    xRange := join(curves,1); yRange := join(curves,2)
  -- print(NUMFUNEVALS::OUT)
  [p.parametric, p.display, [tRange,xRange,yRange], _
    p.axisLabels, curves ]

plot(p:%,tRange:R) ==
  -- re plot p on a new range making use of the points already
  -- computed if possible
  NUMFUNEVALS := 0
  curves: L C := [rangeRefine(c,tRange) for c in p.functions]
  xRange := join(curves,1); yRange := join(curves,2)
  if adaptive? then
    tlimit := if parametric? p then 8 else 1
    curves := [adaptivePlot(c,tRange,xRange,yRange,tlimit) for c in curves]
    xRange := join(curves,1); yRange := join(curves,2)
  -- print(NUMFUNEVALS::OUT)
  [ p.parametric, [xRange,yRange], [tRange,xRange,yRange],
    p.axisLabels, curves ]

pt(xx,yy) == point(1 : L F := [xx,yy])

myTrap: (F-> F, F) -> F
myTrap(ff:F-> F, f:F):F ==
  s := trapNumericErrors(ff(f))$Lisp :: Union(F, "failed")
  s case "failed" => _$NaNvalue$Lisp

```

```

r:F:=s::F
r > max()$F or r < min()$F => _$NaNvalue$Lisp
r

plot(f:F -> F,xRange:R) ==
  p := basicPlot((u1:F):P +-> pt(u1,myTrap(f,u1)),xRange)
  r := p.ranges
  NUMFUNEVALS := minPoints()
  if adaptive? then
    p := adaptivePlot(p,first r,second r,third r,1)
    r := p.ranges
  [ false, rest r, r, nil(), [ p ] ]

plot(f:F -> F,xRange:R,yRange:R) ==
  p := plot(f,xRange)
  p.display := [xRange,checkRange yRange]
  p

plot(f:F -> F,g:F -> F,tRange:R) ==
  p := basicPlot((z1:F):P +-> pt(myTrap(f,z1),myTrap(g,z1)),tRange)
  r := p.ranges
  NUMFUNEVALS := minPoints()
  if adaptive? then
    p := adaptivePlot(p,first r,second r,third r,8)
    r := p.ranges
  [ true, rest r, r, nil(), [ p ] ]

plot(f:F -> F,g:F -> F,tRange:R,xRange:R,yRange:R) ==
  p := plot(f,g,tRange)
  p.display := [checkRange xRange,checkRange yRange]
  p

pointPlot(f:F -> P,tRange:R) ==
  p := basicPlot(f,tRange)
  r := p.ranges
  NUMFUNEVALS := minPoints()
  if adaptive? then
    p := adaptivePlot(p,first r,second r,third r,8)
    r := p.ranges
  [ true, rest r, r, nil(), [ p ] ]

pointPlot(f:F -> P,tRange:R,xRange:R,yRange:R) ==
  p := pointPlot(f,tRange)
  p.display := [checkRange xRange,checkRange yRange]
  p

plot(l:L(F -> F),xRange:R) ==
  if null l then error "empty list of functions"
  t: L C :=
    [ basicPlot((z1:F):P +-> pt(z1,myTrap(f,z1)),xRange) for f in l ]

```

```

yRange := join(t,2)
NUMFUNEVALS := # l * minPoints()
if adaptive? then
  t := [adaptivePlot(p,xRange,xRange,yRange,1) _
        for f in l for p in t]
  yRange := join(t,2)
-- print(NUMFUNEVALS::OUT)
[false, [xRange,yRange], [xRange,xRange,yRange], nil(), t ]

plot(l:L(F -> F),xRange:R,yRange:R) ==
p := plot(l,xRange)
p.display := [xRange,checkRange yRange]
p

plotPolar(f,thetaRange) ==
plot((u1:F):F +-> f(u1) * cos(u1),
      (v1:F):F +-> f(v1) * sin(v1),thetaRange)

plotPolar f == plotPolar(f,segment(0,2*pi()))

--% terminal output

coerce r ==
spaces: OUT := coerce "  "
xSymbol := "x = " :: OUT
ySymbol := "y = " :: OUT
tSymbol := "t = " :: OUT
plotSymbol := "PLOT" :: OUT
tRange := (parametricRange r) :: OUT
f : L OUT := nil()
for curve in r.functions repeat
  xRange := second(curve.ranges) :: OUT
  yRange := third(curve.ranges) :: OUT
  l : L OUT := [xSymbol,xRange,spaces,ySymbol,yRange]
  if parametric? r then
    l := concat_!([tSymbol,tRange,spaces],l)
  h : OUT := hconcat l
  l := [p::OUT for p in curve.points]
  f := concat(vconcat concat(h,l),f)
prefix("PLOT" :: OUT, reverse_! f)

```

— PLOT.dotabb —

```

"PLOT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PLOT"]
"FRAC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FRAC"]
"PLOT" -> "FRAC"

```

domain PLOT3D Plot3D

— Plot3D.input —

```
)set break resume
)sys rm -f Plot3D.output
)spool Plot3D.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Plot3D
--R
--R Plot3D is a domain constructor
--R Abbreviation for Plot3D is PLOT3D
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PLOT3D
--R
--R----- Operations -----
--R adaptive3D? : () -> Boolean          coerce : % -> OutputForm
--R debug3D : Boolean -> Boolean          maxPoints3D : () -> Integer
--R minPoints3D : () -> Integer          numFunEvals3D : () -> Integer
--R refine : % -> %                      screenResolution3D : () -> Integer
--R setAdaptive3D : Boolean -> Boolean
--R listBranches : % -> List(List(Point(DoubleFloat)))
--R plot : (% , Segment(DoubleFloat)) -> %
--R plot : ((DoubleFloat -> DoubleFloat), (DoubleFloat -> DoubleFloat), (DoubleFloat -> DoubleFloat), (DoubleF
--R plot : ((DoubleFloat -> DoubleFloat), (DoubleFloat -> DoubleFloat), (DoubleFloat -> DoubleFloat), (DoubleF
--R pointPlot : ((DoubleFloat -> Point(DoubleFloat)), Segment(DoubleFloat), Segment(DoubleFloat), Segment(Doub
--R pointPlot : ((DoubleFloat -> Point(DoubleFloat)), Segment(DoubleFloat)) -> %
--R refine : (% , Segment(DoubleFloat)) -> %
--R setMaxPoints3D : Integer -> Integer
--R setMinPoints3D : Integer -> Integer
--R setScreenResolution3D : Integer -> Integer
--R tRange : % -> Segment(DoubleFloat)
--R tValues : % -> List(List(DoubleFloat))
--R xRange : % -> Segment(DoubleFloat)
--R yRange : % -> Segment(DoubleFloat)
--R zRange : % -> Segment(DoubleFloat)
--R zoom : (% , Segment(DoubleFloat), Segment(DoubleFloat), Segment(DoubleFloat)) -> %
--R
--E 1
```

```
)spool
)lisp (bye)
```

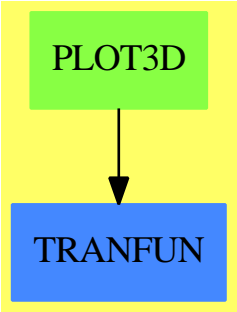
— Plot3D.help —

```
=====
Plot3D examples
=====
```

Plot3D supports parametric plots defined over a real number system. A real number system is a model for the real numbers and as such may be an approximation. For example, floating point numbers and infinite continued fractions are real number systems. The facilities at this point are limited to 3-dimensional parametric plots.

See Also:
o)show Plot3D

Plot3D (PLOT3D)



Exports:

adaptive3D?	coerce	debug3D	listBranches	maxPoints3D
minPoints3D	numFunEvals3D	plot	pointPlot	refine
screenResolution3D	setAdaptive3D	setMaxPoints3D	setMinPoints3D	setScreenResolution3D
tRange	tValues	xRange	yRange	zRange
zoom				

— domain PLOT3D Plot3D —

```

)abbrev domain PLOT3D Plot3D
++ Author: Clifton J. Williamson based on code by Michael Monagan
++ Date Created: Jan 1989
++ Date Last Updated: 22 November 1990 (Jon Steinbach)
++ Description:
++ Plot3D supports parametric plots defined over a real
++ number system. A real number system is a model for the real
++ numbers and as such may be an approximation. For example,
++ floating point numbers and infinite continued fractions are
++ real number systems. The facilities at this point are limited
++ to 3-dimensional parametric plots.

Plot3D(): Exports == Implementation where
  B ==> Boolean
  F ==> DoubleFloat
  I ==> Integer
  L ==> List
  N ==> NonNegativeInteger
  OUT ==> OutputForm
  P ==> Point F
  S ==> String
  R ==> Segment F
  O ==> OutputPackage
  C ==> Record(source: F -> P, ranges: L R, knots: L F, points: L P)

Exports ==> PlottableSpaceCurveCategory with

  pointPlot: (F -> P,R) -> %
    ++ pointPlot(f,g,h,a..b) plots {/emx = f(t), y = g(t), z = h(t)} as
    ++ t ranges over {/em[a,b]}.
  pointPlot: (F -> P,R,R,R,R) -> %
    ++ pointPlot(f,x,y,z,w) is not documented
  plot: (F -> F,F -> F,F -> F,F -> F,R) -> %
    ++ plot(f,g,h,a..b) plots {/emx = f(t), y = g(t), z = h(t)} as
    ++ t ranges over {/em[a,b]}.
  plot: (F -> F,F -> F,F -> F,F -> F,R,R,R,R) -> %
    ++ plot(f1,f2,f3,f4,x,y,z,w) is not documented

  plot: (% ,R) -> % -- change the range
    ++ plot(x,r) is not documented
  zoom: (% ,R,R,R) -> %
    ++ zoom(x,r,s,t) is not documented
  refine: (% ,R) -> %
    ++ refine(x,r) is not documented
  refine: % -> %
    ++ refine(x) is not documented

  tRange: % -> R
    ++ tRange(p) returns the range of the parameter in a parametric plot p.
  tValues: % -> L L F

```

```

++ tValues(p) returns a list of lists of the values of the parameter for
++ which a point is computed, one list for each curve in the plot p.

minPoints3D: () -> I
++ minPoints3D() returns the minimum number of points in a plot.
setMinPoints3D: I -> I
++ setMinPoints3D(i) sets the minimum number of points in a plot to i.
maxPoints3D: () -> I
++ maxPoints3D() returns the maximum number of points in a plot.
setMaxPoints3D: I -> I
++ setMaxPoints3D(i) sets the maximum number of points in a plot to i.
screenResolution3D: () -> I
++ screenResolution3D() returns the screen resolution for a 3d graph.
setScreenResolution3D: I -> I
++ setScreenResolution3D(i) sets the screen resolution for a 3d graph to i.
adaptive3D?: () -> B
++ adaptive3D?() determines whether plotting be done adaptively.
setAdaptive3D: B -> B
++ setAdaptive3D(true) turns adaptive plotting on;
++ setAdaptive3D(false) turns adaptive plotting off.
numFunEvals3D: () -> I
++ numFunEvals3D() returns the number of points computed.
debug3D: B -> B
++ debug3D(true) turns debug mode on;
++ debug3D(false) turns debug mode off.

Implementation ==> add
import PointPackage(F)

--% local functions

fourth      : L R -> R
checkRange  : R -> R
-- checks that left-hand endpoint is less than right-hand endpoint
intersect   : (R,R) -> R
-- intersection of two intervals
union       : (R,R) -> R
-- union of two intervals
join        : (L C,I) -> R
parametricRange: % -> R
-- setColor   : (P,F) -> F
select       : (L P,P -> F,(F,F) -> F) -> F
-- normalizeColor : (P,F,F) -> F
rangeRefine  : (C,R) -> C
adaptivePlot : (C,R,R,R,R,I,I) -> C
basicPlot    : (F -> P,R) -> C
basicRefine  : (C,R) -> C
point        : (F,F,F,F) -> P

--% representation

```

```

Rep := Record( display: L R, _
               bounds: L R, _
               screenres: I, _
               axisLabels: L S, _
               functions: L C )

--% global constants

ADAPTIVE      : B := true
MINPOINTS     : I := 49
MAXPOINTS     : I := 1000
NUMFUNEVALS   : I := 0
SCREENRES     : I := 500
ANGLEBOUND    : F := cos inv (4::F)
DEBUG         : B := false

point(xx,yy,zz,col) == point(l : L F := [xx,yy,zz,col])

fourth list == first rest rest rest list

checkRange r == (lo r > hi r => error "ranges cannot be negative"; r)
intersect(s,t) == checkRange (max(lo s,lo t) .. min(hi s,hi t))
union(s:R,t:R) == min(lo s,lo t) .. max(hi s,hi t)
join(l,i) ==
  rr := first l
  u : R :=
    i = 0 => first(rr.ranges)
    i = 1 => second(rr.ranges)
    i = 2 => third(rr.ranges)
    fourth(rr.ranges)
  for r in rest l repeat
    i = 0 => union(u,first(r.ranges))
    i = 1 => union(u,second(r.ranges))
    i = 2 => union(u,third(r.ranges))
    union(u,fourth(r.ranges))
  u
parametricRange r == first(r.bounds)

minPoints3D() == MINPOINTS
setMinPoints3D n ==
  if n < 3 then error "three points minimum required"
  if MAXPOINTS < n then MAXPOINTS := n
  MINPOINTS := n
maxPoints3D() == MAXPOINTS
setMaxPoints3D n ==
  if n < 3 then error "three points minimum required"
  if MINPOINTS > n then MINPOINTS := n
  MAXPOINTS := n
screenResolution3D() == SCREENRES

```



```

setScreenResolution3D n ==
  if n < 2 then error "buy a new terminal"
  SCREENRES := n
adaptive3D?() == ADAPTIVE
setAdaptive3D b == ADAPTIVE := b

numFunEvals3D() == NUMFUNEVALS
debug3D b == DEBUG := b

--      setColor(p,c) == p.colNum := c

xRange plot == second plot.bounds
yRange plot == third plot.bounds
zRange plot == fourth plot.bounds
tRange plot == first plot.bounds

tValues plot ==
  outList : L L F := nil()
  for curve in plot.functions repeat
    outList := concat(curve.knots,outList)
  outList

select(l,f,g) ==
  m := f first l
  if (EQL(m, _$NaNvalue$Lisp)$Lisp) then m := 0
  --   for p in rest l repeat m := g(m,fp)
  for p in rest l repeat
    fp : F := f p
    if (EQL(fp, _$NaNvalue$Lisp)$Lisp) then fp := 0
    m := g(m,fp)
  m

--      normalizeColor(p,lo,diff) ==
--      p.colNum := (p.colNum - lo)/diff

rangeRefine(curve,nRange) ==
  checkRange nRange; l := lo nRange; h := hi nRange
  t := curve.knots; p := curve.points; f := curve.source
  while not null t and first t < l repeat
    (t := rest t; p := rest p)
  c : L F := nil(); q : L P := nil()
  while not null t and first t <= h repeat
    c := concat(first t,c); q := concat(first p,q)
    t := rest t; p := rest p
  if null c then return basicPlot(f,nRange)
  if first c < h then
    c := concat(h,c); q := concat(f h,q)
    NUMFUNEVALS := NUMFUNEVALS + 1
  t := c := reverse_! c; p := q := reverse_! q
  s := (h-1)/(MINPOINTS::F-1)

```

```

if (first t) ^= 1 then
  t := c := concat(1,c); p := q := concat(f 1,p)
  NUMFUNEVALS := NUMFUNEVALS + 1
while not null rest t repeat
  n := wholePart((second(t) - first(t))/s)
  d := (second(t) - first(t))/((n+1)::F)
  for i in 1..n repeat
    t.rest := concat(first(t) + d,rest t); t1 := second t
    p.rest := concat(f t1,rest p)
    NUMFUNEVALS := NUMFUNEVALS + 1
    t := rest t; p := rest p
  t := rest t
  p := rest p
xRange := select(q,xCoord,min) .. select(q,xCoord,max)
yRange := select(q,yCoord,min) .. select(q,yCoord,max)
zRange := select(q,zCoord,min) .. select(q,zCoord,max)
--   colorLo := select(q,color,min); colorHi := select(q,color,max)
--   (diff := colorHi - colorLo) = 0 =>
--     error "all points are the same color"
--   map(normalizeColor(#1,colorLo,diff),q)$ListPackage1(P)
[f,[nRange,xRange,yRange,zRange],c,q]

adaptivePlot(curve,tRg,xRg,yRg,zRg,pixelfraction,resolution) ==
  xDiff := hi xRg - lo xRg
  yDiff := hi yRg - lo yRg
  zDiff := hi zRg - lo zRg
--   xDiff = 0 or yDiff = 0 or zDiff = 0 => curve--!! delete this?
  if xDiff = 0::F then xDiff := 1::F
  if yDiff = 0::F then yDiff := 1::F
  if zDiff = 0::F then zDiff := 1::F
  l := lo tRg; h := hi tRg
  (tDiff := h-l) = 0 => curve
  t := curve.knots
  #t < 3 => curve
  p := curve.points; f := curve.source
  minLength:F := 4::F/resolution::F
  maxLength := 1/4::F
  tLimit := tDiff/(pixelfraction*resolution)::F
  while not null t and first t < l repeat (t := rest t; p := rest p)
  #t < 3 => curve
  headert := t; headerp := p
  st := t; sp := p
  todot : L L F := nil()
  todop : L L P := nil()
  while not null rest rest st repeat
    todot := concat_!(todot, st)
    todop := concat_!(todop, sp)
    st := rest st; sp := rest sp
  st := headert; sp := headerp

```

```

todo1 := todot; todo2 := todop
n : I := 0

while not null todo1 repeat
  st := first(todo1)
  t0 := first(st); t1 := second(st); t2 := third(st)
  if t2 > h then leave
  t2 - t0 < tLimit =>
    todo1 := rest todo1
    todo2 := rest todo2;
    if not null todo1 then (t := first(todo1); p := first(todo2))
  sp := first(todo2)
  x0 := xCoord first(sp); y0 := yCoord first(sp); z0 := zCoord first(sp)
  x1 := xCoord second(sp); y1 := yCoord second(sp); z1 := zCoord second(sp)
  x2 := xCoord third(sp); y2 := yCoord third(sp); z2 := zCoord third(sp)
  a1 := (x1-x0)/xDiff; b1 := (y1-y0)/yDiff; c1 := (z1-z0)/zDiff
  a2 := (x2-x1)/xDiff; b2 := (y2-y1)/yDiff; c2 := (z2-z1)/zDiff
  s1 := sqrt(a1**2+b1**2+c1**2); s2 := sqrt(a2**2+b2**2+c2**2)
  dp := a1*a2+b1*b2+c1*c2
  s1 < maxLength and s2 < maxLength and _
    (s1 = 0 or s2 = 0 or
      s1 < minLength and s2 < minLength or _
      dp/s1/s2 > ANGLEBOUND) =>
      todo1 := rest todo1
      todo2 := rest todo2
      if not null todo1 then (t := first(todo1); p := first(todo2))
if n = MAXPOINTS then leave else n := n + 1
--if DEBUG then
  --r : L F := [minLength,maxLength,s1,s2,dp/s1/s2,ANGLEBOUND]
  --output(r::E)$0
st := rest t
if not null rest rest st then
  tm := (t0+t1)/2::F
  tj := tm
  t.rest := concat(tj,rest t)
  p.rest := concat(f tj, rest p)
  todo1 := concat_!(todo1, t)
  todo2 := concat_!(todo2, p)
  t := rest t; p := rest p
  todo1 := concat_!(todo1, t)
  todo2 := concat_!(todo2, p)
  t := rest t; p := rest p
  todo1 := rest todo1; todo2 := rest todo2

tm := (t1+t2)/2::F
tj := tm
t.rest := concat(tj, rest t)
p.rest := concat(f tj, rest p)
todo1 := concat_!(todo1, t)
todo2 := concat_!(todo2, p)

```

```

    t := rest t; p := rest p
    todo1 := concat_!(todo1, t)
    todo2 := concat_!(todo2, p)
    todo1 := rest todo1; todo2 := rest todo2
    if not null todo1 then (t := first(todo1); p := first(todo2))
  else
    tm := (t0+t1)/2::F
    tj := tm
    t.rest := concat(tj, rest t)
    p.rest := concat(f tj, rest p)
    todo1 := concat_!(todo1, t)
    todo2 := concat_!(todo2, p)
    t := rest t; p := rest p
    todo1 := concat_!(todo1, t)
    todo2 := concat_!(todo2, p)
    t := rest t; p := rest p

    tm := (t1+t2)/2::F
    tj := tm
    t.rest := concat(tj, rest t)
    p.rest := concat(f tj, rest p)
    todo1 := concat_!(todo1, t)
    todo2 := concat_!(todo2, p)
    todo1 := rest todo1; todo2 := rest todo2
    if not null todo1 then (t := first(todo1); p := first(todo2))
  if n > 0 then
    NUMFUNEVALS := NUMFUNEVALS + n
    t := curve.knots; p := curve.points
    xRg := select(p,xCoord,min) .. select(p,xCoord,max)
    yRg := select(p,yCoord,min) .. select(p,yCoord,max)
    zRg := select(p,zCoord,min) .. select(p,zCoord,max)
    [curve.source, [tRg,xRg,yRg,zRg], t, p]
  else curve

basicPlot(f,tRange) ==
  checkRange tRange; l := lo tRange; h := hi tRange
  t : L F := list l; p : L P := list f l
  s := (h-l)/(MINPOINTS-1)::F
  for i in 2..MINPOINTS-1 repeat
    l := l+s; t := concat(l,t)
    p := concat(f l,p)
  t := reverse_! concat(h,t)
  p := reverse_! concat(f h,p)
  xRange : R := select(p,xCoord,min) .. select(p,xCoord,max)
  yRange : R := select(p,yCoord,min) .. select(p,yCoord,max)
  zRange : R := select(p,zCoord,min) .. select(p,zCoord,max)
  [f, [tRange,xRange,yRange,zRange], t, p]

zoom(p,xRange,yRange,zRange) ==
  [[xRange,yRange,zRange], p.bounds,
```

```

p.screenres,p.axisLabels,p.functions]

basicRefine(curve,nRange) ==
  tRange:R := first curve.ranges
  -- curve := copy$C curve -- Yet another @$$^&* compiler bug
  curve: C := [curve.source,curve.ranges,curve.knots,curve.points]
  t := curve.knots := copy curve.knots
  p := curve.points := copy curve.points
  l := lo nRange; h := hi nRange
  f := curve.source
  while not null rest t and first(t) < h repeat
    second(t) < l => (t := rest t; p := rest p)
    -- insert new point between t.0 and t.1
    tm:F := (first(t) + second(t))/2::F
    -- if DEBUG then output$0 (tm::E)
    pm := f tm
    NUMFUNEVALS := NUMFUNEVALS + 1
    t.rest := concat(tm,rest t); t := rest rest t
    p.rest := concat(pm,rest p); p := rest rest p
  t := curve.knots; p := curve.points
  xRange := select(p,xCoord,min) .. select(p,xCoord,max)
  yRange := select(p,yCoord,min) .. select(p,yCoord,max)
  zRange := select(p,zCoord,min) .. select(p,zCoord,max)
  [curve.source,[tRange,xRange,yRange,zRange],t,p]

refine p == refine(p,parametricRange p)
refine(p,nRange) ==
  NUMFUNEVALS := 0
  tRange := parametricRange p
  nRange := intersect(tRange,nRange)
  curves: L C := [basicRefine(c,nRange) for c in p.functions]
  xRange := join(curves,1); yRange := join(curves,2)
  zRange := join(curves,3)
  scrres := p.screenres
  if adaptive3D? then
    tlimit := 8
    curves := [adaptivePlot(c,nRange,xRange,yRange,zRange, _
      tlimit,scrres := 2*scrres) for c in curves]
    xRange := join(curves,1); yRange := join(curves,2)
    zRange := join(curves,3)
  [p.display,[tRange,xRange,yRange,zRange], _
    scrres,p.axisLabels,curves]

plot(p:%,tRange:R) ==
  -- re plot p on a new range making use of the points already
  -- computed if possible
  NUMFUNEVALS := 0
  curves: L C := [rangeRefine(c,tRange) for c in p.functions]
  xRange := join(curves,1); yRange := join(curves,2)
  zRange := join(curves,3)

```

```

    if adaptive3D? then
        tlimit := 8
        curves := [adaptivePlot(c,tRange,xRange,yRange,zRange,tlimit, _
            p.screenres) for c in curves]
        xRange := join(curves,1); yRange := join(curves,2)
        zRange := join(curves,3)
--      print(NUMFUNEVALS::OUT)
        [[xRange,yRange,zRange],[tRange,xRange,yRange,zRange],
            p.screenres,p.axisLabels,curves]

pointPlot(f:F -> P,tRange:R) ==
    p := basicPlot(f,tRange)
    r := p.ranges
    NUMFUNEVALS := MINPOINTS
    if adaptive3D? then
        p := adaptivePlot(p,first r,second r,third r,fourth r,8,SCREENRES)
--      print(NUMFUNEVALS::OUT)
--      print(p::OUT)
    [ rest r, r, SCREENRES, nil(), [ p ] ]

pointPlot(f:F -> P,tRange:R,xRange:R,yRange:R,zRange:R) ==
    p := pointPlot(f,tRange)
    p.display:= [checkRange xRange,checkRange yRange,checkRange zRange]
    p

myTrap: (F-> F, F) -> F
myTrap(ff:F-> F, f:F):F ==
    s := trapNumericErrors(ff(f))$Lisp :: Union(F, "failed")
    if (s) case "failed" then
        r:F := _$NaNvalue$Lisp
    else
        r:F := s
    r

plot(f1:F -> F,f2:F -> F,f3:F -> F,col:F -> F,tRange:R) ==
    p := basicPlot(
        (z:F):P+>point(myTrap(f1,z),myTrap(f2,z),myTrap(f3,z),col(z)),tRange)
    r := p.ranges
    NUMFUNEVALS := MINPOINTS
    if adaptive3D? then
        p := adaptivePlot(p,first r,second r,third r,fourth r,8,SCREENRES)
--      print(NUMFUNEVALS::OUT)
    [ rest r, r, SCREENRES, nil(), [ p ] ]

plot(f1:F -> F,f2:F -> F,f3:F -> F,col:F -> F,_
    tRange:R,xRange:R,yRange:R,zRange:R) ==
    p := plot(f1,f2,f3,col,tRange)
    p.display:= [checkRange xRange,checkRange yRange,checkRange zRange]
    p

```

--% terminal output

```
coerce r ==
  spaces := " " :: OUT
  xSymbol := "x = " :: OUT; ySymbol := "y = " :: OUT
  zSymbol := "z = " :: OUT; tSymbol := "t = " :: OUT
  tRange := (parametricRange r) :: OUT
  f : L OUT := nil()
  for curve in r.functions repeat
    xRange := coerce curve.ranges.1
    yRange := coerce curve.ranges.2
    zRange := coerce curve.ranges.3
    l : L OUT := [xSymbol,xRange,spaces,ySymbol,yRange,_
                  spaces,zSymbol,zRange]
    l := concat_!([tSymbol,tRange,spaces],l)
    h : OUT := hconcat l
    l := [p::OUT for p in curve.points]
    f := concat(vconcat concat(h,l),f)
  prefix("PLOT" :: OUT,reverse_! f)
```

----% graphics output

```
listBranches plot ==
  outList : L L P := nil()
  for curve in plot.functions repeat
    outList := concat(curve.points,outList)
  outList
```

— PLOT3D.dotabb —

```
"PLOT3D" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PLOT3D"]
"TRANFUN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TRANFUN"]
"PLOT3D" -> "TRANFUN"
```

domain PBWLB PoincareBirkhoffWittLyndonBasis

— PoincareBirkhoffWittLyndonBasis.input —

```
)set break resume
)sys rm -f PoincareBirkhoffWittLyndonBasis.output
```

```

)spool PoincareBirkhoffWittLyndonBasis.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PoincareBirkhoffWittLyndonBasis
--R
--R PoincareBirkhoffWittLyndonBasis(VarSet: OrderedSet) is a domain constructor
--R Abbreviation for PoincareBirkhoffWittLyndonBasis is PBWLB
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PBWLB
--R
--R----- Operations -----
--R ?<? : (%,%) -> Boolean          ?<=? : (%,%) -> Boolean
--R ?=? : (%,%) -> Boolean          ?>? : (%,%) -> Boolean
--R ?>=? : (%,%) -> Boolean          1 : () -> %
--R coerce : VarSet -> %             coerce : LyndonWord(VarSet) -> %
--R coerce : % -> OutputForm          first : % -> LyndonWord(VarSet)
--R hash : % -> SingleInteger          latex : % -> String
--R length : % -> NonNegativeInteger  max : (%,%) -> %
--R min : (%,%) -> %                  rest : % -> %
--R retract : % -> LyndonWord(VarSet) retractable? : % -> Boolean
--R varList : % -> List(VarSet)        ?~=? : (%,%) -> Boolean
--R coerce : % -> OrderedFreeMonoid(VarSet)
--R listOfTerms : % -> List(LyndonWord(VarSet))
--R retractIfCan : % -> Union(LyndonWord(VarSet), "failed")
--R
--E 1

)spool
)lisp (bye)

```

— PoincareBirkhoffWittLyndonBasis.help —

```

=====
PoincareBirkhoffWittLyndonBasis examples
=====

```

This domain provides the internal representation of polynomials in non-commutative variables written over the Poincare-Birkhoff-Witt basis. See the XPBWPolynomial domain constructor.

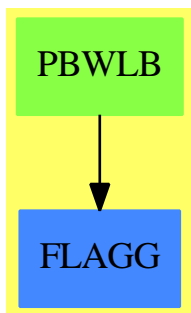
See Also:

```

o )show PoincareBirkhoffWittLyndonBasis
o )show XPBWPolynomial

```


PoincareBirkhoffWittLyndonBasis (PBWLB)



Exports:

1	coerce	first	hash	latex
length	listOfTerms	max	min	rest
retract	retractable?	retractIfCan	varList	?~=?
?<?	?<=?	?=?	?>?	?>=?

— domain PBWLB PoincareBirkhoffWittLyndonBasis —

```

)abbrev domain PBWLB PoincareBirkhoffWittLyndonBasis
++ Author: Michel Petitot (petitot@lifl.fr).
++ Date Created: 91
++ Date Last Updated: 7 Juillet 92
++ Description:
++ This domain provides the internal representation
++ of polynomials in non-commutative variables written
++ over the Poincare-Birkhoff-Witt basis.
++ See the \spadtype{XPBWPolynomial} domain constructor.
++ See Free Lie Algebras by C. Reutenauer
++ (Oxford science publications).

PoincareBirkhoffWittLyndonBasis(VarSet: OrderedSet): Public == Private where
  WORD    ==> OrderedFreeMonoid(VarSet)
  LWORD    ==> LyndonWord(VarSet)
  LWORDS   ==> List(LWORD)
  PI       ==> PositiveInteger
  NNI      ==> NonNegativeInteger
  EX       ==> OutputForm

Public == Join(OrderedSet, RetractableTo LWORD) with
  1: constant -> %
      ++ \spad{1} returns the empty list.
  coerce      : $ -> WORD
  
```

```

    ++ \spad{coerce([l1]*[l2]*...[ln])} returns the word \spad{l1*l2*...*ln},
    ++ where \spad{[l_i]} is the bracketed form of the Lyndon word \spad{l_i}.
coerce      : VarSet -> $
    ++ \spad{coerce(v)} return \spad{v}
first       : $ -> LWORD
    ++ \spad{first([l1]*[l2]*...[ln])} returns the Lyndon word \spad{l1}.
length      : $ -> NNI
    ++ \spad{length([l1]*[l2]*...[ln])} returns the length of the word \spad{l1*l2*...*ln}.
listOfTerms : $ -> LWORDS
    ++ \spad{listOfTerms([l1]*[l2]*...[ln])} returns the list of words \spad{l1, l2, .... ln}.
rest        : $ -> $
    ++ \spad{rest([l1]*[l2]*...[ln])} returns the list \spad{l2, .... ln}.
retractable? : $ -> Boolean
    ++ \spad{retractable?([l1]*[l2]*...[ln])} returns true iff \spad{n} equals \spad{1}.
varList     : $ -> List VarSet
    ++ \spad{varList([l1]*[l2]*...[ln])} returns the list of
    ++ variables in the word \spad{l1*l2*...*ln}.

Private == add

-- Representation
Rep := LWORDS

-- Locales
recursif: ($,$) -> Boolean

-- Define
1 == nil

x = y == x == $Rep y

varList x ==
    null x => nil
    le: List VarSet := "setUnion"/ [varList$LWORD l for l in x]

first x == first(x)$Rep
rest x == rest(x)$Rep

coerce(v: VarSet):$ == [ v::LWORD ]
coerce(l: LWORD):$ == [l]
listOfTerms(x:$):LWORDS == x pretend LWORDS

coerce(x:$):WORD ==
    null x => 1
    x.first :: WORD *$WORD coerce(x.rest)

coerce(x:$):EX ==
    null x => outputForm(1$Integer)$EX
    reduce(_* , [l :: EX for l in x])$List(EX)

```

```

retractable? x ==
  null x => false
  null x.rest

retract x ==
  #x ^= 1 => error "cannot convert to Lyndon word"
  x.first

retractIfCan x ==
  retractable? x => x.first
  "failed"

length x ==
  n: Integer := +/[ length l for l in x]
  n::NNI

recursif(x, y) ==
  null y => false
  null x => true
  x.first = y.first => recursif(rest(x), rest(y))
  lexico(x.first, y.first)

x < y ==
  lx: NNI := length x; ly: NNI := length y
  lx = ly => recursif(x,y)
  lx < ly

```

— PBWLB.dotabb —

```

"PBWLB" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PBWLB"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PBWLB" -> "FLAGG"

```

domain POINT Point

— Point.input —

```

)set break resume
)sys rm -f Point.output
)spool Point.output

```

```

)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Point
--R
--R Point(R: Ring) is a domain constructor
--R Abbreviation for Point is POINT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for POINT
--R
--R----- Operations -----
--R -?: % -> % if R has ABELGRP          concat : List(%) -> %
--R concat : (%,%) -> %                  concat : (R,%) -> %
--R concat : (%,R) -> %                  construct : List(R) -> %
--R convert : List(R) -> %              copy : % -> %
--R cross : (%,%) -> %                  delete : (%,Integer) -> %
--R dimension : % -> PositiveInteger    ?.? : (%,Integer) -> R
--R elt : (%,Integer,R) -> R            empty : () -> %
--R empty? : % -> Boolean               entries : % -> List(R)
--R eq? : (%,%) -> Boolean              extend : (%,List(R)) -> %
--R index? : (Integer,%) -> Boolean     indices : % -> List(Integer)
--R insert : (%,%,Integer) -> %        insert : (R,%,Integer) -> %
--R map : ((R,R) -> R),%,%) -> %      map : ((R -> R),%) -> %
--R new : (NonNegativeInteger,R) -> %  point : List(R) -> %
--R qelt : (%,Integer) -> R            reverse : % -> %
--R sample : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?*? : (%,R) -> % if R has MONOID
--R ?*? : (R,%) -> % if R has MONOID
--R ?*? : (Integer,%) -> % if R has ABELGRP
--R ?+? : (%,%) -> % if R has ABELSG
--R ?-? : (%,%) -> % if R has ABELGRP
--R ?<? : (%,%) -> Boolean if R has ORDSET
--R ?<=? : (%,%) -> Boolean if R has ORDSET
--R ?=? : (%,%) -> Boolean if R has SETCAT
--R ?>? : (%,%) -> Boolean if R has ORDSET
--R ?>=? : (%,%) -> Boolean if R has ORDSET
--R any? : ((R -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if R has SETCAT
--R convert : % -> InputForm if R has KONVERT(INFORM)
--R copyInto! : (%,%,Integer) -> % if $ has shallowlyMutable
--R count : (R,%) -> NonNegativeInteger if $ has finiteAggregate and R has SETCAT
--R count : ((R -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R delete : (%,UniversalSegment(Integer)) -> %
--R dot : (%,%) -> R if R has RING
--R ?.? : (%,UniversalSegment(Integer)) -> %
--R entry? : (R,%) -> Boolean if $ has finiteAggregate and R has SETCAT
--R eval : (%,List(R),List(R)) -> % if R has EVALAB(R) and R has SETCAT

```

```

--R eval : (% , R , R) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (% , Equation(R)) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (% , List(Equation(R))) -> % if R has EVALAB(R) and R has SETCAT
--R every? : ((R -> Boolean) , %) -> Boolean if $ has finiteAggregate
--R fill! : (% , R) -> % if $ has shallowlyMutable
--R find : ((R -> Boolean) , %) -> Union(R , "failed")
--R first : % -> R if Integer has ORDSET
--R hash : % -> SingleInteger if R has SETCAT
--R latex : % -> String if R has SETCAT
--R length : % -> R if R has RADCAT and R has RING
--R less? : (% , NonNegativeInteger) -> Boolean
--R magnitude : % -> R if R has RADCAT and R has RING
--R map! : ((R -> R) , %) -> % if $ has shallowlyMutable
--R max : (% , %) -> % if R has ORDSET
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (R , %) -> Boolean if $ has finiteAggregate and R has SETCAT
--R members : % -> List(R) if $ has finiteAggregate
--R merge : (% , %) -> % if R has ORDSET
--R merge : ((R , R) -> Boolean) , % , % -> %
--R min : (% , %) -> % if R has ORDSET
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (% , NonNegativeInteger) -> Boolean
--R outerProduct : (% , %) -> Matrix(R) if R has RING
--R parts : % -> List(R) if $ has finiteAggregate
--R position : (R , % , Integer) -> Integer if R has SETCAT
--R position : (R , %) -> Integer if R has SETCAT
--R position : ((R -> Boolean) , %) -> Integer
--R qsetelt! : (% , Integer , R) -> R if $ has shallowlyMutable
--R reduce : (((R , R) -> R) , %) -> R if $ has finiteAggregate
--R reduce : (((R , R) -> R) , % , R) -> R if $ has finiteAggregate
--R reduce : (((R , R) -> R) , % , R , R) -> R if $ has finiteAggregate and R has SETCAT
--R remove : ((R -> Boolean) , %) -> % if $ has finiteAggregate
--R remove : (R , %) -> % if $ has finiteAggregate and R has SETCAT
--R removeDuplicates : % -> % if $ has finiteAggregate and R has SETCAT
--R reverse! : % -> % if $ has shallowlyMutable
--R select : ((R -> Boolean) , %) -> % if $ has finiteAggregate
--R setelt : (% , UniversalSegment(Integer) , R) -> R if $ has shallowlyMutable
--R setelt : (% , Integer , R) -> R if $ has shallowlyMutable
--R size? : (% , NonNegativeInteger) -> Boolean
--R sort : % -> % if R has ORDSET
--R sort : (((R , R) -> Boolean) , %) -> %
--R sort! : % -> % if $ has shallowlyMutable and R has ORDSET
--R sort! : (((R , R) -> Boolean) , %) -> % if $ has shallowlyMutable
--R sorted? : % -> Boolean if R has ORDSET
--R sorted? : (((R , R) -> Boolean) , %) -> Boolean
--R swap! : (% , Integer , Integer) -> Void if $ has shallowlyMutable
--R zero : NonNegativeInteger -> % if R has ABELMON
--R ?~=?: (% , %) -> Boolean if R has SETCAT
--R
--E 1

```

```
)spool
)lisp (bye)
```

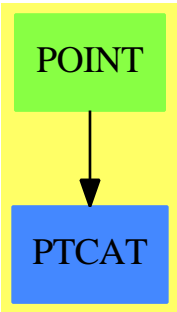
— Point.help —

```
=====
Point examples
=====
```

This domain implements points in coordinate space

See Also:
o)show Point

Point (POINT)



See

- ⇒ “SubSpaceComponentProperty” (COMPPROP) [20](#) on page [2624](#)
- ⇒ “SubSpace” (SUBSPACE) [20](#) on page [2614](#)

Exports:

any?	coerce	concat	construct	convert
copy	copyInto!	count	count	cross
delete	dimension	dot	elt	empty
empty?	entries	entry?	eq?	eval
every?	extend	fill!	find	first
hash	index?	indices	insert	insert
latex	length	less?	magnitude	map
map!	max	maxIndex	member?	members
merge	merge	min	minIndex	more?
new	outerProduct	parts	point	position
qelt	qsetelt!	reduce	remove	removeDuplicates
reverse	reverse!	sample	select	setelt
size?	sort	sort!	sorted?	swap!
zero	#?	?*?	?+?	?-?
?<?	?<=?	?=?	?>?	?>=?
?..?	?~=?	-?	?..?	

— domain POINT Point —

```

)abbrev domain POINT Point
++ Author: Mark Botch
++ Description:
++ This domain implements points in coordinate space

Point(R:Ring) : Exports == Implementation where
-- Domains for points, subspaces and properties of components in
-- a subspace

Exports ==> PointCategory(R)

Implementation ==> Vector (R) add
  PI ==> PositiveInteger

point(l:List R):% ==
  pt := new(#l,R)
  for x in l for i in minIndex(pt).. repeat
    pt.i := x
  pt
dimension p == (# p)::PI -- Vector returns NonNegativeInteger...?
convert(l:List R):% == point(l)
cross(p0, p1) ==
  #p0 ^=3 or #p1^=3 => error "Arguments to cross must be three dimensional"
  point [p0.2 * p1.3 - p1.2 * p0.3, _
    p1.1 * p0.3 - p0.1 * p1.3, _
    p0.1 * p1.2 - p1.1 * p0.2]
extend(p,l) == concat(p,point l)

```

— POINT.dotabb —

```
"POINT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=POINT"]
"PTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PTCAT"]
"POINT" -> "PTCAT"
```

—————

domain POLY Polynomial

— Polynomial.input —

```
)set break resume
)sys rm -f Polynomial.output
)spool Polynomial.output
)set message test on
)set message auto off
--S 1 of 46
x + 1
--R
--R
--R (1) x + 1
--R
--R                                          Type: Polynomial(Integer)
--E 1

--S 2 of 46
z - 2.3
--R
--R
--R (2) z - 2.3
--R
--R                                          Type: Polynomial(Float)
--E 2

--S 3 of 46
y**2 - z + 3/4
--R
--R
--R
--R          2 3
--R (3) - z + y + -
--R          4
--R
--R                                          Type: Polynomial(Fraction(Integer))
--E 3

--S 4 of 46
y **2 + x*y + y
```



```

--R
--R
--R      2
--R (4)  y  + (x + 1)y
--R
--R                                          Type: Polynomial(Integer)
--E 4

```

```

--S 5 of 46
% :: DMP([y,x],INT)
--R
--R
--R      2
--R (5)  y  + y x + y
--R
--R                                          Type: DistributedMultivariatePolynomial([y,x],Integer)
--E 5

```

```

--S 6 of 46
p := (y-1)**2 * x * z
--R
--R
--R      2
--R (6)  (x y  - 2x y + x)z
--R
--R                                          Type: Polynomial(Integer)
--E 6

```

```

--S 7 of 46
q := (y-1) * x * (z+5)
--R
--R
--R (7)  (x y - x)z + 5x y - 5x
--R
--R                                          Type: Polynomial(Integer)
--E 7

```

```

--S 8 of 46
factor(q)
--R
--R
--R (8)  x(y - 1)(z + 5)
--R
--R                                          Type: Factored(Polynomial(Integer))
--E 8

```

```

--S 9 of 46
p - q**2
--R
--R
--R (9)
--R      2 2      2      2 2      2      2      2      2
--R (- x y  + 2x y - x )z  + ((- 10x  + x)y  + (20x  - 2x)y - 10x  + x)z
--R +
--R      2 2      2      2

```

```
--R      - 25x y  + 50x y - 25x
```

```
--R
```

```
Type: Polynomial(Integer)
```

```
--E 9
```

```
--S 10 of 46
```

```
gcd(p,q)
```

```
--R
```

```
--R
```

```
--R      (10)  x y - x
```

```
--R
```

```
Type: Polynomial(Integer)
```

```
--E 10
```

```
--S 11 of 46
```

```
factor %
```

```
--R
```

```
--R
```

```
--R      (11)  x(y - 1)
```

```
--R
```

```
Type: Factored(Polynomial(Integer))
```

```
--E 11
```

```
--S 12 of 46
```

```
lcm(p,q)
```

```
--R
```

```
--R
```

```
--R      2      2      2
--R      (12)  (x y  - 2x y + x)z  + (5x y  - 10x y + 5x)z
```

```
--R
```

```
Type: Polynomial(Integer)
```

```
--E 12
```

```
--S 13 of 46
```

```
content p
```

```
--R
```

```
--R
```

```
--R      (13)  1
```

```
--R
```

```
Type: PositiveInteger
```

```
--E 13
```

```
--S 14 of 46
```

```
resultant(p,q,z)
```

```
--R
```

```
--R
```

```
--R      2 3      2 2      2      2
--R      (14)  5x y  - 15x y  + 15x y - 5x
```

```
--R
```

```
Type: Polynomial(Integer)
```

```
--E 14
```

```
--S 15 of 46
```

```
resultant(p,q,x)
```

```
--R
```

```
--R
```

[illegible]

--E 21

--S 22 of 46

degree(p,y)

--R

--R

--R (22) 2

--R

Type: PositiveInteger

--E 22

--S 23 of 46

degree(p,z)

--R

--R

--R (23) 1

--R

Type: PositiveInteger

--E 23

--S 24 of 46

degree(p,[x,y,z])

--R

--R

--R (24) [1,2,1]

--R

Type: List(NonNegativeInteger)

--E 24

--S 25 of 46

minimumDegree(p,z)

--R

--R

--R (25) 1

--R

Type: PositiveInteger

--E 25

--S 26 of 46

totalDegree p

--R

--R

--R (26) 4

--R

Type: PositiveInteger

--E 26

--S 27 of 46

leadingMonomial p

--R

--R

--R 2

--R (27) x y z

--R

Type: Polynomial(Integer)

--E 27

Type: Polynomial(Integer)

Type: Polynomial(Integer)

Type: PositiveInteger

Type: Polynomial(Integer)

Type: Polynomial(Integer)

Type: Polynomial(Integer)

--E 33

--S 34 of 46

eval(p,x,y^2 - 1)

--R

--R

--R
$$(y^4 - 2y^3 + 2y - 1)z$$

--R (34)

--R

Type: Polynomial(Integer)

--E 34

--S 35 of 46

D(p,x)

--R

--R

--R
$$(y^2 - 2y + 1)z$$

--R

Type: Polynomial(Integer)

--E 35

--S 36 of 46

D(p,y)

--R

--R

--R (36) $(2xy - 2x)z$

--R

Type: Polynomial(Integer)

--E 36

--S 37 of 46

D(p,z)

--R

--R

--R
$$x^2y^2 - 2xy + x$$

--R

Type: Polynomial(Integer)

--E 37

--S 38 of 46

integrate(p,y)

--R

--R

--R
$$(-\frac{1}{3}x^3y^3 - x^3y^2 + x^2y^2)z$$

--R

Type: Polynomial(Fraction(Integer))

--E 38

--S 39 of 46

qr := monicDivide(p,x+1,x)

--R

```

--R
--R      2      2
--R      (39) [quotient= (y  - 2y + 1)z,remainder= (- y  + 2y - 1)z]
--R      Type: Record(quotient: Polynomial(Integer),remainder: Polynomial(Integer))
--E 39

```

```

--S 40 of 46
qr.remainder
--R
--R
--R      2
--R      (40) (- y  + 2y - 1)z
--R
--R                                          Type: Polynomial(Integer)
--E 40

```

```

--S 41 of 46
p - ((x+1) * qr.quotient + qr.remainder)
--R
--R
--R      (41) 0
--R
--R                                          Type: Polynomial(Integer)
--E 41

```

```

--S 42 of 46
p/q
--R
--R
--R      (y - 1)z
--R      (42) -----
--R           z + 5
--R
--R                                          Type: Fraction(Polynomial(Integer))
--E 42

```

```

--S 43 of 46
(2/3) * x**2 - y + 4/5
--R
--R
--R      2  2  4
--R      (43) - y + - x + -
--R           3      5
--R
--R                                          Type: Polynomial(Fraction(Integer))
--E 43

```

```

--S 44 of 46
% :: FRAC POLY INT
--R
--R
--R      2
--R      - 15y + 10x  + 12
--R      (44) -----

```

Polynomial examples

Type: Polynomial Float

And here we have a polynomial in two variables with coefficients which have type Fraction Integer.

$$y^2 - z + \frac{3}{4} - z^2 + y^3 - \frac{1}{4}$$

Type: Polynomial Fraction Integer

The representation of objects of domains created by Polynomial is that of recursive univariate polynomials. The term univariate means "one variable". The term multivariate means "possibly more than one variable".

This recursive structure is sometimes obvious from the display of a polynomial.

$$y^2 + x^2y + y^2 + (x+1)y$$

Type: Polynomial Integer

In this example, you see that the polynomial is stored as a polynomial in y with coefficients that are polynomials in x with integer coefficients. In fact, you really don't need to worry about the representation unless you are working on an advanced application where it is critical. The polynomial types created from DistributedMultivariatePolynomial and NewDistributedMultivariatePolynomial are stored and displayed in a non-recursive manner.

You see a "flat" display of the above polynomial by converting to one of those types.

$$\% :: \text{DMP}([y,x], \text{INT})$$

$$y^2 + yx + y^2$$

Type: DistributedMultivariatePolynomial([y,x], Integer)

We will demonstrate many of the polynomial facilities by using two polynomials with integer coefficients.

By default, the interpreter expands polynomial expressions, even if they are written in a factored format.

$$p := (y-1)^2 * x * z$$

$$(x y^2 - 2x y + x)z$$

Type: Polynomial Integer

See Factored to see how to create objects in factored form directly.

$$q := (y-1) * x * (z+5)$$

```
(x y - x)z + 5x y - 5x
      Type: Polynomial Integer
```

The fully factored form can be recovered by using factor.

```
factor(q)
x(y - 1)(z + 5)
      Type: Factored Polynomial Integer
```

This is the same name used for the operation to factor integers. Such reuse of names is called overloading and makes it much easier to think of solving problems in general ways. Axiom facilities for factoring polynomials created with Polynomial are currently restricted to the integer and rational number coefficient cases.

The standard arithmetic operations are available for polynomials.

```
p - q**2
      2 2      2      2 2      2      2      2      2
      (- x y + 2x y - x )z + ((- 10x + x)y + (20x - 2x)y - 10x + x)z
+
      2 2      2      2
      - 25x y + 50x y - 25x
      Type: Polynomial Integer
```

The operation gcd is used to compute the greatest common divisor of two polynomials.

```
gcd(p,q)
x y - x
      Type: Polynomial Integer
```

In the case of p and q, the gcd is obvious from their definitions. We factor the gcd to show this relationship better.

```
factor %
x(y - 1)
      Type: Factored Polynomial Integer
```

The least common multiple is computed by using lcm.

```
lcm(p,q)
      2      2      2
      (x y - 2x y + x)z + (5x y - 10x y + 5x)z
      Type: Polynomial Integer
```

Use content to compute the greatest common divisor of the coefficients of the polynomial.

```
content p
```

1

Type: PositiveInteger

Many of the operations on polynomials require you to specify a variable. For example, resultant requires you to give the variable in which the polynomials should be expressed.

This computes the resultant of the values of p and q, considering them as polynomials in the variable z. They do not share a root when thought of as polynomials in z.

```
resultant(p,q,z)
      2 3      2 2      2      2
      5x y  - 15x y  + 15x y  - 5x
                                     Type: Polynomial Integer
```

This value is 0 because as polynomials in x the polynomials have a common root.

```
resultant(p,q,x)
0
                                     Type: Polynomial Integer
```

The data type used for the variables created by Polynomial is Symbol. As mentioned above, the representation used by Polynomial is recursive and so there is a main variable for nonconstant polynomials.

The operation mainVariable returns this variable. The return type is actually a union of Symbol and "failed".

```
mainVariable p
z
                                     Type: Union(Symbol,...)
```

The latter branch of the union is be used if the polynomial has no variables, that is, is a constant.

```
mainVariable(1 :: POLY INT)
"failed"
                                     Type: Union("failed",...)
```

You can also use the predicate ground? to test whether a polynomial is in fact a member of its ground ring.

```
ground? p
false
                                     Type: Boolean
```

```
ground?(1 :: POLY INT)
true
```

Type: Boolean

The complete list of variables actually used in a particular polynomial is returned by `variables`. For constant polynomials, this list is empty.

```
variables p
[z,y,x]
```

Type: List Symbol

The degree operation returns the degree of a polynomial in a specific variable.

```
degree(p,x)
1
```

Type: PositiveInteger

```
degree(p,y)
2
```

Type: PositiveInteger

```
degree(p,z)
1
```

Type: PositiveInteger

If you give a list of variables for the second argument, a list of the degrees in those variables is returned.

```
degree(p,[x,y,z])
[1,2,1]
```

Type: List NonNegativeInteger

The minimum degree of a variable in a polynomial is computed using `minimumDegree`.

```
minimumDegree(p,z)
1
```

Type: PositiveInteger

The total degree of a polynomial is returned by `totalDegree`.

```
totalDegree p
4
```

Type: PositiveInteger

It is often convenient to think of a polynomial as a leading monomial plus the remaining terms.

```
leadingMonomial p
2
x y z
```

Type: Polynomial Integer

Derivatives are computed using the D operation.

```
D(p,x)
  2
  (y  - 2y + 1)z
      Type: Polynomial Integer
```

The first argument is the polynomial and the second is the variable.

```
D(p,y)
  (2x y - 2x)z
      Type: Polynomial Integer
```

Even if the polynomial has only one variable, you must specify it.

```
D(p,z)
  2
  x y  - 2x y + x
      Type: Polynomial Integer
```

Integration of polynomials is similar and the integrate operation is used.

Integration requires that the coefficients support division. Axiom converts polynomials over the integers to polynomials over the rational numbers before integrating them.

```
integrate(p,y)
  1   3   2
  (- x y  - x y  + x y)z
  3
      Type: Polynomial Fraction Integer
```

It is not possible, in general, to divide two polynomials. In our example using polynomials over the integers, the operation `monicDivide` divides a polynomial by a monic polynomial (that is, a polynomial with leading coefficient equal to 1). The result is a record of the quotient and remainder of the division.

You must specify the variable in which to express the polynomial.

```
qr := monicDivide(p,x+1,x)
      2               2
[quotient= (y  - 2y + 1)z,remainder= (- y  + 2y - 1)z]
      Type: Record(quotient: Polynomial Integer,remainder: Polynomial Integer)
```

The selectors of the components of the record are `quotient` and `remainder`. Issue this to extract the remainder.

```
qr.remainder
  2
```

$$(-y^2 + 2y - 1)z$$

Type: Polynomial Integer

Now that we can extract the components, we can demonstrate the relationship among them and the arguments to our original expression

```
qr := monicDivide(p,x+1,x).
```

```
p - ((x+1) * qr.quotient + qr.remainder)
0
```

Type: Polynomial Integer

If the / operator is used with polynomials, a fraction object is created. In this example, the result is an object of type Fraction Polynomial Integer.

$$\frac{p/q}{(y-1)z}$$

$$\frac{z+5}{z+5}$$

Type: Fraction Polynomial Integer

If you use rational numbers as polynomial coefficients, the resulting object is of type Polynomial Fraction Integer.

$$\frac{(2/3) * x^2 - y + 4/5}{-y + \frac{2}{3}x^2 + \frac{4}{5}}$$

Type: Polynomial Fraction Integer

This can be converted to a fraction of polynomials and back again, if required.

```
% :: FRAC POLY INT
      2
- 15y + 10x + 12
-----
      15
```

Type: Fraction Polynomial Integer

```
% :: POLY FRAC INT
      2 2 4
- y + - x + -
      3 5
```

Type: Polynomial Fraction Integer

To convert the coefficients to floating point, map the numeric operation on the coefficients of the polynomial.

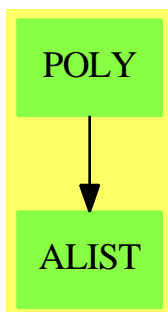
```
map(numeric,%)
```

```
- 1.0 y + 0.6666666666 6666666667 x + 0.8
Type: Polynomial Float
```

See Also:

- o)help Factored
- o)help UnivariatePolynomial
- o)help MultivariatePolynomial
- o)help DistributedMultivariatePolynomial
- o)help NewDistributedMultivariatePolynomial
- o)show Polynomial

Polynomial (POLY)



See

- ⇒ “MultivariatePolynomial” (MPOLY) [14](#) on page [1681](#)
- ⇒ “SparseMultivariatePolynomial” (SMP) [20](#) on page [2422](#)
- ⇒ “IndexedExponents” (INDE) [10](#) on page [1217](#)

Exports:

0	1	associates?
binomThmExpt	characteristic	charthRoot
coefficient	coefficients	coerce
conditionP	content	convert
D	degree	differentiate
discriminant	eval	exquo
factor	factorPolynomial	factorSquareFreePolynomial
gcd	gcdPolynomial	ground
ground?	hash	integrate
isExpt	isPlus	isTimes
latex	lcm	leadingCoefficient
leadingMonomial	mainVariable	map
mapExponents	max	min
minimumDegree	monicDivide	monomial
monomial?	monomials	multivariate
one?	numberOfMonomials	patternMatch
pomopo!	prime?	primitiveMonomials
primitivePart	recip	reducedSystem
reductum	resultant	retract
retractIfCan	sample	solveLinearPolynomialEquation
squareFree	squareFreePart	squareFreePolynomial
subtractIfCan	totalDegree	totalDegree
unit?	unitCanonical	unitNormal
univariate	variables	zero?
?*?	?**?	?+?
?-?	~?	?=?
?^?	?~=?	?/?
?<?	?<=?	?>?
?>=?		

— domain POLY Polynomial —

```

)abbrev domain POLY Polynomial
++ Author: Dave Barton, Barry Trager
++ Description:
++ This type is the basic representation of sparse recursive multivariate
++ polynomials whose variables are arbitrary symbols. The ordering
++ is alphabetic determined by the Symbol type.
++ The coefficient ring may be non commutative,
++ but the variables are assumed to commute.

Polynomial(R:Ring):
  PolynomialCategory(R, IndexedExponents Symbol, Symbol) with
  if R has Algebra Fraction Integer then
    integrate: (%, Symbol) -> %
    ++ integrate(p,x) computes the integral of \spad{p*dx}, i.e.
    ++ integrates the polynomial p with respect to the variable x.
  == SparseMultivariatePolynomial(R, Symbol) add

```

```

import UserDefinedPartialOrdering(Symbol)

coerce(p: %): OutputForm ==
  (r:= retractIfCan(p)@Union(R,"failed")) case R => r::R::OutputForm
a :=
  userOrdered?() => largest variables p
  mainVariable(p)::Symbol
  outputForm(univariate(p, a), a::OutputForm)

if R has Algebra Fraction Integer then
  integrate(p, x) == (integrate univariate(p, x)) (x::%)

-----

— POLY.dotabb —

"POLY" [color="#88FF44",href="bookvol10.3.pdf#nameddest=POLY"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"POLY" -> "ALIST"

-----

```

domain IDEAL PolynomialIdeals

```

— PolynomialIdeals.input —

)set break resume
)sys rm -f PolynomialIdeals.output
)spool PolynomialIdeals.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialIdeals
--R
--R PolynomialIdeals(F: Field,Expon: OrderedAbelianMonoidSup,VarSet: OrderedSet,DPoly: PolynomialCategory(F
--R Abbreviation for PolynomialIdeals is IDEAL
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for IDEAL
--R
--R----- Operations -----
--R ?? : (% , %) -> %
--R ?+? : (% , %) -> %

```

```

--R ?=? : (%,% ) -> Boolean
--R coerce : % -> OutputForm
--R element? : (DPoly,%) -> Boolean
--R groebner : % -> %
--R groebnerIdeal : List(DPoly) -> %
--R ideal : List(DPoly) -> %
--R inRadical? : (DPoly,%) -> Boolean
--R intersect : (%,% ) -> %
--R leadingIdeal : % -> %
--R quotient : (% ,DPoly) -> %
--R saturate : (% ,DPoly) -> %
--R zeroDim? : % -> Boolean
--R ?***? : (% ,NonNegativeInteger) -> %
--R backOldPos : Record(mval: Matrix(F),invnval: Matrix(F),genIdeal: %) -> %
--R dimension : (% ,List(VarSet)) -> Integer
--R generalPosition : (% ,List(VarSet)) -> Record(mval: Matrix(F),invnval: Matrix(F),genIdeal: %)
--R relationsIdeal : List(DPoly) -> SuchThat(List(Polynomial(F)),List(Equation(Polynomial(F))))
--R saturate : (% ,DPoly,List(VarSet)) -> %
--R zeroDim? : (% ,List(VarSet)) -> Boolean
--R
--E 1

```

```

)spool
)lisp (bye)

```

— PolynomialIdeals.help —

=====

PolynomialIdeals examples

=====

This domain represents polynomial ideals with coefficients in any field and supports the basic ideal operations, including intersection sum and quotient.

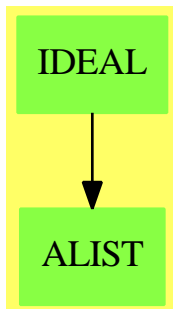
An ideal is represented by a list of polynomials (the generators of the ideal) and a boolean that is true if the generators are a Groebner basis.

The algorithms used are based on Groebner basis computations. The ordering is determined by the datatype of the input polynomials. Users may use refinements of total degree orderings.

See Also:

- o)show PolynomialIdeals

PolynomialIdeals (IDEAL)

**Exports:**

backOldPos	coerce	dimension	element?	generalPosition
generators	groebner	groebner?	groebnerIdeal	hash
ideal	in?	inRadical?	intersect	latex
leadingIdeal	one?	quotient	relationsIdeal	saturate
zero?	zeroDim?	?~=?	?**?	?*?
?+?	?=?			

— domain IDEAL PolynomialIdeals —

```

)abbrev domain IDEAL PolynomialIdeals
++ Author: P. Gianni
++ Date Created: summer 1986
++ Date Last Updated: September 1996
++ Description:
++ This domain represents polynomial ideals with coefficients in any
++ field and supports the basic ideal operations, including intersection
++ sum and quotient.
++ An ideal is represented by a list of polynomials (the generators of
++ the ideal) and a boolean that is true if the generators are a Groebner
++ basis.
++ The algorithms used are based on Groebner basis computations. The
++ ordering is determined by the datatype of the input polynomials.
++ Users may use refinements of total degree orderings.

```

```

PolynomialIdeals(F,Expon,VarSet,DPoly) : C == T
where
  F      : Field
  Expon  : OrderedAbelianMonoidSup
  VarSet : OrderedSet
  DPoly  : PolynomialCategory(F,Expon,VarSet)

  SUP    ==> SparseUnivariatePolynomial(DPoly)
  NNI    ==> NonNegativeInteger
  Z      ==> Integer

```

```

P      ==> Polynomial F
MF     ==> Matrix(F)
ST     ==> SuchThat(List P, List Equation P)

GenMPos ==> Record(mval:MF, invmval:MF, genIdeal:Ideal)
Ideal   ==> %

C == SetCategory with

"*"      : (Ideal,Ideal)      -> Ideal
++ I*J computes the product of the ideal I and J.
"("**"   : (Ideal,NNI)        -> Ideal
++ I**n computes the nth power of the ideal I.
"+"      : (Ideal,Ideal)      -> Ideal
++ I+J computes the ideal generated by the union of I and J.
one?     : Ideal              -> Boolean
++ one?(I) tests whether the ideal I is the unit ideal,
++ i.e. contains 1.
zero?    : Ideal              -> Boolean
++ zero?(I) tests whether the ideal I is the zero ideal
element? : (DPoly,Ideal)      -> Boolean
++ element?(f,I) tests whether the polynomial f belongs to
++ the ideal I.
in?      : (Ideal,Ideal)      -> Boolean
++ in?(I,J) tests if the ideal I is contained in the ideal J.
inRadical? : (DPoly,Ideal)    -> Boolean
++ inRadical?(f,I) tests if some power of the polynomial f
++ belongs to the ideal I.
zeroDim? : (Ideal,List VarSet) -> Boolean
++ zeroDim?(I,lvar) tests if the ideal I is zero dimensional, i.e.
++ all its associated primes are maximal,
++ in the ring \spad{F[lvar]}
zeroDim? : Ideal              -> Boolean
++ zeroDim?(I) tests if the ideal I is zero dimensional, i.e.
++ all its associated primes are maximal,
++ in the ring \spad{F[lvar]},
++ where lvar are the variables appearing in I
intersect : (Ideal,Ideal)      -> Ideal
++ intersect(I,J) computes the intersection of the ideals I and J.
intersect : List(Ideal)        -> Ideal
++ intersect(LI) computes the intersection of the list of ideals LI.
quotient  : (Ideal,Ideal)      -> Ideal
++ quotient(I,J) computes the quotient of the ideals I and J,
++ \spad{(I:J)}.
quotient  : (Ideal,DPoly)      -> Ideal
++ quotient(I,f) computes the quotient of the ideal I by the principal
++ ideal generated by the polynomial f, \spad{(I:(f))}.
groebner  : Ideal              -> Ideal
++ groebner(I) returns a set of generators of I that are a
++ Groebner basis for I.

```

```

generalPosition : (Ideal,List VarSet)      -> GenMPos
++ generalPosition(I,listvar) perform a random linear
++ transformation on the variables in listvar and returns
++ the transformed ideal along with the change of basis matrix.
backOldPos      :      GenMPos      -> Ideal
++ backOldPos(genPos) takes the result
++ produced by generalPosition from PolynomialIdeals
++ and performs the inverse transformation, returning the original ideal
++ \spad{backOldPos(generalPosition(I,listvar))} = I.
dimension       : (Ideal,List VarSet)      -> Z
++ dimension(I,lvar) gives the dimension of the ideal I,
++ in the ring \spad{F[lvar]}
dimension       :      Ideal      -> Z
++ dimension(I) gives the dimension of the ideal I.
++ in the ring \spad{F[lvar]}, where lvar are the variables
++ appearing in I
leadingIdeal    :      Ideal      -> Ideal
++ leadingIdeal(I) is the ideal generated by the
++ leading terms of the elements of the ideal I.
ideal          :      List DPoly      -> Ideal
++ ideal(polyList) constructs the ideal generated by the list
++ of polynomials polyList.
groebnerIdeal   :      List DPoly      -> Ideal
++ groebnerIdeal(polyList) constructs the ideal generated by the list
++ of polynomials polyList which are assumed to be a Groebner
++ basis.
++ Note: this operation avoids a Groebner basis computation.
groebner?       :      Ideal      -> Boolean
++ groebner?(I) tests if the generators of the ideal I are a
++ Groebner basis.
generators      :      Ideal      -> List DPoly
++ generators(I) returns a list of generators for the ideal I.
coerce          :      List DPoly      -> Ideal
++ coerce(polyList) converts the list of polynomials polyList
++ to an ideal.

saturate        : (Ideal,DPoly)      -> Ideal
++ saturate(I,f) is the saturation of the ideal I
++ with respect to the multiplicative
++ set generated by the polynomial f.
saturate        : (Ideal,DPoly,List VarSet) -> Ideal
++ saturate(I,f,lvar) is the saturation with respect to the prime
++ principal ideal which is generated by f in the polynomial ring
++ \spad{F[lvar]}.
if VarSet has ConvertibleTo Symbol then
  relationsIdeal :      List DPoly      -> ST
  ++ relationsIdeal(polyList) returns the ideal of relations among the
  ++ polynomials in polyList.

```

T == add

```

--- Representation ---
Rep := Record(idl:List DPoly,isGr:Boolean)

----- Local Functions -----

contractGrob : newIdeal      -> Ideal
npoly        : DPoly         -> newPoly
oldpoly      : newPoly       -> Union(DPoly,"failed")
leadterm     : (DPoly,VarSet) -> DPoly
choosel      : (DPoly,DPoly)  -> DPoly
isMonic?     : (DPoly,VarSet) -> Boolean
randomat     : List Z        -> Record(mM:MF,imM:MF)
monomDim     : (Ideal,List VarSet) -> NNI
variables    : Ideal         -> List VarSet
subset       : List VarSet    -> List List VarSet
makeleast    : (List VarSet,List VarSet) -> List VarSet

newExpon: OrderedAbelianMonoidSup
newExpon:= Product(NNI,Expon)
newPoly := PolynomialRing(F,newExpon)

import GaloisGroupFactorizer(SparseUnivariatePolynomial Z)
import GroebnerPackage(F,Expon,VarSet,DPoly)
import GroebnerPackage(F,newExpon,VarSet,newPoly)

newIdeal ==> List(newPoly)

npoly(f:DPoly) : newPoly ==
  f=0$DPoly => 0$newPoly
  monomial(leadingCoefficient f,makeprod(0,degree f))$newPoly +
    npoly(reductum f)

oldpoly(q:newPoly) : Union(DPoly,"failed") ==
  q=0$newPoly => 0$DPoly
  dq:newExpon:=degree q
  n:NNI:=selectfirst (dq)
  n^=0 => "failed"
  ((g:=oldpoly reductum q) case "failed") => "failed"
  monomial(leadingCoefficient q,selectsecond dq)$DPoly + (g::DPoly)

leadterm(f:DPoly,lvar:List VarSet) : DPoly ==
  empty?(lf:=variables f) or lf=lvar => f
  leadterm(leadingCoefficient univariate(f,lf.first),lvar)

choosel(f:DPoly,g:DPoly) : DPoly ==
  g=0 => f
  (f1:=f exquo g) case "failed" => f
  choosel(f1::DPoly,g)

```

```

contractGrob(I1:newIdeal) : Ideal ==
  J1:=List(newPoly):=groebner(I1)
  while (oldpoly J1.first) case "failed" repeat J1:=J1.rest
  [[(oldpoly f)::DPoly for f in J1],true]

makeleast(fullVars: List VarSet,leastVars:List VarSet) : List VarSet ==
  n:= # leastVars
  #fullVars < n => error "wrong vars"
  n=0 => fullVars
  append([vv for vv in fullVars | ^member?(vv,leastVars)],leastVars)

isMonic?(f:DPoly,x:VarSet) : Boolean ==
  ground? leadingCoefficient univariate(f,x)

subset(lv : List VarSet) : List List VarSet ==
  #lv =1 => [lv,empty()]
  v:=lv.1
  l1:=subset(rest lv)
  l1:=[concat(v,set) for set in l1]
  concat(l1,l1)

monomDim(listm:Ideal,lv:List VarSet) : NNI ==
  monvar: List List VarSet := []
  for f in generators listm repeat
    mvset := variables f
    #mvset > 1 => monvar:=concat(mvset,monvar)
    lv:=delete(lv,position(mvset.1,lv))
  empty? lv => 0
  lsubset : List List VarSet := sort((a,b)+->#a > #b ,subset(lv))
  for subs in lsubset repeat
    ldif:List VarSet:= lv
    for mvset in monvar while ldif ^=[] repeat
      ldif:=setDifference(mvset,subs)
      if ^(empty? ldif) then return #subs
  0

--      Exported Functions      ----

---- is I = J ? ----
(I:Ideal = J:Ideal) == in?(I,J) and in?(J,I)

---- check if f is in I ----
element?(f:DPoly,I:Ideal) : Boolean ==
  Id:=(groebner I).idl
  empty? Id => f = 0
  normalForm(f,Id) = 0

---- check if I is contained in J ----
in?(I:Ideal,J:Ideal):Boolean ==

```



```

J:= groebner J
empty?(I.idl) => true
"and"/[element?(f,J) for f in I.idl ]

---- groebner base for an Ideal ----
groebner(I:Ideal) : Ideal ==
I.isGr =>
  "or"/[^zero? f for f in I.idl] => I
  [empty(),true]
  [groebner I.idl ,true]

---- Intersection of two ideals ----
intersect(I:Ideal,J:Ideal) : Ideal ==
empty?(Id:=I.idl) => I
empty?(Jd:=J.idl) => J
tp:newPoly := monomial(1,makeprod(1,0$Expon))$newPoly
tp1:newPoly:= tp-1
contractGrob(concat([tp*npoly f for f in Id],
                    [tp1*npoly f for f in Jd]))

---- intersection for a list of ideals ----
intersect(lid:List(Ideal)) : Ideal == "intersect"/[l for l in lid]

---- quotient by an element ----
quotient(I:Ideal,f:DPoly) : Ideal ==
--[[ (g exquo f)::DPoly for g in (intersect(I,[f>::%)).idl ],true]
import GroebnerInternalPackage(F,Expon,VarSet,DPoly)
[minGbasis [(g exquo f)::DPoly
            for g in (intersect(I,[f>::%)).idl ],true]

---- quotient of two ideals ----
quotient(I:Ideal,J:Ideal) : Ideal ==
Jdl := J.idl
empty?(Jdl) => ideal [1]
[("intersect"/[quotient(I,f) for f in Jdl ]).idl ,true]

---- sum of two ideals ----
(I:Ideal + J:Ideal) : Ideal == [groebner(concat(I.idl ,J.idl )),true]

---- product of two ideals ----
(I:Ideal * J:Ideal):Ideal ==
[groebner([f*g for f in I.idl ] for g in J.idl )],true]

---- power of an ideal ----
(I:Ideal ** n:NNI) : Ideal ==
n=0 => [[1$DPoly],true]

```

```

(I * (I**(n-1):NNI))

---- saturation with respect to the multiplicative set f**n ----
saturate(I:Ideal,f:DPoly) : Ideal ==
  f=0 => error "f is zero"
  tp:newPoly := (monomial(1,makeprod(1,0$Expon))$newPoly * npoly f)-1
  contractGrob(concat(tp,[npoly g for g in I.idl ]))

---- saturation with respect to a prime principal ideal in lvar ---
saturate(I:Ideal,f:DPoly,lvar>List(VarSet)) : Ideal ==
  Id := I.idl
  fullVars := "setUnion"/[variables g for g in Id]
  newVars:=makeleast(fullVars,lvar)
  subVars := [monomial(1,vv,1) for vv in newVars]
  J>List DPoly:=groebner([eval(g,fullVars,subVars) for g in Id])
  ltJ:= [leadterm(g,lvar) for g in J]
  s:DPoly:=_*/[choosel(ltg,f) for ltg in ltJ]
  fullPol:= [monomial(1,vv,1) for vv in fullVars]
  [[eval(g,newVars,fullPol) for g in (saturate(J:%,s)).idl],true]

      ---- is the ideal zero dimensional? ----
      ---- in the ring F[lvar]? ----
zeroDim?(I:Ideal,lvar>List VarSet) : Boolean ==
  J:=(groebner I).idl
  empty? J => false
  J = [1] => false
  n:NNI := # lvar
  #J < n => false
  for f in J while ^empty?(lvar) repeat
    x:=(mainVariable f)::VarSet
    if isMonic?(f,x) then lvar:=delete(lvar,position(x,lvar))
  empty?(lvar)

      ---- is the ideal zero dimensional? ----
zeroDim?(I:Ideal):Boolean == zeroDim?(I,"setUnion"/[variables g for g in I.idl])

      ---- test if f is in the radical of I ----
inRadical?(f:DPoly,I:Ideal) : Boolean ==
  f=0$DPoly => true
  tp:newPoly :=(monomial(1,makeprod(1,0$Expon))$newPoly * npoly f)-1
  Id:=I.idl
  normalForm(1$newPoly,groebner concat(tp,[npoly g for g in Id])) = 0

      ---- dimension of an ideal ----
      ---- in the ring F[lvar] ----
dimension(I:Ideal,lvar>List VarSet) : Z ==
  I:=groebner I
  empty?(I.idl) => # lvar
  element?(1,I) => -1
  truelist:="setUnion"/[variables f for f in I.idl]

```

```

"or"/[~member?(vv,lvar) for vv in truelist] => error "wrong variables"
truelist:=setDifference(lvar,setDifference(lvar,truelist))
ed:Z:=#lvar - #truelist
leadid:=leadingIdeal(I)
n1:Z:=monomDim(leadid,truelist)::Z
ed+n1

dimension(I:Ideal) : Z == dimension(I,"setUnion"/[variables g for g in I.idl])

-- leading term ideal --
leadingIdeal(I : Ideal) : Ideal ==
  Idl:= (groebner I).idl
  [[(f-reductum f) for f in Idl],true]

      ---- ideal of relations among the fi ----
if VarSet has ConvertibleTo Symbol then

monompol(df:List NNI,lcf:F,lv:List VarSet) : P ==
  g:P:=lcf::P
  for dd in df for v in lv repeat
    g:= monomial(g,convert v,dd)
  g

relationsIdeal(listf : List DPoly): ST ==
  empty? listf => [empty(),empty()],$ST
  nf:=#listf
  lvint := "setUnion"/[variables g for g in listf]
  vl: List Symbol := [convert vv for vv in lvint]
  nvar:List Symbol:=[new() for i in 1..nf]
  VarSet1:=OrderedVariableList(concat(vl,nvar))
  lv1:=[variable(vv)$VarSet1::VarSet1 for vv in nvar]
  DirP:=DirectProduct(nf,NNI)
  nExponent:=Product(Expon,DirP)
  nPoly := PolynomialRing(F,nExponent)
  gp:=GroebnerPackage(F,nExponent,VarSet1,nPoly)
  lf:List nPoly :=[]
  lp:List P:=[]
  for f in listf for i in 1.. repeat
    vec2:Vector(NNI):=new(nf,0$NNI)
    vec2.i:=1
    g:nPoly:=0$nPoly
    pol:=0$P
    while f~=0 repeat
      df:=degree(f-reductum f,lvint)
      lcf:=leadingCoefficient f
      pol:=pol+monompol(df,lcf,lvint)
      g:=g+monomial(lcf,makeprod(degree f,0))$nPoly
      f:=reductum f
    lp:=concat(pol,lp)
  lf:=concat(monomial(1,makeprod(0,directProduct vec2))-g,lf)

```

```

npol:List P :=[v::P for v in nvar]
leq : List Equation P :=
  [p = pol for p in npol for pol in reverse lp ]
lf:=(groebner lf)$gp
while lf^=[] repeat
  q:=lf.first
  dq:nExponent:=degree q
  n:=selectfirst (dq)
  if n=0 then leave "done"
  lf:=lf.rest
solsn:List P:=[]
for q in lf repeat
  g:Polynomial F :=0
  while q^=0 repeat
    dq:=degree q
    lcq:=leadingCoefficient q
    q:=reductum q
    vdq:=(selectsecond dq):Vector NNI
    g:=g+ lcq*
      _*/[p**vdq.j for p in npol for j in 1..]
  sols:=concat(g,solsn)
[solsn,leq]$ST

coerce(Id:List DPoly) : Ideal == [Id,false]

coerce(I:Ideal) : OutputForm ==
  Idl := I.idl
  empty? Idl => [0$DPoly] :: OutputForm
  Idl :: OutputForm

ideal(Id:List DPoly) :Ideal == [[f for f in Id|f^=0],false]

groebnerIdeal(Id:List DPoly) : Ideal == [Id,true]

generators(I:Ideal) : List DPoly == I.idl

groebner?(I:Ideal) : Boolean == I.isGr

one?(I:Ideal) : Boolean == element?(1, I)

zero?(I:Ideal) : Boolean == empty? (groebner I).idl

```

— IDEAL.dotabb —

"IDEAL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IDEAL"]
 "ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]

"IDEAL" -> "ALIST"

domain PR PolynomialRing

— PolynomialRing.input —

```

)set break resume
)sys rm -f PolynomialRing.output
)spool PolynomialRing.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialRing
--R
--R PolynomialRing(R: Ring,E: OrderedAbelianMonoid) is a domain constructor
--R Abbreviation for PolynomialRing is PR
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PR
--R
--R----- Operations -----
--R ?? : (R,%) -> %           ?? : (%,R) -> %
--R ?? : (%,%) -> %           ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %           ?-? : (%,%) -> %
--R -? : % -> %               ?=? : (%,%) -> Boolean
--R 1 : () -> %               0 : () -> %
--R ?? : (%,PositiveInteger) -> %   coefficient : (%,E) -> R
--R coefficients : % -> List(R)     coerce : R -> %
--R coerce : Integer -> %           coerce : % -> OutputForm
--R degree : % -> E                 ground : % -> R
--R ground? : % -> Boolean          hash : % -> SingleInteger
--R latex : % -> String             leadingCoefficient : % -> R
--R leadingMonomial : % -> %         map : ((R -> R),%) -> %
--R mapExponents : ((E -> E),%) -> %   minimumDegree : % -> E
--R monomial : (R,E) -> %           monomial? : % -> Boolean
--R one? : % -> Boolean              pomopo! : (%,R,E,%) -> %
--R recip : % -> Union(%, "failed")   reductum : % -> %
--R retract : % -> R                 sample : () -> %
--R zero? : % -> Boolean              ~=? : (%,%) -> Boolean
--R ?? : (%,Fraction(Integer)) -> % if R has ALGEBRA(FRAC(INT))
--R ?? : (Fraction(Integer),%) -> % if R has ALGEBRA(FRAC(INT))
--R ?? : (NonNegativeInteger,%) -> %

```

```

--R ??? : (% , NonNegativeInteger) -> %
--R ?/? : (% , R) -> % if R has FIELD
--R ?? : (% , NonNegativeInteger) -> %
--R associates? : (% , %) -> Boolean if R has INTDOM
--R binomThmExpt : (% , %, NonNegativeInteger) -> % if R has COMRING
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if R has CHARNZ
--R coerce : Fraction(Integer) -> % if R has ALGEBRA(FRAC(INT)) or R has RETRACT(FRAC(INT))
--R coerce : % -> % if R has INTDOM
--R content : % -> R if R has GCDDOM
--R exquo : (% , R) -> Union(%, "failed") if R has INTDOM
--R exquo : (% , %) -> Union(%, "failed") if R has INTDOM
--R fmecg : (% , E , R , %) -> % if E has CABMON and R has INTDOM
--R numberOfMonomials : % -> NonNegativeInteger
--R primitivePart : % -> % if R has GCDDOM
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retract : % -> Integer if R has RETRACT(INT)
--R retractIfCan : % -> Union(R , "failed")
--R retractIfCan : % -> Union(Fraction(Integer) , "failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(Integer , "failed") if R has RETRACT(INT)
--R subtractIfCan : (% , %) -> Union(%, "failed")
--R unit? : % -> Boolean if R has INTDOM
--R unitCanonical : % -> % if R has INTDOM
--R unitNormal : % -> Record(unit: %, canonical: %, associate: %) if R has INTDOM
--R
--E 1

```

```

)spool
)lisp (bye)

```

— PolynomialRing.help —

=====

PolynomialRing examples

=====

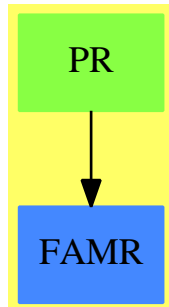
This domain represents generalized polynomials with coefficients (from a not necessarily commutative ring), and terms indexed by their exponents (from an arbitrary ordered abelian monoid).

This type is used, for example, by the DistributedMultivariatePolynomial domain where the exponent domain is a direct product of non negative integers.

See Also:

o)show PolynomialRing

PolynomialRing (PR)



See

⇒ “FreeModule” (FM) [7](#) on page [1013](#)

⇒ “SparseUnivariatePolynomial” (SUP) [20](#) on page [2466](#)

⇒ “UnivariatePolynomial” (UP) [22](#) on page [2827](#)

Exports:

0	1	associates?	binomThmExpt	characteristic
charthRoot	coerce	coefficient	coefficients	content
degree	exquo	exquo	fmecg	ground
ground?	hash	latex	leadingCoefficient	leadingMonomial
map	mapExponents	minimumDegree	monomial	monomial?
numberOfMonomials	one?	pomopo!	primitivePart	recip
reductum	retract	retractIfCan	sample	subtractIfCan
unit?	unitCanonical	unitNormal	zero?	?*?
?**?	?+?	?-?	-?	?=?
?^?	?~=?	?/?		

— domain PR PolynomialRing —

```

)abbrev domain PR PolynomialRing
++ Author: Dave Barton, James Davenport, Barry Trager
++ Date Created:
++ Date Last Updated: 14.08.2000. Improved exponentiation [MMM/TTT]
++ Description:
++ This domain represents generalized polynomials with coefficients
++ (from a not necessarily commutative ring), and terms
++ indexed by their exponents (from an arbitrary ordered abelian monoid).
++ This type is used, for example,
++ by the \spadtype{DistributedMultivariatePolynomial} domain where
++ the exponent domain is a direct product of non negative integers.

```

```

PolynomialRing(R:Ring,E:OrderedAbelianMonoid): T == C
where
  T == FiniteAbelianMonoidRing(R,E) with
    --assertions

```

```

    if R has IntegralDomain and E has CancellationAbelianMonoid then
      fmecg: (%E,R,%) -> %
      ++ fmecg(p1,e,r,p2) finds x : p1 - r * x**e * p2
    if R has canonicalUnitNormal then canonicalUnitNormal
      ++ canonicalUnitNormal guarantees that the function
      ++ unitCanonical returns the same representative for all
      ++ associates of any particular element.

C == FreeModule(R,E) add
--representations
  Term:= Record(k:E,c:R)
  Rep:= List Term

--declarations
  x,y,p,p1,p2: %
  n: Integer
  nn: NonNegativeInteger
  np: PositiveInteger
  e: E
  r: R
--local operations
  1 == [[0$E,1$R]]
  characteristic == characteristic$R
  numberOfMonomials x == (# x)$Rep
  degree p == if null p then 0 else p.first.k
  minimumDegree p == if null p then 0 else (last p).k
  leadingCoefficient p == if null p then 0$R else p.first.c
  leadingMonomial p == if null p then 0 else [p.first]
  reductum p == if null p then p else p.rest
  retractIfCan(p:%):Union(R,"failed") ==
    null p => 0$R
    not null p.rest => "failed"
    zero?(p.first.k) => p.first.c
    "failed"
  coefficient(p,e) ==
    for tm in p repeat
      tm.k=e => return tm.c
      tm.k < e => return 0$R
    0$R
  recip(p) ==
    null p => "failed"
    p.first.k > 0$E => "failed"
    (u:=recip(p.first.c)) case "failed" => "failed"
    (u::R)::%

  coerce(r) == if zero? r then 0$% else [[0$E,r]]
  coerce(n) == (n::R)::%

  ground?(p): Boolean == empty? p or (empty? rest p and zero? degree p)

```



```

qsetrest!: (Rep, Rep) -> Rep
qsetrest!(l: Rep, e: Rep): Rep == RPLACD(l, e)$Lisp

times!: (R, %) -> %
times: (R, E, %) -> %

entireRing? := R has EntireRing

times!(r: R, x: %): % ==
  res, endcell, newend, xx: Rep
  if entireRing? then
    for tx in x repeat tx.c := r*tx.c
  else
    xx := x
    res := empty()
    while not empty? xx repeat
      tx := first xx
      tx.c := r * tx.c
      if zero? tx.c then
        xx := rest xx
      else
        newend := xx
        xx := rest xx
        if empty? res then
          res := newend
          endcell := res
        else
          qsetrest!(endcell, newend)
          endcell := newend
    res;

--- term * polynomial
termTimes: (R, E, Term) -> Term
termTimes(r: R, e: E, tx:Term): Term == [e+tx.k, r*tx.c]
times(tco: R, tex: E, rx: %): % ==
  if entireRing? then
    map(x1+>termTimes(tco, tex, x1), rx::Rep)
  else
    [[tex + tx.k, r] for tx in rx::Rep | not zero? (r := tco * tx.c)]

-- local addm!
addm!: (Rep, R, E, Rep) -> Rep
-- p1 + coef*x^E * p2
-- 'spare' (commented out) is for storage efficiency (not so good for
-- performance though.
addm!(p1:Rep, coef:R, exp: E, p2:Rep): Rep ==
  --local res, newend, last: Rep

```

```

res, newcell, endcell: Rep
spare: List Rep
res      := empty()
endcell := empty()
--spare  := empty()
while not empty? p1 and not empty? p2 repeat
  tx := first p1
  ty := first p2
  exy := exp + ty.k
  newcell := empty();
  if tx.k = exy then
    newcoef := tx.c + coef * ty.c
    if not zero? newcoef then
      tx.c := newcoef
      newcell := p1
    --else
    --      spare := cons(p1, spare)
    p1 := rest p1
    p2 := rest p2
  else if tx.k > exy then
    newcell := p1
    p1 := rest p1
  else
    newcoef := coef * ty.c
    if not entireRing? and zero? newcoef then
      newcell := empty()
    --else if empty? spare then
    --      ttt := [exy, newcoef]
    --      newcell := cons(ttt, empty())
    --else
    --      newcell := first spare
    --      spare := rest spare
    --      ttt := first newcell
    --      ttt.k := exy
    --      ttt.c := newcoef
    else
      ttt := [exy, newcoef]
      newcell := cons(ttt, empty())
    p2 := rest p2
  if not empty? newcell then
    if empty? res then
      res := newcell
      endcell := res
    else
      qsetrest!(endcell, newcell)
      endcell := newcell
  if not empty? p1 then -- then end is const * p1
    newcell := p1
  else -- then end is (coef, exp) * p2
    newcell := times(coef, exp, p2)

```

```

        empty? res => newcell
        qsetrest!(endcell, newcell)
        res
pomopo! (p1, r, e, p2) == addm!(p1, r, e, p2)
p1 * p2 ==
    xx := p1::Rep
    empty? xx => p1
    yy := p2::Rep
    empty? yy => p2
    zero? first(xx).k => first(xx).c * p2
    zero? first(yy).k => p1 * first(yy).c
    --if #xx > #yy then
    --    (xx, yy) := (yy, xx)
    --    (p1, p2) := (p2, p1)
    xx := reverse xx
    res : Rep := empty()
    for tx in xx repeat res:=addm!(res,tx.c,tx.k,yy)
    res

-- if R has EntireRing then
--     p1 * p2 ==
--         null p1 => 0
--         null p2 => 0
--         zero?(p1.first.k) => p1.first.c * p2
--         one? p2 => p1
--         +/[[t1.k+t2.k,t1.c*t2.c]$Term for t2 in p2]
--             for t1 in reverse(p1)]
--             -- This 'reverse' is an efficiency improvement:
--             -- reduces both time and space [Abbott/Bradford/Davenport]
--     else
--         p1 * p2 ==
--             null p1 => 0
--             null p2 => 0
--             zero?(p1.first.k) => p1.first.c * p2
--             one? p2 => p1
--             +/[[t1.k+t2.k,r]$Term for t2 in p2 | (r:=t1.c*t2.c) ^= 0]
--                 for t1 in reverse(p1)]
--                 -- This 'reverse' is an efficiency improvement:
--                 -- reduces both time and space [Abbott/Bradford/Davenport]
-- if R has CommutativeRing then
--     p ** np == p ** (np pretend NonNegativeInteger)
--     p ^ np == p ** (np pretend NonNegativeInteger)
--     p ^ nn == p ** nn

--     p ** nn ==
--         null p => 0
--         zero? nn => 1
--         one? nn => p
--         (nn = 1) => p

```

```

empty? p.rest =>
  zero?(cc:=p.first.c ** nn) => 0
  [[nn * p.first.k, cc]]
binomThmExpt([p.first], p.rest, nn)

if R has Field then
  unitNormal(p) ==
    null p or (lcf:R:=p.first.c) = 1 => [1,p,1]
    a := inv lcf
    [lcf::%, [[p.first.k,1],:(a * p.rest)], a::%]
  unitCanonical(p) ==
    null p or (lcf:R:=p.first.c) = 1 => p
    a := inv lcf
    [[p.first.k,1],:(a * p.rest)]
else if R has IntegralDomain then
  unitNormal(p) ==
    null p or p.first.c = 1 => [1,p,1]
    (u,cf,a):=unitNormal(p.first.c)
    [u::%, [[p.first.k,cf],:(a * p.rest)], a::%]
  unitCanonical(p) ==
    null p or p.first.c = 1 => p
    (u,cf,a):=unitNormal(p.first.c)
    [[p.first.k,cf],:(a * p.rest)]
if R has IntegralDomain then
  associates?(p1,p2) ==
    null p1 => null p2
    null p2 => false
    p1.first.k = p2.first.k and
    associates?(p1.first.c,p2.first.c) and
    ((p2.first.c exquo p1.first.c)::R * p1.rest = p2.rest)
p exquo r ==
  [(if (a:= tm.c exquo r) case "failed"
    then return "failed" else [tm.k,a])
    for tm in p] :: Union(%, "failed")
if E has CancellationAbelianMonoid then
  fmecg(p1:%,e:E,r:R,p2:%)::% == -- p1 - r * X**e * p2
  rout:%= []
  r:= - r
  for tm in p2 repeat
    e2:= e + tm.k
    c2:= r * tm.c
    c2 = 0 => "next term"
    while not null p1 and p1.first.k > e2 repeat
      (rout:=[p1.first,:rout]; p1:=p1.rest) --use PUSH and POP?
      null p1 or p1.first.k < e2 => rout:=[[e2,c2],:rout]
      if (u:=p1.first.c + c2) ^= 0 then rout:=[[e2, u],:rout]
      p1:=p1.rest
  NRECONC(rout,p1)$Lisp
if R has approximate then
  p1 exquo p2 ==

```

```

null p2 => error "Division by 0"
p2 = 1 => p1
p1=p2 => 1
--(p1.lastElt.c exquo p2.lastElt.c) case "failed" => "failed"
rout:= []@List(Term)
while not null p1 repeat
  (a:= p1.first.c exquo p2.first.c)
  a case "failed" => return "failed"
  ee:= subtractIfCan(p1.first.k, p2.first.k)
  ee case "failed" => return "failed"
  p1:= fmecg(p1.rest, ee, a, p2.rest)
  rout:= [[ee,a], :rout]
null p1 => reverse(rout)::%    -- nreverse?
"failed"
else -- R not approximate
p1 exquo p2 ==
  null p2 => error "Division by 0"
  p2 = 1 => p1
  --(p1.lastElt.c exquo p2.lastElt.c) case "failed" => "failed"
  rout:= []@List(Term)
  while not null p1 repeat
    (a:= p1.first.c exquo p2.first.c)
    a case "failed" => return "failed"
    ee:= subtractIfCan(p1.first.k, p2.first.k)
    ee case "failed" => return "failed"
    p1:= fmecg(p1.rest, ee, a, p2.rest)
    rout:= [[ee,a], :rout]
  null p1 => reverse(rout)::%    -- nreverse?
  "failed"
if R has Field then
  x/r == inv(r)*x

```

— PR.dotabb —

```

"PR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PR"]
"FAMR" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAMR"]
"PR" -> "FAMR"

```

domain PI PositiveInteger

— PositiveInteger.input —

```

)set break resume
)sys rm -f PositiveInteger.output
)spool PositiveInteger.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PositiveInteger
--R PositiveInteger is a domain constructor
--R Abbreviation for PositiveInteger is PI
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PI
--R
--R----- Operations -----
--R ?? : (%,%) -> %           ?? : (PositiveInteger,%) -> %
--R ??? : (%,PositiveInteger) -> %   ?+? : (%,%) -> %
--R ?<? : (%,%) -> Boolean       ?<=? : (%,%) -> Boolean
--R ?=? : (%,%) -> Boolean       ?>? : (%,%) -> Boolean
--R ?>=? : (%,%) -> Boolean     1 : () -> %
--R ?^? : (%,PositiveInteger) -> %   coerce : % -> OutputForm
--R gcd : (%,%) -> %             hash : % -> SingleInteger
--R latex : % -> String         max : (%,%) -> %
--R min : (%,%) -> %           one? : % -> Boolean
--R recip : % -> Union(%, "failed")  sample : () -> %
--R ?^=? : (%,%) -> Boolean
--R ??? : (%,NonNegativeInteger) -> %
--R ?^? : (%,NonNegativeInteger) -> %
--R
--E 1

)spool
)lisp (bye)

```

— PositiveInteger.help —

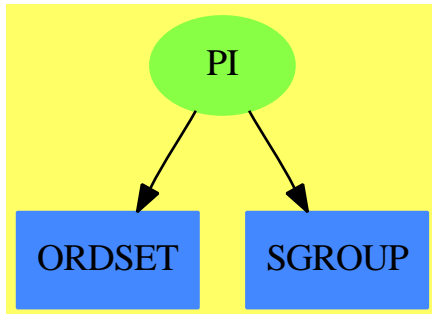
PositiveInteger examples

PositiveInteger provides functions for positive integers.

See Also:

o)show PositiveInteger

PositiveInteger (PI)



See

- ⇒ “Integer” (INT) [10](#) on page [1361](#)
- ⇒ “NonNegativeInteger” (NNI) [15](#) on page [1739](#)
- ⇒ “RomanNumeral” (ROMAN) [19](#) on page [2326](#)

Exports:

1	coerce	gcd	hash	latex
max	min	one?	recip	sample
?^?	?~=?	?**?	?*?	?+?
?<?	?<=?	?=?	?>?	?>=?

— domain **PI** **PositiveInteger** —

```

)abbrev domain PI PositiveInteger
++ Author: Mark Botch
++ Description:
++ \spadtype{PositiveInteger} provides functions for positive integers.

PositiveInteger: Join(AbelianSemiGroup,OrderedSet,Monoid) with
  gcd: (%,% ) -> %
    ++ gcd(a,b) computes the greatest common divisor of two
    ++ positive integers \spad{a} and b.
  commutative("*")
    ++ commutative("*") means multiplication is commutative : x*y = y*x
== SubDomain(NonNegativeInteger,#1 > 0) add
  x:%
  y:%
  
```

— **PI.dotabb** —

```

"PI" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PI",shape=ellipse]
"ORDSET" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ORDSET"]
  
```

```
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"PI" -> "ORDSET"
"PI" -> "SGROUP"
```

domain PF PrimeField

— PrimeField.input —

```
)set break resume
)sys rm -f PrimeField.output
)spool PrimeField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PrimeField
--R PrimeField(p: PositiveInteger) is a domain constructor
--R Abbreviation for PrimeField is PF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PF
--R
--R----- Operations -----
--R ?? : (Fraction(Integer),%) -> %      ?? : (%,Fraction(Integer)) -> %
--R ?? : (%,%) -> %                      ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %        ??? : (%,Integer) -> %
--R ??? : (%,PositiveInteger) -> %      ?+? : (%,%) -> %
--R ?-? : (%,%) -> %                     -? : % -> %
--R ?/? : (%,%) -> %                     ?? : (%,%) -> Boolean
--R D : % -> %                           D : (%,NonNegativeInteger) -> %
--R 1 : () -> %                           0 : () -> %
--R ?? : (%,Integer) -> %                 ?? : (%,PositiveInteger) -> %
--R algebraic? : % -> Boolean              associates? : (%,%) -> Boolean
--R basis : () -> Vector(%)                charthRoot : % -> %
--R coerce : Fraction(Integer) -> %         coerce : % -> %
--R coerce : Integer -> %                  coerce : % -> OutputForm
--R convert : % -> Integer                  coordinates : % -> Vector(%)
--R createPrimitiveElement : () -> %        degree : % -> PositiveInteger
--R differentiate : % -> %                  dimension : () -> CardinalNumber
--R enumerate : () -> List(%)               factor : % -> Factored(%)
--R gcd : List(%) -> %                     gcd : (%,%) -> %
--R hash : % -> SingleInteger               inGroundField? : % -> Boolean
--R index : PositiveInteger -> %           init : () -> %
--R inv : % -> %                           latex : % -> String
```



```

--R lcm : List(%) -> %
--R lookup : % -> PositiveInteger
--R one? : % -> Boolean
--R prime? : % -> Boolean
--R primitive? : % -> Boolean
--R ?quo? : (%,%) -> %
--R recip : % -> Union(%, "failed")
--R represents : Vector(%) -> %
--R sample : () -> %
--R sizeLess? : (%,%) -> Boolean
--R squareFreePart : % -> %
--R transcendental? : % -> Boolean
--R unitCanonical : % -> %
--R ?^=? : (%,%) -> Boolean
--R ?? : (NonNegativeInteger, %) -> %
--R ??? : (%, NonNegativeInteger) -> %
--R Frobenius : % -> % if $ has FINITE
--R Frobenius : (%, NonNegativeInteger) -> % if $ has FINITE
--R ?^? : (%, NonNegativeInteger) -> %
--R basis : PositiveInteger -> Vector(%)
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed")
--R conditionP : Matrix(%) -> Union(Vector(%), "failed")
--R coordinates : Vector(%) -> Matrix(%)
--R createNormalElement : () -> % if $ has FINITE
--R definingPolynomial : () -> SparseUnivariatePolynomial(%)
--R degree : % -> OnePointCompletion(PositiveInteger)
--R differentiate : (%, NonNegativeInteger) -> %
--R discreteLog : % -> NonNegativeInteger
--R discreteLog : (%,%) -> Union(NonNegativeInteger, "failed")
--R divide : (%,%) -> Record(quotient: %, remainder: %)
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(%), %) -> Union(List(%), "failed")
--R exquo : (%,%) -> Union(%, "failed")
--R extendedEuclidean : (%,%,%) -> Union(Record(coef1: %, coef2: %), "failed")
--R extendedEuclidean : (%,%) -> Record(coef1: %, coef2: %, generator: %)
--R extensionDegree : () -> OnePointCompletion(PositiveInteger)
--R extensionDegree : () -> PositiveInteger
--R factorsOfCyclicGroupSize : () -> List(Record(factor: Integer, exponent: Integer))
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R generator : () -> % if $ has FINITE
--R linearAssociatedExp : (%, SparseUnivariatePolynomial(%)) -> % if $ has FINITE
--R linearAssociatedLog : % -> SparseUnivariatePolynomial(%) if $ has FINITE
--R linearAssociatedLog : (%,%) -> Union(SparseUnivariatePolynomial(%), "failed") if $ has FINITE
--R linearAssociatedOrder : % -> SparseUnivariatePolynomial(%) if $ has FINITE
--R minimalPolynomial : % -> SparseUnivariatePolynomial(%)
--R minimalPolynomial : (%, PositiveInteger) -> SparseUnivariatePolynomial(%) if $ has FINITE
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed")
--R nextItem : % -> Union(%, "failed")
--R norm : (%, PositiveInteger) -> % if $ has FINITE
--R lcm : (%,%) -> %
--R norm : % -> %
--R order : % -> PositiveInteger
--R primeFrobenius : % -> %
--R primitiveElement : () -> %
--R random : () -> %
--R ?rem? : (%,%) -> %
--R retract : % -> %
--R size : () -> NonNegativeInteger
--R squareFree : % -> Factored(%)
--R trace : % -> %
--R unit? : % -> Boolean
--R zero? : % -> Boolean

```

```

--R normal? : % -> Boolean if $ has FINITE
--R normalElement : () -> % if $ has FINITE
--R order : % -> OnePointCompletion(PositiveInteger)
--R primeFrobenius : (% , NonNegativeInteger) -> %
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %)
--R representationType : () -> Union("prime", polynomial, normal, cyclic)
--R retractIfCan : % -> Union(%, "failed")
--R subtractIfCan : (% , %) -> Union(%, "failed")
--R tableForDiscreteLogarithm : Integer -> Table(PositiveInteger, NonNegativeInteger)
--R trace : (% , PositiveInteger) -> % if $ has FINITE
--R transcendenceDegree : () -> NonNegativeInteger
--R unitNormal : % -> Record(unit: %, canonical: %, associate: %)
--R
--E 1

)spool
)lisp (bye)

```

— PrimeField.help —

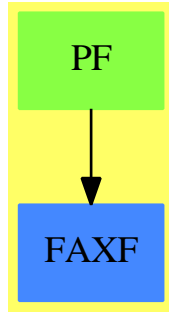
=====

PrimeField examples

=====

PrimeField(p) implements the field with p elements if p is a prime number.
 Error: if p is not prime.
 Note: this domain does not check that argument is a prime.

See Also:
 o)show PrimeField

PrimeField (PF)

See

⇒ “InnerPrimeField” (IPF) [10](#) on page [1303](#)

Exports:

0	1	algebraic?
associates?	basis	characteristic
charthRoot	conditionP	coordinates
coerce	convert	coordinates
createPrimitiveElement	createNormalElement	D
definingPolynomial	degree	differentiate
dimension	discreteLog	discreteLog
divide	euclideanSize	expressIdealMember
exquo	extendedEuclidean	extensionDegree
factor	factorsOfCyclicGroupSize	Frobenius
gcd	gcdPolynomial	generator
hash	inGroundField?	index
init	inv	latex
lcm	linearAssociatedExp	linearAssociatedLog
linearAssociatedOrder	lookup	minimalPolynomial
multiEuclidean	nextItem	norm
normal?	normalElement	one?
order	prime?	primeFrobenius
primitive?	primitiveElement	principalIdeal
random	recip	representationType
represents	retract	retractIfCan
sample	size	sizeLess?
squareFree	squareFreePart	subtractIfCan
tableForDiscreteLogarithm	trace	transcendenceDegree
transcendent?	unit?	unitCanonical
unitNormal	zero?	?*?
***	?+?	?-?
-?	?/?	?=?
?^?	?~=?	?quo?
?rem?		

— domain PF PrimeField —

```

)abbrev domain PF PrimeField
++ Authors: N.N.,
++ Date Created: November 1990, 26.03.1991
++ Date Last Updated: 31 March 1991
++ References:
++ R.Lidl, H.Niederreiter: Finite Field, Encycoldia of Mathematics and
++ Its Applications, Vol. 20, Cambridge Univ. Press, 1983, ISBN 0 521 30240 4
++ Description:
++ PrimeField(p) implements the field with p elements if p is a prime number.
++ Error: if p is not prime.
++ Note: this domain does not check that argument is a prime.

PrimeField(p:PositiveInteger): Exp == Impl where
  Exp ==> Join(FiniteFieldCategory,FiniteAlgebraicExtensionField($),_

```

```

ConvertibleTo(Integer))
Impl ==> InnerPrimeField(p) add
if not prime?(p)$IntegerPrimesPackage(Integer) then
  error "Argument to prime field must be a prime"

```

— PF.dotabb —

```

"PF" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PF"]
"FAXF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAXF"]
"PF" -> "FAXF"

```

domain PRIMARR PrimitiveArray

— PrimitiveArray.input —

```

)set break resume
)sys rm -f PrimitiveArray.output
)spool PrimitiveArray.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PrimitiveArray
--R
--R PrimitiveArray(S: Type) is a domain constructor
--R Abbreviation for PrimitiveArray is PRIMARR
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PRIMARR
--R
--R----- Operations -----
--R concat : List(%) -> %          concat : (%,%) -> %
--R concat : (S,%) -> %          concat : (%,S) -> %
--R construct : List(S) -> %      copy : % -> %
--R delete : (%,Integer) -> %     ?.? : (%,Integer) -> S
--R elt : (%,Integer,S) -> S      empty : () -> %
--R empty? : % -> Boolean         entries : % -> List(S)
--R eq? : (%,%) -> Boolean        index? : (Integer,%) -> Boolean
--R indices : % -> List(Integer)  insert : (%,%,Integer) -> %
--R insert : (S,%,Integer) -> %   map : (((S,S) -> S),%,%) -> %

```

```

--R map : ((S -> S),%) -> %                new : (NonNegativeInteger,S) -> %
--R qelt : (%,Integer) -> S                reverse : % -> %
--R sample : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?<? : (%,%) -> Boolean if S has ORDSET
--R ?<=? : (%,%) -> Boolean if S has ORDSET
--R ?=? : (%,%) -> Boolean if S has SETCAT
--R ?>? : (%,%) -> Boolean if S has ORDSET
--R ?>=? : (%,%) -> Boolean if S has ORDSET
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if S has SETCAT
--R convert : % -> InputForm if S has KONVERT(INFORM)
--R copyInto! : (%,Integer) -> % if $ has shallowlyMutable
--R count : (S,%) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R delete : (%,UniversalSegment(Integer)) -> %
--R ?.? : (%,UniversalSegment(Integer)) -> %
--R entry? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R eval : (%,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (%,S) -> % if $ has shallowlyMutable
--R find : ((S -> Boolean),%) -> Union(S,"failed")
--R first : % -> S if Integer has ORDSET
--R hash : % -> SingleInteger if S has SETCAT
--R latex : % -> String if S has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R max : (%,%) -> % if S has ORDSET
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R members : % -> List(S) if $ has finiteAggregate
--R merge : (%,%) -> % if S has ORDSET
--R merge : (((S,S) -> Boolean),%,%) -> %
--R min : (%,%) -> % if S has ORDSET
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (%,NonNegativeInteger) -> Boolean
--R parts : % -> List(S) if $ has finiteAggregate
--R position : (S,%,Integer) -> Integer if S has SETCAT
--R position : (S,%) -> Integer if S has SETCAT
--R position : ((S -> Boolean),%) -> Integer
--R qsetelt! : (%,Integer,S) -> S if $ has shallowlyMutable
--R reduce : (((S,S) -> S),%) -> S if $ has finiteAggregate
--R reduce : (((S,S) -> S),%,S) -> S if $ has finiteAggregate
--R reduce : (((S,S) -> S),%,S,S) -> S if $ has finiteAggregate and S has SETCAT
--R remove : ((S -> Boolean),%) -> % if $ has finiteAggregate
--R remove : (S,%) -> % if $ has finiteAggregate and S has SETCAT
--R removeDuplicates : % -> % if $ has finiteAggregate and S has SETCAT

```

```

--R reverse! : % -> % if $ has shallowlyMutable
--R select : ((S -> Boolean),%) -> % if $ has finiteAggregate
--R setelt : (%,UniversalSegment(Integer),S) -> S if $ has shallowlyMutable
--R setelt : (%,Integer,S) -> S if $ has shallowlyMutable
--R size? : (%,NonNegativeInteger) -> Boolean
--R sort : % -> % if S has ORDSET
--R sort : (((S,S) -> Boolean),%) -> %
--R sort! : % -> % if $ has shallowlyMutable and S has ORDSET
--R sort! : (((S,S) -> Boolean),%) -> % if $ has shallowlyMutable
--R sorted? : % -> Boolean if S has ORDSET
--R sorted? : (((S,S) -> Boolean),%) -> Boolean
--R swap! : (%,Integer,Integer) -> Void if $ has shallowlyMutable
--R ~=? : (%,%) -> Boolean if S has SETCAT
--R
--E 1

)spool
)lisp (bye)

```

— PrimitiveArray.help —

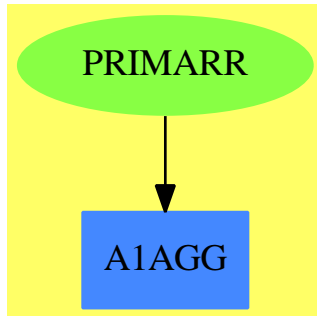
```

=====
PrimitiveArray examples
=====

```

This provides a fast array type with no bound checking on elt's.
 Minimum index is 0 in this type, cannot be changed

See Also:
 o)show PrimitiveArray

PrimitiveArray (PRIMARR)**See**

- ⇒ “Tuple” (TUPLE) [21](#) on page [2752](#)
- ⇒ “IndexedFlexibleArray” (IFARRAY) [10](#) on page [1222](#)
- ⇒ “FlexibleArray” (FARRAY) [7](#) on page [885](#)
- ⇒ “IndexedOneDimensionalArray” (IARRAY1) [10](#) on page [1243](#)
- ⇒ “OneDimensionalArray” (ARRAY1) [16](#) on page [1774](#)

Exports:

any?	coerce	concat	construct	convert
copy	copyInto!	count	delete	entry?
elt	empty	empty?	entries	eq?
eval	every?	fill!	find	first
hash	index?	indices	insert	insert
latex	less?	map	map!	max
maxIndex	member?	members	merge	min
minIndex	more?	new	parts	position
qelt	qsetelt!	reduce	remove	removeDuplicates
reverse	reverse!	sample	select	setelt
size?	sort	sort!	sorted?	swap!
#?	??	?~=?	?<?	?<=?
?=?	?>?	?>=?		

— domain **PRIMARR PrimitiveArray** —

```

)abbrev domain PRIMARR PrimitiveArray
++ Author: Mark Botch
++ Description:
++ This provides a fast array type with no bound checking on elt's.
++ Minimum index is 0 in this type, cannot be changed

```

```

PrimitiveArray(S:Type): OneDimensionalArrayAggregate S == add
  Qmax ==> QVMAXINDEX$Lisp
  Qsize ==> QVSIZE$Lisp
--  Qelt ==> QVELT$Lisp
--  Qsetelt ==> QSETVELT$Lisp

```



```

--S 2 of 6
f(3)
--R
--R
--R (2)  9
--R
--R                                         Type: PositiveInteger
--E 2

--S 3 of 6
g:=(x:INT):INT +-> x^3
--R
--R
--R (3)  theMap(Closure)
--R
--R                                         Type: (Integer -> Integer)
--E 3

--S 4 of 6
g(3)
--R
--R
--R (4)  27
--R
--R                                         Type: PositiveInteger
--E 4

--S 5 of 6
h(x:INT):Product(INT,INT) == makeprod(f x, g x)
--R
--R Function declaration h : Integer -> Product(Integer,Integer) has
--R      been added to workspace.
--R
--R                                         Type: Void
--E 5

--S 6 of 6
h(3)
--R
--R Compiling function h with type Integer -> Product(Integer,Integer)
--R
--R (6)  (9,27)
--R
--R                                         Type: Product(Integer,Integer)
--E 6

)spool
)lisp (bye)

```

— Product.help —

=====

Product examples

=====

The direct product of two functions over the same domain is another function over that domain whose co-domain is the product of their co-domains.

So we can define two functions, f and g , that go from $\text{INT} \rightarrow \text{INT}$

```
f:=(x:INT):INT +-> 3*x
g:=(x:INT):INT +-> x^3
```

so

```
f(3) is 9
g(3) is 27
```

and we can construct the direct product of those functions $h=f,g$

```
h(x:INT):Product(INT,INT) == makeprod(f x, g x)
```

so

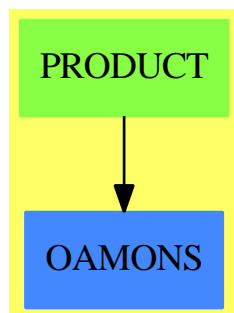
```
h(3) is (9,27)
```

See Also:

```
o )show Product
```

—————→

Product (PRODUCT)



Exports:

0	1	coerce	commutator	conjugate
hash	index	inv	latex	lookup
makeprod	max	min	one?	random
recip	sample	selectfirst	selectsecond	size
subtractIfCan	sup	zero?	?=?	?~=?
?*?	?**?	?+?	?-?	?-?
?/?	?<?	?<=?	?>?	?>=?
?^?				

— domain PRODUCT Product —

```

)abbrev domain PRODUCT Product
++ Author: Mark Botch
++ Description:
++ This domain implements cartesian product

Product (A:SetCategory,B:SetCategory) : C == T
where
  C == SetCategory with
    if A has Finite and B has Finite then Finite
    if A has Monoid and B has Monoid then Monoid
    if A has AbelianMonoid and B has AbelianMonoid then AbelianMonoid
    if A has CancellationAbelianMonoid and
      B has CancellationAbelianMonoid then CancellationAbelianMonoid
    if A has Group and B has Group then Group
    if A has AbelianGroup and B has AbelianGroup then AbelianGroup
    if A has OrderedAbelianMonoidSup and B has OrderedAbelianMonoidSup
      then OrderedAbelianMonoidSup
    if A has OrderedSet and B has OrderedSet then OrderedSet

  makeprod      : (A,B) -> %
  ++ makeprod(a,b) computes the product of two functions
  ++
  ++X f:=(x:INT):INT +-> 3*x
  ++X g:=(x:INT):INT +-> x^3
  ++X h(x:INT):Product(INT,INT) == makeprod(f x, g x)
  ++X h(3)
  selectfirst   : % -> A
  ++ selectfirst(x) is not documented
  selectsecond  : % -> B
  ++ selectsecond(x) is not documented

T == add

--representations
Rep := Record(acomp:A,bcomp:B)

--declarations
x,y: %

```

```

i: NonNegativeInteger
p: NonNegativeInteger
a: A
b: B
d: Integer

--define
coerce(x):OutputForm == paren [(x.accomp)::OutputForm,
                                (x.bcomp)::OutputForm]

x=y ==
  x.accomp = y.accomp => x.bcomp = y.bcomp
  false
makeprod(a:A,b:B) :%    == [a,b]

selectfirst(x:%) : A    == x.accomp

selectsecond (x:%) : B == x.bcomp

if A has Monoid and B has Monoid then
  1 == [1$A,1$B]
  x * y == [x.accomp * y.accomp,x.bcomp * y.bcomp]
  x ** p == [x.accomp ** p ,x.bcomp ** p]

if A has Finite and B has Finite then
  size == size$A () * size$B ()

if A has Group and B has Group then
  inv(x) == [inv(x.accomp),inv(x.bcomp)]

if A has AbelianMonoid and B has AbelianMonoid then
  0 == [0$A,0$B]

  x + y == [x.accomp + y.accomp,x.bcomp + y.bcomp]

  c:NonNegativeInteger * x == [c * x.accomp,c*x.bcomp]

if A has CancellationAbelianMonoid and
B has CancellationAbelianMonoid then
  subtractIfCan(x, y) : Union(%, "failed") ==
    (na:= subtractIfCan(x.accomp, y.accomp)) case "failed" => "failed"
    (nb:= subtractIfCan(x.bcomp, y.bcomp)) case "failed" => "failed"
    [na::A,nb::B]

if A has AbelianGroup and B has AbelianGroup then
  - x == [- x.accomp,-x.bcomp]
  (x - y):% == [x.accomp - y.accomp,x.bcomp - y.bcomp]
  d * x == [d * x.accomp,d * x.bcomp]

if A has OrderedAbelianMonoidSup and B has OrderedAbelianMonoidSup then
  sup(x,y) == [sup(x.accomp,y.accomp),sup(x.bcomp,y.bcomp)]

```

```

if A has OrderedSet and B has OrderedSet then
  x < y ==
    xa:= x.accomp ; ya:= y.accomp
    xa < ya => true
    xb:= x.bcomp ; yb:= y.bcomp
    xa = ya => (xb < yb)
    false

--      coerce(x:%):Symbol ==
--      PrintableForm()
--      formList([x.accomp::Expression,x.bcomp::Expression])$PrintableForm

```

— PRODUCT.dotabb —

```

"PRODUCT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PRODUCT"]
"OAMONS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAMONS"]
"PRODUCT" -> "OAMONS"

```

domain PROJPL ProjectivePlane

— ProjectivePlane.input —

```

)set break resume
)sys rm -f ProjectivePlane.output
)spool ProjectivePlane.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ProjectivePlane
--R
--R ProjectivePlane(K: Field) is a domain constructor
--R Abbreviation for ProjectivePlane is PROJPL
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PROJPL
--R
--R----- Operations -----
--R ==? : (%,%) -> Boolean          coerce : List(K) -> %

```

```

--R coerce : % -> List(K)
--R conjugate : % -> %
--R degree : % -> PositiveInteger
--R hash : % -> SingleInteger
--R homogenize : (%,Integer) -> %
--R lastNonNull : % -> Integer
--R list : % -> List(K)
--R pointValue : % -> List(K)
--R rational? : % -> Boolean
--R ?~=? : (%,% ) -> Boolean
--R conjugate : (% ,NonNegativeInteger) -> %
--R orbit : (% ,NonNegativeInteger) -> List(% )
--R rational? : (% ,NonNegativeInteger) -> Boolean
--R removeConjugate : List(% ) -> List(% )
--R removeConjugate : (List(% ),NonNegativeInteger) -> List(% )
--R
--E 1

)spool
)lisp (bye)

```

— ProjectivePlane.help —

=====

ProjectivePlane examples

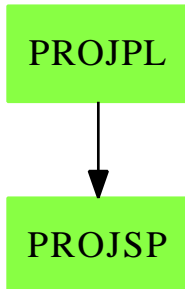
=====

This is part of the PAFF package, related to projective space.

See Also:

o)show ProjectivePlane

ProjectivePlane (PROJPL)

**Exports:**

??	??	?~=?	coerce	conjugate
definingField	degree	hash	homogenize	lastNonNul
lastNonNull	latex	list	orbit	pointValue
projectivePoint	rational?	removeConjugate	setelt	

— domain **PROJPL** ProjectivePlane —

```

)abbrev domain PROJPL ProjectivePlane
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ This is part of the PAFF package, related to projective space.
ProjectivePlane(K):Exports == Implementation where
  K:Field

Exports ==> ProjectiveSpaceCategory(K)

Implementation ==> ProjectiveSpace(3,K)

```

—————

— PROJPL.dotabb —

```

"PROJPL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PROJPL"];
"PROJSP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PROJSP"];
"PROJPL" -> "PROJSP"

```

—————

domain PROJPLPS ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField

— ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField.input —

```
)set break resume
)sys rm -f ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField.output
)spool ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField
--R
--R ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K: FiniteFieldCategory) is a domain
--R Abbreviation for ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField is PROJPLPS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PROJPLPS
--R
--R----- Operations -----
--R ?=? : (%,%) -> Boolean          coerce : % -> OutputForm
--R conjugate : % -> %              degree : % -> PositiveInteger
--R hash : % -> SingleInteger       homogenize : % -> %
--R homogenize : (%,Integer) -> %   lastNonNul : % -> Integer
--R lastNonNull : % -> Integer      latex : % -> String
--R orbit : % -> List(%)           rational? : % -> Boolean
--R ?~=? : (%,%) -> Boolean
--R coerce : List(PseudoAlgebraicClosureOfFiniteField(K)) -> %
--R coerce : % -> List(PseudoAlgebraicClosureOfFiniteField(K))
--R conjugate : (%,NonNegativeInteger) -> %
--R definingField : % -> PseudoAlgebraicClosureOfFiniteField(K)
--R ?.? : (%,Integer) -> PseudoAlgebraicClosureOfFiniteField(K)
--R list : % -> List(PseudoAlgebraicClosureOfFiniteField(K))
--R orbit : (%,NonNegativeInteger) -> List(%)
--R pointValue : % -> List(PseudoAlgebraicClosureOfFiniteField(K))
--R projectivePoint : List(PseudoAlgebraicClosureOfFiniteField(K)) -> %
--R rational? : (%,NonNegativeInteger) -> Boolean
--R removeConjugate : List(%) -> List(%)
--R removeConjugate : (List(%),NonNegativeInteger) -> List(%)
--R setelt : (%,Integer,PseudoAlgebraicClosureOfFiniteField(K)) -> PseudoAlgebraicClosureOfFiniteField(K)
--R
--E 1

)spool
)lisp (bye)
```

— ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField.help —

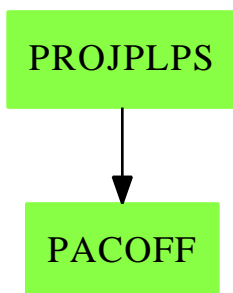
ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField examples

This is part of the PAFF package, related to projective space.

See Also:

o)show ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField

ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField (PROJPLPS)



Exports:

?.?	?=?	?~=?	coerce	conjugate
definingField	degree	hash	homogenize	lastNonNul
lastNonNull	latex	list	orbit	orbit
pointValue	projectivePoint	rational?	removeConjugate	setelt

— domain PROJPLPS ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField —

```

)abbrev domain PROJPLPS ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ This is part of the PAFF package, related to projective space.
  
```

```

ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K):Exp == Impl where
  K:FiniteFieldCategory

  KK ==> PseudoAlgebraicClosureOfFiniteField(K)

  Exp ==> ProjectiveSpaceCategory(KK)

  Impl ==> ProjectivePlane(KK)

```

— PROJPLPS.dotabb —

```

"PROJPLPS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PROJPLPS"];
"PACOFF" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PACOFF"];
"PROJPLPS" -> "PACOFF"

```

domain PROJSP ProjectiveSpace

— ProjectiveSpace.input —

```

)set break resume
)sys rm -f ProjectiveSpace.output
)spool ProjectiveSpace.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ProjectiveSpace
--R
--R ProjectiveSpace(dim: NonNegativeInteger,K: Field) is a domain constructor
--R Abbreviation for ProjectiveSpace is PROJSP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PROJSP
--R
--R----- Operations -----
--R ?=? : (%,%) -> Boolean               coerce : List(K) -> %
--R coerce : % -> List(K)               coerce : % -> OutputForm
--R conjugate : % -> %                  definingField : % -> K
--R degree : % -> PositiveInteger       ?? : (%,Integer) -> K
--R hash : % -> SingleInteger           homogenize : % -> %

```

```

--R homogenize : (%,Integer) -> %          lastNonNul : % -> Integer
--R lastNonNull : % -> Integer              latex : % -> String
--R list : % -> List(K)                     orbit : % -> List(%)
--R pointValue : % -> List(K)               projectivePoint : List(K) -> %
--R rational? : % -> Boolean                setelt : (%,Integer,K) -> K
--R ?~=? : (%,% ) -> Boolean
--R conjugate : (% ,NonNegativeInteger) -> %
--R orbit : (% ,NonNegativeInteger) -> List(%)
--R rational? : (% ,NonNegativeInteger) -> Boolean
--R removeConjugate : List(%) -> List(%)
--R removeConjugate : (List(%) ,NonNegativeInteger) -> List(%)
--R
--E 1

```

```

)spool
)lisp (bye)

```

— ProjectiveSpace.help —

=====

ProjectiveSpace examples

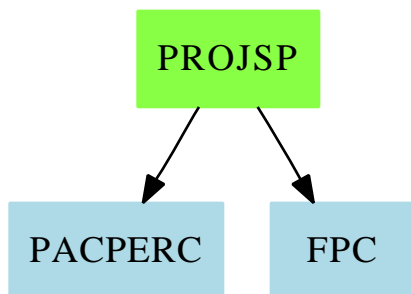
=====

This is part of the PAFF package, related to projective space.

See Also:

o)show ProjectiveSpace

ProjectiveSpace (PROJSP)



Exports:

?.?	?=?	?~=?	coerce	conjugate
definingField	degree	hash	homogenize	lastNonNul
lastNonNull	latex	list	orbit	orbit
pointValue	projectivePoint	rational?	removeConjugate	setelt

— domain PROJSP ProjectiveSpace —

```

)abbrev domain PROJSP ProjectiveSpace
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ This is part of the PAFF package, related to projective space.
ProjectiveSpace(dim,K):Exports == Implementation where

  NNI ==> NonNegativeInteger
  LIST ==> List

  dim:NNI
  K:Field

  Exports ==> ProjectiveSpaceCategory(K)

  Implementation ==> List(K) add

  Rep:= List(K)

  coerce(pt:%):OutputForm ==
    dd:OutputForm:= ":" :: OutputForm
    llout:List(OutputForm):=[ hconcat(dd, a::OutputForm) for a in rest pt]
    lout:= cons( (first pt)::OutputForm , llout)
    out:= hconcat lout
    oo:=paren(out)
    ee:OutputForm:= degree(pt) :: OutputForm
    oo**ee

  definingField(pt)==
    K has PseudoAlgebraicClosureOfPerfectFieldCategory => _
      maxTower(pt pretend Rep)
    1$K

  degree(pt)==
    K has PseudoAlgebraicClosureOfPerfectFieldCategory => _
      extDegree definingField pt
    1

  coerce(pt:%):List(K) == pt pretend Rep

  projectivePoint(pt:LIST(K))==

```

```

    pt :: %

list(ptt)==
    ptt pretend Rep

pointValue(ptt)==
    ptt pretend Rep

conjugate(p,e)==
    lp:Rep:=p
    pc:List(K):=[c**e for c in lp]
    projectivePoint(pc)

homogenize(ptt,nV)==
    if K has Field then
        pt:=list(ptt)%
        zero?(pt.nV) => error "Impossible to homogenize this point"
        divPt:=pt.nV
        [(a/divPt) for a in pt]
    else
        ptt

rational?(p,n)== p=conjugate(p,n)

rational?(p)==rational?(p,characteristic())$K

removeConjugate(l)==removeConjugate(l,characteristic())$K

removeConjugate(l:LIST(%),n:NNI):LIST(%)==
    if K has FiniteFieldCategory then
        allconj:LIST(%):=empty()
        conjrem:LIST(%):=empty()
        for p in l repeat
            if ~member?(p,allconj) then
                conjrem:=cons(p,conjrem)
                allconj:=concat(allconj,orbit(p,n))
        conjrem
    else
        error "The field is not finite"

conjugate(p)==conjugate(p,characteristic())$K

orbit(p)==orbit(p,characteristic())$K

orbit(p,e)==
    if K has FiniteFieldCategory then
        l:LIST(%) := [p]
        np: % := conjugate(p,e)
        flag:=~(np=p)::Boolean
        while flag repeat

```

```

        l:=concat(np,l)
        np:=conjugate(np,e)
        flag:=not (np=p)::Boolean
    1
else
    error "Cannot compute the conjugate"

aa:% = bb:% ==
ah:=homogenize(aa)
bh:=homogenize(bb)
ah =$Rep bh

coerce(pt:LIST(K))==
    ^(dim=#pt) => error "Le point n'a pas la bonne dimension"
    reduce("and",[zero?(a) for a in pt]) => _
        error "Ce n'est pas un point projectif"
    ptt:%= pt
    homogenize ptt

homogenize(ptt)==
    homogenize(ptt,lastNonNull(ptt))

nonZero?: K -> Boolean
nonZero?(a)==
    not(zero?(a))

lastNonNull(ptt)==
    pt:=ptt pretend Rep
    (dim pretend Integer)+1-_
        (position("nonZero?",(reverse(pt)$LIST(K)))$LIST(K))

lastNonNul(pt)==lastNonNull(pt)

```

— PROJSP.dotabb —

```

"PROJSP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PROJSP"];
"PACPERC" [color=lightblue,href="bookvol10.2.pdf#nameddest=PACPERC"];
"FPC" [color=lightblue,href="bookvol10.2.pdf#nameddest=FPC"];
"PROJSP" -> "PACPERC"
"PROJSP" -> "FPC"

```

domain PACEXT PseudoAlgebraicClosureOfAlgExtOfRational- Number

— PseudoAlgebraicClosureOfAlgExtOfRationalNumber.input —

```
)set break resume
)sys rm -f PseudoAlgebraicClosureOfAlgExtOfRationalNumber.output
)spool PseudoAlgebraicClosureOfAlgExtOfRationalNumber.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PseudoAlgebraicClosureOfAlgExtOfRationalNumber
--E 1

)spool
)lisp (bye)
```

— PseudoAlgebraicClosureOfAlgExtOfRationalNumber.help —

```
=====
PseudoAlgebraicClosureOfAlgExtOfRationalNumber examples
=====
```

This domain implement dynamic extension over the
PseudoAlgebraicClosureOfRationalNumber.

A tower extension T of the ground field K is any sequence of field
extension

$(T : K_0, K_1, \dots, K_i, \dots, K_n)$

where $K_0 = K$ and for $i = 1, 2, \dots, n$, K_i is an extension of
 K_{i-1} of degree > 1 and defined by an irreducible polynomial
 $p(Z)$ in K_{i-1} .

Two towers

$(T_1: K_{01}, K_{11}, \dots, K_{i1}, \dots, K_{n1})$

and

$(T_2: K_{02}, K_{12}, \dots, K_{i2}, \dots, K_{n2})$

are said to be related if

$T_1 \leq T_2$ (or $T_1 \geq T_2$),

that is if

$K_{i1} = K_{i2}$ for $i = 1, 2, \dots, n1$

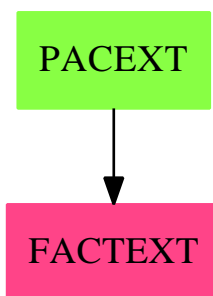
(or $i = 1, 2, \dots, n2$). Any algebraic operations defined for several elements
are only defined if all of the concerned elements are coming from

a set of related tour extensions.

See Also:

o)show PseudoAlgebraicClosureOfAlgExtOfRationalNumber

PseudoAlgebraicClosureOfAlgExtOfRationalNumber (PACEXT)



Exports:

— domain PACEXT PseudoAlgebraicClosureOfAlgExtOfRational-
Number —

```

)abbrev domain PACEXT PseudoAlgebraicClosureOfAlgExtOfRationalNumber
-- PseudoAlgebraicClosureOfAlgExtOfRationalNumber
++ Authors: Gaetan Hache
++ Date Created: jan 1998
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ This domain implement dynamic extension over the
++ PseudoAlgebraicClosureOfRationalNumber.
++ A tower extension T of the ground field K is any sequence of field
++ extension (T : K_0, K_1, ..., K_i...,K_n) where K_0 = K
++ and for i =1,2,...,n, K_i is an extension of K_{i-1} of degree > 1
++ and defined by an irreducible polynomial p(Z) in K_{i-1}.
++ Two towers (T_1: K_01, K_11,...,K_i1,...,K_n1) and
++ (T_2: K_02, K_12,...,K_i2,...,K_n2)
++ are said to be related if T_1 <= T_2 (or T_1 >= T_2),
++ that is if K_i1 = K_i2 for i=1,2,...,n1
++ (or i=1,2,...,n2). Any algebraic operations defined for several elements
++ are only defined if all of the concerned elements are coming from
  
```

```

++ a set of related tour extensions.
PseudoAlgebraicClosureOfAlgExtOfRationalNumber(downLevel:K):Exp == Impl where
  K      ==> PseudoAlgebraicClosureOfRationalNumber
  INT    ==> Integer
  NNI    ==> NonNegativeInteger
  SUP    ==> SparseUnivariatePolynomial
  BOOLEAN ==> Boolean
  PI     ==> PositiveInteger
  FACTRN ==> FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber

recRep ==> Record(recEl:SUP(%),_
                  recTower:SUP(%),_
                  recDeg:PI,_
                  recPrevTower:%,_
                  recName:Symbol)

Exp == PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategory with

  fullOutput: % -> OutputForm

  retractToGrn: % -> K

Impl == add
  Rep := Union(recRep,K)

  -- signature of local function
  replaceRecEl: (% ,SUP(%)) -> %

  down: % -> %

  retractPol( pol:SUP(%)) :SUP(K)==
    zero? pol => 0$SUP(K)
    lc := leadingCoefficient pol
    d := degree pol
    rlc := retractToGrn( lc )
    monomial( rlc , d )$SUP(K) + retractPol( reductum pol )

  retractToGrn(aa)==
    aa case K => aa
    a:=(aa pretend recRep)
    el:= a.recEl
    t:= a.recTower
    d:= a.recDeg * extDegree downLevel
    pt:= a.recPrevTower
    n:= a.recName
    newElement(retractPol el, retractPol t, d, retractToGrn pt, n)$K

  newElement(pol,subF,inName) ==
    -- pol is an irreducible polynomial over the field extension
    -- given by subF.

```

```

-- The output of this function is a root of pol.
dp:=degree pol
listCoef: List % := coefficients pol
a1:% := inv first listCoef
b1:% := last listCoef
rr:% := b1*a1
one?(dp) =>
  one?(#listCoef) => 0
  - rr
ground?(pol) => error "Cannot create a new element with a constant"
d:PI := (dp pretend PI) * extDegree(subF)
[monomial(1$,1)$SUP(%),pol,d,subF,inName] :: Rep

coerce(a:Integer):%== (a :: K)

down(a:%) ==
  a case K => a
  aa:=(a pretend recRep)
  elel := aa.recEl
  ^ground?(elel)$SUP(%) => a
  gel:%:=ground(elel)
  down(gel)

n:INT * a:% ==
  one?(n) => a
  zero?(a) or zero?(n) => 0
  (n < 0) => -((-n)*a)
  mm:PositiveInteger:=(n pretend PositiveInteger)
  double(mm,a)$RepeatedDoubling(%)

replaceRecEl(a,el)==
  a case K => a
  aa:=copy a
  aa.recEl := el
  aa

localTower :% := downLevel

lift(a) ==
  a case K => monomial(a,0)
  (a pretend recRep).recEl

lift(a,b)==
  extDegree a > extDegree b => _
  error "Cannot lift something at lower level !!!!!"
  extDegree a < extDegree b => monomial(a,0)$SUP(%)
  lift a

reduce(a)==
  localTower case K =>

```

```

    coefficient(a,0)
    ar:= a rem (localTower pretend recRep).recTower
    replaceRecEl(localTower,ar)

maxTower(la)==
    --return an element from the list la which is in the largest
    --extension of the ground field
    --PRECONDITION: all elements in same tower, else no meaning?
    m:="max"/[extDegree(a)$% for a in la]
    first [b for b in la | extDegree(b)=m]

ground?(a)== a case K

vectorise(a,lev)==
    da:=extDegree a
    dlev:=extDegree lev
    dlev < da => _
        error "Cannot vectorise at a lower level than the element to vectorise"
    lev case K => [a]
    pa:SUP(%)
    na:%
    ^ (da = dlev) =>
        pa:= monomial(a,0)$SUP(%)
        na:= replaceRecEl(lev,pa)
        vectorise(na,lev)$%

prevLev:=previousTower(lev)
a case K => error "At this point a is not suppose to be in K"
aEl:=(a pretend recRep).recEl
daEl:=degree definingPolynomial(a)$%
lv:=[vectorise(c,prevLev)$% for c in entries(vectorise(aEl,daEl)$SUP(%))]]
concat lv

retractIfCan(a:%):Union(K,"failed")==
    a case K => a
    "failed"

retractIfCan(a:%):Union(Integer,"failed")==
    a case K => retractIfCan(a)$K
    "failed"

setTower!(a) ==
    if a case K then
        localTower := downLevel
    else
        localTower:=a
    void()

definingPolynomial == definingPolynomial(localTower)

```

```

a:% + b:% ==
(a case K) and (b case K) => a +$K b
extDegree(a) > extDegree(b) => b + a
res1:SUP(%)
res2:%
if extDegree(a) = extDegree(b) then
  res1:= b.recEl +$SUP(%) a.recEl
  res2:= replaceRecEl(b,res1)
else
  res1:= b.recEl +$SUP(%) monomial(a,0)$SUP(%)
  res2:= replaceRecEl(b,res1)
down(res2)

a:% * b:% ==
(a case K) and (b case K) => a *$K b
extDegree(a) > extDegree(b) => b * a
res1:SUP(%)
res2:%
if extDegree(a) = extDegree(b) then
  res1:= b.recEl *$SUP(%) a.recEl rem b.recTower
  res2:= replaceRecEl(b,res1)
else
  res1:= b.recEl *$SUP(%) monomial(a,0)$SUP(%)
  res2:= replaceRecEl(b,res1)
down(res2)

distinguishedRootsOf(polyZero,ee) ==
  setTower!(ee)
  zero?(polyZero) => error "to lazy to give you all the roots of 0 !!!"
  factorf: Factored SUP % := factor(polyZero,localTower)$FACTRN(%)
  listFact:List SUP % := [pol.fctr for pol in factorList(factorf)]
  listOfZeros:List(%) := empty()
  for p in listFact repeat
    root:=newElement(p, new(E::Symbol)$Symbol)
    listOfZeros:List(%) := concat([ root ], listOfZeros)
  listOfZeros

1 == 1$K

0 == 0$K

newElement(poll:SUP(%),inName:Symbol)==
  newElement(poll,localTower,inName)$%

--Field operations
inv(a)==
  a case K => inv(a)$K
  aRecEl:= (a pretend recRep).recEl
  aDefPoly:= (a pretend recRep).recTower
  aInv := extendedEuclidean( aRecEl , aDefPoly, 1 )

```

```

aInv case "failed" => error "PACOFF : division by zero"
-- On doit retourner un Record representant l'inverse de a.
-- Ce Record est exactement le mme que celui de a sauf
-- qu'il faut remplacer le polynme du selecteur recEl
-- par le polynme representant l'inverse de a :
-- C'est ce que fait la fonction replaceRecEl.
replaceRecEl( a , aInv.coef1 )

a:% / b:% == a * inv(b)

a:K * b:%==
  (a :: %) * b

b:% * a:K == a*b

a:% - b:% ==
  a + (-b)

a:% * b:Fraction(Integer) ==
  bn:=number b
  bd:=denom b
  ebn:%:= bn * 1$%
  ebd:%:= bd * 1$%
  a * ebn * inv(ebd)

-a:% ==
  a case K => -$K a
  [-$SUP(%) (a pretend recRep).recEl,_
   (a pretend recRep).recTower,_
   (a pretend recRep).recDeg,_
   (a pretend recRep).recPrevTower,_
   (a pretend recRep).recName ]

bb:% = aa:% ==
  b:=down bb
  a:=down aa
  ^( extDegree(b) =$NNI extDegree(a) ) => false
  (b case K) => ( (retract a)@K =$K (retract b)@K )
  rda := a :: recRep
  rdb := b :: recRep
  not (rda.recTower =$SUP(%) rdb.recTower) => false
  rdb.recEl =$SUP(%) rda.recEl

zero?(a:%) ==
  da:=down a -- just to be sure !!!
  ^(da case K) => false
  zero?(da)$K

one?(a:%) ==
  da:= down a -- just to be sure !!!

```

```

      ^ (da case K) => false
      one?(da)$K

coerce(a:K):% == a

coerce(a:%):OutputForm ==
  a case K => ((retract a)@K) ::OutputForm
  outputForm((a pretend recRep).recEl, _
    ((a pretend recRep).recName)::OutputForm) $SUP(%)

fullOutput(a:%):OutputForm==
  a case K => ((retract a)@K) ::OutputForm
  (a pretend recRep)::OutputForm

definingPolynomial(a:%): SUP % ==
  a case K => monomial(1,1)$SUP(%)
  (a pretend recRep).recTower

extDegree(a:%): PI ==
  a case K => 1
  (a pretend recRep).recDeg

previousTower(a:%):% ==
  a case K => error "No previous extension for ground field element"
  (a pretend recRep).recPrevTower

name(a:%):Symbol ==
  a case K => error "No name for ground field element"
  (a pretend recRep).recName

characteristic == characteristic()$K

```

— PACEXT.dotabb —

```

"PACEXT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PACEXT"];
"FACTEXT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FACTEXT"]
"PACEXT" -> "FACTEXT"

```

domain PACOFF PseudoAlgebraicClosureOfFiniteField

— PseudoAlgebraicClosureOfFiniteField.input —

```

)set break resume
)sys rm -f PseudoAlgebraicClosureOfFiniteField.output
)spool PseudoAlgebraicClosureOfFiniteField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PseudoAlgebraicClosureOfFiniteField
--R PseudoAlgebraicClosureOfFiniteField(K: FiniteFieldCategory) is a domain constructor
--R Abbreviation for PseudoAlgebraicClosureOfFiniteField is PACOFF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PACOFF
--R
--R----- Operations -----
--R ?? : (%,K) -> %           ?? : (K,%) -> %
--R ?? : (Fraction(Integer),%) -> %   ?? : (%,Fraction(Integer)) -> %
--R ?? : (%,%) -> %           ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,Integer) -> %
--R ??? : (%,PositiveInteger) -> %   ?+? : (%,%) -> %
--R ?-? : (%,%) -> %           -? : % -> %
--R ?/? : (%,K) -> %           ?/? : (%,%) -> %
--R ?? : (%,%) -> Boolean       D : % -> %
--R D : (%,NonNegativeInteger) -> %   1 : () -> %
--R 0 : () -> %                 ?^? : (%,Integer) -> %
--R ?? : (%,PositiveInteger) -> %   algebraic? : % -> Boolean
--R associates? : (%,%) -> Boolean   charthRoot : % -> %
--R coerce : K -> %               coerce : Fraction(Integer) -> %
--R coerce : % -> %               coerce : Integer -> %
--R coerce : % -> OutputForm       conjugate : % -> %
--R createPrimitiveElement : () -> %   differentiate : % -> %
--R dimension : () -> CardinalNumber   enumerate : () -> List(%)
--R extDegree : % -> PositiveInteger   factor : % -> Factored(%)
--R fullOutput : % -> OutputForm       gcd : List(%) -> %
--R gcd : (%,%) -> %               ground? : % -> Boolean
--R hash : % -> SingleInteger         inGroundField? : % -> Boolean
--R index : PositiveInteger -> %       init : () -> %
--R inv : % -> %                   latex : % -> String
--R lcm : List(%) -> %             lcm : (%,%) -> %
--R lookup : % -> PositiveInteger       maxTower : List(%) -> %
--R one? : % -> Boolean             order : % -> PositiveInteger
--R previousTower : % -> %           prime? : % -> Boolean
--R primeFrobenius : % -> %         primitive? : % -> Boolean
--R primitiveElement : () -> %       ?quo? : (%,%) -> %
--R random : () -> %               recip : % -> Union(%, "failed")
--R ?rem? : (%,%) -> %             retract : % -> K
--R sample : () -> %               setTower! : % -> Void
--R size : () -> NonNegativeInteger   sizeLess? : (%,%) -> Boolean
--R squareFree : % -> Factored(%)     squareFreePart : % -> %
--R transcendent? : % -> Boolean     unit? : % -> Boolean

```



```

--R unitCanonical : % -> %                                vectorise : (%,% ) -> Vector(%)
--R zero? : % -> Boolean                                   ?~=? : (%,% ) -> Boolean
--R ??? : (NonNegativeInteger,% ) -> %
--R ??? : (% ,NonNegativeInteger) -> %
--R Frobenius : % -> % if K has FINITE
--R Frobenius : (% ,NonNegativeInteger) -> % if K has FINITE
--R ?^? : (% ,NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(% ,"failed")
--R conditionP : Matrix(% ) -> Union(Vector(% ),"failed")
--R definingPolynomial : () -> SparseUnivariatePolynomial(%)
--R definingPolynomial : % -> SparseUnivariatePolynomial(%)
--R degree : % -> OnePointCompletion(PositiveInteger)
--R differentiate : (% ,NonNegativeInteger) -> %
--R discreteLog : % -> NonNegativeInteger
--R discreteLog : (% ,%) -> Union(NonNegativeInteger,"failed")
--R distinguishedRootsOf : (SparseUnivariatePolynomial(% ),%) -> List(%)
--R divide : (% ,%) -> Record(quotient: % ,remainder: %)
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(% ),%) -> Union(List(% ),"failed")
--R exquo : (% ,%) -> Union(% ,"failed")
--R extendedEuclidean : (% ,%,%) -> Union(Record(coef1: % ,coef2: % ),"failed")
--R extendedEuclidean : (% ,%) -> Record(coef1: % ,coef2: % ,generator: %)
--R extensionDegree : () -> OnePointCompletion(PositiveInteger)
--R factorsOfCyclicGroupSize : () -> List(Record(factor: Integer,exponent: Integer))
--R gcdPolynomial : (SparseUnivariatePolynomial(% ),SparseUnivariatePolynomial(% )) -> SparseUniva
--R lift : % -> SparseUnivariatePolynomial(%)
--R lift : (% ,%) -> SparseUnivariatePolynomial(%)
--R multiEuclidean : (List(% ),%) -> Union(List(% ),"failed")
--R newElement : (SparseUnivariatePolynomial(% ),%,Symbol) -> %
--R newElement : (SparseUnivariatePolynomial(% ),Symbol) -> %
--R nextItem : % -> Union(% ,"failed")
--R order : % -> OnePointCompletion(PositiveInteger)
--R primeFrobenius : (% ,NonNegativeInteger) -> %
--R principalIdeal : List(%) -> Record(coef: List(% ),generator: %)
--R reduce : SparseUnivariatePolynomial(%) -> %
--R representationType : () -> Union("prime",polynomial,normal,cyclic)
--R retractIfCan : % -> Union(K,"failed")
--R subtractIfCan : (% ,%) -> Union(% ,"failed")
--R tableForDiscreteLogarithm : Integer -> Table(PositiveInteger,NonNegativeInteger)
--R transcendenceDegree : () -> NonNegativeInteger
--R unitNormal : % -> Record(unit: % ,canonical: % ,associate: %)
--R
--E 1

)spool
)lisp (bye)

```

— PseudoAlgebraicClosureOfFiniteField.help —

```
=====
PseudoAlgebraicClosureOfFiniteField examples
=====
```

This domain implement dynamic extension using the simple notion of tower extensions. A tower extension T of the ground field K is any sequence of field extension

($T : K_0, K_1, \dots, K_i, \dots, K_n$)

where $K_0 = K$ and for $i = 1, 2, \dots, n$, K_i is an extension of K_{i-1} of degree > 1 and defined by an irreducible polynomial $p(Z)$ in K_{i-1} .

Two towers

($T_1: K_{01}, K_{11}, \dots, K_{i1}, \dots, K_{n1}$)

and

($T_2: K_{02}, K_{12}, \dots, K_{i2}, \dots, K_{n2}$)

are said to be related if

$T_1 \leq T_2$ (or $T_1 \geq T_2$),

that is if

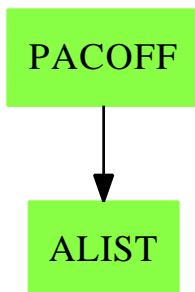
$K_{i1} = K_{i2}$ for $i=1, 2, \dots, n1$

(or $i=1, 2, \dots, n2$). Any algebraic operations defined for several elements are only defined if all of the concerned elements are coming from a set of related tower extensions.

See Also:

o)show PseudoAlgebraicClosureOfFiniteField

PseudoAlgebraicClosureOfFiniteField (PACOFF)



Exports:

0	1	-?
?**?	?*?	?+?
?-?	?/?	?=?
?^?	?~=?	?quo?
?rem?	D	algebraic?
associates?	characteristic	charthRoot
coerce	conditionP	conjugate
createPrimitiveElement	definingPolynomial	degree
differentiate	dimension	discreteLog
distinguishedRootsOf	divide	euclideanSize
expressIdealMember	exquo	extDegree
extendedEuclidean	extensionDegree	factor
factorsOfCyclicGroupSize	Frobenius	fullOutput
gcd	gcdPolynomial	ground?
hash	inGroundField?	index
init	inv	latex
lcm	lift	lookup
maxTower	multiEuclidean	newElement
nextItem	one?	order
previousTower	prime?	primeFrobenius
primitive?	primitiveElement	principalIdeal
random	recip	reduce
representationType	retract	retractIfCan
sample	setTower!	size
sizeLess?	squareFree	squareFreePart
subtractIfCan	tableForDiscreteLogarithm	transcendenceDegree
transcendent?	unit?	unitCanonical
unitNormal	vectorise	zero?

— domain PACOFF PseudoAlgebraicClosureOfFiniteField —

```

)abbrev domain PACOFF PseudoAlgebraicClosureOfFiniteField
++ Authors: Gaetan Hache
++ Date Created: june 1996
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ This domain implement dynamic extension using the simple notion of
++ tower extensions. A tower extension T of the ground field K is any
++ sequence of field extension (T : K_0, K_1, ..., K_i...,K_n) where K_0 = K
++ and for i =1,2,...,n, K_i is an extension of K_{i-1} of degree > 1 and
++ defined by an irreducible polynomial p(Z) in K_{i-1}.
++ Two towers (T_1: K_01, K_11,...,K_i1,...,K_n1)
++ and (T_2: K_02, K_12,...,K_i2,...,K_n2) are said to be related
++ if T_1 <= T_2 (or T_1 >= T_2), that is if K_i1 = K_i2 for i=1,2,...,n1
++ (or i=1,2,...,n2). Any algebraic operations defined for several elements
++ are only defined if all of the concerned elements are coming from
++ a set of related tower extensions.
PseudoAlgebraicClosureOfFiniteField(K):Exports == Implementation where

```

```

K:FiniteFieldCategory

INT      ==> Integer
NNI      ==> NonNegativeInteger
SUP      ==> SparseUnivariatePolynomial
BOOLEAN  ==> Boolean
PI       ==> PositiveInteger
FFFACTSE ==> FiniteFieldFactorizationWithSizeParseBySideEffect

recRep ==> Record(recEl:SUP(%),_
                  recTower:SUP(%),_
                  recDeg:PI,_
                  recPrevTower:%,_
                  recName:Symbol)

Exports == Join(PseudoAlgebraicClosureOfFiniteFieldCategory,_
                ExtensionField(K)) with

fullOutput: % -> OutputForm

Implementation == add
Rep := Union(recRep,K)

-- signature of local function
replaceRecEl: (% ,SUP(%)) -> %
down: % -> %
localRandom: % -> %
repPolynomial : % -> SUP(%)

replaceRecEl(a,el)==
  a case K => a
  aa:=copy a
  aa.recEl := el
  aa

-- local variable
localTower :% := 1$K

localSize :NNI := size()$K
-- implemetation of exported function

degree(a)==
  da:PositiveInteger:= extDegree a
  coerce(da@PositiveInteger)$OnePointCompletion(PositiveInteger)

repPolynomial(a)==
  a case K => error "Is in ground field"
  (a pretend recRep).recEl

inv(a)==

```

```

a case K => inv(a)$K
aRecEl:= repPolynomial a
aDefPoly:= definingPolynomial a
aInv := extendedEuclidean( aRecEl , aDefPoly, 1 )
aInv case "failed" => error "PACOFF : division by zero"
down replaceRecEl( a , aInv.coef1 )

a:% ** n:PositiveInteger ==
zero?(a) => 0
expt( a , n )$RepeatedSquaring(%)

a:% ** n:NonNegativeInteger ==
zero?(a) and zero?(n) => error " --- 0^0 not defined "
zero?(n) => 1$%
a ** ( n pretend PositiveInteger )

a:% ** n:Integer ==
n < 0 => inv( a ** ( (-n) pretend PositiveInteger) )
a ** ( n pretend NonNegativeInteger )

unitNormal(a)==
zero? a => [1,0,1]
[a,1,inv a]

ground?(a)== a case K

vectorise(a,lev)==
da:=extDegree a
dlev:=extDegree lev
dlev < da => _
error "Cannot vectorise at a lower level than the element to vectorise"
lev case K => [a]
pa:SUP(%)
na:%
^(da = dlev) =>
pa:= monomial(a,0)$SUP(%)
na:= replaceRecEl(lev,pa)
vectorise(na,lev)$%
prevLev:=previousTower(lev)
a case K => _
error "At this point a is not suppose to be in K, big error"
aEl:=(a pretend recRep).recEl
daEl:=degree(definingPolynomial a)$SUP(%)
lv:=[vectorise(c,prevLev)$% for c in entries(vectorise(aEl,daEl)$SUP(%))]
concat lv

size == localSize

setTower!(a) ==
localTower:=a

```

```

localSize:=(size()$K)**extDegree(a)
void()

localRandom(a) ==
  --return a random element at the extension of a
  a case K => random()$K
  subF:=previousTower(a)
  d:=degree(a.recTower)-1
  pol:=reduce("+",[monomial(localRandom(subF),i)$SUP(%) for i in 0..d])
  down replaceRecEl(a,pol)

a:% + b:% ==
  (a case K) and (b case K) => a +$K b
  extDegree(a) > extDegree(b) => b + a
  res1:SUP(%)
  res2:%
  if extDegree(a) = extDegree(b) then
    res1:= b.recEl +$SUP(%) a.recEl
    res2:= replaceRecEl(b,res1)
  else
    res1:= b.recEl +$SUP(%) monomial(a,0)$SUP(%)
    res2:= replaceRecEl(b,res1)
  down(res2)

a:% * b:% ==
  (a case K) and (b case K) => a *$K b
  extDegree(a) > extDegree(b) => b * a
  res1:SUP(%)
  res2:%
  if extDegree(a) = extDegree(b) then
    res1:= b.recEl *$SUP(%) a.recEl rem b.recTower
    res2:= replaceRecEl(b,res1)
  else
    res1:= b.recEl *$SUP(%) monomial(a,0)$SUP(%)
    res2:= replaceRecEl(b,res1)
  down(res2)

distinguishedRootsOf(polyZero,ee) ==
  setTower!(ee)
  zero?(polyZero) => error "to lazy to give you all the roots of 0 !!!"
  factorf: Factored SUP % := factor(polyZero)$FFFACTSE(%,SUP(%%))
  listFact:List SUP % := [pol.fctr for pol in factorList(factorf)]
  listOfZeros:List(%) := empty()
  for p in listFact repeat
    root:=newElement(p, new(D::Symbol)$Symbol)
    listOfZeros:List(%) := concat([ root ], listOfZeros)
  listOfZeros

random==
  localRandom(localTower)

```

```

extDegOfGrdField:PI :=
  i: PI := 1
  while characteristic()$K ** i < size()$K repeat
    i:= i + 1
  i

charthRoot(a : %): % ==
  --return a**(1/characteristic )
  a case K => charthRoot(retract a)$K
  b:NNI := extDegree(a) * extDegOfGrdField
  j := subtractIfCan(b,1)
  if (j case "failed") then b:= 0
  else b:= j
  c:= (characteristic()$K) ** b
  a**c

conjugate(a)==
  a ** size()$K

1 == 1$K

0 == 0$K

newElement(pol:SUP(%),subF:%,inName:Symbol): % ==
  -- pol is an irreducible polynomial over the field extension
  -- given by subF.
  -- The output of this function is a root of pol.
  dp:=degree pol
  one?(dp) =>
    listCoef:=coefficients(pol)
    one?(#listCoef) => 0
    - last(listCoef) / first(listCoef)
  ground?(pol) => error "Cannot create a new element with a constant"
  d:PI := (dp pretend PI) * extDegree(subF)
  [monomial(1$%,1),pol,d,subF,inName] :: Rep

newElement(poll:SUP(%),inName:Symbol)==
  newElement(poll,localTower,inName)

maxTower(la)==
  --return an element from the list la which is in the largest
  --extension of the ground field
  --PRECONDITION: all elements in same tower, else no meaning?
  m:=reduce("max",[extDegree(a) for a in la])
  first [b for b in la | extDegree(b)=m]

--Field operations

a:% / b:% == a * inv(b)

```

```

a:K * b:% ==
  (a :: %) * b

b:% * a:K == a*b

a:% - b:% ==
  a + (-b)

a:% * b:Fraction(Integer) ==
  bn:=numer b
  bd:=denom b
  ebn:%= bn * 1$%
  ebd:%= bd * 1$%
  a * ebn * inv(ebd)

-a:% ==
  a case K => -$K a
  [-$SUP(%) (a pretend recRep).recEl, _
   (a pretend recRep).recTower, _
   (a pretend recRep).recDeg, _
   (a pretend recRep).recPrevTower, _
   (a pretend recRep).recName ]

n:INT * a:% ==
  one?(n) => a
  zero?(a) or zero?(n) => 0
  (n < 0) => -((-n)*a)
  mm:PositiveInteger:=(n pretend PositiveInteger)
  double(mm,a)$RepeatedDoubling(%)

bb:% = aa:% ==
  b:=down bb
  a:=down aa
  ^( extDegree(b) =$NNI extDegree(a) ) => false
  (b case K) => ( (retract a) =$K (retract b) )
  rda := a :: recRep
  rdb := b :: recRep
  not (rda.recTower =$SUP(%) rdb.recTower) => false
  rdb.recEl =$SUP(%) rda.recEl

zero?(a:%) ==
  da:=down a -- just to be sure !!!
  ^(da case K) => false
  zero?(da)$K

one?(a:%) ==
  da:= down a -- just to be sure !!!
  ^(da case K) => false
  one?(da)$K

```



```

--Coerce Functions

coerce(a:K) == a

retractIfCan(a)==
  a case K => a
  "failed"

coerce(a:%):OutputForm ==
  a case K => (retract a)::OutputForm
  outputForm((a pretend recRep).recEl, _
    ((a pretend recRep).recName)::OutputForm) $SUP(%)

fullOutput(a:%):OutputForm==
  a case K => (retract a)::OutputForm
  (a pretend recRep)::OutputForm

definingPolynomial(a:%): SUP % ==
  a case K => 1
  (a pretend recRep).recTower

extDegree(a:%): PI ==
  a case K => 1
  (a pretend recRep).recDeg

previousTower(a:%):% ==
  a case K => error "No previous extension for ground field element"
  (a pretend recRep).recPrevTower

name(a:%):Symbol ==
  a case K => error "No name for ground field element"
  (a pretend recRep).recName

-- function related to the ground field

lookup(a:%)==
  aa:=down a
  ^(aa case K) => _
    error "From NonGlobalDynamicExtensionOfFiniteField fnc Lookup: Cannot take i-dex"
  lookup(retract aa)$K

index(i)==(index(i)$K)

fromPrimeField? == characteristic()$K = size()$K

representationType == representationType()$K

characteristic == characteristic()$K

```

```

-- implementation of local functions

down(a:%) ==
  a case K => a
  aa:=(a pretend recRep)
  el1 := aa.recEl
  ^ground?(el1) => a
  gel:]:=ground(el1)
  down(gel)

-----

-- PACOFF.dotabb --

"PACOFF" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PACOFF"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"PACOFF" -> "ALIST"

-----

```

domain PACRAT PseudoAlgebraicClosureOfRationalNumber

```

-- PseudoAlgebraicClosureOfRationalNumber.input --

)set break resume
)sys rm -f PseudoAlgebraicClosureOfRationalNumber.output
)spool PseudoAlgebraicClosureOfRationalNumber.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PseudoAlgebraicClosureOfRationalNumber
--R
--R PseudoAlgebraicClosureOfRationalNumber is a domain constructor
--R Abbreviation for PseudoAlgebraicClosureOfRationalNumber is PACRAT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for PACRAT
--R
--R----- Operations -----
--R ?? : (%,Fraction(Integer)) -> %      ?? : (Fraction(Integer),%) -> %
--R ?? : (Fraction(Integer),%) -> %      ?? : (%,Fraction(Integer)) -> %

```

```

--R ?? : (%,%) -> %
--R ?? : (PositiveInteger,%) -> %
--R ??? : (%,PositiveInteger) -> %
--R ?-? : (%,%) -> %
--R ?/? : (%,Fraction(Integer)) -> %
--R ?? : (%,%) -> Boolean
--R 0 : () -> %
--R ?? : (%,PositiveInteger) -> %
--R associates? : (%,%) -> Boolean
--R coerce : Fraction(Integer) -> %
--R coerce : Fraction(Integer) -> %
--R coerce : Integer -> %
--R conjugate : % -> %
--R extDegree : % -> PositiveInteger
--R fullOutput : % -> OutputForm
--R gcd : (%,%) -> %
--R hash : % -> SingleInteger
--R inv : % -> %
--R lcm : List(%) -> %
--R maxTower : List(%) -> %
--R previousTower : % -> %
--R ?quo? : (%,%) -> %
--R ?rem? : (%,%) -> %
--R retract : % -> Fraction(Integer)
--R sample : () -> %
--R sizeLess? : (%,%) -> Boolean
--R squareFreePart : % -> %
--R unit? : % -> Boolean
--R vectorise : (%,%) -> Vector(%)
--R ?^? : (%,%) -> Boolean
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R Frobenius : % -> % if Fraction(Integer) has FINITE
--R Frobenius : (%,NonNegativeInteger) -> % if Fraction(Integer) has FINITE
--R ?? : (%,NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if Fraction(Integer) has CHARNZ or Fraction(Integer) has
--R definingPolynomial : % -> SparseUnivariatePolynomial(%)
--R definingPolynomial : () -> SparseUnivariatePolynomial(%)
--R degree : % -> OnePointCompletion(PositiveInteger)
--R discreteLog : (%,%) -> Union(NonNegativeInteger, "failed") if Fraction(Integer) has CHARNZ or
--R distinguishedRootsOf : (SparseUnivariatePolynomial(%),%) -> List(%)
--R divide : (%,%) -> Record(quotient: %,remainder: %)
--R euclideanSize : % -> NonNegativeInteger
--R expressIdealMember : (List(%),%) -> Union(List(%), "failed")
--R exquo : (%,%) -> Union(%, "failed")
--R extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %), "failed")
--R extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %)
--R extensionDegree : () -> OnePointCompletion(PositiveInteger)
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUniva
?? : (Integer,%) -> %
??? : (%,Integer) -> %
?+? : (%,%) -> %
-? : % -> %
?/? : (%,%) -> %
1 : () -> %
?^? : (%,Integer) -> %
algebraic? : % -> Boolean
coerce : Fraction(Integer) -> %
coerce : Integer -> %
coerce : % -> %
coerce : % -> OutputForm
dimension : () -> CardinalNumber
factor : % -> Factored(%)
gcd : List(%) -> %
ground? : % -> Boolean
inGroundField? : % -> Boolean
latex : % -> String
lcm : (%,%) -> %
one? : % -> Boolean
prime? : % -> Boolean
recip : % -> Union(%, "failed")
retract : % -> Fraction(Integer)
retract : % -> Integer
setTower! : % -> Void
squareFree : % -> Factored(%)
transcendent? : % -> Boolean
unitCanonical : % -> %
zero? : % -> Boolean

```

```

--R lift : (%,% ) -> SparseUnivariatePolynomial(%)
--R lift : % -> SparseUnivariatePolynomial(%)
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed")
--R newElement : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%),PositiveInteger,%,Symbol) -> %
--R newElement : (SparseUnivariatePolynomial(%),Symbol) -> %
--R newElement : (SparseUnivariatePolynomial(%),%,Symbol) -> %
--R order : % -> OnePointCompletion(PositiveInteger) if Fraction(Integer) has CHARNZ or Fraction(Integer) has FINITE
--R primeFrobenius : % -> % if Fraction(Integer) has CHARNZ or Fraction(Integer) has FINITE
--R primeFrobenius : (% ,NonNegativeInteger) -> % if Fraction(Integer) has CHARNZ or Fraction(Integer) has FINITE
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %)
--R reduce : SparseUnivariatePolynomial(%) -> %
--R retractIfCan : % -> Union(Fraction(Integer),"failed")
--R retractIfCan : % -> Union(Fraction(Integer),"failed")
--R retractIfCan : % -> Union(Integer,"failed")
--R subtractIfCan : (% ,%) -> Union(%,"failed")
--R transcendenceDegree : () -> NonNegativeInteger
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R
--E 1

)spool
)lisp (bye)

```

— PseudoAlgebraicClosureOfRationalNumber.help —

```

=====
PseudoAlgebraicClosureOfRationalNumber examples
=====

```

This domain implements dynamic extension using the simple notion of tower extensions. A tower extension T of the ground field K is any sequence of field extension

$(T : K_0, K_1, \dots, K_{i-1}, K_i, \dots, K_n)$

where $K_0 = K$ and for $i = 1, 2, \dots, n$, K_i is an extension of K_{i-1} of degree > 1 and defined by an irreducible polynomial $p(Z)$ in K_{i-1} .

Two towers

$(T_1: K_{01}, K_{11}, \dots, K_{i1}, \dots, K_{n1})$

and

$(T_2: K_{02}, K_{12}, \dots, K_{i2}, \dots, K_{n2})$

are said to be related if

$T_1 \leq T_2$ (or $T_1 \geq T_2$),

that is if

$K_{i1} = K_{i2}$ for $i=1, 2, \dots, n1$ (or $i=1, 2, \dots, n2$).

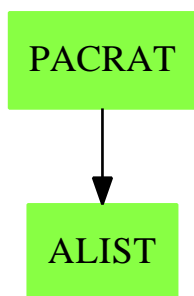
Any algebraic operations defined for several elements are only defined if all of the concerned elements are coming from a set of related tower

extensions.

See Also:

o)show PseudoAlgebraicClosureOfRationalNumber

PseudoAlgebraicClosureOfRationalNumber (PACRAT)



Exports:

0	0	1	-?	***?
?*?	?+?	?-?	?/?	
?=?	?^?	?~=?	?quo?	
?rem?	algebraic?	associates?	characteristic	
charthRoot	coerce	conjugate	definingPolynomial	
degree	dimension	discreteLog	distinguishedRootsOf	
divide	euclideanSize	expressIdealMember	exquo	
extDegree	extendedEuclidean	extensionDegree	factor	
Frobenius	fullOutput	gcd	gcdPolynomial	
ground?	hash	inGroundField?	inv	
latex	lcm	lift	maxTower	
multiEuclidean	newElement	one?	order	
previousTower	prime?	primeFrobenius	principalIdeal	
recip	reduce	retract	retractIfCan	
sample	setTower!	sizeLess?	squareFree	
squareFreePart	subtractIfCan	transcendenceDegree	transcendent?	
unit?	unitCanonical	unitNormal	vectorise	
zero?				

— domain PACRAT PseudoAlgebraicClosureOfRationalNumber —

)abbrev domain PACRAT PseudoAlgebraicClosureOfRationalNumber

```

++ Authors: Gaetan Hache
++ Date Created: feb 1997
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ This domain implements dynamic extension using the simple notion of
++ tower extensions. A tower extension T of the ground field K is any
++ sequence of field extension (T : K_0, K_1, ..., K_i...,K_n) where K_0 = K
++ and for i =1,2,...,n, K_i is an extension of K_{i-1} of degree > 1 and
++ defined by an irreducible polynomial p(Z) in K_{i-1}.
++ Two towers (T_1: K_01, K_11,...,K_i1,...,K_n1) and
++ (T_2: K_02, K_12,...,K_i2,...,K_n2) are said to be related if T_1 <= T_2
++ (or T_1 >= T_2), that is if K_i1 = K_i2 for i=1,2,...,n1
++ (or i=1,2,...,n2). Any algebraic operations defined for several elements
++ are only defined if all of the concerned elements are coming from
++ a set of related tower extensions.
PseudoAlgebraicClosureOfRationalNumber:Exports == Implementation where

INT      ==> Integer
K        ==> Fraction Integer
NNI      ==> NonNegativeInteger
SUP      ==> SparseUnivariatePolynomial
BOOLEAN ==> Boolean
PI       ==> PositiveInteger
FACTRN   ==> FactorisationOverPseudoAlgebraicClosureOfRationalNumber

recRep   ==> Record(recEl:SUP(%),_
                    recTower:SUP(%),_
                    recDeg:PI,_
                    recPrevTower:%,_
                    recName:Symbol)

Exports == PseudoAlgebraicClosureOfRationalNumberCategory with

fullOutput: % -> OutputForm

newElement: (SUP(%), SUP(%), PI, %, Symbol) -> %

Implementation == add
Rep := Union(recRep,K)

-- signature of local function
replaceRecEl: (% ,SUP(%)) -> %
down: % -> %

down(a:%) ==
a case K => a
aa:=(a pretend recRep)
elel := aa.recEl
^ground?(elel)$SUP(%) => a
gel: %:=ground(elel)

```

```

down(gel)

coerce(a:Integer):%== (a :: K)

n:INT * a:% ==
  one?(n) => a
  zero?(a) or zero?(n) => 0
  (n < 0) => -((-n)*a)
mm:PositiveInteger:=(n pretend PositiveInteger)
double(mm,a)$RepeatedDoubling(%)

replaceRecEl(a,el)==
  a case K => a
  aa:=copy a
  aa.recEl := el
  aa

-- local variable
localTower :% := 1$K

-- implemetation of exported function

lift(a) ==
  a case K => monomial(a,0)
  (a pretend recRep).recEl

lift(a,b)==
  extDegree a > extDegree b => _
    error "Cannot lift something at lower level !!!!!"
  extDegree a < extDegree b => monomial(a,0)$SUP(%)
  lift a

reduce(a)==
  localTower case K =>
    coefficient(a,0)
  ar:= a rem (localTower pretend recRep).recTower
  replaceRecEl(localTower,ar)

maxTower(la)==
  --return an element from the list la which is in the largest
  --extension of the ground field
  --PRECONDITION: all elements in same tower, else no meaning?
  m:="max"/[extDegree(a)$% for a in la]
  first [b for b in la | extDegree(b)=m]

ground?(a)== a case K

vectorise(a,lev)==
  da:=extDegree a
  dlev:=extDegree lev

```

```

dlev < da => _
  error "Cannot vectorise at a lower level than the element to vectorise"
lev case K => [a]
pa:SUP(%)
na:%
^(da = dlev) =>
  pa:= monomial(a,0)$SUP(%)
  na:= replaceRecEl(lev,pa)
  vectorise(na,lev)$%
prevLev:=previousTower(lev)
a case K => error "At this point a is not suppose to be in K"
aEl:=(a pretend recRep).recEl
daEl:=degree definingPolynomial(a)$%
lv:=[vectorise(c,prevLev)$% for c in entries(vectorise(aEl,daEl)$SUP(%))]]
concat lv

setTower!(a) ==
  localTower:=a
  void()

definingPolynomial == definingPolynomial(localTower)

a:% + b:% ==
  (a case K) and (b case K) => a +$K b
  extDegree(a) > extDegree(b) => b + a
  res1:SUP(%)
  res2:%
  if extDegree(a) = extDegree(b) then
    res1:= b.recEl +$SUP(%) a.recEl
    res2:= replaceRecEl(b,res1)
  else
    res1:= b.recEl +$SUP(%) monomial(a,0)$SUP(%)
    res2:= replaceRecEl(b,res1)
  down(res2)

a:% * b:% ==
  (a case K) and (b case K) => a *$K b
  extDegree(a) > extDegree(b) => b * a
  res1:SUP(%)
  res2:%
  if extDegree(a) = extDegree(b) then
    res1:= b.recEl *$SUP(%) a.recEl rem b.recTower
    res2:= replaceRecEl(b,res1)
  else
    res1:= b.recEl *$SUP(%) monomial(a,0)$SUP(%)
    res2:= replaceRecEl(b,res1)
  down(res2)

distinguishedRootsOf(polyZero,ee) ==
  setTower!(ee)

```



```

zero?(polyZero) => error "to lazy to give you all the roots of 0 !!!"
factorf: Factored SUP % := factor(polyZero,ee)$FACTRN(%)
listFact:List SUP % := [pol.fctr for pol in factorList(factorf)]
listOfZeros:List(%):=empty()
for p in listFact repeat
  root:=newElement(p, new(D::Symbol)$Symbol)
  listOfZeros:List(%):=concat([ root ], listOfZeros)
listOfZeros

1 == 1$K

0 == 0$K

newElement(pol:SUP(%),subF:%,inName:Symbol): % ==
  -- pol is an irreducible polynomial over the field extension
  -- given by subF.
  -- The output of this function is a root of pol.
dp:=degree pol
one?(dp) =>
  listCoef:=coefficients(pol)
  one?(#listCoef) => 0
  - last(listCoef) / first(listCoef)
ground?(pol) => error "Cannot create a new element with a constant"
d:PI := (dp pretend PI) * extDegree(subF)
[monomial(1$%,1),pol,d,subF,inName] :: Rep

newElement(poll:SUP(%),inName:Symbol)==
  newElement(poll,localTower,inName)

newElement(elPol:SUP(%),pol:SUP(%),d:PI,subF:%,inName:Symbol): % ==
  [elPol, pol,d,subF,inName] :: Rep

--Field operations
inv(a)==
  a case K => inv(a)$K
  aRecEl:= (a pretend recRep).recEl
  aDefPoly:= (a pretend recRep).recTower
  aInv := extendedEuclidean( aRecEl , aDefPoly, 1 )
  aInv case "failed" => error "PACOFF : division by zero"
  -- On doit retourner un Record representant l'inverse de a.
  -- Ce Record est exactement le mme que celui de a sauf
  -- qu'il faut remplacer le polynme du selecteur recEl
  -- par le polynme representant l'inverse de a :
  -- C'est ce que fait la fonction replaceRecEl.
  replaceRecEl( a , aInv.coef1 )

a:% / b:% == a * inv(b)

a:K * b:%==
  (a :: %) * b

```

```

b:% * a:K == a*b

a:% - b:% ==
  a + (-b)

a:% * b:Fraction(Integer) ==
  bn:=numer b
  bd:=denom b
  ebn:%= bn * 1$%
  ebd:%= bd * 1$%
  a * ebn * inv(ebd)

-a:% ==
  a case K => -$K a
  [-$SUP(%) (a pretend recRep).recEl, _
   (a pretend recRep).recTower, _
   (a pretend recRep).recDeg, _
   (a pretend recRep).recPrevTower, _
   (a pretend recRep).recName ]

bb:% = aa:% ==
  b:=down bb
  a:=down aa
  ^(extDegree(b)=$NNI extDegree(a)) => false
  (b case K) => ( (retract a)@K = $K (retract b)@K )
  rda := a :: recRep
  rdb := b :: recRep
  not (rda.recTower = $SUP(%) rdb.recTower) => false
  rdb.recEl = $SUP(%) rda.recEl

zero?(a:%) ==
  da:=down a -- just to be sure !!!
  ^(da case K) => false
  zero?(da)$K

one?(a:%) ==
  da:= down a -- just to be sure !!!
  ^(da case K) => false
  one?(da)$K

--Coerce Functions

coerce(a:K):% == a

retractIfCan(a:%):Union(Integer,"failed")==
  a case K => retractIfCan(a)$K
  "failed"

retractIfCan(a:%):Union(K,"failed")==

```

```

a case K => a
"failed"

coerce(a: %): OutputForm ==
a case K => ((retract a)@K) :: OutputForm
outputForm((a pretend recRep).recEl, _
            ((a pretend recRep).recName)::OutputForm) $SUP(%)

fullOutput(a: %): OutputForm ==
a case K => ((retract a)@K) :: OutputForm
(a pretend recRep)::OutputForm

definingPolynomial(a: %): SUP % ==
a case K => monomial(1,1)$SUP(%)
(a pretend recRep).recTower

extDegree(a: %): PI ==
a case K => 1
(a pretend recRep).recDeg

previousTower(a: %): % ==
a case K => error "No previous extension for ground field element"
(a pretend recRep).recPrevTower

name(a: %): Symbol ==
a case K => error "No name for ground field element"
(a pretend recRep).recName

-- function related to the ground field

characteristic == characteristic()$K

```

— PACRAT.dotabb —

```

"PACRAT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=PRODUCT"]
"ALIST"  [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"PACRAT" -> "ALIST"

```

Chapter 18

Chapter Q

domain QFORM QuadraticForm

— QuadraticForm.input —

```
)set break resume
)sys rm -f QuadraticForm.output
)spool QuadraticForm.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show QuadraticForm
--R QuadraticForm(n: PositiveInteger,K: Field) is a domain constructor
--R Abbreviation for QuadraticForm is QFORM
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for QFORM
--R
--R----- Operations -----
--R ??? : (Integer,%) -> %          ??? : (PositiveInteger,%) -> %
--R ?+? : (%,%) -> %              ?-? : (%,%) -> %
--R -? : % -> %                  ?=? : (%,%) -> Boolean
--R 0 : () -> %                  coerce : % -> OutputForm
--R ?.? : (%,DirectProduct(n,K)) -> K  hash : % -> SingleInteger
--R latex : % -> String            matrix : % -> SquareMatrix(n,K)
--R sample : () -> %              zero? : % -> Boolean
--R ?~=? : (%,%) -> Boolean
--R ??? : (NonNegativeInteger,%) -> %
--R quadraticForm : SquareMatrix(n,K) -> %
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R
```

```
--E 1

)spool
)lisp (bye)
```

— QuadraticForm.help —

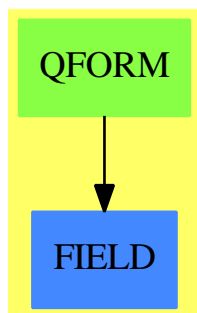
```
=====
QuadraticForm examples
=====
```

This domain provides modest support for quadratic forms.

See Also:

o)show QuadraticForm

QuadraticForm (QFORM)



See

⇒ “CliffordAlgebra” (CLIF) [4](#) on page [401](#)

Exports:

0	coerce	hash	latex	matrix
quadraticForm	sample	subtractIfCan	zero?	?~=?
?*?	?.?	?+?	?-?	-?
?=?				

— domain QFORM QuadraticForm —

```
)abbrev domain QFORM QuadraticForm
++ Author: Stephen M. Watt
```

```

++ Date Created: August 1988
++ Date Last Updated: May 17, 1991
++ Description:
++ This domain provides modest support for quadratic forms.

```

```

QuadraticForm(n, K): T == Impl where
  n: PositiveInteger
  K: Field
  SM ==> SquareMatrix
  V ==> DirectProduct

  T ==> AbelianGroup with
    quadraticForm: SM(n, K) -> %
      ++ quadraticForm(m) creates a quadratic form from a symmetric,
      ++ square matrix m.
    matrix: % -> SM(n, K)
      ++ matrix(qf) creates a square matrix from the quadratic form qf.
    elt: (% , V(n, K)) -> K
      ++ elt(qf,v) evaluates the quadratic form qf on the vector v,
      ++ producing a scalar.

  Impl ==> SM(n,K) add
    Rep := SM(n,K)

    quadraticForm m ==
      not symmetric? m =>
        error "quadraticForm requires a symmetric matrix"
      m::%
    matrix q == q pretend SM(n,K)
    elt(q,v) == dot(v, (matrix q * v))

```

— QFORM.dotabb —

```

"QFORM" [color="#88FF44",href="bookvol10.3.pdf#nameddest=QFORM"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"QFORM" -> "FIELD"

```

domain QALGSET QuasiAlgebraicSet

— QuasiAlgebraicSet.input —

```

)set break resume
)sys rm -f QuasiAlgebraicSet.output
)spool QuasiAlgebraicSet.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show QuasiAlgebraicSet
--R
--R QuasiAlgebraicSet(R: GcdDomain,Var: OrderedSet,Expon: OrderedAbelianMonoidSup,Dpoly: Polynom
--R Abbreviation for QuasiAlgebraicSet is QALGSET
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for QALGSET
--R
--R----- Operations -----
--R ?? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R definingInequation : % -> Dpoly    empty : () -> %
--R empty? : % -> Boolean              hash : % -> SingleInteger
--R idealSimplify : % -> %             latex : % -> String
--R ?~=? : (%,% ) -> Boolean
--R definingEquations : % -> List(Dpoly)
--R quasiAlgebraicSet : (List(Dpoly),Dpoly) -> %
--R setStatus : (% ,Union(Boolean,"failed")) -> %
--R simplify : % -> % if R has CHARZ and R has EUCDOM
--R status : % -> Union(Boolean,"failed")
--R
--E 1

)spool
)lisp (bye)

```

— QuasiAlgebraicSet.help —

=====

QuasiAlgebraicSet examples

=====

QuasiAlgebraicSet constructs a domain representing quasi-algebraic sets, which is the intersection of a Zariski closed set, defined as the common zeros of a given list of polynomials (the defining polynomials for equations), and a principal Zariski open set, defined as the complement of the common zeros of a polynomial f (the defining polynomial for the inequation).

This domain provides simplification of a user-given representation using groebner basis computations.

There are two simplification routines: the first function `idealSimplify` uses groebner basis of ideals alone, while the second, `simplify` uses both groebner basis and factorization. The resulting defining equations `L` always form a groebner basis, and the resulting defining inequation `f` is always reduced. The function `simplify` may be applied several times if desired. A third simplification routine `radicalSimplify` is provided in `QuasiAlgebraicSet2` for comparison study only, as it is inefficient compared to the other two, as well as is restricted to only certain coefficient domains. For detail analysis and a comparison of the three methods, please consult the reference cited.

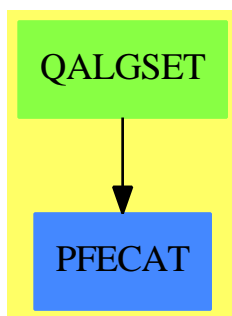
A polynomial function `q` defined on the quasi-algebraic set is equivalent to its reduced form with respect to `L`. While this may be obtained using the usual normal form algorithm, there is no canonical form for `q`.

The ordering in groebner basis computation is determined by the data type of the input polynomials. If it is possible we suggest to use refinements of total degree orderings.

See Also:

o `)show QuasiAlgebraicSet`

QuasiAlgebraicSet (QALGSET)



Exports:

<code>coerce</code>	<code>definingEquations</code>	<code>definingInequation</code>	<code>empty</code>
<code>empty?</code>	<code>hash</code>	<code>idealSimplify</code>	<code>latex</code>
<code>quasiAlgebraicSet</code>	<code>setStatus</code>	<code>simplify</code>	<code>status</code>
<code>?=?</code>	<code>?~=?</code>		

— domain QALGSET QuasiAlgebraicSet —


```

)abbrev domain QALGSET QuasiAlgebraicSet
++ Author: William Sit
++ Date Created: March 13, 1992
++ Date Last Updated: June 12, 1992
++ References: William Sit, "An Algorithm for Parametric Linear Systems"
++             J. Sym. Comp., April, 1992
++ Description:
++ \spadtype{QuasiAlgebraicSet} constructs a domain representing
++ quasi-algebraic sets, which is the intersection of a Zariski
++ closed set, defined as the common zeros of a given list of
++ polynomials (the defining polynomials for equations), and a principal
++ Zariski open set, defined as the complement of the common
++ zeros of a polynomial f (the defining polynomial for the inequation).
++ This domain provides simplification of a user-given representation
++ using groebner basis computations.
++ There are two simplification routines: the first function
++ \spadfun{idealSimplify} uses groebner
++ basis of ideals alone, while the second, \spadfun{simplify} uses both
++ groebner basis and factorization. The resulting defining equations L
++ always form a groebner basis, and the resulting defining
++ inequation f is always reduced. The function \spadfun{simplify} may
++ be applied several times if desired. A third simplification
++ routine \spadfun{radicalSimplify} is provided in
++ \spadtype{QuasiAlgebraicSet2} for comparison study only,
++ as it is inefficient compared to the other two, as well as is
++ restricted to only certain coefficient domains. For detail analysis
++ and a comparison of the three methods, please consult the reference
++ cited.
++
++ A polynomial function q defined on the quasi-algebraic set
++ is equivalent to its reduced form with respect to L. While
++ this may be obtained using the usual normal form
++ algorithm, there is no canonical form for q.
++
++ The ordering in groebner basis computation is determined by
++ the data type of the input polynomials. If it is possible
++ we suggest to use refinements of total degree orderings.

QuasiAlgebraicSet(R, Var, Expon, Dpoly) : C == T
where
  R      : GcdDomain
  Expon   : OrderedAbelianMonoidSup
  Var     : OrderedSet
  Dpoly   : PolynomialCategory(R, Expon, Var)
  NNI     ==> NonNegativeInteger
  newExpon ==> Product(NNI, Expon)
  newPoly ==> PolynomialRing(R, newExpon)
  Ex      ==> OutputForm
  mrf     ==> MultivariateFactorize(Var, Expon, R, Dpoly)
  Status  ==> Union(Boolean, "failed") -- empty or not, or don't know

```

```

C == Join(SetCategory, CoercibleTo OutputForm) with
--- should be Object instead of SetCategory, bug in LIST Object ---
--- equality is not implemented ---
empty: () -> $
  ++ empty() returns the empty quasi-algebraic set
quasiAlgebraicSet: (List Dpoly, Dpoly) -> $
  ++ quasiAlgebraicSet(pl,q) returns the quasi-algebraic set
  ++ with defining equations  $p = 0$  for  $p$  belonging to the list pl, and
  ++ defining inequation  $q \neq 0$ .
status: $ -> Status
  ++ status(s) returns true if the quasi-algebraic set is empty,
  ++ false if it is not, and "failed" if not yet known
setStatus: ($, Status) -> $
  ++ setStatus(s,t) returns the same representation for s, but
  ++ asserts the following: if t is true, then s is empty,
  ++ if t is false, then s is non-empty, and if t = "failed",
  ++ then no assertion is made (that is, "don't know").
  ++ Note: for internal use only, with care.
empty?      : $ -> Boolean
  ++ empty?(s) returns
  ++ true if the quasialgebraic set s has no points,
  ++ and false otherwise.
definingEquations: $ -> List Dpoly
  ++ definingEquations(s) returns a list of defining polynomials
  ++ for equations, that is, for the Zariski closed part of s.
definingInequation: $ -> Dpoly
  ++ definingInequation(s) returns a single defining polynomial for the
  ++ inequation, that is, the Zariski open part of s.
idealSimplify:$ -> $
  ++ idealSimplify(s) returns a different and presumably simpler
  ++ representation of s with the defining polynomials for the
  ++ equations
  ++ forming a groebner basis, and the defining polynomial for the
  ++ inequation reduced with respect to the basis, using Buchberger's
  ++ algorithm.
if (R has EuclideanDomain) and (R has CharacteristicZero) then
  simplify:$ -> $
    ++ simplify(s) returns a different and presumably simpler
    ++ representation of s with the defining polynomials for the
    ++ equations
    ++ forming a groebner basis, and the defining polynomial for the
    ++ inequation reduced with respect to the basis, using a heuristic
    ++ algorithm based on factoring.
T == add
Rep := Record(status:Status,zero:List Dpoly, nzero:Dpoly)
x:$

import GroebnerPackage(R,Expon,Var,Dpoly)
import GroebnerPackage(R,newExpon,Var,newPoly)

```

```

import GroebnerInternalPackage(R,Expon,Var,Dpoly)

----- Local Functions -----

minset    : List List Dpoly -> List List Dpoly
overset?  : (List Dpoly, List List Dpoly) -> Boolean
npoly     : Dpoly           -> newPoly
oldpoly   : newPoly         -> Union(Dpoly,"failed")

if (R has EuclideanDomain) and (R has CharacteristicZero) then
  factorset (y:Dpoly):List Dpoly ==
    ground? y => []
    [j.factor for j in factors factor$mrf y]

simplify x ==
  if x.status case "failed" then
    x:=quasiAlgebraicSet(zro:=groebner x.zero, redPol(x.nzero,zro))
    (pnzero:=x.nzero)=0 => empty()
    nzro:=factorset pnzero
    mset:=minset [factorset p for p in x.zero]
    mset:=[setDifference(s,nzro) for s in mset]
    zro:=groebner [*s for s in mset]
    member? (1$Dpoly, zro) => empty()
    [x.status, zro, primitivePart redPol(*nzro, zro)]

npoly(f:Dpoly) : newPoly ==
  zero? f => 0
  monomial(leadingCoefficient f,makeprod(0,degree f))$newPoly +
    npoly(reductum f)

oldpoly(q:newPoly) : Union(Dpoly,"failed") ==
  q=0$newPoly => 0$Dpoly
  dq:newExpon:=degree q
  n:NNI:=selectfirst (dq)
  n^=0 => "failed"
  ((g:=oldpoly reductum q) case "failed") => "failed"
  monomial(leadingCoefficient q,selectsecond dq)$Dpoly + (g::Dpoly)

coerce x ==
  x.status = true => "Empty":Ex
  bracket [[hconcat(f::Ex, " = 0":Ex) for f in x.zero ]::Ex,
    hconcat( x.nzero::Ex, " != 0":Ex)]

empty? x ==
  if x.status case "failed" then x:=idealSimplify x
  x.status :: Boolean

empty() == [true::Status, [1$Dpoly], 0$Dpoly]
status x == x.status

```

```

setStatus(x,t) == [t,x.zero,x.nzero]
definingEquations x == x.zero
definingInequation x == x.nzero
quasiAlgebraicSet(z0,n0) == ["failed", z0, n0]

idealSimplify x ==
  x.status case Boolean => x
  z0:= x.zero
  n0:= x.nzero
  empty? z0 => [false, z0, n0]
  member? (1$Dpoly, z0) => empty()
  tp:newPoly:=(monomial(1,makeprod(1,0$Expon))$newPoly * npoly n0)-1
  ngb:=groebner concat(tp, [npoly g for g in z0])
  member? (1$newPoly, ngb) => empty()
  gb:List Dpoly:=nil
  while not empty? ngb repeat
    if ((f:=oldpoly ngb.first) case Dpoly) then gb:=concat(f, gb)
    ngb:=ngb.rest
  [false::Status, gb, primitivePart redPol(n0, gb)]

minset lset ==
  empty? lset => lset
  [s for s in lset | ^(overSet?(s,lset))]

overSet?(p,qList) ==
  empty? qList => false
  or/[(brace$(Set Dpoly) q) <$(Set Dpoly) (brace$(Set Dpoly) p) for q in qList]

```

— QALGSET.dotabb —

```

"QALGSET" [color="#88FF44",href="bookvol10.3.pdf#nameddest=QALGSET"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"QALGSET" -> "PFECAT"

```

domain QUAT Quaternion

— Quaternion.input —

```

)set break resume

```

```

)sys rm -f Quaternion.output
)spool Quaternion.output
)set message test on
)set message auto off
)clear all
--S 1 of 13
q := quatern(2/11,-8,3/4,1)
--R
--R
--R      2      3
--R      (1)  -- - 8i + - j + k
--R      11      4
--R
--R                                          Type: Quaternion(Fraction(Integer))
--E 1

--S 2 of 13
[real q, imagI q, imagJ q, imagK q]
--R
--R
--R      2      3
--R      (2)  [--,- 8,-,1]
--R      11      4
--R
--R                                          Type: List(Fraction(Integer))
--E 2

--S 3 of 13
inv q
--R
--R
--R      352      15488      484      1936
--R      (3)  ----- + ----- i - ----- j - ----- k
--R      126993   126993   42331   126993
--R
--R                                          Type: Quaternion(Fraction(Integer))
--E 3

--S 4 of 13
q^6
--R
--R
--R      2029490709319345   48251690851   144755072553   48251690851
--R      (4)  - ----- - ----- i + ----- j + ----- k
--R      7256313856        1288408        41229056        10307264
--R
--R                                          Type: Quaternion(Fraction(Integer))
--E 4

--S 5 of 13
r := quatern(-2,3,23/9,-89); q + r
--R
--R
--R      20      119

```

```

--R (5) 
$$-\frac{5}{11}i + \frac{88}{36}j - 88k$$

--R
--R                                         Type: Quaternion(Fraction(Integer))
--E 5

--S 6 of 13
q * r - r * q
--R
--R
--R (6) 
$$-\frac{2495}{18}i - 1418j - \frac{817}{18}k$$

--R
--R                                         Type: Quaternion(Fraction(Integer))
--E 6

--S 7 of 13
i:=quatern(0,1,0,0)
--R
--R
--R (7) i
--R
--R                                         Type: Quaternion(Integer)
--E 7

--S 8 of 13
j:=quatern(0,0,1,0)
--R
--R
--R (8) j
--R
--R                                         Type: Quaternion(Integer)
--E 8

--S 9 of 13
k:=quatern(0,0,0,1)
--R
--R
--R (9) k
--R
--R                                         Type: Quaternion(Integer)
--E 9

--S 10 of 13
[i*i, j*j, k*k, i*j, j*k, k*i, q*i]
--R
--R
--R (10) 
$$[-1, -1, -1, k, i, j, 8 + \frac{2}{11}i + \frac{3}{4}j - k]$$

--R
--R                                         Type: List(Quaternion(Fraction(Integer)))
--E 10

--S 11 of 13

```

```

norm q
--R
--R
--R      126993
--R (11)  -----
--R      1936
--R
--R                                          Type: Fraction(Integer)
--E 11

--S 12 of 13
conjugate q
--R
--R
--R      2      3
--R (12)  -- + 8i - - j - k
--R      11      4
--R
--R                                          Type: Quaternion(Fraction(Integer))
--E 12

--S 13 of 13
q * %
--R
--R
--R      126993
--R (13)  -----
--R      1936
--R
--R                                          Type: Quaternion(Fraction(Integer))
--E 13
)spool
)lisp (bye)

```

— Quaternion.help —

Quaternion examples

The domain constructor Quaternion implements quaternions over commutative rings.

The basic operation for creating quaternions is `quatern`. This is a quaternion over the rational numbers.

```

q := quatern(2/11,-8,3/4,1)
      2      3
-- - 8i + - j + k
      11      4

```

Type: Quaternion Fraction Integer

The four arguments are the real part, the i imaginary part, the j imaginary part, and the k imaginary part, respectively.

```
[real q, imagI q, imagJ q, imagK q]
      2      3
[--, - 8, -, 1]
     11     4
```

Type: List Fraction Integer

Because q is over the rationals (and nonzero), you can invert it.

```
inv q
      352      15488      484      1936
----- + ----- i - ----- j - ----- k
126993  126993  42331  126993
Type: Quaternion Fraction Integer
```

The usual arithmetic (ring) operations are available

```
q^6
      2029490709319345      48251690851      144755072553      48251690851
- ----- - ----- i + ----- j + ----- k
      7256313856      1288408      41229056      10307264
Type: Quaternion Fraction Integer
```

```
r := quatern(-2,3,23/9,-89); q + r
      20      119
- -- - 5i + --- j - 88k
     11      36
Type: Quaternion Fraction Integer
```

In general, multiplication is not commutative.

```
q * r - r * q
      2495      817
- ---- i - 1418j - --- k
      18      18
Type: Quaternion Fraction Integer
```

There are no predefined constants for the imaginary i, j, and k parts, but you can easily define them.

```
i:=quatern(0,1,0,0)
i
Type: Quaternion Integer
```

```
j:=quatern(0,0,1,0)
j
```


Type: Quaternion Integer

```
k:=quatern(0,0,0,1)
k
```

Type: Quaternion Integer

These satisfy the normal identities.

```
[i*i, j*j, k*k, i*j, j*k, k*i, q*i]
      2      3
[- 1,- 1,- 1,k,i,j,8 + -- i + j - - k]
      11      4
```

Type: List Quaternion Fraction Integer

The norm is the quaternion times its conjugate.

```
norm q
126993
-----
1936
```

Type: Fraction Integer

```
conjugate q
      2      3
-- + 8i - - j - k
      11      4
```

Type: Quaternion Fraction Integer

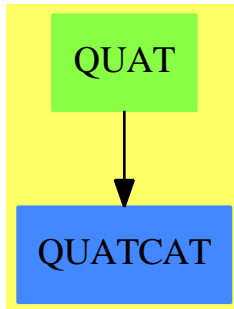
```
q * %
126993
-----
1936
```

Type: Quaternion Fraction Integer

See Also:

- o)help Octonion
- o)help Complex
- o)help CliffordAlgebra
- o)show Quaternion

Quaternion (QUAT)



Exports:

0	1	abs	characteristic	charthRoot
coerce	conjugate	convert	D	differentiate
eval	hash	imagI	imagJ	imagK
inv	latex	map	max	min
norm	one?	quatern	rational	rational?
rationalIfCan	real	recip	reducedSystem	retract
retractIfCan	sample	subtractIfCan	zero?	?*?
?**?	?+?	?-?	-?	?=?
?^?	?~=?	?<?	?<=?	?>?
?>=?	?..?			

— domain QUAT Quaternion —

```

)abbrev domain QUAT Quaternion
++ Author: Robert S. Sutor
++ Date Created: 23 May 1990
++ Change History: 10 September 1990
++ Description:
++ \spadtype{Quaternion} implements quaternions over a
++ commutative ring. The main constructor function is \spadfun{quatern}
++ which takes 4 arguments: the real part, the i imaginary part, the j
++ imaginary part and the k imaginary part.

```

```

Quaternion(R:CommutativeRing): QuaternionCategory(R) == add
  Rep := Record(r:R,i:R,j:R,k:R)

```

```

0 == [0,0,0,0]
1 == [1,0,0,0]

```

```

a,b,c,d : R
x,y : $

```

```

real x == x.r
imagI x == x.i

```

```

imagJ x == x.j
imagK x == x.k

quatern(a,b,c,d) == [a,b,c,d]

x * y == [x.r*y.r-x.i*y.i-x.j*y.j-x.k*y.k,
          x.r*y.i+x.i*y.r+x.j*y.k-x.k*y.j,
          x.r*y.j+x.j*y.r+x.k*y.i-x.i*y.k,
          x.r*y.k+x.k*y.r+x.i*y.j-x.j*y.i]

```

— QUAT.dotabb —

```

"QUAT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=QUAT"]
"QUATCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=QUATCAT"]
"QUAT" -> "QUATCAT"

```

domain QEQUAT QueryEquation

— QueryEquation.input —

```

)set break resume
)sys rm -f QueryEquation.output
)spool QueryEquation.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show QueryEquation
--R QueryEquation is a domain constructor
--R Abbreviation for QueryEquation is QEQUAT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for QEQUAT
--R
--R----- Operations -----
--R coerce : % -> OutputForm          equation : (Symbol,String) -> %
--R value : % -> String                variable : % -> Symbol
--R
--E 1

```

```
)spool
)lisp (bye)
```

— QueryEquation.help —

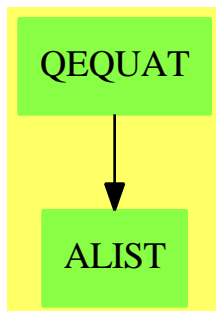
```
=====
QueryEquation examples
=====
```

This domain implements simple database queries

See Also:

o)show QueryEquation

QueryEquation (QEUAT)



See

⇒ “DataList” (DLIST) [5](#) on page [459](#)
 ⇒ “IndexCard” (ICARD) [10](#) on page [1193](#)
 ⇒ “Database” (DBASE) [5](#) on page [454](#)

Exports:

coerce equation value variable

— domain QEUAT QueryEquation —

```
)abbrev domain QEUAT QueryEquation
++ Author: Mark Botch
++ Description:
++ This domain implements simple database queries
```

```

QueryEquation(): Exports == Implementation where
  Exports == CoercibleTo(OutputForm) with
    equation: (Symbol,String) -> %
      ++ equation(s,"a") creates a new equation.
    variable: % -> Symbol
      ++ variable(q) returns the variable (i.e. left hand side) of \axiom{q}.
    value: % -> String
      ++ value(q) returns the value (i.e. right hand side) of \axiom{q}.
  Implementation == add
    Rep := Record(var:Symbol, val:String)
    coerce(u) == coerce(u.var)$Symbol = coerce(u.val)$String
    equation(x,s) == [x,s]
    variable q == q.var
    value q == q.val

```

— QEQUAT.dotabb —

```

"QEQUAT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=QEQUAT"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"QEQUAT" -> "ALIST"

```

domain QUEUE Queue

— Queue.input —

```

)set break resume
)sys rm -f Queue.output
)spool Queue.output
)set message test on
)set message auto off
)clear all

```

```

--S 1 of 46
a:Queue INT:= queue [1,2,3,4,5]
--R
--R
--R (1) [1,2,3,4,5]
--R
--E 1

```

Type: Queue(Integer)

```
--S 2 of 46
dequeue! a
--R
--R
--R (2)  1
--R
--R                                         Type: PositiveInteger
--E 2

--S 3 of 46
a
--R
--R
--R (3)  [2,3,4,5]
--R
--R                                         Type: Queue(Integer)
--E 3

--S 4 of 46
extract! a
--R
--R
--R (4)  2
--R
--R                                         Type: PositiveInteger
--E 4

--S 5 of 46
a
--R
--R
--R (5)  [3,4,5]
--R
--R                                         Type: Queue(Integer)
--E 5

--S 6 of 46
enqueue!(9,a)
--R
--R
--R (6)  9
--R
--R                                         Type: PositiveInteger
--E 6

--S 7 of 46
a
--R
--R
--R (7)  [3,4,5,9]
--R
--R                                         Type: Queue(Integer)
--E 7

--S 8 of 46
```

```

insert!(8,a)
--R
--R
--R (8) [3,4,5,9,8]
--R
--R                                         Type: Queue(Integer)
--E 8

--S 9 of 46
a
--R
--R
--R (9) [3,4,5,9,8]
--R
--R                                         Type: Queue(Integer)
--E 9

--S 10 of 46
inspect a
--R
--R
--R (10) 3
--R
--R                                         Type: PositiveInteger
--E 10

--S 11 of 46
empty? a
--R
--R
--R (11) false
--R
--R                                         Type: Boolean
--E 11

--S 12 of 46
front a
--R
--R
--R (12) 3
--R
--R                                         Type: PositiveInteger
--E 12

--S 13 of 46
back a
--R
--R
--R (13) 8
--R
--R                                         Type: PositiveInteger
--E 13

--S 14 of 46
rotate! a
--R

```

```
--R
--R (14) [4,5,9,8,3]
--R                                         Type: Queue(Integer)
--E 14

--S 15 of 46
#a
--R
--R
--R (15) 5
--R                                         Type: PositiveInteger
--E 15

--S 16 of 46
length a
--R
--R
--R (16) 5
--R                                         Type: PositiveInteger
--E 16

--S 17 of 46
less?(a,9)
--R
--R
--R (17) true
--R                                         Type: Boolean
--E 17

--S 18 of 46
more?(a,9)
--R
--R
--R (18) false
--R                                         Type: Boolean
--E 18

--S 19 of 46
size?(a,#a)
--R
--R
--R (19) true
--R                                         Type: Boolean
--E 19

--S 20 of 46
size?(a,9)
--R
--R
--R (20) false
```



```

--R
--E 20
Type: Boolean

--S 21 of 46
parts a
--R
--R
--R (21) [4,5,9,8,3]
--R
--R
--E 21
Type: List(Integer)

--S 22 of 46
bag([1,2,3,4,5])$Queue(INT)
--R
--R
--R (22) [1,2,3,4,5]
--R
--R
--E 22
Type: Queue(Integer)

--S 23 of 46
b:=empty()$(Queue INT)
--R
--R
--R (23) []
--R
--R
--E 23
Type: Queue(Integer)

--S 24 of 46
empty? b
--R
--R
--R (24) true
--R
--R
--E 24
Type: Boolean

--S 25 of 46
sample()$Queue(INT)
--R
--R
--R (25) []
--R
--R
--E 25
Type: Queue(Integer)

--S 26 of 46
c:=copy a
--R
--R
--R (26) [4,5,9,8,3]
--R
--R
--E 26
Type: Queue(Integer)

```

```
--S 27 of 46
eq?(a,c)
--R
--R
--R (27) false
--R
--R                                         Type: Boolean
--E 27
```

```
--S 28 of 46
eq?(a,a)
--R
--R
--R (28) true
--R
--R                                         Type: Boolean
--E 28
```

```
--S 29 of 46
(a=c)@Boolean
--R
--R
--R (29) true
--R
--R                                         Type: Boolean
--E 29
```

```
--S 30 of 46
(a=a)@Boolean
--R
--R
--R (30) true
--R
--R                                         Type: Boolean
--E 30
```

```
--S 31 of 46
a~c
--R
--R
--R (31) false
--R
--R                                         Type: Boolean
--E 31
```

```
--S 32 of 46
any?(x+->(x=4),a)
--R
--R
--R (32) true
--R
--R                                         Type: Boolean
--E 32
```

```
--S 33 of 46
```

```

any?(x+-->(x=11),a)
--R
--R
--R (33) false
--R
--R                                          Type: Boolean
--E 33

--S 34 of 46
every?(x+-->(x=11),a)
--R
--R
--R (34) false
--R
--R                                          Type: Boolean
--E 34

--S 35 of 46
count(4,a)
--R
--R
--R (35) 1
--R
--R                                          Type: PositiveInteger
--E 35

--S 36 of 46
count(x+-->(x>2),a)
--R
--R
--R (36) 5
--R
--R                                          Type: PositiveInteger
--E 36

--S 37 of 46
map(x+-->x+10,a)
--R
--R
--R (37) [14,15,19,18,13]
--R
--R                                          Type: Queue(Integer)
--E 37

--S 38 of 46
a
--R
--R
--R (38) [4,5,9,8,3]
--R
--R                                          Type: Queue(Integer)
--E 38

--S 39 of 46
map!(x+-->x+10,a)
--R

```

```
--R
--R (39) [14,15,19,18,13]
--R
--E 39
```

Type: Queue(Integer)

```
--S 40 of 46
a
--R
--R
--R (40) [14,15,19,18,13]
--R
--E 40
```

Type: Queue(Integer)

```
--S 41 of 46
members a
--R
--R
--R (41) [14,15,19,18,13]
--R
--E 41
```

Type: List(Integer)

```
--S 42 of 46
member?(14,a)
--R
--R
--R (42) true
--R
--E 42
```

Type: Boolean

```
--S 43 of 46
coerce a
--R
--R
--R (43) [14,15,19,18,13]
--R
--E 43
```

Type: OutputForm

```
--S 44 of 46
hash a
--R
--R
--I (44) 4999531
--R
--E 44
```

Type: SingleInteger

```
--S 45 of 46
latex a
--R
--R
--R (45) "\mbox{\bf Unimplemented}"
```

```

--R                                                    Type: String
--E 45

--S 46 of 46
)show Queue
--R
--R Queue(S: SetCategory) is a domain constructor
--R Abbreviation for Queue is QUEUE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for QUEUE
--R
--R----- Operations -----
--R back : % -> S                                bag : List(S) -> %
--R copy : % -> %                                dequeue! : % -> S
--R empty : () -> %                              empty? : % -> Boolean
--R enqueue! : (S,%) -> S                        eq? : (%,%) -> Boolean
--R extract! : % -> S                            front : % -> S
--R insert! : (S,%) -> %                        inspect : % -> S
--R length : % -> NonNegativeInteger            map : ((S -> S),%) -> %
--R queue : List(S) -> %                       rotate! : % -> %
--R sample : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ==? : (%,%) -> Boolean if S has SETCAT
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if S has SETCAT
--R count : (S,%) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R eval : (%,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R hash : % -> SingleInteger if S has SETCAT
--R latex : % -> String if S has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R member? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R members : % -> List(S) if $ has finiteAggregate
--R more? : (%,NonNegativeInteger) -> Boolean
--R parts : % -> List(S) if $ has finiteAggregate
--R size? : (%,NonNegativeInteger) -> Boolean
--R ==? : (%,%) -> Boolean if S has SETCAT
--R
--E 46

)spool
)lisp (bye)

```

— Queue.help —

=====

Queue examples

=====

A Queue object is represented as a list ordered by first-in, first-out. It operates like a line of people, where the "next" person is the one at the front of the line.

Here we create an queue of integers from a list. Notice that the order in the list is the order in the queue.

```
a:Queue INT:= queue [1,2,3,4,5]
  [1,2,3,4,5]
```

We can remove the top of the queue using dequeue!:

```
dequeue! a
  1
```

Notice that the use of dequeue! is destructive (destructive operations in Axiom usually end with ! to indicate that the underlying data structure is changed).

```
a
  [2,3,4,5]
```

The extract! operation is another name for the pop! operation and has the same effect. This operation treats the queue as a BagAggregate:

```
extract! a
  2
```

and you can see that it also has destructively modified the queue:

```
a
  [3,4,5]
```

Next we use enqueue! to add a new element to the end of the queue:

```
push!(9,a)
  9
```

Again, the push! operation is destructive so the queue is changed:

```
a
```

```
[3,4,5,9]
```

Another name for enqueue! is insert!, which treats the queue as a BagAggregate:

```
insert!(8,a)
[3,4,5,9,8]
```

and it modifies the queue:

```
a
[3,4,5,9,8]
```

The inspect function returns the top of the queue without modification, viewed as a BagAggregate:

```
inspect a
8
```

The empty? operation returns true only if there are no element on the queue, otherwise it returns false:

```
empty? a
false
```

The front operation returns the front of the queue without modification:

```
front a
3
```

The back operation returns the back of the queue without modification:

```
back a
8
```

The rotate! operation moves the item at the front of the queue to the back of the queue:

```
rotate! a
[4,5,9,8,3]
```

The # (length) operation:

```
#a
5
```

The length operation does the same thing:

```
length a
5
```

The less? predicate will compare the queue length to an integer:

```
less?(a,9)
true
```

The more? predicate will compare the queue length to an integer:

```
more?(a,9)
false
```

The size? operation will compare the queue length to an integer:

```
size?(a,#a)
true
```

and since the last computation must always be true we try:

```
size?(a,9)
false
```

The parts function will return the queue as a list of its elements:

```
parts a
[8,9,3,4,5]
```

If we have a BagAggregate of elements we can use it to construct a queue:

```
bag([1,2,3,4,5])$Queue(INT)
[1,2,3,4,5]
```

The empty function will construct an empty queue of a given type:

```
b:=empty()$(Queue INT)
[]
```

and the empty? predicate allows us to find out if a queue is empty:

```
empty? b
true
```

The sample function returns a sample, empty queue:

```
sample()$Queue(INT)
[]
```

We can copy a queue and it does not share storage so subsequent modifications of the original queue will not affect the copy:

```
c:=copy a
```



```
[4,5,9,8,3]
```

The `eq?` function is only true if the lists are the same reference, so even though `c` is a copy of `a`, they are not the same:

```
eq?(a,c)
false
```

However, `a` clearly shares a reference with itself:

```
eq?(a,a)
true
```

But we can compare `a` and `c` for equality:

```
(a=c)@Boolean
true
```

and clearly `a` is equal to itself:

```
(a=a)@Boolean
true
```

and since `a` and `c` are equal, they are clearly NOT not-equal:

```
a~=c
false
```

We can use the `any?` function to see if a predicate is true for any element:

```
any?(x+>(x=4),a)
true
```

or false for every element:

```
any?(x+>(x=11),a)
false
```

We can use the `every?` function to check every element satisfies a predicate:

```
every?(x+>(x=11),a)
false
```

We can count the elements that are equal to an argument of this type:

```
count(4,a)
1
```

or we can count against a boolean function:

```
count(x+-(x>2),a)
5
```

You can also map a function over every element, returning a new queue:

```
map(x+>x+10,a)
[14,15,19,18,13]
```

Notice that the original queue is unchanged:

```
a
[4,5,9,8,3]
```

You can use `map!` to map a function over every element and change the original queue since `map!` is destructive:

```
map!(x+>x+10,a)
[14,15,19,18,13]
```

o

Notice that the original queue has been changed:

```
a
[14,15,19,18,13]
```

The `members` function can also get the element of the queue as a list:

```
members a
[18,19,13,14,15]
```

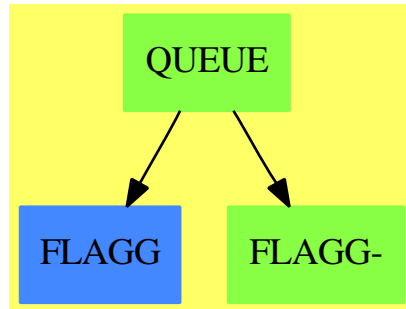
and using `member?` we can test if the queue holds a given element:

```
member?(14,a)
true
```

See Also:

- o `)show Stack`
- o `)show ArrayStack`
- o `)show Queue`
- o `)show Dequeue`
- o `)show Heap`
- o `)show BagAggregate`

Queue (QUEUE)



See

- ⇒ “Stack” (STACK) [20](#) on page [2563](#)
- ⇒ “ArrayStack” (ASTACK) [2](#) on page [65](#)
- ⇒ “Dequeue” (DEQUEUE) [5](#) on page [512](#)
- ⇒ “Heap” (HEAP) [9](#) on page [1134](#)

Exports:

any?	back	bag	coerce	copy
count	dequeue!	empty	empty?	enqueue!
eq?	eval	every?	extract!	front
hash	insert!	inspect	latex	length
less?	map	map!	member?	members
more?	parts	queue	rotate!	sample
size?	#?	?=?	?~=?	

— domain QUEUE Queue —

```

)abbrev domain QUEUE Queue
++ Author: Michael Monagan and Stephen Watt
++ Date Created: June 86 and July 87
++ Date Last Updated: Feb 92
++ Description:
++ Linked List implementation of a Queue
--% Dequeue and Heap data types

Queue(S:SetCategory): QueueAggregate S with
  queue: List S -> %
  ++ queue([x,y,...,z]) creates a queue with first (top)
  ++ element x, second element y,...,and last (bottom) element z.
  ++
  ++E e:Queue INT:= queue [1,2,3,4,5]

-- Inherited Signatures repeated for examples documentation

dequeue_! : % -> S
  
```

```

++
++X a:Queue INT:= queue [1,2,3,4,5]
++X dequeue! a
++X a
extract_! : % -> S
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X extract! a
++X a
enqueue_! : (S,%) -> S
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X enqueue! (9,a)
++X a
insert_! : (S,%) -> %
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X insert! (8,a)
++X a
inspect : % -> S
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X inspect a
front : % -> S
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X front a
back : % -> S
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X back a
rotate_! : % -> %
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X rotate! a
length : % -> NonNegativeInteger
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X length a
less? : (%,NonNegativeInteger) -> Boolean
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X less?(a,9)
more? : (%,NonNegativeInteger) -> Boolean
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X more?(a,9)
size? : (%,NonNegativeInteger) -> Boolean
++
++X a:Queue INT:= queue [1,2,3,4,5]

```

```

++X size?(a,5)
bag : List S -> %
++
++X bag([1,2,3,4,5])$Queue(INT)
empty? : % -> Boolean
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X empty? a
empty : () -> %
++
++X b:=empty()$(Queue INT)
sample : () -> %
++
++X sample()$Queue(INT)
copy : % -> %
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X copy a
eq? : (%,%) -> Boolean
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X b:=copy a
++X eq?(a,b)
map : ((S -> S),%) -> %
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X map(x+>x+10,a)
++X a
if $ has shallowlyMutable then
map! : ((S -> S),%) -> %
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X map!(x+>x+10,a)
++X a
if S has SetCategory then
latex : % -> String
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X latex a
hash : % -> SingleInteger
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X hash a
coerce : % -> OutputForm
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X coerce a
"=: (%,%) -> Boolean
++
++X a:Queue INT:= queue [1,2,3,4,5]

```

```

++X b:Queue INT:= queue [1,2,3,4,5]
++X (a=b)@Boolean
"~=" : (%,%)-> Boolean
++
++X a:Queue INT:= queue [1,2,3,4,5]
++X b:=copy a
++X (a~=b)
if % has finiteAggregate then
  every? : ((S -> Boolean),%) -> Boolean
  ++
  ++X a:Queue INT:= queue [1,2,3,4,5]
  ++X every?(x+-(x=4),a)
  any? : ((S -> Boolean),%) -> Boolean
  ++
  ++X a:Queue INT:= queue [1,2,3,4,5]
  ++X any?(x+-(x=4),a)
  count : ((S -> Boolean),%) -> NonNegativeInteger
  ++
  ++X a:Queue INT:= queue [1,2,3,4,5]
  ++X count(x+-(x>2),a)
  _# : % -> NonNegativeInteger
  ++
  ++X a:Queue INT:= queue [1,2,3,4,5]
  ++X #a
  parts : % -> List S
  ++
  ++X a:Queue INT:= queue [1,2,3,4,5]
  ++X parts a
  members : % -> List S
  ++
  ++X a:Queue INT:= queue [1,2,3,4,5]
  ++X members a
if % has finiteAggregate and S has SetCategory then
  member? : (S,%) -> Boolean
  ++
  ++X a:Queue INT:= queue [1,2,3,4,5]
  ++X member?(3,a)
  count : (S,%) -> NonNegativeInteger
  ++
  ++X a:Queue INT:= queue [1,2,3,4,5]
  ++X count(4,a)

== Stack S add
Rep := Reference List S
lastTail==> LAST$Lisp
enqueue_!(e,q) ==
  if null deref q then setref(q, list e)
  else lastTail.(deref q).rest := list e
  e
insert_!(e,q) == (enqueue_!(e,q);q)

```

```

dequeue_! q ==
  empty? q => error "empty queue"
  e := first deref q
  setref(q,rest deref q)
  e
extract_! q == dequeue_! q
rotate_! q == if empty? q then q else (enqueue_!(dequeue_! q,q); q)
length q == # deref q
front q == if empty? q then error "empty queue" else first deref q
inspect q == front q
back q == if empty? q then error "empty queue" else last deref q
queue q == ref copy q

```

— QUEUE.dotabb —

```

"QUEUE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=QUEUE"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"QUEUE" -> "FLAGG"
"QUEUE" -> "FLAGG-"

```

Chapter 19

Chapter R

domain RADFF RadicalFunctionField

— RadicalFunctionField.input —

```
)set break resume
)sys rm -f RadicalFunctionField.output
)spool RadicalFunctionField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RadicalFunctionField
--R RadicalFunctionField(F: UniqueFactorizationDomain,UP: UnivariatePolynomialCategory(F),UPUP: UnivariateP
--R Abbreviation for RadicalFunctionField is RADFF
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for RADFF
--R
--R----- Operations -----
--R ??? : (Fraction(UP),%) -> %          ??? : (%,Fraction(UP)) -> %
--R ??? : (%,%) -> %                    ??? : (Integer,%) -> %
--R ??? : (PositiveInteger,%) -> %      ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %                    ?-? : (%,%) -> %
--R -? : % -> %                         ?=? : (%,%) -> Boolean
--R 1 : () -> %                         0 : () -> %
--R ??? : (%,PositiveInteger) -> %      basis : () -> Vector(%)
--R branchPoint? : UP -> Boolean         branchPoint? : F -> Boolean
--R coerce : Fraction(UP) -> %           coerce : Integer -> %
--R coerce : % -> OutputForm             convert : UPUP -> %
--R convert : % -> UPUP                  definingPolynomial : () -> UPUP
--R discriminant : () -> Fraction(UP)    elt : (%,F,F) -> F
```



```
--R generator : () -> %
--R hash : % -> SingleInteger
--R integral? : (% , F) -> Boolean
--R integralBasis : () -> Vector(%)
--R lift : % -> UPUP
--R one? : % -> Boolean
--R ramified? : UP -> Boolean
--R rank : () -> PositiveInteger
--R recip : % -> Union(%,"failed")
--R represents : (Vector(UP),UP) -> %
--R sample : () -> %
--R singular? : F -> Boolean
--R zero? : % -> Boolean
--R ?? : (% , Fraction(Integer)) -> % if Fraction(UP) has FIELD
--R ?? : (Fraction(Integer),%) -> % if Fraction(UP) has FIELD
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (% , Integer) -> % if Fraction(UP) has FIELD
--R ??? : (% , NonNegativeInteger) -> %
--R ?/? : (% , %) -> % if Fraction(UP) has FIELD
--R D : % -> % if Fraction(UP) has DIFRING and Fraction(UP) has FIELD or Fraction(UP) has FFIELDC
--R D : (% , NonNegativeInteger) -> % if Fraction(UP) has DIFRING and Fraction(UP) has FIELD or Fraction(UP) has FFIELDC
--R D : (% , Symbol) -> % if Fraction(UP) has FIELD and Fraction(UP) has PDRING(SYMBOL)
--R D : (% , List(Symbol)) -> % if Fraction(UP) has FIELD and Fraction(UP) has PDRING(SYMBOL)
--R D : (% , Symbol, NonNegativeInteger) -> % if Fraction(UP) has FIELD and Fraction(UP) has PDRING(SYMBOL)
--R D : (% , List(Symbol), List(NonNegativeInteger)) -> % if Fraction(UP) has FIELD and Fraction(UP) has PDRING(SYMBOL)
--R D : (% , (Fraction(UP) -> Fraction(UP))) -> % if Fraction(UP) has FIELD
--R D : (% , (Fraction(UP) -> Fraction(UP)), NonNegativeInteger) -> % if Fraction(UP) has FIELD
--R ?? : (% , Integer) -> % if Fraction(UP) has FIELD
--R ?? : (% , NonNegativeInteger) -> %
--R absolutelyIrreducible? : () -> Boolean
--R algSplitSimple : (% , (UP -> UP)) -> Record(num: %, den: UP, derivden: UP, gd: UP)
--R associates? : (% , %) -> Boolean if Fraction(UP) has FIELD
--R branchPointAtInfinity? : () -> Boolean
--R characteristic : () -> NonNegativeInteger
--R characteristicPolynomial : % -> UPUP
--R charthRoot : % -> Union(%,"failed") if Fraction(UP) has CHARNZ
--R charthRoot : % -> % if Fraction(UP) has FFIELDC
--R coerce : % -> % if Fraction(UP) has FIELD
--R coerce : Fraction(Integer) -> % if Fraction(UP) has FIELD or Fraction(UP) has RETRACT(FRAC(I
--R complementaryBasis : Vector(%) -> Vector(%)
--R conditionP : Matrix(%) -> Union(Vector(%),"failed") if Fraction(UP) has FFIELDC
--R convert : Vector(Fraction(UP)) -> %
--R convert : % -> Vector(Fraction(UP))
--R coordinates : Vector(%) -> Matrix(Fraction(UP))
--R coordinates : % -> Vector(Fraction(UP))
--R coordinates : (Vector(%), Vector(%)) -> Matrix(Fraction(UP))
--R coordinates : (% , Vector(%)) -> Vector(Fraction(UP))
--R createPrimitiveElement : () -> % if Fraction(UP) has FFIELDC
--R derivationCoordinates : (Vector(%), (Fraction(UP) -> Fraction(UP))) -> Matrix(Fraction(UP)) i
--R differentiate : % -> % if Fraction(UP) has DIFRING and Fraction(UP) has FIELD or Fraction(UP)
```

```

--R differentiate : (% , NonNegativeInteger) -> % if Fraction(UP) has DIFRING and Fraction(UP) has FIELD or F
--R differentiate : (% , Symbol) -> % if Fraction(UP) has FIELD and Fraction(UP) has PDRING(SYMBOL)
--R differentiate : (% , List(Symbol)) -> % if Fraction(UP) has FIELD and Fraction(UP) has PDRING(SYMBOL)
--R differentiate : (% , Symbol, NonNegativeInteger) -> % if Fraction(UP) has FIELD and Fraction(UP) has PDRING
--R differentiate : (% , List(Symbol), List(NonNegativeInteger)) -> % if Fraction(UP) has FIELD and Fraction(UP)
--R differentiate : (% , (UP -> UP)) -> %
--R differentiate : (% , (Fraction(UP) -> Fraction(UP))) -> % if Fraction(UP) has FIELD
--R differentiate : (% , (Fraction(UP) -> Fraction(UP)), NonNegativeInteger) -> % if Fraction(UP) has FIELD
--R discreteLog : (% , %) -> Union(NonNegativeInteger, "failed") if Fraction(UP) has FFIELDC
--R discreteLog : % -> NonNegativeInteger if Fraction(UP) has FFIELDC
--R discriminant : Vector(%) -> Fraction(UP)
--R divide : (% , %) -> Record(quotient: %, remainder: %) if Fraction(UP) has FIELD
--R elliptic : () -> Union(UP, "failed")
--R enumerate : () -> List(%) if Fraction(UP) has FINITE
--R euclideanSize : % -> NonNegativeInteger if Fraction(UP) has FIELD
--R expressIdealMember : (List(%), %) -> Union(List(%), "failed") if Fraction(UP) has FIELD
--R exquo : (% , %) -> Union(% , "failed") if Fraction(UP) has FIELD
--R extendedEuclidean : (% , %) -> Record(coef1: %, coef2: %, generator: %) if Fraction(UP) has FIELD
--R extendedEuclidean : (% , %, %) -> Union(Record(coef1: %, coef2: %), "failed") if Fraction(UP) has FIELD
--R factor : % -> Factored(%) if Fraction(UP) has FIELD
--R factorsOfCyclicGroupSize : () -> List(Record(factor: Integer, exponent: Integer)) if Fraction(UP) has FF
--R gcd : (% , %) -> % if Fraction(UP) has FIELD
--R gcd : List(%) -> % if Fraction(UP) has FIELD
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynom
--R hyperelliptic : () -> Union(UP, "failed")
--R index : PositiveInteger -> % if Fraction(UP) has FINITE
--R init : () -> % if Fraction(UP) has FFIELDC
--R integralAtInfinity? : % -> Boolean
--R integralBasisAtInfinity : () -> Vector(%)
--R integralCoordinates : % -> Record(num: Vector(UP), den: UP)
--R integralDerivationMatrix : (UP -> UP) -> Record(num: Matrix(UP), den: UP)
--R integralMatrix : () -> Matrix(Fraction(UP))
--R integralMatrixAtInfinity : () -> Matrix(Fraction(UP))
--R integralRepresents : (Vector(UP), UP) -> %
--R inv : % -> % if Fraction(UP) has FIELD
--R inverseIntegralMatrix : () -> Matrix(Fraction(UP))
--R inverseIntegralMatrixAtInfinity : () -> Matrix(Fraction(UP))
--R lcm : (% , %) -> % if Fraction(UP) has FIELD
--R lcm : List(%) -> % if Fraction(UP) has FIELD
--R lookup : % -> PositiveInteger if Fraction(UP) has FINITE
--R minimalPolynomial : % -> UPUP if Fraction(UP) has FIELD
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed") if Fraction(UP) has FIELD
--R nextItem : % -> Union(% , "failed") if Fraction(UP) has FFIELDC
--R nonSingularModel : Symbol -> List(Polynomial(F)) if F has FIELD
--R normalizeAtInfinity : Vector(%) -> Vector(%)
--R numberOfComponents : () -> NonNegativeInteger
--R order : % -> OnePointCompletion(PositiveInteger) if Fraction(UP) has FFIELDC
--R order : % -> PositiveInteger if Fraction(UP) has FFIELDC
--R prime? : % -> Boolean if Fraction(UP) has FIELD
--R primeFrobenius : % -> % if Fraction(UP) has FFIELDC

```

```

--R primeFrobenius : (% , NonNegativeInteger) -> % if Fraction(UP) has FFIELDC
--R primitive? : % -> Boolean if Fraction(UP) has FFIELDC
--R primitiveElement : () -> % if Fraction(UP) has FFIELDC
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %) if Fraction(UP) has FIELD
--R ?quo? : (% , %) -> % if Fraction(UP) has FIELD
--R ramifiedAtInfinity? : () -> Boolean
--R random : () -> % if Fraction(UP) has FINITE
--R rationalPoints : () -> List(List(F)) if F has FINITE
--R reduce : Fraction(UPUP) -> Union(% , "failed") if Fraction(UP) has FIELD
--R reduceBasisAtInfinity : Vector(%) -> Vector(%)
--R reducedSystem : Matrix(%) -> Matrix(Fraction(UP))
--R reducedSystem : (Matrix(%), Vector(%)) -> Record(mat: Matrix(Fraction(UP)), vec: Vector(Fraction(UP)))
--R reducedSystem : (Matrix(%), Vector(%)) -> Record(mat: Matrix(Integer), vec: Vector(Integer)) if F has FIELD
--R reducedSystem : Matrix(%) -> Matrix(Integer) if Fraction(UP) has LINEXP(INT)
--R regularRepresentation : % -> Matrix(Fraction(UP))
--R regularRepresentation : (% , Vector(%)) -> Matrix(Fraction(UP))
--R ?rem? : (% , %) -> % if Fraction(UP) has FIELD
--R representationType : () -> Union("prime", polynomial, normal, cyclic) if Fraction(UP) has FFIELDC
--R represents : Vector(Fraction(UP)) -> %
--R represents : (Vector(Fraction(UP)), Vector(%)) -> %
--R retract : % -> Fraction(Integer) if Fraction(UP) has RETRACT(FRAC(INT))
--R retract : % -> Integer if Fraction(UP) has RETRACT(INT)
--R retractIfCan : % -> Union(Fraction(UP), "failed")
--R retractIfCan : % -> Union(Fraction(Integer), "failed") if Fraction(UP) has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(Integer, "failed") if Fraction(UP) has RETRACT(INT)
--R singularAtInfinity? : () -> Boolean
--R size : () -> NonNegativeInteger if Fraction(UP) has FINITE
--R sizeLess? : (% , %) -> Boolean if Fraction(UP) has FIELD
--R squareFree : % -> Factored(%) if Fraction(UP) has FIELD
--R squareFreePart : % -> % if Fraction(UP) has FIELD
--R subtractIfCan : (% , %) -> Union(% , "failed")
--R tableForDiscreteLogarithm : Integer -> Table(PositiveInteger, NonNegativeInteger) if Fraction(UP) has FIELD
--R traceMatrix : () -> Matrix(Fraction(UP))
--R traceMatrix : Vector(%) -> Matrix(Fraction(UP))
--R unit? : % -> Boolean if Fraction(UP) has FIELD
--R unitCanonical : % -> % if Fraction(UP) has FIELD
--R unitNormal : % -> Record(unit: % , canonical: % , associate: %) if Fraction(UP) has FIELD
--R yCoordinates : % -> Record(num: Vector(UP), den: UP)
--R
--E 1

)spool
)lisp (bye)

```

RadicalFunctionField examples

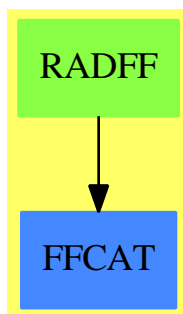
=====

Function field defined by $y^n = f(x)$;

See Also:

o)show RadicalFunctionField

RadicalFunctionField (RADFF)



See

⇒ “AlgebraicFunctionField” (ALGFF) [2](#) on page [27](#)

Exports:

0	1	absolutelyIrreducible?
algSplitSimple	associates?	basis
branchPoint?	branchPointAtInfinity?	characteristic
characteristicPolynomial	charthRoot	coerce
complementaryBasis	conditionP	convert
coordinates	createPrimitiveElement	D
derivationCoordinates	definingPolynomial	differentiate
discreteLog	discriminant	divide
elliptic	elt	euclideanSize
expressIdealMember	exquo	extendedEuclidean
factor	factorsOfCyclicGroupSize	gcd
gcdPolynomial	generator	genus
hash	hyperelliptic	index
init	integral?	integralAtInfinity?
integralBasis	integralBasisAtInfinity	integralCoordinates
integralDerivationMatrix	integralMatrix	integralMatrixAtInfinity
integralRepresents	inv	inverseIntegralMatrix
inverseIntegralMatrixAtInfinity	latex	lcm
lift	lookup	minimalPolynomial
multiEuclidean	nextItem	nonSingularModel
norm	normalizeAtInfinity	numberOfComponents
one?	order	prime?
primeFrobenius	primitive?	primitiveElement
primitivePart	principalIdeal	ramified?
ramifiedAtInfinity?	random	rank
rationalPoint?	rationalPoints	recip
reduce	reduce	reduceBasisAtInfinity
reducedSystem	regularRepresentation	representationType
represents	retract	retractIfCan
sample	singular?	singularAtInfinity?
size	sizeLess?	squareFree
squareFreePart	subtractIfCan	tableForDiscreteLogarithm
trace	traceMatrix	unit?
unitCanonical	unitNormal	yCoordinates
zero?	?*?	?**?
?+?	?-?	-?
?=?	?^?	?~=?
?/?	?quo?	?rem?

— domain RADFF RadicalFunctionField —

```

)abbrev domain RADFF RadicalFunctionField
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 27 July 1993
++ Description:
++ Function field defined by y**n = f(x);

```

```

RadicalFunctionField(F, UP, UPUP, radicnd, n): Exports == Impl where
  F      : UniqueFactorizationDomain
  UP      : UnivariatePolynomialCategory F
  UPUP    : UnivariatePolynomialCategory Fraction UP
  radicnd : Fraction UP
  n       : NonNegativeInteger

  N ==> NonNegativeInteger
  Z  ==> Integer
  RF ==> Fraction UP
  QF ==> Fraction UPUP
  UP2 ==> SparseUnivariatePolynomial UP
  REC ==> Record(factor:UP, exponent:Z)
  MOD ==> monomial(1, n)$UPUP - radicnd::UPUP
  INIT ==> if (deref brandNew?) then startUp false

Exports ==> FunctionFieldCategory(F, UP, UPUP)

Impl ==> SimpleAlgebraicExtension(RF, UPUP, MOD) add
  import ChangeOfVariable(F, UP, UPUP)
  import InnerCommonDenominator(UP, RF, Vector UP, Vector RF)
  import UnivariatePolynomialCategoryFunctions2(RF, UPUP, UP, UP2)

  diag      : Vector RF -> Vector $
  startUp   : Boolean -> Void
  fullVector : (Factored UP, N) -> PrimitiveArray UP
  iBasis     : (UP, N) -> Vector UP
  infBasis   : (RF, N) -> Vector RF
  basisvec   : () -> Vector RF
  char0StartUp: () -> Void
  charPStartUp: () -> Void
  getInfBasis : () -> Void
  radcand     : () -> UP
  charPintbas : (UPUP, RF, Vector RF, Vector RF) -> Void

  brandNew?:Reference(Boolean) := ref true
  discPoly:Reference(RF) := ref(0$RF)
  newrad:Reference(UP) := ref(0$UP)
  n1 := (n - 1)::N
  modulus := MOD
  ibasis:Vector(RF) := new(n, 0)
  invibasis:Vector(RF) := new(n, 0)
  infbasis:Vector(RF) := new(n, 0)
  invinfbasis:Vector(RF) := new(n, 0)
  mini := minIndex ibasis

  discriminant() == (INIT; discPoly())
  radcand() == (INIT; newrad())
  integralBasis() == (INIT; diag ibasis)

```

```

integralBasisAtInfinity()      == (INIT; diag infbasis)
basisvec()                    == (INIT; ibasis)
integralMatrix()              == diagonalMatrix basisvec()
integralMatrixAtInfinity()    == (INIT; diagonalMatrix infbasis)
inverseIntegralMatrix()       == (INIT; diagonalMatrix invbasis)
inverseIntegralMatrixAtInfinity() == (INIT; diagonalMatrix invinfbasis)
definingPolynomial()          == modulus
ramified?(point:F)            == zero?(radcand() point)
branchPointAtInfinity?()     == (degree(radcand()) exquo n) case "failed"
elliptic()                    == (n = 2 and degree(radcand()) = 3 => radcand(); "failed")
hyperelliptic() == (n=2 and odd? degree(radcand()) => radcand(); "failed")
diag v == [reduce monomial(qelt(v,i+mini), i) for i in 0..n1]

integralRepresents(v, d) ==
  ib := basisvec()
  represents
    [qelt(ib, i) * (qelt(v, i) /$RF d) for i in mini .. maxIndex ib]

integralCoordinates f ==
  v := coordinates f
  ib := basisvec()
  splitDenominator
    [qelt(v,i) / qelt(ib,i) for i in mini .. maxIndex ib]$Vector(RF)

integralDerivationMatrix d ==
  dlogp := differentiate(radicnd, d) / (n * radicnd)
  v := basisvec()
  cd := splitDenominator(
    [(i - mini) * dlogp + differentiate(qelt(v, i), d) / qelt(v, i)
      for i in mini..maxIndex v]$Vector(RF))
  [diagonalMatrix(cd.num), cd.den]

-- return (d0,...,d(n-1)) s.t. (1/d0, y/d1,...,y**(n-1)/d(n-1))
-- is an integral basis for the curve y**d = p
-- requires that p has no factor of multiplicity >= d
iBasis(p, d) ==
  p1 := fullVector(squareFree p, d)
  d1 := (d - 1)::N
  [*/[p1.j ** ((i * j) quo d) for j in 0..d1] for i in 0..d1]

-- returns a vector [a0,a1,...,a_{m-1}] of length m such that
-- p = a0^0 a1^1 ... a_{m-1}^{m-1}
fullVector(p, m) ==
  ans:PrimitiveArray(UP) := new(m, 0)
  ans.0 := unit p
  l := factors p
  for i in 1..maxIndex ans repeat
    ans.i :=
      (u := find(s+->s.exponent = i, l)) case "failed" => 1
      (u::REC).factor

```

ans

```
-- return (f0,...,f(n-1)) s.t. (f0, y f1,..., y**(n-1) f(n-1))
-- is a local integral basis at infinity for the curve y**d = p
inftyBasis(p, m) ==
  rt := rootPoly(p(x := inv(monomial(1, 1)$UP :: RF)), m)
  m ^= rt.exponent =>
    error "Curve not irreducible after change of variable 0 -> infinity"
  a := (rt.coef) x
  b:RF := 1
  v := iBasis(rt.radicand, m)
  w:Vector(RF) := new(m, 0)
  for i in mini..maxIndex v repeat
    qsetelt_!(w, i, b / ((qelt(v, i)::RF) x))
    b := b * a
  w

charPintbas(p, c, v, w) ==
  degree(p) ^= n => error "charPintbas: should not happen"
  q:UP2 := map(s+>retract(s)@UP, p)
  ib := integralBasis()$FunctionFieldIntegralBasis(UP, UP2,
    SimpleAlgebraicExtension(UP, UP2, q))
  not diagonal?(ib.basis)=>
    error "charPintbas: integral basis not diagonal"
  a:RF := 1
  for i in minRowIndex(ib.basis) .. maxRowIndex(ib.basis)
    for j in minColIndex(ib.basis) .. maxColIndex(ib.basis)
      for k in mini .. maxIndex v repeat
        qsetelt_!(v, k, (qelt(ib.basis, i, j) / ib.basisDen) * a)
        qsetelt_!(w, k, qelt(ib.basisInv, i, j) * inv a)
      a := a * c
  void

charPStartUp() ==
  r := mkIntegral modulus
  charPintbas(r.poly, r.coef, ibasis, invibasis)
  x := inv(monomial(1, 1)$UP :: RF)
  invmod := monomial(1, n)$UPUP - (radicnd x)::UPUP
  r := mkIntegral invmod
  charPintbas(r.poly, (r.coef) x, infbasis, invinfbasis)

startUp b ==
  brandNew?() := b
  if zero?(p := characteristic()$F) or p > n then char0StartUp()
    else charPStartUp()
  dsc:RF := ((-1)$Z ** ((n *$N n1) quo 2::N) * (n::Z)**n)$Z *
    radicnd ** n1 *
    */[qelt(ibasis, i) ** 2 for i in mini..maxIndex ibasis]
  discPoly() := primitivePart(numer dsc) / denom(dsc)
  void
```



```

char0StartUp() ==
  rp      := rootPoly(radicnd, n)
  rp.exponent ^= n =>
    error "RadicalFunctionField: curve is not irreducible"
  newrad() := rp.radicand
  ib       := iBasis(newrad(), n)
  infb     := infyBasis(radicnd, n)
  invden:RF := 1
  for i in mini..maxIndex ib repeat
    qsetelt_!(invibasis, i, a := qelt(ib, i) * invden)
    qsetelt_!(ibasis, i, inv a)
    invden := invden / rp.coef      -- always equals 1/rp.coef**(i-mini)
    qsetelt_!(infbasis, i, a := qelt(infb, i))
    qsetelt_!(invinfbasis, i, inv a)
  void

ramified?(p:UP) ==
  (r := retractIfCan(p)@Union(F, "failed")) case F =>
    singular?(r::F)
  (radcand() exquo p) case UP

singular?(p:UP) ==
  (r := retractIfCan(p)@Union(F, "failed")) case F =>
    singular?(r::F)
  (radcand() exquo(p**2)) case UP

branchPoint?(p:UP) ==
  (r := retractIfCan(p)@Union(F, "failed")) case F =>
    branchPoint?(r::F)
  ((q := (radcand() exquo p)) case UP) and
  ((q::UP exquo p) case "failed")

singular?(point:F) ==
  zero?(radcand() point) and
  zero?(((radcand() exquo (monomial(1,1)$UP-point::UP))::UP) point)

branchPoint?(point:F) ==
  zero?(radcand() point) and not
  zero?(((radcand() exquo (monomial(1,1)$UP-point::UP))::UP) point)

```

— RADFF.dotabb —

```

"RADFF" [color="#88FF44",href="bookvol10.3.pdf#nameddest=RADFF"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"RADFF" -> "FFCAT"

```

— RadixExpansion.input —

[illegible]

```
--E 4

--S 5 of 17
(5/24)::RadixExpansion(10)
--R
--R
--R
--R (5) 0.2083
--R
--R Type: RadixExpansion(10)
--E 5

--S 6 of 17
(5/24)::RadixExpansion(12)
--R
--R
--R
--R (6) 0.26
--R
--R Type: RadixExpansion(12)
--E 6

--S 7 of 17
(5/24)::RadixExpansion(16)
--R
--R
--R
--R (7) 0.35
--R
--R Type: RadixExpansion(16)
--E 7

--S 8 of 17
(5/24)::RadixExpansion(36)
--R
--R
--R
--R (8) 0.7I
--R
--R Type: RadixExpansion(36)
--E 8

--S 9 of 17
(5/24)::RadixExpansion(38)
--R
--R
--R
--R (9) 0 . 7 34 31 25 12
--R
--R Type: RadixExpansion(38)
--E 9

--S 10 of 17
a := (76543/210)::RadixExpansion(8)
--R
--R
--R
```

```

--R (10) 554.37307
--R
--R                                         Type: RadixExpansion(8)
--E 10

--S 11 of 17
w := wholeRagits a
--R
--R
--R (11) [5,5,4]
--R
--R                                         Type: List(Integer)
--E 11

--S 12 of 17
f0 := prefixRagits a
--R
--R
--R (12) [3]
--R
--R                                         Type: List(Integer)
--E 12

--S 13 of 17
f1 := cycleRagits a
--R
--R
--R (13) [7,3,0,7]
--R
--R                                         Type: List(Integer)
--E 13

--S 14 of 17
u:RadixExpansion(8):=wholeRadix(w)+fractRadix(f0,f1)
--R
--R
--R -----
--R (14) 554.37307
--R
--R                                         Type: RadixExpansion(8)
--E 14

--S 15 of 17
v: RadixExpansion(12) := fractRadix([1,2,3,11], [0])
--R
--R
--R -
--R (15) 0.123B0
--R
--R                                         Type: RadixExpansion(12)
--E 15

--S 16 of 17
fractRagits(u)
--R
--R

```

```

--R      -----
--R   (16) [3,7,3,0,7,7]
--R
--R                                          Type: Stream(Integer)
--E 16

--S 17 of 17
a :: Fraction(Integer)
--R
--R
--R      76543
--R   (17) -----
--R      210
--R
--R                                          Type: Fraction(Integer)
--E 17
)spool
)lisp (bye)

```

— RadixExpansion.help —

=====

RadixExpansion examples

=====

It possible to expand numbers in general bases.

Here we expand 111 in base 5. This means

$$10^2 + 10^1 + 10^0 = 4 * 5^2 + 2 * 5^1 + 5^0$$

```

111::RadixExpansion(5)
421
                                          Type: RadixExpansion 5

```

You can expand fractions to form repeating expansions.

```

(5/24)::RadixExpansion(2)
--
0.00110
                                          Type: RadixExpansion 2

```

```

(5/24)::RadixExpansion(3)
--
0.012
                                          Type: RadixExpansion 3

```

```

(5/24)::RadixExpansion(8)
--
0.152

```

Type: RadixExpansion 8

(5/24)::RadixExpansion(10)

0.2083̄

Type: RadixExpansion 10

For bases from 11 to 36 the letters A through Z are used.

(5/24)::RadixExpansion(12)

0.26

Type: RadixExpansion 12

(5/24)::RadixExpansion(16)

0.35̄

Type: RadixExpansion 16

(5/24)::RadixExpansion(36)

0.7I

Type: RadixExpansion 36

For bases greater than 36, the ragits are separated by blanks.

(5/24)::RadixExpansion(38)

0 . 7 34 31 25 12̄

Type: RadixExpansion 38

The RadixExpansion type provides operations to obtain the individual ragits. Here is a rational number in base 8.

a := (76543/210)::RadixExpansion(8)

554.37307̄

Type: RadixExpansion 8

The operation wholeRagits returns a list of the ragits for the integral part of the number.

w := wholeRagits a

[5,5,4]

Type: List Integer

The operations prefixRagits and cycleRagits return lists of the initial and repeating ragits in the fractional part of the number.

f0 := prefixRagits a

[3]

Type: List Integer

```
f1 := cycleRagits a
      [7,3,0,7]
```

Type: List Integer

You can construct any radix expansion by giving the whole, prefix and cycle parts. The declaration is necessary to let Axiom know the base of the ragits.

```
u:RadixExpansion(8):=wholeRadix(w)+fractRadix(f0,f1)
      ----
    554.37307
```

Type: RadixExpansion 8

If there is no repeating part, then the list [0] should be used.

```
v: RadixExpansion(12) := fractRadix([1,2,3,11], [0])
      -
    0.123B0
```

Type: RadixExpansion 12

If you are not interested in the repeating nature of the expansion, an infinite stream of ragits can be obtained using fractRagits.

```
fractRagits(u)
      -----
    [3,7,3,0,7,7]
```

Type: Stream Integer

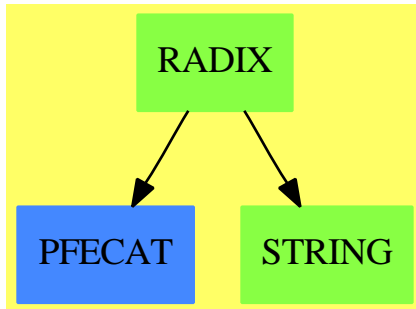
Of course, it's possible to recover the fraction representation:

```
a :: Fraction(Integer)
      76543
      ----
      210
```

Type: Fraction Integer

See Also:

- o)help DecimalExpansion
- o)help BinaryExpansion
- o)help HexadecimalExpansion
- o)show RadixExpansion

RadixExpansion (RADIX)

See

- ⇒ “BinaryExpansion” (BINARY) [3](#) on page [292](#)
- ⇒ “DecimalExpansion” (DECIMAL) [5](#) on page [465](#)
- ⇒ “HexadecimalExpansion” (HEXADEC) [9](#) on page [1142](#)

Exports:

0	1	abs
associates?	ceiling	characteristic
charthRoot	coerce	conditionP
convert	cycleRagits	D
denom	denominator	differentiate
divide	euclideanSize	eval
expressIdealMember	exquo	extendedEuclidean
factor	factorPolynomial	factorSquareFreePolynomial
floor	fractRadix	fractRagits
fractionPart	gcd	gcdPolynomial
hash	init	inv
latex	lcm	map
max	min	multiEuclidean
negative?	nextItem	numer
numerator	one?	patternMatch
positive?	prefixRagits	prime?
principalIdeal	random	recip
reducedSystem	retract	retractIfCan
sample	sign	sizeLess?
solveLinearPolynomialEquation	squareFree	squareFreePart
squareFreePolynomial	subtractIfCan	unit?
unitCanonical	unitNormal	wholePart
wholeRadix	wholeRagits	zero?
?.?	?*?	?**?
?+?	?-?	?-?
?/?	?=?	?^?
?~=?	?<?	?<=?
?>?	?>=?	?quo?
?rem?		

— domain RADIX RadixExpansion —

```
)abbrev domain RADIX RadixExpansion
++ Author: Stephen M. Watt
++ Date Created: October 1986
++ Date Last Updated: May 15, 1991
++ Description:
++ This domain allows rational numbers to be presented as repeating
++ decimal expansions or more generally as repeating expansions in any base.
```

```
RadixExpansion(bb): Exports == Implementation where
  bb   : Integer
  I    ==> Integer
  NNI  ==> NonNegativeInteger
  OUT  ==> OutputForm
  RN   ==> Fraction Integer
  ST   ==> Stream Integer
  QuoRem ==> Record(quotient: Integer, remainder: Integer)
```

```

Exports ==> QuotientFieldCategory(Integer) with
  coerce: % -> Fraction Integer
    ++ coerce(rx) converts a radix expansion to a rational number.
  fractionPart: % -> Fraction Integer
    ++ fractionPart(rx) returns the fractional part of a radix expansion.
  wholeRagits: % -> List Integer
    ++ wholeRagits(rx) returns the ragits of the integer part
    ++ of a radix expansion.
  fractRagits: % -> Stream Integer
    ++ fractRagits(rx) returns the ragits of the fractional part
    ++ of a radix expansion.
  prefixRagits: % -> List Integer
    ++ prefixRagits(rx) returns the non-cyclic part of the ragits
    ++ of the fractional part of a radix expansion.
    ++ For example, if \spad{x = 3/28 = 0.10 714285 714285 ...},
    ++ then \spad{prefixRagits(x)}=[1,0]}.
  cycleRagits: % -> List Integer
    ++ cycleRagits(rx) returns the cyclic part of the ragits of the
    ++ fractional part of a radix expansion.
    ++ For example, if \spad{x = 3/28 = 0.10 714285 714285 ...},
    ++ then \spad{cycleRagits(x)} = [7,1,4,2,8,5]}.
  wholeRadix: List Integer -> %
    ++ wholeRadix(l) creates an integral radix expansion from a list
    ++ of ragits.
    ++ For example, \spad{wholeRadix([1,3,4])} will return \spad{134}.
  fractRadix: (List Integer, List Integer) -> %
    ++ fractRadix(pre,cyc) creates a fractional radix expansion
    ++ from a list of prefix ragits and a list of cyclic ragits.
    ++ e.g., \spad{fractRadix([1],[6])} will return \spad{0.16666666...}.

Implementation ==> add
  -- The efficiency of arithmetic operations is poor.
  -- Could use a lazy eval where either rational rep
  -- or list of ragit rep (the current) or both are kept
  -- as demanded.

  bb < 2 => error "Radix base must be at least 2"
  Rep := Record(sgn: Integer,      int: List Integer,
                pfx: List Integer, cyc: List Integer)

  q:      RN
  qr:     QuoRem
  a,b:    %
  n:      I

  radixInt:  (I, I)    -> List I
  radixFrac: (I, I, I) -> Record(pfx: List I, cyc: List I)
  checkRagits: List I  -> Boolean

```

```

-- Arithmetic operations
characteristic() == 0
differentiate a == 0

0    == [1, nil(), nil(), nil()]
1    == [1, [1], nil(), nil()]
- a  == (a = 0 => 0; [-a.sgn, a.int, a.pfx, a.cyc])
a + b == (a::RN + b::RN)::%
a - b == (a::RN - b::RN)@RN::%
n * a == (n      * a::RN)::%
a * b == (a::RN * b::RN)::%
a / b == (a::RN / b::RN)::%
(i:I) / (j:I) == (i/j)@RN :: %
a < b == a::RN < b::RN
a = b == a.sgn = b.sgn and a.int = b.int and
        a.pfx = b.pfx and a.cyc = b.cyc
numer a == numer(a::RN)
denom a == denom(a::RN)

-- Algebraic coercions
coerce(a):RN == (wholePart a) :: RN + fractionPart a
coerce(n):%  == n :: RN :: %
coerce(q):%  ==
  s := 1; if q < 0 then (s := -1; q := -q)
  qr := divide(numer q,denom q)
  whole := radixInt (qr.quotient,bb)
  fractn := radixFrac(qr.remainder,denom q,bb)
  cycle := (fractn.cyc = [0] => nil(); fractn.cyc)
  [s,whole,fractn.pfx,cycle]

retractIfCan(a):Union(RN,"failed") == a::RN
retractIfCan(a):Union(I,"failed") ==
  empty?(a.pfx) and empty?(a.cyc) => wholePart a
  "failed"

-- Exported constructor/destructors
ceiling a == ceiling(a::RN)
floor a == floor(a::RN)

wholePart a ==
  n0 := 0
  for r in a.int repeat n0 := bb*n0 + r
  a.sgn*n0
fractionPart a ==
  n0 := 0
  for r in a.pfx repeat n0 := bb*n0 + r
  null a.cyc =>
    a.sgn*n0/bb**((#a.pfx)::NNI)
  n1 := n0
  for r in a.cyc repeat n1 := bb*n1 + r

```

```

n := n1 - n0
d := (bb**((#a.cyc)::NNI) - 1) * bb**((#a.pfx)::NNI)
a.sgn*n/d

wholeRagits a == a.int
fractRagits a == concat(construct(a.pfx)@ST, repeating a.cyc)
prefixRagits a == a.pfx
cycleRagits a == a.cyc

wholeRadix li ==
  checkRagits li
  [1, li, nil(), nil()]
fractRadix(lpfx, lcyc) ==
  checkRagits lpfx; checkRagits lcyc
  [1, nil(), lpfx, lcyc]

-- Output

ALPHAS : String := "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

intToExpr(i:I): OUT ==
  -- computes a digit for bases between 11 and 36
  i < 10 => i :: OUT
  elt(ALPHAS, (i-10) + minIndex(ALPHAS)) :: OUT

exprgroup(le: List OUT): OUT ==
  empty? le      => error "exprgroup needs non-null list"
  empty? rest le => first le
  abs bb <= 36 => hconcat le
  blankSeparate le

intgroup(li: List I): OUT ==
  empty? li      => error "intgroup needs non-null list"
  empty? rest li => intToExpr first(li)
  abs bb <= 10 => hconcat [i :: OUT for i in li]
  abs bb <= 36 => hconcat [intToExpr(i) for i in li]
  blankSeparate [i :: OUT for i in li]

overBar(li: List I): OUT == overbar intgroup li

coerce(a): OUT ==
  le : List OUT := nil()
  if not null a.cyc then le := concat(overBar a.cyc, le)
  if not null a.pfx then le := concat(intgroup a.pfx, le)
  if not null le then le := concat(".", :: OUT, le)
  if not null a.int then le := concat(intgroup a.int, le)
  else le := concat(0 :: OUT, le)
  rex := exprgroup le
  if a.sgn < 0 then -rex else rex

```

```

-- Construction utilities
checkRagits li ==
  for i in li repeat if i < 0 or i >= bb then
    error "Each ragit (digit) must be between 0 and base-1"
  true

radixInt(n,bas) ==
  rits: List I := nil()
  while abs n ^= 0 repeat
    qr := divide(n,bas)
    n := qr.quotient
    rits := concat(qr.remainder,rits)
  rits

radixFrac(num,den,bas) ==
  -- Rits is the sequence of quotient/remainder pairs
  -- in calculating the radix expansion of the rational number.
  -- We wish to find p and c such that
  --   rits.i are distinct   for 0<=i<=p+c-1
  --   rits.i = rits.(i+p)   for i>p
  -- I.e. p is the length of the non-periodic prefix and c is
  -- the length of the cycle.

  -- Compute p and c using Floyd's algorithm.
  -- 1. Find smallest n s.t. rits.n = rits.(2*n)
  qr := divide(bas * num, den)
  i : I := 0
  qr1i := qr2i := qr
  rits: List QuoRem := [qr]
  until qr1i = qr2i repeat
    qr1i := divide(bas * qr1i.remainder,den)
    qr2i := divide(bas * qr2i.remainder,den)
    qr2i := divide(bas * qr2i.remainder,den)
    rits := concat(qr2i, concat(qr1i, rits))
    i := i + 1
  rits := reverse_! rits
  n := i
  -- 2. Find p = first i such that rits.i = rits.(i+n)
  ritsi := rits
  ritsn := rits; for i in 1..n repeat ritsn := rest ritsn
  i := 0
  while first(ritsi) ^= first(ritsn) repeat
    ritsi := rest ritsi
    ritsn := rest ritsn
    i := i + 1
  p := i
  -- 3. Find c = first i such that rits.p = rits.(p+i)
  ritsn := rits; for i in 1..n repeat ritsn := rest ritsn
  rn := first ritsn
  cfound:= false

```

```

c : I := 0
for i in 1..p while not cfound repeat
  ritsn := rest ritsn
  if rn = first(ritsn) then
    c := i
    cfound := true
  if not cfound then c := n
-- 4. Now produce the lists of ragits.
ritspfx: List I := nil()
ritscyc: List I := nil()
for i in 1..p repeat
  ritspfx := concat(first(rits).quotient, ritspfx)
  rits := rest rits
for i in 1..c repeat
  ritscyc := concat(first(rits).quotient, ritscyc)
  rits := rest rits
[reverse_! ritspfx, reverse_! ritscyc]

```

— RADIX.dotabb —

```

"RADIX" [color="#88FF44",href="bookvol10.3.pdf#nameddest=RADIX"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"RADIX" -> "PFECAT"
"RADIX" -> "STRING"

```

domain RECLOS RealClosure

The domain constructore **RealClosure** by Renaud Rioboo (University of Paris 6, France) provides the real closure of an ordered field. The implementation is based on interval arithmetic. Moreover, the design of this constructor and its related packages allows an easy use of other codings for real algebraic numbers. ordered field

The RealClosure domain is the end-user code, it provides usual arithmetics with real algebraic numbers, along with the functionalities of a real closed field. It also provides functions to approximate a real algebraic number by an element of the base field. This approximation may either be absolute (approximate) or relative (realtivApprox).

CAVEATS

Since real algebraic expressions are stored as depending on "real roots" which are managed like variables, there is an ordering on these. This ordering is dynamical in the sense that any

new algebraic takes precedence over older ones. In particular every creation function raises a new "real root". This has the effect that when you type something like `sqrt(2) + sqrt(2)` you have two new variables which happen to be equal. To avoid this name the expression such as in `s2 := sqrt(2) ; s2 + s2`

Also note that computing times depend strongly on the ordering you implicitly provide. Please provide algebraics in the order which most natural to you.

LIMITATIONS

The file `reclos.input` show some basic use of the package. This package uses algorithms which are published in [1] and [2] which are based on field arithmetics, in particular for polynomial gcd related algorithms. This can be quite slow for high degree polynomials and subresultants methods usually work best. Beta versions of the package try to use these techniques in a better way and work significantly faster. These are mostly based on unpublished algorithms and cannot be distributed. Please contact the author if you have a particular problem to solve or want to use these versions.

Be aware that approximations behave as post-processing and that all computations are done exactly. They can thus be quite time consuming when depending on several "real roots".

— RealClosure.input —

```
)set break resume
)sys rm -f RealClosure.output
)spool RealClosure.output
)set message test on
)set message auto off
)clear all
--S 1 of 67
Ran := RECLOS(FRAC INT)
--R
--R
--R (1) RealClosure(Fraction(Integer))
--R
--R                                          Type: Domain
--E 1

--S 2 of 67
fourSquares(a:Ran,b:Ran,c:Ran,d:Ran):Ran==sqrt(a)+sqrt(b)-sqrt(c)-sqrt(d)
--R
--R Function declaration fourSquares : (RealClosure(Fraction(Integer)),
--R   RealClosure(Fraction(Integer)),RealClosure(Fraction(Integer)),
--R   RealClosure(Fraction(Integer))) -> RealClosure(Fraction(Integer))
--R   has been added to workspace.
--R
--R                                          Type: Void
--E 2

--S 3 of 67
squareDiff1 := fourSquares(73,548,60,586)
--R
```

```

--R   Compiling function fourSquares with type (RealClosure(Fraction(
--R     Integer)),RealClosure(Fraction(Integer)),RealClosure(Fraction(
--R     Integer)),RealClosure(Fraction(Integer))) -> RealClosure(Fraction
--R     (Integer))
--R
--R           +---+ +---+ +---+ +---+
--R   (3)  - \|586 - \|60 + \|548 + \|73
--R                                           Type: RealClosure(Fraction(Integer))
--E 3

--S 4 of 67
recip(squareDiff1)
--R
--R
--R   (4)
--R           +---+ +---+ +---+ +---+ +---+
--R   ((54602\|548 + 149602\|73 )\|60 + 49502\|73 \|548 + 9900895)\|586
--R   +
--R           +---+ +---+ +---+ +---+ +---+
--R   (154702\|73 \|548 + 30941947)\|60 + 10238421\|548 + 28051871\|73
--R                                           Type: Union(RealClosure(Fraction(Integer)),...)
--E 4

--S 5 of 67
sign(squareDiff1)
--R
--R
--R   (5)  1
--R                                           Type: PositiveInteger
--E 5

--S 6 of 67
squareDiff2 := fourSquares(165,778,86,990)
--R
--R
--R           +---+ +---+ +---+ +---+
--R   (6)  - \|990 - \|86 + \|778 + \|165
--R                                           Type: RealClosure(Fraction(Integer))
--E 6

--S 7 of 67
recip(squareDiff2)
--R
--R
--R   (7)
--R           +---+ +---+ +---+ +---+ +---+
--R   ((556778\|778 + 1209010\|165 )\|86 + 401966\|165 \|778 + 144019431)
--R   *
--R           +---+
--R   \|990

```



```

--R      +
--R      +----+ +----+          +---+          +----+          +----+
--R      (1363822\|165 \|778 + 488640503)\|86 + 162460913\|778 + 352774119\|165
--R                                          Type: Union(RealClosure(Fraction(Integer)),...)
--E 7

--S 8 of 67
sign(squareDiff2)
--R
--R
--R      (8)  1
--R
--R                                          Type: PositiveInteger
--E 8

--S 9 of 67
squareDiff3 := fourSquares(217,708,226,692)
--R
--R
--R      +----+ +----+ +----+ +----+
--R      (9)  - \|692 - \|226 + \|708 + \|217
--R                                          Type: RealClosure(Fraction(Integer))
--E 9

--S 10 of 67
recip(squareDiff3)
--R
--R
--R      (10)
--R      +----+          +----+ +----+          +----+ +----+          +----+
--R      ((- 34102\|708 - 61598\|217 )\|226 - 34802\|217 \|708 - 13641141)\|692
--R      +
--R      +----+ +----+          +----+          +----+          +----+
--R      (- 60898\|217 \|708 - 23869841)\|226 - 13486123\|708 - 24359809\|217
--R                                          Type: Union(RealClosure(Fraction(Integer)),...)
--E 10

--S 11 of 67
sign(squareDiff3)
--R
--R
--R      (11)  - 1
--R
--R                                          Type: Integer
--E 11

--S 12 of 67
squareDiff4 := fourSquares(155,836,162,820)
--R
--R
--R      +----+ +----+ +----+ +----+
--R      (12)  - \|820 - \|162 + \|836 + \|155

```

```

--R                                                    Type: RealClosure(Fraction(Integer))
--E 12

--S 13 of 67
recip(squareDiff4)
--R
--R
--R (13)
--R      +---+      +---+ +---+      +---+ +---+      +---+
--R      ((- 37078\|836 - 86110\|155 )\|162 - 37906\|155 \|836 - 13645107)\|820
--R      +
--R      +---+ +---+      +---+      +---+      +---+      +---+
--R      (- 85282\|155 \|836 - 30699151)\|162 - 13513901\|836 - 31384703\|155
--R                                                    Type: Union(RealClosure(Fraction(Integer)),...)
--E 13

--S 14 of 67
sign(squareDiff4)
--R
--R
--R (14) - 1
--R                                                    Type: Integer
--E 14

--S 15 of 67
squareDiff5 := fourSquares(591,772,552,818)
--R
--R
--R      +---+      +---+      +---+      +---+
--R      (15) - \|818 - \|552 + \|772 + \|591
--R                                                    Type: RealClosure(Fraction(Integer))
--E 15

--S 16 of 67
recip(squareDiff5)
--R
--R
--R (16)
--R      +---+      +---+ +---+      +---+ +---+      +---+
--R      ((70922\|772 + 81058\|591 )\|552 + 68542\|591 \|772 + 46297673)\|818
--R      +
--R      +---+ +---+      +---+      +---+      +---+      +---+
--R      (83438\|591 \|772 + 56359389)\|552 + 47657051\|772 + 54468081\|591
--R                                                    Type: Union(RealClosure(Fraction(Integer)),...)
--E 16

--S 17 of 67
sign(squareDiff5)
--R
--R

```

```

--R (17) 1
--R
--R                                         Type: PositiveInteger
--E 17

--S 18 of 67
squareDiff6 := fourSquares(434,1053,412,1088)
--R
--R
--R          +----+   +----+   +----+   +----+
--R (18)  - \|1088  - \|412  + \|1053  + \|434
--R
--R                                         Type: RealClosure(Fraction(Integer))
--E 18

--S 19 of 67
recip(squareDiff6)
--R
--R
--R (19)
--R          +----+   +----+   +----+   +----+   +----+
--R ((115442\|1053  + 179818\|434 )\|412  + 112478\|434 \|1053  + 76037291)
--R *
--R          +----+
--R          \|1088
--R +
--R          +----+   +----+   +----+   +----+   +----+
--R (182782\|434 \|1053  + 123564147)\|412  + 77290639\|1053  + 120391609\|434
--R
--R                                         Type: Union(RealClosure(Fraction(Integer)),...)
--E 19

--S 20 of 67
sign(squareDiff6)
--R
--R
--R (20) 1
--R
--R                                         Type: PositiveInteger
--E 20

--S 21 of 67
squareDiff7 := fourSquares(514,1049,446,1152)
--R
--R
--R          +----+   +----+   +----+   +----+
--R (21)  - \|1152  - \|446  + \|1049  + \|514
--R
--R                                         Type: RealClosure(Fraction(Integer))
--E 21

--S 22 of 67
recip(squareDiff7)
--R
--R

```

```

--R (22)
--R      +-----+      +----+ +----+      +----+ +-----+
--R      ((349522\|1049 + 499322\|514 )\|446 + 325582\|514 \|1049 + 239072537)
--R      *
--R      +-----+
--R      \|1152
--R      +
--R      +----+ +-----+      +----+      +-----+      +----+
--R      (523262\|514 \|1049 + 384227549)\|446 + 250534873\|1049 + 357910443\|514
--R                                          Type: Union(RealClosure(Fraction(Integer)),...)
--E 22

--S 23 of 67
sign(squareDiff7)
--R
--R
--R (23)  1
--R
--R                                          Type: PositiveInteger
--E 23

--S 24 of 67
squareDiff8 := fourSquares(190,1751,208,1698)
--R
--R
--R      +-----+      +----+      +-----+      +----+
--R (24)  - \|1698 - \|208 + \|1751 + \|190
--R                                          Type: RealClosure(Fraction(Integer))
--E 24

--S 25 of 67
recip(squareDiff8)
--R
--R
--R (25)
--R      +-----+      +----+ +----+      +----+ +-----+
--R      (- 214702\|1751 - 651782\|190 )\|208 - 224642\|190 \|1751
--R      +
--R      - 129571901
--R      *
--R      +-----+
--R      \|1698
--R      +
--R      +----+ +-----+      +----+      +-----+
--R      (- 641842\|190 \|1751 - 370209881)\|208 - 127595865\|1751
--R      +
--R      +----+
--R      - 387349387\|190
--R                                          Type: Union(RealClosure(Fraction(Integer)),...)
--E 25

```

```

--S 26 of 67
sign(squareDiff8)
--R
--R
--R (26) - 1
--R
--R                                          Type: Integer
--E 26

--S 27 of 67
relativeApprox(squareDiff8,10**(-3))::Float
--R
--R
--R (27) - 0.2340527771 5937700123 E -10
--R
--R                                          Type: Float
--E 27

--S 28 of 67
l := allRootsOf((x**2-2)**2-2)$Ran
--R
--R
--R (28) [%A33,%A34,%A35,%A36]
--R
--R                                          Type: List(RealClosure(Fraction(Integer)))
--E 28

--S 29 of 67
removeDuplicates map(mainDefiningPolynomial,l)
--R
--R
--R          4      2
--R (29) [% - 4? + 2]
--RType: List(Union(SparseUnivariatePolynomial(RealClosure(Fraction(Integer))),"failed"))
--E 29

--S 30 of 67
map(mainCharacterization,l)
--R
--R
--R (30) [[- 2,- 1[,- 1,0[, [0,1[, [1,2[
--RType: List(Union(RightOpenIntervalRootCharacterization(...
--E 30

--S 31 of 67
[reduce(+,l),reduce(*,l)-2]
--R
--R
--R (31) [0,0]
--R
--R                                          Type: List(RealClosure(Fraction(Integer)))
--E 31

--S 32 of 67

```

```

(s2, s5, s10) := (sqrt(2)$Ran, sqrt(5)$Ran, sqrt(10)$Ran)
--R
--R
--R      +---+
--R (32)  \||10
--R
--R                                          Type: RealClosure(Fraction(Integer))
--E 32

--S 33 of 67
eq1:=sqrt(s10+3)*sqrt(s5+2) - sqrt(s10-3)*sqrt(s5-2) = sqrt(10*s2+10)
--R
--R
--R      +-----+ +-----+   +-----+ +-----+   +-----+
--R      | +---+   | +-+       | +---+   | +-+       | +---+
--R (33) - \||\|10 - 3 \||\|5 - 2 + \||\|10 + 3 \||\|5 + 2 = \||10\|2 + 10
--R
--R                                          Type: Equation(RealClosure(Fraction(Integer)))
--E 33

--S 34 of 67
eq1::Boolean
--R
--R
--R (34)  true
--R
--R                                          Type: Boolean
--E 34

--S 35 of 67
eq2:=sqrt(s5+2)*sqrt(s2+1) - sqrt(s5-2)*sqrt(s2-1) = sqrt(2*s10+2)
--R
--R
--R      +-----+ +-----+   +-----+ +-----+   +-----+
--R      | +-+       | +-+       | +-+       | +-+       | +-+
--R (35) - \||\|5 - 2 \||\|2 - 1 + \||\|5 + 2 \||\|2 + 1 = \||2\|10 + 2
--R
--R                                          Type: Equation(RealClosure(Fraction(Integer)))
--E 35

--S 36 of 67
eq2::Boolean
--R
--R
--R (36)  true
--R
--R                                          Type: Boolean
--E 36

--S 37 of 67
s3 := sqrt(3)$Ran
--R
--R
--R      +---+
--R (37)  \||3

```

```

--R                                                    Type: RealClosure(Fraction(Integer))
--E 37

--S 38 of 67
s7:= sqrt(7)$Ran
--R
--R
--R      +-+
--R (38)  \|7
--R                                                    Type: RealClosure(Fraction(Integer))
--E 38

--S 39 of 67
e1 := sqrt(2*s7-3*s3,3)
--R
--R
--R      +-----+
--R      3|  +-+  +-+
--R (39)  \|2\|7  - 3\|3
--R                                                    Type: RealClosure(Fraction(Integer))
--E 39

--S 40 of 67
e2 := sqrt(2*s7+3*s3,3)
--R
--R
--R      +-----+
--R      3|  +-+  +-+
--R (40)  \|2\|7  + 3\|3
--R                                                    Type: RealClosure(Fraction(Integer))
--E 40

--S 41 of 67
e2-e1-s3
--R
--R
--R (41)  0
--R                                                    Type: RealClosure(Fraction(Integer))
--E 41

--S 42 of 67
pol : UP(x,Ran) := x**4+(7/3)*x**2+30*x-(100/3)
--R
--R
--R      4      7      2      100
--R (42)  x  + - x  + 30x - ---
--R      3      3      3
--R                                                    Type: UnivariatePolynomial(x,RealClosure(Fraction(Integer)))
--E 42

```

```

--S 43 of 67
r1 := sqrt(7633)$Ran
--R
--R
--R      +-----+
--R  (43)  \|7633
--R
--R                                          Type: RealClosure(Fraction(Integer))
--E 43

--S 44 of 67
alpha := sqrt(5*r1-436,3)/3
--R
--R
--R      +-----+
--R      1 3|  +-----+
--R  (44)  - \|5\|7633  - 436
--R      3
--R
--R                                          Type: RealClosure(Fraction(Integer))
--E 44

--S 45 of 67
beta := -sqrt(5*r1+436,3)/3
--R
--R
--R      +-----+
--R      1 3|  +-----+
--R  (45)  - - \|5\|7633  + 436
--R      3
--R
--R                                          Type: RealClosure(Fraction(Integer))
--E 45

--S 46 of 67
pol.(alpha+beta-1/3)
--R
--R
--R  (46)  0
--R
--R                                          Type: RealClosure(Fraction(Integer))
--E 46

--S 47 of 67
qol : UP(x,Ran) := x**5+10*x**3+20*x+22
--R
--R
--R      5      3
--R  (47)  x  + 10x  + 20x + 22
--R
--R                                          Type: UnivariatePolynomial(x,RealClosure(Fraction(Integer)))
--E 47

--S 48 of 67
r2 := sqrt(153)$Ran

```



```

--R
--R
--R      +----+
--R (48)  \|153
--R
--R                                          Type: RealClosure(Fraction(Integer))
--E 48

--S 49 of 67
alpha2 := sqrt(r2-11,5)
--R
--R
--R      +-----+
--R      5| +----+
--R (49)  \|\|153 - 11
--R
--R                                          Type: RealClosure(Fraction(Integer))
--E 49

--S 50 of 67
beta2 := -sqrt(r2+11,5)
--R
--R
--R      +-----+
--R      5| +----+
--R (50)  - \|\|153 + 11
--R
--R                                          Type: RealClosure(Fraction(Integer))
--E 50

--S 51 of 67
qol(alpha2+beta2)
--R
--R
--R (51)  0
--R
--R                                          Type: RealClosure(Fraction(Integer))
--E 51

--S 52 of 67
dst1:=sqrt(9+4*s2)=1+2*s2
--R
--R
--R      +-----+
--R      |  +-+      +-+
--R (52)  \|4\|2 + 9 = 2\|2 + 1
--R
--R                                          Type: Equation(RealClosure(Fraction(Integer)))
--E 52

--S 53 of 67
dst1::Boolean
--R
--R
--R (53)  true

```

```
--R
--E 53
```

Type: Boolean

```
--S 54 of 67
s6:Ran:=sqrt 6
```

```
--R
--R
--R      +-+
--R (54)  \|6
```

Type: RealClosure(Fraction(Integer))

```
--E 54
```

```
--S 55 of 67
dst2:=sqrt(5+2*s6)+sqrt(5-2*s6) = 2*s3
```

```
--R
--R
--R      +-----+ +-----+
--R      | +-+ | +-+ +-+
--R (55)  \|- 2\|6 + 5 + \|2\|6 + 5 = 2\|3
```

Type: Equation(RealClosure(Fraction(Integer)))

```
--E 55
```

```
--S 56 of 67
dst2::Boolean
```

```
--R
--R
--R (56)  true
```

Type: Boolean

```
--E 56
```

```
--S 57 of 67
s29:Ran:=sqrt 29
```

```
--R
--R
--R      ++++
--R (57)  \|29
```

Type: RealClosure(Fraction(Integer))

```
--E 57
```

```
--S 58 of 67
dst4:=sqrt(16-2*s29+2*sqrt(55-10*s29)) = sqrt(22+2*s5)-sqrt(11+2*s29)+s5
```

```
--R
--R
--R (58)
--R      +-----+
--R      | +-----+ +-----+ +-----+
--R      | | +-+ +-+ | +-+ | +-+ +-+
--R \|2\|- 10\|29 + 55 - 2\|29 + 16 = - \|2\|29 + 11 + \|2\|5 + 22 + \|5
```

Type: Equation(RealClosure(Fraction(Integer)))

```
--E 58
```

```

--S 59 of 67
dst4::Boolean
--R
--R
--R (59) true
--R
--R                                          Type: Boolean
--E 59

```

```

--S 60 of 67
dst6:=sqrt((112+70*s2)+(46+34*s2)*s5) = (5+4*s2)+(3+s2)*s5
--R
--R
--R          +-----+
--R          |  +-+      +-+      +-+      +-+      +-+      +-+
--R (60)  \|(34\|2  + 46)\|5  + 70\|2  + 112 = (\|2  + 3)\|5  + 4\|2  + 5
--R
--R                                          Type: Equation(RealClosure(Fraction(Integer)))
--E 60

```

```

--S 61 of 67
dst6::Boolean
--R
--R
--R (61) true
--R
--R                                          Type: Boolean
--E 61

```

```

--S 62 of 67
f3:Ran:=sqrt(3,5)
--R
--R
--R          5+-+
--R (62)  \|3
--R
--R                                          Type: RealClosure(Fraction(Integer))
--E 62

```

```

--S 63 of 67
f25:Ran:=sqrt(1/25,5)
--R
--R
--R          +---+
--R          | 1
--R (63)  5|--
--R          \|25
--R
--R                                          Type: RealClosure(Fraction(Integer))
--E 63

```

```

--S 64 of 67
f32:Ran:=sqrt(32/5,5)
--R

```

```

--R
--R      +---+
--R      |32
--R  (64) 5|--
--R      \| 5
--R
--R                                          Type: RealClosure(Fraction(Integer))
--E 64

```

```

--S 65 of 67
f27:Ran:=sqrt(27/5,5)
--R
--R
--R      +---+
--R      |27
--R  (65) 5|--
--R      \| 5
--R
--R                                          Type: RealClosure(Fraction(Integer))
--E 65

```

```

--S 66 of 67
dst5:=sqrt((f32-f27,3)) = f25*(1+f3-f3**2)
--R
--R
--R      +-----+
--R      |  +---+  +---+
--R      |  |27  |32      5+--+2  5+--+      +---+
--R  (66) 3|- 5|-- + 5|-- = (- \|3  + \|3  + 1) 5|--
--R      \|  \| 5  \| 5      \|25
--R
--R                                          Type: Equation(RealClosure(Fraction(Integer)))
--E 66

```

```

--S 67 of 67
dst5:Boolean
--R
--R
--R  (67) true
--R
--R                                          Type: Boolean
--E 67
)spool
)lisp (bye)

```

— RealClosure.help —

```

=====
RealClosure examples
=====

```

The Real Closure 1.0 package provided by Renaud Rioboo consists of different packages, categories and domains :

The package `RealPolynomialUtilitiesPackage` which needs a `Field F` and a `UnivariatePolynomialCategory` domain with coefficients in `F`. It computes some simple functions such as Sturm and Sylvester sequences `sturmSequence`, `sylvesterSequence`.

The category `RealRootCharacterizationCategory` provides abstract functions to work with "real roots" of univariate polynomials. These resemble variables with some functionality needed to compute important operations.

The category `RealClosedField` provides common operations available over real closed fields. These include finding all the roots of a univariate polynomial, taking square (and higher) roots, ...

The domain `RightOpenIntervalRootCharacterization` is the main code that provides the functionality of `RealRootCharacterizationCategory` for the case of archimedean fields. Abstract roots are encoded with a left closed right open interval containing the root together with a defining polynomial for the root.

The `RealClosure` domain is the end-user code. It provides usual arithmetic with real algebraic numbers, along with the functionality of a real closed field. It also provides functions to approximate a real algebraic number by an element of the base field. This approximation may either be absolute, approximate or relative (`relativeApprox`).

CAVEATS

Since real algebraic expressions are stored as depending on "real roots" which are managed like variables, there is an ordering on these. This ordering is dynamical in the sense that any new algebraic takes precedence over older ones. In particular every creation function raises a new "real root". This has the effect that when you type something like `sqrt(2) + sqrt(2)` you have two new variables which happen to be equal. To avoid this name the expression such as in `s2 := sqrt(2) ; s2 + s2`

Also note that computing times depend strongly on the ordering you implicitly provide. Please provide algebraics in the order which seems most natural to you.

LIMITATIONS

This packages uses algorithms which are published in [1] and [2] which are based on field arithmetics, in particular for polynomial gcd related algorithms. This can be quite slow for high degree polynomials and subresultants methods usually work best. Beta versions of the package try to use these techniques in a better way and work significantly faster. These are mostly based on unpublished algorithms and cannot be distributed. Please contact the author if you have a particular problem to solve or want to use these versions.

Be aware that approximations behave as post-processing and that all computations are done exactly. They can thus be quite time consuming when depending on several ‘‘real roots’’.

REFERENCES

- [1] R. Rioboo : Real Algebraic Closure of an ordered Field : Implementation in Axiom.
In proceedings of the ISSAC’92 Conference, Berkeley 1992 pp. 206-215.
- [2] Z. Ligatsikas, R. Rioboo, M. F. Roy : Generic computation of the real closure of an ordered field.
In Mathematics and Computers in Simulation Volume 42, Issue 4-6, November 1996.

EXAMPLES

We shall work with the real closure of the ordered field of rational numbers.

```
Ran := RECLOS(FRAC INT)
RealClosure Fraction Integer
Type: Domain
```

Some simple signs for square roots, these correspond to an extension of degree 16 of the rational numbers. Examples provided by J. Abbot.

```
fourSquares(a:Ran,b:Ran,c:Ran,d:Ran):Ran==sqrt(a)+sqrt(b)-sqrt(c)-sqrt(d)
Type: Void
```

These produce values very close to zero.

```
squareDiff1 := fourSquares(73,548,60,586)
+---+ +---+ +---+ +---+
- \|586 - \|60 + \|548 + \|73
Type: RealClosure Fraction Integer
```

```

recip(squareDiff1)
      +----+      +---+ +---+      +---+ +----+      +----+
      ((54602\|548 + 149602\|73 )\|60 + 49502\|73 \|548 + 9900895)\|586
+
      +---+ +----+      +---+      +----+      +---+
      (154702\|73 \|548 + 30941947)\|60 + 10238421\|548 + 28051871\|73
      Type: Union(RealClosure Fraction Integer,...)

sign(squareDiff1)
1
      Type: PositiveInteger

squareDiff2 := fourSquares(165,778,86,990)
      +----+ +---+ +----+ +----+
      - \|990 - \|86 + \|778 + \|165
      Type: RealClosure Fraction Integer

recip(squareDiff2)
      +----+      +----+ +---+      +----+ +----+
      ((556778\|778 + 1209010\|165 )\|86 + 401966\|165 \|778 + 144019431)
*
      +----+
      \|990
+
      +----+ +----+      +---+      +----+      +----+
      (1363822\|165 \|778 + 488640503)\|86 + 162460913\|778 + 352774119\|165
      Type: Union(RealClosure Fraction Integer,...)

sign(squareDiff2)
1
      Type: PositiveInteger

squareDiff3 := fourSquares(217,708,226,692)
      +----+ +----+ +----+ +----+
      - \|692 - \|226 + \|708 + \|217
      Type: RealClosure Fraction Integer

recip(squareDiff3)
      +----+      +----+ +----+      +----+ +----+      +----+
      ((- 34102\|708 - 61598\|217 )\|226 - 34802\|217 \|708 - 13641141)\|692
+
      +----+ +----+      +----+      +----+      +----+
      (- 60898\|217 \|708 - 23869841)\|226 - 13486123\|708 - 24359809\|217
      Type: Union(RealClosure Fraction Integer,...)

sign(squareDiff3)
- 1
      Type: Integer

```

```

squareDiff4 := fourSquares(155,836,162,820)
      +----+      +----+      +----+      +----+
      - \|820  - \|162  + \|836  + \|155
                                     Type: RealClosure Fraction Integer

recip(squareDiff4)
      +----+      +----+ +----+      +----+ +----+      +----+
      ((- 37078\|836  - 86110\|155 )\|162  - 37906\|155 \|836  - 13645107)\|820
+
      +----+ +----+      +----+      +----+      +----+
      (- 85282\|155 \|836  - 30699151)\|162  - 13513901\|836  - 31384703\|155
                                     Type: Union(RealClosure Fraction Integer,...)

sign(squareDiff4)
- 1
                                     Type: Integer

squareDiff5 := fourSquares(591,772,552,818)
      +----+      +----+      +----+      +----+
      - \|818  - \|552  + \|772  + \|591
                                     Type: RealClosure Fraction Integer

recip(squareDiff5)
      +----+      +----+ +----+      +----+ +----+      +----+
      ((70922\|772  + 81058\|591 )\|552  + 68542\|591 \|772  + 46297673)\|818
+
      +----+ +----+      +----+      +----+      +----+
      (83438\|591 \|772  + 56359389)\|552  + 47657051\|772  + 54468081\|591
                                     Type: Union(RealClosure Fraction Integer,...)

sign(squareDiff5)
1
                                     Type: PositiveInteger

squareDiff6 := fourSquares(434,1053,412,1088)
      +-----+      +----+      +-----+      +----+
      - \|1088  - \|412  + \|1053  + \|434
                                     Type: RealClosure Fraction Integer

recip(squareDiff6)
      +-----+      +----+ +----+      +----+ +-----+
      ((115442\|1053  + 179818\|434 )\|412  + 112478\|434 \|1053  + 76037291)
*
      +-----+
      \|1088
+
      +----+ +-----+      +----+      +-----+      +----+
      (182782\|434 \|1053  + 123564147)\|412  + 77290639\|1053  + 120391609\|434
                                     Type: Union(RealClosure Fraction Integer,...)

```



```

sign(squareDiff6)
1
Type: PositiveInteger

squareDiff7 := fourSquares(514,1049,446,1152)
+-----+ +-----+ +-----+ +-----+
- \|1152 - \|446 + \|1049 + \|514
Type: RealClosure Fraction Integer

recip(squareDiff7)
+-----+ +-----+ +-----+ +-----+ +-----+
((349522\|1049 + 499322\|514 )\|446 + 325582\|514 \|1049 + 239072537)
*
+-----+
\|1152
+
+-----+ +-----+ +-----+ +-----+ +-----+
(523262\|514 \|1049 + 384227549)\|446 + 250534873\|1049 + 357910443\|514
Type: Union(RealClosure Fraction Integer,...)

sign(squareDiff7)
1
Type: PositiveInteger

squareDiff8 := fourSquares(190,1751,208,1698)
+-----+ +-----+ +-----+ +-----+
- \|1698 - \|208 + \|1751 + \|190
Type: RealClosure Fraction Integer

recip(squareDiff8)
+-----+ +-----+ +-----+ +-----+ +-----+
(- 214702\|1751 - 651782\|190 )\|208 - 224642\|190 \|1751
+
- 129571901
*
+-----+
\|1698
+
+-----+ +-----+ +-----+ +-----+ +-----+
(- 641842\|190 \|1751 - 370209881)\|208 - 127595865\|1751
+
+-----+
- 387349387\|190
Type: Union(RealClosure Fraction Integer,...)

sign(squareDiff8)
- 1
Type: Integer

```

This should give three digits of precision

```
relativeApprox(squareDiff8,10**(-3))::Float
- 0.2340527771 5937700123 E -10
Type: Float
```

The sum of these 4 roots is 0

```
l := allRootsOf((x**2-2)**2-2)$Ran
[%A33,%A34,%A35,%A36]
Type: List RealClosure Fraction Integer
```

Check that they are all roots of the same polynomial

```
removeDuplicates map(mainDefiningPolynomial,l)
4      2
[? - 4? + 2]
Type: List Union(
  SparseUnivariatePolynomial RealClosure Fraction Integer,
  "failed")
```

We can see at a glance that they are separate roots

```
map(mainCharacterization,l)
[[- 2,- 1],[- 1,0],[0,1],[1,2]]
Type: List Union(
  RightOpenIntervalRootCharacterization(
    RealClosure Fraction Integer,
    SparseUnivariatePolynomial RealClosure Fraction Integer),
  "failed")
```

Check the sum and product

```
[reduce(+,l),reduce(*,l)-2]
[0,0]
Type: List RealClosure Fraction Integer
```

A more complicated test that involve an extension of degree 256.
This is a way of checking nested radical identities.

```
(s2, s5, s10) := (sqrt(2)$Ran, sqrt(5)$Ran, sqrt(10)$Ran)
+---+
\|10
Type: RealClosure Fraction Integer
```

```
eq1:=sqrt(s10+3)*sqrt(s5+2) - sqrt(s10-3)*sqrt(s5-2) = sqrt(10*s2+10)
+-----+ +-----+ +-----+ +-----+ +-----+
| +---+ | +-+ | +---+ | +-+ | +-+
- \| \|10 - 3 \| \|5 - 2 + \| \|10 + 3 \| \|5 + 2 = \|10 \|2 + 10
Type: Equation RealClosure Fraction Integer
```

```
eq1::Boolean
true
```

Type: Boolean

```
eq2:=sqrt(s5+2)*sqrt(s2+1) - sqrt(s5-2)*sqrt(s2-1) = sqrt(2*s10+2)
      +-----+ +-----+      +-----+ +-----+ +-----+
      | +-+      | +-+      | +-+      | +-+      | +---+
      - \|\|5  - 2 \|\|2  - 1  + \|\|5  + 2 \|\|2  + 1 = \|2\|10  + 2
                                     Type: Equation RealClosure Fraction Integer
```

```
eq2::Boolean
true
```

Type: Boolean

Some more examples from J. M. Arnaudies

```
s3 := sqrt(3)$Ran
      +-+
      \|3
```

Type: RealClosure Fraction Integer

```
s7:= sqrt(7)$Ran
      +-+
      \|7
```

Type: RealClosure Fraction Integer

```
e1 := sqrt(2*s7-3*s3,3)
      +-----+
      3| +-+      +-+
      \|2\|7  - 3\|3
```

Type: RealClosure Fraction Integer

```
e2 := sqrt(2*s7+3*s3,3)
      +-----+
      3| +-+      +-+
      \|2\|7  + 3\|3
```

Type: RealClosure Fraction Integer

This should be null

```
e2-e1-s3
0
```

Type: RealClosure Fraction Integer

A quartic polynomial

```
pol : UP(x,Ran) := x**4+(7/3)*x**2+30*x-(100/3)
      4      7      2      100
      x  + - x  + 30x - ---
      3          3
```

Type: UnivariatePolynomial(x,RealClosure Fraction Integer)

Add some cubic roots

r1 := sqrt(7633)\$Ran

$$\sqrt[3]{7633}$$

Type: RealClosure Fraction Integer

alpha := sqrt(5*r1-436,3)/3

$$\sqrt[3]{\frac{5\sqrt[3]{7633} - 436}{3}}$$

Type: RealClosure Fraction Integer

beta := -sqrt(5*r1+436,3)/3

$$-\sqrt[3]{\frac{5\sqrt[3]{7633} + 436}{3}}$$

Type: RealClosure Fraction Integer

this should be null

pol.(alpha+beta-1/3)
 0

Type: RealClosure Fraction Integer

A quintic polynomial

qol : UP(x,Ran) := x**5+10*x**3+20*x+22

$$x^5 + 10x^3 + 20x + 22$$

Type: UnivariatePolynomial(x,RealClosure Fraction Integer)

Add some cubic roots

r2 := sqrt(153)\$Ran

$$\sqrt[3]{153}$$

Type: RealClosure Fraction Integer

alpha2 := sqrt(r2-11,5)

$$\sqrt[5]{\sqrt[3]{153} - 11}$$

Type: RealClosure Fraction Integer

beta2 := -sqrt(r2+11,5)

```

+-----+
5| +---+
- \|\|153 + 11

```

Type: RealClosure Fraction Integer

this should be null

```

qol(alpha2+beta2)
0

```

Type: RealClosure Fraction Integer

Finally, some examples from the book Computer Algebra by Davenport, Siret and Tournier (page 77). The last one is due to Ramanujan.

dst1:=sqrt(9+4*s2)=1+2*s2

```

+-----+
| +-+ +-+
\|4\|2 + 9 = 2\|2 + 1

```

Type: Equation RealClosure Fraction Integer

```

dst1::Boolean
true

```

Type: Boolean

```

s6:Ran:=sqrt 6
+-+
\|6

```

Type: RealClosure Fraction Integer

dst2:=sqrt(5+2*s6)+sqrt(5-2*s6) = 2*s3

```

+-----+ +-----+
| +-+ | +-+ +-+
\|- 2\|6 + 5 + \|2\|6 + 5 = 2\|3

```

Type: Equation RealClosure Fraction Integer

```

dst2::Boolean
true

```

Type: Boolean

```

s29:Ran:=sqrt 29
+---+
\|29

```

Type: RealClosure Fraction Integer

dst4:=sqrt(16-2*s29+2*sqrt(55-10*s29)) = sqrt(22+2*s5)-sqrt(11+2*s29)+s5

```

+-----+
| +-----+ +-----+ +-----+
| | +-+ +-+ | +-+ | +-+ +-+
\|2\|- 10\|29 + 55 - 2\|29 + 16 = - \|2\|29 + 11 + \|2\|5 + 22 + \|5

```

Type: Equation RealClosure Fraction Integer

dst4::Boolean

true

Type: Boolean

dst6:=sqrt((112+70*s2)+(46+34*s2)*s5) = (5+4*s2)+(3+s2)*s5

+-----+

| +-+ +-+ +-+ +-+ +-+ +-+
 $\sqrt{(34\sqrt{2} + 46)\sqrt{5} + 70\sqrt{2} + 112} = (\sqrt{2} + 3)\sqrt{5} + 4\sqrt{2} + 5$

Type: Equation RealClosure Fraction Integer

dst6::Boolean

true

Type: Boolean

f3:Ran:=sqrt(3,5)

5+-+

$\sqrt{3}$

Type: RealClosure Fraction Integer

f25:Ran:=sqrt(1/25,5)

+---+

| 1

5|--

$\sqrt{25}$

Type: RealClosure Fraction Integer

f32:Ran:=sqrt(32/5,5)

+---+

|32

5|--

$\sqrt{5}$

Type: RealClosure Fraction Integer

f27:Ran:=sqrt(27/5,5)

+---+

|27

5|--

$\sqrt{5}$

Type: RealClosure Fraction Integer

dst5:=sqrt((f32-f27,3)) = f25*(1+f3-f3**2)

+-----+

| +---+ +---+ 5+-+2 5+-+ +---+

| |27 |32 5+-+2 5+-+ | 1

3|- 5|-- + 5|-- = (- $\sqrt{3}$ + $\sqrt{3}$ + 1) 5|--

$\sqrt{\sqrt{5} \sqrt{5} \sqrt{5}} = (-\sqrt{3} + \sqrt{3} + 1)\sqrt{25}$

Type: Equation RealClosure Fraction Integer

dst5::Boolean

`true`

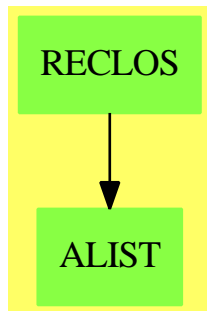
Type: Boolean

See Also:

- o `)help RightOpenIntervalRootCharacterization`
- o `)help RealClosedField`
- o `)help RealRootCharacterizationCategory`
- o `)help UnivariatePolynomialCategory`
- o `)help Field`
- o `)help RealPolynomialUtilitiesPackage`
- o `)show RealClosure`

—————

RealClosure (RECLOS)



See

⇒ “RightOpenIntervalRootCharacterization” (ROIRC) [19](#) on page [2310](#)

Exports:

0	1	abs	algebraicOf
allRootsOf	approximate	associates?	characteristic
coerce	divide	euclideanSize	expressIdealMember
exquo	extendedEuclidean	factor	gcd
gcdPolynomial	hash	inv	latex
lcm	mainCharacterization	mainDefiningPolynomial	mainForm
mainValue	max	multiEuclidean	min
negative?	nthRoot	one?	positive?
prime?	principalIdeal	recip	relativeApprox
rename	rename!	retract	retractIfCan
rootOf	sample	sign	sizeLess?
sqrt	squareFree	squareFreePart	subtractIfCan
unit?	unitCanonical	unitNormal	zero?
?*?	?**?	?+?	?-?
-?	?/?	?<?	?<=?
?=?	?>?	?>=?	?^?
?~=?	?quo?	?rem?	

— domain RECLOS RealClosure —

```
)abbrev domain RECLOS RealClosure
++ Author: Renaud Rioboo
++ Date Created: summer 1988
++ Date Last Updated: January 2004
++ Description:
++ This domain implements the real closure of an ordered field.
++ Note:
++ The code here is generic i.e. it does not depend of the way the operations
++ are done. The two macros PME and SEG should be passed as functorial
++ arguments to the domain. It does not help much to write a category
++ since non trivial methods cannot be placed there either.
++
```

```
RealClosure(TheField): PUB == PRIV where
```

```
    TheField    : Join(OrderedRing, Field, RealConstant)
```

```
-- ThePols      : UnivariatePolynomialCategory($)
-- PME          ==> ThePols
-- TheCharDom    : RealRootCharacterizationCategory($, ThePols )
-- SEG          ==> TheCharDom
-- this does not work yet
```

```
E          ==> OutputForm
Z          ==> Integer
SE         ==> Symbol
B          ==> Boolean
SUP        ==> SparseUnivariatePolynomial($)
N          ==> PositiveInteger
```



```

RN      ==> Fraction Z
LF      ==> ListFunctions2($,N)
PME     ==> SparseUnivariatePolynomial($)
SEG     ==> RightOpenIntervalRootCharacterization($,PME)

PUB == Join(RealClosedField,
            FullyRetractableTo TheField,
            Algebra TheField) with

algebraicOf : (SEG,E) -> $
  ++ \axiom{algebraicOf(char)} is the external number

mainCharacterization : $ -> Union(SEG,"failed")
  ++ \axiom{mainCharacterization(x)} is the main algebraic
  ++ quantity of \axiom{x} (\axiom{SEG})

relativeApprox : ($,$) -> RN
  ++ \axiom{relativeApprox(n,p)} gives a relative
  ++ approximation of \axiom{n}
  ++ that has precision \axiom{p}

PRIV == add

-- local functions

lessAlgebraic : $ -> $
newElementIfneeded : (SEG,E) -> $

-- Representation

Rec := Record(seg: SEG, val:PME, outForm:E, order:N)
Rep := Union(TheField,Rec)

-- global (mutable) variables

orderOfCreation : N := 1$N
  -- it is internally used to sort the algebraic levels

instanceName : Symbol := new()$Symbol
  -- this used to print the results, thus different instantiations
  -- use different names

-- now the code

relativeApprox(nbe,prec) ==
  nbe case TheField => retract(nbe)
  appr := relativeApprox(nbe.val, nbe.seg, prec)
  -- now appr has the good exact precision but is $
  relativeApprox(appr,prec)

```

```

approximate(nbe,prec) ==
  abs(nbe) < prec => 0
  nbe case TheField => retract(nbe)
  appr := approximate(nbe.val, nbe.seg, prec)
  -- now appr has the good exact precision but is $
  approximate(appr,prec)

newElementIfneeded(s,o) ==
  p := definingPolynomial(s)
  degree(p) = 1 =>
    - coefficient(p,0) / leadingCoefficient(p)
  res := [s, monomial(1,1), o, orderOfCreation ]$Rec
  orderOfCreation := orderOfCreation + 1
  res :: $

algebraicOf(s,o) ==
  pol := definingPolynomial(s)
  degree(pol) = 1 =>
    -coefficient(pol,0) / leadingCoefficient(pol)
  res := [s, monomial(1,1), o, orderOfCreation ]$Rec
  orderOfCreation := orderOfCreation + 1
  res :: $

rename!(x,o) ==
  x.outForm := o
  x

rename(x,o) ==
  [x.seg, x.val, o, x.order]$Rec

rootOf(pol,n) ==
  degree(pol) = 0 => "failed"
  degree(pol) = 1 =>
    if n=1
    then
      -coefficient(pol,0) / leadingCoefficient(pol)
    else
      "failed"
  r := rootOf(pol,n)$SEG
  r case "failed" => "failed"
  o := hconcat(instanceName :: E , orderOfCreation :: E)$E
  algebraicOf(r,o)

allRootsOf(pol:SUP):List($$) ==
  degree(pol)=0 => []
  degree(pol)=1 => [-coefficient(pol,0) / leadingCoefficient(pol)]
  liste := allRootsOf(pol)$SEG
  res : List $$ := []
  for term in liste repeat

```

```

    o := hconcat(instanceName :: E , orderOfCreation :: E)$E
    res := cons(algebraicOf(term,o), res)
reverse! res

coerce(x:$):$ ==
  x case TheField => x
  [x.seg,x.val rem$PME definingPolynomial(x.seg),x.outForm,x.order]$Rec

positive?(x) ==
  x case TheField => positive?(x)$TheField
  positive?(x.val,x.seg)$SEG

negative?(x) ==
  x case TheField => negative?(x)$TheField
  negative?(x.val,x.seg)$SEG

abs(x) == sign(x)*x

sign(x) ==
  x case TheField => sign(x)$TheField
  sign(x.val,x.seg)$SEG

x < y == positive?(y-x)

x = y == zero?(x-y)

mainCharacterization(x) ==
  x case TheField => "failed"
  x.seg

mainDefiningPolynomial(x) ==
  x case TheField => "failed"
  definingPolynomial x.seg

mainForm(x) ==
  x case TheField => "failed"
  x.outForm

mainValue(x) ==
  x case TheField => "failed"
  x.val

coerce(x:$):E ==
  x case TheField => x::TheField :: E
  xx:$ := coerce(x)
  outputForm(univariate(xx.val),x.outForm)$SUP

inv(x) ==
  (res:= recip x) case "failed" => error "Division by 0"

```

```

    res :: $

recip(x) ==
  x case TheField =>
    if ((r := recip(x)$TheField) case TheField)
    then r::$
    else "failed"
  if ((r := recip(x.val,x.seg)$SEG) case "failed")
  then "failed"
  else lessAlgebraic([x.seg,r::PME,x.outForm,x.order]$Rec)

(n:Z * x:$):$ ==
  x case TheField => n *$TheField x
  zero?(n) => 0
  one?(n) => x
  [x.seg,map(z+>n*z, x.val),x.outForm,x.order]$Rec

(rn:TheField * x:$):$ ==
  x case TheField => rn *$TheField x
  zero?(rn) => 0
  one?(rn) => x
  [x.seg,map(z+>rn*z, x.val),x.outForm,x.order]$Rec

(x:$ * y:$):$ ==
  (x case TheField) and (y case TheField) => x *$TheField y
  (x case TheField) => x::TheField * y
    -- x is no longer TheField
  (y case TheField) => y::TheField * x
    -- now both are algebraic
  y.order > x.order =>
    [y.seg,map(z+>x*z, y.val),y.outForm,y.order]$Rec
  x.order > y.order =>
    [x.seg,map(z+>z*y, x.val),x.outForm,x.order]$Rec
    -- now x.exp = y.exp
    -- we will multiply the polynomials and then reduce
    -- however we need to call lessAlgebraic
  lessAlgebraic([x.seg,
    (x.val * y.val) rem definingPolynomial(x.seg),
    x.outForm,
    x.order]$Rec)

nonNull(rep:Rec):$ ==
  degree(rep.val)=0 => leadingCoefficient(rep.val)
  numberOfMonomials(rep.val) = 1 => rep
  zero?(rep.val,rep.seg)$SEG => 0
  rep

--
-- zero?(x) ==
--   x case TheField => zero?(x)$TheField
--   zero?(x.val,x.seg)$SEG

```

```

zero?(x) ==
  x case TheField => zero?(x)$TheField
  false

x + y ==
  (x case TheField) and (y case TheField) => x +$TheField y
  (x case TheField) =>
    if zero?(x)
    then
      y
    else
      nonNull([y.seg,x::PME+(y.val),y.outForm,y.order]$Rec)
      -- x is no longer TheField
  (y case TheField) =>
    if zero?(y)
    then
      x
    else
      nonNull([x.seg,(x.val)+y::PME,x.outForm,x.order]$Rec)
      -- now both are algebraic
  y.order > x.order =>
    nonNull([y.seg,x::PME+y.val,y.outForm,y.order]$Rec)
  x.order > y.order =>
    nonNull([x.seg,(x.val)+y::PME,x.outForm,x.order]$Rec)
    -- now x.exp = y.exp
    -- we simply add polynomials (since degree cannot increase)
    -- however we need to call lessAlgebraic
  nonNull([x.seg,x.val + y.val,x.outForm,x.order])

-x ==
  x case TheField => -$TheField (x::TheField)
  [x.seg,-$PME x.val,x.outForm,x.order]$Rec

retractIfCan(x:$):Union(TheField,"failed") ==
  x case TheField => x
  o := x.order
  res := lessAlgebraic x
  res case TheField => res
  o = res.order => "failed"
  retractIfCan res

retract(x:$):TheField ==
  x case TheField => x
  o := x.order
  res := lessAlgebraic x
  res case TheField => res
  o = res.order => error "Can't retract"

```

```

    retract res

lessAlgebraic(x) ==
  x case TheField => x
  degree(x.val) = 0 => leadingCoefficient(x.val)
  def := definingPolynomial(x.seg)
  degree(def) = 1 =>
    x.val.(- coefficient(def,0) / leadingCoefficient(def))
  x

0 == (0$TheField) :: $

1 == (1$TheField) :: $

coerce(rn:TheField):$ == rn :: $

```

— RECLOS.dotabb —

```

"RECLOS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=RECLOS"]
"ALIST"  [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"RECLOS" -> "ALIST"

```

domain RMATRIX RectangularMatrix

— RectangularMatrix.input —

```

)set break resume
)sys rm -f RectangularMatrix.output
)spool RectangularMatrix.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RectangularMatrix
--R
--R RectangularMatrix(m: NonNegativeInteger,n: NonNegativeInteger,R: Ring) is a domain constructor
--R Abbreviation for RectangularMatrix is RMATRIX
--R This constructor is not exposed in this frame.

```

```

--R Issue )edit bookvol10.3.pamphlet to see algebra source code for RMATRIX
--R
--R----- Operations -----
--R ??? : (%,R) -> %                ??? : (R,%) -> %
--R ??? : (Integer,%) -> %          ??? : (PositiveInteger,%) -> %
--R ?+? : (%,%) -> %                ?-? : (%,%) -> %
--R -? : % -> %                      ?=? : (%,%) -> Boolean
--R 0 : () -> %                      antisymmetric? : % -> Boolean
--R coerce : % -> Matrix(R)          coerce : % -> OutputForm
--R copy : % -> %                     diagonal? : % -> Boolean
--R elt : (%,Integer,Integer,R) -> R  elt : (%,Integer,Integer) -> R
--R empty : () -> %                   empty? : % -> Boolean
--R eq? : (%,%) -> Boolean            hash : % -> SingleInteger
--R latex : % -> String               listOfLists : % -> List(List(R))
--R map : ((R,R) -> R,%,%) -> %       map : ((R -> R),%) -> %
--R matrix : List(List(R)) -> %       maxColIndex : % -> Integer
--R maxRowIndex : % -> Integer         minColIndex : % -> Integer
--R minRowIndex : % -> Integer         ncols : % -> NonNegativeInteger
--R nrows : % -> NonNegativeInteger    qelt : (%,Integer,Integer) -> R
--R sample : () -> %                  square? : % -> Boolean
--R symmetric? : % -> Boolean          zero? : % -> Boolean
--R ?~=? : (%,%) -> Boolean
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ??? : (NonNegativeInteger,%) -> %
--R ?/? : (%,R) -> % if R has FIELD
--R any? : ((R -> Boolean),%) -> Boolean if $ has finiteAggregate
--R column : (%,Integer) -> DirectProduct(m,R)
--R convert : % -> InputForm if R has KONVERT(INFORM)
--R count : ((R -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R count : (R,%) -> NonNegativeInteger if $ has finiteAggregate and R has SETCAT
--R dimension : () -> CardinalNumber if R has FIELD
--R eval : (%,List(Equation(R))) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (%,Equation(R)) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (%,R,R) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (%,List(R),List(R)) -> % if R has EVALAB(R) and R has SETCAT
--R every? : ((R -> Boolean),%) -> Boolean if $ has finiteAggregate
--R exquo : (%,R) -> Union(%, "failed") if R has INTDOM
--R less? : (%,NonNegativeInteger) -> Boolean
--R map! : ((R -> R),%) -> % if $ has shallowlyMutable
--R member? : (R,%) -> Boolean if $ has finiteAggregate and R has SETCAT
--R members : % -> List(R) if $ has finiteAggregate
--R more? : (%,NonNegativeInteger) -> Boolean
--R nullSpace : % -> List(DirectProduct(m,R)) if R has INTDOM
--R nullity : % -> NonNegativeInteger if R has INTDOM
--R parts : % -> List(R) if $ has finiteAggregate
--R rank : % -> NonNegativeInteger if R has INTDOM
--R rectangularMatrix : Matrix(R) -> %
--R row : (%,Integer) -> DirectProduct(n,R)
--R rowEchelon : % -> % if R has EUCDOM
--R size? : (%,NonNegativeInteger) -> Boolean

```

```
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R
--E 1
```

```
)spool
)lisp (bye)
```

— RectangularMatrix.help —

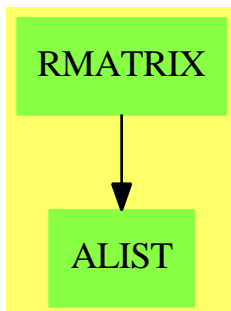
=====
 RectangularMatrix examples
 =====

RectangularMatrix is a matrix domain where the number of rows
 and the number of columns are parameters of the domain.

See Also:

o)show RectangularMatrix

RectangularMatrix (RMATRIX)



See

⇒ “IndexedMatrix” (IMATRIX) [10](#) on page [1238](#)

⇒ “Matrix” (MATRIX) [14](#) on page [1622](#)

⇒ “SquareMatrix” (SQMATRIX) [20](#) on page [2547](#)

Exports:

0	antisymmetric?	any?	coerce	column
convert	copy	count	diagonal?	dimension
elt	empty	empty?	eq?	eval
every?	exquo	hash	latex	less?
listOfLists	map	map!	matrix	maxColIndex
maxRowIndex	member?	members	minColIndex	minRowIndex
more?	ncols	nrows	nullSpace	nullity
parts	qelt	rank	rectangularMatrix	row
rowEchelon	sample	size?	square?	subtractIfCan
symmetric?	zero?	#?	?~=?	?*?
?/?	?+?	?-?	-?	?=?

— domain RMATRIX RectangularMatrix —

```

)abbrev domain RMATRIX RectangularMatrix
++ Author: Grabmeier, Gschnitzer, Williamson
++ Date Created: 1987
++ Date Last Updated: July 1990
++ Description:
++ \spadtype{RectangularMatrix} is a matrix domain where the number of rows
++ and the number of columns are parameters of the domain.

RectangularMatrix(m,n,R): Exports == Implementation where
  m,n : NonNegativeInteger
  R   : Ring
  Row ==> DirectProduct(n,R)
  Col ==> DirectProduct(m,R)
  Exports ==> Join(RectangularMatrixCategory(m,n,R,Row,Col),_
    CoercibleTo Matrix R) with

  if R has Field then VectorSpace R

  if R has ConvertibleTo InputForm then ConvertibleTo InputForm

  rectangularMatrix: Matrix R -> $
    ++ \spad{rectangularMatrix(m)} converts a matrix of type \spadtype{Matrix}
    ++ to a matrix of type \spad{RectangularMatrix}.
  coerce: $ -> Matrix R
    ++ \spad{coerce(m)} converts a matrix of type \spadtype{RectangularMatrix}
    ++ to a matrix of type \spad{Matrix}.

Implementation ==> Matrix R add
  minr ==> minRowIndex
  maxr ==> maxRowIndex
  minc ==> minColIndex
  maxc ==> maxColIndex
  mini ==> minIndex
  maxi ==> maxIndex

```

```

ZERO := new(m,n,0)$Matrix(R) pretend $
0    == ZERO

coerce(x:$):OutputForm == coerce(x pretend Matrix R)$Matrix(R)

matrix(l: List List R) ==
  -- error check: this is a top level function
  #l ^= m => error "matrix: wrong number of rows"
  for ll in l repeat
    #ll ^= n => error "matrix: wrong number of columns"
  ans : Matrix R := new(m,n,0)
  for i in minr(ans)..maxr(ans) for ll in l repeat
    for j in minc(ans)..maxc(ans) for r in ll repeat
      qsetelt_!(ans,i,j,r)
  ans pretend $

row(x,i)    == directProduct row(x pretend Matrix(R),i)
column(x,j) == directProduct column(x pretend Matrix(R),j)

coerce(x:$):Matrix(R) == copy(x pretend Matrix(R))

rectangularMatrix x ==
  (nrows(x) ^= m) or (ncols(x) ^= n) =>
    error "rectangularMatrix: matrix of bad dimensions"
  copy(x) pretend $

if R has EuclideanDomain then

  rowEchelon x == rowEchelon(x pretend Matrix(R)) pretend $

if R has IntegralDomain then

  rank x    == rank(x pretend Matrix(R))
  nullity x == nullity(x pretend Matrix(R))
  nullSpace x ==
    [directProduct c for c in nullSpace(x pretend Matrix(R))]]

if R has Field then

  dimension() == (m * n) :: CardinalNumber

if R has ConvertibleTo InputForm then
  convert(x:$):InputForm ==
    convert [convert("rectangularMatrix"::Symbol)@InputForm,
             convert(x::Matrix(R)):$List(InputForm)]

```

— RMATRIX.dotabb —

```
"RMATRIX" [color="#88FF44",href="bookvol10.3.pdf#nameddest=RMATRIX"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"RMATRIX" -> "ALIST"
```

domain REF Reference

— Reference.input —

```
)set break resume
)sys rm -f Reference.output
)spool Reference.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Reference
--R
--R Reference(S: Type) is a domain constructor
--R Abbreviation for Reference is REF
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for REF
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          deref : % -> S
--R elt : % -> S                    ref : S -> %
--R setelt : (% ,S) -> S            setref : (% ,S) -> S
--R coerce : % -> OutputForm if S has SETCAT
--R hash : % -> SingleInteger if S has SETCAT
--R latex : % -> String if S has SETCAT
--R ?~=? : (%,% ) -> Boolean if S has SETCAT
--R
--E 1

)spool
)lisp (bye)
```

— Reference.help —

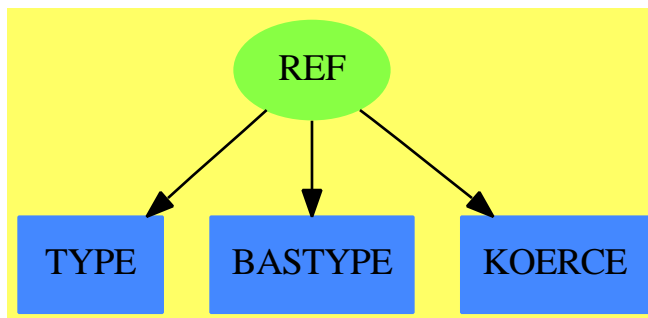
```
=====
Reference examples
=====
```

Reference is for making a changeable instance of something.

See Also:

o)show Reference

Reference (REF)



See

⇒ “Boolean” (BOOLEAN) [3](#) on page [322](#)

⇒ “IndexedBits” (IBITS) [10](#) on page [1199](#)

⇒ “Bits” (BITS) [3](#) on page [314](#)

Exports:

coerce	deref	elt	hash	latex
ref	setelt	setref	?=?	?~=?

— domain REF Reference —

```
)abbrev domain REF Reference
++ Author: Stephen M. Watt
++ Description:
++ \spadtype{Reference} is for making a changeable instance
++ of something.
```

```
Reference(S:Type): Type with
  ref    : S -> %
    ++ ref(n) creates a pointer (reference) to the object n.
  elt    : % -> S
    ++ elt(n) returns the object n.
```

```

setelt: (% , S) -> S
  ++ setelt(n,m) changes the value of the object n to m.
  -- alternates for when bugs don't allow the above
deref : % -> S
  ++ deref(n) is equivalent to \spad{elt(n)}.
setref: (% , S) -> S
  ++ setref(n,m) same as \spad{setelt(n,m)}.
_=      : (% , %) -> Boolean
  ++ a=b tests if \spad{a} and b are equal.
if S has SetCategory then SetCategory

== add
Rep := Record(value: S)

p = q      == EQ(p, q)$Lisp
ref v      == [v]
elt p      == p.value
setelt(p, v) == p.value := v
deref p    == p.value
setref(p, v) == p.value := v

if S has SetCategory then
  coerce p ==
    prefix(message("ref"@String), [p.value::OutputForm])



---


— REF.dotabb —

"REF" [color="#88FF44",href="bookvol10.3.pdf#nameddest=REF",shape=ellipse]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"REF" -> "TYPE"
"REF" -> "BASTYPE"
"REF" -> "KOERCE"



---



```

domain RGCHAIN RegularChain

— RegularChain.input —

```
)set break resume
```

```

)sys rm -f RegularChain.output
)spool RegularChain.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RegularChain
--R
--R RegularChain(R: GcdDomain,ls: List(Symbol)) is a domain constructor
--R Abbreviation for RegularChain is RGCHAIN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for RGCHAIN
--R
--R----- Operations -----
--R ==? : (% ,%) -> Boolean               coerce : % -> OutputForm
--R collectQuasiMonic : % -> %           copy : % -> %
--R degree : % -> NonNegativeInteger     empty : () -> %
--R empty? : % -> Boolean                eq? : (% ,%) -> Boolean
--R hash : % -> SingleInteger            headReduced? : % -> Boolean
--R infRittWu? : (% ,%) -> Boolean        initiallyReduced? : % -> Boolean
--R latex : % -> String                  normalized? : % -> Boolean
--R purelyAlgebraic? : % -> Boolean       rest : % -> Union(%, "failed")
--R sample : () -> %                    stronglyReduced? : % -> Boolean
--R trivialIdeal? : % -> Boolean          ~=? : (% ,%) -> Boolean
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R algebraic? : (OrderedVariableList(ls),%) -> Boolean
--R algebraicCoefficients? : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> Boolean
--R algebraicVariables : % -> List(OrderedVariableList(ls))
--R any? : ((NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)) -> Boolean),%) -> Boolean if $ has f
--R augment : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),List(%)) -> List(%)
--R augment : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),%) -> List(%)
--R augment : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),List(%)) -> List(%)
--R augment : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> List(%)
--R autoReduced? : (% ,((NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)))) -> Boolean),%) -> Boolean
--R basicSet : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))) -> List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)))
--R basicSet : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),((NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)))) -> List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)))
--R coHeight : % -> NonNegativeInteger if OrderedVariableList(ls) has FINITE
--R coerce : % -> List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)))
--R collect : (% ,OrderedVariableList(ls)) -> %
--R collectUnder : (% ,OrderedVariableList(ls)) -> %
--R collectUpper : (% ,OrderedVariableList(ls)) -> %
--R construct : List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))) -> %
--R convert : % -> InputForm if NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)) has KONVERT(INFORMAL)
--R count : ((NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)) -> Boolean),%) -> NonNegativeInteger
--R count : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> NonNegativeInteger if $ has f
--R eval : (% ,List(Equation(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)))) -> % if NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)) has FINITE
--R eval : (% ,Equation(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)))) -> % if NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)) has FINITE
--R eval : (% ,NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))) -> List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)))
--R eval : (% ,List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)))) -> List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)))

```

```

--R every? : ((NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)) -> Boolean),%) -> Boolean
--R extend : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),List(%)) -> List(%)
--R extend : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),%) -> List(%)
--R extend : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),List(%)) -> List(%)
--R extend : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> List(%)
--R extend : (%,NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))) -> %
--R extendIfCan : (%,NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))) -> Union(%, "failed")
--R find : ((NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)) -> Boolean),%) -> Union(%, "failed")
--R first : % -> Union(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)), "failed")
--R headReduce : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))
--R headReduced? : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> Boolean
--R headRemainder : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> Record(rnum: Integer, rvar: OrderedVariableList(ls))
--R initiallyReduce : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))
--R initiallyReduced? : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> Boolean
--R initials : % -> List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)))
--R internalAugment : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),%) -> %
--R internalAugment : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> %
--R intersect : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),List(%)) -> List(%)
--R intersect : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),List(%)) -> List(%)
--R intersect : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),%) -> List(%)
--R intersect : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> List(%)
--R invertible? : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> Boolean
--R invertible? : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> List(Record(rnum: Integer, rvar: OrderedVariableList(ls)))
--R invertibleElseSplit? : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> Union(%, "failed")
--R invertibleSet : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> List(%)
--R last : % -> Union(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)), "failed")
--R lastSubResultant : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))) -> NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))
--R lastSubResultantElseSplit : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))) -> Union(%, "failed")
--R less? : (%,NonNegativeInteger) -> Boolean
--R mainVariable? : (OrderedVariableList(ls),%) -> Boolean
--R mainVariables : % -> List(OrderedVariableList(ls))
--R map : ((NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)) -> NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),List(%)) -> List(%)
--R map! : ((NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)) -> NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),List(%)) -> List(%)
--R member? : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> Boolean if $ has finit
--R members : % -> List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))) if $ has finit
--R more? : (%,NonNegativeInteger) -> Boolean
--R mvar : % -> OrderedVariableList(ls)
--R normalized? : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> Boolean
--R parts : % -> List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))) if $ has finit
--R purelyAlgebraic? : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> Boolean
--R purelyAlgebraicLeadingMonomial? : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> Boolean
--R purelyTranscendental? : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> Boolean
--R quasiComponent : % -> Record(close: List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))), rnum: Integer, rvar: OrderedVariableList(ls))
--R reduce : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%,((NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)))) -> NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))
--R reduce : ((NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),List(%)) -> List(%)
--R reduce : ((NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))) -> NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))
--R reduceByQuasiMonic : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))
--R reduced? : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%,((NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)))) -> Boolean
--R remainder : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> Record(rnum: Integer, rvar: OrderedVariableList(ls))

```

```

--R remove : ((NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)) -> Boolean),%) -> % if $ has finite
--R remove : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> % if $ has finiteAggregate and
--R removeDuplicates : % -> % if $ has finiteAggregate and NewSparseMultivariatePolynomial(R,OrderedVariable
--R removeZero : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> NewSparseMultivariatePol
--R retract : List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))) -> %
--R retractIfCan : List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))) -> Union(%, "failed")
--R rewriteIdealWithHeadRemainder : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),%) ->
--R rewriteIdealWithRemainder : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),%) -> List
--R rewriteSetWithReduction : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),%, ((NewSparse
--R roughBase? : % -> Boolean if R has INTDOM
--R roughEqualIdeals? : (%,%) -> Boolean if R has INTDOM
--R roughSubIdeal? : (%,%) -> Boolean if R has INTDOM
--R roughUnitIdeal? : % -> Boolean if R has INTDOM
--R select : (%,OrderedVariableList(ls)) -> Union(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),
--R select : ((NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)) -> Boolean),%) -> % if $ has finite
--R size? : (%,NonNegativeInteger) -> Boolean
--R sort : (%,OrderedVariableList(ls)) -> Record(under: %,floor: %,upper: %)
--R squareFreePart : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> List(Record(val: NewS
--R stronglyReduce : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> NewSparseMultivariate
--R stronglyReduced? : (NewSparseMultivariatePolynomial(R,OrderedVariableList(ls)),%) -> Boolean
--R triangular? : % -> Boolean if R has INTDOM
--R variables : % -> List(OrderedVariableList(ls))
--R zeroSetSplit : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),Boolean,Boolean) -> List
--R zeroSetSplit : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),Boolean) -> List(%)
--R zeroSetSplit : List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))) -> List(%)
--R zeroSetSplitIntoTriangularSystems : List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))) ->
--R
--E 1

)spool
)lisp (bye)

```

— RegularChain.help —

=====

RegularChain examples

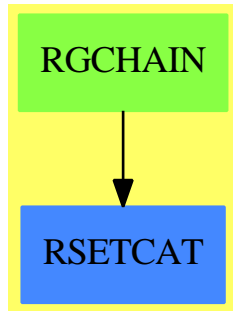
=====

A domain for regular chains (i.e. regular triangular sets) over
a Gcd-Domain and with a fix list of variables.

This is just a front-end for the RegularTriangularSet domain constructor.

See Also:

o)show RegularChain

RegularChain (RGCHAIN)**Exports:**

algebraic?	algebraicCoefficients?
algebraicVariables	any?
augment	autoReduced?
basicSet	coHeight
coerce	collect
collectQuasiMonic	collectUnder
collectUpper	construct
convert	copy
count	degree
empty	empty?
eq?	eval
every?	extend
extendIfCan	find
first	hash
headReduce	headReduced?
headRemainder	inFRittWu?
initiallyReduce	initiallyReduced?
initials	internalAugment
intersect	invertible?
invertibleElseSplit?	invertibleSet
last	lastSubResultant
lastSubResultantElseSplit	latex
less?	mainVariable?
mainVariables	map
map!	member?
members	more?
mvar	normalized?
parts	purelyAlgebraic?
purelyAlgebraicLeadingMonomial?	purelyTranscendental?
quasiComponent	reduce
reduceByQuasiMonic	reduced?
remainder	remove
removeDuplicates	removeZero
rest	retract
retractIfCan	rewriteIdealWithHeadRemainder
rewriteIdealWithRemainder	rewriteSetWithReduction
roughBase?	roughEqualIdeals?
roughSubIdeal?	roughUnitIdeal?
sample	select
size?	sort
squareFreePart	stronglyReduce
stronglyReduced?	triangular?
trivialIdeal?	variables
zeroSetSplit	zeroSetSplitIntoTriangularSystems
#?	?~=?
?=?	

— domain RGCHAIN RegularChain —

```

)abbrev domain RGCHAIN RegularChain
++ Author: Marc Moreno Maza
++ Date Created: 01/1999
++ Date Last Updated: 23/01/1999
++ Description:
++ A domain for regular chains (i.e. regular triangular sets) over
++ a Gcd-Domain and with a fix list of variables.
++ This is just a front-end for the \spadtype{RegularTriangularSet}
++ domain constructor.

RegularChain(R,ls): Exports == Implementation where
  R : GcdDomain
  ls: List Symbol
  V ==> OrderedVariableList ls
  E ==> IndexedExponents V
  P ==> NewSparseMultivariatePolynomial(R,V)
  TS ==> RegularTriangularSet(R,E,V,P)

Exports == RegularTriangularSetCategory(R,E,V,P) with
  zeroSetSplit: (List P, Boolean, Boolean) -> List $
    ++ \spad{zeroSetSplit(lp,clos?,info?)} returns a list \spad{lhs} of
    ++ regular chains such that the union of the closures of their regular
    ++ zero sets equals the affine variety associated with \spad{lp}.
    ++ Moreover, if \spad{clos?} is \spad{false} then the union of the
    ++ regular zero set of the \spad{ts} (for \spad{ts} in \spad{lhs})
    ++ equals this variety.
    ++ If \spad{info?} is \spad{true} then some information is
    ++ displayed during the computations. See
    ++ zeroSetSplit from RegularTriangularSet.

Implementation == RegularTriangularSet(R,E,V,P)

```

— RGCHAIN.dotabb —

```

"RGCHAIN" [color="#88FF44",href="bookvol10.3.pdf#nameddest=RGCHAIN"]
"RSETCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RSETCAT"]
"RGCHAIN" -> "RSETCAT"

```

Several domain constructors implement regular triangular sets (or regular chains). Among them **RegularTriangularSet** and **SquareFreeRegularTriangularSet**. They also implement an algorithm by Marc Moreno Maza for computing triangular decompositions of polynomial systems. This method is refined in the package **LazardSetSolvingPackage** in order to produce decompositions by means of Lazard triangular sets.

[illegible]

```

P := NSMP(R, V)
--R
--R
--R (5)
--R NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--R Type: Domain
--E 5

--S 6 of 34
x: P := 'x
--R
--R
--R (6) x
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--E 6

--S 7 of 34
y: P := 'y
--R
--R
--R (7) y
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--E 7

--S 8 of 34
z: P := 'z
--R
--R
--R (8) z
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--E 8

--S 9 of 34
t: P := 't
--R
--R
--R (9) t
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--E 9

--S 10 of 34
T := REGSET(R,E,V,P)
--R
--R
--R (10)
--R RegularTriangularSet(Integer,IndexedExponents(OrderedVariableList([x,y,z,t]))
--R ,OrderedVariableList([x,y,z,t]),NewSparseMultivariatePolynomial(Integer,Order
--R edVariableList([x,y,z,t])))
--R
--R Type: Domain
--E 10

```

```

--S 11 of 34
p1 := x ** 31 - x ** 6 - x - y
--R
--R
--R      31      6
--R (11)  x  - x  - x - y
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--E 11

--S 12 of 34
p2 := x ** 8 - z
--R
--R
--R      8
--R (12)  x  - z
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--E 12

--S 13 of 34
p3 := x ** 10 - t
--R
--R
--R      10
--R (13)  x  - t
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--E 13

--S 14 of 34
lp := [p1, p2, p3]
--R
--R
--R      31      6      8      10
--R (14)  [x  - x  - x - y, x  - z, x  - t]
--RType: List(NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t])))
--E 14

--S 15 of 34
zeroSetSplit(lp)$T
--R
--R
--R      5      4      2      3      8      5      3      2      4      2
--R (15)  [{z  - t , t z y + 2z y - t + 2t + t - t , (t - t)x - t y - z }]
```

--RType: List(RegularTriangularSet(Integer,IndexedExponents(...

```

--E 15

--S 16 of 34
lts := zeroSetSplit(lp,false)$T
--R
--R

```

```

--R (16)
--R      5      4      2      3      8      5      3      2      4      2
--R      [z - t , t z y + 2z y - t + 2t + t - t , (t - t)x - t y - z },
--R      3      5      2      3      2
--R      {t - 1, z - t, t z y + 2z y + 1, z x - t}, {t, z, y, x}]
--RType: List(RegularTriangularSet(Integer, IndexedExponents(...
--E 16

--S 17 of 34
[coHeight(ts) for ts in lts]
--R
--R
--R (17) [1,0,0]
--R
--RType: List(NonNegativeInteger)
--E 17

--S 18 of 34
f1 := y**2*z+2*x*y*t-2*x-z
--R
--R
--R      2
--R (18) (2t y - 2)x + z y - z
--RType: NewSparseMultivariatePolynomial(Integer, OrderedVariableList([x,y,z,t]))
--E 18

--S 19 of 34
f2:=-x**3*z+ 4*x*y**2*z+4*x**2*y*t+2*y**3*t+4*x**2-10*y**2+4*x*z-10*y*t+2
--R
--R
--R      3      2      2      3      2
--R (19) - z x + (4t y + 4)x + (4z y + 4z)x + 2t y - 10y - 10t y + 2
--RType: NewSparseMultivariatePolynomial(Integer, OrderedVariableList([x,y,z,t]))
--E 19

--S 20 of 34
f3 := 2*y*z*t+x*t**2-x-2*z
--R
--R
--R      2
--R (20) (t - 1)x + 2t z y - 2z
--RType: NewSparseMultivariatePolynomial(Integer, OrderedVariableList([x,y,z,t]))
--E 20

--S 21 of 34
f4:=-x*z**3+4*y*z**2*t+4*x*z*t**2+2*y*t**3+4*x*z+4*z**2-10*y*t- 10*t**2+2
--R
--R
--R      3      2      2      3      2      2
--R (21) (- z + (4t + 4)z)x + (4t z + 2t - 10t)y + 4z - 10t + 2
--RType: NewSparseMultivariatePolynomial(Integer, OrderedVariableList([x,y,z,t]))

```

--E 21

--S 22 of 34

lf := [f1, f2, f3, f4]

--R

--R

--R (22)

--R

--R
$$\begin{aligned} & [(2t^2y - 2)x + z^2y^2 - z^3, \\ & -z^3x^2 + (4t^2y + 4)x^2 + (4z^2y^2 + 4z)x^3 + 2t^3y^2 - 10y^2 - 10t^2y + 2, \\ & (t^3 - 1)x + 2t^2zy - 2z, \end{aligned}$$

--R

--R
$$(-z^3 + (4t^2 + 4)z)x^2 + (4t^2z^2 + 2t^3 - 10t)y^2 + 4z^2 - 10t^2 + 2]$$

--RType: List(NewSparseMultivariatePolynomial(Integer, OrderedVariableList([x,y,z,t])))

--E 22

--S 23 of 34

zeroSetSplit(lf)\$T

--R

--R

--R (23)

--R
$$\begin{aligned} & \{t^2 - 1, z^8 - 16z^6 + 256z^2 - 256, t^3y - 1, (z^3 - 8z)x - 8z^2 + 16\}, \\ & \{3t^2 + 1, z^2 - 7t^2 - 1, y + t, x + z\}, \\ & \{t^8 - 10t^6 + 10t^2 - 1, z, (t^3 - 5t)y - 5t^2 + 1, x\}, \\ & \{t^2 + 3, z^2 - 4, y + t, x - z\} \end{aligned}$$

--RType: List(RegularTriangularSet(Integer, IndexedExponents(...

--E 23

--S 24 of 34

lts2 := zeroSetSplit(lf, false)\$T

--R

--R

--R (24)

--R
$$\begin{aligned} & \{t^8 - 10t^6 + 10t^2 - 1, z, (t^3 - 5t)y - 5t^2 + 1, x\}, \\ & \{t^2 - 1, z^8 - 16z^6 + 256z^2 - 256, t^3y - 1, (z^3 - 8z)x - 8z^2 + 16\}, \\ & \{3t^2 + 1, z^2 - 7t^2 - 1, y + t, x + z\}, \{t^2 + 3, z^2 - 4, y + t, x - z\} \end{aligned}$$

--RType: List(RegularTriangularSet(Integer, IndexedExponents(...

--E 24

--S 25 of 34

[coHeight(ts) for ts in lts2]


```

--R
--R
--R (25) [0,0,0,0]
--R
--R                                         Type: List(NonNegativeInteger)
--E 25

--S 26 of 34
degrees := [degree(ts) for ts in lts2]
--R
--R
--R (26) [8,16,4,4]
--R
--R                                         Type: List(NonNegativeInteger)
--E 26

--S 27 of 34
reduce(+,degrees)
--R
--R
--R (27) 32
--R
--R                                         Type: PositiveInteger
--E 27

--S 28 of 34
u : R := 2
--R
--R
--R (28) 2
--R
--R                                         Type: Integer
--E 28

--S 29 of 34
q1 := 2*(u-1)**2+ 2*(x-z*x+z**2)+ y**2*(x-1)**2- 2*u*x+ 2*y*t*(1-x)*(x-z)+_
      2*u*z*t*(t-y)+ u**2*t**2*(1-2*z)+ 2*u*t**2*(z-x)+ 2*u*t*y*(z-1)+_
      2*u*z*x*(y+1)+ (u**2-2*u)*z**2*t**2+ 2*u**2*z**2+ 4*u*(1-u)*z+_
      t**2*(z-x)**2
--R
--R
--R (29)
--R      2      2      2      2      2      2      2      2
--R      (y  - 2t y + t )x  + (- 2y  + ((2t + 4)z + 2t)y + (- 2t  + 2)z - 4t  - 2)x
--R      +
--R      2      2      2      2
--R      y  + (- 2t z - 4t)y + (t  + 10)z  - 8z + 4t  + 2
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--E 29

--S 30 of 34
q2 := t*(2*z+1)*(x-z)+ y*(z+2)*(1-x)+ u*(u-2)*t+ u*(1-2*u)*z*t+_
      u*y*(x+u-z*x-1)+ u*(u+1)*z**2*t
--R

```

```

--R
--R
--R      2
--R      (30)  (- 3z y + 2t z + t)x + (z + 4)y + 4t z - 7t z
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--E 30

--S 31 of 34
q3 := -u**2*(z-1)**2+ 2*z*(z-x)-2*(x-1)
--R
--R
--R      2
--R      (31)  (- 2z - 2)x - 2z + 8z - 2
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--E 31

--S 32 of 34
q4 := u**2+4*(z-x**2)+3*y**2*(x-1)**2- 3*t**2*(z-x)**2+_
      3*u**2*t**2*(z-1)**2+u**2*z*(z-2)+6*u*t*y*(z+x+z*x-1)
--R
--R
--R      (32)
--R      2      2      2      2      2      2
--R      (3y - 3t - 4)x + (- 6y + (12t z + 12t)y + 6t z)x + 3y + (12t z - 12t)y
--R      +
--R      2      2      2      2
--R      (9t + 4)z + (- 24t - 4)z + 12t + 4
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--E 32

--S 33 of 34
lq := [q1, q2, q3, q4]
--R
--R
--R      (33)
--R      [
--R      2      2      2
--R      (y - 2t y + t )x
--R      +
--R      2      2      2      2      2
--R      (- 2y + ((2t + 4)z + 2t)y + (- 2t + 2)z - 4t - 2)x + y
--R      +
--R      2      2      2
--R      (- 2t z - 4t)y + (t + 10)z - 8z + 4t + 2
--R      ,
--R      2      2
--R      (- 3z y + 2t z + t)x + (z + 4)y + 4t z - 7t z, (- 2z - 2)x - 2z + 8z - 2,
--R
--R      2      2      2      2      2      2
--R      (3y - 3t - 4)x + (- 6y + (12t z + 12t)y + 6t z)x + 3y
--R      +

```

```

--R      2      2      2      2
--R      (12t z - 12t)y + (9t + 4)z + (- 24t - 4)z + 12t + 4
--R      ]
--RType: List(NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t])))
--E 33

--S 34 of 34
zeroSetSplit(lq,true,true)$T
--R
--R [1 <4,0> -> |4|; {0}]W[2 <5,0>,<3,1> -> |8|; ...
--R *** QCMACK Statistics ***
--R      Table      size: 36
--R      Entries reused: 255
--R
--R *** REGSETGCD: Gcd Statistics ***
--R      Table      size: 125
--R      Entries reused: 0
--R
--R *** REGSETGCD: Inv Set Statistics ***
--R      Table      size: 30
--R      Entries reused: 0
--R
--R (34)
--R [
--R {
--R      24      23      22
--R      960725655771966t + 386820897948702t + 8906817198608181t
--R      +
--R      21      20      19
--R      2704966893949428t + 37304033340228264t + 7924782817170207t
--R      +
--R      18      17      16
--R      93126799040354990t + 13101273653130910t + 156146250424711858t
--R      +
--R      15      14      13
--R      16626490957259119t + 190699288479805763t + 24339173367625275t
--R      +
--R      12      11      10
--R      180532313014960135t + 35288089030975378t + 135054975747656285t
--R      +
--R      9      8      7
--R      34733736952488540t + 75947600354493972t + 19772555692457088t
--R      +
--R      6      5      4
--R      28871558573755428t + 5576152439081664t + 6321711820352976t
--R      +
--R      3      2
--R      438314209312320t + 581105748367008t - 60254467992576t + 1449115951104
--R      ,
--R

```

```
--R 23
--R 26604210869491302385515265737052082361668474181372891857784t
--R +
--R 22
--R 443104378424686086067294899528296664238693556855017735265295t
--R +
--R 21
--R 279078393286701234679141342358988327155321305829547090310242t
--R +
--R 20
--R 3390276361413232465107617176615543054620626391823613392185226t
--R +
--R 19
--R 941478179503540575554198645220352803719793196473813837434129t
--R +
--R 18
--R 11547855194679475242211696749673949352585747674184320988144390t
--R +
--R 17
--R 1343609566765597789881701656699413216467215660333356417241432t
--R +
--R 16
--R 23233813868147873503933551617175640859899102987800663566699334t
--R +
--R 15
--R 869574020537672336950845440508790740850931336484983573386433t
--R +
--R 14
--R 31561554305876934875419461486969926554241750065103460820476969t
--R +
--R 13
--R 1271400990287717487442065952547731879554823889855386072264931t
--R +
--R 12
--R 31945089913863736044802526964079540198337049550503295825160523t
--R +
--R 11
--R 3738735704288144509871371560232845884439102270778010470931960t
--R +
--R 10
--R 25293997512391412026144601435771131587561905532992045692885927t
--R +
--R 9
--R 5210239009846067123469262799870052773410471135950175008046524t
--R +
--R 8
--R 15083887986930297166259870568608270427403187606238713491129188t
--R +
--R 7
--R 3522087234692930126383686270775779553481769125670839075109000t
```

```

--R      +
--R
--R      6
--R      6079945200395681013086533792568886491101244247440034969288588t
--R      +
--R      5
--R      1090634852433900888199913756247986023196987723469934933603680t
--R      +
--R      4
--R      1405819430871907102294432537538335402102838994019667487458352t
--R      +
--R      3
--R      88071527950320450072536671265507748878347828884933605202432t
--R      +
--R      2
--R      135882489433640933229781177155977768016065765482378657129440t
--R      +
--R      - 13957283442882262230559894607400314082516690749975646520320t
--R      +
--R      334637692973189299277258325709308472592117112855749713920
--R      *
--R      z
--R      +
--R      23
--R      8567175484043952879756725964506833932149637101090521164936t
--R      +
--R      22
--R      149792392864201791845708374032728942498797519251667250945721t
--R      +
--R      21
--R      77258371783645822157410861582159764138123003074190374021550t
--R      +
--R      20
--R      1108862254126854214498918940708612211184560556764334742191654t
--R      +
--R      19
--R      213250494460678865219774480106826053783815789621501732672327t
--R      +
--R      18
--R      3668929075160666195729177894178343514501987898410131431699882t
--R      +
--R      17
--R      171388906471001872879490124368748236314765459039567820048872t
--R      +
--R      16
--R      7192430746914602166660233477331022483144921771645523139658986t
--R      +
--R      15
--R      - 128798674689690072812879965633090291959663143108437362453385t
--R      +
--R      14

```

```

--R      9553010858341425909306423132921134040856028790803526430270671t
--R      +
--R      13
--R      - 13296096245675492874538687646300437824658458709144441096603t
--R      +
--R      12
--R      9475806805814145326383085518325333106881690568644274964864413t
--R      +
--R      11
--R      803234687925133458861659855664084927606298794799856265539336t
--R      +
--R      10
--R      7338202759292865165994622349207516400662174302614595173333825t
--R      +
--R      9
--R      1308004628480367351164369613111971668880538855640917200187108t
--R      +
--R      8
--R      4268059455741255498880229598973705747098216067697754352634748t
--R      +
--R      7
--R      892893526858514095791318775904093300103045601514470613580600t
--R      +
--R      6
--R      1679152575460683956631925852181341501981598137465328797013652t
--R      +
--R      5
--R      269757415767922980378967154143357835544113158280591408043936t
--R      +
--R      4
--R      380951527864657529033580829801282724081345372680202920198224t
--R      +
--R      3
--R      19785545294228495032998826937601341132725035339452913286656t
--R      +
--R      2
--R      36477412057384782942366635303396637763303928174935079178528t
--R      +
--R      - 3722212879279038648713080422224976273210890229485838670848t
--R      +
--R      89079724853114348361230634484013862024728599906874105856
--R      ,
--R      3      2      3      2
--R      (3z - 11z + 8z + 4)y + 2t z + 4t z - 5t z - t,
--R      2
--R      (z + 1)x + z - 4z + 1}
--R      ]
--R      --IType: List(RegularTriangularSet(Integer,IndexedExponents(...
--E 34
)spool

```

```
)lisp (bye)
```

— RegularTriangularSet.help —

```
=====
RegularTriangularSet examples
=====
```

The RegularTriangularSet domain constructor implements regular triangular sets. These particular triangular sets were introduced by M. Kalkbrener (1991) in his PhD Thesis under the name regular chains. Regular chains and their related concepts are presented in the paper "On the Theories of Triangular sets" By P. Aubry, D. Lazard and M. Moreno Maza (to appear in the Journal of Symbolic Computation). The RegularTriangularSet constructor also provides a new method (by the third author) for solving polynomial system by means of regular chains. This method has two ways of solving. One has the same specifications as Kalkbrener's algorithm (1991) and the other is closer to Lazard's method (Discr. App. Math, 1991). Moreover, this new method removes redundant component from the decompositions when this is not too expensive. This is always the case with square-free regular chains. So if you want to obtain decompositions without redundant components just use the SquareFreeRegularTriangularSet domain constructor or the LazardSetSolvingPackage package constructor. See also the LexTriangularPackage and ZeroDimensionalSolvePackage for the case of algebraic systems with a finite number of (complex) solutions.

One of the main features of regular triangular sets is that they naturally define towers of simple extensions of a field. This allows to perform with multivariate polynomials the same kind of operations as one can do in an EuclideanDomain.

The RegularTriangularSet constructor takes four arguments. The first one, R, is the coefficient ring of the polynomials; it must belong to the category GcdDomain. The second one, E, is the exponent monoid of the polynomials; it must belong to the category OrderedAbelianMonoidSup. The third one, V, is the ordered set of variables; it must belong to the category OrderedSet. The last one is the polynomial ring; it must belong to the category RecursivePolynomialCategory(R,E,V). The abbreviation for RegularTriangularSet is REGSET. See also the constructor RegularChain which only takes two arguments, the coefficient ring and the ordered set of variables; in that case, polynomials are necessarily built with the NewSparseMultivariatePolynomial domain constructor.

We shall explain now how to use the constructor REGSET and how to read the decomposition of a polynomial system by means of regular sets.

Let us give some examples. We start with an easy one (Donati-Traverso) in order to understand the two ways of solving polynomial systems provided by the REGSET constructor.

Define the coefficient ring.

```
R := Integer
Integer
Type: Domain
```

Define the list of variables,

```
ls : List Symbol := [x,y,z,t]
[x,y,z,t]
Type: List Symbol
```

and make it an ordered set;

```
V := OVAR(ls)
OrderedVariableList [x,y,z,t]
Type: Domain
```

then define the exponent monoid.

```
E := IndexedExponents V
IndexedExponents OrderedVariableList [x,y,z,t]
Type: Domain
```

Define the polynomial ring.

```
P := NSMP(R, V)
NewSparseMultivariatePolynomial(Integer,OrderedVariableList [x,y,z,t])
Type: Domain
```

Let the variables be polynomial.

```
x: P := 'x
x
Type: NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t])
y: P := 'y
y
Type: NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t])
z: P := 'z
z
Type: NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t])
```



```

t: P := 't
t
Type: NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t])

```

Now call the RegularTriangularSet domain constructor.

```

T := REGSET(R,E,V,P)
RegularTriangularSet(Integer,IndexedExponents OrderedVariableList [x,y,z,t],O
rderedVariableList [x,y,z,t],NewSparseMultivariatePolynomial(Integer,OrderedV
ariableList [x,y,z,t]))
Type: Domain

```

Define a polynomial system.

```

p1 := x ** 31 - x ** 6 - x - y
31 6
x - x - x - y
Type: NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t])

p2 := x ** 8 - z
8
x - z
Type: NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t])

p3 := x ** 10 - t
10
x - t
Type: NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t])

lp := [p1, p2, p3]
31 6 8 10
[x - x - x - y, x - z, x - t]
Type: List NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t])

```

First of all, let us solve this system in the sense of Kalkbrener.

```

zeroSetSplit(lp)$T
5 4 2 3 8 5 3 2 4 2
[{z - t , t z y + 2z y - t + 2t + t - t , (t - t)x - t y - z }]
Type: List RegularTriangularSet(Integer,
IndexedExponents OrderedVariableList [x,y,z,t],
OrderedVariableList [x,y,z,t],
NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t]))

```

And now in the sense of Lazard (or Wu and other authors).

```
lts := zeroSetSplit(lp,false)$T
      5      4      2      3      8      5      3      2      4      2
      [{z  - t ,t z y  + 2z y - t  + 2t  + t  - t ,(t  - t)x - t y - z },
      3      5      2      3      2
      {t  - 1,z  - t,t z y  + 2z y + 1,z x  - t}, {t,z,y,x}]
      Type: List RegularTriangularSet(Integer,
      IndexedExponents OrderedVariableList [x,y,z,t],
      OrderedVariableList [x,y,z,t],
      NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t]))
```

We can see that the first decomposition is a subset of the second.
So how can both be correct ?

Recall first that polynomials from a domain of the category
RecursivePolynomialCategory are regarded as univariate polynomials in
their main variable. For instance the second polynomial in the first
set of each decomposition has main variable y and its initial
(i.e. its leading coefficient w.r.t. its main variable) is t z.

Now let us explain how to read the second decomposition. Note that
the non-constant initials of the first set are t^4-t and $t z$. Then
the solutions described by this first set are the common zeros of
its polynomials that do not cancel the polynomials t^4-t and $ty z$.
Now the solutions of the input system lp satisfying these equations
are described by the second and the third sets of the decomposition.
Thus, in some sense, they can be considered as degenerated solutions.
The solutions given by the first set are called the generic points of
the system; they give the general form of the solutions. The first
decomposition only provides these generic points. This latter
decomposition is useful when there are many degenerated solutions
(which is sometimes hard to compute) and when one is only interested
in general informations, like the dimension of the input system.

We can get the dimensions of each component of a decomposition as follows.

```
[coHeight(ts) for ts in lts]
[1,0,0]
      Type: List NonNegativeInteger
```

Thus the first set has dimension one. Indeed t can take any value,
except 0 or any third root of 1, whereas z is completely determined
from t, y is given by z and t, and finally x is given by the other
three variables. In the second and the third sets of the second
decomposition the four variables are completely determined and thus
these sets have dimension zero.

We give now the precise specifications of each decomposition. This

assume some mathematical knowledge. However, for the non-expert user, the above explanations will be sufficient to understand the other features of the RSEGSET constructor.

The input system lp is decomposed in the sense of Kalkbrener as finitely many regular sets T_1, \dots, T_s such that the radical ideal generated by lp is the intersection of the radicals of the saturated ideals of T_1, \dots, T_s . In other words, the affine variety associated with lp is the union of the closures (w.r.t. Zarisky topology) of the regular-zeros sets of T_1, \dots, T_s .

N. B. The prime ideals associated with the radical of the saturated ideal of a regular triangular set have all the same dimension; moreover these prime ideals can be given by characteristic sets with the same main variables. Thus a decomposition in the sense of Kalkbrener is unmixed dimensional. Then it can be viewed as a lazy decomposition into prime ideals (some of these prime ideals being merged into unmixed dimensional ideals).

Now we explain the other way of solving by means of regular triangular sets. The input system lp is decomposed in the sense of Lazard as finitely many regular triangular sets T_1, \dots, T_s such that the affine variety associated with lp is the union of the regular-zeros sets of T_1, \dots, T_s . Thus a decomposition in the sense of Lazard is also a decomposition in the sense of Kalkbrener; the converse is false as we have seen before.

When the input system has a finite number of solutions, both ways of solving provide similar decompositions as we shall see with this second example (Caprasse).

Define a polynomial system.

```
f1 := y**2*z+2*x*y*t-2*x-z
      2
(2t y - 2)x + z y - z
      Type: NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t])

f2:=-x**3*z+ 4*x*y**2*z+4*x**2*y*t+2*y**3*t+4*x**2-10*y**2+4*x*z-10*y*t+2
      3      2      2      3      2
- z x + (4t y + 4)x + (4z y + 4z)x + 2t y - 10y - 10t y + 2
      Type: NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t])

f3 := 2*y*z*t+x*t**2-x-2*z
      2
(t - 1)x + 2t z y - 2z
      Type: NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t])
```

```
f4:=-x*z**3+4*y*z**2*t+4*x*z*t**2+2*y*t**3+4*x*z+4*z**2-10*y*t- 10*t**2+2
      3      2      2      3      2      2
      (- z  + (4t  + 4)z)x + (4t z  + 2t  - 10t)y + 4z  - 10t  + 2
      Type: NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t])
```

```
lf := [f1, f2, f3, f4]
      2
      [(2t y - 2)x + z y  - z,
      3      2      2      3      2
      - z x  + (4t y + 4)x  + (4z y  + 4z)x + 2t y  - 10y  - 10t y + 2,
      2
      (t  - 1)x + 2t z y - 2z,
      3      2      2      3      2      2
      (- z  + (4t  + 4)z)x + (4t z  + 2t  - 10t)y + 4z  - 10t  + 2]
      Type: List NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t])
```

First of all, let us solve this system in the sense of Kalkbrener.

```
zeroSetSplit(lf)$T
      2      8      6      2      3      2
      [{t  - 1, z  - 16z  + 256z  - 256, t y - 1, (z  - 8z)x - 8z  + 16},
      2      2      2
      {3t  + 1, z  - 7t  - 1, y + t, x + z},
      8      6      2      3      2
      {t  - 10t  + 10t  - 1, z, (t  - 5t)y - 5t  + 1, x},
      2      2
      {t  + 3, z  - 4, y + t, x - z}]
      Type: List RegularTriangularSet(Integer,
      IndexedExponents OrderedVariableList [x,y,z,t],
      OrderedVariableList [x,y,z,t],
      NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t]))
```

And now in the sense of Lazard (or Wu and other authors).

```
lts2 := zeroSetSplit(lf,false)$T
      8      6      2      3      2
      [{t  - 10t  + 10t  - 1, z, (t  - 5t)y - 5t  + 1, x},
      2      8      6      2      3      2
      {t  - 1, z  - 16z  + 256z  - 256, t y - 1, (z  - 8z)x - 8z  + 16},
      2      2      2      2      2
      {3t  + 1, z  - 7t  - 1, y + t, x + z}, {t  + 3, z  - 4, y + t, x - z}]
      Type: List RegularTriangularSet(Integer,
      IndexedExponents OrderedVariableList [x,y,z,t],
      OrderedVariableList [x,y,z,t],
      NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t]))
```

Up to the ordering of the components, both decompositions are identical.

Let us check that each component has a finite number of solutions.

```
[coHeight(ts) for ts in lts2]
[0,0,0,0]
Type: List NonNegativeInteger
```

Let us count the degrees of each component,

```
degrees := [degree(ts) for ts in lts2]
[8,16,4,4]
Type: List NonNegativeInteger
```

and compute their sum.

```
reduce(+,degrees)
32
Type: PositiveInteger
```

We study now the options of the `zeroSetSplit` operation. As we have seen yet, there is an optional second argument which is a boolean value. If this value is true (this is the default) then the decomposition is computed in the sense of Kalkbrener, otherwise it is computed in the sense of Lazard.

There is a second boolean optional argument that can be used (in that case the first optional argument must be present). This second option allows you to get some information during the computations.

Therefore, we need to understand a little what is going on during the computations. An important feature of the algorithm is that the intermediate computations are managed in some sense like the processes of a Unix system. Indeed, each intermediate computation may generate other intermediate computations and the management of all these computations is a crucial task for the efficiency. Thus any intermediate computation may be suspended, killed or resumed, depending on algebraic considerations that determine priorities for these processes. The goal is of course to go as fast as possible towards the final decomposition which means to avoid as much as possible unnecessary computations.

To follow the computations, one needs to set to true the second argument. Then a lot of numbers and letters are displayed. Between a [and a] one has the state of the processes at a given time. Just after [one can see the number of processes. Then each process is represented by two numbers between < and >. A process consists of a list of polynomial `ps` and a triangular set `ts`; its goal is to compute the common zeros of `ps` that belong to the regular-zeros set of `ts`. After the processes, the number between pipes gives the total number

of polynomials in all the sets `ps`. Finally, the number between braces gives the number of components of a decomposition that are already computed. This number may decrease.

Let us take a third example (Czapor-Geddes-Wang) to see how this information is displayed.

Define a polynomial system.

```

u : R := 2
  2
                                     Type: Integer

q1 := 2*(u-1)**2+ 2*(x-z*x+z**2)+ y**2*(x-1)**2- 2*u*x+ 2*y*t*(1-x)*(x-z)+_
      2*u*z*t*(t-y)+ u**2*t**2*(1-2*z)+ 2*u*t**2*(z-x)+ 2*u*t*y*(z-1)+_
      2*u*z*x*(y+1)+ (u**2-2*u)*z**2*t**2+ 2*u**2*z**2+ 4*u*(1-u)*z+_
      t**2*(z-x)**2}
      2      2      2      2      2      2
      (y  - 2t y + t )x  + (- 2y  + ((2t + 4)z + 2t)y + (- 2t  + 2)z - 4t  - 2)x
+
      2      2      2      2
      y  + (- 2t z - 4t)y + (t  + 10)z  - 8z + 4t  + 2
      Type: NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t])

q2 := t*(2*z+1)*(x-z)+ y*(z+2)*(1-x)+ u*(u-2)*t+ u*(1-2*u)*z*t+_
      u*y*(x+u-z*x-1)+ u*(u+1)*z**2*t}
      (- 3z y + 2t z + t)x + (z + 4)y + 4t z  - 7t z
      Type: NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t])

q3 := -u**2*(z-1)**2+ 2*z*(z-x)-2*(x-1)
      (- 2z - 2)x - 2z  + 8z - 2
      Type: NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t])

q4 := u**2+4*(z-x**2)+3*y**2*(x-1)**2- 3*t**2*(z-x)**2+_
      3*u**2*t**2*(z-1)**2+u**2*z*(z-2)+6*u*t*y*(z+x+z*x-1)}
      2      2      2      2      2      2
      (3y  - 3t  - 4)x  + (- 6y  + (12t z + 12t)y + 6t z)x + 3y  + (12t z - 12t)y
+
      2      2      2      2
      (9t  + 4)z  + (- 24t  - 4)z + 12t  + 4
      Type: NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t])

lq := [q1, q2, q3, q4]
[
      2      2      2
      (y  - 2t y + t )x

```

```

+
      2      2      2      2
      (- 2y  + ((2t + 4)z + 2t)y + (- 2t  + 2)z - 4t  - 2)x + y
+
      2      2      2
      (- 2t z - 4t)y + (t  + 10)z  - 8z + 4t  + 2
,
      2      2
      (- 3z y + 2t z + t)x + (z + 4)y + 4t z  - 7t z, (- 2z - 2)x - 2z  + 8z - 2,

      2      2      2      2      2      2
      (3y  - 3t  - 4)x  + (- 6y  + (12t z + 12t)y + 6t z)x + 3y
+
      2      2      2      2
      (12t z - 12t)y + (9t  + 4)z  + (- 24t  - 4)z + 12t  + 4
]
Type: List NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t])

```

Let us try the information option. N.B. The timing should be between 1 and 10 minutes, depending on your machine.

```

zeroSetSplit(lq,true,true)$T
[1 <4,0> -> |4|; {0}]W[2 <5,0>,<3,1> -> |8|; {0}]
[2 <4,1>,<3,1> -> |7|; {0}]
[1 <3,1> -> |3|; {0}]G
[2 <4,1>,<4,1> -> |8|; {0}]W
[3 <5,1>,<4,1>,<3,2> -> |12|; {0}]GI
[3 <4,2>,<4,1>,<3,2> -> |11|; {0}]GWw
[3 <4,1>,<3,2>,<5,2> -> |12|; {0}]
[3 <3,2>,<3,2>,<5,2> -> |11|; {0}]GIwWWWw
[4 <3,2>,<4,2>,<5,2>,<2,3> -> |14|; {0}]
[4 <2,2>,<4,2>,<5,2>,<2,3> -> |13|; {0}]Gwww
[5 <3,2>,<3,2>,<4,2>,<5,2>,<2,3> -> |17|; {0}]Gwwwww
[8 <3,2>,<4,2>,<4,2>,<4,2>,<4,2>,<4,2>,<5,2>,<2,3> -> |30|; {0}]Gwwwww
[8 <4,2>,<4,2>,<4,2>,<4,2>,<4,2>,<4,2>,<5,2>,<2,3> -> |31|; {0}]
[8 <3,3>,<4,2>,<4,2>,<4,2>,<4,2>,<4,2>,<5,2>,<2,3> -> |30|; {0}]
[8 <2,3>,<4,2>,<4,2>,<4,2>,<4,2>,<4,2>,<5,2>,<2,3> -> |29|; {0}]
[8 <1,3>,<4,2>,<4,2>,<4,2>,<4,2>,<4,2>,<5,2>,<2,3> -> |28|; {0}]
[7 <4,2>,<4,2>,<4,2>,<4,2>,<4,2>,<5,2>,<2,3> -> |27|; {0}]
[6 <4,2>,<4,2>,<4,2>,<4,2>,<5,2>,<2,3> -> |23|; {0}]
[5 <4,2>,<4,2>,<4,2>,<5,2>,<2,3> -> |19|; {0}]GIGIWwww
[6 <5,2>,<4,2>,<4,2>,<5,2>,<3,3>,<2,3> -> |23|; {0}]
[6 <4,3>,<4,2>,<4,2>,<5,2>,<3,3>,<2,3> -> |22|; {0}]GIGI
[6 <3,4>,<4,2>,<4,2>,<5,2>,<3,3>,<2,3> -> |21|; {0}]
[6 <2,4>,<4,2>,<4,2>,<5,2>,<3,3>,<2,3> -> |20|; {0}]GGG
[5 <4,2>,<4,2>,<5,2>,<3,3>,<2,3> -> |18|; {0}]GIGIWwww
[6 <5,2>,<4,2>,<5,2>,<3,3>,<3,3>,<2,3> -> |22|; {0}]
[6 <4,3>,<4,2>,<5,2>,<3,3>,<3,3>,<2,3> -> |21|; {0}]
GIwwWWWWWWWWWWWWWWWWWW

```

[illegible]

[illegible]

```

[9 <2,4>,<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |26|; {1}]
[9 <1,4>,<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |25|; {1}]
[8 <3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |24|; {1}]W
[8 <2,4>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |23|; {1}]
[8 <1,4>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |22|; {1}]
[7 <4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |21|; {1}]w
[7 <3,4>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |20|; {1}]
[7 <2,4>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |19|; {1}]
[7 <1,4>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |18|; {1}]
[6 <2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |17|; {1}]GGwwwww
[7 <3,3>,<3,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |21|; {1}]GIW
[7 <2,4>,<3,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |20|; {1}]GG
[6 <3,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |18|; {1}]Gwwwww
[7 <4,3>,<4,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |23|; {1}]GIW
[7 <3,4>,<4,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |22|; {1}]
[6 <4,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |19|; {1}]GIW
[6 <3,4>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |18|; {1}]GGW
[6 <2,4>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |17|; {1}]
[6 <1,4>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |16|; {1}]GGG
[5 <3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |15|; {1}]GIW
[5 <2,4>,<3,3>,<3,3>,<3,4>,<3,4> -> |14|; {1}]GG
[4 <3,3>,<3,3>,<3,4>,<3,4> -> |12|; {1}]
[3 <3,3>,<3,4>,<3,4> -> |9|; {1}]W
[3 <2,4>,<3,4>,<3,4> -> |8|; {1}]
[3 <1,4>,<3,4>,<3,4> -> |7|; {1}]G
[2 <3,4>,<3,4> -> |6|; {1}]G
[1 <3,4> -> |3|; {1}]
[1 <2,4> -> |2|; {1}]
[1 <1,4> -> |1|; {1}]

```

*** QCMACK Statistics ***

```

Table      size: 36
Entries reused: 255

```

*** REGSETGCD: Gcd Statistics ***

```

Table      size: 125
Entries reused: 0

```

*** REGSETGCD: Inv Set Statistics ***

```

Table      size: 30
Entries reused: 0

```

```

[
{
          24          23          22
960725655771966t + 386820897948702t + 8906817198608181t
+
          21          20          19
2704966893949428t + 37304033340228264t + 7924782817170207t
+
          18          17          16

```

93126799040354990t + 13101273653130910t + 156146250424711858t
+
15 14 13
16626490957259119t + 190699288479805763t + 24339173367625275t
+
12 11 10
180532313014960135t + 35288089030975378t + 135054975747656285t
+
9 8 7
34733736952488540t + 75947600354493972t + 19772555692457088t
+
6 5 4
28871558573755428t + 5576152439081664t + 6321711820352976t
+
3 2
438314209312320t + 581105748367008t - 60254467992576t + 1449115951104
,
23
26604210869491302385515265737052082361668474181372891857784t
+
22
443104378424686086067294899528296664238693556855017735265295t
+
21
279078393286701234679141342358988327155321305829547090310242t
+
20
3390276361413232465107617176615543054620626391823613392185226t
+
19
941478179503540575554198645220352803719793196473813837434129t
+
18
11547855194679475242211696749673949352585747674184320988144390t
+
17
1343609566765597789881701656699413216467215660333356417241432t
+
16
23233813868147873503933551617175640859899102987800663566699334t
+
15
869574020537672336950845440508790740850931336484983573386433t
+
14
31561554305876934875419461486969926554241750065103460820476969t
+
13
1271400990287717487442065952547731879554823889855386072264931t

```

+
12
31945089913863736044802526964079540198337049550503295825160523t
+
11
3738735704288144509871371560232845884439102270778010470931960t
+
10
25293997512391412026144601435771131587561905532992045692885927t
+
9
5210239009846067123469262799870052773410471135950175008046524t
+
8
15083887986930297166259870568608270427403187606238713491129188t
+
7
3522087234692930126383686270775779553481769125670839075109000t
+
6
6079945200395681013086533792568886491101244247440034969288588t
+
5
1090634852433900888199913756247986023196987723469934933603680t
+
4
1405819430871907102294432537538335402102838994019667487458352t
+
3
88071527950320450072536671265507748878347828884933605202432t
+
2
135882489433640933229781177155977768016065765482378657129440t
+
- 13957283442882262230559894607400314082516690749975646520320t
+
334637692973189299277258325709308472592117112855749713920
*
Z
+
23
8567175484043952879756725964506833932149637101090521164936t
+
22
149792392864201791845708374032728942498797519251667250945721t
+
21
77258371783645822157410861582159764138123003074190374021550t
+
20

```

1108862254126854214498918940708612211184560556764334742191654t
 +
 19
 213250494460678865219774480106826053783815789621501732672327t
 +
 18
 3668929075160666195729177894178343514501987898410131431699882t
 +
 17
 171388906471001872879490124368748236314765459039567820048872t
 +
 16
 7192430746914602166660233477331022483144921771645523139658986t
 +
 15
 - 128798674689690072812879965633090291959663143108437362453385t
 +
 14
 9553010858341425909306423132921134040856028790803526430270671t
 +
 13
 - 13296096245675492874538687646300437824658458709144441096603t
 +
 12
 9475806805814145326383085518325333106881690568644274964864413t
 +
 11
 803234687925133458861659855664084927606298794799856265539336t
 +
 10
 7338202759292865165994622349207516400662174302614595173333825t
 +
 9
 1308004628480367351164369613111971668880538855640917200187108t
 +
 8
 4268059455741255498880229598973705747098216067697754352634748t
 +
 7
 892893526858514095791318775904093300103045601514470613580600t
 +
 6
 1679152575460683956631925852181341501981598137465328797013652t
 +
 5
 269757415767922980378967154143357835544113158280591408043936t
 +
 4
 380951527864657529033580829801282724081345372680202920198224t
 +

```

3
19785545294228495032998826937601341132725035339452913286656t
+
2
36477412057384782942366635303396637763303928174935079178528t
+
- 3722212879279038648713080422224976273210890229485838670848t
+
89079724853114348361230634484013862024728599906874105856
,
3      2      3      2
(3z  - 11z  + 8z + 4)y + 2t z  + 4t z  - 5t z - t,
2
(z + 1)x + z  - 4z + 1}
]
Type: List RegularTriangularSet(Integer,
    IndexedExponents OrderedVariableList [x,y,z,t],
    OrderedVariableList [x,y,z,t],
    NewSparseMultivariatePolynomial(Integer,
        OrderedVariableList [x,y,z,t]))

```

Between a sequence of processes, thus between a] and a [you can see capital letters W, G, I and lower case letters i, w. Each time a capital letter appears a non-trivial computation has been performed and its result is put in a hash-table. Each time a lower case letter appears a needed result has been found in an hash-table. The use of these hash-tables generally speed up the computations. However, on very large systems, it may happen that these hash-tables become too big to be handle by your AXIOM configuration. Then in these exceptional cases, you may prefer getting a result (even if it takes a long time) than getting nothing. Hence you need to know how to prevent the RSEGET constructor from using these hash-tables. In that case you will be using the zeroSetSplit with five arguments. The first one is the input system lp as above. The second one is a boolean value hash? which is true iff you want to use hash-tables. The third one is boolean value clos? which is true iff you want to solve your system in the sense of Kalkbrener, the other way remaining that of Lazard. The fourth argument is boolean value info? which is true iff you want to display information during the computations. The last one is boolean value prep? which is true iff you want to use some heuristics that are performed on the input system before starting the real algorithm. The value of this flag is true when you are using zeroSetSplit with less than five arguments. Note that there is no available signature for zeroSetSplit with four arguments.

We finish this section by some remarks about both ways of solving, in the sense of Kalkbrener or in the sense of Lazard. For problems with a finite number of solutions, there are theoretically equivalent and the resulting decompositions are identical, up to the ordering of the components. However, when solving in the sense of Lazard, the

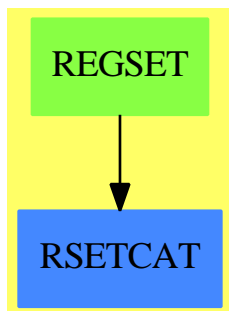
algorithm behaves differently. In that case, it becomes more incremental than in the sense of Kalkbrener. That means the polynomials of the input system are considered one after another whereas in the sense of Kalkbrener the input system is treated more globally.

This makes an important difference in positive dimension. Indeed when solving in the sense of Kalkbrener, the Primeidealkettensatz of Krull is used. That means any regular triangular containing more polynomials than the input system can be deleted. This is not possible when solving in the sense of Lazard. This explains why Kalkbrener's decompositions usually contain less components than those of Lazard. However, it may happen with some examples that the incremental process (that cannot be used when solving in the sense of Kalkbrener) provide a more efficient way of solving than the global one even if the Primeidealkettensatz is used. Thus just try both, with the various options, before concluding that you cannot solve your favorite system with `zeroSetSplit`. There exist more options at the development level that are not currently available in this public version.

See Also:

- o `)help GcdDomain`
- o `)help OrderedAbelianMonoidSup`
- o `)help OrderedSet`
- o `)help RecursivePolynomialCategory`
- o `)help RegularChain`
- o `)help NewSparseMultivariatePolynomial`
- o `)help ZeroDimensionalSolvePackage`
- o `)help LexTriangularPackage`
- o `)help LazardSetSolvingPackage`
- o `)help SquareFreeRegularTriangularSet`
- o `)show RegularTriangularSet`

RegularTriangularSet (REGSET)



Exports:

algebraic?	algebraicCoefficients?
algebraicVariables	any?
augment	autoReduced?
basicSet	coerce
coHeight	collect
collectQuasiMonic	collectUnder
collectUpper	construct
convert	copy
count	degree
empty	empty?
eq?	eval
every?	extend
extendIfCan	find
first	hash
headRemainder	headReduce
headReduced?	inFRittWu?
initiallyReduce	initiallyReduced?
initials	internalAugment
internalZeroSetSplit	intersect
invertible?	invertibleElseSplit?
invertibleSet	last
lastSubResultant	lastSubResultantElseSplit
latex	less?
mainVariable?	mainVariables
map	map!
member?	members
more?	mvar
normalized?	parts
preprocess	purelyAlgebraic?
purelyAlgebraicLeadingMonomial?	purelyTranscendental?
quasiComponent	reduce
reduced?	reduceByQuasiMonic
remainder	remove
removeDuplicates	removeZero
rest	retract
retractIfCan	rewriteIdealWithHeadRemainder
rewriteIdealWithRemainder	rewriteSetWithReduction
roughBase?	roughEqualIdeals?
roughSubIdeal?	roughUnitIdeal?
sample	select
size?	sort
squareFreePart	stronglyReduce
stronglyReduced?	triangular?
trivialIdeal?	variables
zeroSetSplit	zeroSetSplitIntoTriangularSystems
#?	?~=?
?=?	

— domain REGSET RegularTriangularSet —

```

)abbrev domain REGSET RegularTriangularSet
++ Author: Marc Moreno Maza
++ Date Created: 08/25/1998
++ Date Last Updated: 16/12/1998
++ References :
++ [1] M. MORENO MAZA "A new algorithm for computing triangular
++      decomposition of algebraic varieties" NAG Tech. Rep. 4/98.
++ Description:
++ This domain provides an implementation of regular chains.
++ Moreover, the operation zeroSetSplit is an implementation of a new
++ algorithm for solving polynomial systems by means of regular chains.

RegularTriangularSet(R,E,V,P) : Exports == Implementation where

  R : GcdDomain
  E : OrderedAbelianMonoidSup
  V : OrderedSet
  P : RecursivePolynomialCategory(R,E,V)
  N ==> NonNegativeInteger
  Z ==> Integer
  B ==> Boolean
  LP ==> List P
  PtoP ==> P -> P
  PS ==> GeneralPolynomialSet(R,E,V,P)
  PWT ==> Record(val : P, tower : $)
  BWT ==> Record(val : Boolean, tower : $)
  LpWT ==> Record(val : (List P), tower : $)
  Split ==> List $
  iprintpack ==> InternalPrintPackage()
  polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
  quasicomppack ==> QuasiComponentPackage(R,E,V,P,$)
  regsetgcdpack ==> RegularTriangularSetGcdPackage(R,E,V,P,$)
  regsetdecomppack ==> RegularSetDecompositionPackage(R,E,V,P,$)

Exports == RegularTriangularSetCategory(R,E,V,P) with

  internalAugment: (P,$,B,B,B,B,B) -> List $
    ++ \axiom{internalAugment(p,ts,b1,b2,b3,b4,b5)}
    ++ is an internal subroutine, exported only for developement.
  zeroSetSplit: (LP, B, B) -> Split
    ++ \axiom{zeroSetSplit(lp,clos?,info?)} has the same specifications as
    ++ zeroSetSplit from RegularTriangularSetCategory.
    ++ Moreover, if \axiom{clos?} then solves in the sense of the Zariski closure
    ++ else solves in the sense of the regular zeros. If \axiom{info?} then
    ++ do print messages during the computations.
  zeroSetSplit: (LP, B, B, B, B, B) -> Split
    ++ \axiom{zeroSetSplit(lp,b1,b2.b3,b4)}

```

```

    ++ is an internal subroutine, exported only for developement.
internalZeroSetSplit: (LP, B, B, B) -> Split
    ++ \axiom{internalZeroSetSplit(lp,b1,b2,b3)}
    ++ is an internal subroutine, exported only for developement.
pre_process: (LP, B, B) -> Record(val: LP, towers: Split)
    ++ \axiom{pre_process(lp,b1,b2)}
    ++ is an internal subroutine, exported only for developement.

```

```
Implementation == add
```

```
Rep ==> LP
```

```

rep(s:$):Rep == s pretend Rep
per(l:Rep):$ == l pretend $

```

```

copy ts ==
  per(copy(rep(ts))$LP)
empty() ==
  per([])
empty?(ts:$) ==
  empty?(rep(ts))
parts ts ==
  rep(ts)
members ts ==
  rep(ts)
map (f : PtoP, ts : $) : $ ==
  construct(map(f,rep(ts))$LP)$
map! (f : PtoP, ts : $) : $ ==
  construct(map!(f,rep(ts))$LP)$
member? (p,ts) ==
  member?(p,rep(ts))$LP
unitIdealIfCan() ==
  "failed"::Union($,"failed")
roughUnitIdeal? ts ==
  false
coerce(ts:$) : OutputForm ==
  lp : List(P) := reverse(rep(ts))
  brace([p::OutputForm for p in lp]$List(OutputForm))$OutputForm
mvar ts ==
  empty? ts => error "mvar$REGSET: #1 is empty"
  mvar(first(rep(ts)))$P
first ts ==
  empty? ts => "failed"::Union(P,"failed")
  first(rep(ts))::Union(P,"failed")
last ts ==
  empty? ts => "failed"::Union(P,"failed")
  last(rep(ts))::Union(P,"failed")
rest ts ==
  empty? ts => "failed"::Union($,"failed")
  per(rest(rep(ts)))::Union($,"failed")

```

```

coerce(ts:$) : (List P) ==
  rep(ts)

collectUpper (ts,v) ==
  empty? ts => ts
  lp := rep(ts)
  newlp : Rep := []
  while (not empty? lp) and (mvar(first(lp)) > v) repeat
    newlp := cons(first(lp),newlp)
    lp := rest lp
  per(reverse(newlp))

collectUnder (ts,v) ==
  empty? ts => ts
  lp := rep(ts)
  while (not empty? lp) and (mvar(first(lp)) >= v) repeat
    lp := rest lp
  per(lp)

construct(lp:List(P)) ==
  ts : $ := per([])
  empty? lp => ts
  lp := sort(infRittWu?,lp)
  while not empty? lp repeat
    eif := extendIfCan(ts,first(lp))
    not (eif case $) =>
      error"in construct : List P -> $ from REGSET : bad #1"
    ts := eif::$
    lp := rest lp
  ts

extendIfCan(ts:$,p:P) ==
  ground? p => "failed"::Union($,"failed")
  empty? ts =>
    p := primitivePart p
    (per([p]))::Union($,"failed")
  not (mvar(ts) < mvar(p)) => "failed"::Union($,"failed")
  invertible?(init(p),ts)@Boolean =>
    (per(cons(p,rep(ts))))::Union($,"failed")
  "failed"::Union($,"failed")

removeZero(p:P, ts:$): P ==
  (ground? p) or (empty? ts) => p
  v := mvar(p)
  ts_v_- := collectUnder(ts,v)
  if algebraic?(v,ts)
  then
    q := lazyPrem(p,select(ts,v)::P)
    zero? q => return q
    zero? removeZero(q,ts_v_-) => return 0

```

```

empty? ts_v_- => p
q: P := 0
while positive? degree(p,v) repeat
  q := removeZero(init(p),ts_v_-) * mainMonomial(p) + q
  p := tail(p)
q + removeZero(p,ts_v_-)

internalAugment(p:P,ts:$): $ ==
-- ASSUME that adding p to ts DOES NOT require any split
ground? p => error "in internalAugment$REGSET: ground? #1"
first(internalAugment(p,ts,false,false,false,false,false))

internalAugment(lp:List(P),ts:$): $ ==
-- ASSUME that adding p to ts DOES NOT require any split
empty? lp => ts
internalAugment(rest lp, internalAugment(first lp, ts))

internalAugment(p:P,ts:$,rem?:B,red?:B,prim?:B,sqfr?:B,extend?:B): Split ==
-- ASSUME p is not a constant
-- ASSUME mvar(p) is not algebraic w.r.t. ts
-- ASSUME init(p) invertible modulo ts
-- if rem? then REDUCE p by remainder
-- if prim? then REPLACE p by its main primitive part
-- if sqfr? then FACTORIZE SQUARE FREE p over R
-- if extend? DO NOT ASSUME every pol in ts_v_+ is invertible modulo ts
v := mvar(p)
ts_v_- := collectUnder(ts,v)
ts_v_+ := collectUpper(ts,v)
if rem? then p := remainder(p,ts_v_-).polnum
-- if rem? then p := reduceByQuasiMonic(p,ts_v_-)
if red? then p := removeZero(p,ts_v_-)
if prim? then p := mainPrimitivePart p
if sqfr?
  then
    lsfp := squareFreeFactors(p)$polsetpack
    lts: Split := [per(cons(f,rep(ts_v_-))) for f in lsfp]
  else
    lts: Split := [per(cons(p,rep(ts_v_-)))]
extend? => extend(members(ts_v_+),lts)
[per(concat(rep(ts_v_+),rep(us))) for us in lts]

augment(p:P,ts:$): List $ ==
ground? p => error "in augment$REGSET: ground? #1"
algebraic?(mvar(p),ts) => error "in augment$REGSET: bad #1"
-- ASSUME init(p) invertible modulo ts
-- DOES NOT ASSUME anything else.
-- THUS reduction, mainPrimitivePart and squareFree are NEEDED
internalAugment(p,ts,true,true,true,true,true)

extend(p:P,ts:$): List $ ==

```

```

ground? p => error "in extend$REGSET: ground? #1"
v := mvar(p)
not (mvar(ts) < mvar(p)) => error "in extend$REGSET: bad #1"
lts: List($) := []
split: List($) := invertibleSet(init(p),ts)
for us in split repeat
  lts := concat(augment(p,us),lts)
lts

invertible?(p:P,ts:$): Boolean ==
  toseInvertible?(p,ts)$regsetgcdpack

invertible?(p:P,ts:$): List BWT ==
  toseInvertible?(p,ts)$regsetgcdpack

invertibleSet(p:P,ts:$): Split ==
  toseInvertibleSet(p,ts)$regsetgcdpack

lastSubResultant(p1:P,p2:P,ts:$): List PWT ==
  toseLastSubResultant(p1,p2,ts)$regsetgcdpack

squareFreePart(p:P, ts: $): List PWT ==
  toseSquareFreePart(p,ts)$regsetgcdpack

intersect(p:P, ts: $): List($) == decompose([p], [ts], false, false)$regsetdecomppack

intersect(lp: LP, lts: List($)): List($) == decompose(lp, lts, false, false)$regsetdecomppack
  -- SOLVE in the regular zero sense
  -- and DO NOT PRINT info

decompose(p:P, ts: $): List($) == decompose([p], [ts], true, false)$regsetdecomppack

decompose(lp: LP, lts: List($)): List($) == decompose(lp, lts, true, false)$regsetdecomppack
  -- SOLVE in the closure sense
  -- and DO NOT PRINT info

zeroSetSplit(lp:List(P)) == zeroSetSplit(lp,true,false)
  -- by default SOLVE in the closure sense
  -- and DO NOT PRINT info

zeroSetSplit(lp:List(P), clos?: B) == zeroSetSplit(lp,clos?, false)
  -- DO NOT PRINT info

zeroSetSplit(lp:List(P), clos?: B, info?: B) ==
  -- if clos? then SOLVE in the closure sense
  -- if info? then PRINT info
  -- by default USE hash-tables
  -- and PREPROCESS the input system
  zeroSetSplit(lp,true,clos?,info?,true)

```

```

zeroSetSplit(lp:List(P),hash?:B,clos?:B,info?:B,prep?:B) ==
-- if hash? then USE hash-tables
-- if info? then PRINT information
-- if clos? then SOLVE in the closure sense
-- if prep? then PREPROCESS the input system
if hash?
then
  s1, s2, s3, dom1, dom2, dom3: String
  e: String := empty()$String
  if info? then (s1,s2,s3) := ("w","g","i") else (s1,s2,s3) := (e,e,e)
  if info?
  then
    (dom1, dom2, dom3) := ("QCMACK", "REGSETGCD: Gcd", "REGSETGCD: Inv Set")
  else
    (dom1, dom2, dom3) := (e,e,e)
  startTable!(s1,"W",dom1)$quasicomppack
  startTableGcd!(s2,"G",dom2)$regsetgcdpack
  startTableInvSet!(s3,"I",dom3)$regsetgcdpack
  lts := internalZeroSetSplit(lp,clos?,info?,prep?)
if hash?
then
  stopTable!()$quasicomppack
  stopTableGcd!()$regsetgcdpack
  stopTableInvSet!()$regsetgcdpack
lts

internalZeroSetSplit(lp:LP,clos?:B,info?:B,prep?:B) ==
-- if info? then PRINT information
-- if clos? then SOLVE in the closure sense
-- if prep? then PREPROCESS the input system
if prep?
then
  pp := pre_process(lp,clos?,info?)
  lp := pp.val
  lts := pp.towers
else
  ts: $ := [[]]
  lts := [ts]
lp := remove(zero?, lp)
any?(ground?, lp) => []
empty? lp => lts
empty? lts => lts
lp := sort(infRittWu?,lp)
clos? => decompose(lp,lts, clos?, info?)$regsetdecomppack
-- IN DIM > 0 with clos? the following is false ...
for p in lp repeat
  lts := decompose([p],lts, clos?, info?)$regsetdecomppack
lts

largeSystem?(lp:LP): Boolean ==

```

```

-- Gonnet and Gerdt and not Wu-Wang.2
#lp > 16 => true
#lp < 13 => false
lts: List($) := []
(#lp :: Z - numberOfVariables(lp,lts)$regsetdecomppack :: Z) > 3

smallSystem?(lp:LP): Boolean ==
-- neural, Vermeer, Liu, and not f-633 and not Hairer-2
#lp < 5

mediumSystem?(lp:LP): Boolean ==
-- f-633 and not Hairer-2
lts: List($) := []
(numberOfVariables(lp,lts)$regsetdecomppack :: Z - #lp :: Z) < 2

--
lin?(p:P):Boolean == ground?(init(p)) and one?(mdeg(p))
lin?(p:P):Boolean == ground?(init(p)) and (mdeg(p) = 1)

pre_process(lp:LP,clos?:B,info?:B): Record(val: LP, towers: Split) ==
-- if info? then PRINT information
-- if clos? then SOLVE in the closure sense
ts: $ := [[]];
lts: Split := [ts]
empty? lp => [lp,lts]
lp1: List P := []
lp2: List P := []
for p in lp repeat
  ground? (tail p) => lp1 := cons(p, lp1)
  lp2 := cons(p, lp2)
lts: Split := decompose(lp1,[ts],clos?,info?)$regsetdecomppack
probablyZeroDim?(lp)$polsetpack =>
  largeSystem?(lp) => return [lp2,lts]
  if #lp > 7
  then
    -- Butcher (8,8) + Wu-Wang.2 (13,16)
    lp2 := crushedSet(lp2)$polsetpack
    lp2 := remove(zero?,lp2)
    any?(ground?,lp2) => return [lp2, lts]
    lp3 := [p for p in lp2 | lin?(p)]
    lp4 := [p for p in lp2 | not lin?(p)]
    if clos?
    then
      lts := decompose(lp4,lts, clos?, info?)$regsetdecomppack
    else
      lp4 := sort(infRittWu?,lp4)
      for p in lp4 repeat
        lts := decompose([p],lts, clos?, info?)$regsetdecomppack
      lp2 := lp3
    else
      lp2 := crushedSet(lp2)$polsetpack

```



```

    lp2 := remove(zero?,lp2)
    any?(ground?,lp2) => return [lp2, lts]
  if clos?
  then
    lts := decompose(lp2,lts, clos?, info?)$regsetdecomppack
  else
    lp2 := sort(infRittWu?,lp2)
    for p in lp2 repeat
      lts := decompose([p],lts, clos?, info?)$regsetdecomppack
    lp2 := []
    return [lp2,lts]
  smallSystem?(lp) => [lp2,lts]
  mediumSystem?(lp) => [crushedSet(lp2)$polsetpack,lts]
  lp3 := [p for p in lp2 | lin?(p)]
  lp4 := [p for p in lp2 | not lin?(p)]
  if clos?
  then
    lts := decompose(lp4,lts, clos?, info?)$regsetdecomppack
  else
    lp4 := sort(infRittWu?,lp4)
    for p in lp4 repeat
      lts := decompose([p],lts, clos?, info?)$regsetdecomppack
  if clos?
  then
    lts := decompose(lp3,lts, clos?, info?)$regsetdecomppack
  else
    lp3 := sort(infRittWu?,lp3)
    for p in lp3 repeat
      lts := decompose([p],lts, clos?, info?)$regsetdecomppack
  lp2 := []
  return [lp2,lts]

```

— REGSET.dotabb —

```

"REGSET" [color="#88FF44",href="bookvol10.3.pdf#nameddest=REGSET"]
"RSETCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RSETCAT"]
"REGSET" -> "RSETCAT"

```

domain RESRING ResidueRing

— ResidueRing.input —

```

)set break resume
)sys rm -f ResidueRing.output
)spool ResidueRing.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ResidueRing
--R
--R ResidueRing(F: Field,Expon: OrderedAbelianMonoidSup,VarSet: OrderedSet,FPol: PolynomialCategory(F,Expon
--R Abbreviation for ResidueRing is RESRING
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for RESRING
--R
--R----- Operations -----
--R ??? : (%,F) -> %          ??? : (F,%) -> %
--R ??? : (%,%) -> %          ??? : (Integer,%) -> %
--R ??? : (PositiveInteger,%) -> %    ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %          ?-? : (%,%) -> %
--R -? : % -> %              ?=? : (%,%) -> Boolean
--R 1 : () -> %              0 : () -> %
--R ??? : (%,PositiveInteger) -> %    coerce : FPol -> %
--R coerce : F -> %           coerce : Integer -> %
--R coerce : % -> OutputForm    hash : % -> SingleInteger
--R latex : % -> String        lift : % -> FPol
--R one? : % -> Boolean        recip : % -> Union(%, "failed")
--R reduce : FPol -> %         sample : () -> %
--R zero? : % -> Boolean       ?~=? : (%,%) -> Boolean
--R ??? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)

```

— ResidueRing.help —

=====

ResidueRing examples

=====

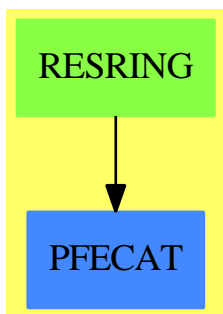
ResidueRing is the quotient of a polynomial ring by an ideal.

The ideal is given as a list of generators. The elements of the domain are equivalence classes expressed in terms of reduced elements

See Also:

o)show ResidueRing

ResidueRing (RESRING)



Exports:

0	1	characteristic	coerce	hash
latex	lift	one?	recip	reduce
sample	subtractIfCan	zero?	?~=?	?*?
?**?	?^?	?+?	?-?	-?
?=?				

— domain RESRING ResidueRing —

```

)abbrev domain RESRING ResidueRing
++ Author: P.Gianni
++ Date Created: December 1992
++ Description:
++ ResidueRing is the quotient of a polynomial ring by an ideal.
++ The ideal is given as a list of generators. The elements of the domain
++ are equivalence classes expressed in terms of reduced elements
  
```

```

ResidueRing(F,Expon,VarSet,FPol,LFPol) : Dom == Body
where
  F      : Field
  Expon  : OrderedAbelianMonoidSup
  VarSet : OrderedSet
  FPol   : PolynomialCategory(F, Expon, VarSet)
  LFPol  : List FPol
  
```

```

Dom == Join(CommutativeRing, Algebra F) with
  reduce   :   FPol -> $
  ++ reduce(f) produces the equivalence class of f in the residue ring
  coerce   :   FPol -> $
  ++ coerce(f) produces the equivalence class of f in the residue ring
  lift     :       $  -> FPol
  ++ lift(x) return the canonical representative of the equivalence class x
Body == add
--representation
Rep:= FPol
import GroebnerPackage(F,Expon,VarSet,FPol)
relations:= groebner(LFPol)
relations = [1] => error "the residue ring is the zero ring"
--declarations
x,y: $
--definitions
0 == 0$Rep
1 == 1$Rep
reduce(f : FPol) : $ == normalForm(f,relations)
coerce(f : FPol) : $ == normalForm(f,relations)
lift x == x :: Rep :: FPol
x + y == x +$Rep y
-x == -$Rep x
x*y == normalForm(lift(x *$Rep y),relations)
(n : Integer) * x == n *$Rep x
(a : F) * x == a *$Rep x
x = y == x =$Rep y
characteristic() == characteristic()$F
coerce(x) : OutputForm == coerce(x)$Rep

```

— RESRING.dotabb —

```

"RESRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=RESRING"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"RESRING" -> "PFECAT"

```

domain RESULT Result

— Result.input —

```
)set break resume
```

```

)sys rm -f Result.output
)spool Result.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Result
--R
--R Result is a domain constructor
--R Abbreviation for Result is RESULT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for RESULT
--R
--R----- Operations -----
--R copy : % -> % dictionary : () -> %
--R elt : (% , Symbol, Any) -> Any ?..? : (% , Symbol) -> Any
--R empty : () -> % empty? : % -> Boolean
--R entries : % -> List(Any) eq? : (% , %) -> Boolean
--R index? : (Symbol, %) -> Boolean indices : % -> List(Symbol)
--R key? : (Symbol, %) -> Boolean keys : % -> List(Symbol)
--R map : ((Any -> Any), %) -> % qelt : (% , Symbol) -> Any
--R sample : () -> % setelt : (% , Symbol, Any) -> Any
--R table : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?? : (% , %) -> Boolean if Any has SETCAT or Record(key: Symbol, entry: Any) has SETCAT
--R any? : ((Any -> Boolean), %) -> Boolean if $ has finiteAggregate
--R any? : ((Record(key: Symbol, entry: Any) -> Boolean), %) -> Boolean if $ has finiteAggregate
--R bag : List(Record(key: Symbol, entry: Any)) -> %
--R coerce : % -> OutputForm if Any has SETCAT or Record(key: Symbol, entry: Any) has SETCAT
--R construct : List(Record(key: Symbol, entry: Any)) -> %
--R convert : % -> InputForm if Record(key: Symbol, entry: Any) has KONVERT(INFORM)
--R count : ((Any -> Boolean), %) -> NonNegativeInteger if $ has finiteAggregate
--R count : (Any, %) -> NonNegativeInteger if $ has finiteAggregate and Any has SETCAT
--R count : (Record(key: Symbol, entry: Any), %) -> NonNegativeInteger if $ has finiteAggregate and Any has SETCAT
--R count : ((Record(key: Symbol, entry: Any) -> Boolean), %) -> NonNegativeInteger if $ has finiteAggregate and Any has SETCAT
--R dictionary : List(Record(key: Symbol, entry: Any)) -> %
--R entry? : (Any, %) -> Boolean if $ has finiteAggregate and Any has SETCAT
--R eval : (% , List(Equation(Any))) -> % if Any has EVALAB(ANY) and Any has SETCAT
--R eval : (% , Equation(Any)) -> % if Any has EVALAB(ANY) and Any has SETCAT
--R eval : (% , Any, Any) -> % if Any has EVALAB(ANY) and Any has SETCAT
--R eval : (% , List(Any), List(Any)) -> % if Any has EVALAB(ANY) and Any has SETCAT
--R eval : (% , List(Record(key: Symbol, entry: Any)), List(Record(key: Symbol, entry: Any))) -> % if Any has EVALAB(ANY) and Any has SETCAT
--R eval : (% , Record(key: Symbol, entry: Any), Record(key: Symbol, entry: Any)) -> % if Record(key: Symbol, entry: Any) has SETCAT
--R eval : (% , Equation(Record(key: Symbol, entry: Any))) -> % if Record(key: Symbol, entry: Any) has SETCAT
--R eval : (% , List(Equation(Record(key: Symbol, entry: Any)))) -> % if Record(key: Symbol, entry: Any) has SETCAT
--R every? : ((Any -> Boolean), %) -> Boolean if $ has finiteAggregate
--R every? : ((Record(key: Symbol, entry: Any) -> Boolean), %) -> Boolean if $ has finiteAggregate
--R extract! : % -> Record(key: Symbol, entry: Any)
--R fill! : (% , Any) -> % if $ has shallowlyMutable

```

```

--R find : ((Record(key: Symbol,entry: Any) -> Boolean),%) -> Union(Record(key: Symbol,entry: Any),"failed")
--R first : % -> Any if Symbol has ORDSET
--R hash : % -> SingleInteger if Any has SETCAT or Record(key: Symbol,entry: Any) has SETCAT
--R insert! : (Record(key: Symbol,entry: Any),%) -> %
--R inspect : % -> Record(key: Symbol,entry: Any)
--R latex : % -> String if Any has SETCAT or Record(key: Symbol,entry: Any) has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R map : (((Any,Any) -> Any),%,%) -> %
--R map : ((Record(key: Symbol,entry: Any) -> Record(key: Symbol,entry: Any)),%) -> %
--R map! : ((Any -> Any),%) -> % if $ has shallowlyMutable
--R map! : ((Record(key: Symbol,entry: Any) -> Record(key: Symbol,entry: Any)),%) -> % if $ has shallowlyMutable
--R maxIndex : % -> Symbol if Symbol has ORDSET
--R member? : (Any,%) -> Boolean if $ has finiteAggregate and Any has SETCAT
--R member? : (Record(key: Symbol,entry: Any),%) -> Boolean if $ has finiteAggregate and Record(key: Symbol,entry: Any) has SETCAT
--R members : % -> List(Any) if $ has finiteAggregate
--R members : % -> List(Record(key: Symbol,entry: Any)) if $ has finiteAggregate
--R minIndex : % -> Symbol if Symbol has ORDSET
--R more? : (%,NonNegativeInteger) -> Boolean
--R parts : % -> List(Any) if $ has finiteAggregate
--R parts : % -> List(Record(key: Symbol,entry: Any)) if $ has finiteAggregate
--R qsetelt! : (%,Symbol,Any) -> Any if $ has shallowlyMutable
--R reduce : (((Record(key: Symbol,entry: Any),Record(key: Symbol,entry: Any)) -> Record(key: Symbol,entry: Any)),%) -> %
--R reduce : (((Record(key: Symbol,entry: Any),Record(key: Symbol,entry: Any)) -> Record(key: Symbol,entry: Any)),%) -> %
--R reduce : (((Record(key: Symbol,entry: Any),Record(key: Symbol,entry: Any)) -> Record(key: Symbol,entry: Any)),%) -> %
--R remove : (Record(key: Symbol,entry: Any) -> Boolean),%) -> % if $ has finiteAggregate
--R remove : (Record(key: Symbol,entry: Any),%) -> % if $ has finiteAggregate and Record(key: Symbol,entry: Any) has SETCAT
--R remove! : (Symbol,%) -> Union(Any,"failed")
--R remove! : ((Record(key: Symbol,entry: Any) -> Boolean),%) -> % if $ has finiteAggregate
--R remove! : (Record(key: Symbol,entry: Any),%) -> % if $ has finiteAggregate
--R removeDuplicates : % -> % if $ has finiteAggregate and Record(key: Symbol,entry: Any) has SETCAT
--R search : (Symbol,%) -> Union(Any,"failed")
--R select : ((Record(key: Symbol,entry: Any) -> Boolean),%) -> % if $ has finiteAggregate
--R select! : ((Record(key: Symbol,entry: Any) -> Boolean),%) -> % if $ has finiteAggregate
--R showArrayValues : Boolean -> Boolean
--R showScalarValues : Boolean -> Boolean
--R size? : (%,NonNegativeInteger) -> Boolean
--R swap! : (%,Symbol,Symbol) -> Void if $ has shallowlyMutable
--R table : List(Record(key: Symbol,entry: Any)) -> %
--R ?~=? : (%,%) -> Boolean if Any has SETCAT or Record(key: Symbol,entry: Any) has SETCAT
--R
--E 1

```

```

)spool
)lisp (bye)

```

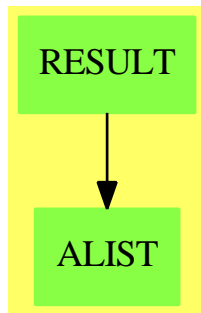
```
=====
Result examples
=====
```

A domain used to return the results from a call to the NAG library. It prints as a list of names and types, though the user may choose to display values automatically if he or she wishes.

See Also:

o)show Result

Result (RESULT)



See

- ⇒ “FortranCode” (FC) [7](#) on page [931](#)
- ⇒ “FortranProgram” (FORTRAN) [7](#) on page [955](#)
- ⇒ “ThreeDimensionalMatrix” (M3D) [21](#) on page [2702](#)
- ⇒ “SimpleFortranProgram” (SFORT) [20](#) on page [2405](#)
- ⇒ “Switch” (SWITCH) [20](#) on page [2629](#)
- ⇒ “FortranTemplate” (FTEM) [7](#) on page [967](#)
- ⇒ “FortranExpression” (FEXPR) [7](#) on page [946](#)

Exports:

any?	bag	coerce	construct	convert
copy	count	dictionary	entry?	elt
empty	empty?	entries	eq?	eval
every?	extract!	fill!	find	first
hash	index?	indices	insert!	inspect
key?	keys	latex	less?	map
map!	maxIndex	member?	members	minIndex
more?	parts	qelt	qsetelt!	reduce
remove	remove!	removeDuplicates	sample	search
select	select!	setelt	showArrayValues	showScalarValues
size?	swap!	table	#?	?=?
?^=?	??			

— domain RESULT Result —

```

)abbrev domain RESULT Result
++ Author: Didier Pinchon and Mike Dewar
++ Date Created:  8 April 1994
++ Date Last Updated: 28 June 1994
++ Description:
++ A domain used to return the results from a call to the NAG
++ Library.  It prints as a list of names and types, though the user may
++ choose to display values automatically if he or she wishes.

Result():Exports==Implementation where

0 ==> OutputForm

Exports ==> TableAggregate(Symbol,Any) with
  showScalarValues : Boolean -> Boolean
    ++ showScalarValues(true) forces the values of scalar components to be
    ++ displayed rather than just their types.
  showArrayValues : Boolean -> Boolean
    ++ showArrayValues(true) forces the values of array components to be
    ++ displayed rather than just their types.
  finiteAggregate

Implementation ==> Table(Symbol,Any) add

-- Constant
colon := ":"::Symbol::0
elide := "... "::Symbol::0

-- Flags
showScalarValuesFlag : Boolean := false
showArrayValuesFlag  : Boolean := false

cleanUpDomainForm(d:SEExpression):0 ==

```



```

not list? d => d::0
#d=1 => (car d)::0
-- If the car is an atom then we have a domain constructor, if not
-- then we have some kind of value. Since we often can't print these
-- **tters we just elide them.
not atom? car d => elide
prefix((car d)::0,[cleanUpDomainForm(u) for u in destruct cdr(d)]$List(0))

display(v:Any,d:SExpression):0 ==
not list? d => error "Domain form is non-list"
#d=1 =>
  showScalarValuesFlag => objectOf v
  cleanUpDomainForm d
car(d) = convert("Complex"::Symbol)@SExpression =>
  showScalarValuesFlag => objectOf v
  cleanUpDomainForm d
showArrayValuesFlag => objectOf v
cleanUpDomainForm d

makeEntry(k:Symbol,v:Any):0 ==
  hconcat [k::0,colon,display(v,dom v)]

coerce(r:%):0 ==
  bracket [makeEntry(key,r.key) for key in reverse! keys(r)]

showArrayValues(b:Boolean):Boolean == showArrayValuesFlag := b
showScalarValues(b:Boolean):Boolean == showScalarValuesFlag := b

```

— RESULT.dotabb —

```

"RESULT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=RESULT"]
"ALIST"  [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"RESULT" -> "ALIST"

```

domain RULE RewriteRule

— RewriteRule.input —

```

)set break resume
)sys rm -f RewriteRule.output

```

```

)spool RewriteRule.output
)set message test on
)set message auto off
)clear all

--S 1 of 4
)show RewriteRule
--R
--R RewriteRule(Base: SetCategory,R: Join(Ring,PatternMatchable(Base),OrderedSet,ConvertibleTo(Pattern(Base),
--R Abbreviation for RewriteRule is RULE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for RULE
--R
--R----- Operations -----
--R ==? : (%,% ) -> Boolean               coerce : Equation(F) -> %
--R coerce : % -> OutputForm              elt : (% ,F,PositiveInteger) -> F
--R ==? : (% ,F) -> F                    hash : % -> SingleInteger
--R latex : % -> String                  lhs : % -> F
--R pattern : % -> Pattern(Base)          retract : % -> Equation(F)
--R rhs : % -> F                         rule : (F,F,List(Symbol)) -> %
--R rule : (F,F) -> %                   ~=? : (%,% ) -> Boolean
--R quotedOperators : % -> List(Symbol)
--R retractIfCan : % -> Union(Equation(F),"failed")
--R suchThat : (% ,List(Symbol),(List(F) -> Boolean)) -> %
--R
--E 1

--S 2 of 4
logrule := rule log(x) + log(y) == log(x*y)
--R
--R (1) log(y) + log(x) + %B == log(x y) + %B
--R                                     Type: RewriteRule(Integer,Integer,Expression(Integer))
--E 2

--S 3 of 4
f := log(sin(x)) + log(x)
--R
--R (2) log(sin(x)) + log(x)
--R                                     Type: Expression(Integer)
--E 3

--S 4 of 4
logrule f
--R
--R (3) log(x sin(x))
--R                                     Type: Expression(Integer)
--E 4

)spool
)lisp (bye)

```

— RewriteRule.help —

```
=====
RewriteRule examples
=====
```

For example:

```
logrule := rule log(x) + log(y) == log(x*y)
          log(y) + log(x) + %C == log(x y) + %C

f := log(sin(x)) + log(x)
   log(sin(x)) + log(x)

logrule f
   log(x sin(x))
```

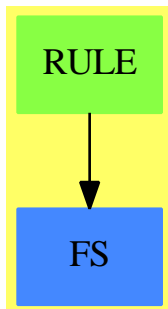
Note that you cannot write the simple form of a rule as:

```
rule1 := rule a*x+b == b*x+a
```

as this causes an infinite loop. The pattern is properly matched but the result also properly matches and the pattern is applied again.

See Also:

- o)show ApplyRules
 - o)show RewriteRule
-

RewriteRule (RULE)

See

⇒ “Ruleset” (RULESET) [19](#) on page [2342](#)

Exports:

coerce	elt	hash	latex	lhs
pattern	quotedOperators	retract	retractIfCan	rhs
rule	suchThat	?.?	?~=?	?=?

— domain **RULE RewriteRule** —

```

)abbrev domain RULE RewriteRule
++ Author: Manuel Bronstein
++ Date Created: 24 Oct 1988
++ Date Last Updated: 26 October 1993
++ Description:
++ Rules for the pattern matcher

RewriteRule(Base, R, F): Exports == Implementation where
  Base  : SetCategory
  R      : Join(Ring, PatternMatchable Base, OrderedSet,
                ConvertibleTo Pattern Base)
  F      : Join(FunctionSpace R, PatternMatchable Base,
                ConvertibleTo Pattern Base)

P      ==> Pattern Base

Exports ==>
  Join(SetCategory, Eltable(F, F), RetractableTo Equation F) with
    rule      : (F, F) -> $
      ++ rule(f, g) creates the rewrite rule: \spad{f == eval(g, g is f)},
      ++ with left-hand side f and right-hand side g.
      ++
      ++X logrule := rule log(x) + log(y) == log(x*y)
      ++X f := log(sin(x)) + log(x)
      ++X logrule f
    rule      : (F, F, List Symbol) -> $
  
```

```

++ rule(f, g, [f1,...,fn]) creates the rewrite rule
++ \spad{f == eval(eval(g, g is f), [f1,...,fn])},
++ that is a rule with left-hand side f and right-hand side g;
++ The symbols f1,...,fn are the operators that are considered
++ quoted, that is they are not evaluated during any rewrite,
++ but just applied formally to their arguments.
suchThat: ($, List Symbol, List F -> Boolean) -> $
++ suchThat(r, [a1,...,an], f) returns the rewrite rule r with
++ the predicate \spad{f(a1,...,an)} attached to it.
pattern : $ -> P
++ pattern(r) returns the pattern corresponding to
++ the left hand side of the rule r.
lhs      : $ -> F
++ lhs(r) returns the left hand side of the rule r.
rhs      : $ -> F
++ rhs(r) returns the right hand side of the rule r.
elt      : ($, F, PositiveInteger) -> F
++ elt(r,f,n) or r(f, n) applies the rule r to f at most n times.
quotedOperators: $ -> List Symbol
++ quotedOperators(r) returns the list of operators
++ on the right hand side of r that are considered
++ quoted, that is they are not evaluated during any rewrite,
++ but just applied formally to their arguments.

Implementation ==> add
import ApplyRules(Base, R, F)
import PatternFunctions1(Base, F)
import FunctionSpaceAssertions(R, F)

Rep := Record(pat: P, lft: F, rgt: F, qot: List Symbol)

mkRule      : (P, F, F, List Symbol) -> $
transformLhs: P -> Record(plus: F, times: F)
bad?        : Union(List P, "failed") -> Boolean
appear?     : (P, List P) -> Boolean
opt         : F -> P
F2Symbol    : F -> F

pattern x          == x.pat
lhs x              == x.lft
rhs x              == x.rgt
quotedOperators x  == x.qot
mkRule(pt, p, s, l) == [pt, p, s, l]
coerce(eq:Equation F):$ == rule(lhs eq, rhs eq, empty())
rule(l, r)          == rule(l, r, empty())
elt(r:$, s:F) == applyRules([r pretend RewriteRule(Base, R, F)], s)

suchThat(x, l, f) ==
  mkRule(suchThat(pattern x,l,f), lhs x, rhs x, quotedOperators x)

```

```

x = y ==
  (lhs x = lhs y) and (rhs x = rhs y) and
    (quotedOperators x = quotedOperators y)

elt(r:$, s:F, n:PositiveInteger) ==
  applyRules([r pretend RewriteRule(Base, R, F)], s, n)

-- remove the extra properties from the constant symbols in f
F2Symbol f ==
  l := select_!(z+>symbolIfCan z case Symbol, tower f)$List(Kernel F)
  eval(f, l, [symbolIfCan(k)::Symbol::F for k in l])

retractIfCan r ==
  constant? pattern r =>
    (u:= retractIfCan(lhs r)@Union(Kernel F,"failed")) case "failed"
      => "failed"
    F2Symbol(u::Kernel(F)::F) = rhs r
  "failed"

rule(p, s, l) ==
  lh := transformLhs(pt := convert(p)@P)
  mkRule(opt(lh.times) * (opt(lh.plus) + pt),
    lh.times * (lh.plus + p), lh.times * (lh.plus + s), l)

opt f ==
  retractIfCan(f)@Union(R, "failed") case R => convert f
  convert optional f

-- appear?(x, [p1,...,pn]) is true if x appears as a variable in
-- a composite pattern pi.
appear?(x, l) ==
  for p in l | p ^ x repeat
    member?(x, variables p) => return true
  false

-- a sum/product p1 @ ... @ pn is "bad" if it will not match
-- a sum/product p1 @ ... @ pn @ p(n+1)
-- in which case one should transform p1 @ ... @ pn to
-- p1 @ ... @ ?p(n+1) which does not change its meaning.
-- examples of "bad" combinations
--   sin(x) @ sin(y)      sin(x) @ x
-- examples of "good" combinations
--   sin(x) @ y
bad? u ==
  u case List(P) =>
    for x in u::List(P) repeat
      generic? x and not appear?(x, u::List(P)) => return false
    true
  false

```

```

transformLhs p ==
  bad? isPlus p  => [new()$Symbol :: F, 1]
  bad? isTimes p => [0, new()$Symbol :: F]
  [0, 1]

coerce(x:$):OutputForm ==
  infix(" == " :: Symbol :: OutputForm,
        lhs(x)::OutputForm, rhs(x)::OutputForm)

```

— RULE.dotabb —

```

"RULE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=RULE"]
"FS"   [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"RULE" -> "FS"

```

domain ROIRC RightOpenIntervalRootCharacterization

The domain RightOpenIntervalRootCharacterization is the main code that provides the functionalities of RealRootCharacterizationCategory for the case of archimedean fields. Abstract roots are encoded with a left closed right open interval containing the root together with a defining polynomial for the root.

CAVEATS

Since real algebraic expressions are stored as depending on "real roots" which are managed like variables, there is an ordering on these. This ordering is dynamical in the sense that any new algebraic takes precedence over older ones. In particular every creation function raises a new "real root". This has the effect that when you type something like $\sqrt{2} + \sqrt{2}$ you have two new variables which happen to be equal. To avoid this name the expression such as in $s2 := \sqrt{2}$; $s2 + s2$

Also note that computing times depend strongly on the ordering you implicitly provide. Please provide algebraics in the order which most natural to you.

LIMITATIONS

The file reclos.input show some basic use of the package. This packages uses algorithms which are published in [1] and [2] which are based on field arithmetics, inparticular for polynomial gcd related algorithms. This can be quite slow for high degree polynomials and subresultants methods usually work best. Betas versions of the package try to use these techniques in a better way and work significantly faster. These are mostly based on unpublished algorithms and cannot be distributed. Please contact the author if you have a particular problem to solve or want to use these versions.

Be aware that approximations behave as post-processing and that all computations are done exactly. They can thus be quite time consuming when depending on several "real roots".

— RightOpenIntervalRootCharacterization.input —

```
)set break resume
)sys rm -f RightOpenIntervalRootCharacterization.output
)spool RightOpenIntervalRootCharacterization.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RightOpenIntervalRootCharacterization
--R
--R RightOpenIntervalRootCharacterization(TheField: Join(OrderedRing,Field),ThePolDom: UnivariatePolynomial
--R Abbreviation for RightOpenIntervalRootCharacterization is ROIRC
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ROIRC
--R
--R----- Operations -----
--R ?? : (%,% ) -> Boolean          allRootsOf : ThePolDom -> List(%)
--R coerce : % -> OutputForm        hash : % -> SingleInteger
--R latex : % -> String             left : % -> TheField
--R middle : % -> TheField          refine : % -> %
--R right : % -> TheField           sign : (ThePolDom,%) -> Integer
--R size : % -> TheField            zero? : (ThePolDom,%) -> Boolean
--R ?~=? : (%,% ) -> Boolean
--R approximate : (ThePolDom,%,TheField) -> TheField
--R definingPolynomial : % -> ThePolDom
--R mightHaveRoots : (ThePolDom,%) -> Boolean
--R negative? : (ThePolDom,%) -> Boolean
--R positive? : (ThePolDom,%) -> Boolean
--R recip : (ThePolDom,%) -> Union(ThePolDom,"failed")
--R relativeApprox : (ThePolDom,%,TheField) -> TheField
--R rootOf : (ThePolDom,PositiveInteger) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)
```

— RightOpenIntervalRootCharacterization.help —

```
=====
RightOpenIntervalRootCharacterization examples
=====
```

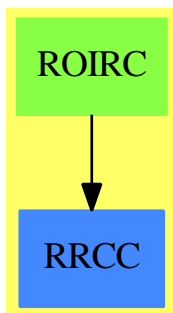

`RightOpenIntervalRootCharacterization` provides work with interval root coding.

See Also:

o `)show RightOpenIntervalRootCharacterization`

—————

RightOpenIntervalRootCharacterization (ROIRC)



See

⇒ “RealClosure” (RECLOS) [19](#) on page [2236](#)

Exports:

<code>allRootsOf</code>	<code>approximate</code>	<code>coerce</code>	<code>definingPolynomial</code>	<code>hash</code>
<code>latex</code>	<code>left</code>	<code>middle</code>	<code>mightHaveRoots</code>	<code>negative?</code>
<code>positive?</code>	<code>recip</code>	<code>refine</code>	<code>relativeApprox</code>	<code>right</code>
<code>rootOf</code>	<code>sign</code>	<code>size</code>	<code>zero?</code>	<code>?=?</code>
<code>?~=?</code>				

— domain **ROIRC** `RightOpenIntervalRootCharacterization` —

```

)abbrev domain ROIRC RightOpenIntervalRootCharacterization
++ Author: Renaud Rioboo
++ Date Created: summer 1992
++ Date Last Updated: January 2004
++ Description:
++ \axiomType{RightOpenIntervalRootCharacterization} provides work with
++ interval root coding.
  
```

`RightOpenIntervalRootCharacterization(TheField,ThePolDom) : PUB == PRIV` where

```

TheField : Join(OrderedRing,Field)
ThePolDom : UnivariatePolynomialCategory(TheField)
  
```

```

Z      ==> Integer
P      ==> ThePolDom
N      ==> NonNegativeInteger
B      ==> Boolean
UTIL   ==> RealPolynomialUtilitiesPackage(TheField,ThePolDom)
RRCC   ==> RealRootCharacterizationCategory
O ==> OutputForm
TwoPoints ==> Record(low:TheField , high:TheField)

PUB == RealRootCharacterizationCategory(TheField, ThePolDom) with

  left      :      $      -> TheField
    ++ \axiom{left(rootChar)} is the left bound of the isolating
    ++ interval
  right     :      $      -> TheField
    ++ \axiom{right(rootChar)} is the right bound of the isolating
    ++ interval
  size      :      $      -> TheField
    ++ The size of the isolating interval
  middle    :      $      -> TheField
    ++ \axiom{middle(rootChar)} is the middle of the isolating
    ++ interval
  refine    :      $      -> $
    ++ \axiom{refine(rootChar)} shrinks isolating interval around
    ++ \axiom{rootChar}
  mightHaveRoots : (P,$) -> B
    ++ \axiom{mightHaveRoots(p,r)} is false if \axiom{p.r} is not 0
  relativeApprox : (P,$,TheField) -> TheField
    ++ \axiom{relativeApprox(exp,c,p) = a} is relatively close to exp
    ++ as a polynomial in c ip to precision p

PRIV == add

-- local functions

makeChar:      (TheField,TheField,ThePolDom) -> $
refine! :      $ -> $
sturmIsolate : (List(P), TheField, TheField,N,N) -> List TwoPoints
isolate :      List(P) -> List TwoPoints
rootBound :    P -> TheField
-- varStar :    P -> N
linearRecip :   ( P , $) -> Union(P, "failed")
linearZero? :   (TheField,$) -> B
linearSign :     (P,$) -> Z
sturmNthRoot : (List(P), TheField, TheField,N,N,N) -> Union(TwoPoints,"failed")
addOne :        P -> P

```

```

minus :                               P          ->    P
translate :                          (P,TheField) ->    P
dilate :                             (P,TheField) ->    P
invert :                             P          ->    P
evalOne :                            P          ->    TheField
hasVars1:                           List(TheField) ->    B
hasVars:                             P          ->    B

-- Representation

Rep:= Record(low:TheField,high:TheField,defPol:ThePolDom)

-- and now the code !

size(rootCode) ==
  rootCode.high - rootCode.low

relativeApprox(pval,rootCode,prec) ==
  -- beurk !
  dPol := rootCode.defPol
  degree(dPol) = 1 =>
    c := -coefficient(dPol,0)/leadingCoefficient(dPol)
    pval.c
  pval := pval rem dPol
  degree(pval) = 0 => leadingCoefficient(pval)
  zero?(pval,rootCode) => 0
  while mightHaveRoots(pval,rootCode) repeat
    rootCode := refine(rootCode)
  dpval := differentiate(pval)
  degree(dpval) = 0 =>
    l := left(rootCode)
    r := right(rootCode)
    a := pval.l
    b := pval.r
    while ( abs(2*(a-b)/(a+b)) > prec ) repeat
      rootCode := refine(rootCode)
      l := left(rootCode)
      r := right(rootCode)
      a := pval.l
      b := pval.r
      (a+b)/(2::TheField)
    zero?(dpval,rootCode) =>
      relativeApprox(pval,
        [left(rootCode),
         right(rootCode),
         gcd(dpval,rootCode.defPol)]$Rep,
        prec)
  while mightHaveRoots(dpval,rootCode) repeat
    rootCode := refine(rootCode)

```

```

l := left(rootCode)
r := right(rootCode)
a := pval.l
b := pval.r
while ( abs(2*(a-b)/(a+b)) > prec ) repeat
  rootCode := refine(rootCode)
  l := left(rootCode)
  r := right(rootCode)
  a := pval.l
  b := pval.r
(a+b)/(2::TheField)

approximate(pval,rootCode,prec) ==
-- glurp
dPol := rootCode.defPol
degree(dPol) = 1 =>
  c := -coefficient(dPol,0)/leadingCoefficient(dPol)
  pval.c
pval := pval rem dPol
degree(pval) = 0 => leadingCoefficient(pval)
dpval := differentiate(pval)
degree(dpval) = 0 =>
  l := left(rootCode)
  r := right(rootCode)
  while ( abs((a := pval.l) - (b := pval.r)) > prec ) repeat
    rootCode := refine(rootCode)
    l := left(rootCode)
    r := right(rootCode)
  (a+b)/(2::TheField)
zero?(dpval,rootCode) =>
  approximate(pval,
    [left(rootCode),
     right(rootCode),
     gcd(dpval,rootCode.defPol)]$Rep,
    prec)
while mightHaveRoots(dpval,rootCode) repeat
  rootCode := refine(rootCode)
l := left(rootCode)
r := right(rootCode)
while ( abs((a := pval.l) - (b := pval.r)) > prec ) repeat
  rootCode := refine(rootCode)
  l := left(rootCode)
  r := right(rootCode)
(a+b)/(2::TheField)

addOne(p) == p.(monomial(1,1)+(1::P))

minus(p) == p.(monomial(-1,1))

```

```

translate(p,a) == p.(monomial(1,1)+(a::P))

dilate(p,a) == p.(monomial(a,1))

evalOne(p) == "+" / coefficients(p)

invert(p) ==
  d := degree(p)
  mapExponents(z +-> (d-z)::N, p)

rootBound(p) ==
  res : TheField := 1
  raw : TheField := 1+boundOfCauchy(p)$UTIL
  while (res < raw) repeat
    res := 2*(res)
  res

sturmNthRoot(lp,l,r,vl,vr,n) ==
  nv := (vl - vr)::N
  nv < n => "failed"
  ((nv = 1) and (n = 1)) => [l,r]
  int := (l+r)/(2::TheField)
  lt:List(TheField):=[]
  for t in lp repeat
    lt := cons(t.int , lt)
  vi := sturmVariationsOf(reverse! lt)$UTIL
  o :Z := n - vl + vi
  if o > 0
  then
    sturmNthRoot(lp,int,r,vi,vr,o::N)
  else
    sturmNthRoot(lp,l,int,vl,vi,n)

sturmIsolate(lp,l,r,vl,vr) ==
  r <= l => error "ROIRC: sturmIsolate: bad bounds"
  n := (vl - vr)::N
  zero?(n) => []
  one?(n) => [[l,r]]
  int := (l+r)/(2::TheField)
  vi := sturmVariationsOf( [t.int for t in lp ] )$UTIL
  append(sturmIsolate(lp,l,int,vl,vi),sturmIsolate(lp,int,r,vi,vr))

isolate(lp) ==
  b := rootBound(first(lp))
  l1,l2 : List(TheField)
  (l1,l2) := ([], [])
  for t in reverse(lp) repeat
    if odd?(degree(t))
    then
      (l1,l2):= (cons(-leadingCoefficient(t),l1),

```

```

        cons(leadingCoefficient(t),l2))
    else
        (l1,l2):= (cons(leadingCoefficient(t),l1),
                    cons(leadingCoefficient(t),l2))
    sturmIsolate(lp,
                -b,
                b,
                sturmVariationsOf(l1)$UTIL,
                sturmVariationsOf(l2)$UTIL)

rootOf(pol,n) ==
  ls := sturmSequence(pol)$UTIL
  pol := unitCanonical(first(ls)) -- this one is SqFR
  degree(pol) = 0 => "failed"
  numberOfMonomials(pol) = 1 => ([0,1,monomial(1,1)]$Rep)::$
  b := rootBound(pol)
  l1,l2 : List(TheField)
  (l1,l2) := ([], [])
  for t in reverse(ls) repeat
    if odd?(degree(t))
    then
      (l1,l2):= (cons(leadingCoefficient(t),l1),
                  cons(-leadingCoefficient(t),l2))
    else
      (l1,l2):= (cons(leadingCoefficient(t),l1),
                  cons(leadingCoefficient(t),l2))
  res := sturmNthRoot(ls,
                    -b,
                    b,
                    sturmVariationsOf(l2)$UTIL,
                    sturmVariationsOf(l1)$UTIL,
                    n)
  res case "failed" => "failed"
  makeChar(res.low,res.high,pol)

allRootsOf(pol) ==
  ls := sturmSequence(unitCanonical pol)$UTIL
  pol := unitCanonical(first(ls)) -- this one is SqFR
  degree(pol) = 0 => []
  numberOfMonomials(pol) = 1 => [[0,1,monomial(1,1)]$Rep]
  [ makeChar(term.low,term.high,pol) for term in isolate(ls) ]

hasVarsl(l:List(TheField)) ==
  null(l) => false
  f := sign(first(l))
  for term in rest(l) repeat
    if f*term < 0 then return(true)
  false

```

```

hasVars(p:P) ==
  zero?(p) => error "ROIRC: hasVars: null polynomial"
  zero?(coefficient(p,0)) => true
  hasVars1(coefficients(p))

mightHaveRoots(p,rootChar) ==
  a := rootChar.low
  q := translate(p,a)
  not(hasVars(q)) => false
--   varStar(q) = 0 => false
  a := (rootChar.high) - a
  q := dilate(q,a)
  sign(coefficient(q,0))*sign(evalOne(q)) <= 0 => true
  q := minus(addOne(q))
  not(hasVars(q)) => false
--   varStar(q) = 0 => false
  q := invert(q)
  hasVars(addOne(q))
--   ~(varStar(addOne(q)) = 0)

coerce(rootChar:$):0 ==
  commaSeparate([ hconcat("[ " :: 0 , (rootChar.low)::0),
                  hconcat((rootChar.high)::0,"[ " ::0 ) ])

c1 = c2 ==
  mM := max(c1.low,c2.low)
  Mm := min(c1.high,c2.high)
  mM >= Mm => false
  rr : ThePolDom := gcd(c1.defPol,c2.defPol)
  degree(rr) = 0 => false
  sign(rr.mM) * sign(rr.Mm) <= 0

makeChar(left,right,pol) ==
-- The following lines of code, which check for a possible error,
-- cause major performance problems and were removed by Renaud Rioboo,
-- the original author. They were originally inserted for debugging.
--   right <= left => error "ROIRC: makeChar: Bad interval"
--   (pol.left * pol.right) > 0 => error "ROIRC: makeChar: Bad pol"
  res :$ := [left,right,leadingMonomial(pol)+reductum(pol)]$Rep -- safe copy
  while zero?(pol.(res.high)) repeat refine!(res)
  while (res.high * res.low < 0 ) repeat refine!(res)
  zero?(pol.(res.low)) => [res.low,res.high,monomial(1,1)-(res.low)::P]
  res

definingPolynomial(rootChar) == rootChar.defPol

linearRecip(toTest,rootChar) ==
  c := - inv(leadingCoefficient(toTest)) * coefficient(toTest,0)
  r := recip(rootChar.defPol.c)

```

```

    if (r case "failed")
    then
      if (c - rootChar.low) * (c - rootChar.high) <= 0
      then
        "failed"
      else
        newPol := (rootChar.defPol exquo toTest)::P
        ((1$ThePolDom - inv(newPol.c)*newPol) exquo toTest)::P
    else
      ((1$ThePolDom - (r::TheField)*rootChar.defPol) exquo toTest)::P

  recip(toTest,rootChar) ==
    degree(toTest) = 0 or degree(rootChar.defPol) <= degree(toTest) =>
      error "IRC: recip: Not reduced"
    degree(rootChar.defPol) = 1 =>
      error "IRC: recip: Linear Defining Polynomial"
    degree(toTest) = 1 =>
      linearRecip(toTest, rootChar)
  d := extendedEuclidean((rootChar.defPol),toTest)
  (degree(d.generator) = 0 ) =>
    d.coef2
  d.generator := unitCanonical(d.generator)
  (d.generator.(rootChar.low) *
  d.generator.(rootChar.high)<= 0) => "failed"
  newPol := (rootChar.defPol exquo (d.generator))::P
  degree(newPol) = 1 =>
    c := - inv(leadingCoefficient(newPol)) * coefficient(newPol,0)
    inv(toTest.c)::P
  degree(toTest) = 1 =>
    c := - coefficient(toTest,0)/ leadingCoefficient(toTest)
    ((1$ThePolDom - inv(newPol.c)*newPol) exquo toTest)::P
  d := extendedEuclidean(newPol,toTest)
  d.coef2

  linearSign(toTest,rootChar) ==
    c := - inv(leadingCoefficient(toTest)) * coefficient(toTest,0)
    ev := sign(rootChar.defPol.c)
    if zero?(ev)
    then
      if (c - rootChar.low) * (c - rootChar.high) <= 0
      then
        0
      else
        sign(toTest.(rootChar.high))
    else
      if (ev*sign(rootChar.defPol.(rootChar.high)) <= 0 )
      then
        sign(toTest.(rootChar.high))
      else
        sign(toTest.(rootChar.low))

```



```

sign(toTest,rootChar) ==
  degree(toTest) = 0 or degree(rootChar.defPol) <= degree(toTest) =>
    error "IRC: sign: Not reduced"
  degree(rootChar.defPol) = 1 =>
    error "IRC: sign: Linear Defining Polynomial"
  degree(toTest) = 1 =>
    linearSign(toTest, rootChar)
  s := sign(leadingCoefficient(toTest))
  toTest := monomial(1,degree(toTest))+
    inv(leadingCoefficient(toTest))*reductum(toTest)
  delta := gcd(toTest,rootChar.defPol)
  newChar := [rootChar.low,rootChar.high,rootChar.defPol]$Rep
  if degree(delta) > 0
  then
    if sign(delta.(rootChar.low) * delta.(rootChar.high)) <= 0
    then
      return(0)
    else
      newChar.defPol := (newChar.defPol exquo delta) :: P
      toTest := toTest rem (newChar.defPol)
      degree(toTest) = 0 => s * sign(leadingCoefficient(toTest))
      degree(toTest) = 1 => s * linearSign(toTest, newChar)
      while mightHaveRoots(toTest,newChar) repeat
        newChar := refine(newChar)
      s*sign(toTest.(newChar.low))

linearZero?(c,rootChar) ==
  zero?((rootChar.defPol).c) and
  (c - rootChar.low) * (c - rootChar.high) <= 0

zero?(toTest,rootChar) ==
  degree(toTest) = 0 or degree(rootChar.defPol) <= degree(toTest) =>
    error "IRC: zero?: Not reduced"
  degree(rootChar.defPol) = 1 =>
    error "IRC: zero?: Linear Defining Polynomial"
  degree(toTest) = 1 =>
    linearZero?(- inv(leadingCoefficient(toTest)) * coefficient(toTest,0),
      rootChar)
  toTest := monomial(1,degree(toTest))+
    inv(leadingCoefficient(toTest))*reductum(toTest)
  delta := gcd(toTest,rootChar.defPol)
  degree(delta) = 0 => false
  sign(delta.(rootChar.low) * delta.(rootChar.high)) <= 0

refine!(rootChar) ==
  -- this is not a safe function, it can work with badly created object
  -- we do not assume (rootChar.defPol).(rootChar.high) <> 0
  int := middle(rootChar)

```

```

s1 := sign((rootChar.defPol).(rootChar.low))
zero?(s1) =>
  rootChar.high := int
  rootChar.defPol := monomial(1,1) - (rootChar.low)::P
  rootChar
s2 := sign((rootChar.defPol).int)
zero?(s2) =>
  rootChar.low := int
  rootChar.defPol := monomial(1,1) - int::P
  rootChar
if (s1*s2 < 0)
then
  rootChar.high := int
else
  rootChar.low := int
  rootChar

refine(rootChar) ==
-- we assume (rootChar.defPol).(rootChar.high) <> 0
int := middle(rootChar)
s := (rootChar.defPol).int * (rootChar.defPol).(rootChar.high)
zero?(s) => [int,rootChar.high,monomial(1,1)-int::P]
if s < 0
then
  [int,rootChar.high,rootChar.defPol]
else
  [rootChar.low,int,rootChar.defPol]

left(rootChar) == rootChar.low

right(rootChar) == rootChar.high

middle(rootChar) == (rootChar.low + rootChar.high)/(2::TheField)

-- varStar(p) == -- if 0 no roots in [0,:infy[
--   res : N := 0
--   lsg := sign(coefficient(p,0))
--   l := [ sign(i) for i in reverse!(coefficients(p))]
--   for sg in l repeat
--     if (sg ^= lsg) then res := res + 1
--     lsg := sg
--   res

```

— ROIRC.dotabb —

"ROIRC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ROIRC"]
"RRCC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RRCC"]

```
"R0IRC" -> "RRCC"
```

domain ROMAN RomanNumeral

— RomanNumeral.input —

```
)set break resume
)sys rm -f RomanNumeral.output
)spool RomanNumeral.output
)set message test on
)set message auto off
)clear all
--S 1 of 15
f := operator 'f
--R
--R
--R (1) f
--R
--R                                          Type: BasicOperator
--E 1

--S 2 of 15
D(f x,x,7)
--R
--R
--R (vii)
--R (2) f (x)
--R
--R                                          Type: Expression(Integer)
--E 2

--S 3 of 15
a := roman(1978 - 1965)
--R
--R
--R (3) XIII
--R
--R                                          Type: RomanNumeral
--E 3

--S 4 of 15
x : UTS(ROMAN,'x,0) := x
--R
--R
--R (4) x
--R
--R                                          Type: UnivariateTaylorSeries(RomanNumeral,x,0)
```

--E 4

--S 5 of 15

recip(1 - x - x**2)

--R

--R

--R (5)

--R
$$I + x + II x^2 + III x^3 + V x^4 + VIII x^5 + XIII x^6 + XXI x^7 + XXXIV x^8$$

--R +

--R
$$LV x^9 + LXXXIX x^{10} + O(x^{11})$$

--R Type: Union(UnivariateTaylorSeries(RomanNumeral,x,0),...)

--E 5

--S 6 of 15

m : MATRIX FRAC ROMAN

--R

--R

Type: Void

--E 6

--S 7 of 15

m := matrix [[1/(i + j) for i in 1..3] for j in 1..3]

--R

--R

--R
$$\begin{array}{ccc} + I & I & I+ \\ |-- & --- & --| \\ |II & III & IV| \\ | & & | \\ | I & I & I | \\ (7) & |--- & -- & - | \\ |III & IV & V | \\ | & & | \\ | I & I & I| \\ |-- & - & --| \\ +IV & V & VI+ \end{array}$$

--R

--R

--R

--R

--R

--R

--R

--R

--R

--R

Type: Matrix(Fraction(RomanNumeral))

--E 7

--S 8 of 15

inverse m

--R

--R

--R
$$\begin{array}{ccc} +LXXII & - CCXL & CLXXX + \\ | & & | \\ (8) & |- CCXL & CM & - DCCXX| \\ | & & | \\ +CLXXX & - DCCXX & DC & + \end{array}$$

--R

--R

--R

--R

--R

Type: Union(Matrix(Fraction(RomanNumeral)),...)

--E 8

[illegible]

```

--S 15 of 15
b rem a
--R
--R
--R (15) IX
--R
--R                                         Type: RomanNumeral
--E 15
)spool
)lisp (bye)

```

— RomanNumeral.help —

=====

RomanNumeral Examples

=====

The Roman numeral package was added to Axiom in MCMLXXXVI for use in denoting higher order derivatives.

For example, let f be a symbolic operator.

```

f := operator 'f
f

```

Type: BasicOperator

This is the seventh derivative of f with respect to x .

```

D(f x,x,7)
(vii)
f      (x)

```

Type: Expression Integer

You can have integers printed as Roman numerals by declaring variables to be of type RomanNumeral (abbreviation ROMAN).

```

a := roman(1978 - 1965)
XIII

```

Type: RomanNumeral

This package now has a small but devoted group of followers that claim this domain has shown its efficacy in many other contexts. They claim that Roman numerals are every bit as useful as ordinary integers.

In a sense, they are correct, because Roman numerals form a ring and you can therefore construct polynomials with Roman numeral coefficients, matrices over Roman numerals, etc..

```
x : UTS(ROMAN,'x,0) := x
x
Type: UnivariateTaylorSeries(RomanNumeral,x,0)
```

Was Fibonacci Italian or ROMAN?

```
recip(1 - x - x**2)
      2      3      4      5      6      7      8
      I + x + II x + III x + V x + VIII x + XIII x + XXI x + XXXIV x
+
      9      10      11
      LV x + LXXXIX x + O(x )
Type: Union(UnivariateTaylorSeries(RomanNumeral,x,0),...)
```

You can also construct fractions with Roman numeral numerators and denominators, as this matrix Hilberticus illustrates.

```
m : MATRIX FRAC ROMAN
Type: Void

m := matrix [ [1/(i + j) for i in 1..3] for j in 1..3]
      + I      I      I+
      |--    ---  --|
      |II     III  IV|
      |       |
      | I      I   I |
      |---   --   - |
      |III    IV   V |
      |       |
      | I      I   I |
      |--    -   --|
      +IV     V    VI+
Type: Matrix Fraction RomanNumeral
```

Note that the inverse of the matrix has integral ROMAN entries.

```
inverse m
      +LXXII   - CCXL   CLXXX +
      |         |
      |- CCXL   CM      - DCCXX|
      |         |
      +CLXXX   - DCCXX   DC      +
Type: Union(Matrix Fraction RomanNumeral,...)
```

Unfortunately, the spoil-sports say that the fun stops when the numbers get big---mostly because the Romans didn't establish conventions about representing very large numbers.

```
y := factorial 10
3628800
```

Type: PositiveInteger

You work it out!

```
roman y
(((I)))(((I)))(((I))) ((I))((I))((I))((I))((I))((I)) ((I))(
(I)) MMMMMMMDCCC
```

Type: RomanNumeral

Issue the system command `)show RomanNumeral` to display the full list of operations defined by RomanNumeral.

```
a := roman(78)
LXXVIII
```

Type: RomanNumeral

```
b := roman(87)
LXXXVII
```

Type: RomanNumeral

```
a + b
CLXV
```

Type: RomanNumeral

```
a * b
MMMMMDCCCLXXXVI
```

Type: RomanNumeral

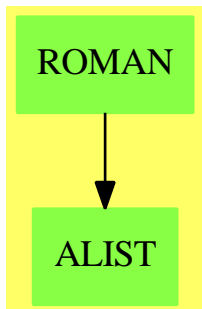
```
b rem a
IX
```

Type: RomanNumeral

See Also:

- o)help Integer
- o)help Complex
- o)help Factored
- o)help Records
- o)help Fraction
- o)help RadixExpansion
- o)help HexadecimalExpansion
- o)help BinaryExpansion
- o)help DecimalExpansion
- o)help IntegerNumberTheoryFunctions
- o)show RomanNumeral

RomanNumeral (ROMAN)



See

⇒ “Integer” (INT) [10](#) on page [1361](#)

⇒ “NonNegativeInteger” (NNI) [15](#) on page [1739](#)

⇒ “PositiveInteger” (PI) [17](#) on page [2100](#)

Exports:

0	1	abs	addmod
associates?	base	binomial	bit?
characteristic	coerce	convert	copy
D	dec	differentiate	divide
euclideanSize	even?	expressIdealMember	exquo
extendedEuclidean	extendedEuclidean	factor	factorial
gcd	gcdPolynomial	hash	inc
init	invmod	latex	lcm
length	mask	max	min
mulmod	multiEuclidean	negative?	nextItem
odd?	one?	patternMatch	permutation
positive?	positiveRemainder	powmod	prime?
principalIdeal	random	rational	rational?
rationalIfCan	recip	reducedSystem	retract
retractIfCan	roman	sample	shift
sign	sizeLess?	squareFree	squareFreePart
submod	subtractIfCan	symmetricRemainder	unit?
unitCanonical	unitNormal	zero?	???
??*	?+?	?-?	-?
?<?	?<=?	?=?	?>?
?>=?	?^?	?~=?	?quo?
?rem?			

— domain ROMAN RomanNumeral —

```
)abbrev domain ROMAN RomanNumeral
++ Author: Mark Botch
++ Description:
```

```

++ \spadtype{RomanNumeral} provides functions for converting
++ integers to roman numerals.

RomanNumeral(): IntegerNumberSystem with
  canonical
  ++ mathematical equality is data structure equality.
  canonicalsClosed
  ++ two positives multiply to give positive.
  noetherian
  ++ ascending chain condition on ideals.
  convert: Symbol -> %
  ++ convert(n) creates a roman numeral for symbol n.
  roman : Symbol -> %
  ++ roman(n) creates a roman numeral for symbol n.
  roman : Integer -> %
  ++ roman(n) creates a roman numeral for n.

== Integer add
  import NumberFormats()

  roman(n:Integer) == n::%
  roman(sy:Symbol) == convert sy
  convert(sy:Symbol):% == ScanRoman(string sy)::%

  coerce(r:%):OutputForm ==
    n := convert(r)@Integer
    -- okay, we stretch it
    zero? n => n::OutputForm
    negative? n => -((-r)::OutputForm)
    FormatRoman(n::PositiveInteger)::Symbol::OutputForm



---



— ROMAN.dotabb —

"ROMAN" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ROMAN"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ROMAN" -> "ALIST"



---



```

domain ROUTINE RoutinesTable

— RoutinesTable.input —

```

)set break resume
)sys rm -f RoutinesTable.output
)spool RoutinesTable.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RoutinesTable
--R
--R RoutinesTable is a domain constructor
--R Abbreviation for RoutinesTable is ROUTINE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ROUTINE
--R
--R----- Operations -----
--R concat : (% , %) -> %                copy : % -> %
--R deleteRoutine! : (% , Symbol) -> %    dictionary : () -> %
--R elt : (% , Symbol , Any) -> Any        ? . ? : (% , Symbol) -> Any
--R empty : () -> %                       empty? : % -> Boolean
--R entries : % -> List(Any)              eq? : (% , %) -> Boolean
--R getMeasure : (% , Symbol) -> Float    index? : (Symbol , %) -> Boolean
--R indices : % -> List(Symbol)           key? : (Symbol , %) -> Boolean
--R keys : % -> List(Symbol)              map : ((Any -> Any) , %) -> %
--R qelt : (% , Symbol) -> Any            routines : () -> %
--R sample : () -> %                     selectFiniteRoutines : % -> %
--R selectNonFiniteRoutines : % -> %      selectODEIVPRoutines : % -> %
--R selectPDERoutines : % -> %           setelt : (% , Symbol , Any) -> Any
--R showTheRoutinesTable : () -> %        table : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?= ? : (% , %) -> Boolean if Any has SETCAT or Record(key: Symbol, entry: Any) has SETCAT
--R any? : ((Any -> Boolean) , %) -> Boolean if $ has finiteAggregate
--R any? : ((Record(key: Symbol, entry: Any) -> Boolean) , %) -> Boolean if $ has finiteAggregate
--R bag : List(Record(key: Symbol, entry: Any)) -> %
--R changeMeasure : (% , Symbol , Float) -> %
--R changeThreshold : (% , Symbol , Float) -> %
--R coerce : % -> OutputForm if Any has SETCAT or Record(key: Symbol, entry: Any) has SETCAT
--R construct : List(Record(key: Symbol, entry: Any)) -> %
--R convert : % -> InputForm if Record(key: Symbol, entry: Any) has KONVERT(INFORM)
--R count : ((Any -> Boolean) , %) -> NonNegativeInteger if $ has finiteAggregate
--R count : (Any , %) -> NonNegativeInteger if $ has finiteAggregate and Any has SETCAT
--R count : (Record(key: Symbol, entry: Any) , %) -> NonNegativeInteger if $ has finiteAggregate and Any has SETCAT
--R count : ((Record(key: Symbol, entry: Any) -> Boolean) , %) -> NonNegativeInteger if $ has finiteAggregate
--R dictionary : List(Record(key: Symbol, entry: Any)) -> %
--R entry? : (Any , %) -> Boolean if $ has finiteAggregate and Any has SETCAT
--R eval : (% , List(Equation(Any))) -> % if Any has EVALAB(ANY) and Any has SETCAT
--R eval : (% , Equation(Any)) -> % if Any has EVALAB(ANY) and Any has SETCAT
--R eval : (% , Any , Any) -> % if Any has EVALAB(ANY) and Any has SETCAT
--R eval : (% , List(Any) , List(Any)) -> % if Any has EVALAB(ANY) and Any has SETCAT
--R eval : (% , List(Record(key: Symbol, entry: Any)) , List(Record(key: Symbol, entry: Any))) -> % if

```

```

--R eval : (% ,Record(key: Symbol,entry: Any),Record(key: Symbol,entry: Any)) -> % if Record(key: Symbol,entry: Any) has EVALAB(R
--R eval : (% ,Equation(Record(key: Symbol,entry: Any))) -> % if Record(key: Symbol,entry: Any) has EVALAB(R
--R eval : (% ,List(Equation(Record(key: Symbol,entry: Any)))) -> % if Record(key: Symbol,entry: Any) has EVALAB(R
--R every? : ((Any -> Boolean),%) -> Boolean if $ has finiteAggregate
--R every? : ((Record(key: Symbol,entry: Any) -> Boolean),%) -> Boolean if $ has finiteAggregate
--R extract! : % -> Record(key: Symbol,entry: Any)
--R fill! : (% ,Any) -> % if $ has shallowlyMutable
--R find : ((Record(key: Symbol,entry: Any) -> Boolean),%) -> Union(Record(key: Symbol,entry: Any),"failed")
--R first : % -> Any if Symbol has ORDSET
--R getExplanations : (% ,String) -> List(String)
--R hash : % -> SingleInteger if Any has SETCAT or Record(key: Symbol,entry: Any) has SETCAT
--R insert! : (Record(key: Symbol,entry: Any),%) -> %
--R inspect : % -> Record(key: Symbol,entry: Any)
--R latex : % -> String if Any has SETCAT or Record(key: Symbol,entry: Any) has SETCAT
--R less? : (% ,NonNegativeInteger) -> Boolean
--R map : (((Any,Any) -> Any),%,%) -> %
--R map : ((Record(key: Symbol,entry: Any) -> Record(key: Symbol,entry: Any)),%) -> %
--R map! : ((Any -> Any),%) -> % if $ has shallowlyMutable
--R map! : ((Record(key: Symbol,entry: Any) -> Record(key: Symbol,entry: Any)),%) -> % if $ has shallowlyMutable
--R maxIndex : % -> Symbol if Symbol has ORDSET
--R member? : (Any,%) -> Boolean if $ has finiteAggregate and Any has SETCAT
--R member? : (Record(key: Symbol,entry: Any),%) -> Boolean if $ has finiteAggregate and Record(key: Symbol,entry: Any) has SETCAT
--R members : % -> List(Any) if $ has finiteAggregate
--R members : % -> List(Record(key: Symbol,entry: Any)) if $ has finiteAggregate
--R minIndex : % -> Symbol if Symbol has ORDSET
--R more? : (% ,NonNegativeInteger) -> Boolean
--R parts : % -> List(Any) if $ has finiteAggregate
--R parts : % -> List(Record(key: Symbol,entry: Any)) if $ has finiteAggregate
--R qsetelt! : (% ,Symbol,Any) -> Any if $ has shallowlyMutable
--R recoverAfterFail : (% ,String,Integer) -> Union(String,"failed")
--R reduce : (((Record(key: Symbol,entry: Any),Record(key: Symbol,entry: Any)) -> Record(key: Symbol,entry: Any)) -> Record(key: Symbol,entry: Any)) -> Record(key: Symbol,entry: Any)
--R reduce : (((Record(key: Symbol,entry: Any),Record(key: Symbol,entry: Any)) -> Record(key: Symbol,entry: Any)) -> Record(key: Symbol,entry: Any)) -> Record(key: Symbol,entry: Any)
--R reduce : (((Record(key: Symbol,entry: Any),Record(key: Symbol,entry: Any)) -> Record(key: Symbol,entry: Any)) -> Record(key: Symbol,entry: Any)) -> Record(key: Symbol,entry: Any)
--R remove : ((Record(key: Symbol,entry: Any) -> Boolean),%) -> % if $ has finiteAggregate
--R remove : (Record(key: Symbol,entry: Any),%) -> % if $ has finiteAggregate and Record(key: Symbol,entry: Any) has SETCAT
--R remove! : (Symbol,%) -> Union(Any,"failed")
--R remove! : ((Record(key: Symbol,entry: Any) -> Boolean),%) -> % if $ has finiteAggregate
--R remove! : (Record(key: Symbol,entry: Any),%) -> % if $ has finiteAggregate
--R removeDuplicates : % -> % if $ has finiteAggregate and Record(key: Symbol,entry: Any) has SETCAT
--R search : (Symbol,%) -> Union(Any,"failed")
--R select : ((Record(key: Symbol,entry: Any) -> Boolean),%) -> % if $ has finiteAggregate
--R select! : ((Record(key: Symbol,entry: Any) -> Boolean),%) -> % if $ has finiteAggregate
--R selectIntegrationRoutines : % -> %
--R selectMultiDimensionalRoutines : % -> %
--R selectOptimizationRoutines : % -> %
--R selectSumOfSquaresRoutines : % -> %
--R size? : (% ,NonNegativeInteger) -> Boolean
--R swap! : (% ,Symbol,Symbol) -> Void if $ has shallowlyMutable
--R table : List(Record(key: Symbol,entry: Any)) -> %
--R ~=? : (% ,%) -> Boolean if Any has SETCAT or Record(key: Symbol,entry: Any) has SETCAT

```

```
--R
--E 1

)spool
)lisp (bye)
```

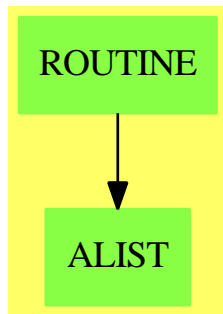
— **RoutinesTable.help** —

```
=====
RoutinesTable examples
=====
```

RoutinesTable implements a database and associated tuning mechanisms for a set of known NAG routines

See Also:
o)show RoutinesTable

RoutinesTable (ROUTINE)



Exports:

any?	bag
changeMeasure	changeThreshold
coerce	concat
construct	convert
copy	count
deleteRoutine!	dictionary
elt	empty
empty?	entries
entry?	eq?
eval	every?
extract!	fill!
find	first
getExplanations	getMeasure
hash	index?
indices	insert!
inspect	key?
keys	latex
less?	map
map!	maxIndex
member?	members
minIndex	more?
parts	qelt
qsetelt!	recoverAfterFail
reduce	remove
remove!	removeDuplicates
routines	sample
search	select
select!	selectFiniteRoutines
selectIntegrationRoutines	selectNonFiniteRoutines
selectMultiDimensionalRoutines	selectODEIVPRoutines
selectOptimizationRoutines	selectPDERoutines
selectSumOfSquaresRoutines	setelt
showTheRoutinesTable	size?
swap!	table
#?	?=?
?~=?	?..?

— domain ROUTINE RoutinesTable —

```

)abbrev domain ROUTINE RoutinesTable
++ Author: Brian Dupee
++ Date Created: August 1994
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{RoutinesTable} implements a database and associated tuning
++ mechanisms for a set of known NAG routines

```

```

RoutinesTable(): E == I where
  F ==> Float
  ST ==> String
  LST ==> List String
  Rec ==> Record(key:Symbol,entry:Any)
  RList ==> List(Record(key:Symbol,entry:Any))
  IFL ==> List(Record(iffail:Integer,instruction:ST))
  Entry ==> Record(chapter:ST, type:ST, domainName: ST,
                    defaultMin:F, measure:F, failList:IFL, explList:LST)

E ==> TableAggregate(Symbol,Any) with

concat:(%,%) -> %
  ++ concat(x,y) merges two tables x and y
routines:() -> %
  ++ routines() initialises a database of known NAG routines
selectIntegrationRoutines:% -> %
  ++ selectIntegrationRoutines(R) chooses only those routines from
  ++ the database which are for integration
selectOptimizationRoutines:% -> %
  ++ selectOptimizationRoutines(R) chooses only those routines from
  ++ the database which are for integration
selectPDERoutines:% -> %
  ++ selectPDERoutines(R) chooses only those routines from the
  ++ database which are for the solution of PDE's
selectODEIVPRoutines:% -> %
  ++ selectODEIVPRoutines(R) chooses only those routines from the
  ++ database which are for the solution of ODE's
selectFiniteRoutines:% -> %
  ++ selectFiniteRoutines(R) chooses only those routines from the
  ++ database which are designed for use with finite expressions
selectSumOfSquaresRoutines:% -> %
  ++ selectSumOfSquaresRoutines(R) chooses only those routines from the
  ++ database which are designed for use with sums of squares
selectNonFiniteRoutines:% -> %
  ++ selectNonFiniteRoutines(R) chooses only those routines from the
  ++ database which are designed for use with non-finite expressions.
selectMultiDimensionalRoutines:% -> %
  ++ selectMultiDimensionalRoutines(R) chooses only those routines from
  ++ the database which are designed for use with multi-dimensional
  ++ expressions
changeThreshhold:(%,Symbol,F) -> %
  ++ changeThreshhold(R,s,newValue) changes the value below which,
  ++ given a NAG routine generating a higher measure, the routines will
  ++ make no attempt to generate a measure.
changeMeasure:(%,Symbol,F) -> %
  ++ changeMeasure(R,s,newValue) changes the maximum value for a
  ++ measure of the given NAG routine.
getMeasure:(%,Symbol) -> F
  ++ getMeasure(R,s) gets the current value of the maximum measure for

```

```

    ++ the given NAG routine.
getExplanations:(%,ST) -> LST
    ++ getExplanations(R,s) gets the explanations of the output parameters for
    ++ the given NAG routine.
deleteRoutine!:(%,Symbol) -> %
    ++ deleteRoutine!(R,s) destructively deletes the given routine from
    ++ the current database of NAG routines
showTheRoutinesTable:() -> %
    ++ showTheRoutinesTable() returns the current table of NAG routines.
recoverAfterFail:(%,ST,Integer) -> Union(ST,"failed")
    ++ recoverAfterFail(routs,routineName,ifailValue) acts on the
    ++ instructions given by the ifail list
finiteAggregate

I ==> Result add

Rep := Result
import Rep

theRoutinesTable:% := routines()

showTheRoutinesTable():% == theRoutinesTable

integrationRoutine?(r:Record(key:Symbol,entry:Any)):Boolean ==
    (a := retractIfCan(r.entry)$AnyFunctions1(Entry)) case Entry =>
        elt(a,chapter) = "Integration"
    false

selectIntegrationRoutines(R:%):% == select(integrationRoutine?,R)

optimizationRoutine?(r:Record(key:Symbol,entry:Any)):Boolean ==
    (a := retractIfCan(r.entry)$AnyFunctions1(Entry)) case Entry =>
        elt(a,chapter) = "Optimization"
    false

selectOptimizationRoutines(R:%):% == select(optimizationRoutine?,R)

PDERoutine?(r:Record(key:Symbol,entry:Any)):Boolean ==
    (a := retractIfCan(r.entry)$AnyFunctions1(Entry)) case Entry =>
        elt(a,chapter) = "PDE"
    false

selectPDERoutines(R:%):% == select(PDERoutine?,R)

ODERoutine?(r:Record(key:Symbol,entry:Any)):Boolean ==
    (a := retractIfCan(r.entry)$AnyFunctions1(Entry)) case Entry =>
        elt(a,chapter) = "ODE"
    false

selectODEIVPRoutines(R:%):% == select(ODERoutine?,R)

```



```

sumOfSquaresRoutine?(r:Record(key:Symbol,entry:Any)):Boolean ==
  (a := retractIfCan(r.entry)$AnyFunctions1(Entry)) case Entry =>
    elt(a,type) = "SS"
  false

selectSumOfSquaresRoutines(R:%):% == select(sumOfSquaresRoutine?,R)

finiteRoutine?(r:Record(key:Symbol,entry:Any)):Boolean ==
  (a := retractIfCan(r.entry)$AnyFunctions1(Entry)) case Entry =>
    elt(a,type) = "One-dimensional finite"
  false

selectFiniteRoutines(R:%):% == select(finiteRoutine?,R)

infiniteRoutine?(r:Record(key:Symbol,entry:Any)):Boolean ==
  (a := retractIfCan(r.entry)$AnyFunctions1(Entry)) case Entry =>
    elt(a,type) = "One-dimensional infinite"
  false

semiInfiniteRoutine?(r:Record(key:Symbol,entry:Any)):Boolean ==
  (a := retractIfCan(r.entry)$AnyFunctions1(Entry)) case Entry =>
    elt(a,type) = "One-dimensional semi-infinite"
  false

nonFiniteRoutine?(r:Record(key:Symbol,entry:Any)):Boolean ==
  (semiInfiniteRoutine?(r) or infiniteRoutine?(r))

selectNonFiniteRoutines(R:%):% == select(nonFiniteRoutine?,R)

multiDimensionalRoutine?(r:Record(key:Symbol,entry:Any)):Boolean ==
  (a := retractIfCan(r.entry)$AnyFunctions1(Entry)) case Entry =>
    elt(a,type) = "Multi-dimensional"
  false

selectMultiDimensionalRoutines(R:%):% == select(multiDimensionalRoutine?,R)

concat(a:%,b:%):% ==
  membersOfa := (members(a)@List(Record(key:Symbol,entry:Any)))
  membersOfb := (members(b)@List(Record(key:Symbol,entry:Any)))
  allMembers:=
    concat(membersOfa,membersOfb)$List(Record(key:Symbol,entry:Any))
  construct(allMembers)

changeThreshhold(R:%,s:Symbol,newValue:F):% ==
  (a := search(s,R)) case Any =>
    e := retract(a)$AnyFunctions1(Entry)
    e.defaultMin := newValue
    a := coerce(e)$AnyFunctions1(Entry)
    insert!([s,a],R)

```

```

error("changeThreshold","Cannot find routine of that name")$ErrorFunctions

changeMeasure(R:%,s:Symbol,newValue:F):% ==
(a := search(s,R)) case Any =>
  e := retract(a)$AnyFunctions1(Entry)
  e.measure := newValue
  a := coerce(e)$AnyFunctions1(Entry)
  insert!([s,a],R)
error("changeMeasure","Cannot find routine of that name")$ErrorFunctions

getMeasure(R:%,s:Symbol):F ==
(a := search(s,R)) case Any =>
  e := retract(a)$AnyFunctions1(Entry)
  e.measure
error("getMeasure","Cannot find routine of that name")$ErrorFunctions

deleteRoutine!(R:%,s:Symbol):% ==
(a := search(s,R)) case Any =>
  e:Record(key:Symbol,entry:Any) := [s,a]
  remove!(e,R)
error("deleteRoutine!","Cannot find routine of that name")$ErrorFunctions

routines():% ==
f := "One-dimensional finite"
s := "One-dimensional semi-infinite"
i := "One-dimensional infinite"
m := "Multi-dimensional"
int := "Integration"
ode := "ODE"
pde := "PDE"
opt := "Optimization"
d01ajfExplList:LST := ["result: Calculated value of the integral",
  "iw: iw(1) contains the actual number of sub-intervals used, the rest is v",
  "w: contains the end-points of the sub-intervals used along with the inte",
  "abserr: the estimate of the absolute error of the result",
  "ifail: the error warning parameter",
  "method: details of the method used and measures of all methods",
  "attributes: a list of the attributes pertaining to the integrand wh",
d01asfExplList:LST := ["result: Calculated value of the integral",
  "iw: iw(1) contains the actual number of sub-intervals used, the rest is v",
  "lst: contains the actual number of sub-intervals used",
  "erlst: contains the error estimates over the sub-intervals",
  "rslst: contains the integral contributions of the sub-intervals",
  "ierlst: contains the error flags corresponding to the values in rslst",
  "abserr: the estimate of the absolute error of the result",
  "ifail: the error warning parameter",
  "method: details of the method used and measures of all methods",
  "attributes: a list of the attributes pertaining to the integrand wh",
d01fcfExplList:LST := ["result: Calculated value of the integral",
  "acc: the estimate of the relative error of the result",

```

```

"minpts: the number of integrand evaluations",
"ifail: the error warning parameter",
"method: details of the method used and measures of all met
"attributes: a list of the attributes pertaining to the in
d01transExplList:LST := ["result: Calculated value of the integral",
"abserr: the estimate of the absolute error of the result",
"method: details of the method and transformation used and
"d01***AnnaTypeAnswer: the individual results from the rou
"attributes: a list of the attributes pertaining to the i
d02bhfExplList:LST := ["x: the value of x at the end of the calculation",
"y: the computed values of Y\1\..Y\N\ at x",
"tol: the (possible) estimate of the error; this is not gua
"ifail: the error warning parameter",
"method: details of the method used and measures of all me
"intensityFunctions: a list of the attributes and values
d02bbfExplList:LST := concat(["result: the computed values of the solution at the require
d03eefExplList:LST := ["See the NAG On-line Documentation for D03EEF/D03EDF",
"u: the computed solution u[i][j] is returned in u(i+(j-1)*ng
e04fdfExplList:LST := ["x: the position of the minimum",
"objf: the value of the objective function at x",
"ifail: the error warning parameter",
"method: details of the method used and measures of all met
"attributes: a list of the attributes pertaining to the f
e04dggExplList:LST := concat(e04fdfExplList,
["objgrd: the values of the derivatives at x",
"iter: the number of iterations performed"])$LST
e04jafExplList:LST := concat(e04fdfExplList,
["bu: the values of the upper bounds used",
"bl: the values of the lower bounds used"])$LST
e04ucfExplList:LST := concat(e04dggExplList,
["istate: the status of every constraint at x",
"clamda: the QP multipliers for the last QP sub-probl
"For other output parameters see the NAG On-line Docu
e04mbfExplList:LST := concat(e04fdfExplList,
["istate: the status of every constraint at x",
"clamda: the Lagrange multipliers for each constraint
d01ajfIfail:IFL := [[1,"incrFunEvals"], [2,"delete"], [3,"delete"], [4,"delete"],
[5,"delete"], [6,"delete"]]
d01akfIfail:IFL := [[1,"incrFunEvals"], [2,"delete"], [3,"delete"], [4,"delete"]]
d01alfIfail:IFL := [[1,"incrFunEvals"], [2,"delete"], [3,"delete"], [4,"delete"],
[5,"delete"], [6,"delete"], [7,"delete"]]
d01amfIfail:IFL := [[1,"incrFunEvals"], [2,"delete"], [3,"delete"], [4,"delete"],
[5,"delete"], [6,"delete"]]
d01anfIfail:IFL := [[1,"incrFunEvals"], [2,"delete"], [3,"delete"], [4,"delete"],
[5,"delete"], [6,"delete"], [7,"delete"]]
d01apfIfail:IFL :=
[[1,"incrFunEvals"], [2,"delete"], [3,"delete"], [4,"delete"], [5,"delete"]]
d01aqfIfail:IFL :=
[[1,"incrFunEvals"], [2,"delete"], [3,"delete"], [4,"delete"], [5,"delete"]]
d01asfIfail:IFL := [[1,"incrFunEvals"], [2,"delete"], [3,"delete"], [4,"delete"],

```

```

    [5,"delete"], [6,"delete"], [7,"delete"], [8,"delete"], [9,"delete"]]
d01fcfIfail:IFL := [[1,"delete"], [2,"incrFunEvals"], [3,"delete"]]
d01gbfIfail:IFL := [[1,"delete"], [2,"incrFunEvals"]]
d02bbfIfail:IFL :=
    [[1,"delete"], [2,"decrease tolerance"], [3,"increase tolerance"],
    [4,"delete"], [5,"delete"], [6,"delete"], [7,"delete"]]
d02bhfIfail:IFL :=
    [[1,"delete"], [2,"decrease tolerance"], [3,"increase tolerance"],
    [4,"no action"], [5,"delete"], [6,"delete"], [7,"delete"]]
d02cjfIfail:IFL :=
    [[1,"delete"], [2,"decrease tolerance"], [3,"increase tolerance"],
    [4,"delete"], [5,"delete"], [6,"no action"], [7,"delete"]]
d02ejfIfail:IFL :=
    [[1,"delete"], [2,"decrease tolerance"], [3,"increase tolerance"],
    [4,"delete"], [5,"delete"], [6,"no action"], [7,"delete"], [8,"delete"],
    [9,"delete"]]
e04dgfIfail:IFL := [[3,"delete"], [4,"no action"], [6,"delete"],
    [7,"delete"], [8,"delete"], [9,"delete"]]
e04fdfIfail:IFL :=
    [[1,"delete"], [2,"delete"], [3,"delete"], [4,"delete"],
    [5,"no action"], [6,"no action"], [7,"delete"], [8,"delete"]]
e04gcfIfail:IFL := [[1,"delete"], [2,"delete"], [3,"delete"], [4,"delete"],
    [5,"no action"], [6,"no action"], [7,"delete"], [8,"delete"], [9,"delete"]]
e04jafIfail:IFL := [[1,"delete"], [2,"delete"], [3,"delete"], [4,"delete"],
    [5,"no action"], [6,"no action"], [7,"delete"], [8,"delete"], [9,"delete"]]
e04mbfIfail:IFL :=
    [[1,"delete"], [2,"delete"], [3,"delete"], [4,"delete"], [5,"delete"]]
e04nafIfail:IFL :=
    [[1,"delete"], [2,"delete"], [3,"delete"], [4,"delete"], [5,"delete"],
    [6,"delete"], [7,"delete"], [8,"delete"], [9,"delete"]]
e04ucfIfail:IFL := [[1,"delete"], [2,"delete"], [3,"delete"], [4,"delete"],
    [5,"delete"], [6,"delete"], [7,"delete"], [8,"delete"], [9,"delete"]]
d01ajfEntry:Entry := [int, f, "d01ajfAnnaType",0.4,0.4,d01ajfIfail,d01ajfExplList]
d01akfEntry:Entry := [int, f, "d01akfAnnaType",0.6,1.0,d01akfIfail,d01ajfExplList]
d01alfEntry:Entry := [int, f, "d01alfAnnaType",0.6,0.6,d01alfIfail,d01ajfExplList]
d01amfEntry:Entry := [int, i, "d01amfAnnaType",0.5,0.5,d01amfIfail,d01ajfExplList]
d01anfEntry:Entry := [int, f, "d01anfAnnaType",0.6,0.9,d01anfIfail,d01ajfExplList]
d01apfEntry:Entry := [int, f, "d01apfAnnaType",0.7,0.7,d01apfIfail,d01ajfExplList]
d01aqfEntry:Entry := [int, f, "d01aqfAnnaType",0.6,0.7,d01aqfIfail,d01ajfExplList]
d01asfEntry:Entry := [int, s, "d01asfAnnaType",0.6,0.9,d01asfIfail,d01asfExplList]
d01transEntry:Entry:= [int, i, "d01TransformFunctionType",0.6,0.9,[],d01transExplList]
d01gbfEntry:Entry := [int, m, "d01gbfAnnaType",0.6,0.6,d01gbfIfail,d01fcfExplList]
d01fcfEntry:Entry := [int, m, "d01fcfAnnaType",0.5,0.5,d01fcfIfail,d01fcfExplList]
d02bbfEntry:Entry := [ode, "IVP", "d02bbfAnnaType",0.7,0.5,d02bbfIfail,d02bbfExplList]
d02bhfEntry:Entry := [ode, "IVP", "d02bhfAnnaType",0.7,0.49,d02bhfIfail,d02bhfExplList]
d02cjfEntry:Entry := [ode, "IVP", "d02cjfAnnaType",0.7,0.5,d02cjfIfail,d02bbfExplList]
d02ejfEntry:Entry := [ode, "IVP", "d02ejfAnnaType",0.7,0.5,d02ejfIfail,d02bbfExplList]
d03eefEntry:Entry := [pde, "2", "d03eefAnnaType",0.6,0.5,[],d03eefExplList]
--d03fafEntry:Entry := [pde, "3", "d03fafAnnaType",0.6,0.5,[],[]]
e04dgfEntry:Entry := [opt, "CGA", "e04dgfAnnaType",0.4,0.4,e04dgfIfail,e04dgfExplList]

```

```

e04fdfEntry:Entry := [opt, "SS", "e04fdfAnnaType",0.7,0.7,e04fdfIfail,e04fdfExplList]
e04gcfEntry:Entry := [opt, "SS", "e04gcfAnnaType",0.8,0.8,e04gcfIfail,e04fdfExplList]
e04jafEntry:Entry := [opt, "QNA", "e04jafAnnaType",0.5,0.5,e04jafIfail,e04jafExplList]
e04mbfEntry:Entry := [opt, "LP", "e04mbfAnnaType",0.7,0.7,e04mbfIfail,e04mbfExplList]
e04nafEntry:Entry := [opt, "QP", "e04nafAnnaType",0.7,0.7,e04nafIfail,e04mbfExplList]
e04ucfEntry:Entry := [opt, "SQP", "e04ucfAnnaType",0.6,0.6,e04ucfIfail,e04ucfExplList]
rl:RList :=
  ["d01apf" :: Symbol, coerce(d01apfEntry)$AnyFunctions1(Entry)],_
  ["d01aqf" :: Symbol, coerce(d01aqfEntry)$AnyFunctions1(Entry)],_
  ["d01alf" :: Symbol, coerce(d01alfEntry)$AnyFunctions1(Entry)],_
  ["d01anf" :: Symbol, coerce(d01anfEntry)$AnyFunctions1(Entry)],_
  ["d01akf" :: Symbol, coerce(d01akfEntry)$AnyFunctions1(Entry)],_
  ["d01ajf" :: Symbol, coerce(d01ajfEntry)$AnyFunctions1(Entry)],_
  ["d01asf" :: Symbol, coerce(d01asfEntry)$AnyFunctions1(Entry)],_
  ["d01amf" :: Symbol, coerce(d01amfEntry)$AnyFunctions1(Entry)],_
  ["d01transform" :: Symbol, coerce(d01transEntry)$AnyFunctions1(Entry)],_
  ["d01gbf" :: Symbol, coerce(d01gbfEntry)$AnyFunctions1(Entry)],_
  ["d01fcf" :: Symbol, coerce(d01fcfEntry)$AnyFunctions1(Entry)],_
  ["d02bbf" :: Symbol, coerce(d02bbfEntry)$AnyFunctions1(Entry)],_
  ["d02bhf" :: Symbol, coerce(d02bhfEntry)$AnyFunctions1(Entry)],_
  ["d02cjf" :: Symbol, coerce(d02cjfEntry)$AnyFunctions1(Entry)],_
  ["d02ejf" :: Symbol, coerce(d02ejfEntry)$AnyFunctions1(Entry)],_
  ["d03eef" :: Symbol, coerce(d03eefEntry)$AnyFunctions1(Entry)],_
  --["d03faf" :: Symbol, coerce(d03fafEntry)$AnyFunctions1(Entry)],_
  ["e04dggf" :: Symbol, coerce(e04dggEntry)$AnyFunctions1(Entry)],_
  ["e04fdf" :: Symbol, coerce(e04fdfEntry)$AnyFunctions1(Entry)],_
  ["e04gcf" :: Symbol, coerce(e04gcfEntry)$AnyFunctions1(Entry)],_
  ["e04jaf" :: Symbol, coerce(e04jafEntry)$AnyFunctions1(Entry)],_
  ["e04mbf" :: Symbol, coerce(e04mbfEntry)$AnyFunctions1(Entry)],_
  ["e04naf" :: Symbol, coerce(e04nafEntry)$AnyFunctions1(Entry)],_
  ["e04ucf" :: Symbol, coerce(e04ucfEntry)$AnyFunctions1(Entry)]
construct(rl)

getIFL(s:Symbol,l:%):Union(IFL,"failed") ==
  o := search(s,l)$%
  o case "failed" => "failed"
  e := retractIfCan(o)$AnyFunctions1(Entry)
  e case "failed" => "failed"
  e.failList

getInstruction(l:IFL,ifailValue:Integer):Union(ST,"failed") ==
  output := empty()$ST
  for i in 1..#l repeat
    if ((l.i).ifail=ifailValue)@Boolean then
      output := (l.i).instruction
  empty?(output)$ST => "failed"
  output

recoverAfterFail(routs:%,routineName:ST,
  ifailValue:Integer):Union(ST,"failed") ==

```

```

name := routineName :: Symbol
failedList := getIFL(name,routs)
failedList case "failed" => "failed"
empty? failedList => "failed"
instr := getInstruction(failedList,ifailValue)
instr case "failed" => concat(routineName," failed")$ST
(instr = "delete")@Boolean =>
  deleteRoutine!(routs,name)
  concat(routineName," failed - trying alternatives")$ST
instr

getExplanations(R:%,routineName:ST):LST ==
name := routineName :: Symbol
(a := search(name,R)) case Any =>
  e := retract(a)$AnyFunctions1(Entry)
  e.explList
empty()$LST

```

— ROUTINE.dotabb —

```

"ROUTINE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ROUTINE"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ROUTINE" -> "ALIST"

```

domain RULECOLD RuleCalled

— RuleCalled.input —

```

)set break resume
)sys rm -f RuleCalled.output
)spool RuleCalled.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RuleCalled
--R
--R RuleCalled(f: Symbol) is a domain constructor
--R Abbreviation for RuleCalled is RULECOLD

```

```

--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for RULECOLD
--R
--R----- Operations -----
--R ?=? : (% ,%) -> Boolean          coerce : % -> OutputForm
--R hash : % -> SingleInteger        latex : % -> String
--R name : % -> Symbol              ?~=? : (% ,%) -> Boolean
--R
--E 1

)spool
)lisp (bye)

```

— RuleCalled.help —

=====

RuleCalled examples

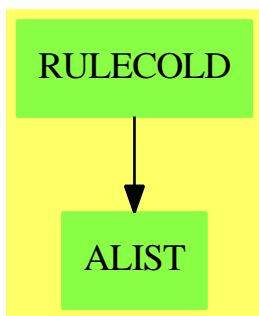
=====

This domain implements named rules

See Also:

- o)show RuleCalled

RuleCalled (RULECOLD)



Exports:

coerce hash latex name ?=? ?~=?

— domain RULECOLD RuleCalled —

```

)abbrev domain RULECOLD RuleCalled
++ Author: Mark Botch
++ Description:
++ This domain implements named rules

RuleCalled(f:Symbol): SetCategory with
    name: % -> Symbol
    ++ name(x) returns the symbol

== add
    name r == f
    coerce(r: %):OutputForm == f::OutputForm
    x = y == true
    latex(x: %):String == latex f

```

— RULECOLD.dotabb —

```

"RULECOLD" [color="#88FF44",href="bookvol10.3.pdf#nameddest=RULECOLD"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"RULECOLD" -> "ALIST"

```

domain RULESET Ruleset

— Ruleset.input —

```

)set break resume
)sys rm -f Ruleset.output
)spool Ruleset.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Ruleset
--R
--R Ruleset(Base: SetCategory,R: Join(Ring,PatternMatchable(Base),OrderedSet,ConvertibleTo(Pattern(Base))),
--R Abbreviation for Ruleset is RULESET
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for RULESET
--R
--R----- Operations -----

```



```

--R ?=? : (%,% ) -> Boolean               coerce : % -> OutputForm
--R elt : (% ,F,PositiveInteger) -> F      ?? : (% ,F) -> F
--R hash : % -> SingleInteger              latex : % -> String
--R ?~=? : (%,% ) -> Boolean
--R rules : % -> List(RewriteRule(Base,R,F))
--R ruleset : List(RewriteRule(Base,R,F)) -> %
--R
--E 1

```

```

)spool
)lisp (bye)

```

— Ruleset.help —

=====

Ruleset examples

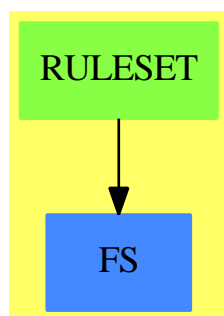
=====

Sets of rules for the pattern matcher. A ruleset is a set of pattern matching rules grouped together.

See Also:

o)show Ruleset

Ruleset (RULESET)



See

⇒ “RewriteRule” (RULE) [19](#) on page [2305](#)

Exports:

coerce	elt	hash	latex	rules
ruleset	?=?	?~=?	??	

— domain RULESET Ruleset —

```

)abbrev domain RULESET Ruleset
++ Author: Manuel Bronstein
++ Date Created: 20 Mar 1990
++ Date Last Updated: 29 Jun 1990
++ Description:
++ Sets of rules for the pattern matcher.
++ A ruleset is a set of pattern matching rules grouped together.

Ruleset(Base, R, F): Exports == Implementation where
  Base   : SetCategory
  R       : Join(Ring, PatternMatchable Base, OrderedSet,
                  ConvertibleTo Pattern Base)
  F       : Join(FunctionSpace R, PatternMatchable Base,
                  ConvertibleTo Pattern Base)

RR ==> RewriteRule(Base, R, F)

Exports ==> Join(SetCategory, Eltable(F, F)) with
  ruleset: List RR -> $
    ++ ruleset([r1,...,rn]) creates the rule set \spad{{r1,...,rn}}.
  rules  : $ -> List RR
    ++ rules(r) returns the rules contained in r.
  elt    : ($, F, PositiveInteger) -> F
    ++ elt(r,f,n) or r(f, n) applies all the rules of r to f at most n times.

Implementation ==> add
  import ApplyRules(Base, R, F)

  Rep := Set RR

  ruleset l                               == {l}$Rep
  coerce(x:$):OutputForm                  == coerce(x)$Rep
  x = y                                   == x =$Rep y
  elt(x:$, f:F)                           == applyRules(rules x, f)
  elt(r:$, s:F, n:PositiveInteger) == applyRules(rules r, s, n)
  rules x                                 == parts(x)$Rep

```

— RULESET.dotabb —

```

"RULESET" [color="#88FF44",href="bookvol10.3.pdf#nameddest=RULESET"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"RULESET" -> "FS"

```

Chapter 20

Chapter S

domain FORMULA ScriptFormulaFormat

— ScriptFormulaFormat.input —

```
)set break resume
)sys rm -f ScriptFormulaFormat.output
)spool ScriptFormulaFormat.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ScriptFormulaFormat
--R
--R ScriptFormulaFormat is a domain constructor
--R Abbreviation for ScriptFormulaFormat is FORMULA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for FORMULA
--R
--R----- Operations -----
--R ?? : (%,%) -> Boolean          coerce : OutputForm -> %
--R coerce : % -> OutputForm      display : % -> Void
--R display : (%,Integer) -> Void epilogue : % -> List(String)
--R formula : % -> List(String)   hash : % -> SingleInteger
--R latex : % -> String          new : () -> %
--R prologue : % -> List(String)  ~=? : (%,%) -> Boolean
--R convert : (OutputForm,Integer) -> %
--R setEpilogue! : (%,List(String)) -> List(String)
--R setFormula! : (%,List(String)) -> List(String)
--R setPrologue! : (%,List(String)) -> List(String)
--R
```

```
--E 1

)spool
)lisp (bye)
```

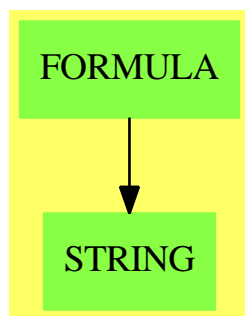
— ScriptFormulaFormat.help —

```
=====
ScriptFormulaFormat examples
=====
```

ScriptFormulaFormat provides a coercion from OutputForm} to IBM SCRIPT/VS Mathematical Formula Format. The basic SCRIPT formula format object consists of three parts: a prologue, a formula part and an epilogue. The functions prologue, formula and epilogue extract these parts, respectively. The central parts of the expression go into the formula part. The other parts can be set (setPrologue!, setEpilogue!) so that contain the appropriate tags for printing. For example, the prologue and epilogue might simply contain ":df." and ":edf." so that the formula section will be printed in display math mode.

See Also:
o)show ScriptFormulaFormat

ScriptFormulaFormat (FORMULA)



Exports:

coerce	display	epilogue	formula	hash
latex	new	prologue	convert	setEpilogue!
setFormula!	setPrologue!	?=?	?~=?	

— domain FORMULA ScriptFormulaFormat —

```

)abbrev domain FORMULA ScriptFormulaFormat
++ Author: Robert S. Sutor
++ Date Created: 1987 through 1990
++ References:
++   SCRIPT Mathematical Formula Formatter User's Guide, SH20-6453,
++   IBM Corporation, Publishing Systems Information Development,
++   Dept. G68, P.O. Box 1900, Boulder, Colorado, USA 80301-9191.
++ Description:
++ \spadtype{ScriptFormulaFormat} provides a coercion from
++ \spadtype{OutputForm} to IBM SCRIPT/VS Mathematical Formula Format.
++ The basic SCRIPT formula format object consists of three parts: a
++ prologue, a formula part and an epilogue. The functions
++ \spadfun{prologue}, \spadfun{formula} and \spadfun{epilogue}
++ extract these parts, respectively. The central parts of the expression
++ go into the formula part. The other parts can be set
++ (\spadfun{setPrologue!}, \spadfun{setEpilogue!}) so that contain the
++ appropriate tags for printing. For example, the prologue and
++ epilogue might simply contain ":df." and ":edf." so that the
++ formula section will be printed in display math mode.

ScriptFormulaFormat(): public == private where
  E      ==> OutputForm
  I      ==> Integer
  L      ==> List
  S      ==> String

public == SetCategory with
  coerce: E -> %
    ++ coerce(o) changes o in the standard output format to
    ++ SCRIPT formula format.
  convert: (E,I) -> %
    ++ convert(o,step) changes o in standard output format to
    ++ SCRIPT formula format and also adds the given step number.
    ++ This is useful if you want to create equations with given numbers
    ++ or have the equation numbers correspond to the interpreter step
    ++ numbers.
  display: (% , I) -> Void
    ++ display(t,width) outputs the formatted code t so that each
    ++ line has length less than or equal to \spadvar{width}.
  display: % -> Void
    ++ display(t) outputs the formatted code t so that each
    ++ line has length less than or equal to the value set by
    ++ the system command \spadsyscom{set output length}.
  epilogue: % -> L S
    ++ epilogue(t) extracts the epilogue section of a formatted object t.
  formula: % -> L S
    ++ formula(t) extracts the formula section of a formatted object t.

```

```

new:      () -> %
  ++ new() create a new, empty object. Use \spadfun{setPrologue!},
  ++ \spadfun{setFormula!} and \spadfun{setEpilogue!} to set the
  ++ various components of this object.
prologue: % -> L S
  ++ prologue(t) extracts the prologue section of a formatted object t.
setEpilogue!: (% , L S) -> L S
  ++ setEpilogue!(t,strings) sets the epilogue section of a
  ++ formatted object t to strings.
setFormula!: (% , L S) -> L S
  ++ setFormula!(t,strings) sets the formula section of a
  ++ formatted object t to strings.
setPrologue!: (% , L S) -> L S
  ++ setPrologue!(t,strings) sets the prologue section of a
  ++ formatted object t to strings.

private == add
import OutputForm
import Character
import Integer
import List OutputForm
import List String

Rep := Record(prolog : L S, formula : L S, epilogs : L S)

-- local variables declarations and definitions

expr: E
prec,opPrec: I
str: S
blank      : S := " @@ "

maxPrec      : I      := 1000000
minPrec      : I      := 0

splitChars   : S      := " <>[](){}+*=-%"

unaryOps     : L S := ["-", "^"]$(L S)
unaryPrecs   : L I := [700,260]$(L I)

-- the precedence of / in the following is relatively low because
-- the bar obviates the need for parentheses.
binaryOps    : L S := ["+>", "|", "**", "/", "<", ">", "=", "OVER"]$(L S)
binaryPrecs  : L I := [0,0,900, 700,400,400,400, 700]$(L I)

naryOps      : L S := ["-", "+", "*", blank, ",", ";", " ", "ROW", "",
  " habove ", " here ", " labove "]$(L S)
naryPrecs    : L I := [700,700,800, 800,110,110, 0, 0, 0,
  0, 0, 0]$(L I)
-- naryNGOps  : L S := ["ROW", " here "]$(L S)

```

```

naryNGOps      : L S := nil$(L S)

plexOps        : L S := ["SIGMA","PI","INTSIGN","INDEFINTEGRAL"]$(L S)
plexPrecs      : L I := [ 700, 800, 700, 700]$(L I)

specialOps     : L S := ["MATRIX","BRACKET","BRACE","CONCATB", _
                        "AGGLST","CONCAT","OVERBAR","ROOT","SUB", _
                        "SUPERSUB","ZAG","AGGSET","SC","PAREN"]

-- the next two lists provide translations for some strings for
-- which the formula formatter provides special variables.

specialStrings : L S :=
  ["5","..."]
specialStringsInFormula : L S :=
  [" alpha "," ellipsis "]

-- local function signatures

addBraces:      S -> S
addBrackets:    S -> S
group:          S -> S
formatBinary:   (S,L E, I) -> S
formatFunction: (S,L E, I) -> S
formatMatrix:   L E -> S
formatNary:     (S,L E, I) -> S
formatNaryNoGroup: (S,L E, I) -> S
formatNullary:  S -> S
formatPlex:     (S,L E, I) -> S
formatSpecial:  (S,L E, I) -> S
formatUnary:    (S, E, I) -> S
formatFormula:  (E,I) -> S
parenthesize:   S -> S
precondition:   E -> E
postcondition:  S -> S
splitLong:      (S,I) -> L S
splitLong1:     (S,I) -> L S
stringify:      E -> S

-- public function definitions

new() : % == [".eq set blank @",".df." ]$(L S),
  ["" ]$(L S), [".edf." ]$(L S)]$Rep

coerce(expr : E): % ==
  f : % := new()$%
  f.formula := [postcondition
    formatFormula(precondition expr, minPrec)]$(L S)
  f

```



```

convert(expr : E, stepNum : I): % ==
  f : % := new()$%
  f.formula := concat(["<leqno lparen ",string(stepNum)$S,
    " rparen>"], [postcondition
    formatFormula(precondition expr, minPrec)]$(L S))
  f

display(f : %, len : I) ==
  s,t : S
  for s in f.prolog repeat sayFORMULA(s)$Lisp
  for s in f.formula repeat
    for t in splitLong(s, len) repeat sayFORMULA(t)$Lisp
  for s in f.epilog repeat sayFORMULA(s)$Lisp
  void()$Void

display(f : %) ==
  display(f, _$LINELENGTH$Lisp pretend I)

prologue(f : %) == f.prolog
formula(f : %) == f.formula
epilogue(f : %) == f.epilog

setPrologue!(f : %, l : L S) == f.prolog := l
setFormula!(f : %, l : L S) == f.formula := l
setEpilogue!(f : %, l : L S) == f.epilog := l

coerce(f : %): E ==
  s,t : S
  l : L S := nil
  for s in f.prolog repeat l := concat(s,l)
  for s in f.formula repeat
    for t in splitLong(s, (_$LINELENGTH$Lisp pretend Integer) - 4) repeat
      l := concat(t,l)
  for s in f.epilog repeat l := concat(s,l)
  (reverse l) :: E

-- local function definitions

postcondition(str: S): S ==
  len : I := #str
  len < 4 => str
  plus : Character := char "+"
  minus: Character := char "-"
  for i in 1..(len-1) repeat
    if (str.i =$Character plus) and (str.(i+1) =$Character minus)
      then setelt(str,i,char " ")$S
  str

stringify expr == object2String(expr)$Lisp pretend S

```

```

splitLong(str : S, len : I): L S ==
  -- this blocks into lines
  if len < 20 then len := _$LINELENGTH$Lisp
  splitLong1(str, len)

splitLong1(str : S, len : I) ==
  l : List S := nil
  s : S := ""
  ls : I := 0
  ss : S
  lss : I
  for ss in split(str,char " ") repeat
    lss := #ss
    if ls + lss > len then
      l := concat(s,l)$List(S)
      s := ""
      ls := 0
    lss > len => l := concat(ss,l)$List(S)
    ls := ls + lss + 1
    s := concat(s,concat(ss," ")$S)$S
  if ls > 0 then l := concat(s,l)$List(S)
  reverse l

group str ==
  concat ["<",str,">"]

addBraces str ==
  concat ["left lbrace ",str," right rbrace"]

addBrackets str ==
  concat ["left lb ",str," right rb"]

parenthesize str ==
  concat ["left lparen ",str," right rparen"]

precondition expr ==
  outputTran(expr)$Lisp

formatSpecial(op : S, args : L E, prec : I) : S ==
  op = "AGGLST" =>
    formatNary(",",args,prec)
  op = "AGGSET" =>
    formatNary(";",args,prec)
  op = "CONCATB" =>
    formatNary(" ",args,prec)
  op = "CONCAT" =>
    formatNary("",args,prec)
  op = "BRACKET" =>
    group addBrackets formatFormula(first args, minPrec)
  op = "BRACE" =>

```

```

    group addBraces formatFormula(first args, minPrec)
op = "PAREN" =>
    group parenthesize formatFormula(first args, minPrec)
op = "OVERBAR" =>
    null args => ""
    group concat [formatFormula(first args, minPrec), " bar"]
op = "ROOT" =>
    null args => ""
    tmp : S := formatFormula(first args, minPrec)
    null rest args => group concat ["sqrt ", tmp]
    group concat ["midsup adjust(u 1.5 r 9) ",
        formatFormula(first rest args, minPrec), " sqrt ", tmp]
op = "SC" =>
    formatNary(" labove ", args, prec)
op = "SUB" =>
    group concat [formatFormula(first args, minPrec), " sub ",
        formatSpecial("AGGLST", rest args, minPrec)]
op = "SUPERSUB" =>
    -- variable name
    form : List S := [formatFormula(first args, minPrec)]
    -- subscripts
    args := rest args
    null args => concat form
    tmp : S := formatFormula(first args, minPrec)
    if tmp ^= "" then form := append(form, [" sub ", tmp])$(List S)
    -- superscripts
    args := rest args
    null args => group concat form
    tmp : S := formatFormula(first args, minPrec)
    if tmp ^= "" then form := append(form, [" sup ", tmp])$(List S)
    -- presuperscripts
    args := rest args
    null args => group concat form
    tmp : S := formatFormula(first args, minPrec)
    if tmp ^= "" then form := append(form, [" presup ", tmp])$(List S)
    -- presubscripts
    args := rest args
    null args => group concat form
    tmp : S := formatFormula(first args, minPrec)
    if tmp ^= "" then form := append(form, [" presub ", tmp])$(List S)
    group concat form
op = "MATRIX" => formatMatrix rest args
-- op = "ZAG" =>
--     concat ["\zag{", formatFormula(first args, minPrec), "}{",
--         formatFormula(first rest args, minPrec), "}"]
--     concat ["not done yet for ", op]

formatPlex(op : S, args : L E, prec : I) : S ==
    hold : S
    p : I := position(op, plexOps)

```

```

p < 1 => error "unknown Script Formula Formatter unary op"
opPrec := plexPrecs.p
n : I := #args
(n ^= 2) and (n ^= 3) => error "wrong number of arguments for plex"
s : S :=
  op = "SIGMA"    => "sum"
  op = "PI"       => "product"
  op = "INTSIGN"  => "integral"
  op = "INDEFINTEGRAL" => "integral"
  "???"
hold := formatFormula(first args,minPrec)
args := rest args
if op ^= "INDEFINTEGRAL" then
  if hold ^= "" then
    s := concat [s," from",group concat ["\displaystyle ",hold]]
  if not null rest args then
    hold := formatFormula(first args,minPrec)
    if hold ^= "" then
      s := concat [s," to",group concat ["\displaystyle ",hold]]
    args := rest args
    s := concat [s," ",formatFormula(first args,minPrec)]
  else
    hold := group concat [hold," ",formatFormula(first args,minPrec)]
    s := concat [s," ",hold]
  if opPrec < prec then s := parenthesize s
  group s

formatMatrix(args : L E) : S ==
  -- format for args is [[ROW ...],[ROW ...],[ROW ...]]
  group addBrackets formatNary(" habove ",args,minPrec)

formatFunction(op : S, args : L E, prec : I) : S ==
  group concat [op, " ", parenthesize formatNary(" ",args,minPrec)]

formatNullary(op : S) ==
  op = "NOTHING" => ""
  group concat [op,"()"]

formatUnary(op : S, arg : E, prec : I) ==
  p : I := position(op,unaryOps)
  p < 1 => error "unknown Script Formula Formatter unary op"
  opPrec := unaryPrecs.p
  s : S := concat [op,formatFormula(arg,opPrec)]
  opPrec < prec => group parenthesize s
  op = "-" => s
  group s

formatBinary(op : S, args : L E, prec : I) : S ==
  p : I := position(op,binaryOps)
  p < 1 => error "unknown Script Formula Formatter binary op"

```

```

op :=
  op = "**"    => " sup "
  op = "/"    => " over "
  op = "OVER" => " over "
  op
opPrec := binaryPrecs.p
s : S := formatFormula(first args, opPrec)
s := concat [s,op,formatFormula(first rest args, opPrec)]
group
  op = " over " => s
  opPrec < prec => parenthesize s
  s

formatNary(op : S, args : L E, prec : I) : S ==
  group formatNaryNoGroup(op, args, prec)

formatNaryNoGroup(op : S, args : L E, prec : I) : S ==
  null args => ""
  p : I := position(op,naryOps)
  p < 1 => error "unknown Script Formula Formatter nary op"
  op :=
    op = ","    => ", @@ "
    op = ";"    => "; @@ "
    op = "*"    => blank
    op = " "    => blank
    op = "ROW"  => " here "
    op
  l : L S := nil
  opPrec := naryPrecs.p
  for a in args repeat
    l := concat(op,concat(formatFormula(a,opPrec),l)$L(S))$L(S)
  s : S := concat reverse rest l
  opPrec < prec => parenthesize s
  s

formatFormula(expr,prec) ==
  i : Integer
  ATOM(expr)$Lisp pretend Boolean =>
    str := stringify expr
    INTEGERP(expr)$Lisp =>
      i := expr : Integer
      if (i < 0) or (i > 9) then group str else str
      (i := position(str,specialStrings)) > 0 =>
        specialStringsInFormula.i
      str
  l : L E := (expr pretend L E)
  null l => blank
  op : S := stringify first l
  args : L E := rest l
  nargs : I := #args

```

```

-- special cases
member?(op, specialOps) => formatSpecial(op,args,prec)
member?(op, plexOps)    => formatPlex(op,args,prec)

-- nullary case
0 = nargs => formatNullary op

-- unary case
(1 = nargs) and member?(op, unaryOps) =>
  formatUnary(op, first args, prec)

-- binary case
(2 = nargs) and member?(op, binaryOps) =>
  formatBinary(op, args, prec)

-- nary case
member?(op,naryNGOps) => formatNaryNoGroup(op,args, prec)
member?(op,naryOps)   => formatNary(op,args, prec)
op := formatFormula(first l,minPrec)
formatFunction(op,args,prec)

```

— FORMULA.dotabb —

```

"FORMULA" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FORMULA"]
"STRING"  [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"FORMULA" -> "STRING"

```

domain SEG Segment

— Segment.input —

```

)set break resume
)sys rm -f Segment.output
)spool Segment.output
)set message test on
)set message auto off
)clear all
--S 1 of 10
s := 3..10

```

```

--R
--R
--R (1) 3..10
--R
--R                                         Type: Segment(PositiveInteger)
--E 1

--S 2 of 10
lo s
--R
--R
--R (2) 3
--R
--R                                         Type: PositiveInteger
--E 2

--S 3 of 10
hi s
--R
--R
--R (3) 10
--R
--R                                         Type: PositiveInteger
--E 3

--S 4 of 10
t := 10..3 by -2
--R
--R
--R (4) 10..3 by - 2
--R
--R                                         Type: Segment(PositiveInteger)
--E 4

--S 5 of 10
incr s
--R
--R
--R (5) 1
--R
--R                                         Type: PositiveInteger
--E 5

--S 6 of 10
incr t
--R
--R
--R (6) - 2
--R
--R                                         Type: Integer
--E 6

--S 7 of 10
l := [1..3, 5, 9, 15..11 by -1]
--R
--R

```

```

--R (7) [1..3,5..5,9..9,15..11 by - 1]
--R                                         Type: List(Segment(PositiveInteger))
--E 7

--S 8 of 10
expand s
--R
--R
--R (8) [3,4,5,6,7,8,9,10]
--R                                         Type: List(Integer)
--E 8

--S 9 of 10
expand t
--R
--R
--R (9) [10,8,6,4]
--R                                         Type: List(Integer)
--E 9

--S 10 of 10
expand l
--R
--R
--R (10) [1,2,3,5,9,15,14,13,12,11]
--R                                         Type: List(Integer)
--E 10
)spool
)lisp (bye)

```

— Segment.help —

=====

Segment examples

=====

The Segment domain provides a generalized interval type.

Segments are created using the .. construct by indicating the (included) end points.

```

s := 3..10
3..10
                                         Type: Segment PositiveInteger

```

The first end point is called the lo and the second is called hi.


```
lo s
  3
```

Type: PositiveInteger

These names are used even though the end points might belong to an unordered set.

```
hi s
  10
```

Type: PositiveInteger

In addition to the end points, each segment has an integer "increment". An increment can be specified using the "by" construct.

```
t := 10..3 by -2
    10..3 by - 2
```

Type: Segment PositiveInteger

This part can be obtained using the incr function.

```
incr s
  1
```

Type: PositiveInteger

Unless otherwise specified, the increment is 1.

```
incr t
  - 2
```

Type: Integer

A single value can be converted to a segment with equal end points. This happens if segments and single values are mixed in a list.

```
l := [1..3, 5, 9, 15..11 by -1]
     [1..3,5..5,9..9,15..11 by - 1]
```

Type: List Segment PositiveInteger

If the underlying type is an ordered ring, it is possible to perform additional operations. The expand operation creates a list of points in a segment.

```
expand s
  [3,4,5,6,7,8,9,10]
```

Type: List Integer

If $k > 0$, then `expand(l..h by k)` creates the list `[l, l+k, ..., lN]` where $lN \leq h < lN+k$. If $k < 0$, then $lN \geq h > lN+k$.

```
expand t
  [10,8,6,4]
```

Type: List Integer

It is also possible to expand a list of segments. This is equivalent to appending lists obtained by expanding each segment individually.

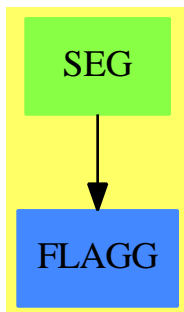
```
expand 1
  [1,2,3,5,9,15,14,13,12,11]
```

Type: List Integer

See Also:

- o)help UniversalSegment
- o)help SegmentBinding
- o)show Segment

Segment (SEG)



See

⇒ “SegmentBinding” (SEGBIND) [20](#) on page [2364](#)

⇒ “UniversalSegment” (UNISEG) [22](#) on page [2896](#)

Exports:

BY	coerce	convert	expand	hash
hi	high	incr	latex	lo
low	map	segment	?=?	?~=?
?...?				

— domain SEG Segment —

)abbrev domain SEG Segment

++ Author: Stephen M. Watt

++ Date Created: December 1986

++ Date Last Updated: June 3, 1991

```

++ Description:
++ This type is used to specify a range of values from type \spad{S}.

Segment(S:Type): SegmentCategory(S) with
  if S has SetCategory then SetCategory
  if S has OrderedRing then SegmentExpansionCategory(S, List S)
== add

Rep := Record(low: S, high: S, incr: Integer)

a..b == [a,b,1]
lo s == s.low
low s == s.low
hi s == s.high
high s == s.high
incr s == s.incr
segment(a,b) == [a,b,1]
BY(s, r) == [lo s, hi s, r]

if S has SetCategory then
  (s1:%) = (s2:%) ==
    s1.low = s2.low and s1.high=s2.high and s1.incr = s2.incr

coerce(s:%):OutputForm ==
  seg := SEGMENT(s.low::OutputForm, s.high::OutputForm)
  s.incr = 1 => seg
  infix(" by " ::OutputForm, seg, s.incr::OutputForm)

convert a == [a,a,1]

if S has OrderedRing then
  expand(ls: List %):List S ==
    lr := nil()$List(S)
    for s in ls repeat
      l := lo s
      h := hi s
      inc := (incr s)::S
      zero? inc => error "Cannot expand a segment with an increment of zero"
      if inc > 0 then
        while l <= h repeat
          lr := concat(l, lr)
          l := l + inc
      else
        while l >= h repeat
          lr := concat(l, lr)
          l := l + inc
    reverse_! lr

expand(s : %) == expand([s]$List(%))$%
map(f : S->S, s : %): List S ==

```

```

lr := nil()$List(S)
l := lo s
h := hi s
inc := (incr s)::S
if inc > 0 then
  while l <= h repeat
    lr := concat(f l, lr)
    l := l + inc
else
  while l >= h repeat
    lr := concat(f l, lr)
    l := l + inc
reverse_! lr

```

— SEG.dotabb —

```

"SEG" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SEG"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"SEG" -> "FLAGG"

```

domain SEGBIND SegmentBinding

— SegmentBinding.input —

```

)set break resume
)sys rm -f SegmentBinding.output
)spool SegmentBinding.output
)set message test on
)set message auto off
)clear all
--S 1 of 5
x = a..b
--R
--R
--R (1) x= a..b
--R
--E 1

```

Type: SegmentBinding Symbol

```

--S 2 of 5
sum(i**2, i = 0..n)

```

```

--R
--R
--R      3      2
--R      2n  + 3n  + n
--R  (2)  -----
--R           6
--R
--R                                          Type: Fraction Polynomial Integer
--E 2

--S 3 of 5
sb := y = 1/2..3/2
--R
--R
--R      1      3
--R  (3)  y= (-)..(-)
--R      2      2
--R
--R                                          Type: SegmentBinding Fraction Integer
--E 3

--S 4 of 5
variable(sb)
--R
--R
--R  (4)  y
--R
--R                                          Type: Symbol
--E 4

--S 5 of 5
segment(sb)
--R
--R
--R      1      3
--R  (5)  (-)..(-)
--R      2      2
--R
--R                                          Type: Segment Fraction Integer
--E 5
)spool
)lisp (bye)

```

— SegmentBinding.help —

```

=====
SegmentBinding examples
=====

```

The SegmentBinding type is used to indicate a range for a named symbol.

First give the symbol, then an = and finally a segment of values.

```
x = a..b
x= a..b
```

Type: SegmentBinding Symbol

This is used to provide a convenient syntax for arguments to certain operations.

```
sum(i**2, i = 0..n)
      3      2
    2n  + 3n  + n
    -----
           6
```

Type: Fraction Polynomial Integer

```
draw(x**2, x = -2..2)
TwoDimensionalViewport: "x*x"
```

Type: TwoDimensionalViewport

The left-hand side must be of type Symbol but the right-hand side can be a segment over any type.

```
sb := y = 1/2..3/2
      1      3
    y= (-)..(-)
      2      2
```

Type: SegmentBinding Fraction Integer

The left- and right-hand sides can be obtained using the variable and segment operations.

```
variable(sb)
y
```

Type: Symbol

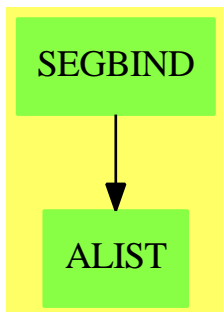
```
segment(sb)
      1      3
    (-)..(-)
      2      2
```

Type: Segment Fraction Integer

See Also:

- o)help Segment
- o)help UniversalSegment
- o)show SegmentBinding

SegmentBinding (SEGBIND)



See

⇒ “Segment” (SEG) [20](#) on page [2359](#)

⇒ “UniversalSegment” (UNISEG) [22](#) on page [2896](#)

Exports:

coerce	equation	hash	latex	segment
variable	?=?	?~=?		

— domain SEGBIND SegmentBinding —

```

)abbrev domain SEGBIND SegmentBinding
++ Author: Mark Botch
++ Date Last Updated: June 4, 1991
++ Description:
++ This domain is used to provide the function argument syntax \spad{v=a..b}.
++ This is used, for example, by the top-level \spadfun{draw} functions.

```

```

SegmentBinding(S:Type): Type with
  equation: (Symbol, Segment S) -> %
    ++ equation(v,a..b) creates a segment binding value with variable
    ++ \spad{v} and segment \spad{a..b}. Note that the interpreter parses
    ++ \spad{v=a..b} to this form.
  variable: % -> Symbol
    ++ variable(segb) returns the variable from the left hand side of
    ++ the \spadtype{SegmentBinding}. For example, if \spad{segb} is
    ++ \spad{v=a..b}, then \spad{variable(segb)} returns \spad{v}.
  segment : % -> Segment S
    ++ segment(segb) returns the segment from the right hand side of
    ++ the \spadtype{SegmentBinding}. For example, if \spad{segb} is
    ++ \spad{v=a..b}, then \spad{segment(segb)} returns \spad{a..b}.

  if S has SetCategory then SetCategory
== add

```

```

Rep := Record(var:Symbol, seg:Segment S)
equation(x,s) == [x, s]
variable b    == b.var
segment b     == b.seg

if S has SetCategory then

  b1 = b2      == variable b1 = variable b2 and segment b1 = segment b2

  coerce(b:%):OutputForm ==
    variable(b)::OutputForm = segment(b)::OutputForm

```

— SEGBIND.dotabb —

```

"SEGBIND" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SEGBIND"]
"ALIST"   [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"SEGBIND" -> "ALIST"

```

domain SET Set

— Set.input —

```

)set break resume
)sys rm -f Set.output
)spool Set.output
)set message test on
)set message auto off
)clear all
--S 1 of 20
s := set [x**2-1, y**2-1, z**2-1]
--R
--R
--R      2      2      2
--R   (1) {x  - 1, y  - 1, z  - 1}
--R
--R                                          Type: Set(Polynomial(Integer))
--E 1

--S 2 of 20
t := set [x**i - i+1 for i in 2..10 | prime? i]
--R

```


Type: Set(Polynomial(Integer))

Type: Set(Polynomial(Integer))

Type: Set(Polynomial(Integer))

Type: Set(Polynomial(Integer))

Type: Set(Polynomial(Integer))

Type: Boolean

```
--S 8 of 20
member?((y+1)*(y-1), s)
```

```
--R
```

```
--R
```

```
--R (8) true
```

```
--R
```

Type: Boolean

```
--E 8
```

```
--S 9 of 20
```

```
subset?(i, s)
```

```
--R
```

```
--R
```

```
--R (9) true
```

```
--R
```

Type: Boolean

```
--E 9
```

```
--S 10 of 20
```

```
subset?(u, s)
```

```
--R
```

```
--R
```

```
--R (10) false
```

```
--R
```

Type: Boolean

```
--E 10
```

```
--S 11 of 20
```

```
gs := set [g for i in 1..11 | primitive?(g := i::PF 11)]
```

```
--R
```

```
--R
```

```
--R (11) {2,6,7,8}
```

```
--R
```

Type: Set(PrimeField(11))

```
--E 11
```

```
--S 12 of 20
```

```
complement gs
```

```
--R
```

```
--R
```

```
--R (12) {1,3,4,5,9,10,0}
```

```
--R
```

Type: Set(PrimeField(11))

```
--E 12
```

```
--S 13 of 20
```

```
a := set [i**2 for i in 1..5]
```

```
--R
```

```
--R
```

```
--R (13) {1,4,9,16,25}
```

```
--R
```

Type: Set(PositiveInteger)

```
--E 13
```

```
--S 14 of 20
```

```
insert!(32, a)
```

```

--R
--R
--R (14) {1,4,9,16,25,32}
--R
--R                                         Type: Set(PositiveInteger)
--E 14

--S 15 of 20
remove!(25, a)
--R
--R
--R (15) {1,4,9,16,32}
--R
--R                                         Type: Set(PositiveInteger)
--E 15

--S 16 of 20
a
--R
--R
--R (16) {1,4,9,16,32}
--R
--R                                         Type: Set(PositiveInteger)
--E 16

--S 17 of 20
b := b0 := set [i**2 for i in 1..5]
--R
--R
--R (17) {1,4,9,16,25}
--R
--R                                         Type: Set(PositiveInteger)
--E 17

--S 18 of 20
b := union(b, {32})
--R
--R
--R (18) {1,4,9,16,25,32}
--R
--R                                         Type: Set(PositiveInteger)
--E 18

--S 19 of 20
b := difference(b, {25})
--R
--R
--R (19) {1,4,9,16,32}
--R
--R                                         Type: Set(PositiveInteger)
--E 19

--S 20 of 20
b0
--R
--R

```

```

--R (20) {1,4,9,16,25}
--R                                         Type: Set(PositiveInteger)
--E 20
)spool
)lisp (bye)

```

— Set.help —

=====

Set examples

=====

The Set domain allows one to represent explicit finite sets of values. These are similar to lists, but duplicate elements are not allowed.

Sets can be created by giving a fixed set of values ...

```

s := set [x**2-1, y**2-1, z**2-1]
      2      2      2
      {x  - 1, y  - 1, z  - 1}
                                         Type: Set Polynomial Integer

```

or by using a collect form, just as for lists. In either case, the set is formed from a finite collection of values.

```

t := set [x**i - i+1 for i in 2..10 | prime? i]
      2      3      5      7
      {x  - 1, x  - 2, x  - 4, x  - 6}
                                         Type: Set Polynomial Integer

```

The basic operations on sets are intersect, union, difference, and symmetricDifference.

```

i := intersect(s,t)
      2
      {x  - 1}
                                         Type: Set Polynomial Integer

```

```

u := union(s,t)
      2      3      5      7      2      2
      {x  - 1, x  - 2, x  - 4, x  - 6, y  - 1, z  - 1}
                                         Type: Set Polynomial Integer

```

The set difference(s,t) contains those members of s which are not in t.

```

difference(s,t)
      2      2

```

```
{y - 1, z - 1}
```

```
Type: Set Polynomial Integer
```

The set `symmetricDifference(s,t)` contains those elements which are in `s` or `t` but not in both.

```
symmetricDifference(s,t)
```

```
      3      5      7      2      2
{x - 2, x - 4, x - 6, y - 1, z - 1}
```

```
Type: Set Polynomial Integer
```

Set membership is tested using the `member?` operation.

```
member?(y, s)
```

```
false
```

```
Type: Boolean
```

```
member?((y+1)*(y-1), s)
```

```
true
```

```
Type: Boolean
```

The `subset?` function determines whether one set is a subset of another.

```
subset?(i, s)
```

```
true
```

```
Type: Boolean
```

```
subset?(u, s)
```

```
false
```

```
Type: Boolean
```

When the base type is finite, the absolute complement of a set is defined. This finds the set of all multiplicative generators of `PrimeField 11`---the integers mod 11.

```
gs := set [g for i in 1..11 | primitive?(g := i::PF 11)]
{2,6,7,8}
```

```
Type: Set PrimeField 11
```

The following values are not generators.

```
complement gs
```

```
{1,3,4,5,9,10,0}
```

```
Type: Set PrimeField 11
```

Often the members of a set are computed individually; in addition, values can be inserted or removed from a set over the course of a computation.

There are two ways to do this:

```
a := set [i**2 for i in 1..5]
      {1,4,9,16,25}
                        Type: Set PositiveInteger
```

One is to view a set as a data structure and to apply updating operations.

```
insert!(32, a)
      {1,4,9,16,25,32}
                        Type: Set PositiveInteger
```

```
remove!(25, a)
      {1,4,9,16,32}
                        Type: Set PositiveInteger
```

```
a
      {1,4,9,16,32}
                        Type: Set PositiveInteger
```

The other way is to view a set as a mathematical entity and to create new sets from old.

```
b := b0 := set [i**2 for i in 1..5]
      {1,4,9,16,25}
                        Type: Set PositiveInteger
```

```
b := union(b, {32})
      {1,4,9,16,25,32}
                        Type: Set PositiveInteger
```

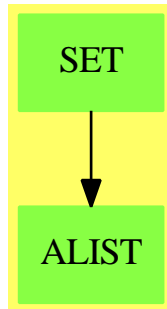
```
b := difference(b, {25})
      {1,4,9,16,32}
                        Type: Set PositiveInteger
```

```
b0
      {1,4,9,16,25}
                        Type: Set PositiveInteger
```

See Also:

- o)help List
- o)show Set

Set (SET)

**Exports:**

any?	bag	brace	cardinality	coerce
complement	construct	convert	copy	count
dictionary	difference	empty	empty?	eq?
eval	every?	extract!	find	hash
index	insert!	inspect	intersect	latex
less?	lookup	map	map!	max
member?	members	min	more?	parts
random	reduce	remove	remove!	removeDuplicates
sample	select	select!	set	size
size?	subset?	symmetricDifference	union	universe
#?	?~=?	?<?	?=?	

— domain SET Set —

```

)abbrev domain SET Set
++ Author: Michael Monagan; revised by Richard Jenks
++ Date Created: August 87 through August 88
++ Date Last Updated: May 1991
++ Description:
++ A set over a domain D models the usual mathematical notion of a finite set
++ of elements from D.
++ Sets are unordered collections of distinct elements
++ (that is, order and duplication does not matter).
++ The notation \spad{set [a,b,c]} can be used to create
++ a set and the usual operations such as union and intersection are available
++ to form new sets.
++ In our implementation, \Language{} maintains the entries in
++ sorted order. Specifically, the parts function returns the entries
++ as a list in ascending order and
++ the extract operation returns the maximum entry.
++ Given two sets s and t where \spad{#s = m} and \spad{#t = n},
++ the complexity of\br
++ \tab{5}\spad{s = t} is \spad{O(min(n,m))}\br
  
```

```

++ \tab{5}\spad{s < t} is \spad{0(max(n,m))}\br
++ \tab{5}\spad{union(s,t)}, \spad{intersect(s,t)}, \spad{minus(s,t)},\br
++ \tab{10 \spad{symmetricDifference(s,t)} is \spad{0(max(n,m))}\br
++ \tab{5}\spad{member(x,t)} is \spad{0(n log n)}\br
++ \tab{5}\spad{insert(x,t)} and \spad{remove(x,t)} is \spad{0(n)}

```

```

Set(S:SetCategory): FiniteSetAggregate S == add
  Rep := FlexibleArray(S)
  # s      == _#$Rep s
  brace()  == empty()
  set()    == empty()
  empty()  == empty()$Rep
  copy s   == copy(s)$Rep
  parts s  == parts(s)$Rep
  inspect s == (empty? s => error "Empty set"; s(maxIndex s))

  extract_! s ==
    x := inspect s
    delete_!(s, maxIndex s)
    x

  find(f, s) == find(f, s)$Rep

  map(f, s) == map_!(f, copy s)

  map_!(f, s) ==
    map_!(f, s)$Rep
    removeDuplicates_! s

  reduce(f, s) == reduce(f, s)$Rep

  reduce(f, s, x) == reduce(f, s, x)$Rep

  reduce(f, s, x, y) == reduce(f, s, x, y)$Rep

  if S has ConvertibleTo InputForm then
    convert(x:%):InputForm ==
      convert [convert("set":Symbol)@InputForm,
               convert(parts x)@InputForm]

  if S has OrderedSet then
    s = t == s = $Rep t
    max s == inspect s
    min s == (empty? s => error "Empty set"; s(minIndex s))

  construct l ==
    zero?(n := #l) => empty()
    a := new(n, first l)
    for i in minIndex(a).. for x in l repeat a.i := x
    removeDuplicates_! sort_! a

```



```

insert_!(x, s) ==
  n := inc maxIndex s
  k := minIndex s
  while k < n and x > s.k repeat k := inc k
  k < n and s.k = x => s
  insert_!(x, s, k)

member?(x, s) == -- binary search
  empty? s => false
  t := maxIndex s
  b := minIndex s
  while b < t repeat
    m := (b+t) quo 2
    if x > s.m then b := m+1 else t := m
  x = s.t

remove_!(x:S, s:%) ==
  n := inc maxIndex s
  k := minIndex s
  while k < n and x > s.k repeat k := inc k
  k < n and x = s.k => delete_!(s, k)
  s

-- the set operations are implemented as variations of merging
intersect(s, t) ==
  m := maxIndex s
  n := maxIndex t
  i := minIndex s
  j := minIndex t
  r := empty()
  while i <= m and j <= n repeat
    s.i = t.j => (concat_!(r, s.i); i := i+1; j := j+1)
    if s.i < t.j then i := i+1 else j := j+1
  r

difference(s:%, t:%) ==
  m := maxIndex s
  n := maxIndex t
  i := minIndex s
  j := minIndex t
  r := empty()
  while i <= m and j <= n repeat
    s.i = t.j => (i := i+1; j := j+1)
    s.i < t.j => (concat_!(r, s.i); i := i+1)
    j := j+1
  while i <= m repeat (concat_!(r, s.i); i := i+1)
  r

symmetricDifference(s, t) ==

```

```

m := maxIndex s
n := maxIndex t
i := minIndex s
j := minIndex t
r := empty()
while i <= m and j <= n repeat
  s.i < t.j => (concat_!(r, s.i); i := i+1)
  s.i > t.j => (concat_!(r, t.j); j := j+1)
  i := i+1; j := j+1
while i <= m repeat (concat_!(r, s.i); i := i+1)
while j <= n repeat (concat_!(r, t.j); j := j+1)
r

subset?(s, t) ==
m := maxIndex s
n := maxIndex t
m > n => false
i := minIndex s
j := minIndex t
while i <= m and j <= n repeat
  s.i = t.j => (i := i+1; j := j+1)
  s.i > t.j => j := j+1
  return false
i > m

union(s:%, t:%) ==
m := maxIndex s
n := maxIndex t
i := minIndex s
j := minIndex t
r := empty()
while i <= m and j <= n repeat
  s.i = t.j => (concat_!(r, s.i); i := i+1; j := j+1)
  s.i < t.j => (concat_!(r, s.i); i := i+1)
  (concat_!(r, t.j); j := j+1)
while i <= m repeat (concat_!(r, s.i); i := i+1)
while j <= n repeat (concat_!(r, t.j); j := j+1)
r

else
insert_!(x, s) ==
  for k in minIndex s .. maxIndex s repeat
    s.k = x => return s
  insert_!(x, s, inc maxIndex s)

remove_!(x:S, s:%) ==
  n := inc maxIndex s
  k := minIndex s
  while k < n repeat
    x = s.k => return delete_!(s, k)

```

```

      k := inc k
    s

```

— SET.dotabb —

```

"SET" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SET"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"SET" -> "ALIST"

```

domain SETMN SetOfMIntegersInOneToN

— SetOfMIntegersInOneToN.input —

```

)set break resume
)sys rm -f SetOfMIntegersInOneToN.output
)spool SetOfMIntegersInOneToN.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SetOfMIntegersInOneToN
--R SetOfMIntegersInOneToN(m: PositiveInteger,n: PositiveInteger) is a domain constructor
--R Abbreviation for SetOfMIntegersInOneToN is SETMN
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SETMN
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean               coerce : % -> OutputForm
--R enumerate : () -> Vector(%)           enumerate : () -> List(%)
--R hash : % -> SingleInteger             index : PositiveInteger -> %
--R latex : % -> String                   lookup : % -> PositiveInteger
--R random : () -> %                      size : () -> NonNegativeInteger
--R ?~=? : (%,% ) -> Boolean
--R delta : (% ,PositiveInteger,PositiveInteger) -> NonNegativeInteger
--R elements : % -> List(PositiveInteger)
--R incrementKthElement : (% ,PositiveInteger) -> Union(%, "failed")
--R member? : (PositiveInteger,%) -> Boolean
--R replaceKthElement : (% ,PositiveInteger,PositiveInteger) -> Union(%, "failed")
--R setOfMinN : List(PositiveInteger) -> %

```

```
--R
--E 1
```

```
)spool
)lisp (bye)
```

— SetOfMIntegersInOneToN.help —

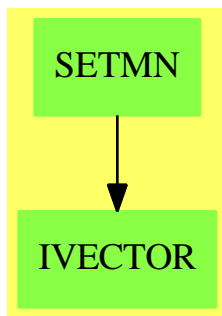
```
=====
SetOfMIntegersInOneToN examples
=====
```

SetOfMIntegersInOneToN implements the subsets of M integers in the interval [1..n]

See Also:

o)show SetOfMIntegersInOneToN

SetOfMIntegersInOneToN (SETMN)



Exports:

coerce	delta	elements	enumerate	hash
incrementKthElement	index	latex	lookup	member?
random	replaceKthElement	setOfMinN	size	?=?
?~=?				

— domain SETMN SetOfMIntegersInOneToN —

```
)abbrev domain SETMN SetOfMIntegersInOneToN
```

```
++ Author: Manuel Bronstein
```

```

++ Date Created: 10 January 1994
++ Date Last Updated: 10 January 1994
++ Description:
++ \spadtype{SetOfMIntegersInOneToN} implements the subsets of M integers
++ in the interval \spad{[1..n]}

SetOfMIntegersInOneToN(m, n): Exports == Implementation where
  PI ==> PositiveInteger
  N ==> NonNegativeInteger
  U ==> Union(%, "failed")
  n,m: PI

Exports ==> Finite with
  incrementKthElement: (%, PI) -> U
    ++ incrementKthElement(S,k) increments the kth element of S,
    ++ and returns "failed" if the result is not a set of M integers
    ++ in \spad{[1..n]} any more.
  replaceKthElement: (%, PI, PI) -> U
    ++ replaceKthElement(S,k,p) replaces the kth element of S by p,
    ++ and returns "failed" if the result is not a set of M integers
    ++ in \spad{[1..n]} any more.
  elements: % -> List PI
    ++ elements(S) returns the list of the elements of S in increasing order.
  setOfMinN: List PI -> %
    ++ setOfMinN([a_1,...,a_m]) returns the set {a_1,...,a_m}.
    ++ Error if {a_1,...,a_m} is not a set of M integers in \spad{[1..n]}.
  enumerate: () -> Vector %
    ++ enumerate() returns a vector of all the sets of M integers in
    ++ \spad{[1..n]}.
  member?: (PI, %) -> Boolean
    ++ member?(p, s) returns true if p is in s, false otherwise.
  delta: (%, PI, PI) -> N
    ++ delta(S,k,p) returns the number of elements of S which are strictly
    ++ between p and the kth element of S.

Implementation ==> add
  Rep := Record(bits:Bits, pos:N)

  reallyEnumerate: () -> Vector %
  enum: (N, N, PI) -> List Bits

  all:Reference Vector % := ref empty()
  sz:Reference N := ref 0

  s1 = s2 == s1.bits =$Bits s2.bits
  coerce(s:%):OutputForm == brace [i::OutputForm for i in elements s]
  random() == index((1 + (random())$Integer rem size())):PI
  reallyEnumerate() == [[b, i] for b in enum(m, n, n) for i in 1..]
  member?(p, s) == s.bits.p

```

```

enumerate() ==
  if empty? all() then all() := reallyEnumerate()
  all()

-- enumerates the sets of p integers in 1..q, returns them as sets in 1..n
-- must have p <= q
enum(p, q, n) ==
  zero? p or zero? q => empty()
  p = q =>
    b := new(n, false)$Bits
    for i in 1..p repeat b.i := true
    [b]
  q1 := (q - 1)::N
  l := enum((p - 1)::N, q1, n)
  if empty? l then l := [new(n, false)$Bits]
  for s in l repeat s.q := true
  concat_!(enum(p, q1, n), l)

size() ==
  if zero? sz() then
    sz() := binomial(n, m)$IntegerCombinatoricFunctions(Integer) :: N
  sz()

lookup s ==
  if empty? all() then all() := reallyEnumerate()
  if zero?(s.pos) then s.pos := position(s, all()) :: N
  s.pos :: PI

index p ==
  p > size() => error "index: argument too large"
  if empty? all() then all() := reallyEnumerate()
  all().p

setOfMinN l ==
  s := new(n, false)$Bits
  count:N := 0
  for i in l repeat
    count := count + 1
    count > m or zero? i or i > n or s.i =>
      error "setOfMinN: improper set of integers"
    s.i := true
  count < m => error "setOfMinN: improper set of integers"
  [s, 0]

elements s ==
  b := s.bits
  l:List PI := empty()
  found:N := 0
  i:PI := 1
  while found < m repeat

```

```

        if b.i then
            l := concat(i, l)
            found := found + 1
        i := i + 1
reverse_! l

incrementKthElement(s, k) ==
    b := s.bits
    found:N := 0
    i:N := 1
    while found < k repeat
        if b.i then found := found + 1
        i := i + 1
    i > n or b.i => "failed"
    newb := copy b
    newb.i := true
    newb.((i-1):N) := false
    [newb, 0]

delta(s, k, p) ==
    b := s.bits
    count:N := found:N := 0
    i:PI := 1
    while found < k repeat
        if b.i then
            found := found + 1
            if i > p and found < k then count := count + 1
        i := i + 1
    count

replaceKthElement(s, k, p) ==
    b := s.bits
    found:N := 0
    i:PI := 1
    while found < k repeat
        if b.i then found := found + 1
        if found < k then i := i + 1
    b.p and i ^= p => "failed"
    newb := copy b
    newb.p := true
    newb.i := false
    [newb, (i = p => s.pos; 0)]

```

— SETMN.dotabb —

"SETMN" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SETMN"]

"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
 "SETMN" -> "IVECTOR"

domain SDPOL SequentialDifferentialPolynomial

— SequentialDifferentialPolynomial.input —

```
)set break resume
)sys rm -f SequentialDifferentialPolynomial.output
)spool SequentialDifferentialPolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SequentialDifferentialPolynomial
--R
--R SequentialDifferentialPolynomial(R: Ring) is a domain constructor
--R Abbreviation for SequentialDifferentialPolynomial is SDPOL
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SDPOL
--R
--R----- Operations -----
--R ??? : (R,% ) -> %          ??? : (R,% ) -> %
--R ??? : (%,%) -> %          ??? : (Integer,% ) -> %
--R ??? : (PositiveInteger,% ) -> %    ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %          ?-? : (%,%) -> %
--R -? : % -> %              ?? : (%,%) -> Boolean
--R D : (%,(R -> R)) -> %      D : % -> % if R has DIFRING
--R 1 : () -> %              0 : () -> %
--R ?^? : (%,PositiveInteger) -> %    coefficients : % -> List(R)
--R coerce : Symbol -> %          coerce : R -> %
--R coerce : Integer -> %        coerce : % -> OutputForm
--R eval : (%,%,%) -> %          eval : (%,Equation(%)) -> %
--R ground : % -> R              ground? : % -> Boolean
--R hash : % -> SingleInteger      initial : % -> %
--R isobaric? : % -> Boolean        latex : % -> String
--R leadingCoefficient : % -> R      leadingMonomial : % -> %
--R map : ((R -> R),%) -> %        monomial? : % -> Boolean
--R monomials : % -> List(%)        one? : % -> Boolean
--R order : % -> NonNegativeInteger  recip : % -> Union(%, "failed")
--R reductum : % -> %              retract : % -> Symbol
--R retract : % -> R              sample : () -> %
--R separant : % -> %              weight : % -> NonNegativeInteger
```



```

--R zero? : % -> Boolean          ?~=? : (%,% ) -> Boolean
--R ?? : (Fraction(Integer),%) -> % if R has ALGEBRA(FRAC(INT))
--R ?? : (% ,Fraction(Integer)) -> % if R has ALGEBRA(FRAC(INT))
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (% ,NonNegativeInteger) -> %
--R ?/? : (% ,R) -> % if R has FIELD
--R ?<? : (% ,%) -> Boolean if R has ORDSET
--R ?<=? : (% ,%) -> Boolean if R has ORDSET
--R ?>? : (% ,%) -> Boolean if R has ORDSET
--R ?>=? : (% ,%) -> Boolean if R has ORDSET
--R D : (% ,(R -> R),NonNegativeInteger) -> %
--R D : (% ,List(Symbol),List(NonNegativeInteger)) -> % if R has PDRING(SYMBOL)
--R D : (% ,Symbol,NonNegativeInteger) -> % if R has PDRING(SYMBOL)
--R D : (% ,List(Symbol)) -> % if R has PDRING(SYMBOL)
--R D : (% ,Symbol) -> % if R has PDRING(SYMBOL)
--R D : (% ,NonNegativeInteger) -> % if R has DIFRING
--R D : (% ,List(SequentialDifferentialVariable(Symbol)),List(NonNegativeInteger)) -> %
--R D : (% ,SequentialDifferentialVariable(Symbol),NonNegativeInteger) -> %
--R D : (% ,List(SequentialDifferentialVariable(Symbol))) -> %
--R D : (% ,SequentialDifferentialVariable(Symbol)) -> %
--R ?? : (% ,NonNegativeInteger) -> %
--R associates? : (% ,%) -> Boolean if R has INTDOM
--R binomThmExpt : (% ,%,NonNegativeInteger) -> % if R has COMRING
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(% ,"failed") if $ has CHARNZ and R has PFECAT or R has CHARNZ
--R coefficient : (% ,List(SequentialDifferentialVariable(Symbol)),List(NonNegativeInteger)) -> %
--R coefficient : (% ,SequentialDifferentialVariable(Symbol),NonNegativeInteger) -> %
--R coefficient : (% ,IndexedExponents(SequentialDifferentialVariable(Symbol))) -> R
--R coerce : % -> % if R has INTDOM
--R coerce : Fraction(Integer) -> % if R has ALGEBRA(FRAC(INT)) or R has RETRACT(FRAC(INT))
--R coerce : SparseMultivariatePolynomial(R,Symbol) -> %
--R coerce : SequentialDifferentialVariable(Symbol) -> %
--R conditionP : Matrix(% ) -> Union(Vector(% ),"failed") if $ has CHARNZ and R has PFECAT
--R content : (% ,SequentialDifferentialVariable(Symbol)) -> % if R has GCDDOM
--R content : % -> R if R has GCDDOM
--R convert : % -> InputForm if SequentialDifferentialVariable(Symbol) has KONVERT(INFORM) and R
--R convert : % -> Pattern(Integer) if SequentialDifferentialVariable(Symbol) has KONVERT(PATTERN)
--R convert : % -> Pattern(Float) if SequentialDifferentialVariable(Symbol) has KONVERT(PATTERN)
--R degree : (% ,Symbol) -> NonNegativeInteger
--R degree : (% ,List(SequentialDifferentialVariable(Symbol))) -> List(NonNegativeInteger)
--R degree : (% ,SequentialDifferentialVariable(Symbol)) -> NonNegativeInteger
--R degree : % -> IndexedExponents(SequentialDifferentialVariable(Symbol))
--R differentialVariables : % -> List(Symbol)
--R differentiate : (% ,(R -> R)) -> %
--R differentiate : (% ,(R -> R),NonNegativeInteger) -> %
--R differentiate : (% ,List(Symbol),List(NonNegativeInteger)) -> % if R has PDRING(SYMBOL)
--R differentiate : (% ,Symbol,NonNegativeInteger) -> % if R has PDRING(SYMBOL)
--R differentiate : (% ,List(Symbol)) -> % if R has PDRING(SYMBOL)
--R differentiate : (% ,Symbol) -> % if R has PDRING(SYMBOL)
--R differentiate : (% ,NonNegativeInteger) -> % if R has DIFRING

```

```

--R differentiate : % -> % if R has DIFRING
--R differentiate : (% ,List(SequentialDifferentialVariable(Symbol)),List(NonNegativeInteger)) -> %
--R differentiate : (% ,SequentialDifferentialVariable(Symbol),NonNegativeInteger) -> %
--R differentiate : (% ,List(SequentialDifferentialVariable(Symbol))) -> %
--R differentiate : (% ,SequentialDifferentialVariable(Symbol)) -> %
--R discriminant : (% ,SequentialDifferentialVariable(Symbol)) -> % if R has COMRING
--R eval : (% ,List(Symbol),List(R)) -> % if R has DIFRING
--R eval : (% ,Symbol,R) -> % if R has DIFRING
--R eval : (% ,List(Symbol),List(%)) -> % if R has DIFRING
--R eval : (% ,Symbol,%) -> % if R has DIFRING
--R eval : (% ,List(SequentialDifferentialVariable(Symbol)),List(%)) -> %
--R eval : (% ,SequentialDifferentialVariable(Symbol),%) -> %
--R eval : (% ,List(SequentialDifferentialVariable(Symbol)),List(R)) -> %
--R eval : (% ,SequentialDifferentialVariable(Symbol),R) -> %
--R eval : (% ,List(%),List(%)) -> %
--R eval : (% ,List(Equation(%))) -> %
--R exquo : (% ,%) -> Union(%,"failed") if R has INTDOM
--R exquo : (% ,R) -> Union(%,"failed") if R has INTDOM
--R factor : % -> Factored(%) if R has PFECAT
--R factorPolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if R has PF
--R factorSquareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if
--R gcd : (% ,%) -> % if R has GCDDOM
--R gcd : List(%) -> % if R has GCDDOM
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynom
--R isExpt : % -> Union(Record(var: SequentialDifferentialVariable(Symbol),exponent: NonNegativeInteger),"f
--R isPlus : % -> Union(List(%),"failed")
--R isTimes : % -> Union(List(%),"failed")
--R lcm : (% ,%) -> % if R has GCDDOM
--R lcm : List(%) -> % if R has GCDDOM
--R leader : % -> SequentialDifferentialVariable(Symbol)
--R mainVariable : % -> Union(SequentialDifferentialVariable(Symbol),"failed")
--R makeVariable : % -> (NonNegativeInteger -> %) if R has DIFRING
--R makeVariable : Symbol -> (NonNegativeInteger -> %)
--R mapExponents : ((IndexedExponents(SequentialDifferentialVariable(Symbol)) -> IndexedExponents(Sequential
--R max : (% ,%) -> % if R has ORDSET
--R min : (% ,%) -> % if R has ORDSET
--R minimumDegree : (% ,List(SequentialDifferentialVariable(Symbol))) -> List(NonNegativeInteger)
--R minimumDegree : (% ,SequentialDifferentialVariable(Symbol)) -> NonNegativeInteger
--R minimumDegree : % -> IndexedExponents(SequentialDifferentialVariable(Symbol))
--R monicDivide : (% ,%,SequentialDifferentialVariable(Symbol)) -> Record(quotient: %,remainder: %)
--R monomial : (% ,List(SequentialDifferentialVariable(Symbol)),List(NonNegativeInteger)) -> %
--R monomial : (% ,SequentialDifferentialVariable(Symbol),NonNegativeInteger) -> %
--R monomial : (R,IndexedExponents(SequentialDifferentialVariable(Symbol))) -> %
--R multivariate : (SparseUnivariatePolynomial(%),SequentialDifferentialVariable(Symbol)) -> %
--R multivariate : (SparseUnivariatePolynomial(R),SequentialDifferentialVariable(Symbol)) -> %
--R numberOfMonomials : % -> NonNegativeInteger
--R order : (% ,Symbol) -> NonNegativeInteger
--R patternMatch : (% ,Pattern(Integer),PatternMatchResult(Integer,%)) -> PatternMatchResult(Integer,%) if S
--R patternMatch : (% ,Pattern(Float),PatternMatchResult(Float,%)) -> PatternMatchResult(Float,%) if Sequen
--R pomopo! : (% ,R,IndexedExponents(SequentialDifferentialVariable(Symbol)),%) -> %

```

```

--R prime? : % -> Boolean if R has PFECAT
--R primitiveMonomials : % -> List(%)
--R primitivePart : (%,SequentialDifferentialVariable(Symbol)) -> % if R has GCDDOM
--R primitivePart : % -> % if R has GCDDOM
--R reducedSystem : Matrix(%) -> Matrix(R)
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(R),vec: Vector(R))
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) if R has LINEXP(INT)
--R reducedSystem : Matrix(%) -> Matrix(Integer) if R has LINEXP(INT)
--R resultant : (%,%,SequentialDifferentialVariable(Symbol)) -> % if R has COMRING
--R retract : % -> SparseMultivariatePolynomial(R,Symbol)
--R retract : % -> SequentialDifferentialVariable(Symbol)
--R retract : % -> Integer if R has RETRACT(INT)
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(SparseMultivariatePolynomial(R,Symbol),"failed")
--R retractIfCan : % -> Union(Symbol,"failed")
--R retractIfCan : % -> Union(SequentialDifferentialVariable(Symbol),"failed")
--R retractIfCan : % -> Union(Integer,"failed") if R has RETRACT(INT)
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(R,"failed")
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(%)),SparseUnivariatePolynomial(%)) -> List(SparseUnivariatePolynomial(%))
--R squareFree : % -> Factored(%) if R has GCDDOM
--R squareFreePart : % -> % if R has GCDDOM
--R squareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%))
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R totalDegree : (%,List(SequentialDifferentialVariable(Symbol))) -> NonNegativeInteger
--R totalDegree : % -> NonNegativeInteger
--R unit? : % -> Boolean if R has INTDOM
--R unitCanonical : % -> % if R has INTDOM
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %) if R has INTDOM
--R univariate : % -> SparseUnivariatePolynomial(R)
--R univariate : (%,SequentialDifferentialVariable(Symbol)) -> SparseUnivariatePolynomial(%)
--R variables : % -> List(SequentialDifferentialVariable(Symbol))
--R weight : (%,Symbol) -> NonNegativeInteger
--R weights : (%,Symbol) -> List(NonNegativeInteger)
--R weights : % -> List(NonNegativeInteger)
--R
--E 1

)spool
)lisp (bye)

```

— SequentialDifferentialPolynomial.help —

```

=====
SequentialDifferentialPolynomial examples
=====

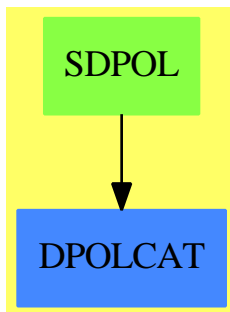
```

SequentialDifferentialPolynomial implements an ordinary differential polynomial ring in arbitrary number of differential indeterminates, with coefficients in a ring. The ranking on the differential indeterminate is sequential.

See Also:

o)show SequentialDifferentialPolynomial

SequentialDifferentialPolynomial (SDPOL)



See

- ⇒ “OrderlyDifferentialVariable” (ODVAR) [16](#) on page [1855](#)
- ⇒ “SequentialDifferentialVariable” (SDVAR) [20](#) on page [2389](#)
- ⇒ “DifferentialSparseMultivariatePolynomial” (DSMP) [5](#) on page [541](#)
- ⇒ “OrderlyDifferentialPolynomial” (ODPOL) [16](#) on page [1851](#)

Exports:

0	1	associates?
binomThmExpt	characteristic	charthRoot
coefficient	coefficients	coerce
conditionP	content	convert
D	degree	differentialVariables
differentiate	discriminant	eval
exquo	factor	factorPolynomial
factorSquareFreePolynomial	gcd	gcdPolynomial
ground	ground?	hash
initial	isExpt	isobaric?
isPlus	isTimes	latex
lcm	leader	leadingCoefficient
leadingMonomial	mainVariable	makeVariable
map	mapExponents	max
min	minimumDegree	monicDivide
monomial	monomial?	monomials
multivariate	numberOfMonomials	one?
order	patternMatch	pomopo!
prime?	primitiveMonomials	primitivePart
recip	reducedSystem	reductum
resultant	retract	retractIfCan
sample	separant	solveLinearPolynomialEquation
squareFree	squareFreePart	squareFreePolynomial
subtractIfCan	totalDegree	unit?
unitCanonical	unitNormal	univariate
variables	weight	weights
zero?	?^?	?*?
?**?	?+?	?-?
-?	?=?	?~=?
?/?	?<?	?<=?
?>?	?>=?	

— domain SDPOL SequentialDifferentialPolynomial —

```

)abbrev domain SDPOL SequentialDifferentialPolynomial
++ Author: William Sit
++ Date Created: 24 September, 1991
++ Date Last Updated: 7 February, 1992
++ References:
++ Kolchin, E.R. "Differential Algebra and Algebraic Groups"
++ (Academic Press, 1973).
++ Description:
++ \spadtype{SequentialDifferentialPolynomial} implements
++ an ordinary differential polynomial ring in arbitrary number
++ of differential indeterminates, with coefficients in a
++ ring. The ranking on the differential indeterminate is sequential.

```

```

SequentialDifferentialPolynomial(R):
  Exports == Implementation where
    R: Ring
    S ==> Symbol
    V ==> SequentialDifferentialVariable S
    E ==> IndexedExponents(V)
    SMP ==> SparseMultivariatePolynomial(R, S)
    Exports ==> Join(DifferentialPolynomialCategory(R,S,V,E),
                     RetractableTo SMP)

Implementation ==> DifferentialSparseMultivariatePolynomial(R,S,V)

```

— SDPOL.dotabb —

```

"SDPOL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SDPOL"]
"DPOLCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DPOLCAT"]
"SDPOL" -> "DPOLCAT"

```

domain SDVAR SequentialDifferentialVariable

— SequentialDifferentialVariable.input —

```

)set break resume
)sys rm -f SequentialDifferentialVariable.output
)spool SequentialDifferentialVariable.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SequentialDifferentialVariable
--R
--R SequentialDifferentialVariable(S: OrderedSet) is a domain constructor
--R Abbreviation for SequentialDifferentialVariable is SDVAR
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SDVAR
--R
--R----- Operations -----
--R ?<? : (%,% ) -> Boolean          ?<=? : (%,% ) -> Boolean
--R ?=? : (%,% ) -> Boolean          ?>? : (%,% ) -> Boolean

```

```

--R ?>=? : (%,%) -> Boolean
--R coerce : % -> OutputForm
--R hash : % -> SingleInteger
--R max : (%,%) -> %
--R order : % -> NonNegativeInteger
--R variable : % -> S
--R ?=? : (%,%) -> Boolean
--R differentiate : (%,NonNegativeInteger) -> %
--R makeVariable : (S,NonNegativeInteger) -> %
--R retractIfCan : % -> Union(S,"failed")
--R
--E 1

)spool
)lisp (bye)

```

— SequentialDifferentialVariable.help —

=====

SequentialDifferentialVariable examples

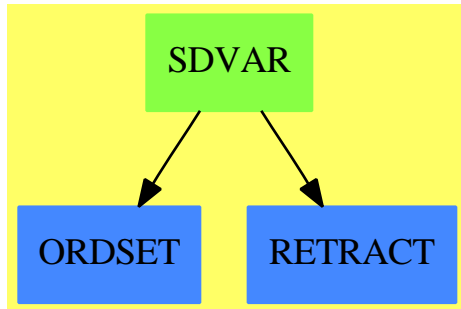
=====

OrderlyDifferentialVariable adds a commonly used sequential ranking to the set of derivatives of an ordered list of differential indeterminates. A sequential ranking is a ranking $<$ of the derivatives with the property that for any derivative v , there are only a finite number of derivatives u with $u < v$. This domain belongs to DifferentialVariableCategory. It defines weight to be just order, and it defines a sequential ranking $<$ on derivatives u by the lexicographic order on the pair $(\text{variable}(u), \text{order}(u))$.

See Also:

o)show SequentialDifferentialVariable

SequentialDifferentialVariable (SDVAR)



See

- ⇒ “OrderlyDifferentialVariable” (ODVAR) [16](#) on page [1855](#)
- ⇒ “DifferentialSparseMultivariatePolynomial” (DSMP) [5](#) on page [541](#)
- ⇒ “OrderlyDifferentialPolynomial” (ODPOL) [16](#) on page [1851](#)
- ⇒ “SequentialDifferentialPolynomial” (SDPOL) [20](#) on page [2385](#)

Exports:

coerce	differentiate	hash	latex	makeVariable
max	min	order	retract	retractIfCan
variable	weight	?~=?	?<?	?<=?
?=?	?>?	?>=?		

— domain SDVAR SequentialDifferentialVariable —

```

)abbrev domain SDVAR SequentialDifferentialVariable
++ Author: William Sit
++ Date Created: 19 July 1990
++ Date Last Updated: 13 September 1991
++ References:
++ Kolchin, E.R. "Differential Algebra and Algebraic Groups"
++ (Academic Press, 1973).
++ Description:
++ \spadtype{OrderlyDifferentialVariable} adds a commonly used sequential
++ ranking to the set of derivatives of an ordered list of differential
++ indeterminates. A sequential ranking is a ranking \spadfun{<} of the
++ derivatives with the property that for any derivative v,
++ there are only a finite number of derivatives u with u \spadfun{<} v.
++ This domain belongs to \spadtype{DifferentialVariableCategory}. It
++ defines \spadfun{weight} to be just \spadfun{order}, and it
++ defines a sequential ranking \spadfun{<} on derivatives u by the
++ lexicographic order on the pair
++ (\spadfun{variable}(u), \spadfun{order}(u)).

SequentialDifferentialVariable(S:OrderedSet):DifferentialVariableCategory(S)
== add
  Rep := Record(var:S, ord:NonNegativeInteger)

```



```

makeVariable(s,n) == [s, n]
variable v      == v.var
order v        == v.ord
v < u ==
  variable v = variable u => order v < order u
  variable v < variable u

```

— SDVAR.dotabb —

```

"SDVAR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SDVAR"]
"ORDSET" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ORDSET"]
"RETRACT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RETRACT"]
"SDVAR" -> "ORDSET"
"SDVAR" -> "RETRACT"

```

domain SEX SExpression

— SExpression.input —

```

)set break resume
)sys rm -f SExpression.output
)spool SExpression.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SExpression
--R
--R SExpression is a domain constructor
--R Abbreviation for SExpression is SEX
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SEX
--R
--R----- Operations -----
--R #? : % -> Integer           ?? : (%,%) -> Boolean
--R atom? : % -> Boolean       car : % -> %
--R cdr : % -> %               coerce : % -> OutputForm
--R convert : OutputForm -> %   convert : DoubleFloat -> %
--R convert : Integer -> %     convert : Symbol -> %

```

```

--R convert : String -> %
--R destruct : % -> List(%)
--R ?.? : (%,Integer) -> %
--R expr : % -> OutputForm
--R float? : % -> Boolean
--R integer : % -> Integer
--R latex : % -> String
--R null? : % -> Boolean
--R string : % -> String
--R symbol : % -> Symbol
--R ?~=? : (%,% ) -> Boolean
--R
--E 1

```

```

convert : List(%) -> %
?.? : (% ,List(Integer)) -> %
eq : (% ,%) -> Boolean
float : % -> DoubleFloat
hash : % -> SingleInteger
integer? : % -> Boolean
list? : % -> Boolean
pair? : % -> Boolean
string? : % -> Boolean
symbol? : % -> Boolean

```

```

)spool
)lisp (bye)

```

— SExpression.help —

=====

SExpression examples

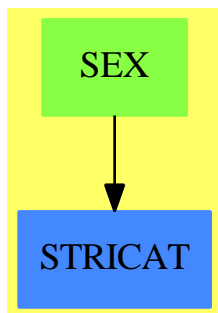
=====

This domain allows the manipulation of the usual Lisp values.

See Also:

o)show SExpression

SExpression (SEX)



See

⇒ “SExpressionOf” (SEXOF) [20](#) on page [2394](#)

Exports:

atom?	car	cdr	coerce	convert
destruct	eq	expr	float	float?
hash	integer	integer?	latex	list?
null?	pair?	string	string?	symbol
symbol?	#?	?.?	?=?	?~=?

— domain SEX SExpression —

```
)abbrev domain SEX SExpression
++ Author: S.M.Watt
++ Date Created: July 1987
++ Date Last Modified: 23 May 1991
++ Description:
++ This domain allows the manipulation of the usual Lisp values;

SExpression()
== SExpressionOf(String, Symbol, Integer, DoubleFloat, OutputForm)
```

—————

— SEX.dotabb —

```
"SEX" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SEX"]
"STRICAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=STRICAT"]
"SEX" -> "STRICAT"
```

—————

domain SEXOF SExpressionOf

— SExpressionOf.input —

```
)set break resume
)sys rm -f SExpressionOf.output
)spool SExpressionOf.output
)set message test on
)set message auto off
)clear all
```

--S 1 of 1

```

)show SExpressionOf
--R
--R SExpressionOf(Str: SetCategory,Sym: SetCategory,Int: SetCategory,Flt: SetCategory,Expr: SetCategory) is
--R Abbreviation for SExpressionOf is SEXOF
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SEXOF
--R
--R----- Operations -----
--R #? : % -> Integer           ?? : (%,%) -> Boolean
--R atom? : % -> Boolean       car : % -> %
--R cdr : % -> %              coerce : % -> OutputForm
--R convert : Expr -> %        convert : Flt -> %
--R convert : Int -> %         convert : Sym -> %
--R convert : Str -> %         convert : List(%) -> %
--R destruct : % -> List(%)    ?? : (%,List(Integer)) -> %
--R ?? : (%,Integer) -> %      eq : (%,%) -> Boolean
--R expr : % -> Expr           float : % -> Flt
--R float? : % -> Boolean      hash : % -> SingleInteger
--R integer : % -> Int          integer? : % -> Boolean
--R latex : % -> String         list? : % -> Boolean
--R null? : % -> Boolean        pair? : % -> Boolean
--R string : % -> Str           string? : % -> Boolean
--R symbol : % -> Sym           symbol? : % -> Boolean
--R ?~=? : (%,%) -> Boolean
--R
--E 1

)spool
)lisp (bye)

```

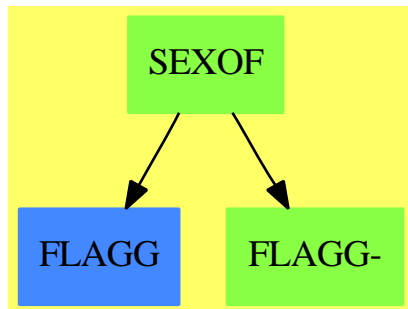
— SExpressionOf.help —

SExpressionOf examples

This domain allows the manipulation of Lisp values over arbitrary atomic types.

See Also:

o)show SExpressionOf

SExpressionOf (SEXOF)

See

⇒ “SExpression” (SEX) 20 on page 2391

Exports:

atom?	car	cdr	coerce	convert
destruct	eq	expr	float	float?
hash	integer	integer?	latex	list?
null?	pair?	string	string?	symbol
symbol?	#?	?.?	?=?	?~=?

— domain SEXOF SExpressionOf —

```

)abbrev domain SEXOF SExpressionOf
++ Author: S.M.Watt
++ Date Created: July 1987
++ Date Last Modified: 23 May 1991
++ Description:
++ This domain allows the manipulation of Lisp values over
++ arbitrary atomic types.
-- Allows the names of the atomic types to be chosen.
-- *** Warning *** Although the parameters are declared only to be Sets,
-- *** Warning *** they must have the appropriate representations.

SExpressionOf(Str, Sym, Int, Flt, Expr): Decl == Body where
  Str, Sym, Int, Flt, Expr: SetCategory

  Decl ==> SExpressionCategory(Str, Sym, Int, Flt, Expr)

  Body ==> add
    Rep := Expr

    dotex:OutputForm := INTERN(".")$Lisp

    coerce(b:%):OutputForm ==
      null? b => paren empty()
      atom? b => coerce(b)$Rep
  
```

```

r := b
while not atom? r repeat r := cdr r
l1 := [b1::OutputForm for b1 in (l := destruct b)]
not null? r =>
  paren blankSeparate concat_!(l1, [dotex, r::OutputForm])
#l = 2 and (first(l1) = QUOTE)@Boolean => quote first rest l1
paren blankSeparate l1

b1 = b2      == EQUAL(b1,b2)$Lisp
eq(b1, b2)   == EQ(b1,b2)$Lisp

null? b      == NULL(b)$Lisp
atom? b      == ATOM(b)$Lisp
pair? b      == CONSP(b)$Lisp

list?  b     == CONSP(b)$Lisp or NULL(b)$Lisp
string? b    == STRINGP(b)$Lisp
symbol? b    == IDENTP(b)$Lisp
integer? b   == INTEGERP(b)$Lisp
float?  b    == FLOATP(b)$Lisp

destruct b == (list? b    => b pretend List %; error "Non-list")
string b == (STRINGP(b)$Lisp=> b pretend Str;error "Non-string")
symbol b == (IDENTP(b)$Lisp => b pretend Sym;error "Non-symbol")
float b == (FLOATP(b)$Lisp => b pretend Flt;error "Non-float")
integer b == (INTEGERP(b)$Lisp => b pretend Int;error "Non-integer")
expr b == b pretend Expr

convert(l: List %) == l pretend %
convert(st: Str)   == st pretend %
convert(sy: Sym)   == sy pretend %
convert(n: Int)    == n pretend %
convert(f: Flt)    == f pretend %
convert(e: Expr)   == e

car b      == CAR(b)$Lisp
cdr b      == CDR(b)$Lisp
# b        == LENGTH(b)$Lisp
elt(b:%, i:Integer) == destruct(b).i
elt(b:%, li>List Integer) ==
  for i in li repeat b := destruct(b).i
  b

```

— SEXOF.dotabb —

"SEXOF" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SEXOF"]

```
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"SEXOF" -> "FLAGG"
"SEXOF" -> "FLAGG-"
```

domain SAE SimpleAlgebraicExtension

— SimpleAlgebraicExtension.input —

```
)set break resume
)sys rm -f SimpleAlgebraicExtension.output
)spool SimpleAlgebraicExtension.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SimpleAlgebraicExtension
--R SimpleAlgebraicExtension(R: CommutativeRing,UP: UnivariatePolynomialCategory(R),M: UP) is a
--R Abbreviation for SimpleAlgebraicExtension is SAE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SAE
--R
--R----- Operations -----
--R ?? : (R,%) -> %
--R ?? : (%,%) -> %
--R ?? : (PositiveInteger,%) -> %
--R ?+? : (%,%) -> %
--R -? : % -> %
--R 1 : () -> %
--R ^? : (%,PositiveInteger) -> %
--R coerce : R -> %
--R coerce : % -> OutputForm
--R convert : % -> UP
--R convert : % -> Vector(R)
--R definingPolynomial : () -> UP
--R discriminant : Vector(%) -> R
--R hash : % -> SingleInteger
--R latex : % -> String
--R norm : % -> R
--R rank : () -> PositiveInteger
--R reduce : UP -> %
--R retract : % -> R
--R trace : % -> R
--R ?? : (%,R) -> %
--R ?? : (Integer,%) -> %
--R ***? : (%,PositiveInteger) -> %
--R ?-? : (%,%) -> %
--R ==? : (%,%) -> Boolean
--R 0 : () -> %
--R basis : () -> Vector(%)
--R coerce : Integer -> %
--R convert : UP -> %
--R convert : Vector(R) -> %
--R coordinates : % -> Vector(R)
--R discriminant : () -> R
--R generator : () -> %
--R inv : % -> % if R has FIELD
--R lift : % -> UP
--R one? : % -> Boolean
--R recip : % -> Union(%, "failed")
--R represents : Vector(R) -> %
--R sample : () -> %
--R traceMatrix : () -> Matrix(R)
```

```

--R zero? : % -> Boolean          ?~=? : (%,% ) -> Boolean
--R ?? : (% , Fraction(Integer)) -> % if R has FIELD
--R ?? : (Fraction(Integer), %) -> % if R has FIELD
--R ?? : (NonNegativeInteger, %) -> %
--R ??? : (% , Integer) -> % if R has FIELD
--R ??? : (% , NonNegativeInteger) -> %
--R ?/? : (% , %) -> % if R has FIELD
--R D : (% , (R -> R)) -> % if R has FIELD
--R D : (% , (R -> R), NonNegativeInteger) -> % if R has FIELD
--R D : (% , List(Symbol), List(NonNegativeInteger)) -> % if R has FIELD and R has PDRING(SYMBOL)
--R D : (% , Symbol, NonNegativeInteger) -> % if R has FIELD and R has PDRING(SYMBOL)
--R D : (% , List(Symbol)) -> % if R has FIELD and R has PDRING(SYMBOL)
--R D : (% , Symbol) -> % if R has FIELD and R has PDRING(SYMBOL)
--R D : (% , NonNegativeInteger) -> % if R has DIFRING and R has FIELD or R has FFIELDC
--R D : % -> % if R has DIFRING and R has FIELD or R has FFIELDC
--R ?? : (% , Integer) -> % if R has FIELD
--R ?? : (% , NonNegativeInteger) -> %
--R associates? : (% , %) -> Boolean if R has FIELD
--R characteristic : () -> NonNegativeInteger
--R characteristicPolynomial : % -> UP
--R charthRoot : % -> Union(% , "failed") if R has CHARNZ
--R charthRoot : % -> % if R has FFIELDC
--R coerce : Fraction(Integer) -> % if R has FIELD or R has RETRACT(FRAC(INT))
--R coerce : % -> % if R has FIELD
--R conditionP : Matrix(% ) -> Union(Vector(% ) , "failed") if R has FFIELDC
--R coordinates : Vector(% ) -> Matrix(R)
--R coordinates : (Vector(% ) , Vector(% )) -> Matrix(R)
--R coordinates : (% , Vector(% )) -> Vector(R)
--R createPrimitiveElement : () -> % if R has FFIELDC
--R derivationCoordinates : (Vector(% ) , (R -> R)) -> Matrix(R) if R has FIELD
--R differentiate : (% , (R -> R)) -> % if R has FIELD
--R differentiate : (% , (R -> R), NonNegativeInteger) -> % if R has FIELD
--R differentiate : (% , List(Symbol), List(NonNegativeInteger)) -> % if R has FIELD and R has PDRING(SYMBOL)
--R differentiate : (% , Symbol, NonNegativeInteger) -> % if R has FIELD and R has PDRING(SYMBOL)
--R differentiate : (% , List(Symbol)) -> % if R has FIELD and R has PDRING(SYMBOL)
--R differentiate : (% , Symbol) -> % if R has FIELD and R has PDRING(SYMBOL)
--R differentiate : (% , NonNegativeInteger) -> % if R has DIFRING and R has FIELD or R has FFIELDC
--R differentiate : % -> % if R has DIFRING and R has FIELD or R has FFIELDC
--R discreteLog : (% , %) -> Union(NonNegativeInteger , "failed") if R has FFIELDC
--R discreteLog : % -> NonNegativeInteger if R has FFIELDC
--R divide : (% , %) -> Record(quotient: % , remainder: % ) if R has FIELD
--R enumerate : () -> List(% ) if R has FINITE
--R euclideanSize : % -> NonNegativeInteger if R has FIELD
--R expressIdealMember : (List(% ) , %) -> Union(List(% ) , "failed") if R has FIELD
--R exquo : (% , %) -> Union(% , "failed") if R has FIELD
--R extendedEuclidean : (% , %) -> Record(coef1: % , coef2: % , generator: % ) if R has FIELD
--R extendedEuclidean : (% , % , %) -> Union(Record(coef1: % , coef2: % ) , "failed") if R has FIELD
--R factor : % -> Factored(% ) if R has FIELD
--R factorsOfCyclicGroupSize : () -> List(Record(factor: Integer , exponent: Integer)) if R has FFIELDC
--R gcd : (% , %) -> % if R has FIELD

```



```

--R gcd : List(%) -> % if R has FIELD
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUniva
--R index : PositiveInteger -> % if R has FINITE
--R init : () -> % if R has FFIELDC
--R lcm : (%,%) -> % if R has FIELD
--R lcm : List(%) -> % if R has FIELD
--R lookup : % -> PositiveInteger if R has FINITE
--R minimalPolynomial : % -> UP if R has FIELD
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed") if R has FIELD
--R nextItem : % -> Union(%,"failed") if R has FFIELDC
--R order : % -> OnePointCompletion(PositiveInteger) if R has FFIELDC
--R order : % -> PositiveInteger if R has FFIELDC
--R prime? : % -> Boolean if R has FIELD
--R primeFrobenius : % -> % if R has FFIELDC
--R primeFrobenius : (%,NonNegativeInteger) -> % if R has FFIELDC
--R primitive? : % -> Boolean if R has FFIELDC
--R primitiveElement : () -> % if R has FFIELDC
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %) if R has FIELD
--R ?quo? : (%,%) -> % if R has FIELD
--R random : () -> % if R has FINITE
--R reduce : Fraction(UP) -> Union(%,"failed") if R has FIELD
--R reducedSystem : Matrix(%) -> Matrix(R)
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(R),vec: Vector(R))
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) i
--R reducedSystem : Matrix(%) -> Matrix(Integer) if R has LINEXP(INT)
--R regularRepresentation : % -> Matrix(R)
--R regularRepresentation : (%,Vector(%)) -> Matrix(R)
--R ?rem? : (%,%) -> % if R has FIELD
--R representationType : () -> Union("prime",polynomial,normal,cyclic) if R has FFIELDC
--R represents : (Vector(R),Vector(%)) -> %
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retract : % -> Integer if R has RETRACT(INT)
--R retractIfCan : % -> Union(R,"failed")
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(Integer,"failed") if R has RETRACT(INT)
--R size : () -> NonNegativeInteger if R has FINITE
--R sizeLess? : (%,%) -> Boolean if R has FIELD
--R squareFree : % -> Factored(%) if R has FIELD
--R squareFreePart : % -> % if R has FIELD
--R subtractIfCan : (%,%) -> Union(%,"failed")
--R tableForDiscreteLogarithm : Integer -> Table(PositiveInteger,NonNegativeInteger) if R has FF
--R traceMatrix : Vector(%) -> Matrix(R)
--R unit? : % -> Boolean if R has FIELD
--R unitCanonical : % -> % if R has FIELD
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %) if R has FIELD
--R
--E 1

)spool
)lisp (bye)

```

— SimpleAlgebraicExtension.help —

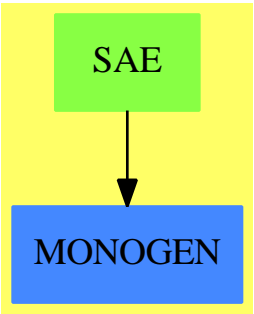
SimpleAlgebraicExtension examples

Algebraic extension of a ring by a single polynomial. Domain which represents simple algebraic extensions of arbitrary rings. The first argument to the domain, R, is the underlying ring, the second argument is a domain of univariate polynomials over K, while the last argument specifies the defining minimal polynomial.

The elements of the domain are canonically represented as polynomials of degree less than that of the minimal polynomial with coefficients in R. The second argument is both the type of the third argument and the underlying representation used by SAE itself.

See Also:
o)show SimpleAlgebraicExtension

SimpleAlgebraicExtension (SAE)



Exports:

0	1	associates?
basis	characteristic	characteristicPolynomial
charthRoot	coerce	conditionP
convert	coordinates	createPrimitiveElement
D	definingPolynomial	derivationCoordinates
differentiate	discreteLog	discriminant
divide	euclideanSize	expressIdealMember
exquo	extendedEuclidean	factor
factorsOfCyclicGroupSize	gcd	gcdPolynomial
generator	hash	index
init	inv	latex
lcm	lift	lookup
minimalPolynomial	multiEuclidean	nextItem
norm	one?	order
prime?	primeFrobenius	primitive?
primitiveElement	principalIdeal	random
rank	recip	reduce
reducedSystem	regularRepresentation	representationType
represents	retract	retractIfCan
sample	size	sizeLess?
squareFree	squareFreePart	subtractIfCan
tableForDiscreteLogarithm	trace	traceMatrix
unit?	unitCanonical	unitNormal
zero?	?*?	?**?
?+?	?-?	-?
?=?	?^?	?~=?
?/?	?quo?	?rem?

— domain SAE SimpleAlgebraicExtension —

```

)abbrev domain SAE SimpleAlgebraicExtension
++ Author: Barry Trager, Manuel Bronstein, Clifton Williamson
++ Date Created: 1986
++ Date Last Updated: 9 May 1994
++ Description:
++ Algebraic extension of a ring by a single polynomial.
++ Domain which represents simple algebraic extensions of arbitrary
++ rings. The first argument to the domain, R, is the underlying ring,
++ the second argument is a domain of univariate polynomials over K,
++ while the last argument specifies the defining minimal polynomial.
++ The elements of the domain are canonically represented as polynomials
++ of degree less than that of the minimal polynomial with coefficients
++ in R. The second argument is both the type of the third argument and
++ the underlying representation used by \spadtype{SAE} itself.

SimpleAlgebraicExtension(R:CommutativeRing,
  UP:UnivariatePolynomialCategory R, M:UP): MonogenicAlgebra(R, UP) == add
  --sqFr(pb): FactorS(Poly) from UnivPolySquareFree(Poly)

```

```

--degree(M) > 0 and M must be monic if R is not a field.
if (r := recip leadingCoefficient M) case "failed" then
    error "Modulus cannot be made monic"

Rep := UP
x,y :$
c: R

mkDisc    : Boolean -> Void
mkDiscMat: Boolean -> Void

M    := r::R * M
d    := degree M
d1   := subtractIfCan(d,1)::NonNegativeInteger
discmat:Matrix(R) := zero(d, d)
nodiscmat?:Reference(Boolean) := ref true
disc:Reference(R) := ref 0
nodisc?:Reference(Boolean) := ref true
bsis := [monomial(1, i)$Rep for i in 0..d1]$Vector(Rep)

if R has Finite then
    size == size$R ** d
    random == represents([random()$R for i in 0..d1])
0 == 0$Rep
1 == 1$Rep
c * x == c *$Rep x
n:Integer * x == n *$Rep x
coerce(n:Integer):$ == coerce(n)$Rep
coerce(c) == monomial(c,0)$Rep
coerce(x):OutputForm == coerce(x)$Rep
lift(x) == x pretend Rep
reduce(p:UP):$ == (monicDivide(p,M)$Rep).remainder
x = y == x ==$Rep y
x + y == x +$Rep y
- x == -$Rep x
x * y == reduce((x *$Rep y) pretend UP)
coordinates(x) == [coefficient(lift(x),i) for i in 0..d1]
represents(vect) == +/[monomial(vect.(i+1),i) for i in 0..d1]
definingPolynomial() == M
characteristic() == characteristic()$R
rank() == d::PositiveInteger
basis() == copy(bsis@Vector(Rep) pretend Vector($))
--!! I inserted 'copy' in the definition of 'basis' -- cjlw 7/19/91

if R has Field then
    minimalPolynomial x == squareFreePart characteristicPolynomial x

if R has Field then
    coordinates(x:$,bas: Vector $) ==
        (m := inverse transpose coordinates bas) case "failed" =>

```

```

        error "coordinates: second argument must be a basis"
      (m :: Matrix R) * coordinates(x)

else if R has IntegralDomain then
  coordinates(x:$,bas: Vector $) ==
    -- we work over the quotient field of R to invert a matrix
    qf := Fraction R
    imatqf := InnerMatrixQuotientFieldFunctions(R,Vector R,Vector R,
      Matrix R,qf,Vector qf,Vector qf,Matrix qf)
    mat := transpose coordinates bas
    (m := inverse(mat)$imatqf) case "failed" =>
      error "coordinates: second argument must be a basis"
    coordsQF: Vector qf :=
      map(y +-> y::qf,coordinates x)$VectorFunctions2(R,qf)
    -- here are the coordinates as elements of the quotient field:
    vecQF := (m :: Matrix qf) * coordsQF
    vec : Vector R := new(d,0)
    for i in 1..d repeat
      xi := qelt(vecQF,i)
      denom(xi) = 1 => qsetelt_!(vec,i,numer xi)
      error "coordinates: coordinates are not integral over ground ring"
    vec

reducedSystem(m:Matrix $):Matrix(R) ==
  reducedSystem(map(lift, m)$MatrixCategoryFunctions2($, Vector $,
    Vector $, Matrix $, UP, Vector UP, Vector UP, Matrix UP))

reducedSystem(m:Matrix $, v:Vector $):Record(mat:Matrix R,vec:Vector R) ==
  reducedSystem(map(lift, m)$MatrixCategoryFunctions2($, Vector $,
    Vector $, Matrix $, UP, Vector UP, Vector UP, Matrix UP),
    map(lift, v)$VectorFunctions2($, UP))

discriminant() ==
  if nodisc?() then mkDisc false
  disc()

mkDisc b ==
  nodisc?() := b
  disc() := discriminant M
  void

traceMatrix() ==
  if nodiscmat?() then mkDiscMat false
  discmat

mkDiscMat b ==
  nodiscmat?() := b
  mr := minRowIndex discmat; mc := minColIndex discmat
  for i in 0..d1 repeat
    for j in 0..d1 repeat

```

```

    qsetelt_!(discmat,mr + i,mc + j,trace reduce monomial(1,i + j))
void

trace x ==          --this could be coded perhaps more efficiently
xn := x;  ans := coefficient(lift xn, 0)
for n in 1..d1 repeat
  (xn := generator() * xn;  ans := coefficient(lift xn, n) + ans)
ans

if R has Finite then
  index k ==
    i:Integer := k rem size()
    p:Integer := size()$R
    ans:$ := 0
    for j in 0.. while i > 0 repeat
      h := i rem p
      -- index(p) = 0$R
      if h ^= 0 then
        -- here was a bug: "index" instead of
        -- "coerce", otherwise it wouldn't work for
        -- Rings R where "coerce: I-> R" is not surjective
        a := index(h :: PositiveInteger)$R
        ans := ans + reduce monomial(a, j)
      i := i quo p
    ans
  lookup(z : $) : PositiveInteger ==
    -- z = index lookup z, n = lookup index n
    -- the answer is merely the Horner evaluation of the
    -- representation with the size of R (as integers).
    zero?(z) => size()$$ pretend PositiveInteger
    p : Integer := size()$R
    co : Integer := lookup(leadingCoefficient z)$R
    n : NonNegativeInteger := degree(z)
    while not zero?(z := reductum z) repeat
      co := co * p ** ((n - (n := degree z)) pretend
        NonNegativeInteger) + lookup(leadingCoefficient z)$R
    n = 0 => co pretend PositiveInteger
    (co * p ** n) pretend PositiveInteger

--
-- KA:=BasicPolynomialFunctions(Poly)
-- minPoly(x) ==
--   ffe:= SqFr(resultant(M::KA, KA.var - lift(x)::KA)).fs.first
--   ffe.flag = "SQFR" => ffe.f
--   mdeg:= (degree(ffe.f) // K.characteristic)::Integer
--   mat:= Zero()::Matrix<mdeg+1,deg+mdeg+1>(K)
--   xi:=L.1;  setelt(mat,1,1,K.1);  setelt(mat,1,(deg+1),K.1)
--   for i in 1..mdeg repeat
--     xi:= x * xi;  xp:= lift(xi)
--     while xp ^= KA.0 repeat

```

```

--      setelt(mat,(mdeg+1),(degree(xp)+1),LeadingCoef(xp))
--      xp:=reductum(xp)
--      setelt(mat,(mdeg+1),(deg+i+1),K.1)
--      EchelonLastRow(mat)
--      if and/(elt(mat,(i+1),j) = K.0 for j in 1..deg)
--      then return unitNormal(+/(elt(mat,(i+1),(deg+j+1))*(B::KA)**j
--                               for j in 0..i)).a
--      ffe.f

```

— SAE.dotabb —

```

"SAE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SAE"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"SAE" -> "MONOGEN"

```

domain SFORT SimpleFortranProgram

— SimpleFortranProgram.input —

```

)set break resume
)sys rm -f SimpleFortranProgram.output
)spool SimpleFortranProgram.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SimpleFortranProgram
--R
--R SimpleFortranProgram(R: OrderedSet,FS: FunctionSpace(R)) is a domain constructor
--R Abbreviation for SimpleFortranProgram is SFORT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SFORT
--R
--R----- Operations -----
--R coerce : % -> OutputForm          outputAsFortran : % -> Void
--R fortran : (Symbol,FortranScalarType,FS) -> %
--R
--E 1

```

```
)spool
)lisp (bye)
```

— SimpleFortranProgram.help —

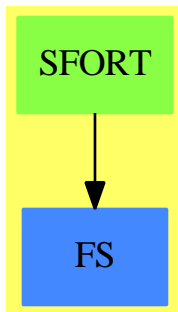
```
=====
SimpleFortranProgram examples
=====
```

SimpleFortranProgram(f,type) provides a simple model of some FORTRAN subprograms, making it possible to coerce objects of various domains into a FORTRAN subprogram called f. These can then be translated into legal FORTRAN code.

See Also:

o)show SimpleFortranProgram

SimpleFortranProgram (SFORT)



See

- ⇒ “Result” (RESULT) [19](#) on page [2300](#)
- ⇒ “FortranCode” (FC) [7](#) on page [931](#)
- ⇒ “FortranProgram” (FORTRAN) [7](#) on page [955](#)
- ⇒ “ThreeDimensionalMatrix” (M3D) [21](#) on page [2702](#)
- ⇒ “Switch” (SWITCH) [20](#) on page [2629](#)
- ⇒ “FortranTemplate” (FTEM) [7](#) on page [967](#)
- ⇒ “FortranExpression” (FEXPR) [7](#) on page [946](#)

Exports:

coerce fortran outputAsFortran

— domain SFORT SimpleFortranProgram —

```

)abbrev domain SFORT SimpleFortranProgram
-- Because of a bug in the compiler:
--)bo $noSubsumption:=true

++ Author: Mike Dewar
++ Date Created: November 1992
++ Description:
++ \axiomType{SimpleFortranProgram(f,type)} provides a simple model of some
++ FORTRAN subprograms, making it possible to coerce objects of various
++ domains into a FORTRAN subprogram called \axiom{f}.
++ These can then be translated into legal FORTRAN code.

SimpleFortranProgram(R,FS): Exports == Implementation where
  R : OrderedSet
  FS : FunctionSpace(R)

FST ==> FortranScalarType

Exports ==> FortranProgramCategory with
  fortran : (Symbol,FST,FS) -> $
  ++fortran(fname,ftype,body) builds an object of type
  ++\axiomType{FortranProgramCategory}. The three arguments specify
  ++the name, the type and the body of the program.

Implementation ==> add

Rep := Record(name : Symbol, type : FST, body : FS )

fortran(fname, ftype, res) ==
  construct(fname,ftype,res)$Rep

nameOf(u:$):Symbol == u . name

typeOf(u:$):Union(FST,"void") == u . type

bodyOf(u:$):FS == u . body

argumentsOf(u:$):List Symbol == variables(bodyOf u)$FS

coerce(u:$):OutputForm ==
  coerce(nameOf u)$Symbol

outputAsFortran(u:$):Void ==
  ftype := (checkType(typeOf(u)::OutputForm)$Lisp)::OutputForm
  fname := nameOf(u)::OutputForm
  args := argumentsOf(u)
  nargs:=args::OutputForm

```

```

val := bodyOf(u)::OutputForm
fortFormatHead(ftype,fname,nargs)$Lisp
fortFormatTypes(ftype,args)$Lisp
dispfortexp1$Lisp ["="::OutputForm, fname, val]@List(OutputForm)
dispfortexp1$Lisp "RETURN"::OutputForm
dispfortexp1$Lisp "END"::OutputForm
void()$Void

```

— SFORT.dotabb —

```

"SFORT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SFORT"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"SFORT" -> "FS"

```

domain SINT SingleInteger

This type represents finite (machine) precision integers which wrap around at a machine dependent value. However, **SingleInteger** has type **EntireRing** which should have no non-zero divisors. But

$$(2 * 16 : SINT) ** 2 = 0$$

on a 32 bit machine. Using **SingleInteger** is a pragmatic efficiency choice. In this condition, Axiom will complain that the result is not of type FIXNUM, which is the Common Lisp type for machine integers.

— SingleInteger.input —

```

)set break resume
)sys rm -f SingleInteger.output
)spool SingleInteger.output
)set message test on
)set message auto off
)clear all
--S 1 of 11
min()$SingleInteger
--R
--R
--R (1) - 2147483648
--R
--E 1

```

Type: SingleInteger

```

--S 2 of 11
max()$SingleInteger
--R
--R
--R (2) 2147483647
--R
--R Type: SingleInteger
--E 2

--S 3 of 11
a := 1234 :: SingleInteger
--R
--R
--R (3) 1234
--R
--R Type: SingleInteger
--E 3

--S 4 of 11
b := 124$SingleInteger
--R
--R
--R (4) 124
--R
--R Type: SingleInteger
--E 4

--S 5 of 11
gcd(a,b)
--R
--R
--R (5) 2
--R
--R Type: SingleInteger
--E 5

--S 6 of 11
lcm(a,b)
--R
--R
--R (6) 76508
--R
--R Type: SingleInteger
--E 6

--S 7 of 11
mulmod(5,6,13)$SingleInteger
--R
--R
--R (7) 4
--R
--R Type: SingleInteger
--E 7

--S 8 of 11

```

```

positiveRemainder(37,13)$SingleInteger
--R
--R
--R   (8)  11
--R
--R                                          Type: SingleInteger
--E 8

--S 9 of 11
And(3,4)$SingleInteger
--R
--R
--R   (9)  0
--R
--R                                          Type: SingleInteger
--E 9

--S 10 of 11
shift(1,4)$SingleInteger
--R
--R
--R   (10)  16
--R
--R                                          Type: SingleInteger
--E 10

--S 11 of 11
shift(31,-1)$SingleInteger
--R
--R
--R   (11)  15
--R
--R                                          Type: SingleInteger
--E 11
)spool
)lisp (bye)

```

— SingleInteger.help —

```

=====
SingleInteger examples
=====

```

The SingleInteger domain is intended to provide support in Axiom for machine integer arithmetic. It is generally much faster than (bignum) Integer arithmetic but suffers from a limited range of values. Since Axiom can be implemented on top of various dialects of Lisp, the actual representation of small integers may not correspond exactly to the host machines integer representation.

You can discover the minimum and maximum values in your implementation

by using min and max.

```
min()$SingleInteger
- 2147483648
Type: SingleInteger
```

```
max()$SingleInteger
2147483647
Type: SingleInteger
```

To avoid confusion with Integer, which is the default type for integers, you usually need to work with declared variables.

```
a := 1234 :: SingleInteger
1234
Type: SingleInteger
```

or use package calling

```
b := 124$SingleInteger
124
Type: SingleInteger
```

You can add, multiply and subtract SingleInteger objects, and ask for the greatest common divisor (gcd).

```
gcd(a,b)
2
Type: SingleInteger
```

The least common multiple (lcm) is also available.

```
lcm(a,b)
76508
Type: SingleInteger
```

Operations mulmod, addmod, submod, and invmod are similar - they provide arithmetic modulo a given small integer.

Here is $5 * 6 \bmod 13$.

```
mulmod(5,6,13)$SingleInteger
4
Type: SingleInteger
```

To reduce a small integer modulo a prime, use positiveRemainder.

```
positiveRemainder(37,13)$SingleInteger
11
Type: SingleInteger
```

Operations And, Or, xor, and Not provide bit level operations on small integers.

```
And(3,4)$SingleInteger
0
```

Type: SingleInteger

Use shift(int,numToShift) to shift bits, where i is shifted left if numToShift is positive, right if negative.

```
shift(1,4)$SingleInteger
16
```

Type: SingleInteger

```
shift(31,-1)$SingleInteger
15
```

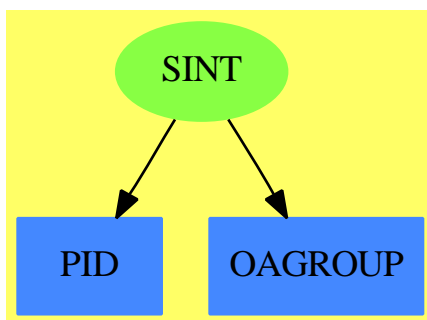
Type: SingleInteger

Many other operations are available for small integers, including many of those provided for Integer.

See Also:

- o)help Integer
- o)show SingleInteger

SingleInteger (SINT)



Exports:

0	1	abs	addmod
And	associates?	base	binomial
bit?	characteristic	coerce	convert
copy	D	dec	differentiate
divide	euclideanSize	even?	expressIdealMember
exquo	extendedEuclidean	factor	factorial
gcd	gcdPolynomial	hash	inc
init	invmod	latex	lcm
length	mask	max	min
mulmod	multiEuclidean	negative?	nextItem
Not	not?	odd?	OMwrite
one?	Or	patternMatch	permutation
principalIdeal	positive?	positiveRemainder	powmod
prime?	random	rational	rationalIfCan
rational?	recip	reducedSystem	retract
retractIfCan	sample	shift	sign
sizeLess?	squareFree	squareFreePart	subtractIfCan
submod	symmetricRemainder	unit?	unitCanonical
unitNormal	xor	zero?	?*?
?**?	?+?	?-?	-?
?/\?	?<?	?<=?	?=?
?>?	?>=?	?\/?	?^?
?	?~=?	?quo?	?rem?

— domain SINT SingleInteger —

```

)abbrev domain SINT SingleInteger

-- following patch needed to deal with *(I,%) -> %
-- affects behavior of SourceLevelSubset
--)bo $noSubsets := true
-- No longer - JHD !! still needed 5/3/91 BMT

++ Author: Michael Monagan
++ Date Created: January 1988
++ Change History:
++ Description:
++ SingleInteger is intended to support machine integer arithmetic.

-- MAXINT, BASE (machine integer constants)
-- MODULUS, MULTIPLIER (random number generator constants)

-- Lisp dependencies
-- EQ, ABSVAL, TIMES, INTEGER-LENGTH, SXHASH, REMAINDER
-- QSLESSP, QSGREATERP, QSADD1, QSSUB1, QSMINUS, QSPLUS, QSDIFFERENCE
-- QSTIMES, QSREMAINDER, QSODDP, QSZEROP, QSMAX, QSMIN, QSNOT, QSAND

```

```
-- QSOR, QSXOR, QSLEFTSHIFT, QSADDMOD, QSDIFMOD, QSMULTMOD
```

```
SingleInteger(): Join(IntegerNumberSystem,Logic,OpenMath) with
  canonical
    ++ \spad{canonical} means that mathematical equality is
    ++ implied by data structure equality.
  canonicalsClosed
    ++ \spad{canonicalClosed} means two positives multiply to
    ++ give positive.
  noetherian
    ++ \spad{noetherian} all ideals are finitely generated
    ++ (in fact principal).

max      : () -> %
  ++ max() returns the largest single integer.
min      : () -> %
  ++ min() returns the smallest single integer.

-- bit operations
"not": % -> %
  ++ not(n) returns the bit-by-bit logical not of the single integer n.
"~" : % -> %
  ++ ~ n returns the bit-by-bit logical not of the single integer n.
"/\" : (% , %) -> %
  ++ n /\ m returns the bit-by-bit logical and of
  ++ the single integers n and m.
"\/" : (% , %) -> %
  ++ n \/ m returns the bit-by-bit logical or of
  ++ the single integers n and m.
"xor": (% , %) -> %
  ++ xor(n,m) returns the bit-by-bit logical xor of
  ++ the single integers n and m.
Not : % -> %
  ++ Not(n) returns the bit-by-bit logical not of the single integer n.
And : (% , %) -> %
  ++ And(n,m) returns the bit-by-bit logical and of
  ++ the single integers n and m.
Or : (% , %) -> %
  ++ Or(n,m) returns the bit-by-bit logical or of
  ++ the single integers n and m.

== add

seed : % := 1$Lisp -- for random()
MAXINT ==> MOST_-POSITIVE_-FIXNUM$Lisp
MININT ==> MOST_-NEGATIVE_-FIXNUM$Lisp
BASE ==> 67108864$Lisp -- 2**26
MULTIPLIER ==> 314159269$Lisp -- from Knuth's table
MODULUS ==> 2147483647$Lisp -- 2**31-1
```



```

writeOMSingleInt(dev: OpenMathDevice, x: %): Void ==
  if x < 0 then
    OMputApp(dev)
    OMputSymbol(dev, "arith1", "unary_minus")
    OMputInteger(dev, convert(-x))
    OMputEndApp(dev)
  else
    OMputInteger(dev, convert(x))

OMwrite(x: %): String ==
  s: String := ""
  sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
  dev: OpenMathDevice := OMopenString(sp @ String, OMencodingXML)
  OMputObject(dev)
  writeOMSingleInt(dev, x)
  OMputEndObject(dev)
  OMclose(dev)
  s := OM_-STRINGPTRTOSTRING(sp)$Lisp @ String
  s

OMwrite(x: %, wholeObj: Boolean): String ==
  s: String := ""
  sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
  dev: OpenMathDevice := OMopenString(sp @ String, OMencodingXML)
  if wholeObj then
    OMputObject(dev)
  writeOMSingleInt(dev, x)
  if wholeObj then
    OMputEndObject(dev)
  OMclose(dev)
  s := OM_-STRINGPTRTOSTRING(sp)$Lisp @ String
  s

OMwrite(dev: OpenMathDevice, x: %): Void ==
  OMputObject(dev)
  writeOMSingleInt(dev, x)
  OMputEndObject(dev)

OMwrite(dev: OpenMathDevice, x: %, wholeObj: Boolean): Void ==
  if wholeObj then
    OMputObject(dev)
  writeOMSingleInt(dev, x)
  if wholeObj then
    OMputEndObject(dev)

reducedSystem m      == m pretend Matrix(Integer)
coerce(x):OutputForm == (convert(x)@Integer)::OutputForm
convert(x:%):Integer == x pretend Integer
i:Integer * y:%      == i::% * y

```

```

0          == 0$Lisp
1          == 1$Lisp
base()     == 2$Lisp
max()      == MAXINT
min()      == MININT
x = y      == EQL(x,y)$Lisp
_~ x       == LOGNOT(x)$Lisp
not(x)     == LOGNOT(x)$Lisp
_/_\ (x,y) == LOGAND(x,y)$Lisp
_\_/(x,y)  == LOGIOR(x,y)$Lisp
Not(x)     == LOGNOT(x)$Lisp
And(x,y)   == LOGAND(x,y)$Lisp
Or(x,y)    == LOGIOR(x,y)$Lisp
xor(x,y)   == LOGXOR(x,y)$Lisp
x < y      == QSLESSP(x,y)$Lisp
inc x      == QSADD1(x)$Lisp
dec x      == QSSUB1(x)$Lisp
- x        == QSMINUS(x)$Lisp
x + y      == QSPLUS(x,y)$Lisp
x:% - y:%  == QSDIFFERENCE(x,y)$Lisp
x:% * y:%  == QSTIMES(x,y)$Lisp
x:% ** n:NonNegativeInteger == ((EXPT(x, n)$Lisp) @ Integer)::%
x quo y    == QSQUOTIENT(x,y)$Lisp
x rem y    == QSREMAINDER(x,y)$Lisp
divide(x, y) == CONS(QSQUOTIENT(x,y)$Lisp, QSREMAINDER(x,y)$Lisp)$Lisp
gcd(x,y)   == GCD(x,y)$Lisp
abs(x)     == QSABSVAL(x)$Lisp
odd?(x)    == QSODDP(x)$Lisp
zero?(x)   == QSZEROP(x)$Lisp
-- one?(x) == ONEP(x)$Lisp
one?(x)    == x = 1
max(x,y)   == QSMAX(x,y)$Lisp
min(x,y)   == QSMIN(x,y)$Lisp
hash(x)    == SXHASH(x)$Lisp
length(x)  == INTEGER_-LENGTH(x)$Lisp
shift(x,n) == QSLEFTSHIFT(x,n)$Lisp
mulmod(a,b,p) == QSMULTMOD(a,b,p)$Lisp
addmod(a,b,p) == QSADDMOD(a,b,p)$Lisp
submod(a,b,p) == QSDIFMOD(a,b,p)$Lisp
negative?(x) == QSMINUSP$Lisp x

reducedSystem(m, v) ==
  [m pretend Matrix(Integer), v pretend Vector(Integer)]

positiveRemainder(x,n) ==
  r := QSREMAINDER(x,n)$Lisp
  QSMINUSP(r)$Lisp =>
    QSMINUSP(n)$Lisp => QSDIFFERENCE(x, n)$Lisp
    QSPLUS(r, n)$Lisp

```

```

r

coerce(x:Integer):% ==
  (x <= max pretend Integer) and (x >= min pretend Integer) =>
    x pretend %
  error "integer too large to represent in a machine word"

random() ==
  seed := REMAINDER(TIMES(MULTIPLIER,seed)$Lisp,MODULUS)$Lisp
  REMAINDER(seed,BASE)$Lisp

random(n) == RANDOM(n)$Lisp

UCA ==> Record(unit:%,canonical:%,associate:%)
unitNormal x ==
  x < 0 => [-1,-x,-1]$UCA
  [1,x,1]$UCA

)bo $noSubsets := false

```

— SINT.dotabb —

```

"SINT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SINT",shape=ellipse]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"SINT" -> "PID"
"SINT" -> "OAGROUP"

```

domain SAOS SingletonAsOrderedSet

— SingletonAsOrderedSet.input —

```

)set break resume
)sys rm -f SingletonAsOrderedSet.output
)spool SingletonAsOrderedSet.output
)set message test on
)set message auto off
)clear all

```

--S 1 of 1

```

)show SingletonAsOrderedSet
--R SingletonAsOrderedSet  is a domain constructor
--R Abbreviation for SingletonAsOrderedSet is SAOS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SAOS
--R
--R----- Operations -----
--R ?<? : (%,% ) -> Boolean          ?<=? : (%,% ) -> Boolean
--R ?=? : (%,% ) -> Boolean          ?>? : (%,% ) -> Boolean
--R ?>=? : (%,% ) -> Boolean          coerce : % -> OutputForm
--R convert : % -> Symbol             create : () -> %
--R hash : % -> SingleInteger          latex : % -> String
--R max : (%,% ) -> %                 min : (%,% ) -> %
--R ?^=? : (%,% ) -> Boolean
--R
--E 1

)spool
)lisp (bye)

```

— SingletonAsOrderedSet.help —

=====

SingletonAsOrderedSet examples

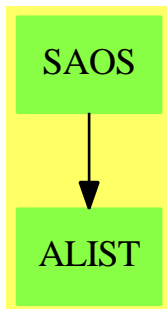
=====

This trivial domain lets us build Univariate Polynomials
in an anonymous variable

See Also:

o)show SingletonAsOrderedSet

SingletonAsOrderedSet (SAOS)



Exports:

```

coerce  convert  create  hash  latex
max      min      ?~=?   ?<?   ?<=?
?=?      ?>?      ?>=?

```

— domain SAOS SingletonAsOrderedSet —

```

)abbrev domain SAOS SingletonAsOrderedSet
++ Author: Mark Botch
++ Description:
++ This trivial domain lets us build Univariate Polynomials
++ in an anonymous variable

SingletonAsOrderedSet(): OrderedSet with
    create:() -> %
    convert:% -> Symbol

== add
create() == "?" pretend %
a<b == false -- only one element
coerce(a) == outputForm "?" -- CJW doesn't like this: change ?
a=b == true -- only one element
min(a,b) == a -- only one element
max(a,b) == a -- only one element
convert a == coerce("?")

```

— SAOS.dotabb —

```

"SAOS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SAOS"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"SAOS" -> "ALIST"

```

domain SMP SparseMultivariatePolynomial

— SparseMultivariatePolynomial.input —

```
)set break resume
)sys rm -f SparseMultivariatePolynomial.output
)spool SparseMultivariatePolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SparseMultivariatePolynomial
--R
--R SparseMultivariatePolynomial(R: Ring,VarSet: OrderedSet) is a domain constructor
--R Abbreviation for SparseMultivariatePolynomial is SMP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SMP
--R
--R----- Operations -----
--R ?? : (%,R) -> %           ?? : (R,%) -> %
--R ?? : (%,%) -> %           ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %           ?-? : (%,%) -> %
--R -? : % -> %               ?=? : (%,%) -> Boolean
--R D : (%,List(VarSet)) -> %     D : (%,VarSet) -> %
--R 1 : () -> %                 0 : () -> %
--R ?? : (%,PositiveInteger) -> % coefficients : % -> List(R)
--R coerce : VarSet -> %         coerce : R -> %
--R coerce : Integer -> %        coerce : % -> OutputForm
--R differentiate : (%,VarSet) -> % eval : (%,VarSet,%) -> %
--R eval : (%,VarSet,R) -> %     eval : (%,%,%) -> %
--R eval : (%,Equation(%)) -> % ground : % -> R
--R ground? : % -> Boolean       hash : % -> SingleInteger
--R latex : % -> String         leadingCoefficient : % -> R
--R leadingMonomial : % -> %     map : ((R -> R),%) -> %
--R monomial? : % -> Boolean     monomials : % -> List(%)
--R one? : % -> Boolean         recip : % -> Union(%, "failed")
--R reductum : % -> %           retract : % -> VarSet
--R retract : % -> R            sample : () -> %
--R variables : % -> List(VarSet) zero? : % -> Boolean
--R ~=? : (%,%) -> Boolean
--R ?? : (Fraction(Integer),%) -> % if R has ALGEBRA(FRAC(INT))
--R ?? : (%,Fraction(Integer)) -> % if R has ALGEBRA(FRAC(INT))
--R ?? : (NonNegativeInteger,%) -> %
```

```

--R ??? : (% , NonNegativeInteger) -> %
--R ?/? : (% , R) -> % if R has FIELD
--R ?<? : (% , %) -> Boolean if R has ORDSET
--R ?<=? : (% , %) -> Boolean if R has ORDSET
--R ?>? : (% , %) -> Boolean if R has ORDSET
--R ?>=? : (% , %) -> Boolean if R has ORDSET
--R D : (% , List(VarSet), List(NonNegativeInteger)) -> %
--R D : (% , VarSet, NonNegativeInteger) -> %
--R ?^? : (% , NonNegativeInteger) -> %
--R associates? : (% , %) -> Boolean if R has INTDOM
--R binomThmExpt : (% , %, NonNegativeInteger) -> % if R has COMRING
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(% , "failed") if $ has CHARNZ and R has PFECAT or R has CHARNZ
--R coefficient : (% , List(VarSet), List(NonNegativeInteger)) -> %
--R coefficient : (% , VarSet, NonNegativeInteger) -> %
--R coefficient : (% , IndexedExponents(VarSet)) -> R
--R coerce : Fraction(Integer) -> % if R has ALGEBRA(FRAC(INT)) or R has RETRACT(FRAC(INT))
--R coerce : % -> % if R has INTDOM
--R conditionP : Matrix(%) -> Union(Vector(%), "failed") if $ has CHARNZ and R has PFECAT
--R content : (% , VarSet) -> % if R has GCDDOM
--R content : % -> R if R has GCDDOM
--R convert : % -> InputForm if R has KONVERT(INFORM) and VarSet has KONVERT(INFORM)
--R convert : % -> Pattern(Integer) if R has KONVERT(PATTERN(INT)) and VarSet has KONVERT(PATTER
--R convert : % -> Pattern(Float) if R has KONVERT(PATTERN(FLOAT)) and VarSet has KONVERT(PATTER
--R degree : (% , List(VarSet)) -> List(NonNegativeInteger)
--R degree : (% , VarSet) -> NonNegativeInteger
--R degree : % -> IndexedExponents(VarSet)
--R differentiate : (% , List(VarSet), List(NonNegativeInteger)) -> %
--R differentiate : (% , VarSet, NonNegativeInteger) -> %
--R differentiate : (% , List(VarSet)) -> %
--R discriminant : (% , VarSet) -> % if R has COMRING
--R eval : (% , List(VarSet), List(%)) -> %
--R eval : (% , List(VarSet), List(R)) -> %
--R eval : (% , List(%), List(%)) -> %
--R eval : (% , List(Equation(%))) -> %
--R exquo : (% , %) -> Union(% , "failed") if R has INTDOM
--R exquo : (% , R) -> Union(% , "failed") if R has INTDOM
--R factor : % -> Factored(%) if R has PFECAT
--R factorPolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%))
--R factorSquareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolyn
--R gcd : (% , %) -> % if R has GCDDOM
--R gcd : List(%) -> % if R has GCDDOM
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUniva
--R isExpt : % -> Union(Record(var: VarSet, exponent: NonNegativeInteger), "failed")
--R isPlus : % -> Union(List(%), "failed")
--R isTimes : % -> Union(List(%), "failed")
--R lcm : (% , %) -> % if R has GCDDOM
--R lcm : List(%) -> % if R has GCDDOM
--R mainVariable : % -> Union(VarSet, "failed")
--R mapExponents : ((IndexedExponents(VarSet) -> IndexedExponents(VarSet)), %) -> %

```

```

--R max : (% ,%) -> % if R has ORDSET
--R min : (% ,%) -> % if R has ORDSET
--R minimumDegree : (% ,List(VarSet)) -> List(NonNegativeInteger)
--R minimumDegree : (% ,VarSet) -> NonNegativeInteger
--R minimumDegree : % -> IndexedExponents(VarSet)
--R monicDivide : (% ,%,VarSet) -> Record(quotient: %,remainder: %)
--R monomial : (% ,List(VarSet),List(NonNegativeInteger)) -> %
--R monomial : (% ,VarSet,NonNegativeInteger) -> %
--R monomial : (R,IndexedExponents(VarSet)) -> %
--R multivariate : (SparseUnivariatePolynomial(%),VarSet) -> %
--R multivariate : (SparseUnivariatePolynomial(R),VarSet) -> %
--R numberOfMonomials : % -> NonNegativeInteger
--R patternMatch : (% ,Pattern(Integer),PatternMatchResult(Integer,%)) -> PatternMatchResult(Integer,%) if R
--R patternMatch : (% ,Pattern(Float),PatternMatchResult(Float,%)) -> PatternMatchResult(Float,%) if R has P
--R pomopo! : (% ,R,IndexedExponents(VarSet),%) -> %
--R prime? : % -> Boolean if R has PFECAT
--R primitiveMonomials : % -> List(%)
--R primitivePart : (% ,VarSet) -> % if R has GCDDOM
--R primitivePart : % -> % if R has GCDDOM
--R reducedSystem : Matrix(%) -> Matrix(R)
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(R),vec: Vector(R))
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) if R has LIN
--R reducedSystem : Matrix(%) -> Matrix(Integer) if R has LINEXP(INT)
--R resultant : (% ,%,VarSet) -> % if R has COMRING
--R retract : % -> Integer if R has RETRACT(INT)
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(VarSet,"failed")
--R retractIfCan : % -> Union(Integer,"failed") if R has RETRACT(INT)
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(R,"failed")
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(%)),SparseUnivariatePolynomial(%)) -> U
--R squareFree : % -> Factored(%) if R has GCDDOM
--R squareFreePart : % -> % if R has GCDDOM
--R squareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if R has
--R subtractIfCan : (% ,%) -> Union(%,"failed")
--R totalDegree : (% ,List(VarSet)) -> NonNegativeInteger
--R totalDegree : % -> NonNegativeInteger
--R unit? : % -> Boolean if R has INTDOM
--R unitCanonical : % -> % if R has INTDOM
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %) if R has INTDOM
--R univariate : % -> SparseUnivariatePolynomial(R)
--R univariate : (% ,VarSet) -> SparseUnivariatePolynomial(%)
--R
--E 1

```

```

)spool
)lisp (bye)

```


— SparseMultivariatePolynomial.help —

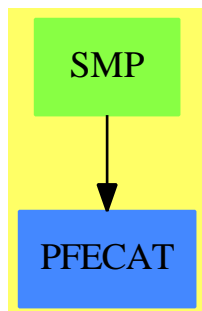
```
=====
SparseMultivariatePolynomial examples
=====
```

This type is the basic representation of sparse recursive multivariate polynomials. It is parameterized by the coefficient ring and the variable set which may be infinite. The variable ordering is determined by the variable set parameter. The coefficient ring may be non-commutative, but the variables are assumed to commute.

See Also:

```
o )show SparseMultivariatePolynomial
```

SparseMultivariatePolynomial (SMP)



See

- ⇒ “Polynomial” (POLY) [17](#) on page [2077](#)
- ⇒ “MultivariatePolynomial” (MPOLY) [14](#) on page [1681](#)
- ⇒ “IndexedExponents” (INDE) [10](#) on page [1217](#)

Exports:

0	1	associates?
binomThmExpt	characteristic	charthRoot
coefficient	coefficients	coerce
conditionP	content	convert
D	degree	differentiate
discriminant	eval	exquo
factor	factorPolynomial	factorSquareFreePolynomial
gcd	gcdPolynomial	ground
ground?	hash	latex
isExpt	isPlus	isTimes
lcm	leadingCoefficient	leadingMonomial
mainVariable	map	mapExponents
max	min	minimumDegree
monicDivide	monomial	monomial?
monomials	multivariate	numberOfMonomials
one?	patternMatch	pomopo!
prime?	primitivePart	primitiveMonomials
recip	reducedSystem	reductum
resultant	retract	retractIfCan
sample	solveLinearPolynomialEquation	squareFree
squareFreePart	squareFreePolynomial	subtractIfCan
totalDegree	unit?	unitCanonical
unitNormal	univariate	variables
zero?	?*?	***?
?+?	?-?	-?
?=?	?^?	?~=?
?/?	?<?	?<=?
?>?	?>=?	

— domain SMP SparseMultivariatePolynomial —

```

)abbrev domain SMP SparseMultivariatePolynomial
++ Author: Dave Barton, Barry Trager
++ Date Last Updated: 30 November 1994
++ Fix History: 30 Nov 94
++ Description:
++ This type is the basic representation of sparse recursive multivariate
++ polynomials. It is parameterized by the coefficient ring and the
++ variable set which may be infinite. The variable ordering is determined
++ by the variable set parameter. The coefficient ring may be non-commutative,
++ but the variables are assumed to commute.

SparseMultivariatePolynomial(R: Ring, VarSet: OrderedSet): C == T where
  pgcd ==> PolynomialGcdPackage(IndexedExponents VarSet, VarSet, R, %)
  C == PolynomialCategory(R, IndexedExponents(VarSet), VarSet)
  SUP ==> SparseUnivariatePolynomial

```

```

T == add
--constants
--D := F(%) replaced by next line until compiler support completed

--representations
D := SparseUnivariatePolynomial(%)
VPoly:= Record(v:VarSet,ts:D)
Rep:= Union(R,VPoly)

--local function

--declarations
fn: R -> R
n: Integer
k: NonNegativeInteger
kp:PositiveInteger
k1:NonNegativeInteger
c: R
mvar: VarSet
val : R
var:VarSet
up: D
p,p1,p2,pval: %
Lval : List(R)
Lpval : List(%)
Lvar : List(VarSet)

--define
0 == 0$R::%
1 == 1$R::%

zero? p == p case R and zero?(p)$R
-- one? p == p case R and one?(p)$R
one? p == p case R and ((p) = 1)$R
-- a local function
red(p:~):~ ==
  p case R => 0
  if ground?(reductum p.ts) then
    leadingCoefficient(reductum p.ts) else [p.v,reductum p.ts]$VPoly

numberOfMonomials(p): NonNegativeInteger ==
  p case R =>
    zero?(p)$R => 0
    1
    +/[numberOfMonomials q for q in coefficients(p.ts)]

coerce(mvar):~ == [mvar,monomial(1,1)$D]$VPoly

```

```

monomial? p ==
  p case R => true
  sup : D := p.ts
  1 ^= numberOfMonomials(sup) => false
  monomial? leadingCoefficient(sup)$D

-- local
moreThanOneVariable?: % -> Boolean

moreThanOneVariable? p ==
  p case R => false
  q:=p.ts
  any?(x1+-->not ground? x1 ,coefficients q) => true
  false

-- if we already know we use this (slighlty) faster function
univariateKnown: % -> SparseUnivariatePolynomial R

univariateKnown p ==
  p case R => (leadingCoefficient p) :: SparseUnivariatePolynomial(R)
  monomial( leadingCoefficient p,degree p.ts)+ univariateKnown(red p)

univariate p ==
  p case R =>(leadingCoefficient p) :: SparseUnivariatePolynomial(R)
  moreThanOneVariable? p => error "not univariate"
  monomial( leadingCoefficient p,degree p.ts)+ univariate(red p)

multivariate (u: SparseUnivariatePolynomial(R),var: VarSet) ==
  ground? u => (leadingCoefficient u) ::%
  [var,monomial(leadingCoefficient u,degree u)$D]$VPoly +
  multivariate(reductum u,var)

univariate(p:%,mvar: VarSet): SparseUnivariatePolynomial(%) ==
  p case R or mvar>p.v => monomial(p,0)$D
  pt:=p.ts
  mvar=p.v => pt
  monomial(1,p.v,degree pt)*univariate(leadingCoefficient pt,mvar)+
  univariate(red p,mvar)

-- a local functions, used in next definition
unlikeUnivReconstruct(u: SparseUnivariatePolynomial(%),mvar: VarSet):% ==
  zero? (d:=degree u) => coefficient(u,0)
  monomial(leadingCoefficient u,mvar,d)+
  unlikeUnivReconstruct(reductum u,mvar)

multivariate(u: SparseUnivariatePolynomial(%),mvar: VarSet):% ==
  ground? u => coefficient(u,0)
  uu:=u
  while not zero? uu repeat
    cc:=leadingCoefficient uu

```

```

    cc case R or mvar > cc.v => uu:=reductum uu
    return unlikeUnivReconstruct(u,mvar)
[mvar,u]$VPoly

ground?(p:%):Boolean ==
  p case R => true
  false

--      const p ==
--      p case R => p
--      error "the polynomial is not a constant"

monomial(p,mvar,k1) ==
  zero? k1 or zero? p => p
  p case R or mvar>p.v => [mvar,monomial(p,k1)$D]$VPoly
  p*[mvar,monomial(1,k1)$D]$VPoly

monomial(c:R,e:IndexedExponents(VarSet)):% ==
  zero? e => (c::%)
  monomial(1,leadingSupport e, leadingCoefficient e) *
    monomial(c,reductum e)

coefficient(p:%, e:IndexedExponents(VarSet)) : R ==
  zero? e =>
    p case R => p::R
    coefficient(coefficient(p.ts,0),e)
  p case R => 0
  ve := leadingSupport e
  vp := p.v
  ve < vp =>
    coefficient(coefficient(p.ts,0),e)
  ve > vp => 0
  coefficient(coefficient(p.ts,leadingCoefficient e),reductum e)

--      coerce(e:IndexedExponents(VarSet)) : % ==
--      e = 0 => 1
--      monomial(1,leadingSupport e, leadingCoefficient e) *
--      (reductum e)::%

--      retract(p:%):IndexedExponents(VarSet) ==
--      q:Union(IndexedExponents(VarSet),"failed"):=retractIfCan p
--      q :: IndexedExponents(VarSet)

--      retractIfCan(p:%):Union(IndexedExponents(VarSet),"failed") ==
--      p = 0 => degree p
--      reductum(p)=0 and leadingCoefficient(p)=1 => degree p
--      "failed"

coerce(n) == n::R::%
coerce(c) == c::%
```

```

characteristic == characteristic$R

recip(p) ==
  p case R => (uu:=recip(p::R);uu case "failed" => "failed"; uu::%)
  "failed"

- p ==
  p case R => -$R p
  [p.v, - p.ts]$VPoly
n * p ==
  p case R => n * p::R
  mvar:=p.v
  up:=n*p.ts
  if ground? up then leadingCoefficient(up) else [mvar,up]$VPoly
c * p ==
  c = 1 => p
  p case R => c * p::R
  mvar:=p.v
  up:=c*p.ts
  if ground? up then leadingCoefficient(up) else [mvar,up]$VPoly
p1 + p2 ==
  p1 case R and p2 case R => p1 +$R p2
  p1 case R => [p2.v, p1::D + p2.ts]$VPoly
  p2 case R => [p1.v, p1.ts + p2::D]$VPoly
  p1.v = p2.v =>
    mvar:=p1.v
    up:=p1.ts+p2.ts
    if ground? up then leadingCoefficient(up) else [mvar,up]$VPoly
  p1.v < p2.v =>
    [p2.v, p1::D + p2.ts]$VPoly
    [p1.v, p1.ts + p2::D]$VPoly

p1 - p2 ==
  p1 case R and p2 case R => p1 -$R p2
  p1 case R => [p2.v, p1::D - p2.ts]$VPoly
  p2 case R => [p1.v, p1.ts - p2::D]$VPoly
  p1.v = p2.v =>
    mvar:=p1.v
    up:=p1.ts-p2.ts
    if ground? up then leadingCoefficient(up) else [mvar,up]$VPoly
  p1.v < p2.v =>
    [p2.v, p1::D - p2.ts]$VPoly
    [p1.v, p1.ts - p2::D]$VPoly

p1 = p2 ==
  p1 case R =>
    p2 case R => p1 =$R p2
    false
  p2 case R => false
  p1.v = p2.v => p1.ts = p2.ts

```

```

false

p1 * p2 ==
  p1 case R => p1::R * p2
  p2 case R =>
    mvar:=p1.v
    up:=p1.ts*p2
    if ground? up then leadingCoefficient(up) else [mvar,up]$VPoly
  p1.v = p2.v =>
    mvar:=p1.v
    up:=p1.ts*p2.ts
    if ground? up then leadingCoefficient(up) else [mvar,up]$VPoly
  p1.v > p2.v =>
    mvar:=p1.v
    up:=p1.ts*p2
    if ground? up then leadingCoefficient(up) else [mvar,up]$VPoly
  --- p1.v < p2.v
  mvar:=p2.v
  up:=p1*p2.ts
  if ground? up then leadingCoefficient(up) else [mvar,up]$VPoly

p ^ kp == p ** (kp pretend NonNegativeInteger)
p ** kp == p ** (kp pretend NonNegativeInteger )
p ^ k == p ** k
p ** k ==
  p case R => p::R ** k
  -- univariate special case
  not moreThanOneVariable? p =>
    multivariate( (univariateKnown p) ** k , p.v)
  mvar:=p.v
  up:=p.ts ** k
  if ground? up then leadingCoefficient(up) else [mvar,up]$VPoly

if R has IntegralDomain then
  UnitCorrAssoc ==> Record(unit:%,canonical:%,associate:%)
  unitNormal(p) ==
    u,c,a:R
    p case R =>
      (u,c,a):= unitNormal(p::R)$R
      [u::%,c::%,a::%]$UnitCorrAssoc
      (u,c,a):= unitNormal(leadingCoefficient(p))$R
      [u::%,(a*p)::%,a::%]$UnitCorrAssoc
  unitCanonical(p) ==
    p case R => unitCanonical(p::R)$R
    (u,c,a):= unitNormal(leadingCoefficient(p))$R
    a*p
  unit? p ==
    p case R => unit?(p::R)$R
    false
  associates?(p1,p2) ==

```

```

p1 case R => p2 case R and associates?(p1,p2)$R
p2 case VPoly and p1.v = p2.v and associates?(p1.ts,p2.ts)

if R has approximate then
  p1 exquo p2 ==
    p1 case R and p2 case R =>
      a:= (p1::R exquo p2::R)
      if a case "failed" then "failed" else a::%
    zero? p1 => p1
    one? p2 => p1
    (p2 = 1) => p1
    p1 case R or p2 case VPoly and p1.v < p2.v => "failed"
    p2 case R or p1.v > p2.v =>
      a:= (p1.ts exquo p2::D)
      a case "failed" => "failed"
      [p1.v,a]$VPoly::%
    -- The next test is useful in the case that R has inexact
    -- arithmetic (in particular when it is Interval(...)).
    -- In the case where the test succeeds, empirical evidence
    -- suggests that it can speed up the computation several times,
    -- but in other cases where there are a lot of variables
    -- and p1 and p2 differ only in the low order terms (e.g. p1=p2+1)
    -- it slows exquo down by about 15-20%.
    p1 = p2 => 1
    a:= p1.ts exquo p2.ts
    a case "failed" => "failed"
    mvar:=p1.v
    up:SUP %:=a
    if ground? (up) then
      leadingCoefficient(up) else [mvar,up]$VPoly::%
  else
    p1 exquo p2 ==
      p1 case R and p2 case R =>
        a:= (p1::R exquo p2::R)
        if a case "failed" then "failed" else a::%
      zero? p1 => p1
      one? p2 => p1
      (p2 = 1) => p1
      p1 case R or p2 case VPoly and p1.v < p2.v => "failed"
      p2 case R or p1.v > p2.v =>
        a:= (p1.ts exquo p2::D)
        a case "failed" => "failed"
        [p1.v,a]$VPoly::%
      a:= p1.ts exquo p2.ts
      a case "failed" => "failed"
      mvar:=p1.v
      up:SUP %:=a
      if ground? up then leadingCoefficient(up) else [mvar,up]$VPoly::%

map(fn,p) ==

```



```

    p case R => fn(p)
    mvar:=p.v
    up:=map(x1+-->map(fn,x1),p.ts)
    if ground? up then leadingCoefficient(up) else [mvar,up]$VPoly

if R has Field then
  (p : %) / (r : R) == inv(r) * p

if R has GcdDomain then
  content(p) ==
    p case R => p
    c :R :=0
    up:=p.ts
--    while not(zero? up) and not(one? c) repeat
    while not(zero? up) and not(c = 1) repeat
      c:=gcd(c,content leadingCoefficient(up))
      up := reductum up
    c

if R has EuclideanDomain and
  R has CharacteristicZero and
  not(R has FloatingPointSystem) then

  content(p,mvar) ==
    p case R => p
    gcd(coefficients univariate(p,mvar))$pgcd

  gcd(p1,p2) == gcd(p1,p2)$pgcd

  gcd(lp:List %) == gcd(lp)$pgcd

  gcdPolynomial(a:SUP $,b:SUP $):SUP $ == gcd(a,b)$pgcd

else if R has GcdDomain then
  content(p,mvar) ==
    p case R => p
    content univariate(p,mvar)

  gcd(p1,p2) ==
    p1 case R =>
      p2 case R => gcd(p1,p2)$R::%
      zero? p1 => p2
      gcd(p1, content(p2.ts))
    p2 case R =>
      zero? p2 => p1
      gcd(p2, content(p1.ts))
    p1.v < p2.v => gcd(p1, content(p2.ts))
    p1.v > p2.v => gcd(content(p1.ts), p2)
    mvar:=p1.v
    up:=gcd(p1.ts, p2.ts)

```

```

    if ground? up then leadingCoefficient(up) else [mvar,up]$VPoly

if R has FloatingPointSystem then
  -- eventually need a better notion of gcd's over floats
  -- this essentially computes the gcds of the monomial contents
  gcdPolynomial(a:SUP $,b:SUP $):SUP $ ==
    ground? (a) =>
      zero? a => b
      gcd(leadingCoefficient a, content b)::SUP $
    ground?(b) =>
      zero? b => b
      gcd(leadingCoefficient b, content a)::SUP $
    conta := content a
    mona:SUP $ := monomial(conta, minimumDegree a)
    if mona ^= 1 then
      a := (a exquo mona)::SUP $
    contb := content b
    monb:SUP $ := monomial(contb, minimumDegree b)
    if monb ^= 1 then
      b := (b exquo monb)::SUP $
    mong:SUP $ := monomial(gcd(conta, contb),
                          min(degree mona, degree monb))
    degree(a) >= degree b =>
      not((a exquo b) case "failed") =>
        mong * b
      mong
    not((b exquo a) case "failed") => mong * a
    mong

coerce(p):OutputForm ==
  p case R => (p::R)::OutputForm
  outputForm(p.ts,p.v::OutputForm)

coefficients p ==
  p case R => list(p :: R)$List(R)
  "append"/[coefficients(p1)$% for p1 in coefficients(p.ts)]

retract(p:%):R ==
  p case R => p :: R
  error "cannot retract nonconstant polynomial"

retractIfCan(p:%):Union(R, "failed") ==
  p case R => p::R
  "failed"

-- leadingCoefficientRecursive(p:%):% ==
--   p case R => p
--   leadingCoefficient p.ts

mymerge:(List VarSet,List VarSet) ->List VarSet

```

```

mymerge(l:List VarSet,m:List VarSet):List VarSet ==
  empty? l => m
  empty? m => l
  first l = first m =>
    empty? rest l =>
      setrest!(l,rest m)
    l
    empty? rest m => l
    setrest!(l, mymerge(rest l, rest m))
  l
  first l > first m =>
    empty? rest l =>
      setrest!(l,m)
    l
    setrest!(l, mymerge(rest l, m))
  l
  empty? rest m =>
    setrest!(m,l)
  m
  setrest!(m,mymerge(l,rest m))
  m

variables p ==
  p case R => empty()
  lv:List VarSet:=empty()
  q := p.ts
  while not zero? q repeat
    lv:=mymerge(lv,variables leadingCoefficient q)
    q := reductum q
  cons(p.v,lv)

mainVariable p ==
  p case R => "failed"
  p.v

eval(p,mvar,pval) == univariate(p,mvar)(pval)
eval(p,mvar,val) == univariate(p,mvar)(val)

evalSortedVarlist(p,Lvar,Lpval):% ==
  p case R => p
  empty? Lvar or empty? Lpval => p
  mvar := Lvar.first
  mvar > p.v => evalSortedVarlist(p,Lvar.rest,Lpval.rest)
  pval := Lpval.first
  pts := map(x1+>evalSortedVarlist(x1,Lvar,Lpval),p.ts)
  mvar=p.v =>
    pval case R => pts (pval::R)
    pts pval
  multivariate(pts,p.v)

```

```

eval(p,Lvar,Lpval) ==
  empty? rest Lvar => evalSortedVarlist(p,Lvar,Lpval)
  sorted?((x1,x2) +-> x1 > x2, Lvar) => evalSortedVarlist(p,Lvar,Lpval)
  nlvar := sort((x1,x2) +-> x1 > x2,Lvar)
  nlpval :=
    Lvar = nlvar => Lpval
    nlpval := [Lpval.position(mvar,Lvar) for mvar in nlvar]
  evalSortedVarlist(p,nlvar,nlpval)

eval(p,Lvar,Lval) ==
  eval(p,Lvar,[val::% for val in Lval]$(List %)) -- kill?

degree(p,mvar) ==
  p case R => 0
  mvar= p.v => degree p.ts
  mvar > p.v => 0 -- might as well take advantage of the order
  max(degree(leadingCoefficient p.ts,mvar),degree(red p,mvar))

degree(p,Lvar) == [degree(p,mvar) for mvar in Lvar]

degree p ==
  p case R => 0
  degree(leadingCoefficient(p.ts)) + monomial(degree(p.ts), p.v)

minimumDegree p ==
  p case R => 0
  md := minimumDegree p.ts
  minimumDegree(coefficient(p.ts,md)) + monomial(md, p.v)

minimumDegree(p,mvar) ==
  p case R => 0
  mvar = p.v => minimumDegree p.ts
  md:=minimumDegree(leadingCoefficient p.ts,mvar)
  zero? (p1:=red p) => md
  min(md,minimumDegree(p1,mvar))

minimumDegree(p,Lvar) ==
  [minimumDegree(p,mvar) for mvar in Lvar]

totalDegree(p, Lvar) ==
  ground? p => 0
  null setIntersection(Lvar, variables p) => 0
  u := univariate(p, mv := mainVariable(p)::VarSet)
  weight:NonNegativeInteger := (member?(mv,Lvar) => 1; 0)
  tdeg:NonNegativeInteger := 0
  while u ^= 0 repeat
    termdeg:NonNegativeInteger := weight*degree u +
      totalDegree(leadingCoefficient u, Lvar)
    tdeg := max(tdeg, termdeg)
    u := reductum u

```

```

tdeg

if R has CommutativeRing then
  differentiate(p,mvar) ==
    p case R => 0
    mvar=p.v =>
      up:=differentiate p.ts
      if ground? up then leadingCoefficient(up) else [mvar,up]$VPoly
    up:=map(x1 +-> differentiate(x1,mvar),p.ts)
    if ground? up then leadingCoefficient(up) else [p.v,up]$VPoly

leadingCoefficient(p) ==
  p case R => p
  leadingCoefficient(leadingCoefficient(p.ts))

-- trailingCoef(p) ==
--   p case R => p
--   coef(p.ts,0) case R => coef(p.ts,0)
--   trailingCoef(red p)
--   TrailingCoef(p) == trailingCoef(p)

leadingMonomial p ==
  p case R => p
  monomial(leadingMonomial leadingCoefficient(p.ts),
    p.v, degree(p.ts))

reductum(p) ==
  p case R => 0
  p - leadingMonomial p

-- if R is Integer then
--   pgcd := PolynomialGcdPackage(%,VarSet)
--   gcd(p1,p2) ==
--     gcd(p1,p2)$pgcd
--
-- else if R is RationalNumber then
--   gcd(p1,p2) ==
--     mrat:= MRationalFactorize(VarSet,%)
--     gcd(p1,p2)$mrat
--
-- else gcd(p1,p2) ==
--   p1 case R =>
--     p2 case R => gcd(p1,p2)$R::%
--     p1 = 0 => p2
--     gcd(p1, content(p2.ts))
--   p2 case R =>
--     p2 = 0 => p1
--     gcd(p2, content(p1.ts))
--   p1.v < p2.v => gcd(p1, content(p2.ts))

```

```
--      p1.v > p2.v => gcd(content(p1.ts), p2)
--      PSimp(p1.v, gcd(p1.ts, p2.ts))
```

— SMP.dotabb —

```
"SMP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SMP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SMP" -> "PFECAT"
```

domain SMTS SparseMultivariateTaylorSeries

— SparseMultivariateTaylorSeries.input —

```
)set break resume
)sys rm -f SparseMultivariateTaylorSeries.output
)spool SparseMultivariateTaylorSeries.output
)set message test on
)set message auto off
)clear all

--S 1 of 10
xts:=x::TaylorSeries Fraction Integer
--R
--R
--R (1) x
--R
--R                                          Type: TaylorSeries(Fraction(Integer))
--E 1

--S 2 of 10
yts:=y::TaylorSeries Fraction Integer
--R
--R
--R (2) y
--R
--R                                          Type: TaylorSeries(Fraction(Integer))
--E 2

--S 3 of 10
zts:=z::TaylorSeries Fraction Integer
--R
--R
```

```

--R (3) z
--R
--R                                         Type: TaylorSeries(Fraction(Integer))
--E 3

--S 4 of 10
t1:=sin(xts)
--R
--R
--R
--R          1 3      1 5      1 7      1 9
--R (4) x - - x + --- x - ---- x + ----- x + 0(11)
--R          6      120     5040    362880
--R
--R                                         Type: TaylorSeries(Fraction(Integer))
--E 4

--S 5 of 10
coefficient(t1,3)
--R
--R
--R
--R          1 3
--R (5) - - x
--R          6
--R
--R                                         Type: Polynomial(Fraction(Integer))
--E 5

--S 6 of 10
coefficient(t1,monomial(3,x)$IndexedExponents Symbol)
--R
--R
--R
--R          1
--R (6) - -
--R          6
--R
--R                                         Type: Fraction(Integer)
--E 6

--S 7 of 10
t2:=sin(xts + yts)
--R
--R
--R (7)
--R          1 3      1 2      1 2      1 3
--R (y + x) + (- - y - - x y - - x y - - x )
--R          6      2      2      6
--R
--R +
--R          1 5      1 4      1 2 3      1 3 2      1 4      1 5
--R (--- y + -- x y + -- x y + -- x y + -- x y + --- x )
--R          120     24      12      12      24      120
--R
--R +
--R PAREN
--R          1 7      1 6      1 2 5      1 3 4      1 4 3      1 5 2
--R - ---- y - --- x y - --- x y - --- x y - --- x y - --- x y

```

```

--R      5040      720      240      144      144      240
--R      +
--R      1  6      1  7
--R      - --- x y - ---- x
--R      720      5040
--R      +
--R      PAREN
--R      1  9      1  8      1  2 7      1  3 6      1  4 5
--R      ----- y + ----- x y + ----- x y + ----- x y + ----- x y
--R      362880      40320      10080      4320      2880
--R      +
--R      1  5 4      1  6 3      1  7 2      1  8      1  9
--R      ---- x y + ---- x y + ---- x y + ---- x y + ---- x
--R      2880      4320      10080      40320      362880
--R      +
--R      0(11)
--R
--R                                          Type: TaylorSeries(Fraction(Integer))
--E 7

```

```

--S 8 of 10
coefficient(t2,3)
--R
--R
--R      1  3      1  2      1  2      1  3
--R      (8)  - - y - - x y - - x y - - x
--R      6      2      2      6
--R
--R                                          Type: Polynomial(Fraction(Integer))
--E 8

```

```

--S 9 of 10
coefficient(t2,monomial(3,x)$IndexedExponents Symbol)
--R
--R
--R      1
--R      (9)  - -
--R      6
--R
--R                                          Type: Fraction(Integer)
--E 9

```

```

--S 10 of 10
polynomial(t2,5)
--R
--R
--R      (10)
--R      1  5      1  4      1  2      1  3      1  3      1  2      1  4      1  2
--R      --- y + -- x y + (-- x - -)y + (-- x - - x)y + (-- x - - x + 1)y
--R      120      24      12      6      12      2      24      2
--R      +
--R      1  5      1  3
--R      --- x - - x + x

```



```

--R      120      6
--R
--R                                          Type: Polynomial(Fraction(Integer))
--E 10

)spool
)lisp (bye)

```

— SparseMultivariateTaylorSeries.help —

```

=====
SparseMultivariateTaylorSeries examples
=====

```

Assume we have three variables which get expressed as sparse multivariate taylor series.

```

xts:=x::TaylorSeries Fraction Integer
yts:=y::TaylorSeries Fraction Integer
zts:=z::TaylorSeries Fraction Integer

```

These will cause traditional routines to expand in series form:

```
t1:=sin(xts)
```

$$x - \frac{1}{6}x^3 + \frac{1}{120}x^5 - \frac{1}{5040}x^7 + \frac{1}{362880}x^9 + O(11)$$

We can ask for a specific coefficient, in this case, the coefficient of the third power.

```
coefficient(t1,3)
```

$$-\frac{1}{6}x^3$$

And we can get that coefficient, expressed as a monomial.

```
coefficient(t1,monomial(3,x)$IndexedExponents Symbol)
```

$$-\frac{1}{6}$$

In a multivariate version we get a polynomial in x and y

```
t2:=sin(xts + yts)
```

$$\begin{aligned}
 & (y + x) + \left(-\frac{1}{6}y^3 - \frac{1}{2}xy^2 - \frac{1}{2}x^2y - \frac{1}{6}x^3 \right) \\
 & + \left(\frac{1}{120}y^5 + \frac{1}{24}xy^4 + \frac{1}{12}x^2y^3 + \frac{1}{12}x^3y^2 + \frac{1}{24}x^4y + \frac{1}{120}x^5 \right) \\
 & + \text{PAREN} \\
 & \quad -\frac{1}{5040}y^7 - \frac{1}{720}xy^6 - \frac{1}{240}x^2y^5 - \frac{1}{144}x^3y^4 - \frac{1}{144}x^4y^3 - \frac{1}{240}x^5y^2 \\
 & \quad -\frac{1}{720}x^6y - \frac{1}{5040}x^7 \\
 & + \text{PAREN} \\
 & \quad \frac{1}{362880}y^9 + \frac{1}{40320}xy^8 + \frac{1}{10080}x^2y^7 + \frac{1}{4320}x^3y^6 + \frac{1}{2880}x^4y^5 \\
 & \quad + \frac{1}{2880}x^5y^4 + \frac{1}{4320}x^6y^3 + \frac{1}{10080}x^7y^2 + \frac{1}{40320}x^8y + \frac{1}{362880}x^9 \\
 & + 0(11)
 \end{aligned}$$

We can ask for the third coefficient which is

```
coefficient(t2,3)
```

$$-\frac{1}{6}y^3 - \frac{1}{2}xy^2 - \frac{1}{2}x^2y - \frac{1}{6}x^3$$

And we can ask for the third coefficient of that coefficient in x

```
coefficient(t2,monomial(3,x)$IndexedExponents Symbol)
```

$$-\frac{1}{6}$$

And we can convert that result to a polynomial

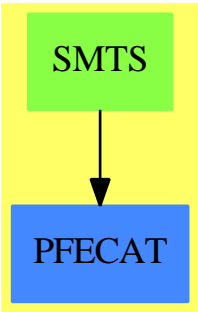
polynomial(t2,5)

$$\begin{aligned} & \frac{1}{120} y^5 + \frac{1}{24} x y^4 + \left(-\frac{1}{12} x^2 - \frac{1}{6} \right) y^3 + \left(-\frac{1}{12} x^3 - \frac{1}{2} x \right) y^2 + \left(-\frac{1}{24} x^4 - \frac{1}{2} x^2 + 1 \right) y \\ & + \frac{1}{120} x^5 - \frac{1}{6} x^3 + x \end{aligned}$$

See Also:
o)show SparseMultivariateTaylorSeries
o)display op coefficient

—————→

SparseMultivariateTaylorSeries (SMTS)



See

⇒ “TaylorSeries” (TS) [21](#) on page [2670](#)

Exports:

0	1	acos	acosh	acot
acoth	acsc	acsch	asec	asech
asin	asinh	associates?	atan	atanh
characteristic	charthRoot	coefficient	coerce	complete
cos	cosh	cot	coth	csc
csch	csubst	D	degree	differentiate
eval	exp	exquo	extend	fintegrate
hash	integrate	latex	leadingCoefficient	leadingMonomial
log	map	monomial	monomial?	nthRoot
one?	order	pi	pole?	polynomial
recip	reductum	sample	sec	sech
sin	sinh	sqrt	subtractIfCan	tan
tanh	unit?	unitCanonical	unitNormal	variables
zero?	?*?	?**?	?+?	?-?
-?	?=?	?^?	?~=?	

— domain SMTS SparseMultivariateTaylorSeries —

```
)abbrev domain SMTS SparseMultivariateTaylorSeries
++ Authors: William Burge, Stephen Watt, Clifton Williamson
++ Date Created: 15 August 1988
++ Date Last Updated: 18 May 1991
++ Description:
++ This domain provides multivariate Taylor series with variables
++ from an arbitrary ordered set. A Taylor series is represented
++ by a stream of polynomials from the polynomial domain SMP.
++ The nth element of the stream is a form of degree n. SMTS is an
++ internal domain.
```

```
SparseMultivariateTaylorSeries(Coef,Var,SMP):_
Exports == Implementation where
  Coef : Ring
  Var : OrderedSet
  SMP : PolynomialCategory(Coef,IndexedExponents Var,Var)
  I ==> Integer
  L ==> List
  NNI ==> NonNegativeInteger
  OUT ==> OutputForm
  PS ==> InnerTaylorSeries SMP
  RN ==> Fraction Integer
  ST ==> Stream
  StS ==> Stream SMP
  STT ==> StreamTaylorSeriesOperations SMP
  STF ==> StreamTranscendentalFunctions SMP
  ST2 ==> StreamFunctions2
  ST3 ==> StreamFunctions3

Exports ==> MultivariateTaylorSeriesCategory(Coef,Var) with
```

```

coefficient: (% , NNI) -> SMP
  ++ \spad{coefficient(s, n)} gives the terms of total degree n.
  ++
  ++X xts:=x::TaylorSeries Fraction Integer
  ++X t1:=sin(xts)
  ++X coefficient(t1,3)

coerce: Var -> %
  ++ \spad{coerce(var)} converts a variable to a Taylor series
coerce: SMP -> %
  ++ \spad{coerce(poly)} regroups the terms by total degree and forms
  ++ a series.
"*": (SMP, %) -> %
  ++ \spad{smp*ts} multiplies a TaylorSeries by a monomial SMP.
csubst: (L Var, L StS) -> (SMP -> StS)
  ++ \spad{csubst(a,b)} is for internal use only

if Coef has Algebra Fraction Integer then
  integrate: (% , Var, Coef) -> %
    ++ \spad{integrate(s,v,c)} is the integral of s with respect
    ++ to v and having c as the constant of integration.
  fintegrate: (() -> % , Var, Coef) -> %
    ++ \spad{fintegrate(f,v,c)} is the integral of \spad{f()} with respect
    ++ to v and having c as the constant of integration.
    ++ The evaluation of \spad{f()} is delayed.

```

Implementation ==> PS add

```

Rep := StS -- Below we use the fact that Rep of PS is Stream SMP.
extend(x,n) == extend(x,n + 1)$Rep
complete x == complete(x)$Rep
stream(x:%):Rep == x @ Rep

evalstream:(% , L Var, L SMP) -> StS
evalstream(s:% , lv:(L Var), lsmp:(L SMP)):(ST SMP) ==
  scan(0, _+$SMP,
    map((z1:SMP):SMP+-->eval(z1,lv,lsmp), s pretend StS))$ST2(SMP, SMP)

addvariable:(Var, InnerTaylorSeries Coef) -> %
addvariable(v,s) ==
  ints := integers(0)$STT pretend ST NNI
  map((n1:NNI, c2:Coef):SMP+-->monomial(c2 :: SMP, v, n1)$SMP,
    ints, s pretend ST Coef)$ST3(NNI, Coef, SMP)

-- We can extract a polynomial giving the terms of given total degree
coefficient(s,n) == elt(s,n + 1)$Rep -- 1-based indexing for streams

-- Here we have to take into account that we reduce the degree of each
-- term of the stream by a constant
coefficient(s:% , lv:List Var, ln:List NNI):% ==

```

```

    map ((z1:SMP):SMP +-> coefficient(z1,lv,ln),rest(s,reduce(_+,ln)))

-- the coefficient of a particular monomial:
coefficient(s:%,m:IndexedExponents Var):Coef ==
  n:=leadingCoefficient(mon:=m)
  while not zero?(mon:=reductum mon) repeat
    n:=n+leadingCoefficient mon
  coefficient(coefficient(s,n),m)

--% creation of series

coerce(r:Coef) == monom(r::SMP,0)$STT
smp:SMP * p:% == ((smp) * (p @ Rep))$STT @ %
r:Coef * p:% == ((r::SMP) * (p @ Rep))$STT @ %
p:% * r:Coef == ((r::SMP) * (p @ Rep))$STT @ %
mts(p:SMP):% ==
  (uv := mainVariable p) case "failed" => monom(p,0)$STT
  v := uv :: Var
  s : % := 0
  up := univariate(p,v)
  while not zero? up repeat
    s := s + monomial(1,v,degree up) * mts(leadingCoefficient up)
    up := reductum up
  s

coerce(p:SMP) == mts p
coerce(v:Var) == v :: SMP :: %

monomial(r:%,v:Var,n:NNI) ==
  r * monom(monomial(1,v,n)$SMP,n)$STT

--% evaluation

substvar: (SMP,L Var,L %) -> %
substvar(p,vl,q) ==
  null vl => monom(p,0)$STT
  (uv := mainVariable p) case "failed" => monom(p,0)$STT
  v := uv :: Var
  v = first vl =>
    s : % := 0
    up := univariate(p,v)
    while not zero? up repeat
      c := leadingCoefficient up
      s := s + first q ** degree up * substvar(c,rest vl,rest q)
      up := reductum up
    s
  substvar(p,rest vl,rest q)

sortmfirst:(SMP,L Var,L %) -> %
sortmfirst(p,vl,q) ==

```

```

nlv : L Var := sort((v1:Var,v2:Var):Boolean +-> v1 > v2,v1)
nq : L % := [q position$(L Var) (i,v1) for i in nlv]
substvar(p,nlv,nq)

csubst(v1,q) == (p1:SMP):StS+>sortmfirst(p1,v1,q pretend L(%)) pretend StS

restCheck(s:StS):StS ==
  -- checks that stream is null or first element is 0
  -- returns empty() or rest of stream
  empty? s => s
  not zero? first s =>
    error "eval: constant coefficient should be 0"
  rst s

eval(s:%,v:L Var,q:L %) ==
  #v ^= #q =>
    error "eval: number of variables should equal number of values"
  nq : L StS := [restCheck(i pretend StS) for i in q]
  addiag(map(csubst(v,nq),s pretend StS)$ST2(SMP,StS))$STT @ %

substmts(v:Var,p:SMP,q:%):% ==
  up := univariate(p,v)
  ss : % := 0
  while not zero? up repeat
    d:=degree up
    c:SMP:=leadingCoefficient up
    ss := ss + c* q ** d
    up := reductum up
  ss

subststream(v:Var,p:SMP,q:StS):StS==
  substmts(v,p,q @ %) pretend StS

comp1:(Var,StS,StS) -> StS
comp1(v,r,t)==
  addiag(map((p1:SMP):StS +-> subststream(v,p1,t),r)$ST2(SMP,StS))$STT

comp(v:Var,s:StS,t:StS):StS == delay
  empty? s => s
  f := first s; r : StS := rst s;
  empty? r => s
  empty? t => concat(f,comp1(v,r,empty()$StS))
  not zero? first t =>
    error "eval: constant coefficient should be zero"
  concat(f,comp1(v,r,rst t))

eval(s:%,v:Var,t:%) == comp(v,s pretend StS,t pretend StS)

--% differentiation and integration

```

```

differentiate(s:%,v:Var):% ==
  empty? s => 0
  map((z1:SMP):SMP +-> differentiate(z1,v),rst s)

if Coef has Algebra Fraction Integer then

(x:%) ** (r:RN) == powern(r,stream x)$STT
(r:RN) * (x:%) ==
  map((z1:SMP):SMP +-> r*z1,stream x)$ST2(SMP,SMP) @ %
(x:%) * (r:RN) ==
  map((z1:SMP):SMP +-> z1*r,stream x)$ST2(SMP,SMP) @ %

exp x == exp(stream x)$STF
log x == log(stream x)$STF

sin x == sin(stream x)$STF
cos x == cos(stream x)$STF
tan x == tan(stream x)$STF
cot x == cot(stream x)$STF
sec x == sec(stream x)$STF
csc x == csc(stream x)$STF

asin x == asin(stream x)$STF
acos x == acos(stream x)$STF
atan x == atan(stream x)$STF
acot x == acot(stream x)$STF
asec x == asec(stream x)$STF
acsc x == acsc(stream x)$STF

sinh x == sinh(stream x)$STF
cosh x == cosh(stream x)$STF
tanh x == tanh(stream x)$STF
coth x == coth(stream x)$STF
sech x == sech(stream x)$STF
csch x == csch(stream x)$STF

asinh x == asinh(stream x)$STF
acosh x == acosh(stream x)$STF
atanh x == atanh(stream x)$STF
acoth x == acoth(stream x)$STF
asech x == asech(stream x)$STF
acsch x == acsch(stream x)$STF

intsmp(v:Var,p: SMP): SMP ==
  up := univariate(p,v)
  ss : SMP := 0
  while not zero? up repeat
    d := degree up
    c := leadingCoefficient up
    ss := ss + inv((d+1) :: RN) * monomial(c,v,d+1)$SMP

```



```

      up := reductum up
    ss

    fintegrate(f,v,r) ==
      concat(r::SMP,delay map((z1:SMP):SMP +-> intsmp(v,z1),f() pretend StS))
    integrate(s,v,r) ==
      concat(r::SMP,map((z1:SMP):SMP +-> intsmp(v,z1),s pretend StS))

-- If there is more than one term of the same order, group them.
tout(p:SMP):OUT ==
  pe := p :: OUT
  monomial? p => pe
  paren pe

showAll?: () -> Boolean
-- check a global Lisp variable
showAll?() == true

coerce(s:%):OUT ==
  uu := s pretend Stream(SMP)
  empty? uu => (0$SMP) :: OUT
  n : NNI; count : NNI := _$streamCount$Lisp
  l : List OUT := empty()
  for n in 0..count while not empty? uu repeat
    if frst(uu) ^= 0 then l := concat(tout frst uu,l)
    uu := rst uu
  if showAll?() then
    for n in n.. while explicitEntries? uu and _
      not eq?(uu,rst uu) repeat
        if frst(uu) ^= 0 then l := concat(tout frst uu,l)
        uu := rst uu
  l :=
    explicitlyEmpty? uu => l
    eq?(uu,rst uu) and frst uu = 0 => l
    concat(prefix("0" :: OUT,[n :: OUT]),l)
  empty? l => (0$SMP) :: OUT
  reduce("+",reverse! l)
if Coef has Field then
  SF2==> StreamFunctions2
  p:% / r:Coef ==
    (map((z1:SMP):SMP +-> z1/$SMP r,stream p)$SF2(SMP,SMP)) @ %

```

— SMTS.dotabb —

"SMTS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SMTS"]

"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]

```
=====
SparseTable examples
=====
```

```
t.3 := "Number three"
      "Number three"
                                     Type: String
```

```
t.4 := "Number four"
      "Number four"
```

Type: String

These values can be retrieved as usual, but if a look up fails the default entry will be returned.

```
t.3
      "Number three"
```

Type: String

```
t.2
      "Try again!"
```

Type: String

To see which values are explicitly stored, the keys and entries functions can be used.

```
keys t
      [4,3]
```

Type: List Integer

```
entries t
      ["Number four","Number three"]
```

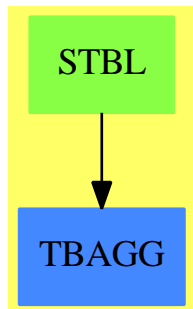
Type: List String

If a specific table representation is required, the GeneralSparseTable constructor should be used. The domain SparseTable(K, E, dflt)} is equivalent to GeneralSparseTable(K,E,Table(K,E), dflt).

See Also:

- o)help Table
- o)help GeneralSparseTable
- o)show SparseTable

SparseTable (STBL)



See

- ⇒ “HashTable” (HASHTBL) [9](#) on page [1120](#)
- ⇒ “InnerTable” (INTABL) [10](#) on page [1335](#)
- ⇒ “Table” (TABLE) [21](#) on page [2663](#)
- ⇒ “EqTable” (EQTBL) [6](#) on page [698](#)
- ⇒ “StringTable” (STRTBL) [20](#) on page [2610](#)
- ⇒ “GeneralSparseTable” (GSTBL) [8](#) on page [1078](#)

Exports:

any?	bag	coerce	construct	convert
copy	count	dictionary	elt	empty
empty?	entries	entry?	eq?	eval
every?	extract!	fill!	find	first
hash	index?	indices	insert!	inspect
key?	keys	latex	less?	map
map!	maxIndex	member?	members	minIndex
more?	parts	qelt	qsetelt!	reduce
remove	remove!	removeDuplicates	sample	search
setelt	select	select!	size?	swap!
table	#?	?=?	?~=?	?.?

— domain STBL SparseTable —

```

)abbrev domain STBL SparseTable
++ Author: Stephen M. Watt
++ Date Created: 1986
++ Date Last Updated: June 21, 1991
++ Description:
++ A sparse table has a default entry, which is returned if no other
++ value has been explicitly stored for a key.

```

```

SparseTable(Key:SetCategory, Ent:SetCategory, dent:Ent) ==
  GeneralSparseTable(Key, Ent, Table(Key, Ent), dent)

```

— STBL.dotabb —

```
"STBL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STBL"]
"TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]
"STBL" -> "TBAGG"
```

domain SULS SparseUnivariateLaurentSeries

— SparseUnivariateLaurentSeries.input —

```
)set break resume
)sys rm -f SparseUnivariateLaurentSeries.output
)spool SparseUnivariateLaurentSeries.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SparseUnivariateLaurentSeries
--R
--R SparseUnivariateLaurentSeries(Coef: Ring,var: Symbol,cen: Coef) is a domain constructor
--R Abbreviation for SparseUnivariateLaurentSeries is SULS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SULS
--R
--R----- Operations -----
--R ?? : (Coef,%) -> %           ?? : (%,Coef) -> %
--R ?? : (%,%) -> %             ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R +? : (%,%) -> %             ?-? : (%,%) -> %
--R -? : % -> %                 ?? : (%,%) -> Boolean
--R 1 : () -> %                 0 : () -> %
--R ^? : (%,PositiveInteger) -> %   center : % -> Coef
--R coefficient : (%,Integer) -> Coef   coerce : Variable(var) -> %
--R coerce : Integer -> %               coerce : % -> OutputForm
--R complete : % -> %                 degree : % -> Integer
--R ?.? : (%,Integer) -> Coef         extend : (%,Integer) -> %
--R hash : % -> SingleInteger         latex : % -> String
--R leadingCoefficient : % -> Coef     leadingMonomial : % -> %
--R map : ((Coef -> Coef),%) -> %     monomial : (Coef,Integer) -> %
--R monomial? : % -> Boolean           one? : % -> Boolean
```

```

--R order : (%,Integer) -> Integer          order : % -> Integer
--R pole? : % -> Boolean                    recip : % -> Union(%, "failed")
--R reductum : % -> %                       removeZeroes : (Integer,%) -> %
--R removeZeroes : % -> %                   sample : () -> %
--R truncate : (%,Integer) -> %             variable : % -> Symbol
--R zero? : % -> Boolean                     ?~=? : (%,%) -> Boolean
--R ?? : (% , Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (Fraction(Integer),%) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (SparseUnivariateTaylorSeries(Coef,var,cen),%) -> % if Coef has FIELD
--R ?? : (% , SparseUnivariateTaylorSeries(Coef,var,cen)) -> % if Coef has FIELD
--R ?? : (NonNegativeInteger,%) -> %
--R *** : (% , Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R *** : (% ,%) -> % if Coef has ALGEBRA(FRAC(INT))
--R *** : (% ,Integer) -> % if Coef has FIELD
--R *** : (% ,NonNegativeInteger) -> %
--R ?/? : (SparseUnivariateTaylorSeries(Coef,var,cen),SparseUnivariateTaylorSeries(Coef,var,cen))
--R ?/? : (% ,%) -> % if Coef has FIELD
--R ?/? : (% ,Coef) -> % if Coef has FIELD
--R ?<? : (% ,%) -> Boolean if SparseUnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD
--R ?<=? : (% ,%) -> Boolean if SparseUnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD
--R ?>? : (% ,%) -> Boolean if SparseUnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD
--R ?>=? : (% ,%) -> Boolean if SparseUnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD
--R D : (% ,Symbol) -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has PDRING(SYMBOL) and Coef has FIELD
--R D : (% ,List(Symbol)) -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has PDRING(SYMBOL) and Coef has FIELD
--R D : (% ,Symbol,NonNegativeInteger) -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has PDRING(SYMBOL) and Coef has FIELD
--R D : (% ,List(Symbol),List(NonNegativeInteger)) -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has PDRING(SYMBOL) and Coef has FIELD
--R D : % -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has DIFRING and Coef has FIELD or Coef has ALGEBRA(FRAC(INT))
--R D : (% ,NonNegativeInteger) -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has DIFRING and Coef has FIELD
--R D : (% , (SparseUnivariateTaylorSeries(Coef,var,cen) -> SparseUnivariateTaylorSeries(Coef,var,cen)))
--R D : (% , (SparseUnivariateTaylorSeries(Coef,var,cen) -> SparseUnivariateTaylorSeries(Coef,var,cen)))
--R ?? : (% ,Integer) -> % if Coef has FIELD
--R ?? : (% ,NonNegativeInteger) -> %
--R abs : % -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD
--R acos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acoth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R approximate : (% ,Integer) -> Coef if Coef has **: (Coef,Integer) -> Coef and Coef has coercion
--R asec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R associates? : (% ,%) -> Boolean if SparseUnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD
--R atan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R atanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R ceiling : % -> SparseUnivariateTaylorSeries(Coef,var,cen) if SparseUnivariateTaylorSeries(Coef,var,cen) has FIELD
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if $ has CHARNZ and SparseUnivariateTaylorSeries(Coef,var,cen) has FIELD

```

```

--R coerce : Fraction(Integer) -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has RETRACT(INT) and Coef
--R coerce : % -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD or SparseU
--R coerce : Symbol -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has RETRACT(SYMBOL) and Coef has FIE
--R coerce : SparseUnivariateTaylorSeries(Coef,var,cen) -> %
--R coerce : Coef -> % if Coef has COMRING
--R conditionP : Matrix(%) -> Union(Vector(%),"failed") if $ has CHARNZ and SparseUnivariateTaylorSeries(Co
--R convert : % -> Pattern(Integer) if SparseUnivariateTaylorSeries(Coef,var,cen) has KONVERT(PATTERN(INT))
--R convert : % -> Pattern(Float) if SparseUnivariateTaylorSeries(Coef,var,cen) has KONVERT(PATTERN(FLOAT))
--R convert : % -> DoubleFloat if SparseUnivariateTaylorSeries(Coef,var,cen) has REAL and Coef has FIELD
--R convert : % -> Float if SparseUnivariateTaylorSeries(Coef,var,cen) has REAL and Coef has FIELD
--R convert : % -> InputForm if SparseUnivariateTaylorSeries(Coef,var,cen) has KONVERT(INFORM) and Coef has
--R cos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R coth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R denom : % -> SparseUnivariateTaylorSeries(Coef,var,cen) if Coef has FIELD
--R denominator : % -> % if Coef has FIELD
--R differentiate : (%,Symbol) -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has PDRING(SYMBOL) and Co
--R differentiate : (%,List(Symbol)) -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has PDRING(SYMBOL) :
--R differentiate : (%,Symbol,NonNegativeInteger) -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has PD
--R differentiate : (%,List(Symbol),List(NonNegativeInteger)) -> % if SparseUnivariateTaylorSeries(Coef,var
--R differentiate : % -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has DIFRING and Coef has FIELD or
--R differentiate : (%,NonNegativeInteger) -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has DIFRING an
--R differentiate : (%,Variable(var)) -> %
--R differentiate : (%,(SparseUnivariateTaylorSeries(Coef,var,cen) -> SparseUnivariateTaylorSeries(Coef,var
--R differentiate : (%,(SparseUnivariateTaylorSeries(Coef,var,cen) -> SparseUnivariateTaylorSeries(Coef,var
--R divide : (%,%) -> Record(quotient: %,remainder: %) if Coef has FIELD
--R ?? : (%,SparseUnivariateTaylorSeries(Coef,var,cen)) -> % if SparseUnivariateTaylorSeries(Coef,var,cen)
--R ?? : (%,%) -> % if Integer has SGROUP
--R euclideanSize : % -> NonNegativeInteger if Coef has FIELD
--R eval : (%,List(SparseUnivariateTaylorSeries(Coef,var,cen)),List(SparseUnivariateTaylorSeries(Coef,var,c
--R eval : (%,SparseUnivariateTaylorSeries(Coef,var,cen),SparseUnivariateTaylorSeries(Coef,var,cen)) -> % i
--R eval : (%,Equation(SparseUnivariateTaylorSeries(Coef,var,cen))) -> % if SparseUnivariateTaylorSeries(Co
--R eval : (%,List(Equation(SparseUnivariateTaylorSeries(Coef,var,cen)))) -> % if SparseUnivariateTaylorSer
--R eval : (%,List(Symbol),List(SparseUnivariateTaylorSeries(Coef,var,cen))) -> % if SparseUnivariateTaylor
--R eval : (%,Symbol,SparseUnivariateTaylorSeries(Coef,var,cen)) -> % if SparseUnivariateTaylorSeries(Coef,
--R eval : (%,Coef) -> Stream(Coef) if Coef has **: (Coef,Integer) -> Coef
--R exp : % -> % if Coef has ALGEBRA(FRAC(INT))
--R expressIdealMember : (List(%),%) -> Union(List(%),"failed") if Coef has FIELD
--R exquo : (%,%) -> Union(%,"failed") if SparseUnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef ha
--R extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %) if Coef has FIELD
--R extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %),"failed") if Coef has FIELD
--R factor : % -> Factored(%) if Coef has FIELD
--R factorPolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if SparseUn
--R factorSquareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) i
--R floor : % -> SparseUnivariateTaylorSeries(Coef,var,cen) if SparseUnivariateTaylorSeries(Coef,var,cen) ha
--R fractionPart : % -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has EUCDOM and Coef has FIELD
--R gcd : (%,%) -> % if Coef has FIELD

```



```

--R gcd : List(%) -> % if Coef has FIELD
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUniva
--R init : () -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has STEP and Coef has FIELD
--R integrate : (%,Variable(var)) -> % if Coef has ALGEBRA(FRAC(INT))
--R integrate : (%,Symbol) -> % if Coef has integrate: (Coef,Symbol) -> Coef and Coef has variab
--R integrate : % -> % if Coef has ALGEBRA(FRAC(INT))
--R inv : % -> % if Coef has FIELD
--R laurent : (Integer,SparseUnivariateTaylorSeries(Coef,var,cen)) -> %
--R lcm : (%,%) -> % if Coef has FIELD
--R lcm : List(%) -> % if Coef has FIELD
--R log : % -> % if Coef has ALGEBRA(FRAC(INT))
--R map : ((SparseUnivariateTaylorSeries(Coef,var,cen) -> SparseUnivariateTaylorSeries(Coef,var,
--R max : (%,%) -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIEL
--R min : (%,%) -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIEL
--R monomial : (%,List(SingletonAsOrderedSet),List(Integer)) -> %
--R monomial : (%,SingletonAsOrderedSet,Integer) -> %
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed") if Coef has FIELD
--R multiplyCoefficients : ((Integer -> Coef),%) -> %
--R multiplyExponents : (%,PositiveInteger) -> %
--R negative? : % -> Boolean if SparseUnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef
--R nextItem : % -> Union(%,"failed") if SparseUnivariateTaylorSeries(Coef,var,cen) has STEP and
--R nthRoot : (%,Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R numer : % -> SparseUnivariateTaylorSeries(Coef,var,cen) if Coef has FIELD
--R numerator : % -> % if Coef has FIELD
--R patternMatch : (%,Pattern(Float),PatternMatchResult(Float,%)) -> PatternMatchResult(Float,%)
--R patternMatch : (%,Pattern(Integer),PatternMatchResult(Integer,%)) -> PatternMatchResult(Inte
--R pi : () -> % if Coef has ALGEBRA(FRAC(INT))
--R positive? : % -> Boolean if SparseUnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef
--R prime? : % -> Boolean if Coef has FIELD
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %) if Coef has FIELD
--R ?quo? : (%,%) -> % if Coef has FIELD
--R random : () -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has INS and Coef has FIELD
--R rationalFunction : (%,Integer,Integer) -> Fraction(Polynomial(Coef)) if Coef has INTDOM
--R rationalFunction : (%,Integer) -> Fraction(Polynomial(Coef)) if Coef has INTDOM
--R reducedSystem : Matrix(%) -> Matrix(Integer) if SparseUnivariateTaylorSeries(Coef,var,cen) h
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) i
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(SparseUnivariateTaylorSeries(Coe
--R reducedSystem : Matrix(%) -> Matrix(SparseUnivariateTaylorSeries(Coef,var,cen)) if Coef has
--R ?rem? : (%,%) -> % if Coef has FIELD
--R retract : % -> Integer if SparseUnivariateTaylorSeries(Coef,var,cen) has RETRACT(INT) and Co
--R retract : % -> Fraction(Integer) if SparseUnivariateTaylorSeries(Coef,var,cen) has RETRACT(I
--R retract : % -> Symbol if SparseUnivariateTaylorSeries(Coef,var,cen) has RETRACT(SYMBOL) and
--R retract : % -> SparseUnivariateTaylorSeries(Coef,var,cen)
--R retractIfCan : % -> Union(Integer,"failed") if SparseUnivariateTaylorSeries(Coef,var,cen) ha
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if SparseUnivariateTaylorSeries(Coef,v
--R retractIfCan : % -> Union(Symbol,"failed") if SparseUnivariateTaylorSeries(Coef,var,cen) has
--R retractIfCan : % -> Union(SparseUnivariateTaylorSeries(Coef,var,cen),"failed")
--R sec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R series : Stream(Record(k: Integer,c: Coef)) -> %

```

```

--R sign : % -> Integer if SparseUnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD
--R sin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sizeLess? : (%,% ) -> Boolean if Coef has FIELD
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(%)),SparseUnivariatePolynomial(%)) -> U
--R sqrt : % -> % if Coef has ALGEBRA(FRAC(INT))
--R squareFree : % -> Factored(%) if Coef has FIELD
--R squareFreePart : % -> % if Coef has FIELD
--R squareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if Spar
--R subtractIfCan : (%,% ) -> Union(%, "failed")
--R tan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R tanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R taylor : % -> SparseUnivariateTaylorSeries(Coef,var,cen)
--R taylorIfCan : % -> Union(SparseUnivariateTaylorSeries(Coef,var,cen), "failed")
--R taylorRep : % -> SparseUnivariateTaylorSeries(Coef,var,cen)
--R terms : % -> Stream(Record(k: Integer,c: Coef))
--R truncate : (%,Integer,Integer) -> %
--R unit? : % -> Boolean if SparseUnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD or Sp
--R unitCanonical : % -> % if SparseUnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD or S
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %) if SparseUnivariateTaylorSeries(Coef,var,cen)
--R variables : % -> List(SingletonAsOrderedSet)
--R wholePart : % -> SparseUnivariateTaylorSeries(Coef,var,cen) if SparseUnivariateTaylorSeries(Coef,var,cen)
--R
--E 1

)spool
)lisp (bye)

```

— SparseUnivariateLaurentSeries.help —

=====

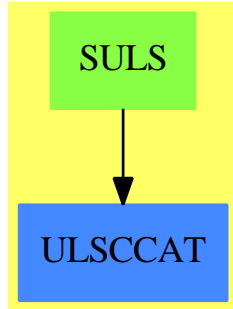
SparseUnivariateLaurentSeries examples

=====

SparseUnivariateLaurentSeries is a domain representing Laurent series in one variable with coefficients in an arbitrary ring. The parameters of the type specify the coefficient ring, the power series variable, and the center of the power series expansion. For example, SparseUnivariateLaurentSeries(Integer,x,3) represents Laurent series in (x - 3) with integer coefficients.

See Also:

o)show SparseUnivariateLaurentSeries

SparseUnivariateLaurentSeries (SULS)

Exports:

0	1	abs
acos	acosh	acot
acoth	acsc	acsch
approximate	asec	asech
asin	asinh	associates?
atan	atanh	ceiling
characteristic	charthRoot	center
coefficient	coerce	complete
conditionP	convert	cos
cosh	cot	coth
csc	csch	D
degree	denom	denominator
differentiate	divide	euclideanSize
eval	exp	expressIdealMember
exquo	extend	extendedEuclidean
factor	factorPolynomial	factorSquareFreePolynomial
floor	fractionPart	gcd
gcdPolynomial	hash	init
integrate	inv	latex
laurent	lcm	leadingCoefficient
leadingMonomial	log	map
max	min	monomial
monomial?	multiEuclidean	multiplyCoefficients
multiplyExponents	negative?	nextItem
nthRoot	numer	numerator
one?	order	patternMatch
pi	pole?	positive?
prime?	principalIdeal	random
rationalFunction	recip	reducedSystem
reductum	removeZeroes	retract
retractIfCan	sample	sec
sech	series	sign
sin	sinh	sizeLess?
solveLinearPolynomialEquation	sqrt	squareFree
squareFreePart	squareFreePolynomial	subtractIfCan
tan	tanh	taylor
taylorIfCan	taylorRep	terms
truncate	unit?	unitCanonical
unitNormal	variable	variables
wholePart	zero?	?*?
?**?	?+?	?-?
-?	?=?	?^?
?..?	?~=?	?/?
?<?	?<=?	?>?
?>=?	?^?	?quo?
?rem?		

— domain SULL SparseUnivariateLaurentSeries —

```

)abbrev domain SULL SparseUnivariateLaurentSeries
++ Author: Clifton J. Williamson
++ Date Created: 11 November 1994
++ Date Last Updated: 10 March 1995
++ Description:
++ Sparse Laurent series in one variable
++ \spadtype{SparseUnivariateLaurentSeries} is a domain representing Laurent
++ series in one variable with coefficients in an arbitrary ring. The
++ parameters of the type specify the coefficient ring, the power series
++ variable, and the center of the power series expansion. For example,
++ \spad{SparseUnivariateLaurentSeries(Integer,x,3)} represents Laurent
++ series in \spad{(x - 3)} with integer coefficients.

SparseUnivariateLaurentSeries(Coef,var,cen): Exports == Implementation where
  Coef : Ring
  var  : Symbol
  cen  : Coef
  I    ==> Integer
  NNI  ==> NonNegativeInteger
  OUT  ==> OutputForm
  P    ==> Polynomial Coef
  RF   ==> Fraction Polynomial Coef
  RN   ==> Fraction Integer
  S    ==> String
  SUTS ==> SparseUnivariateTaylorSeries(Coef,var,cen)
  EFULS ==> ElementaryFunctionsUnivariateLaurentSeries(Coef,SUTS,%)

Exports ==> UnivariateLaurentSeriesConstructorCategory(Coef,SUTS) with
  coerce: Variable(var) -> %
    ++ \spad{coerce(var)} converts the series variable \spad{var} into a
    ++ Laurent series.
  differentiate: (%,Variable(var)) -> %
    ++ \spad{differentiate(f(x),x)} returns the derivative of
    ++ \spad{f(x)} with respect to \spad{x}.
  if Coef has Algebra Fraction Integer then
    integrate: (%,Variable(var)) -> %
      ++ \spad{integrate(f(x))} returns an anti-derivative of the power
      ++ series \spad{f(x)} with constant coefficient 0.
      ++ We may integrate a series when we can divide coefficients
      ++ by integers.

Implementation ==> InnerSparseUnivariatePowerSeries(Coef) add

Rep := InnerSparseUnivariatePowerSeries(Coef)

variable x == var
center x == cen

```

```

coerce(v: Variable(var)) ==
  zero? cen => monomial(1,1)
  monomial(1,1) + monomial(cen,0)

pole? x == negative? order(x,0)

--% operations with Taylor series

coerce(uts:SUTS) == uts pretend %

taylorIfCan uls ==
  pole? uls => "failed"
  uls pretend SUTS

taylor uls ==
  (uts := taylorIfCan uls) case "failed" =>
    error "taylor: Laurent series has a pole"
  uts :: SUTS

retractIfCan(x:%):Union(SUTS,"failed") == taylorIfCan x

laurent(n,uts) == monomial(1,n) * (uts :: %)

removeZeroes uls == uls
removeZeroes(n,uls) == uls

taylorRep uls == taylor(monomial(1,-order(uls,0)) * uls)
degree uls == order(uls,0)

numer uls == taylorRep uls
denom uls == monomial(1,(-order(uls,0)) :: NNI)$SUTS

(uts:SUTS) * (uls:%) == (uts :: %) * uls
(uls:%) * (uts:SUTS) == uls * (uts :: %)

if Coef has Field then
  (uts1:SUTS) / (uts2:SUTS) == (uts1 :: %) / (uts2 :: %)

recip(uls) == iExquo(1,uls,false)

if Coef has IntegralDomain then
  uls1 exquo uls2 == iExquo(uls1,uls2,false)

if Coef has Field then
  uls1:% / uls2:% ==
    (q := uls1 exquo uls2) case "failed" =>
      error "quotient cannot be computed"
    q :: %

```

```

differentiate(uls:%,v:Variable(var)) == differentiate uls

elt(uls1:%,uls2:%) ==
  order(uls2,1) < 1 =>
    error "elt: second argument must have positive order"
  negative?(ord := order(uls1,0)) =>
    (recipr := recip uls2) case "failed" =>
      error "elt: second argument not invertible"
    uls3 := uls1 * monomial(1,-ord)
    iCompose(uls3,uls2) * (recipr :: %) ** ((-ord) :: NNI)
  iCompose(uls1,uls2)

if Coef has IntegralDomain then
  rationalFunction(uls,n) ==
    zero?(e := order(uls,0)) =>
      negative? n => 0
      polynomial(taylor uls,n :: NNI) :: RF
    negative?(m := n - e) => 0
    poly := polynomial(taylor(monomial(1,-e) * uls),m :: NNI) :: RF
    v := variable(uls) :: RF; c := center(uls) :: P :: RF
    poly / (v - c) ** ((-e) :: NNI)

  rationalFunction(uls,n1,n2) == rationalFunction(truncate(uls,n1,n2),n2)

if Coef has Algebra Fraction Integer then

  integrate uls ==
    zero? coefficient(uls,-1) =>
      error "integrate: series has term of order -1"
    integrate(uls)$Rep

  integrate(uls:%,v:Variable(var)) == integrate uls

  (uls1:%) ** (uls2:%) == exp(log(uls1) * uls2)

  exp uls == exp(uls)$EFULS
  log uls == log(uls)$EFULS
  sin uls == sin(uls)$EFULS
  cos uls == cos(uls)$EFULS
  tan uls == tan(uls)$EFULS
  cot uls == cot(uls)$EFULS
  sec uls == sec(uls)$EFULS
  csc uls == csc(uls)$EFULS
  asin uls == asin(uls)$EFULS
  acos uls == acos(uls)$EFULS
  atan uls == atan(uls)$EFULS
  acot uls == acot(uls)$EFULS
  asec uls == asec(uls)$EFULS
  acsc uls == acsc(uls)$EFULS
  sinh uls == sinh(uls)$EFULS

```

```

cosh uls == cosh(uls)$EFULS
tanh uls == tanh(uls)$EFULS
coth uls == coth(uls)$EFULS
sech uls == sech(uls)$EFULS
csch uls == csch(uls)$EFULS
asinh uls == asinh(uls)$EFULS
acosh uls == acosh(uls)$EFULS
atanh uls == atanh(uls)$EFULS
acoth uls == acoth(uls)$EFULS
asech uls == asech(uls)$EFULS
acsch uls == acsch(uls)$EFULS

if Coef has CommutativeRing then

  (uls:%) ** (r:RN) == cRationalPower(uls,r)

else

  (uls:%) ** (r:RN) ==
    negative?(ord0 := order(uls,0)) =>
      order := ord0 :: I
      (n := order exquo denom(r)) case "failed" =>
        error "**: rational power does not exist"
      uts := retract(uls * monomial(1,-order))@SUTS
      utsPow := (uts ** r) :: %
      monomial(1,(n :: I) * numer(r)) * utsPow
      uts := retract(uls)@SUTS
      (uts ** r) :: %

--% OutputForms

coerce(uls:%) : OUT ==
  st := getStream uls
  if not(explicitlyEmpty? st or explicitEntries? st) _
    and (nx := retractIfCan(elt getRef uls))@Union(I,"failed") case I then
    count : NNI := _$streamCount$Lisp
    degr := min(count,(nx :: I) + count + 1)
    extend(uls,degr)
  seriesToOutputForm(st,getRef uls,variable uls,center uls,1)

-----

— SULS.dotabb —

"SULS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SULS"]
"ULSCCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ULSCCAT"]
"SULS" -> "ULSCCAT"

```


domain SUP SparseUnivariatePolynomial

— SparseUnivariatePolynomial.input —

```

)set break resume
)sys rm -f SparseUnivariatePolynomial.output
)spool SparseUnivariatePolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SparseUnivariatePolynomial
--R
--R SparseUnivariatePolynomial(R: Ring) is a domain constructor
--R Abbreviation for SparseUnivariatePolynomial is SUP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SUP
--R
--R----- Operations -----
--R ?? : (%,R) -> %           ?? : (R,%) -> %
--R ?? : (%,%) -> %           ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R ?? : (%,%) -> %           ?-? : (%,%) -> %
--R -? : % -> %               ?=? : (%,%) -> Boolean
--R D : (%,(R -> R)) -> %       D : % -> %
--R D : (%,NonNegativeInteger) -> % 1 : () -> %
--R 0 : () -> %               ?? : (%,PositiveInteger) -> %
--R coefficients : % -> List(R)    coerce : R -> %
--R coerce : Integer -> %         coerce : % -> OutputForm
--R degree : % -> NonNegativeInteger differentiate : % -> %
--R ?.? : (%,%) -> %             ?.? : (%,R) -> R
--R eval : (%,%,%) -> %         eval : (%,Equation(%)) -> %
--R ground : % -> R             ground? : % -> Boolean
--R hash : % -> SingleInteger    init : () -> % if R has STEP
--R latex : % -> String         leadingCoefficient : % -> R
--R leadingMonomial : % -> %     map : ((R -> R),%) -> %
--R monomial? : % -> Boolean     monomials : % -> List(%)
--R one? : % -> Boolean         pseudoRemainder : (%,%) -> %
--R recip : % -> Union(%, "failed") reductum : % -> %
--R retract : % -> R           sample : () -> %
--R zero? : % -> Boolean       ?~=? : (%,%) -> Boolean
--R ?? : (Fraction(Integer),%) -> % if R has ALGEBRA(FRAC(INT))
--R ?? : (%,Fraction(Integer)) -> % if R has ALGEBRA(FRAC(INT))
--R ?? : (NonNegativeInteger,%) -> %

```

```

--R ***? : (% , NonNegativeInteger) -> %
--R ?/? : (% , R) -> % if R has FIELD
--R ?<? : (% , %) -> Boolean if R has ORDSET
--R ?<=? : (% , %) -> Boolean if R has ORDSET
--R ?>? : (% , %) -> Boolean if R has ORDSET
--R ?>=? : (% , %) -> Boolean if R has ORDSET
--R D : (% , (R -> R) , NonNegativeInteger) -> %
--R D : (% , List(Symbol) , List(NonNegativeInteger)) -> % if R has PDRING(SYMBOL)
--R D : (% , Symbol , NonNegativeInteger) -> % if R has PDRING(SYMBOL)
--R D : (% , List(Symbol)) -> % if R has PDRING(SYMBOL)
--R D : (% , Symbol) -> % if R has PDRING(SYMBOL)
--R D : (% , List(SingletonAsOrderedSet) , List(NonNegativeInteger)) -> %
--R D : (% , SingletonAsOrderedSet , NonNegativeInteger) -> %
--R D : (% , List(SingletonAsOrderedSet)) -> %
--R D : (% , SingletonAsOrderedSet) -> %
--R ?? : (% , NonNegativeInteger) -> %
--R associates? : (% , %) -> Boolean if R has INTDOM
--R binomThmExpt : (% , % , NonNegativeInteger) -> % if R has COMRING
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(% , "failed") if $ has CHARNZ and R has PFECAT or R has CHARNZ
--R coefficient : (% , List(SingletonAsOrderedSet) , List(NonNegativeInteger)) -> %
--R coefficient : (% , SingletonAsOrderedSet , NonNegativeInteger) -> %
--R coefficient : (% , NonNegativeInteger) -> R
--R coerce : % -> % if R has INTDOM
--R coerce : Fraction(Integer) -> % if R has ALGEBRA(FRAC(INT)) or R has RETRACT(FRAC(INT))
--R coerce : SingletonAsOrderedSet -> %
--R composite : (Fraction(% , %) -> Union(Fraction(% , "failed") if R has INTDOM
--R composite : (% , %) -> Union(% , "failed") if R has INTDOM
--R conditionP : Matrix(%) -> Union(Vector(% , "failed") if $ has CHARNZ and R has PFECAT
--R content : (% , SingletonAsOrderedSet) -> % if R has GCDDOM
--R content : % -> R if R has GCDDOM
--R convert : % -> InputForm if SingletonAsOrderedSet has KONVERT(INFORM) and R has KONVERT(INFORM)
--R convert : % -> Pattern(Integer) if SingletonAsOrderedSet has KONVERT(PATTERN(INT)) and R has KONVERT(PA
--R convert : % -> Pattern(Float) if SingletonAsOrderedSet has KONVERT(PATTERN(FLOAT)) and R has KONVERT(PA
--R degree : (% , List(SingletonAsOrderedSet)) -> List(NonNegativeInteger)
--R degree : (% , SingletonAsOrderedSet) -> NonNegativeInteger
--R differentiate : (% , (R -> R) , %) -> %
--R differentiate : (% , (R -> R)) -> %
--R differentiate : (% , (R -> R) , NonNegativeInteger) -> %
--R differentiate : (% , List(Symbol) , List(NonNegativeInteger)) -> % if R has PDRING(SYMBOL)
--R differentiate : (% , Symbol , NonNegativeInteger) -> % if R has PDRING(SYMBOL)
--R differentiate : (% , List(Symbol)) -> % if R has PDRING(SYMBOL)
--R differentiate : (% , Symbol) -> % if R has PDRING(SYMBOL)
--R differentiate : (% , NonNegativeInteger) -> %
--R differentiate : (% , List(SingletonAsOrderedSet) , List(NonNegativeInteger)) -> %
--R differentiate : (% , SingletonAsOrderedSet , NonNegativeInteger) -> %
--R differentiate : (% , List(SingletonAsOrderedSet)) -> %
--R differentiate : (% , SingletonAsOrderedSet) -> %
--R discriminant : % -> R if R has COMRING
--R discriminant : (% , SingletonAsOrderedSet) -> % if R has COMRING

```

```

--R divide : (% , %) -> Record(quotient: %, remainder: %) if R has FIELD
--R divideExponents : (% , NonNegativeInteger) -> Union(% , "failed")
--R ? . ? : (% , Fraction(%)) -> Fraction(%) if R has INTDOM
--R elt : (Fraction(%), R) -> R if R has FIELD
--R elt : (Fraction(%), Fraction(%)) -> Fraction(%) if R has INTDOM
--R euclideanSize : % -> NonNegativeInteger if R has FIELD
--R eval : (% , List(SingletonAsOrderedSet), List(%)) -> %
--R eval : (% , SingletonAsOrderedSet, %) -> %
--R eval : (% , List(SingletonAsOrderedSet), List(R)) -> %
--R eval : (% , SingletonAsOrderedSet, R) -> %
--R eval : (% , List(%), List(%)) -> %
--R eval : (% , List(Equation(%))) -> %
--R expressIdealMember : (List(%), %) -> Union(List(%), "failed") if R has FIELD
--R exquo : (% , %) -> Union(% , "failed") if R has INTDOM
--R exquo : (% , R) -> Union(% , "failed") if R has INTDOM
--R extendedEuclidean : (% , %) -> Record(coef1: %, coef2: %, generator: %) if R has FIELD
--R extendedEuclidean : (% , %, %) -> Union(Record(coef1: %, coef2: %), "failed") if R has FIELD
--R factor : % -> Factored(%) if R has PFECAT
--R factorPolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%))
--R factorSquareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%))
--R fmeq : (% , NonNegativeInteger, R, %) -> %
--R gcd : (% , %) -> % if R has GCDDOM
--R gcd : List(%) -> % if R has GCDDOM
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R integrate : % -> % if R has ALGEBRA(FRAC(INT))
--R isExpt : % -> Union(Record(var: SingletonAsOrderedSet, exponent: NonNegativeInteger), "failed")
--R isPlus : % -> Union(List(%), "failed")
--R isTimes : % -> Union(List(%), "failed")
--R karatsubaDivide : (% , NonNegativeInteger) -> Record(quotient: %, remainder: %)
--R lcm : (% , %) -> % if R has GCDDOM
--R lcm : List(%) -> % if R has GCDDOM
--R mainVariable : % -> Union(SingletonAsOrderedSet, "failed")
--R makeSUP : % -> SparseUnivariatePolynomial(R)
--R mapExponents : ((NonNegativeInteger -> NonNegativeInteger), %) -> %
--R max : (% , %) -> % if R has ORDSET
--R min : (% , %) -> % if R has ORDSET
--R minimumDegree : (% , List(SingletonAsOrderedSet)) -> List(NonNegativeInteger)
--R minimumDegree : (% , SingletonAsOrderedSet) -> NonNegativeInteger
--R minimumDegree : % -> NonNegativeInteger
--R monicDivide : (% , %) -> Record(quotient: %, remainder: %)
--R monicDivide : (% , %, SingletonAsOrderedSet) -> Record(quotient: %, remainder: %)
--R monomial : (% , List(SingletonAsOrderedSet), List(NonNegativeInteger)) -> %
--R monomial : (% , SingletonAsOrderedSet, NonNegativeInteger) -> %
--R monomial : (R, NonNegativeInteger) -> %
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed") if R has FIELD
--R multiplyExponents : (% , NonNegativeInteger) -> %
--R multivariate : (SparseUnivariatePolynomial(%), SingletonAsOrderedSet) -> %
--R multivariate : (SparseUnivariatePolynomial(R), SingletonAsOrderedSet) -> %
--R nextItem : % -> Union(% , "failed") if R has STEP
--R numberOfMonomials : % -> NonNegativeInteger

```

```

--R order : (%,% ) -> NonNegativeInteger if R has INTDOM
--R outputForm : (% ,OutputForm) -> OutputForm
--R patternMatch : (% ,Pattern(Integer),PatternMatchResult(Integer,%)) -> PatternMatchResult(Integer,% ) if S
--R patternMatch : (% ,Pattern(Float),PatternMatchResult(Float,%)) -> PatternMatchResult(Float,% ) if Singlet
--R pomopo! : (% ,R,NonNegativeInteger,% ) -> %
--R prime? : % -> Boolean if R has PFECAT
--R primitiveMonomials : % -> List(%)
--R primitivePart : (% ,SingletonAsOrderedSet) -> % if R has GCDDOM
--R primitivePart : % -> % if R has GCDDOM
--R principalIdeal : List(%) -> Record(coef: List(%),generator: % ) if R has FIELD
--R pseudoDivide : (% ,%) -> Record(coef: R,quotient: %,remainder: % ) if R has INTDOM
--R pseudoQuotient : (% ,%) -> % if R has INTDOM
--R ?quo? : (% ,%) -> % if R has FIELD
--R reducedSystem : Matrix(%) -> Matrix(R)
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(R),vec: Vector(R))
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) if R has LIN
--R reducedSystem : Matrix(%) -> Matrix(Integer) if R has LINEXP(INT)
--R ?rem? : (% ,%) -> % if R has FIELD
--R resultant : (% ,%) -> R if R has COMRING
--R resultant : (% ,%,SingletonAsOrderedSet) -> % if R has COMRING
--R retract : % -> SingletonAsOrderedSet
--R retract : % -> Integer if R has RETRACT(INT)
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(SingletonAsOrderedSet,"failed")
--R retractIfCan : % -> Union(Integer,"failed") if R has RETRACT(INT)
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(R,"failed")
--R separate : (% ,%) -> Record(primePart: %,commonPart: % ) if R has GCDDOM
--R shiftLeft : (% ,NonNegativeInteger) -> %
--R shiftRight : (% ,NonNegativeInteger) -> %
--R sizeLess? : (% ,%) -> Boolean if R has FIELD
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(%)),SparseUnivariatePolynomial(%)) -> U
--R squareFree : % -> Factored(%) if R has GCDDOM
--R squareFreePart : % -> % if R has GCDDOM
--R squareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if R has
--R subResultantGcd : (% ,%) -> % if R has INTDOM
--R subtractIfCan : (% ,%) -> Union(%, "failed")
--R totalDegree : (% ,List(SingletonAsOrderedSet)) -> NonNegativeInteger
--R totalDegree : % -> NonNegativeInteger
--R unit? : % -> Boolean if R has INTDOM
--R unitCanonical : % -> % if R has INTDOM
--R unitNormal : % -> Record(unit: %,canonical: %,associate: % ) if R has INTDOM
--R univariate : % -> SparseUnivariatePolynomial(R)
--R univariate : (% ,SingletonAsOrderedSet) -> SparseUnivariatePolynomial(%)
--R unmakeSUP : SparseUnivariatePolynomial(R) -> %
--R variables : % -> List(SingletonAsOrderedSet)
--R vectorise : (% ,NonNegativeInteger) -> Vector(R)
--R
--E 1

```

```
)spool
)lisp (bye)
```

— SparseUnivariatePolynomial.help —

=====

SparseUnivariatePolynomial examples

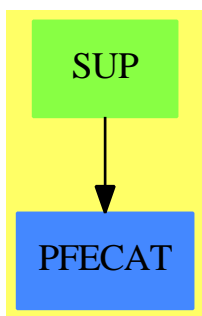
=====

This domain represents univariate polynomials over arbitrary (not necessarily commutative) coefficient rings. The variable is unspecified so that the variable displays as ? on output. If it is necessary to specify the variable name, use type UnivariatePolynomial. The representation is sparse in the sense that only non-zero terms are represented. Note that if the coefficient ring is a field, this domain forms a euclidean domain.

See Also:

- o)show SparseUnivariatePolynomial
- o)show UnivariatePolynomial

SparseUnivariatePolynomial (SUP)



See

- ⇒ “FreeModule” (FM) [7](#) on page [1013](#)
- ⇒ “PolynomialRing” (PR) [17](#) on page [2092](#)
- ⇒ “UnivariatePolynomial” (UP) [22](#) on page [2827](#)

Exports:

0	1
associates?	binomThmExpt
characteristic	charthRoot
coefficient	coefficients
coerce	composite
conditionP	content
convert	D
degree	differentiate
discriminant	divide
divideExponents	elt
euclideanSize	eval
expressIdealMember	exquo
extendedEuclidean	factor
factorPolynomial	factorSquareFreePolynomial
fmeqg	gcd
gcdPolynomial	ground
ground?	hash
init	integrate
isExpt	isPlus
isTimes	karatsubaDivide
latex	lcm
leadingCoefficient	leadingMonomial
mainVariable	makeSUP
map	mapExponents
max	min
minimumDegree	monicDivide
monomial	monomial?
monomials	multiEuclidean
multiplyExponents	multivariate
nextItem	numberOfMonomials
one?	order
outputForm	patternMatch
popopo!	prime?
primitiveMonomials	primitivePart
principalIdeal	pseudoDivide
pseudoQuotient	pseudoRemainder
recip	reducedSystem
reductum	resultant
retract	retractIfCan
sample	separate
shiftLeft	shiftRight
sizeLess?	solveLinearPolynomialEquation
squareFree	squareFreePart
squareFreePolynomial	subResultantGcd
subtractIfCan	totalDegree
totalDegree	unit?
unitCanonical	unitNormal
univariate	univariate
unmakeSUP	variables
vectorise	zero?
?*?	?**?
?+?	?-?
-?	?=?
?^?	?..?
?~=?	?/?

— domain SUP SparseUnivariatePolynomial —

```

)abbrev domain SUP SparseUnivariatePolynomial
++ Author: Dave Barton, Barry Trager
++ Description:
++ This domain represents univariate polynomials over arbitrary
++ (not necessarily commutative) coefficient rings. The variable is
++ unspecified so that the variable displays as \spad{?} on output.
++ If it is necessary to specify the variable name,
++ use type \spadtype{UnivariatePolynomial}. The representation is sparse
++ in the sense that only non-zero terms are represented.
++ Note that if the coefficient ring is a field,
++ this domain forms a euclidean domain.

SparseUnivariatePolynomial(R:Ring): UnivariatePolynomialCategory(R) with
  outputForm : (%,OutputForm) -> OutputForm
    ++ outputForm(p,var) converts the SparseUnivariatePolynomial p to
    ++ an output form (see \spadtype{OutputForm}) printed as a polynomial in the
    ++ output form variable.
  fmeq: (%,NonNegativeInteger,R,%) -> %
    ++ fmeq(p1,e,r,p2) finds x : p1 - r * x**e * p2
  == PolynomialRing(R,NonNegativeInteger)
add
--representations
Term := Record(k:NonNegativeInteger,c:R)
Rep  := List Term
p:%
n:NonNegativeInteger
np: PositiveInteger
FP ==> SparseUnivariatePolynomial %
pp,qq: FP
lpp>List FP

-- for karatsuba
kBound: NonNegativeInteger := 63
upmp := UnivariatePolynomialMultiplicationPackage(R,%)

if R has FieldOfPrimeCharacteristic then
  p ** np == p ** (np pretend NonNegativeInteger)
  p ^ np  == p ** (np pretend NonNegativeInteger)
  p ^ n   == p ** n
  p ** n  ==
    null p => 0
    zero? n => 1
    one? n => p
    (n = 1) => p
    empty? p.rest =>
      zero?(cc:=p.first.c ** n) => 0

```

```

      [[n * p.first.k, cc]]
      -- not worth doing special trick if characteristic is too small
      if characteristic()$R < 3 then return expt(p,n pretend PositiveInteger)$RepeatedSquaring(%)
      y:=%:=1
      -- break up exponent in qn * characteristic + rn
      -- exponentiating by the characteristic is fast
      rec := divide(n, characteristic()$R)
      qn:= rec.quotient
      rn:= rec.remainder
      repeat
        if rn = 1 then y := y * p
        if rn > 1 then y:= y * binomThmExpt([p.first], p.rest, rn)
        zero? qn => return y
        -- raise to the characteristic power
        p:= [[t.k * characteristic()$R , primeFrobenius(t.c)$R ]$Term for t in p]
        rec := divide(qn, characteristic()$R)
        qn:= rec.quotient
        rn:= rec.remainder
      y
y

zero?(p): Boolean == empty?(p)
-- one?(p): Boolean == not empty? p and (empty? rest p and zero? first(p).k and one? first(p).c)
one?(p): Boolean == not empty? p and (empty? rest p and zero? first(p).k and (first(p).c = 1))
ground?(p): Boolean == empty? p or (empty? rest p and zero? first(p).k)
multiplyExponents(p,n) == [ [u.k*n,u.c] for u in p]
divideExponents(p,n) ==
  null p => p
  m:= (p.first.k :: Integer exquo n::Integer)
  m case "failed" => "failed"
  u:= divideExponents(p.rest,n)
  u case "failed" => "failed"
  [[m::Integer::NonNegativeInteger,p.first.c],:u]
karatsubaDivide(p, n) ==
  zero? n => [p, 0]
  lowp: Rep := p
  highp: Rep := []
  repeat
    if empty? lowp then break
    t := first lowp
    if t.k < n then break
    lowp := rest lowp
    highp := cons([subtractIfCan(t.k,n)::NonNegativeInteger,t.c]$Term,highp)
  [ reverse highp, lowp]
shiftRight(p, n) ==
  [[subtractIfCan(t.k,n)::NonNegativeInteger,t.c]$Term for t in p]
shiftLeft(p, n) ==
  [[t.k + n,t.c]$Term for t in p]
pomopo!(p1,r,e,p2) ==

```



```

rout:=%:= []
for tm in p2 repeat
  e2:= e + tm.k
  c2:= r * tm.c
  c2 = 0 => "next term"
  while not null p1 and p1.first.k > e2 repeat
    (rout:=[p1.first,:rout]; p1:=p1.rest) --use PUSH and POP?
  null p1 or p1.first.k < e2 => rout:=[[e2,c2],:rout]
  if (u:=p1.first.c + c2) ^= 0 then rout:=[[e2, u],:rout]
  p1:=p1.rest
NRECONC(rout,p1)$Lisp

-- implementation using karatsuba algorithm conditionally
--
--  p1 * p2 ==
--  xx := p1::Rep
--  empty? xx => p1
--  yy := p2::Rep
--  empty? yy => p2
--  zero? first(xx).k => first(xx).c * p2
--  zero? first(yy).k => p1 * first(yy).c
--  (first(xx).k > kBound) and (first(yy).k > kBound) and (#xx > kBound) and (#yy > kBound) ==
--  karatsubaOnce(p1,p2)$supmp
--  xx := reverse xx
--  res : Rep := empty()
--  for tx in xx repeat res:= rep pomopo!( res,tx.c,tx.k,p2)
--  res

univariate(p:%) == p pretend SparseUnivariatePolynomial(R)
multivariate(sup:SparseUnivariatePolynomial(R),v:SingletonAsOrderedSet) ==
  sup pretend %
univariate(p:%,v:SingletonAsOrderedSet) ==
  zero? p => 0
  monomial(leadingCoefficient(p)::%,degree p) +
  univariate(reductum p,v)
multivariate(supp:SparseUnivariatePolynomial(%),v:SingletonAsOrderedSet) ==
  zero? supp => 0
  lc:=leadingCoefficient supp
  degree lc > 0 => error "bad form polynomial"
  monomial(leadingCoefficient lc,degree supp) +
  multivariate(reductum supp,v)
if R has FiniteFieldCategory and R has PolynomialFactorizationExplicit then
  RXY ==> SparseUnivariatePolynomial SparseUnivariatePolynomial R
  squareFreePolynomial pp ==
    squareFree(pp)$UnivariatePolynomialSquareFree(%,FP)
  factorPolynomial pp ==
    (generalTwoFactor(pp pretend RXY)$TwoFactorize(R))
    pretend Factored SparseUnivariatePolynomial %
  factorSquareFreePolynomial pp ==

```

```

        (generalTwoFactor(pp pretend RXY)$TwoFactorize(R))
        pretend Factored SparseUnivariatePolynomial %
gcdPolynomial(pp,qq) == gcd(pp,qq)$FP
factor p == factor(p)$DistinctDegreeFactorize(R,%)
solveLinearPolynomialEquation(lpp,pp) ==
    solveLinearPolynomialEquation(lpp, pp)$FiniteFieldSolveLinearPolynomialEquation(R,%,FP)
else if R has PolynomialFactorizationExplicit then
    import PolynomialFactorizationByRecursionUnivariate(R,%)
    solveLinearPolynomialEquation(lpp,pp)==
        solveLinearPolynomialEquationByRecursion(lpp,pp)
    factorPolynomial(pp) ==
        factorByRecursion(pp)
    factorSquareFreePolynomial(pp) ==
        factorSquareFreeByRecursion(pp)

if R has IntegralDomain then
if R has approximate then
    p1 exquo p2 ==
        null p2 => error "Division by 0"
        p2 = 1 => p1
        p1=p2 => 1
    --(p1.lastElt.c exquo p2.lastElt.c) case "failed" => "failed"
    rout:= []@List(Term)
    while not null p1 repeat
        (a:= p1.first.c exquo p2.first.c)
        a case "failed" => return "failed"
        ee:= subtractIfCan(p1.first.k, p2.first.k)
        ee case "failed" => return "failed"
        p1:= fmech(p1.rest, ee, a, p2.rest)
        rout:= [[ee,a], :rout]
        null p1 => reverse(rout)::%    -- nreverse?
        "failed"
else -- R not approximate
    p1 exquo p2 ==
        null p2 => error "Division by 0"
        p2 = 1 => p1
    --(p1.lastElt.c exquo p2.lastElt.c) case "failed" => "failed"
    rout:= []@List(Term)
    while not null p1 repeat
        (a:= p1.first.c exquo p2.first.c)
        a case "failed" => return "failed"
        ee:= subtractIfCan(p1.first.k, p2.first.k)
        ee case "failed" => return "failed"
        p1:= fmech(p1.rest, ee, a, p2.rest)
        rout:= [[ee,a], :rout]
        null p1 => reverse(rout)::%    -- nreverse?
        "failed"
fmech(p1,e,r,p2) ==          -- p1 - r * x**e * p2
    rout:=%:= []
    r:= - r

```

```

    for tm in p2 repeat
      e2:= e + tm.k
      c2:= r * tm.c
      c2 = 0 => "next term"
      while not null p1 and p1.first.k > e2 repeat
        (rout:=[p1.first,:rout]; p1:=p1.rest) --use PUSH and POP?
      null p1 or p1.first.k < e2 => rout:=[[e2,c2],:rout]
      if (u:=p1.first.c + c2) ^= 0 then rout:=[[e2, u],:rout]
      p1:=p1.rest
    NRECONC(rout,p1)$Lisp
pseudoRemainder(p1,p2) ==
  null p2 => error "PseudoDivision by Zero"
  null p1 => 0
  co:=p2.first.c;
  e:=p2.first.k;
  p2:=p2.rest;
  e1:=max(p1.first.k:Integer-e+1,0):NonNegativeInteger
  while not null p1 repeat
    if (u:=subtractIfCan(p1.first.k,e)) case "failed" then leave
    p1:=fmecg(co * p1.rest, u, p1.first.c, p2)
    e1:= (e1 - 1):NonNegativeInteger
  e1 = 0 => p1
  co ** e1 * p1
toutput(t1:Term,v:OutputForm):OutputForm ==
  t1.k = 0 => t1.c :: OutputForm
  if t1.k = 1
    then mon:= v
    else mon := v ** t1.k::OutputForm
  t1.c = 1 => mon
  t1.c = -1 and
    ((t1.c :: OutputForm) = (-1$Integer)::OutputForm)@Boolean => - mon
  t1.c::OutputForm * mon
outputForm(p:%,v:OutputForm) ==
  l: List(OutputForm)
  l:=[toutput(t,v) for t in p]
  null l => (0$Integer)::OutputForm -- else FreeModule 0 problems
  reduce("+",l)

coerce(p:%):OutputForm == outputForm(p, "?"::OutputForm)
elt(p:%,val:R) ==
  null p => 0$R
  co:=p.first.c
  n:=p.first.k
  for tm in p.rest repeat
    co:= co * val ** (n - (n:=tm.k)):NonNegativeInteger + tm.c
  n = 0 => co
  co * val ** n
elt(p:%,val:%) ==
  null p => 0$%
  coef:% := p.first.c :: %

```

```

n:=p.first.k
for tm in p.rest repeat
  coef:= coef * val ** (n-(n:=tm.k)):NonNegativeInteger+(tm.c::%)
n = 0 => coef
coef * val ** n

monicDivide(p1:%,p2:%) ==
  null p2 => error "monicDivide: division by 0"
  leadingCoefficient p2 ^= 1 => error "Divisor Not Monic"
  p2 = 1 => [p1,0]
  null p1 => [0,0]
  degree p1 < (n:=degree p2) => [0,p1]
  rout:Rep := []
  p2 := p2.rest
  while not null p1 repeat
    (u:=subtractIfCan(p1.first.k, n)) case "failed" => leave
    rout:=[u, p1.first.c], :rout]
    p1:=fmecg(p1.rest, rout.first.k, rout.first.c, p2)
  [reverse_!(rout),p1]

if R has IntegralDomain then
  discriminant(p) == discriminant(p)$PseudoRemainderSequence(R,%)
-- discriminant(p) ==
--   null p or zero?(p.first.k) => error "cannot take discriminant of constants"
--   dp:=differentiate p
--   corr:= p.first.c ** ((degree p - 1 - degree dp)::NonNegativeInteger)
--   (-1)**((p.first.k*(p.first.k-1)) quo 2):NonNegativeInteger
--   * (corr * resultant(p,dp) exquo p.first.c)::R

  subResultantGcd(p1,p2) == subResultantGcd(p1,p2)$PseudoRemainderSequence(R,%)
-- subResultantGcd(p1,p2) == --args # 0, non-coef, prim, ans not prim
--   --see algorithm 1 (p. 4) of Brown's latest (unpublished) paper
--   if p1.first.k < p2.first.k then (p1,p2):=(p2,p1)
--   p:=pseudoRemainder(p1,p2)
--   co:=1$R;
--   e:= (p1.first.k - p2.first.k):NonNegativeInteger
--   while not null p and p.first.k ^= 0 repeat
--     p1:=p2; p2:=p; p:=pseudoRemainder(p1,p2)
--     null p or p.first.k = 0 => "enuf"
--     co:=(p1.first.c ** e exquo co ** max(0, (e-1))):NonNegativeInteger)::R
--     e:= (p1.first.k - p2.first.k):NonNegativeInteger; c1:=co**e
--     p:=[tm.k,((tm.c exquo p1.first.c)::R exquo c1)::R] for tm in p]
--   if null p then p2 else 1$%

  resultant(p1,p2) == resultant(p1,p2)$PseudoRemainderSequence(R,%)
-- resultant(p1,p2) == --SubResultant PRS Algorithm
--   null p1 or null p2 => 0$R
--   0 = degree(p1) => ((first p1).c)**degree(p2)
--   0 = degree(p2) => ((first p2).c)**degree(p1)
--   if p1.first.k < p2.first.k then

```

```

--      (if odd?(p1.first.k) then p1:=-p1; (p1,p2):=(p2,p1))
--      p:=pseudoRemainder(p1,p2)
--      co:=1$R; e:=(p1.first.k-p2.first.k):NonNegativeInteger
--      while not null p repeat
--        if not odd?(e) then p:=-p
--        p1:=p2; p2:=p; p:=pseudoRemainder(p1,p2)
--        co:=(p1.first.c ** e exquo co ** max(e:Integer-1,0):NonNegativeInteger)::R
--        e:= (p1.first.k - p2.first.k):NonNegativeInteger; c1:=co**e
--        p:=(p exquo ((leadingCoefficient p1) * c1))::%
--        degree p2 > 0 => 0$R
--        (p2.first.c**e exquo co**((e-1)::NonNegativeInteger))::R
if R has GcdDomain then
  content(p) == if null p then 0$R else "gcd"/[tm.c for tm in p]
  --make CONTENT more efficient?

  primitivePart(p) ==
    null p => p
    ct :=content(p)
    unitCanonical((p exquo ct)::%)
    -- exquo present since % is now an IntegralDomain

  gcd(p1,p2) ==
    gcdPolynomial(p1 pretend SparseUnivariatePolynomial R,
      p2 pretend SparseUnivariatePolynomial R) pretend %

if R has Field then
  divide( p1, p2) ==
    zero? p2 => error "Division by 0"
    one? p2 => [p1,0]
--  (p2 = 1) => [p1,0]
  ct:=inv(p2.first.c)
  n:=p2.first.k
  p2:=p2.rest
  rout:=empty()$List(Term)
  while p1 ^= 0 repeat
    (u:=subtractIfCan(p1.first.k, n)) case "failed" => leave
    rout:=[[u, ct * p1.first.c], :rout]
    p1:=fmecg(p1.rest, rout.first.k, rout.first.c, p2)
  [reverse_!(rout),p1]

  p / co == inv(co) * p

```

— SUP.dotabb —

"SUP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SUP"]

"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]

"SUP" -> "PFECAT"

domain SUEXPR SparseUnivariatePolynomialExpressions

This domain is a hack, in some sense. What I'd really like to do - automatically - is to provide all operations supported by the coefficient domain, as long as the polynomials can be retracted to that domain, i.e., as long as they are just constants. I don't see another way to do this, unfortunately.

— SparseUnivariatePolynomialExpressions.input —

```
)set break resume
)sys rm -f SparseUnivariatePolynomialExpressions.output
)spool SparseUnivariatePolynomialExpressions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SparseUnivariatePolynomialExpressions
--R
--R SparseUnivariatePolynomialExpressions(R: Ring) is a domain constructor
--R Abbreviation for SparseUnivariatePolynomialExpressions is SUEXPR
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SUEXPR
--R
--R----- Operations -----
--R ?? : (% , R) -> %           ?? : (R , %) -> %
--R ?? : (% , %) -> %           ?? : (Integer , %) -> %
--R ?? : (PositiveInteger , %) -> %   ??? : (% , PositiveInteger) -> %
--R ?? : (% , %) -> %           ?-? : (% , %) -> %
--R -? : % -> %                 ?=? : (% , %) -> Boolean
--R D : (% , (R -> R)) -> %       D : % -> %
--R D : (% , NonNegativeInteger) -> %   1 : () -> %
--R 0 : () -> %                 ?^? : (% , PositiveInteger) -> %
--R coefficients : % -> List(R)       coerce : R -> %
--R coerce : Integer -> %             coerce : % -> OutputForm
--R degree : % -> NonNegativeInteger   differentiate : % -> %
--R ?? : (% , %) -> %               ?.? : (% , R) -> R
--R eval : (% , % , %) -> %         eval : (% , Equation(%)) -> %
--R ground : % -> R                 ground? : % -> Boolean
--R hash : % -> SingleInteger         init : () -> % if R has STEP
```

```

--R latex : % -> String
--R leadingMonomial : % -> %
--R monomial? : % -> Boolean
--R one? : % -> Boolean
--R pseudoRemainder : (%,% ) -> %
--R reductum : % -> %
--R sample : () -> %
--R ?^=? : (%,% ) -> Boolean
--R ?*? : (Fraction(Integer),%) -> % if R has ALGEBRA(FRAC(INT))
--R ?*? : (% ,Fraction(Integer)) -> % if R has ALGEBRA(FRAC(INT))
--R ?*? : (NonNegativeInteger,%) -> %
--R ?**? : (%,% ) -> % if R has TRANFUN
--R ?**? : (% ,NonNegativeInteger) -> %
--R ?/? : (% ,R) -> % if R has FIELD
--R ?<? : (%,% ) -> Boolean if R has ORDSET
--R ?<=? : (%,% ) -> Boolean if R has ORDSET
--R ?>? : (%,% ) -> Boolean if R has ORDSET
--R ?>=? : (%,% ) -> Boolean if R has ORDSET
--R D : (% ,(R -> R),NonNegativeInteger) -> %
--R D : (% ,List(Symbol),List(NonNegativeInteger)) -> % if R has PDRING(SYMBOL)
--R D : (% ,Symbol,NonNegativeInteger) -> % if R has PDRING(SYMBOL)
--R D : (% ,List(Symbol)) -> % if R has PDRING(SYMBOL)
--R D : (% ,Symbol) -> % if R has PDRING(SYMBOL)
--R D : (% ,List(SingletonAsOrderedSet),List(NonNegativeInteger)) -> %
--R D : (% ,SingletonAsOrderedSet,NonNegativeInteger) -> %
--R D : (% ,List(SingletonAsOrderedSet)) -> %
--R D : (% ,SingletonAsOrderedSet) -> %
--R ?^? : (% ,NonNegativeInteger) -> %
--R acos : % -> % if R has TRANFUN
--R acosh : % -> % if R has TRANFUN
--R acot : % -> % if R has TRANFUN
--R acoth : % -> % if R has TRANFUN
--R acsc : % -> % if R has TRANFUN
--R acsch : % -> % if R has TRANFUN
--R asec : % -> % if R has TRANFUN
--R asech : % -> % if R has TRANFUN
--R asin : % -> % if R has TRANFUN
--R asinh : % -> % if R has TRANFUN
--R associates? : (%,% ) -> Boolean if R has INTDOM
--R atan : % -> % if R has TRANFUN
--R atanh : % -> % if R has TRANFUN
--R binomThmExpt : (% ,%,NonNegativeInteger) -> % if R has COMRING
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(% ,"failed") if $ has CHARNZ and R has PFECAT or R has CHARNZ
--R coefficient : (% ,List(SingletonAsOrderedSet),List(NonNegativeInteger)) -> %
--R coefficient : (% ,SingletonAsOrderedSet,NonNegativeInteger) -> %
--R coefficient : (% ,NonNegativeInteger) -> R
--R coerce : % -> % if R has INTDOM
--R coerce : Fraction(Integer) -> % if R has ALGEBRA(FRAC(INT)) or R has RETRACT(FRAC(INT))
--R coerce : SingletonAsOrderedSet -> %
--R leadingCoefficient : % -> R
--R map : ((R -> R),%) -> %
--R monomials : % -> List(% )
--R pi : () -> % if R has TRANFUN
--R recip : % -> Union(% ,"failed")
--R retract : % -> R
--R zero? : % -> Boolean

```

```

--R composite : (Fraction(%),%) -> Union(Fraction(%),"failed") if R has INTDOM
--R composite : (%,%)-> Union(%,"failed") if R has INTDOM
--R conditionP : Matrix(%) -> Union(Vector(%),"failed") if $ has CHARNZ and R has PFECAT
--R content : (% ,SingletonAsOrderedSet) -> % if R has GCDDOM
--R content : % -> R if R has GCDDOM
--R convert : % -> InputForm if SingletonAsOrderedSet has KONVERT(INFORM) and R has KONVERT(INFORM)
--R convert : % -> Pattern(Integer) if SingletonAsOrderedSet has KONVERT(PATTERN(INT)) and R has KONVERT(PA
--R convert : % -> Pattern(Float) if SingletonAsOrderedSet has KONVERT(PATTERN(FLOAT)) and R has KONVERT(PA
--R cos : % -> % if R has TRANFUN
--R cosh : % -> % if R has TRANFUN
--R cot : % -> % if R has TRANFUN
--R coth : % -> % if R has TRANFUN
--R csc : % -> % if R has TRANFUN
--R csch : % -> % if R has TRANFUN
--R degree : (% ,List(SingletonAsOrderedSet)) -> List(NonNegativeInteger)
--R degree : (% ,SingletonAsOrderedSet) -> NonNegativeInteger
--R differentiate : (% ,(R -> R),%) -> %
--R differentiate : (% ,(R -> R)) -> %
--R differentiate : (% ,(R -> R),NonNegativeInteger) -> %
--R differentiate : (% ,List(Symbol),List(NonNegativeInteger)) -> % if R has PDRING(SYMBOL)
--R differentiate : (% ,Symbol,NonNegativeInteger) -> % if R has PDRING(SYMBOL)
--R differentiate : (% ,List(Symbol)) -> % if R has PDRING(SYMBOL)
--R differentiate : (% ,Symbol) -> % if R has PDRING(SYMBOL)
--R differentiate : (% ,NonNegativeInteger) -> %
--R differentiate : (% ,List(SingletonAsOrderedSet),List(NonNegativeInteger)) -> %
--R differentiate : (% ,SingletonAsOrderedSet,NonNegativeInteger) -> %
--R differentiate : (% ,List(SingletonAsOrderedSet)) -> %
--R differentiate : (% ,SingletonAsOrderedSet) -> %
--R discriminant : % -> R if R has COMRING
--R discriminant : (% ,SingletonAsOrderedSet) -> % if R has COMRING
--R divide : (% ,%) -> Record(quotient: %,remainder: %) if R has FIELD
--R divideExponents : (% ,NonNegativeInteger) -> Union(%,"failed")
--R ?? : (% ,Fraction(%)) -> Fraction(%) if R has INTDOM
--R elt : (Fraction(%),R) -> R if R has FIELD
--R elt : (Fraction(%),Fraction(%)) -> Fraction(%) if R has INTDOM
--R euclideanSize : % -> NonNegativeInteger if R has FIELD
--R eval : (% ,List(SingletonAsOrderedSet),List(%)) -> %
--R eval : (% ,SingletonAsOrderedSet,%) -> %
--R eval : (% ,List(SingletonAsOrderedSet),List(R)) -> %
--R eval : (% ,SingletonAsOrderedSet,R) -> %
--R eval : (% ,List(%),List(%)) -> %
--R eval : (% ,List(Equation(%))) -> %
--R exp : % -> % if R has TRANFUN
--R expressIdealMember : (List(%),%) -> Union(List(%),"failed") if R has FIELD
--R exquo : (% ,%) -> Union(%,"failed") if R has INTDOM
--R exquo : (% ,R) -> Union(%,"failed") if R has INTDOM
--R extendedEuclidean : (% ,%) -> Record(coef1: %,coef2: %,generator: %) if R has FIELD
--R extendedEuclidean : (% ,%,%) -> Union(Record(coef1: %,coef2: %),"failed") if R has FIELD
--R factor : % -> Factored(%) if R has PFECAT
--R factorPolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if R has PF

```



```

--R factorSquareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%))
--R gcd : (%,%) -> % if R has GCDDOM
--R gcd : List(%) -> % if R has GCDDOM
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R integrate : % -> % if R has ALGEBRA(FRAC(INT))
--R isExpt : % -> Union(Record(var: SingletonAsOrderedSet,exponent: NonNegativeInteger),"failed")
--R isPlus : % -> Union(List(%) ,"failed")
--R isTimes : % -> Union(List(%) ,"failed")
--R karatsubaDivide : (%,NonNegativeInteger) -> Record(quotient: %,remainder: %)
--R lcm : (%,%) -> % if R has GCDDOM
--R lcm : List(%) -> % if R has GCDDOM
--R log : % -> % if R has TRANFUN
--R mainVariable : % -> Union(SingletonAsOrderedSet,"failed")
--R makeSUP : % -> SparseUnivariatePolynomial(R)
--R mapExponents : ((NonNegativeInteger -> NonNegativeInteger),%) -> %
--R max : (%,%) -> % if R has ORDSET
--R min : (%,%) -> % if R has ORDSET
--R minimumDegree : (%,List(SingletonAsOrderedSet)) -> List(NonNegativeInteger)
--R minimumDegree : (%,SingletonAsOrderedSet) -> NonNegativeInteger
--R minimumDegree : % -> NonNegativeInteger
--R monicDivide : (%,%) -> Record(quotient: %,remainder: %)
--R monicDivide : (%,%,SingletonAsOrderedSet) -> Record(quotient: %,remainder: %)
--R monomial : (%,List(SingletonAsOrderedSet),List(NonNegativeInteger)) -> %
--R monomial : (%,SingletonAsOrderedSet,NonNegativeInteger) -> %
--R monomial : (R,NonNegativeInteger) -> %
--R multiEuclidean : (List(%),%) -> Union(List(%) ,"failed") if R has FIELD
--R multiplyExponents : (%,NonNegativeInteger) -> %
--R multivariate : (SparseUnivariatePolynomial(%),SingletonAsOrderedSet) -> %
--R multivariate : (SparseUnivariatePolynomial(R),SingletonAsOrderedSet) -> %
--R nextItem : % -> Union(%) ,"failed" if R has STEP
--R numberOfMonomials : % -> NonNegativeInteger
--R order : (%,%) -> NonNegativeInteger if R has INTDOM
--R patternMatch : (%,Pattern(Integer),PatternMatchResult(Integer,%)) -> PatternMatchResult(Integer,%)
--R patternMatch : (%,Pattern(Float),PatternMatchResult(Float,%)) -> PatternMatchResult(Float,%)
--R pomopo! : (%,R,NonNegativeInteger,%) -> %
--R prime? : % -> Boolean if R has PFECAT
--R primitiveMonomials : % -> List(%)
--R primitivePart : (%,SingletonAsOrderedSet) -> % if R has GCDDOM
--R primitivePart : % -> % if R has GCDDOM
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %) if R has FIELD
--R pseudoDivide : (%,%) -> Record(coef: R,quotient: %,remainder: %) if R has INTDOM
--R pseudoQuotient : (%,%) -> % if R has INTDOM
--R ?quo? : (%,%) -> % if R has FIELD
--R reducedSystem : Matrix(%) -> Matrix(R)
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(R),vec: Vector(R))
--R reducedSystem : (Matrix(%),Vector(%)) -> Record(mat: Matrix(Integer),vec: Vector(Integer)) if R has LINEXP(INT)
--R ?rem? : (%,%) -> % if R has FIELD
--R resultant : (%,%) -> R if R has COMRING
--R resultant : (%,%,SingletonAsOrderedSet) -> % if R has COMRING

```

```

--R retract : % -> SingletonAsOrderedSet
--R retract : % -> Integer if R has RETRACT(INT)
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(SingletonAsOrderedSet,"failed")
--R retractIfCan : % -> Union(Integer,"failed") if R has RETRACT(INT)
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(R,"failed")
--R sec : % -> % if R has TRANFUN
--R sech : % -> % if R has TRANFUN
--R separate : (%,% ) -> Record(primePart: %,commonPart: %) if R has GCDDOM
--R shiftLeft : (% ,NonNegativeInteger) -> %
--R shiftRight : (% ,NonNegativeInteger) -> %
--R sin : % -> % if R has TRANFUN
--R sinh : % -> % if R has TRANFUN
--R sizeLess? : (%,% ) -> Boolean if R has FIELD
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(%)),SparseUnivariatePolynomial(%)) -> U
--R squareFree : % -> Factored(%) if R has GCDDOM
--R squareFreePart : % -> % if R has GCDDOM
--R squareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if R has
--R subResultantGcd : (%,% ) -> % if R has INTDOM
--R subtractIfCan : (%,% ) -> Union(%, "failed")
--R tan : % -> % if R has TRANFUN
--R tanh : % -> % if R has TRANFUN
--R totalDegree : (% ,List(SingletonAsOrderedSet)) -> NonNegativeInteger
--R totalDegree : % -> NonNegativeInteger
--R unit? : % -> Boolean if R has INTDOM
--R unitCanonical : % -> % if R has INTDOM
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %) if R has INTDOM
--R univariate : % -> SparseUnivariatePolynomial(R)
--R univariate : (% ,SingletonAsOrderedSet) -> SparseUnivariatePolynomial(%)
--R unmakeSUP : SparseUnivariatePolynomial(R) -> %
--R variables : % -> List(SingletonAsOrderedSet)
--R vectorise : (% ,NonNegativeInteger) -> Vector(R)
--R
--E 1

)spool
)lisp (bye)

```

— SparseUnivariatePolynomialExpressions.help —

```

=====
SparseUnivariatePolynomialExpressions examples
=====

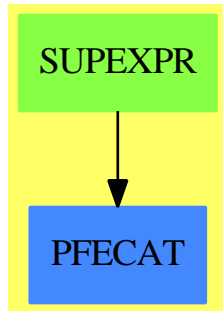
```

This domain has no description

See Also:

- o `)show SparseUnivariatePolynomialExpressions`

SparseUnivariatePolynomialExpressions (SUPEXPR)



Exports:

0	1	acos
acosh	acot	acoth
acsc	acsch	asec
asech	asin	asinh
associates?	atan	atanh
binomThmExpt	characteristic	charthRoot
coefficient	coefficients	coerce
composite	conditionP	content
convert	cos	cosh
cot	coth	csc
csch	D	degree
differentiate	discriminant	divide
divideExponents	elt	euclideanSize
eval	exp	expressIdealMember
exquo	extendedEuclidean	factor
factorPolynomial	factorSquareFreePolynomial	gcd
gcdPolynomial	ground	ground?
hash	init	integrate
isExpt	isPlus	isTimes
karatsubaDivide	latex	lcm
leadingCoefficient	leadingMonomial	log
mainVariable	makeSUP	map
mapExponents	max	min
minimumDegree	monicDivide	monomial
monomial?	monomials	multiEuclidean
multiplyExponents	multivariate	nextItem
numberOfMonomials	one?	order
patternMatch	pi	popo!
prime?	primitiveMonomials	primitivePart
principalIdeal	pseudoDivide	pseudoQuotient
pseudoRemainder	recip	reducedSystem
reductum	resultant	retract
retractIfCan	sample	sec
sech	separate	shiftLeft
shiftRight	sin	sinh
sizeLess?	solveLinearPolynomialEquation	squareFree
squareFreePart	squareFreePolynomial	subResultantGcd
subtractIfCan	tan	tanh
totalDegree	unit?	unitCanonical
unitNormal	univariate	unmakeSUP
variables	vectorise	zero?
?*?	?**?	?+?
?-?	-?	?=?
?^?	??	?~=?
?/?	?<?	?<=?
?>?	?>=?	?quo?
?rem?		

— domain SUPEXPR SparseUnivariatePolynomialExpressions —

```

)abbrev domain SUPEXPR SparseUnivariatePolynomialExpressions
++ Author: Mark Botch
++ Description:
++ This domain has no description

SparseUnivariatePolynomialExpressions(R: Ring): Exports == Implementation where

  Exports == UnivariatePolynomialCategory R with

    if R has TranscendentalFunctionCategory
    then TranscendentalFunctionCategory

  Implementation == SparseUnivariatePolynomial R add

    if R has TranscendentalFunctionCategory then
      log(p: %): % ==
        ground? p => coerce log ground p
        output(hconcat("log p for p= ", p::OutputForm))$OutputPackage
        error "SUPTRAFUN: log only defined for elements of the coefficient ring"

      exp(p: %): % ==
        ground? p => coerce exp ground p
        output(hconcat("exp p for p= ", p::OutputForm))$OutputPackage
        error "SUPTRAFUN: exp only defined for elements of the coefficient ring"
      sin(p: %): % ==
        ground? p => coerce sin ground p
        output(hconcat("sin p for p= ", p::OutputForm))$OutputPackage
        error "SUPTRAFUN: sin only defined for elements of the coefficient ring"
      asin(p: %): % ==
        ground? p => coerce asin ground p
        output(hconcat("asin p for p= ", p::OutputForm))$OutputPackage
        error "SUPTRAFUN: asin only defined for elements of the coefficient ring"
      cos(p: %): % ==
        ground? p => coerce cos ground p
        output(hconcat("cos p for p= ", p::OutputForm))$OutputPackage
        error "SUPTRAFUN: cos only defined for elements of the coefficient ring"
      acos(p: %): % ==
        ground? p => coerce acos ground p
        output(hconcat("acos p for p= ", p::OutputForm))$OutputPackage
        error "SUPTRAFUN: acos only defined for elements of the coefficient ring"

```

— SUPEXPR.dotabb —

"SUPEXPR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SUPEXPR"]

"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
 "SUEXPR" -> "PFECAT"

domain SUPXS SparseUnivariatePuisseuxSeries

— SparseUnivariatePuisseuxSeries.input —

```
)set break resume
)sys rm -f SparseUnivariatePuisseuxSeries.output
)spool SparseUnivariatePuisseuxSeries.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SparseUnivariatePuisseuxSeries
--R
--R SparseUnivariatePuisseuxSeries(Coef: Ring,var: Symbol,cen: Coef) is a domain constructor
--R Abbreviation for SparseUnivariatePuisseuxSeries is SUPXS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SUPXS
--R
--R----- Operations -----
--R ??? : (Coef,%) -> %          ??? : (%,Coef) -> %
--R ??? : (%,%) -> %           ??? : (Integer,%) -> %
--R ??? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %           ?-? : (%,%) -> %
--R -? : % -> %                ?? : (%,%) -> Boolean
--R 1 : () -> %                0 : () -> %
--R ?? : (%,PositiveInteger) -> %   center : % -> Coef
--R coerce : Variable(var) -> %     coerce : Integer -> %
--R coerce : % -> OutputForm        complete : % -> %
--R degree : % -> Fraction(Integer) hash : % -> SingleInteger
--R latex : % -> String             leadingCoefficient : % -> Coef
--R leadingMonomial : % -> %        map : ((Coef -> Coef),%) -> %
--R monomial? : % -> Boolean        one? : % -> Boolean
--R order : % -> Fraction(Integer) pole? : % -> Boolean
--R recip : % -> Union(%, "failed") reductum : % -> %
--R sample : () -> %              variable : % -> Symbol
--R zero? : % -> Boolean          ?~=? : (%,%) -> Boolean
--R ??? : (%,Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (Fraction(Integer),%) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (NonNegativeInteger,%) -> %
--R ??? : (%,Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
```

```

--R ??? : (%,%) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (%,Integer) -> % if Coef has FIELD
--R ??? : (%,NonNegativeInteger) -> %
--R ??/? : (%,%) -> % if Coef has FIELD
--R ??/? : (%,Coef) -> % if Coef has FIELD
--R D : % -> % if Coef has *: (Fraction(Integer),Coef) -> Coef
--R D : (%,NonNegativeInteger) -> % if Coef has *: (Fraction(Integer),Coef) -> Coef
--R D : (%,Symbol) -> % if Coef has *: (Fraction(Integer),Coef) -> Coef and Coef has PDRING(SYMB
--R D : (%,List(Symbol)) -> % if Coef has *: (Fraction(Integer),Coef) -> Coef and Coef has PDRIN
--R D : (%,Symbol,NonNegativeInteger) -> % if Coef has *: (Fraction(Integer),Coef) -> Coef and C
--R D : (%,List(Symbol),List(NonNegativeInteger)) -> % if Coef has *: (Fraction(Integer),Coef) -
--R ?? : (%,Integer) -> % if Coef has FIELD
--R ?? : (%,NonNegativeInteger) -> %
--R acos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acoth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R approximate : (%,Fraction(Integer)) -> Coef if Coef has **: (Coef,Fraction(Integer)) -> Coef
--R asec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R associates? : (%,%) -> Boolean if Coef has INTDOM
--R atan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R atanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if Coef has CHARNZ
--R coefficient : (%,Fraction(Integer)) -> Coef
--R coerce : % -> % if Coef has INTDOM
--R coerce : Fraction(Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R coerce : SparseUnivariateTaylorSeries(Coef,var,cen) -> %
--R coerce : SparseUnivariateLaurentSeries(Coef,var,cen) -> %
--R coerce : Coef -> % if Coef has COMRING
--R cos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R coth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R differentiate : (%,Variable(var)) -> %
--R differentiate : % -> % if Coef has *: (Fraction(Integer),Coef) -> Coef
--R differentiate : (%,NonNegativeInteger) -> % if Coef has *: (Fraction(Integer),Coef) -> Coef
--R differentiate : (%,Symbol) -> % if Coef has *: (Fraction(Integer),Coef) -> Coef and Coef has
--R differentiate : (%,List(Symbol)) -> % if Coef has *: (Fraction(Integer),Coef) -> Coef and Co
--R differentiate : (%,Symbol,NonNegativeInteger) -> % if Coef has *: (Fraction(Integer),Coef) -
--R differentiate : (%,List(Symbol),List(NonNegativeInteger)) -> % if Coef has *: (Fraction(Inte
--R divide : (%,%) -> Record(quotient: %,remainder: %) if Coef has FIELD
--R ?.? : (%,%) -> % if Fraction(Integer) has SGROUP

```

```

--R ?.? : (% , Fraction(Integer)) -> Coef
--R euclideanSize : % -> NonNegativeInteger if Coef has FIELD
--R eval : (% , Coef) -> Stream(Coef) if Coef has **: (Coef, Fraction(Integer)) -> Coef
--R exp : % -> % if Coef has ALGEBRA(FRAC(INT))
--R expressIdealMember : (List(%), %) -> Union(List(%), "failed") if Coef has FIELD
--R exquo : (% , %) -> Union(% , "failed") if Coef has INTDOM
--R extend : (% , Fraction(Integer)) -> %
--R extendedEuclidean : (% , %) -> Record(coef1: % , coef2: % , generator: %) if Coef has FIELD
--R extendedEuclidean : (% , % , %) -> Union(Record(coef1: % , coef2: % , "failed") if Coef has FIELD
--R factor : % -> Factored(%) if Coef has FIELD
--R gcd : (% , %) -> % if Coef has FIELD
--R gcd : List(%) -> % if Coef has FIELD
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R integrate : (% , Variable(var)) -> % if Coef has ALGEBRA(FRAC(INT))
--R integrate : (% , Symbol) -> % if Coef has integrate: (Coef, Symbol) -> Coef and Coef has variables: Coef -
--R integrate : % -> % if Coef has ALGEBRA(FRAC(INT))
--R inv : % -> % if Coef has FIELD
--R laurent : % -> SparseUnivariateLaurentSeries(Coef, var, cen)
--R laurentIfCan : % -> Union(SparseUnivariateLaurentSeries(Coef, var, cen), "failed")
--R laurentRep : % -> SparseUnivariateLaurentSeries(Coef, var, cen)
--R lcm : (% , %) -> % if Coef has FIELD
--R lcm : List(%) -> % if Coef has FIELD
--R log : % -> % if Coef has ALGEBRA(FRAC(INT))
--R monomial : (% , List(SingletonAsOrderedSet), List(Fraction(Integer))) -> %
--R monomial : (% , SingletonAsOrderedSet, Fraction(Integer)) -> %
--R monomial : (Coef, Fraction(Integer)) -> %
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed") if Coef has FIELD
--R multiplyExponents : (% , Fraction(Integer)) -> %
--R multiplyExponents : (% , PositiveInteger) -> %
--R nthRoot : (% , Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R order : (% , Fraction(Integer)) -> Fraction(Integer)
--R pi : () -> % if Coef has ALGEBRA(FRAC(INT))
--R prime? : % -> Boolean if Coef has FIELD
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %) if Coef has FIELD
--R puseux : (Fraction(Integer), SparseUnivariateLaurentSeries(Coef, var, cen)) -> %
--R ?quo? : (% , %) -> % if Coef has FIELD
--R rationalPower : % -> Fraction(Integer)
--R ?rem? : (% , %) -> % if Coef has FIELD
--R retract : % -> SparseUnivariateTaylorSeries(Coef, var, cen)
--R retract : % -> SparseUnivariateLaurentSeries(Coef, var, cen)
--R retractIfCan : % -> Union(SparseUnivariateTaylorSeries(Coef, var, cen), "failed")
--R retractIfCan : % -> Union(SparseUnivariateLaurentSeries(Coef, var, cen), "failed")
--R sec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R series : (NonNegativeInteger, Stream(Record(k: Fraction(Integer), c: Coef))) -> %
--R sin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sizeLess? : (% , %) -> Boolean if Coef has FIELD
--R sqrt : % -> % if Coef has ALGEBRA(FRAC(INT))
--R squareFree : % -> Factored(%) if Coef has FIELD

```



```

--R squareFreePart : % -> % if Coef has FIELD
--R subtractIfCan : (%,% ) -> Union(%, "failed")
--R tan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R tanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R terms : % -> Stream(Record(k: Fraction(Integer), c: Coef))
--R truncate : (% , Fraction(Integer), Fraction(Integer)) -> %
--R truncate : (% , Fraction(Integer)) -> %
--R unit? : % -> Boolean if Coef has INTDOM
--R unitCanonical : % -> % if Coef has INTDOM
--R unitNormal : % -> Record(unit: %, canonical: %, associate: %) if Coef has INTDOM
--R variables : % -> List(SingletonAsOrderedSet)
--R
--E 1

)spool
)lisp (bye)

```

— SparseUnivariatePuisseuxSeries.help —

```

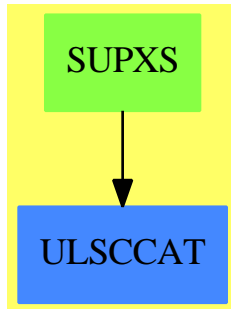
=====
SparseUnivariatePuisseuxSeries examples
=====

```

SparseUnivariatePuisseuxSeries is a domain representing Puiseux series in one variable with coefficients in an arbitrary ring. The parameters of the type specify the coefficient ring, the power series variable, and the center of the power series expansion. For example, SparseUnivariatePuisseuxSeries(Integer,x,3) represents Puiseux series in $(x - 3)$ with Integer coefficients.

See Also:

```
o )show SparseUnivariatePuisseuxSeries
```

SparseUnivariatePuisseuxSeries (SUPXS)**Exports:**

0	1	acos	acosh
acot	acoth	acsc	acsch
approximate	asec	asech	asin
asinh	associates?	atan	atanh
characteristic	charthRoot	center	coefficient
coerce	complete	cos	cosh
cot	coth	csc	csch
D	degree	differentiate	divide
euclideanSize	eval	exp	expressIdealMember
exquo	extend	extendedEuclidean	factor
gcd	gcdPolynomial	hash	integrate
inv	latex	laurent	laurentIfCan
laurentRep	lcm	leadingCoefficient	leadingMonomial
log	map	monomial	monomial?
multiEuclidean	multiplyExponents	nthRoot	one?
order	pi	pole?	prime?
principalIdeal	puiseux	rationalPower	recip
reductum	retract	retractIfCan	sample
sec	sech	series	sin
sinh	sizeLess?	sqrt	squareFree
squareFreePart	subtractIfCan	tan	tanh
terms	truncate	unit?	unitCanonical
unitNormal	variable	variables	zero?
?.?	?*?	?**?	?+?
?-?	-?	?=?	?^?
?~=?	?/?	?quo?	?rem?

— domain SUPXS SparseUnivariatePuisseuxSeries —

```

)abbrev domain SUPXS SparseUnivariatePuisseuxSeries
++ Author: Clifton J. Williamson
++ Date Created: 11 November 1994
  
```

```

++ Date Last Updated: 28 February 1995
++ Description:
++ Sparse Puiseux series in one variable
++ \spadtype{SparseUnivariatePuisseuxSeries} is a domain representing Puiseux
++ series in one variable with coefficients in an arbitrary ring. The
++ parameters of the type specify the coefficient ring, the power series
++ variable, and the center of the power series expansion. For example,
++ \spad{SparseUnivariatePuisseuxSeries(Integer,x,3)} represents Puiseux
++ series in \spad{(x - 3)} with \spadtype{Integer} coefficients.

SparseUnivariatePuisseuxSeries(Coef,var,cen): Exports == Implementation where
  Coef : Ring
  var  : Symbol
  cen  : Coef
  I    ==> Integer
  NNI  ==> NonNegativeInteger
  OUT  ==> OutputForm
  RN   ==> Fraction Integer
  SUTS ==> SparseUnivariateTaylorSeries(Coef,var,cen)
  SULS ==> SparseUnivariateLaurentSeries(Coef,var,cen)
  SUPS ==> InnerSparseUnivariatePowerSeries(Coef)

Exports ==> Join(UnivariatePuisseuxSeriesConstructorCategory(Coef,SULS),_
  RetractableTo SUTS) with
  coerce: Variable(var) -> %
    ++ coerce(var) converts the series variable \spad{var} into a
    ++ Puiseux series.
  differentiate: (%,Variable(var)) -> %
    ++ \spad{differentiate(f(x),x)} returns the derivative of
    ++ \spad{f(x)} with respect to \spad{x}.
  if Coef has Algebra Fraction Integer then
    integrate: (%,Variable(var)) -> %
      ++ \spad{integrate(f(x))} returns an anti-derivative of the power
      ++ series \spad{f(x)} with constant coefficient 0.
      ++ We may integrate a series when we can divide coefficients
      ++ by integers.

Implementation ==> UnivariatePuisseuxSeriesConstructor(Coef,SULS) add

Rep := Record(expon:RN,lSeries:SULS)

getExpon: % -> RN
getExpon pxs == pxs.expon

variable x == var
center   x == cen

coerce(v: Variable(var)) ==
  zero? cen => monomial(1,1)
  monomial(1,1) + monomial(cen,0)

```

```

coerce(uts:SUTS) == uts :: SULS :: %

retractIfCan(upxs:%):Union(SUTS,"failed") ==
  (uls := retractIfCan(upxs)@Union(SULS,"failed")) case "failed" =>
    "failed"
  retractIfCan(uls :: SULS)@Union(SUTS,"failed")

if Coef has "*": (Fraction Integer, Coef) -> Coef then
  differentiate(upxs:%,v:Variable(var)) == differentiate upxs

if Coef has Algebra Fraction Integer then
  integrate(upxs:%,v:Variable(var)) == integrate upxs

--% OutputForms

coerce(x:%): OUT ==
  sups : SUPS := laurentRep(x) pretend SUPS
  st := getStream sups; refer := getRef sups
  if not(explicitlyEmpty? st or explicitEntries? st) _
    and (nx := retractIfCan(elt refer)@Union(I,"failed")) case I then
    count : NNI := _$streamCount$Lisp
    degr := min(count,(nx :: I) + count + 1)
    extend(sups,degr)
  seriesToOutputForm(st,refer,variable x,center x,rationalPower x)



---



— SUPXS.dotabb —

"SUPXS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SUPXS"]
"ULSCCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ULSCCAT"]
"SUPXS" -> "ULSCCAT"



---



```

domain ORESUP SparseUnivariateSkewPolynomial

— SparseUnivariateSkewPolynomial.input —

```

)set break resume
)sys rm -f SparseUnivariateSkewPolynomial.output
)spool SparseUnivariateSkewPolynomial.output
)set message test on

```

```

)set message auto off
)clear all

--S 1 of 1
)show SparseUnivariateSkewPolynomial
--R
--R SparseUnivariateSkewPolynomial(R: Ring, sigma: Automorphism(R), delta: (R -> R)) is a domain
--R Abbreviation for SparseUnivariateSkewPolynomial is ORESUP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ORESUP
--R
--R----- Operations -----
--R ???: (R,%) -> %          ??: (% ,R) -> %
--R ???: (% ,%) -> %          ??: (Integer,%) -> %
--R ???: (PositiveInteger,%) -> %    ????: (% ,PositiveInteger) -> %
--R ?+?: (% ,%) -> %          ?-?: (% ,%) -> %
--R -?: % -> %                ?=: (% ,%) -> Boolean
--R 1 : () -> %                0 : () -> %
--R ?^?: (% ,PositiveInteger) -> %    apply : (% ,R,R) -> R
--R coefficients : % -> List(R)        coerce : R -> %
--R coerce : Integer -> %              coerce : % -> OutputForm
--R degree : % -> NonNegativeInteger    hash : % -> SingleInteger
--R latex : % -> String                leadingCoefficient : % -> R
--R one? : % -> Boolean                recip : % -> Union(% , "failed")
--R reductum : % -> %                retract : % -> R
--R sample : () -> %                zero? : % -> Boolean
--R ~=?: (% ,%) -> Boolean
--R ???: (NonNegativeInteger,%) -> %
--R ????: (% ,NonNegativeInteger) -> %
--R ?^?: (% ,NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R coefficient : (% ,NonNegativeInteger) -> R
--R coerce : Fraction(Integer) -> % if R has RETRACT(FRAC(INT))
--R content : % -> R if R has GCDDOM
--R exquo : (% ,R) -> Union(% , "failed") if R has INTDOM
--R leftDivide : (% ,%) -> Record(quotient: % , remainder: %) if R has FIELD
--R leftExactQuotient : (% ,%) -> Union(% , "failed") if R has FIELD
--R leftExtendedGcd : (% ,%) -> Record(coef1: % , coef2: % , generator: %) if R has FIELD
--R leftGcd : (% ,%) -> % if R has FIELD
--R leftLcm : (% ,%) -> % if R has FIELD
--R leftQuotient : (% ,%) -> % if R has FIELD
--R leftRemainder : (% ,%) -> % if R has FIELD
--R minimumDegree : % -> NonNegativeInteger
--R monicLeftDivide : (% ,%) -> Record(quotient: % , remainder: %) if R has INTDOM
--R monicRightDivide : (% ,%) -> Record(quotient: % , remainder: %) if R has INTDOM
--R monomial : (R, NonNegativeInteger) -> %
--R outputForm : (% , OutputForm) -> OutputForm
--R primitivePart : % -> % if R has GCDDOM
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retract : % -> Integer if R has RETRACT(INT)

```

```

--R retractIfCan : % -> Union(R,"failed")
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(Integer,"failed") if R has RETRACT(INT)
--R rightDivide : (%,% ) -> Record(quotient: %,remainder: %) if R has FIELD
--R rightExactQuotient : (%,% ) -> Union(%,"failed") if R has FIELD
--R rightExtendedGcd : (%,% ) -> Record(coef1: %,coef2: %,generator: %) if R has FIELD
--R rightGcd : (%,% ) -> % if R has FIELD
--R rightLcm : (%,% ) -> % if R has FIELD
--R rightQuotient : (%,% ) -> % if R has FIELD
--R rightRemainder : (%,% ) -> % if R has FIELD
--R subtractIfCan : (%,% ) -> Union(%,"failed")
--R
--E 1

```

```

)spool
)lisp (bye)

```

— SparseUnivariateSkewPolynomial.help —

```

=====
SparseUnivariateSkewPolynomial examples
=====

```

This is the domain of sparse univariate skew polynomials over an Ore coefficient field.

The multiplication is given by $x a = \sigma(a) x + \delta a$.

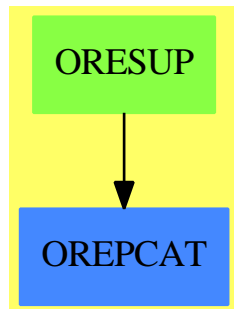
See Also:

```

o )show SparseUnivariateSkewPolynomial

```

SparseUnivariateSkewPolynomial (ORESUP)



See

⇒ “Automorphism” (AUTOMOR) [2](#) on page [245](#)

⇒ “UnivariateSkewPolynomial” (OREUP) [22](#) on page [2872](#)

Exports:

0	1	apply	characteristic	coefficient
coefficients	coerce	content	degree	exquo
hash	latex	leadingCoefficient	leftDivide	leftExactQuotient
leftExtendedGcd	leftGcd	leftLcm	leftQuotient	leftRemainder
minimumDegree	monicLeftDivide	monicRightDivide	monomial	one?
outputForm	primitivePart	recip	reductum	retract
retractIfCan	rightDivide	rightExactQuotient	rightExtendedGcd	rightGcd
rightLcm	rightQuotient	rightRemainder	sample	subtractIfCan
zero?	?*?	?**?	?+?	?-?
-?	?=?	?^?	?~=?	

— domain ORESUP SparseUnivariateSkewPolynomial —

```

)abbrev domain ORESUP SparseUnivariateSkewPolynomial
++ Author: Manuel Bronstein
++ Date Created: 19 October 1993
++ Date Last Updated: 1 February 1994
++ Description:
++ This is the domain of sparse univariate skew polynomials over an Ore
++ coefficient field.
++ The multiplication is given by \spad{x a = \sigma(a) x + \delta a}.

SparseUnivariateSkewPolynomial(R:Ring, sigma:Automorphism R, delta: R -> R):
  UnivariateSkewPolynomialCategory R with
    outputForm: (%, OutputForm) -> OutputForm
      ++ outputForm(p, x) returns the output form of p using x for the
      ++ otherwise anonymous variable.
  == SparseUnivariatePolynomial R add
    import UnivariateSkewPolynomialCategoryOps(R, %)
  
```

```

x:% * y:%      == times(x, y, sigma, delta)
apply(p, c, r) == apply(p, c, r, sigma, delta)

if R has IntegralDomain then
  monicLeftDivide(a, b) == monicLeftDivide(a, b, sigma)
  monicRightDivide(a, b) == monicRightDivide(a, b, sigma)

if R has Field then
  leftDivide(a, b) == leftDivide(a, b, sigma)
  rightDivide(a, b) == rightDivide(a, b, sigma)

```

— ORESUP.dotabb —

```

"ORESUP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ORESUP"]
"OREPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OREPCAT"]
"ORESUP" -> "OREPCAT"

```

domain SUTS SparseUnivariateTaylorSeries

— SparseUnivariateTaylorSeries.input —

```

)set break resume
)sys rm -f SparseUnivariateTaylorSeries.output
)spool SparseUnivariateTaylorSeries.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SparseUnivariateTaylorSeries
--R
--R SparseUnivariateTaylorSeries(Coef: Ring,var: Symbol,cen: Coef) is a domain constructor
--R Abbreviation for SparseUnivariateTaylorSeries is SUTS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SUTS
--R
--R----- Operations -----
--R ?? : (Coef,%) -> %          ?? : (%,Coef) -> %
--R ?? : (%,%) -> %           ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %

```



```

--R ?+? : (% , %) -> %
--R -? : % -> %
--R 1 : () -> %
--R ?^? : (% , PositiveInteger) -> %
--R coefficients : % -> Stream(Coef)
--R coerce : Integer -> %
--R complete : % -> %
--R hash : % -> SingleInteger
--R leadingCoefficient : % -> Coef
--R map : ((Coef -> Coef), %) -> %
--R one? : % -> Boolean
--R pole? : % -> Boolean
--R recip : % -> Union(%, "failed")
--R sample : () -> %
--R variable : % -> Symbol
--R ?~=? : (% , %) -> Boolean
--R ?*? : (% , Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?*? : (Fraction(Integer), %) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?*? : (NonNegativeInteger, %) -> %
--R ??? : (% , Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (% , %) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (% , Coef) -> % if Coef has FIELD
--R ??? : (% , NonNegativeInteger) -> %
--R ?/? : (% , Coef) -> % if Coef has FIELD
--R D : % -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef
--R D : (% , NonNegativeInteger) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef
--R D : (% , Symbol) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and Coef has PDRING(SYM
--R D : (% , List(Symbol)) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and Coef has PDRI
--R D : (% , Symbol, NonNegativeInteger) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and
--R D : (% , List(Symbol), List(NonNegativeInteger)) -> % if Coef has *: (NonNegativeInteger, Coef)
--R ?^? : (% , NonNegativeInteger) -> %
--R acos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acoth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R approximate : (% , NonNegativeInteger) -> Coef if Coef has **: (Coef, NonNegativeInteger) -> Co
--R asec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R associates? : (% , %) -> Boolean if Coef has INTDOM
--R atan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R atanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if Coef has CHARNZ
--R coefficient : (% , NonNegativeInteger) -> Coef
--R coerce : UnivariatePolynomial(var, Coef) -> %
--R coerce : Coef -> % if Coef has COMRING
--R ?-? : (% , %) -> %
--R ?=? : (% , %) -> Boolean
--R 0 : () -> %
--R center : % -> Coef
--R coerce : Variable(var) -> %
--R coerce : % -> OutputForm
--R degree : % -> NonNegativeInteger
--R latex : % -> String
--R leadingMonomial : % -> %
--R monomial? : % -> Boolean
--R order : % -> NonNegativeInteger
--R quoByVar : % -> %
--R reductum : % -> %
--R series : Stream(Coef) -> %
--R zero? : % -> Boolean

```

```

--R coerce : % -> % if Coef has INTDOM
--R coerce : Fraction(Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R cos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R coth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R differentiate : (%,Variable(var)) -> %
--R differentiate : % -> % if Coef has *: (NonNegativeInteger,Coef) -> Coef
--R differentiate : (%,NonNegativeInteger) -> % if Coef has *: (NonNegativeInteger,Coef) -> Coef
--R differentiate : (%,Symbol) -> % if Coef has *: (NonNegativeInteger,Coef) -> Coef and Coef has PDRING(SY
--R differentiate : (%,List(Symbol)) -> % if Coef has *: (NonNegativeInteger,Coef) -> Coef and Coef has PDR
--R differentiate : (%,Symbol,NonNegativeInteger) -> % if Coef has *: (NonNegativeInteger,Coef) -> Coef and
--R differentiate : (%,List(Symbol),List(NonNegativeInteger)) -> % if Coef has *: (NonNegativeInteger,Coef)
--R ?? : (%,%) -> % if NonNegativeInteger has SGROUP
--R ?? : (%,NonNegativeInteger) -> Coef
--R eval : (%,Coef) -> Stream(Coef) if Coef has **: (Coef,NonNegativeInteger) -> Coef
--R exp : % -> % if Coef has ALGEBRA(FRAC(INT))
--R exquo : (%,%) -> Union(%, "failed") if Coef has INTDOM
--R extend : (%,NonNegativeInteger) -> %
--R integrate : (%,Variable(var)) -> % if Coef has ALGEBRA(FRAC(INT))
--R integrate : (%,Symbol) -> % if Coef has integrate: (Coef,Symbol) -> Coef and Coef has variables: Coef -
--R integrate : % -> % if Coef has ALGEBRA(FRAC(INT))
--R log : % -> % if Coef has ALGEBRA(FRAC(INT))
--R monomial : (%,List(SingletonAsOrderedSet),List(NonNegativeInteger)) -> %
--R monomial : (%,SingletonAsOrderedSet,NonNegativeInteger) -> %
--R monomial : (Coef,NonNegativeInteger) -> %
--R multiplyCoefficients : ((Integer -> Coef),%) -> %
--R multiplyExponents : (%,PositiveInteger) -> %
--R nthRoot : (%,Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R order : (%,NonNegativeInteger) -> NonNegativeInteger
--R pi : () -> % if Coef has ALGEBRA(FRAC(INT))
--R polynomial : (%,NonNegativeInteger,NonNegativeInteger) -> Polynomial(Coef)
--R polynomial : (%,NonNegativeInteger) -> Polynomial(Coef)
--R sec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R series : Stream(Record(k: NonNegativeInteger,c: Coef)) -> %
--R sin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sqrt : % -> % if Coef has ALGEBRA(FRAC(INT))
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R tan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R tanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R terms : % -> Stream(Record(k: NonNegativeInteger,c: Coef))
--R truncate : (%,NonNegativeInteger,NonNegativeInteger) -> %
--R truncate : (%,NonNegativeInteger) -> %
--R unit? : % -> Boolean if Coef has INTDOM
--R unitCanonical : % -> % if Coef has INTDOM
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %) if Coef has INTDOM

```

```

--R univariatePolynomial : (% , NonNegativeInteger) -> UnivariatePolynomial(var, Coef)
--R variables : % -> List(SingletonAsOrderedSet)
--R
--E 1

)spool
)lisp (bye)

```

— SparseUnivariateTaylorSeries.help —

```

=====
SparseUnivariateTaylorSeries examples
=====

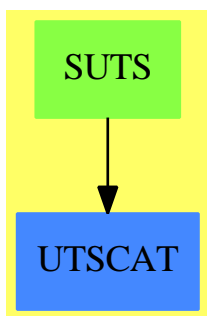
```

SparseUnivariateTaylorSeries is a domain representing Taylor series in one variable with coefficients in an arbitrary ring. The parameters of the type specify the coefficient ring, the power series variable, and the center of the power series expansion. For example, `SparseUnivariateTaylorSeries(Integer,x,3)` represents Taylor series in $(x - 3)$ with Integer coefficients.

See Also:

o `)show SparseUnivariateTaylorSeries`

SparseUnivariateTaylorSeries (SUTS)



Exports:

0	1	acos	acosh	acot
acoth	acsc	acsch	approximate	asec
asech	asin	asinh	associates?	atan
atanh	center	characteristic	charthRoot	coefficient
coefficients	coerce	complete	cos	cosh
cot	coth	csc	csch	D
degree	differentiate	eval	exp	exquo
extend	hash	integrate	latex	leadingCoefficient
leadingMonomial	log	map	monomial	monomial?
multiplyCoefficients	multiplyExponents	nthRoot	one?	order
pole?	pi	polynomial	polynomial	quoByVar
recip	reductum	sample	sec	sech
series	sin	sinh	sqrt	subtractIfCan
tan	tanh	terms	truncate	truncate
unit?	unitCanonical	unitNormal	univariatePolynomial	variable
variables	zero?	?*?	?**?	?+?
?-?	-?	?=?	?^?	?~=?
?/?	?..?			

— domain SUTS SparseUnivariateTaylorSeries —

```

)abbrev domain SUTS SparseUnivariateTaylorSeries
++ Author: Clifton J. Williamson
++ Date Created: 16 February 1990
++ Date Last Updated: 10 March 1995
++ Description:
++ Sparse Taylor series in one variable
++ \spadtype{SparseUnivariateTaylorSeries} is a domain representing Taylor
++ series in one variable with coefficients in an arbitrary ring. The
++ parameters of the type specify the coefficient ring, the power series
++ variable, and the center of the power series expansion. For example,
++ \spadtype{SparseUnivariateTaylorSeries}(Integer,x,3) represents Taylor
++ series in \spad{(x - 3)} with \spadtype{Integer} coefficients.

SparseUnivariateTaylorSeries(Coef,var,cen): Exports == Implementation where
  Coef : Ring
  var : Symbol
  cen : Coef
  COM ==> OrderedCompletion Integer
  I ==> Integer
  L ==> List
  NNI ==> NonNegativeInteger
  OUT ==> OutputForm
  P ==> Polynomial Coef
  REF ==> Reference OrderedCompletion Integer
  RN ==> Fraction Integer
  Term ==> Record(k:Integer,c:Coef)

```

```

SG  ==> String
ST  ==> Stream Term
UP  ==> UnivariatePolynomial(var,Coef)

Exports ==> UnivariateTaylorSeriesCategory(Coef) with
  coerce: UP -> %
    ++\spad{coerce(p)} converts a univariate polynomial p in the variable
    ++\spad{var} to a univariate Taylor series in \spad{var}.
  univariatePolynomial: (%,NNI) -> UP
    ++\spad{univariatePolynomial(f,k)} returns a univariate polynomial
    ++ consisting of the sum of all terms of f of degree \spad{<= k}.
  coerce: Variable(var) -> %
    ++\spad{coerce(var)} converts the series variable \spad{var} into a
    ++ Taylor series.
  differentiate: (%,Variable(var)) -> %
    ++ \spad{differentiate(f(x),x)} computes the derivative of
    ++ \spad{f(x)} with respect to \spad{x}.
  if Coef has Algebra Fraction Integer then
    integrate: (%,Variable(var)) -> %
      ++ \spad{integrate(f(x),x)} returns an anti-derivative of the power
      ++ series \spad{f(x)} with constant coefficient 0.
      ++ We may integrate a series when we can divide coefficients
      ++ by integers.

Implementation ==> InnerSparseUnivariatePowerSeries(Coef) add
  import REF

  Rep := InnerSparseUnivariatePowerSeries(Coef)

  makeTerm: (Integer,Coef) -> Term
  makeTerm(exp,coef) == [exp,coef]
  getCoef: Term -> Coef
  getCoef term == term.c
  getExpon: Term -> Integer
  getExpon term == term.k

  monomial(coef,expon) == monomial(coef,expon)$Rep
  extend(x,n) == extend(x,n)$Rep

  0 == monomial(0,0)$Rep
  1 == monomial(1,0)$Rep

  recip uts == iExquo(1,uts,true)

  if Coef has IntegralDomain then
    uts1 exquo uts2 == iExquo(uts1,uts2,true)

  quoByVar uts == taylorQuoByVar(uts)$Rep

  differentiate(x:%,v:Variable(var)) == differentiate x

```

```
--% Creation and destruction of series
```

```
coerce(v: Variable(var)) ==
  zero? cen => monomial(1,1)
  monomial(1,1) + monomial(cen,0)

coerce(p:UP) ==
  zero? p => 0
  if not zero? cen then p := p(monomial(1,1)$UP + monomial(cen,0)$UP)
  st : ST := empty()
  while not zero? p repeat
    st := concat(makeTerm(degree p,leadingCoefficient p),st)
    p := reductum p
  makeSeries(ref plusInfinity(),st)

univariatePolynomial(x,n) ==
  extend(x,n); st := getStream x
  ans : UP := 0; oldDeg : I := 0;
  mon := monomial(1,1)$UP - monomial(center x,0)$UP; monPow : UP := 1
  while explicitEntries? st repeat
    (xExpon := getExpon(xTerm := frst st)) > n => return ans
    pow := (xExpon - oldDeg) :: NNI; oldDeg := xExpon
    monPow := monPow * mon ** pow
    ans := ans + getCoef(xTerm) * monPow
    st := rst st
  ans

polynomial(x,n) ==
  extend(x,n); st := getStream x
  ans : P := 0; oldDeg : I := 0;
  mon := (var :: P) - (center(x) :: P); monPow : P := 1
  while explicitEntries? st repeat
    (xExpon := getExpon(xTerm := frst st)) > n => return ans
    pow := (xExpon - oldDeg) :: NNI; oldDeg := xExpon
    monPow := monPow * mon ** pow
    ans := ans + getCoef(xTerm) * monPow
    st := rst st
  ans

polynomial(x,n1,n2) == polynomial(truncate(x,n1,n2),n2)

truncate(x,n) == truncate(x,n)$Rep
truncate(x,n1,n2) == truncate(x,n1,n2)$Rep

iCoefficients: (ST,REF,I) -> Stream Coef
iCoefficients(x,refer,n) == delay
  -- when this function is called, we are computing the nth order
  -- coefficient of the series
  explicitlyEmpty? x => empty()
```

```

-- if terms up to order n have not been computed,
-- apply lazy evaluation
nn := n :: COM
while (nx := elt refer) < nn repeat lazyEvaluate x
-- must have nx >= n
explicitEntries? x =>
  xCoef := getCoef(xTerm := frst x); xExpon := getExpon xTerm
  xExpon = n => concat(xCoef,iCoefficients(rst x,refer,n + 1))
  -- must have nx > n
  concat(0,iCoefficients(x,refer,n + 1))
concat(0,iCoefficients(x,refer,n + 1))

coefficients uts ==
  refer := getRef uts; x := getStream uts
  iCoefficients(x,refer,0)

terms uts == terms(uts)$Rep pretend Stream Record(k:NNI,c:Coef)

iSeries: (Stream Coef,I,REF) -> ST
iSeries(st,n,refer) == delay
  -- when this function is called, we are creating the nth order
  -- term of a series
  empty? st => (setelt(refer,plusInfinity()); empty())
  setelt(refer,n :: COM)
  zero? (coef := frst st) => iSeries(rst st,n + 1,refer)
  concat(makeTerm(n,coef),iSeries(rst st,n + 1,refer))

series(st:Stream Coef) ==
  refer := ref(-1)
  makeSeries(refer,iSeries(st,0,refer))

nniToI: Stream Record(k:NNI,c:Coef) -> ST
nniToI st ==
  empty? st => empty()
  term : Term := [(frst st).k,(frst st).c]
  concat(term,nniToI rst st)

series(st:Stream Record(k:NNI,c:Coef)) == series(nniToI st)$Rep

--% Values

variable x == var
center x == cen

coefficient(x,n) == coefficient(x,n)$Rep
elt(x:%,n:NonNegativeInteger) == coefficient(x,n)

pole? x == false

order x == (order(x)$Rep) :: NNI

```

```

order(x,n) == (order(x,n)$Rep) :: NNI

--% Composition

elt(uts1:%,uts2:%) ==
  zero? uts2 => coefficient(uts1,0) :: %
  not zero? coefficient(uts2,0) =>
    error "elt: second argument must have positive order"
  iCompose(uts1,uts2)

--% Integration

if Coef has Algebra Fraction Integer then

  integrate(x:%,v:Variable(var)) == integrate x

--% Transcendental functions

(uts1:%) ** (uts2:%) == exp(log(uts1) * uts2)

if Coef has CommutativeRing then

  (uts:%) ** (r:RN) == cRationalPower(uts,r)

  exp uts == cExp uts
  log uts == cLog uts

  sin uts == cSin uts
  cos uts == cCos uts
  tan uts == cTan uts
  cot uts == cCot uts
  sec uts == cSec uts
  csc uts == cCsc uts

  asin uts == cAsin uts
  acos uts == cAcos uts
  atan uts == cAtan uts
  acot uts == cAcot uts
  asec uts == cAsec uts
  acsc uts == cAcsc uts

  sinh uts == cSinh uts
  cosh uts == cCosh uts
  tanh uts == cTanh uts
  coth uts == cCoth uts
  sech uts == cSech uts
  csch uts == cCsch uts

  asinh uts == cAsinh uts
  acosh uts == cAcosh uts

```



```

    atanh uts == cAtanh uts
    acoth uts == cAcoth uts
    asech uts == cAsech uts
    acsch uts == cAcsch uts

else

    ZERO    : SG := "series must have constant coefficient zero"
    ONE     : SG := "series must have constant coefficient one"
    NPOWERS : SG := "series expansion has terms of negative degree"

    (uts:%) ** (r:RN) ==
--      not one? coefficient(uts,0) =>
        not (coefficient(uts,0) = 1) =>
            error "**: constant coefficient must be one"
        onePlusX : % := monomial(1,0) + monomial(1,1)
        ratPow := cPower(uts,r :: Coef)
        iCompose(ratPow,uts - 1)

    exp uts ==
        zero? coefficient(uts,0) =>
            expx := cExp monomial(1,1)
            iCompose(expx,uts)
            error concat("exp: ",ZERO)

    log uts ==
--      one? coefficient(uts,0) =>
        (coefficient(uts,0) = 1) =>
            log1PlusX := cLog(monomial(1,0) + monomial(1,1))
            iCompose(log1PlusX,uts - 1)
            error concat("log: ",ONE)

    sin uts ==
        zero? coefficient(uts,0) =>
            sinx := cSin monomial(1,1)
            iCompose(sinx,uts)
            error concat("sin: ",ZERO)

    cos uts ==
        zero? coefficient(uts,0) =>
            cosx := cCos monomial(1,1)
            iCompose(cosx,uts)
            error concat("cos: ",ZERO)

    tan uts ==
        zero? coefficient(uts,0) =>
            tanx := cTan monomial(1,1)
            iCompose(tanx,uts)
            error concat("tan: ",ZERO)

```

```

cot uts ==
  zero? uts => error "cot: cot(0) is undefined"
  zero? coefficient(uts,0) => error concat("cot: ",NPOWERS)
  error concat("cot: ",ZERO)

sec uts ==
  zero? coefficient(uts,0) =>
    secx := cSec monomial(1,1)
    iCompose(secx,uts)
  error concat("sec: ",ZERO)

csc uts ==
  zero? uts => error "csc: csc(0) is undefined"
  zero? coefficient(uts,0) => error concat("csc: ",NPOWERS)
  error concat("csc: ",ZERO)

asin uts ==
  zero? coefficient(uts,0) =>
    asinx := cAsin monomial(1,1)
    iCompose(asinx,uts)
  error concat("asin: ",ZERO)

atan uts ==
  zero? coefficient(uts,0) =>
    atanx := cAtan monomial(1,1)
    iCompose(atanx,uts)
  error concat("atan: ",ZERO)

acos z == error "acos: acos undefined on this coefficient domain"
acot z == error "acot: acot undefined on this coefficient domain"
asec z == error "asec: asec undefined on this coefficient domain"
acsc z == error "acsc: acsc undefined on this coefficient domain"

sinh uts ==
  zero? coefficient(uts,0) =>
    sinhx := cSinh monomial(1,1)
    iCompose(sinhx,uts)
  error concat("sinh: ",ZERO)

cosh uts ==
  zero? coefficient(uts,0) =>
    coshx := cCosh monomial(1,1)
    iCompose(coshx,uts)
  error concat("cosh: ",ZERO)

tanh uts ==
  zero? coefficient(uts,0) =>
    tanhx := cTanh monomial(1,1)
    iCompose(tanhx,uts)
  error concat("tanh: ",ZERO)

```

```

coth uts ==
  zero? uts => error "coth: coth(0) is undefined"
  zero? coefficient(uts,0) => error concat("coth: ",NPOWERS)
  error concat("coth: ",ZERO)

sech uts ==
  zero? coefficient(uts,0) =>
    sechx := cSech monomial(1,1)
    iCompose(sechx,uts)
  error concat("sech: ",ZERO)

csch uts ==
  zero? uts => error "csch: csch(0) is undefined"
  zero? coefficient(uts,0) => error concat("csch: ",NPOWERS)
  error concat("csch: ",ZERO)

asinh uts ==
  zero? coefficient(uts,0) =>
    asinhx := cAsinh monomial(1,1)
    iCompose(asinhx,uts)
  error concat("asinh: ",ZERO)

atanh uts ==
  zero? coefficient(uts,0) =>
    atanhx := cAtanh monomial(1,1)
    iCompose(atanhx,uts)
  error concat("atanh: ",ZERO)

acosh uts == error "acosh: acosh undefined on this coefficient domain"
acoth uts == error "acoth: acoth undefined on this coefficient domain"
asech uts == error "asech: asech undefined on this coefficient domain"
acsch uts == error "acsch: acsch undefined on this coefficient domain"

if Coef has Field then
  if Coef has Algebra Fraction Integer then

    (uts:%) ** (r:Coef) ==
--      not one? coefficient(uts,1) =>
        not (coefficient(uts,1) = 1) =>
          error "***: constant coefficient should be 1"
        cPower(uts,r)

--% OutputForms

coerce(x:%): OUT ==
  count : NNI := _$streamCount$Lisp
  extend(x,count)
  seriesToOutputForm(getStream x,getRef x,variable x,center x,1)

```

— SUTS.dotabb —

```
"SUTS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SUTS"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"SUTS" -> "UTSCAT"
```

domain SHDP SplitHomogeneousDirectProduct

— SplitHomogeneousDirectProduct.input —

```
)set break resume
)sys rm -f SplitHomogeneousDirectProduct.output
)spool SplitHomogeneousDirectProduct.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SplitHomogeneousDirectProduct
--R SplitHomogeneousDirectProduct(dimtot: NonNegativeInteger,dim1: NonNegativeInteger,S: OrderedAbelianMonoid)
--R Abbreviation for SplitHomogeneousDirectProduct is SHDP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SHDP
--R
--R----- Operations -----
--R ? : % -> % if S has RING          1 : () -> % if S has MONOID
--R 0 : () -> % if S has CABMON       coerce : % -> Vector(S)
--R copy : % -> %                     directProduct : Vector(S) -> %
--R ?.? : (%,Integer) -> S           elt : (%,Integer,S) -> S
--R empty : () -> %                  empty? : % -> Boolean
--R entries : % -> List(S)           eq? : (%,% ) -> Boolean
--R index? : (Integer,% ) -> Boolean  indices : % -> List(Integer)
--R map : ((S -> S),%) -> %          qelt : (%,Integer) -> S
--R sample : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?? : (PositiveInteger,% ) -> % if S has ABELSG
--R ?? : (NonNegativeInteger,% ) -> % if S has CABMON
--R ?? : (S,% ) -> % if S has MONOID
--R ?? : (% ,S) -> % if S has MONOID
--R ?? : (% ,%) -> % if S has MONOID
--R ?? : (Integer,% ) -> % if S has RING
```

```

--R ?**? : (%,PositiveInteger) -> % if S has MONOID
--R ?**? : (%,NonNegativeInteger) -> % if S has MONOID
--R ?+? : (%,%) -> % if S has ABELSG
--R ?-? : (%,%) -> % if S has RING
--R ?/? : (%,S) -> % if S has FIELD
--R ?<? : (%,%) -> Boolean if S has OAMONS or S has ORDRING
--R ?<=? : (%,%) -> Boolean if S has OAMONS or S has ORDRING
--R ?=? : (%,%) -> Boolean if S has SETCAT
--R ?>? : (%,%) -> Boolean if S has OAMONS or S has ORDRING
--R ?>=? : (%,%) -> Boolean if S has OAMONS or S has ORDRING
--R D : (%,(S -> S)) -> % if S has RING
--R D : (%,(S -> S),NonNegativeInteger) -> % if S has RING
--R D : (%,List(Symbol),List(NonNegativeInteger)) -> % if S has PDRING(SYMBOL) and S has RING
--R D : (%,Symbol,NonNegativeInteger) -> % if S has PDRING(SYMBOL) and S has RING
--R D : (%,List(Symbol)) -> % if S has PDRING(SYMBOL) and S has RING
--R D : (%,Symbol) -> % if S has PDRING(SYMBOL) and S has RING
--R D : (%,NonNegativeInteger) -> % if S has DIFRING and S has RING
--R D : % -> % if S has DIFRING and S has RING
--R ?? : (%,PositiveInteger) -> % if S has MONOID
--R ?? : (%,NonNegativeInteger) -> % if S has MONOID
--R abs : % -> % if S has ORDRING
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R characteristic : () -> NonNegativeInteger if S has RING
--R coerce : S -> % if S has SETCAT
--R coerce : Fraction(Integer) -> % if S has RETRACT(FRAC(INT)) and S has SETCAT
--R coerce : Integer -> % if S has RETRACT(INT) and S has SETCAT or S has RING
--R coerce : % -> OutputForm if S has SETCAT
--R count : (S,%) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R differentiate : (%,(S -> S)) -> % if S has RING
--R differentiate : (%,(S -> S),NonNegativeInteger) -> % if S has RING
--R differentiate : (%,List(Symbol),List(NonNegativeInteger)) -> % if S has PDRING(SYMBOL) and S
--R differentiate : (%,Symbol,NonNegativeInteger) -> % if S has PDRING(SYMBOL) and S has RING
--R differentiate : (%,List(Symbol)) -> % if S has PDRING(SYMBOL) and S has RING
--R differentiate : (%,Symbol) -> % if S has PDRING(SYMBOL) and S has RING
--R differentiate : (%,NonNegativeInteger) -> % if S has DIFRING and S has RING
--R differentiate : % -> % if S has DIFRING and S has RING
--R dimension : () -> CardinalNumber if S has FIELD
--R dot : (%,%) -> S if S has RING
--R entry? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R enumerate : () -> List(%) if S has FINITE
--R eval : (%,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (%,S) -> % if $ has shallowlyMutable
--R first : % -> S if Integer has ORDSET
--R hash : % -> SingleInteger if S has SETCAT
--R index : PositiveInteger -> % if S has FINITE

```

```

--R latex : % -> String if S has SETCAT
--R less? : (% , NonNegativeInteger) -> Boolean
--R lookup : % -> PositiveInteger if S has FINITE
--R map! : ((S -> S), %) -> % if $ has shallowlyMutable
--R max : (% , %) -> % if S has OAMONS or S has ORDRING
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (S, %) -> Boolean if $ has finiteAggregate and S has SETCAT
--R members : % -> List(S) if $ has finiteAggregate
--R min : (% , %) -> % if S has OAMONS or S has ORDRING
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (% , NonNegativeInteger) -> Boolean
--R negative? : % -> Boolean if S has ORDRING
--R one? : % -> Boolean if S has MONOID
--R parts : % -> List(S) if $ has finiteAggregate
--R positive? : % -> Boolean if S has ORDRING
--R qsetelt! : (% , Integer, S) -> S if $ has shallowlyMutable
--R random : () -> % if S has FINITE
--R recip : % -> Union(% , "failed") if S has MONOID
--R reducedSystem : Matrix(%) -> Matrix(S) if S has RING
--R reducedSystem : (Matrix(%), Vector(%)) -> Record(mat: Matrix(S), vec: Vector(S)) if S has RING
--R reducedSystem : (Matrix(%), Vector(%)) -> Record(mat: Matrix(Integer), vec: Vector(Integer)) if S has LINEAR
--R reducedSystem : Matrix(%) -> Matrix(Integer) if S has LINEAR and S has RING
--R retract : % -> S if S has SETCAT
--R retract : % -> Fraction(Integer) if S has RETRACT(FRAC(INT)) and S has SETCAT
--R retract : % -> Integer if S has RETRACT(INT) and S has SETCAT
--R retractIfCan : % -> Union(S, "failed") if S has SETCAT
--R retractIfCan : % -> Union(Fraction(Integer), "failed") if S has RETRACT(FRAC(INT)) and S has SETCAT
--R retractIfCan : % -> Union(Integer, "failed") if S has RETRACT(INT) and S has SETCAT
--R setelt : (% , Integer, S) -> S if $ has shallowlyMutable
--R sign : % -> Integer if S has ORDRING
--R size : () -> NonNegativeInteger if S has FINITE
--R size? : (% , NonNegativeInteger) -> Boolean
--R subtractIfCan : (% , %) -> Union(% , "failed") if S has CABMON
--R sup : (% , %) -> % if S has OAMONS
--R swap! : (% , Integer, Integer) -> Void if $ has shallowlyMutable
--R unitVector : PositiveInteger -> % if S has RING
--R zero? : % -> Boolean if S has CABMON
--R ?~=? : (% , %) -> Boolean if S has SETCAT
--R
--E 1

)spool
)lisp (bye)

```

— SplitHomogeneousDirectProduct.help —

SplitHomogeneousDirectProduct examples

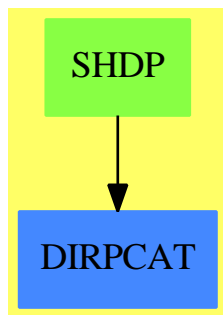
=====

This type represents the finite direct or cartesian product of an underlying ordered component type. The vectors are ordered as if they were split into two blocks. The `dim1` parameter specifies the length of the first block. The ordering is lexicographic between the blocks but acts like `HomogeneousDirectProduct` within each block. This type is a suitable third argument for `GeneralDistributedMultivariatePolynomial`.

See Also:

o `)show SplitHomogeneousDirectProduct`

SplitHomogeneousDirectProduct (SHDP)



See

⇒ “OrderedDirectProduct” (ODP) [16](#) on page [1816](#)

⇒ “HomogeneousDirectProduct” (HDP) [9](#) on page [1172](#)

Exports:

0	1	abs	any?	characteristic
coerce	copy	count	D	differentiate
dimension	directProduct	dot	elt	empty
empty?	entries	entry?	eq?	eval
every?	fill!	first	hash	index
index?	indices	latex	less?	lookup
map	map!	max	maxIndex	member?
members	min	minIndex	more?	negative?
one?	parts	positive?	qelt	qsetelt!
random	recip	reducedSystem	retract	retractIfCan
sample	setelt	sign	size	size?
subtractIfCan	sup	swap!	unitVector	zero?
#?	?*?	?**?	?+?	?-?
?/?	?<?	?<=?	?=?	?>?
?>=?	?^?	?~=?	~?	?.?

— domain SHDP SplitHomogeneousDirectProduct —

```
)abbrev domain SHDP SplitHomogeneousDirectProduct
++ Author: Mark Botch
++ Description:
++ This type represents the finite direct or cartesian product of an
++ underlying ordered component type. The vectors are ordered as if
++ they were split into two blocks. The dim1 parameter specifies the
++ length of the first block. The ordering is lexicographic between
++ the blocks but acts like \spadtype{HomogeneousDirectProduct}
++ within each block. This type is a suitable third argument for
++ \spadtype{GeneralDistributedMultivariatePolynomial}.
```

```
SplitHomogeneousDirectProduct(dimtot,dim1,S) : T == C where
  NNI ==> NonNegativeInteger
  dim1,dimtot : NNI
  S          : OrderedAbelianMonoidSup

  T == DirectProductCategory(dimtot,S)
  C == DirectProduct(dimtot,S) add
    Rep:=Vector(S)
    lessThanRlex(v1:%,v2:%,low:NNI,high:NNI):Boolean ==
-- reverse lexicographical ordering
  n1:S:=0
  n2:S:=0
  for i in low..high repeat
    n1:= n1+qelt(v1,i)
    n2:=n2+qelt(v2,i)
  n1<n2 => true
  n2<n1 => false
  for i in reverse(low..high) repeat
    if qelt(v2,i) < qelt(v1,i) then return true
```



```

        if qelt(v1,i) < qelt(v2,i) then return false
    false

(v1:% < v2:%):Boolean ==
    lessThanRlex(v1,v2,1,dim1) => true
    for i in 1..dim1 repeat
        if qelt(v1,i) ^= qelt(v2,i) then return false
    lessThanRlex(v1,v2,dim1+1,dimtot)

```

— SHDP.dotabb —

```

"SHDP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SHDP"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"SHDP" -> "DIRPCAT"

```

domain SPLNODE SplittingNode

— SplittingNode.input —

```

)set break resume
)sys rm -f SplittingNode.output
)spool SplittingNode.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SplittingNode
--R
--R SplittingNode(V: Join(SetCategory,Aggregate),C: Join(SetCategory,Aggregate)) is a domain co
--R Abbreviation for SplittingNode is SPLNODE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SPLNODE
--R
--R----- Operations -----
--R ?=? : (%,%) -> Boolean          coerce : % -> OutputForm
--R condition : % -> C              construct : (V,C) -> %
--R construct : (V,C,Boolean) -> %  copy : % -> %
--R empty : () -> %                 empty? : % -> Boolean
--R hash : % -> SingleInteger       latex : % -> String

```

```

--R setCondition! : (% , C) -> %           setEmpty! : % -> %
--R setStatus! : (% , Boolean) -> %       setValue! : (% , V) -> %
--R status : % -> Boolean                 value : % -> V
--R ?~=? : (% , %) -> Boolean
--R construct : (V , List(C)) -> List(%)
--R construct : List(Record(val: V , tower: C)) -> List(%)
--R construct : Record(val: V , tower: C) -> %
--R infLex? : (% , % , ((V , V) -> Boolean) , ((C , C) -> Boolean)) -> Boolean
--R subNode? : (% , % , ((C , C) -> Boolean)) -> Boolean
--R
--E 1

```

```

)spool
)lisp (bye)

```

— SplittingNode.help —

=====

SplittingNode examples

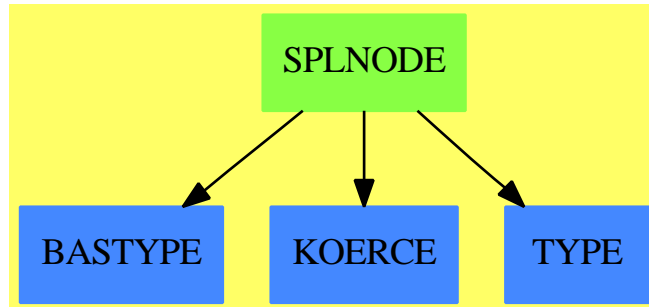
=====

This domain exports a modest implementation for the vertices of splitting trees. These vertices are called here splitting nodes. Every of these nodes store 3 informations. The first one is its value, that is the current expression to evaluate. The second one is its condition, that is the hypothesis under which the value has to be evaluated. The last one is its status, that is a boolean flag which is true iff the value is the result of its evaluation under its condition. Two splitting vertices are equal iff they have the same values and the same conditions (so their status do not matter).

See Also:

- o)show SplittingNode

SplittingNode (SPLNODE)



See

⇒ “SplittingNode” (SPLNODE) 20 on page 2512

Exports:

coerce	condition	construct	copy	empty
empty?	hash	infLex?	latex	setCondition!
setEmpty!	setStatus!	setValue!	status	subNode?
value	?=?	?~=?		

— domain SPLNODE SplittingNode —

```

)abbrev domain SPLNODE SplittingNode
++ Author: Marc Moereno Maza
++ Date Created: 07/05/1996
++ Date Last Updated: 07/19/1996
++ Description:
++ This domain exports a modest implementation for the
++ vertices of splitting trees. These vertices are called
++ here splitting nodes. Every of these nodes store 3 informations.
++ The first one is its value, that is the current expression
++ to evaluate. The second one is its condition, that is the
++ hypothesis under which the value has to be evaluated.
++ The last one is its status, that is a boolean flag
++ which is true iff the value is the result of its
++ evaluation under its condition. Two splitting vertices
++ are equal iff they have the same values and the same
++ conditions (so their status do not matter).
  
```

SplittingNode(V,C) : Exports == Implementation where

```

V:Join(SetCategory,Aggregate)
C:Join(SetCategory,Aggregate)
Z ==> Integer
B ==> Boolean
O ==> OutputForm
VT ==> Record(val:V, tower:C)
  
```

```
VTB ==> Record(val:V, tower:C, flag:B)
```

```
Exports == SetCategory with
```

```

empty : () -> %
  ++ \axiom{empty()} returns the same as
  ++ \axiom{[empty()$V,empty()$C,false]$%}
empty? : % -> B
  ++ \axiom{empty?(n)} returns true iff the node n is \axiom{empty()$%}.
value : % -> V
  ++ \axiom{value(n)} returns the value of the node n.
condition : % -> C
  ++ \axiom{condition(n)} returns the condition of the node n.
status : % -> B
  ++ \axiom{status(n)} returns the status of the node n.
construct : (V,C,B) -> %
  ++ \axiom{construct(v,t,b)} returns the non-empty node with
  ++ value v, condition t and flag b
construct : (V,C) -> %
  ++ \axiom{construct(v,t)} returns the same as
  ++ \axiom{construct(v,t,false)}
construct : VT -> %
  ++ \axiom{construct(vt)} returns the same as
  ++ \axiom{construct(vt.val,vt.tower)}
construct : List VT -> List %
  ++ \axiom{construct(lvt)} returns the same as
  ++ \axiom{[construct(vt.val,vt.tower) for vt in lvt]}
construct : (V, List C) -> List %
  ++ \axiom{construct(v,lt)} returns the same as
  ++ \axiom{[construct(v,t) for t in lt]}
copy : % -> %
  ++ \axiom{copy(n)} returns a copy of n.
setValue! : (% ,V) -> %
  ++ \axiom{setValue!(n,v)} returns n whose value
  ++ has been replaced by v if it is not
  ++ empty, else an error is produced.
setCondition! : (% ,C) -> %
  ++ \axiom{setCondition!(n,t)} returns n whose condition
  ++ has been replaced by t if it is not
  ++ empty, else an error is produced.
setStatus! : (% ,B) -> %
  ++ \axiom{setStatus!(n,b)} returns n whose status
  ++ has been replaced by b if it is not
  ++ empty, else an error is produced.
setEmpty! : % -> %
  ++ \axiom{setEmpty!(n)} replaces n by \axiom{empty()$%}.
inflex? : (% ,%, (V,V) -> B, (C,C) -> B) -> B
  ++ \axiom{inflex?(n1,n2,o1,o2)} returns true iff
  ++ \axiom{o1(value(n1),value(n2))} or
  ++ \axiom{value(n1) = value(n2)} and

```

```

++ \axiom{o2(condition(n1),condition(n2))}.
subNode? : (%,(C,C) -> B) -> B
++ \axiom{subNode?(n1,n2,o2)} returns true iff
++ \axiom{value(n1) = value(n2)} and
++ \axiom{o2(condition(n1),condition(n2))}

```

```
Implementation == add
```

```
Rep ==> VTB
```

```

rep(n:%):Rep == n pretend Rep
per(r:Rep):% == r pretend %

```

```

empty() == per [empty()$V,empty()$C,false]$Rep
empty?(n:%) == empty?((rep n).val)$V and empty?((rep n).tower)$C
value(n:%) == (rep n).val
condition(n:%) == (rep n).tower
status(n:%) == (rep n).flag
construct(v:V,t:C,b:B) == per [v,t,b]$Rep
construct(v:V,t:C) == [v,t,false]$%
construct(vt:VT) == [vt.val,vt.tower]$%
construct(lvt>List VT) == [[vt]$% for vt in lvt]
construct(v:V,lt: List C) == [[v,t]$% for t in lt]
copy(n:%) == per copy rep n
setValue!(n:%,v:V) ==
  (rep n).val := v
  n
setCondition!(n:%,t:C) ==
  (rep n).tower := t
  n
setStatus!(n:%,b:B) ==
  (rep n).flag := b
  n
setEmpty!(n:%) ==
  (rep n).val := empty()$V
  (rep n).tower := empty()$C
  n
infLex?(n1,n2,o1,o2) ==
  o1((rep n1).val,(rep n2).val) => true
  (rep n1).val = (rep n2).val =>
    o2((rep n1).tower,(rep n2).tower)
  false
subNode?(n1,n2,o2) ==
  (rep n1).val = (rep n2).val =>
    o2((rep n1).tower,(rep n2).tower)
  false
-- sample() == empty()
n1:% = n2:% ==
  (rep n1).val ~= (rep n2).val => false
  (rep n1).tower = (rep n2).tower

```

```

n1:% ~= n2:% ==
  (rep n1).val = (rep n2).val => false
  (rep n1).tower ~= (rep n2).tower
coerce(n:%):0 ==
  l1,l2,l3,l : List 0
  l1 := [message("value == "), ((rep n).val)::0]
  o1 : 0 := blankSeparate l1
  l2 := [message(" tower == "), ((rep n).tower)::0]
  o2 : 0 := blankSeparate l2
  if ((rep n).flag)
    then
      o3 := message(" closed == true")
    else
      o3 := message(" closed == false")
  l := [o1,o2,o3]
  bracket commaSeparate l

```

— SPLNODE.dotabb —

```

"SPLNODE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SPLNODE"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"SPLNODE" -> "BASTYPE"
"SPLNODE" -> "KOERCE"
"SPLNODE" -> "TYPE"

```

domain SPLTREE SplittingTree

— SplittingTree.input —

```

)set break resume
)sys rm -f SplittingTree.output
)spool SplittingTree.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SplittingTree

```

```

--R
--R SplittingTree(V: Join(SetCategory,Aggregate),C: Join(SetCategory,Aggregate)) is a domain co
--R Abbreviation for SplittingTree is SPLTREE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SPLTREE
--R
--R----- Operations -----
--R children : % -> List(%)                conditions : % -> List(C)
--R construct : (V,C,V,List(C)) -> %      construct : (V,C,List(%)) -> %
--R copy : % -> %                          cyclic? : % -> Boolean
--R distance : (%,% ) -> Integer           empty : () -> %
--R empty? : % -> Boolean                  eq? : (%,% ) -> Boolean
--R leaf? : % -> Boolean                   nodes : % -> List(%)
--R sample : () -> %                      updateStatus! : % -> %
--R value : % -> SplittingNode(V,C)
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?=? : (%,% ) -> Boolean if SplittingNode(V,C) has SETCAT
--R any? : ((SplittingNode(V,C) -> Boolean),%) -> Boolean if $ has finiteAggregate
--R child? : (%,% ) -> Boolean if SplittingNode(V,C) has SETCAT
--R coerce : % -> OutputForm if SplittingNode(V,C) has SETCAT
--R construct : (V,C,List(SplittingNode(V,C))) -> %
--R construct : SplittingNode(V,C) -> %
--R count : (SplittingNode(V,C),%) -> NonNegativeInteger if $ has finiteAggregate and SplittingM
--R count : ((SplittingNode(V,C) -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R ?.value : (% ,value) -> SplittingNode(V,C)
--R eval : (% ,List(SplittingNode(V,C)),List(SplittingNode(V,C))) -> % if SplittingNode(V,C) has
--R eval : (% ,SplittingNode(V,C),SplittingNode(V,C)) -> % if SplittingNode(V,C) has EVALAB(SPLNO
--R eval : (% ,Equation(SplittingNode(V,C))) -> % if SplittingNode(V,C) has EVALAB(SPLNODE(V,C))
--R eval : (% ,List(Equation(SplittingNode(V,C)))) -> % if SplittingNode(V,C) has EVALAB(SPLNODE(
--R every? : ((SplittingNode(V,C) -> Boolean),%) -> Boolean if $ has finiteAggregate
--R extractSplittingLeaf : % -> Union(%,"failed")
--R hash : % -> SingleInteger if SplittingNode(V,C) has SETCAT
--R latex : % -> String if SplittingNode(V,C) has SETCAT
--R leaves : % -> List(SplittingNode(V,C))
--R less? : (% ,NonNegativeInteger) -> Boolean
--R map : ((SplittingNode(V,C) -> SplittingNode(V,C)),%) -> %
--R map! : ((SplittingNode(V,C) -> SplittingNode(V,C)),%) -> % if $ has shallowlyMutable
--R member? : (SplittingNode(V,C),%) -> Boolean if $ has finiteAggregate and SplittingNode(V,C)
--R members : % -> List(SplittingNode(V,C)) if $ has finiteAggregate
--R more? : (% ,NonNegativeInteger) -> Boolean
--R node? : (%,% ) -> Boolean if SplittingNode(V,C) has SETCAT
--R nodeOf? : (SplittingNode(V,C),%) -> Boolean
--R parts : % -> List(SplittingNode(V,C)) if $ has finiteAggregate
--R remove : (SplittingNode(V,C),%) -> %
--R remove! : (SplittingNode(V,C),%) -> %
--R result : % -> List(Record(val: V,tower: C))
--R setchildren! : (% ,List(%)) -> % if $ has shallowlyMutable
--R setelt : (% ,value,SplittingNode(V,C)) -> SplittingNode(V,C) if $ has shallowlyMutable
--R setvalue! : (% ,SplittingNode(V,C)) -> SplittingNode(V,C) if $ has shallowlyMutable
--R size? : (% ,NonNegativeInteger) -> Boolean

```

```

--R splitNodeOf! : (%,% ,List(SplittingNode(V,C)),((C,C) -> Boolean)) -> %
--R splitNodeOf! : (%,% ,List(SplittingNode(V,C))) -> %
--R subNodeOf? : (SplittingNode(V,C),% ,((C,C) -> Boolean)) -> Boolean
--R ?~=? : (%,% ) -> Boolean if SplittingNode(V,C) has SETCAT
--R
--E 1

)spool
)lisp (bye)

```

— SplittingTree.help —

=====

SplittingTree examples

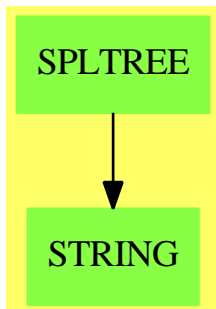
=====

This domain exports a modest implementation of splitting trees. Splitting trees are needed when the evaluation of some quantity under some hypothesis requires to split the hypothesis into sub-cases. For instance by adding some new hypothesis on one hand and its negation on another hand. The computations are terminated is a splitting tree *a* when `status(value(a))` is true. Thus, if for the splitting tree *a* the flag `status(value(a))` is true, then `status(value(d))` is true for any subtree *d* of *a*. This property of splitting trees is called the termination condition. If no vertex in a splitting tree *a* is equal to another, *a* is said to satisfy the no-duplicates condition. The splitting tree *a* will satisfy this condition if nodes are added to *a* by mean of `splitNodeOf!` and if `construct` is only used to create the root of *a* with no children.

See Also:

o `)show SplittingTree`

SplittingTree (SPLTREE)



See

⇒ “SplittingTree” (SPLTREE) [20](#) on page [2518](#)

Exports:

any?	child?	children	coerce
conditions	construct	copy	count
cyclic?	distance	empty	empty?
eq?	eval	every?	extractSplittingLeaf
hash	latex	leaf?	leaves
less?	map	map!	member?
members	more?	node?	nodeOf?
nodes	parts	remove	remove!
result	sample	setchildren!	setelt
setValue!	size?	splitNodeOf!	splitNodeOf!
subNodeOf?	updateStatus!	value	#?
?=?	?.value	?~=?	

— domain SPLTREE SplittingTree —

```

)abbrev domain SPLTREE SplittingTree
++ Author: Marc Moereno Maza
++ Date Created: 07/05/1996
++ Date Last Updated: 07/19/1996
++ References:
++   M. MORENO MAZA "Calculs de pgcd au-dessus des tours
++   d'extensions simples et resolution des systemes d'equations
++   algebriques" These, Universite P.etM. Curie, Paris, 1997.
++ Description:
++ This domain exports a modest implementation of splitting
++ trees. Splitting trees are needed when the
++ evaluation of some quantity under some hypothesis
++ requires to split the hypothesis into sub-cases.
++ For instance by adding some new hypothesis on one
++ hand and its negation on another hand. The computations
++ are terminated is a splitting tree \axiom{a} when
  
```

```

++ \axiom{status(value(a))} is \axiom{true}. Thus,
++ if for the splitting tree \axiom{a} the flag
++ \axiom{status(value(a))} is \axiom{true}, then
++ \axiom{status(value(d))} is \axiom{true} for any
++ subtree \axiom{d} of \axiom{a}. This property
++ of splitting trees is called the termination
++ condition. If no vertex in a splitting tree \axiom{a}
++ is equal to another, \axiom{a} is said to satisfy
++ the no-duplicates condition. The splitting
++ tree \axiom{a} will satisfy this condition
++ if nodes are added to \axiom{a} by mean of
++ \axiom{splitNodeOf!} and if \axiom{construct}
++ is only used to create the root of \axiom{a}
++ with no children.

```

SplittingTree(V,C) : Exports == Implementation where

```

V:Join(SetCategory,Aggregate)
C:Join(SetCategory,Aggregate)
B ==> Boolean
O ==> OutputForm
NNI ==> NonNegativeInteger
VT ==> Record(val:V, tower:C)
VTB ==> Record(val:V, tower:C, flag:B)
S ==> SplittingNode(V,C)
A ==> Record(root:S,subTrees:List(%))

Exports == RecursiveAggregate(S) with
  shallowlyMutable
  finiteAggregate
  extractSplittingLeaf : % -> Union(%, "failed")
    ++ \axiom{extractSplittingLeaf(a)} returns the left
    ++ most leaf (as a tree) whose status is false
    ++ if any, else "failed" is returned.
  updateStatus! : % -> %
    ++ \axiom{updateStatus!(a)} returns a where the status
    ++ of the vertices are updated to satisfy
    ++ the "termination condition".
  construct : S -> %
    ++ \axiom{construct(s)} creates a splitting tree
    ++ with value (i.e. root vertex) given by
    ++ \axiom{s} and no children. Thus, if the
    ++ status of \axiom{s} is false, \axiom{[s]}
    ++ represents the starting point of the
    ++ evaluation \axiom{value(s)} under the
    ++ hypothesis \axiom{condition(s)}.
  construct : (V,C, List %) -> %
    ++ \axiom{construct(v,t,la)} creates a splitting tree
    ++ with value (i.e. root vertex) given by
    ++ \axiom{[v,t]$S} and with \axiom{la} as

```

```

++ children list.
construct : (V,C,List S) -> %
++ \axiom{construct(v,t,ls)} creates a splitting tree
++ with value (i.e. root vertex) given by
++ \axiom{[v,t]$S} and with children list given by
++ \axiom{[[s]$% for s in ls]}.
construct : (V,C,V,List C) -> %
++ \axiom{construct(v1,t,v2,lt)} creates a splitting tree
++ with value (i.e. root vertex) given by
++ \axiom{[v,t]$S} and with children list given by
++ \axiom{[[[v,t]$S]$% for s in ls]}.
conditions : % -> List C
++ \axiom{conditions(a)} returns the list of the conditions
++ of the leaves of a
result : % -> List VT
++ \axiom{result(a)} where \axiom{ls} is the leaves list of \axiom{a}
++ returns \axiom{[[value(s),condition(s)]$VT for s in ls]}
++ if the computations are terminated in \axiom{a} else
++ an error is produced.
nodeOf? : (S,%) -> B
++ \axiom{nodeOf?(s,a)} returns true iff some node of \axiom{a}
++ is equal to \axiom{s}
subNodeOf? : (S,%,(C,C) -> B) -> B
++ \axiom{subNodeOf?(s,a,sub?) } returns true iff for some node
++ \axiom{n} in \axiom{a} we have \axiom{s = n} or
++ \axiom{status(n)} and \axiom{subNode?(s,n,sub?)}.
remove : (S,%) -> %
++ \axiom{remove(s,a)} returns the splitting tree obtained
++ from a by removing every sub-tree \axiom{b} such
++ that \axiom{value(b)} and \axiom{s} have the same
++ value, condition and status.
remove! : (S,%) -> %
++ \axiom{remove!(s,a)} replaces a by remove(s,a)
splitNodeOf! : (%,%,List(S)) -> %
++ \axiom{splitNodeOf!(l,a,ls)} returns \axiom{a} where the children
++ list of \axiom{l} has been set to
++ \axiom{[[s]$% for s in ls | not nodeOf?(s,a)]}.
++ Thus, if \axiom{l} is not a node of \axiom{a}, this
++ latter splitting tree is unchanged.
splitNodeOf! : (%,%,List(S),(C,C) -> B) -> %
++ \axiom{splitNodeOf!(l,a,ls,sub?) } returns \axiom{a} where the children
++ list of \axiom{l} has been set to
++ \axiom{[[s]$% for s in ls | not subNodeOf?(s,a,sub?)}.
++ Thus, if \axiom{l} is not a node of \axiom{a}, this
++ latter splitting tree is unchanged.

```

Implementation == add

Rep ==> A

```

rep(n:%):Rep == n pretend Rep
per(r:Rep):% == r pretend %

construct(s:S) ==
  per [s,[]]$A
construct(v:V,t:C,la:List(%)) ==
  per [[v,t]$S,la]$A
construct(v:V,t:C,ls:List(S)) ==
  per [[v,t]$S,[[s]$% for s in ls]]$A
construct(v1:V,t:C,v2:V,lt:List(C)) ==
  [v1,t,([v2,lt]$S)@(List S)]$%

empty?(a:%) == empty?((rep a).root) and empty?((rep a).subTrees)
empty() == [empty()]$S$%

remove(s:S,a:%) ==
  empty? a => a
  (s = value(a)) and (status(s) = status(value(a))) => empty()$%
  la := children(a)
  lb : List % := []
  while (not empty? la) repeat
    lb := cons(remove(s,first la), lb)
    la := rest la
  lb := reverse remove(empty?,lb)
  [value(value(a)),condition(value(a)),lb]$%

remove!(s:S,a:%) ==
  empty? a => a
  (s = value(a)) and (status(s) = status(value(a))) =>
    (rep a).root := empty()$S
    (rep a).subTrees := []
    a
  la := children(a)
  lb : List % := []
  while (not empty? la) repeat
    lb := cons(remove!(s,first la), lb)
    la := rest la
  lb := reverse remove(empty()$%,lb)
  setchildren!(a,lb)

value(a:%) ==
  (rep a).root
children(a:%) ==
  (rep a).subTrees
leaf?(a:%) ==
  empty? a => false
  empty? (rep a).subTrees
setchildren!(a:%,la:List(%)) ==
  (rep a).subTrees := la

```

```

a
setvalue!(a:%,s:S) ==
  (rep a).root := s
s
cyclic?(a:%) == false
map(foo:(S -> S),a:%) ==
  empty? a => a
  b : % := [foo(value(a))]%
  leaf? a => b
  setchildren!(b,[map(foo,c) for c in children(a)])
map!(foo:(S -> S),a:%) ==
  empty? a => a
  setvalue!(a,foo(value(a)))
  leaf? a => a
  setchildren!(a,[map!(foo,c) for c in children(a)])
copy(a:%) ==
  map(copy,a)
eq?(a1:%,a2:%) ==
  error"in eq? from SPLTREE : la vache qui rit est-elle folle?"
nodes(a:%) ==
  empty? a => []
  leaf? a => [a]
  cons(a,concat([nodes(c) for c in children(a)]))
leaves(a:%) ==
  empty? a => []
  leaf? a => [value(a)]
  concat([leaves(c) for c in children(a)])
members(a:%) ==
  empty? a => []
  leaf? a => [value(a)]
  cons(value(a),concat([members(c) for c in children(a)]))
#(a:%) ==
  empty? a => 0$NNI
  leaf? a => 1$NNI
  reduce("+",[#c for c in children(a)],1$NNI)$(List NNI)
a1:% = a2:% ==
  empty? a1 => empty? a2
  empty? a2 => false
  leaf? a1 =>
    not leaf? a2 => false
    value(a1) == value(a2)
  leaf? a2 => false
  value(a1) ~= value(a2) => false
  children(a1) = children(a2)
-- sample() == [sample()$S]%
localCoerce(a:%,k:NNI):0 ==
  s : String
  if k = 1 then s := "*" else s := "-> "
  for i in 2..k repeat s := concat("--+",s)$String
  ro : 0 := left(hconcat(message(s)$0,value(a)::0)$0)$0

```

```

leaf? a => ro
lo : List 0 := [localCoerce(c,k+1) for c in children(a)]
lo := cons(ro,lo)
vconcat(lo)$0
coerce(a:%) : 0 ==
  empty? a => vconcat(message(" ")$0,message("* []")$0)
  vconcat(message(" ")$0,localCoerce(a,1))

extractSplittingLeaf(a:%) ==
  empty? a => "failed"::Union(%, "failed")
  status(value(a))$S => "failed"::Union(%, "failed")
  la := children(a)
  empty? la => a
  while (not empty? la) repeat
    esl := extractSplittingLeaf(first la)
    (esl case %) => return(esl)
    la := rest la
  "failed"::Union(%, "failed")

updateStatus!(a:%) ==
  la := children(a)
  (empty? la) or (status(value(a))$S) => a
  done := true
  while (not empty? la) and done repeat
    done := done and status(value(updateStatus! first la))
    la := rest la
  setStatus!(value(a),done)$S
  a

result(a:%) ==
  empty? a => []
  not status(value(a))$S =>
    error"in result from SPLTREE : mad cow!"
  ls : List S := leaves(a)
  [[value(s),condition(s)]$VT for s in ls]

conditions(a:%) ==
  empty? a => []
  ls : List S := leaves(a)
  [condition(s) for s in ls]

nodeOf?(s:S,a:%) ==
  empty? a => false
  s = $S value(a) => true
  la := children(a)
  while (not empty? la) and (not nodeOf?(s,first la)) repeat
    la := rest la
  not empty? la

subNodeOf?(s:S,a:%,sub?:((C,C) -> B)) ==

```

```

empty? a => false
-- s = $S value(a) => true
status(value(a)$%)$S and subNode?(s,value(a),sub?)$S => true
la := children(a)
while (not empty? la) and (not subNodeOf?(s,first la,sub?)) repeat
  la := rest la
not empty? la

splitNodeOf!(l:%,a:%,ls:List(S)) ==
  ln := removeDuplicates ls
  la : List % := []
  while not empty? ln repeat
    if not nodeOf?(first ln,a)
    then
      la := cons([first ln]$%, la)
      ln := rest ln
  la := reverse la
  setchildren!(l,la)$%
  if empty? la then (rep l).root := [empty()$V,empty()$C,true]$S
  updateStatus!(a)

splitNodeOf!(l:%,a:%,ls:List(S),sub?:((C,C) -> B)) ==
  ln := removeDuplicates ls
  la : List % := []
  while not empty? ln repeat
    if not subNodeOf?(first ln,a,sub?)
    then
      la := cons([first ln]$%, la)
      ln := rest ln
  la := reverse la
  setchildren!(l,la)$%
  if empty? la then (rep l).root := [empty()$V,empty()$C,true]$S
  updateStatus!(a)

```

— SPLTREE.dotabb —

```

"SPLTREE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SPLTREE"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"SPLTREE" -> "STRING"

```

domain SREGSET SquareFreeRegularTriangularSet

— SquareFreeRegularTriangularSet.input —

```

)set break resume
)sys rm -f SquareFreeRegularTriangularSet.output
)spool SquareFreeRegularTriangularSet.output
)set message test on
)set message auto off
)clear all
--S 1 of 23
R := Integer
--R
--R
--R (1) Integer
--R
--R                                          Type: Domain
--E 1

--S 2 of 23
ls : List Symbol := [x,y,z,t]
--R
--R
--R (2) [x,y,z,t]
--R
--R                                          Type: List(Symbol)
--E 2

--S 3 of 23
V := OVAR(ls)
--R
--R
--R (3) OrderedVariableList([x,y,z,t])
--R
--R                                          Type: Domain
--E 3

--S 4 of 23
E := IndexedExponents V
--R
--R
--R (4) IndexedExponents(OrderedVariableList([x,y,z,t]))
--R
--R                                          Type: Domain
--E 4

--S 5 of 23
P := NSMP(R, V)
--R
--R
--R (5)
--R NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))

```



```

--R                                                    Type: Domain
--E 5

--S 6 of 23
x: P := 'x
--R
--R
--R (6) x
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--E 6

--S 7 of 23
y: P := 'y
--R
--R
--R (7) y
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--E 7

--S 8 of 23
z: P := 'z
--R
--R
--R (8) z
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--E 8

--S 9 of 23
t: P := 't
--R
--R
--R (9) t
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--E 9

--S 10 of 23
ST := SREGSET(R,E,V,P)
--R
--R
--R (10)
--R SquareFreeRegularTriangularSet(Integer,IndexedExponents(OrderedVariableList([
--R x,y,z,t])),OrderedVariableList([x,y,z,t]),NewSparseMultivariatePolynomial(Int
--R eger,OrderedVariableList([x,y,z,t])))
--R                                                    Type: Domain
--E 10

--S 11 of 23
p1 := x ** 31 - x ** 6 - x - y
--R
--R

```

```

--R      31      6
--R (11)  $x^{31} - x^6 - x - y$ 
--RType: NewSparseMultivariatePolynomial(Integer, OrderedVariableList([x,y,z,t]))
--E 11

--S 12 of 23
p2 := x ** 8 - z
--R
--R
--R      8
--R (12)  $x^8 - z$ 
--RType: NewSparseMultivariatePolynomial(Integer, OrderedVariableList([x,y,z,t]))
--E 12

--S 13 of 23
p3 := x ** 10 - t
--R
--R
--R      10
--R (13)  $x^{10} - t$ 
--RType: NewSparseMultivariatePolynomial(Integer, OrderedVariableList([x,y,z,t]))
--E 13

--S 14 of 23
lp := [p1, p2, p3]
--R
--R
--R      31      6      8      10
--R (14)  $[x^{31} - x^6 - x - y, x^8 - z, x^{10} - t]$ 
--RType: List(NewSparseMultivariatePolynomial(Integer, ...
--E 14

--S 15 of 23
zeroSetSplit(lp)$ST
--R
--R
--R      5      4      2      3      8      5      3      2      4      2
--R (15)  $[\{z^5 - t^4, t^2 z y + 2z^2 y - t^3 + 2t^5 + t^3 - t^2, (t^8 - t)x - t^2 y - z^2\}]$ 
--RType: List(SquareFreeRegularTriangularSet(Integer, ...
--E 15

--S 16 of 23
zeroSetSplit(lp, false)$ST
--R
--R
--R      5      4      2      3      8      5      3      2      4      2
--R (16)  $[\{z^5 - t^4, t^2 z y + 2z^2 y - t^3 + 2t^5 + t^3 - t^2, (t^8 - t)x - t^2 y - z^2\},$ 
--R      3      5      2      2
--R       $\{t^3 - 1, z^5 - t, t^2 y + z^2, z^2 x - t\}, \{t, z, y, x\}]$ 

```

```

--IType: List(SquareFreeRegularTriangularSet(Integer,...
--E 16

--S 17 of 23
T := REGSET(R,E,V,P)
--R
--R
--R (17)
--R RegularTriangularSet(Integer,IndexedExponents(OrderedVariableList([x,y,z,t]))
--R ,OrderedVariableList([x,y,z,t]),NewSparseMultivariatePolynomial(Integer,Order
--R edVariableList([x,y,z,t])))
--R
--R Type: Domain
--E 17

--S 18 of 23
lts := zeroSetSplit(lp,false)$T
--R
--R
--R (18)
--R 
$$\{z^5 - t^4, t^2 z^3 y + 2z^2 y - t^8 + 2t^5 + t^3 - t^2, (t^5 - t^2)x - t^4 y - z^2\},$$

--R 
$$\{t^3 - 1, z^5 - t^2, t^2 z^3 y + 2z^2 y + 1, z^3 x - t^2\}, \{t, z, y, x\}$$

--IType: List(RegularTriangularSet(Integer,IndexedExponents(...
--E 18

--S 19 of 23
ts := lts.2
--R
--R
--R (19)  $\{t^3 - 1, z^5 - t^2, t^2 z^3 y + 2z^2 y + 1, z^3 x - t^2\}$ 
--IType: RegularTriangularSet(Integer,IndexedExponents(...
--E 19

--S 20 of 23
pol := select(ts,'y')$T
--R
--R
--R (20)  $t^2 z^3 y + 2z^2 y + 1$ 
--IType: Union(NewSparseMultivariatePolynomial(Integer,...
--E 20

--S 21 of 23
tower := collectUnder(ts,'y')$T
--R
--R
--R (21)  $\{t^3 - 1, z^5 - t^2\}$ 

```

```

--IType: RegularTriangularSet(Integer,IndexedExponents(...)
--E 21

--S 22 of 23
pack := RegularTriangularSetGcdPackage(R,E,V,P,T)
--R
--R
--R (22)
--R RegularTriangularSetGcdPackage(Integer,IndexedExponents(OrderedVariableList([
--R x,y,z,t])),OrderedVariableList([x,y,z,t]),NewSparseMultivariatePolynomial(Int
--R eger,OrderedVariableList([x,y,z,t]),RegularTriangularSet(Integer,IndexedExpo
--R nents(OrderedVariableList([x,y,z,t]),OrderedVariableList([x,y,z,t]),NewSpars
--R eMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t])))
--R
--R                                         Type: Domain
--E 22

--S 23 of 23
toseSquareFreePart(pol,tower)$pack
--R
--R
--R
--R (23) [[val= t y + z ,tower= {t - 1,z - t}]]
--IType: List(Record(val: NewSparseMultivariatePolynomial(...)
--E 23
)spool
)lisp (bye)

```

— SquareFreeRegularTriangularSet.help —

=====

SquareFreeRegularTriangularSet examples

=====

The SquareFreeRegularTriangularSet domain constructor implements square-free regular triangular sets. See the RegularTriangularSet domain constructor for general regular triangular sets. Let T be a regular triangular set consisting of polynomials t1, ..., tm ordered by increasing main variables. The regular triangular set T is square-free if T is empty or if t1, ..., tm-1 is square-free and if the polynomial tm is square-free as a univariate polynomial with coefficients in the tower of simple extensions associated with t1,...,tm-1.

The main interest of square-free regular triangular sets is that their associated towers of simple extensions are product of fields. Consequently, the saturated ideal of a square-free regular triangular set is radical. This property simplifies some of the operations related to regular triangular sets. However, building square-free

regular triangular sets is generally more expensive than building general regular triangular sets.

As the `RegularTriangularSet` domain constructor, the `SquareFreeRegularTriangularSet` domain constructor also implements a method for solving polynomial systems by means of regular triangular sets. This is in fact the same method with some adaptations to take into account the fact that the computed regular chains are square-free. Note that it is also possible to pass from a decomposition into general regular triangular sets to a decomposition into square-free regular triangular sets. This conversion is used internally by the `LazardSetSolvingPackage` package constructor.

N.B. When solving polynomial systems with the `SquareFreeRegularTriangularSet` domain constructor or the `LazardSetSolvingPackage` package constructor, decompositions have no redundant components. See also `LexTriangularPackage` and `ZeroDimensionalSolvePackage` for the case of algebraic systems with a finite number of (complex) solutions.

We shall explain now how to use the constructor `SquareFreeRegularTriangularSet`.

This constructor takes four arguments. The first one, `R`, is the coefficient ring of the polynomials; it must belong to the category `GcdDomain`. The second one, `E`, is the exponent monoid of the polynomials; it must belong to the category `OrderedAbelianMonoidSup`. The third one, `V`, is the ordered set of variables; it must belong to the category `OrderedSet`. The last one is the polynomial ring; it must belong to the category `RecursivePolynomialCategory(R,E,V)`. The abbreviation for `SquareFreeRegularTriangularSet` is `SREGSET`.

Note that the way of understanding triangular decompositions is detailed in the example of the `RegularTriangularSet` constructor.

Let us illustrate the use of this constructor with one example (Donati-Traverso). Define the coefficient ring.

```
R := Integer
Integer
Type: Domain
```

Define the list of variables,

```
ls : List Symbol := [x,y,z,t]
[x,y,z,t]
Type: List Symbol
```

and make it an ordered set;

```
V := OVAR(ls)
```

```

OrderedVariableList [x,y,z,t]
                                Type: Domain

then define the exponent monoid.

E := IndexedExponents V
IndexedExponents OrderedVariableList [x,y,z,t]
                                Type: Domain

Define the polynomial ring.

P := NSMP(R, V)
NewSparseMultivariatePolynomial(Integer,OrderedVariableList [x,y,z,t])
                                Type: Domain

Let the variables be polynomial.

x: P := 'x
x
                                Type: NewSparseMultivariatePolynomial(Integer,
                                OrderedVariableList [x,y,z,t])

y: P := 'y
y
                                Type: NewSparseMultivariatePolynomial(Integer,
                                OrderedVariableList [x,y,z,t])

z: P := 'z
z
                                Type: NewSparseMultivariatePolynomial(Integer,
                                OrderedVariableList [x,y,z,t])

t: P := 't
t
                                Type: NewSparseMultivariatePolynomial(Integer,
                                OrderedVariableList [x,y,z,t])

Now call the SquareFreeRegularTriangularSet domain constructor.

ST := SREGSET(R,E,V,P)
SquareFreeRegularTriangularSet(Integer,IndexedExponents OrderedVariableList [
x,y,z,t],OrderedVariableList [x,y,z,t],NewSparseMultivariatePolynomial(Intege
r,OrderedVariableList [x,y,z,t]))
                                Type: Domain

Define a polynomial system.

p1 := x ** 31 - x ** 6 - x - y
      31      6
      x  - x  - x - y

```

```

Type: NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t])

p2 := x ** 8 - z
      8
      x - z
Type: NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t])

p3 := x ** 10 - t
      10
      x - t
Type: NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t])

lp := [p1, p2, p3]
      31      6      8      10
[x - x - x - y, x - z, x - t]
Type: List NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t])

```

First of all, let us solve this system in the sense of Kalkbrener.

```

zeroSetSplit(lp)$ST
      5      4      2      3      8      5      3      2      4      2
[{z - t , t z y + 2z y - t + 2t + t - t , (t - t)x - t y - z },
Type: List SquareFreeRegularTriangularSet(Integer,
IndexedExponents OrderedVariableList [x,y,z,t],
OrderedVariableList [x,y,z,t],
NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t]))

```

And now in the sense of Lazard (or Wu and other authors).

```

zeroSetSplit(lp,false)$ST
      5      4      2      3      8      5      3      2      4      2
[{z - t , t z y + 2z y - t + 2t + t - t , (t - t)x - t y - z },
      3      5      2      2
{t - 1, z - t, t y + z , z x - t}, {t,z,y,x}]
Type: List SquareFreeRegularTriangularSet(Integer,
IndexedExponents OrderedVariableList [x,y,z,t],
OrderedVariableList [x,y,z,t],
NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t]))

```

Now to see the difference with the RegularTriangularSet domain constructor, we define:

```

T := REGSET(R,E,V,P)
RegularTriangularSet(Integer,IndexedExponents OrderedVariableList [x,y,z,t],0

```

```

rderedVariableList [x,y,z,t],NewSparseMultivariatePolynomial(Integer,OrderedV
variableList [x,y,z,t]))

```

Type: Domain

and compute:

```

lts := zeroSetSplit(lp,false)$T
      5      4      2      3      8      5      3      2      4      2
      [z  - t ,t z y  + 2z y - t  + 2t  + t  - t ,(t  - t)x - t y - z },
      3      5      2      3      2
      {t  - 1,z  - t,t z y  + 2z y + 1,z x  - t}, {t,z,y,x}]
      Type: List RegularTriangularSet(Integer,
      IndexedExponents OrderedVariableList [x,y,z,t],
      OrderedVariableList [x,y,z,t],
      NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t]))

```

If you look at the second set in both decompositions in the sense of Lazard, you will see that the polynomial with main variable y is not the same.

Let us understand what has happened.

We define:

```

ts := lts.2
      3      5      2      3      2
(19) {t  - 1,z  - t,t z y  + 2z y + 1,z x  - t}
      Type: RegularTriangularSet(Integer,
      IndexedExponents OrderedVariableList [x,y,z,t],
      OrderedVariableList [x,y,z,t],
      NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t]))

pol := select(ts,'y)$T
      2      3
      t z y  + 2z y + 1
      Type: Union(NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t]),...)

tower := collectUnder(ts,'y)$T
      3      5
      {t  - 1,z  - t}
      Type: RegularTriangularSet(Integer,
      IndexedExponents OrderedVariableList [x,y,z,t],
      OrderedVariableList [x,y,z,t],
      NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t]))

pack := RegularTriangularSetGcdPackage(R,E,V,P,T)
RegularTriangularSetGcdPackage(Integer,IndexedExponents OrderedVariableList [

```



```

x,y,z,t],OrderedVariableList [x,y,z,t],NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t]),RegularTriangularSet(Integer,IndexedExponents
OrderedVariableList [x,y,z,t],OrderedVariableList [x,y,z,t],NewSparseMultiv
ariatePolynomial(Integer,OrderedVariableList [x,y,z,t])))
Type: Domain

```

Then we compute:

```

toseSquareFreePart(pol,tower)$pack
      2      3      5
[[val= t y + z ,tower= {t - 1,z - t}]]
Type: List Record(val: NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t]),
tower: RegularTriangularSet(Integer,
IndexedExponents OrderedVariableList [x,y,z,t],
OrderedVariableList [x,y,z,t],
NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t])))

```

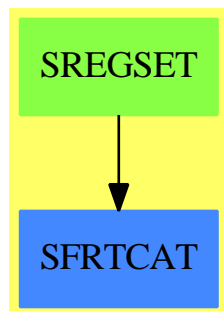
See Also:

```

o )help GcdDomain
o )help OrderedAbelianMonoidSup
o )help OrderedSet
o )help RecursivePolynomialCategory
o )help ZeroDimensionalSolvePackage
o )help LexTriangularPackage
o )help LazardSetSolvingPackage
o )help RegularTriangularSet
o )show SquareFreeRegularTriangularSet

```

SquareFreeRegularTriangularSet (SREGSET)



Exports:

algebraic?	algebraicCoefficients?
algebraicVariables	any?
augment	autoReduced?
basicSet	coerce
coHeight	collect
collectQuasiMonic	collectUnder
collectUpper	convert
construct	copy
count	degree
empty	empty?
eq?	eval
every?	extend
extendIfCan	find
first	hash
headReduce	headReduced?
headRemainder	inFRittWu?
initiallyReduce	initiallyReduced?
initials	internalAugment
internalZeroSetSplit	intersect
invertible?	invertibleSet
invertibleElseSplit?	last
lastSubResultant	lastSubResultantElseSplit
less?	latex
mainVariable?	mainVariables
map	map!
member?	members
more?	mvar
normalized?	parts
preprocess	purelyAlgebraic?
purelyAlgebraicLeadingMonomial?	purelyTranscendental?
quasiComponent	reduce
reduceByQuasiMonic	reduced?
remainder	remove
removeDuplicates	removeZero
rest	retract
retractIfCan	rewriteIdealWithHeadRemainder
rewriteIdealWithRemainder	rewriteSetWithReduction
roughBase?	roughEqualIdeals?
roughSubIdeal?	roughUnitIdeal?
sample	select
size?	sort
squareFreePart	stronglyReduce
stronglyReduced?	triangular?
trivialIdeal?	variables
zeroSetSplit	zeroSetSplitIntoTriangularSystems
#?	?=?
?~=?	

— domain SREGSET SquareFreeRegularTriangularSet —

```

)abbrev domain SREGSET SquareFreeRegularTriangularSet
++ Author: Marc Moreno Maza
++ Date Created: 08/25/1998
++ Date Last Updated: 16/12/1998
++ References :
++ [1] M. MORENO MAZA "A new algorithm for computing triangular
++      decomposition of algebraic varieties" NAG Tech. Rep. 4/98.
++ Description:
++ This domain provides an implementation of square-free regular chains.
++ Moreover, the operation zeroSetSplit
++ is an implementation of a new algorithm for solving polynomial systems by
++ means of regular chains.

SquareFreeRegularTriangularSet(R,E,V,P) : Exports == Implementation where

  R : GcdDomain
  E : OrderedAbelianMonoidSup
  V : OrderedSet
  P : RecursivePolynomialCategory(R,E,V)
  N ==> NonNegativeInteger
  Z ==> Integer
  B ==> Boolean
  LP ==> List P
  PtoP ==> P -> P
  PS ==> GeneralPolynomialSet(R,E,V,P)
  PWT ==> Record(val : P, tower : $)
  BWT ==> Record(val : Boolean, tower : $)
  LpWT ==> Record(val : (List P), tower : $)
  Split ==> List $
  iprintpack ==> InternalPrintPackage()
  polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
  quasicomppack ==> SquareFreeQuasiComponentPackage(R,E,V,P,$)
  regsetgcdpack ==> SquareFreeRegularTriangularSetGcdPackage(R,E,V,P,$)
  regsetdecomppack ==> SquareFreeRegularSetDecompositionPackage(R,E,V,P,$)

Exports == SquareFreeRegularTriangularSetCategory(R,E,V,P) with

  internalAugment: (P,$,B,B,B,B,B) -> List $
    ++ \axiom{internalAugment(p,ts,b1,b2,b3,b4,b5)}
    ++ is an internal subroutine, exported only for development.
  zeroSetSplit: (LP, B, B) -> Split
    ++ \axiom{zeroSetSplit(lp,clos?,info?)} has the same specifications as
    ++ zeroSetSplit from RegularTriangularSetCategory
    ++ from \spadtype{RegularTriangularSetCategory}
    ++ Moreover, if clos? then solves in the sense of the Zariski closure
    ++ else solves in the sense of the regular zeros. If \axiom{info?} then
    ++ do print messages during the computations.

```

```

zeroSetSplit: (LP, B, B, B, B) -> Split
  ++ \axiom{zeroSetSplit(lp,b1,b2.b3,b4)}
  ++ is an internal subroutine, exported only for developement.
internalZeroSetSplit: (LP, B, B, B) -> Split
  ++ \axiom{internalZeroSetSplit(lp,b1,b2,b3)}
  ++ is an internal subroutine, exported only for developement.
pre_process: (LP, B, B) -> Record(val: LP, towers: Split)
  ++ \axiom{pre_process(lp,b1,b2)}
  ++ is an internal subroutine, exported only for developement.

```

Implementation == add

```

Rep ==> LP

rep(s:$):Rep == s pretend Rep
per(l:Rep):$ == l pretend $

copy ts ==
  per(copy(rep(ts))$LP)
empty() ==
  per([])
empty?(ts:$) ==
  empty?(rep(ts))
parts ts ==
  rep(ts)
members ts ==
  rep(ts)
map (f : PtoP, ts : $) : $ ==
  construct(map(f,rep(ts))$LP)$
map! (f : PtoP, ts : $) : $ ==
  construct(map!(f,rep(ts))$LP)$
member? (p,ts) ==
  member?(p,rep(ts))$LP
unitIdealIfCan() ==
  "failed"::Union($,"failed")
roughUnitIdeal? ts ==
  false
coerce(ts:$) : OutputForm ==
  lp : List(P) := reverse(rep(ts))
  brace([p::OutputForm for p in lp]$List(OutputForm))$OutputForm
mvar ts ==
  empty? ts => error "mvar$SREGSET: #1 is empty"
  mvar(first(rep(ts)))$P
first ts ==
  empty? ts => "failed"::Union(P,"failed")
  first(rep(ts))::Union(P,"failed")
last ts ==
  empty? ts => "failed"::Union(P,"failed")
  last(rep(ts))::Union(P,"failed")
rest ts ==

```

```

    empty? ts => "failed"::Union($,"failed")
    per(rest(rep(ts)))::Union($,"failed")
coerce(ts:$) : (List P) ==
    rep(ts)

collectUpper (ts,v) ==
    empty? ts => ts
    lp := rep(ts)
    newlp : Rep := []
    while (not empty? lp) and (mvar(first(lp)) > v) repeat
        newlp := cons(first(lp),newlp)
        lp := rest lp
    per(reverse(newlp))

collectUnder (ts,v) ==
    empty? ts => ts
    lp := rep(ts)
    while (not empty? lp) and (mvar(first(lp)) >= v) repeat
        lp := rest lp
    per(lp)

construct(lp:List(P)) ==
    ts : $ := per([])
    empty? lp => ts
    lp := sort(infRittWu?,lp)
    while not empty? lp repeat
        eif := extendIfCan(ts,first(lp))
        not (eif case $) =>
            error"in construct : List P -> $ from SREGSET : bad #1"
        ts := eif::$
        lp := rest lp
    ts

extendIfCan(ts:$,p:P) ==
    ground? p => "failed"::Union($,"failed")
    empty? ts =>
        p := squareFreePart primitivePart p
        (per([p]))::Union($,"failed")
    not (mvar(ts) < mvar(p)) => "failed"::Union($,"failed")
    invertible?(init(p),ts)@Boolean =>
        lts: Split := augment(p,ts)
        #lts ~ 1 => "failed"::Union($,"failed")
        (first lts)::Union($,"failed")
    "failed"::Union($,"failed")

removeZero(p:P, ts:$): P ==
    (ground? p) or (empty? ts) => p
    v := mvar(p)
    ts_v_- := collectUnder(ts,v)
    if algebraic?(v,ts)

```

```

    then
      q := lazyPrem(p,select(ts,v)::P)
      zero? q => return q
      zero? removeZero(q,ts_v_-) => return 0
    empty? ts_v_- => p
  q: P := 0
  while positive? degree(p,v) repeat
    q := removeZero(init(p),ts_v_-) * mainMonomial(p) + q
    p := tail(p)
  q + removeZero(p,ts_v_-)

internalAugment(p:P,ts:$): $ ==
  -- ASSUME that adding p to ts DOES NOT require any split
  ground? p => error "in internalAugment$SREGSET: ground? #1"
  first(internalAugment(p,ts,false,false,false,false,false))

internalAugment(lp:List(P),ts:$): $ ==
  -- ASSUME that adding p to ts DOES NOT require any split
  empty? lp => ts
  internalAugment(rest lp, internalAugment(first lp, ts))

internalAugment(p:P,ts:$,rem?:B,red?:B,prim?:B,sqfr?:B,extend?:B):Split ==
  -- ASSUME p is not a constant
  -- ASSUME mvar(p) is not algebraic w.r.t. ts
  -- ASSUME init(p) invertible modulo ts
  -- if rem? then REDUCE p by remainder
  -- if prim? then REPLACE p by its main primitive part
  -- if sqfr? then FACTORIZE SQUARE FREE p over R
  -- if extend? DO NOT ASSUME every pol in ts_v_+ is invertible modulo ts
  v := mvar(p)
  ts_v_- := collectUnder(ts,v)
  ts_v_+ := collectUpper(ts,v)
  if rem? then p := remainder(p,ts_v_-).polnum
  -- if rem? then p := reduceByQuasiMonic(p,ts_v_-)
  if red? then p := removeZero(p,ts_v_-)
  if prim? then p := mainPrimitivePart p
  lts: Split
  if sqfr?
  then
    lts: Split := []
    lsfp := squareFreeFactors(p)$polsetpack
    for f in lsfp repeat
      (ground? f) or (mvar(f) < v) => "leave"
      lpwt := squareFreePart(f,ts_v_-)
      for pwt in lpwt repeat
        sfp := pwt.val; us := pwt.tower
        lts := cons( per(cons(pwt.val, rep(pwt.tower))), lts)
  else
    lts: Split := [per(cons(p,rep(ts_v_-)))]
  extend? => extend(members(ts_v_+),lts)

```

```

[per(concat(rep(ts_v_+),rep(us))) for us in lts]

augment(p:P,ts:$): List $ ==
  ground? p => error "in augment$SREGSET: ground? #1"
  algebraic?(mvar(p),ts) => error "in augment$SREGSET: bad #1"
  -- ASSUME init(p) invertible modulo ts
  -- DOES NOT ASSUME anything else.
  -- THUS reduction, mainPrimitivePart and squareFree are NEEDED
  internalAugment(p,ts,true,true,true,true,true)

extend(p:P,ts:$): List $ ==
  ground? p => error "in extend$SREGSET: ground? #1"
  v := mvar(p)
  not (mvar(ts) < mvar(p)) => error "in extend$SREGSET: bad #1"
  split: List($):= invertibleSet(init(p),ts)
  lts: List($):= []
  for us in split repeat
    lts := concat(augment(p,us),lts)
  lts

invertible?(p:P,ts:$): Boolean ==
  stoseInvertible?(p,ts)$regsetgcdpack

invertible?(p:P,ts:$): List BWT ==
  stoseInvertible?_sqfreg(p,ts)$regsetgcdpack

invertibleSet(p:P,ts:$): Split ==
  stoseInvertibleSet_sqfreg(p,ts)$regsetgcdpack

lastSubResultant(p1:P,p2:P,ts:$): List PWT ==
  stoseLastSubResultant(p1,p2,ts)$regsetgcdpack

squareFreePart(p:P, ts: $): List PWT ==
  stoseSquareFreePart(p,ts)$regsetgcdpack

intersect(p:P, ts: $): List($):=
  decompose([p], [ts], false, false)$regsetdecomppack

intersect(lp: LP, lts: List($)): List($):=
  decompose(lp, lts, false, false)$regsetdecomppack
  -- SOLVE in the regular zero sense
  -- and DO NOT PRINT info

decompose(lp: LP, lts: List($)): List($):=
  decompose(lp, lts, true, false)$regsetdecomppack
  -- SOLVE in the closure sense
  -- and DO NOT PRINT info

zeroSetSplit(lp:List(P)) == zeroSetSplit(lp,true,false)
  -- by default SOLVE in the closure sense

```

```

-- and DO NOT PRINT info

zeroSetSplit(lp:List(P), clos?: B) == zeroSetSplit(lp,clos?, false)

zeroSetSplit(lp:List(P), clos?: B, info?: B) ==
  -- if clos? then SOLVE in the closure sense
  -- if info? then PRINT info
  -- by default USE hash-tables
  -- and PREPROCESS the input system
  zeroSetSplit(lp,true,clos?,info?,true)

zeroSetSplit(lp:List(P),hash?:B,clos?:B,info?:B,prep?:B) ==
  -- if hash? then USE hash-tables
  -- if info? then PRINT information
  -- if clos? then SOLVE in the closure sense
  -- if prep? then PREPROCESS the input system
  if hash?
  then
    s1, s2, s3, dom1, dom2, dom3: String
    e: String := empty()$String
    if info? then (s1,s2,s3) := ("w","g","i") else (s1,s2,s3) := (e,e,e)
    if info?
    then
      (dom1, dom2, dom3) := _
        ("QCMPPACK", "REGSETGCD: Gcd", "REGSETGCD: Inv Set")
    else
      (dom1, dom2, dom3) := (e,e,e)
    startTable!(s1,"W",dom1)$quasicomppack
    startTableGcd!(s2,"G",dom2)$regsetgcdpack
    startTableInvSet!(s3,"I",dom3)$regsetgcdpack
  lts := internalZeroSetSplit(lp,clos?,info?,prep?)
  if hash?
  then
    stopTable!()$quasicomppack
    stopTableGcd!()$regsetgcdpack
    stopTableInvSet!()$regsetgcdpack
  lts

internalZeroSetSplit(lp:LP,clos?:B,info?:B,prep?:B) ==
  -- if info? then PRINT information
  -- if clos? then SOLVE in the closure sense
  -- if prep? then PREPROCESS the input system
  if prep?
  then
    pp := pre_process(lp,clos?,info?)
    lp := pp.val
    lts := pp.towers
  else
    ts: $ := [[]]
    lts := [ts]

```



```

lp := remove(zero?, lp)
any?(ground?, lp) => []
empty? lp => lts
empty? lts => lts
lp := sort(infRittWu?,lp)
clos? => decompose(lp,lts, clos?, info?)$regsetdecomppack
-- IN DIM > 0 with clos? the following is not false ...
for p in lp repeat
  lts := decompose([p],lts, clos?, info?)$regsetdecomppack
lts

largeSystem?(lp:LP): Boolean ==
-- Gonnet and Gerdt and not Wu-Wang.2
#lp > 16 => true
#lp < 13 => false
lts: List($) := []
(#lp :: Z - numberOfVariables(lp,lts)$regsetdecomppack :: Z) > 3

smallSystem?(lp:LP): Boolean ==
-- neural, Vermeer, Liu, and not f-633 and not Hairer-2
#lp < 5

mediumSystem?(lp:LP): Boolean ==
-- f-633 and not Hairer-2
lts: List($) := []
(numberOfVariables(lp,lts)$regsetdecomppack :: Z - #lp :: Z) < 2

--   lin?(p:P):Boolean == ground?(init(p)) and one?(mdeg(p))
lin?(p:P):Boolean == ground?(init(p)) and (mdeg(p) = 1)

pre_process(lp:LP,clos?:B,info?:B): Record(val: LP, towers: Split) ==
-- if info? then PRINT information
-- if clos? then SOLVE in the closure sense
ts: $ := [[]];
lts: Split := [ts]
empty? lp => [lp,lts]
lp1: List P := []
lp2: List P := []
for p in lp repeat
  ground? (tail p) => lp1 := cons(p, lp1)
  lp2 := cons(p, lp2)
lts: Split := decompose(lp1,[ts],clos?,info?)$regsetdecomppack
probablyZeroDim?(lp)$polsetpack =>
  largeSystem?(lp) => return [lp2,lts]
  if #lp > 7
  then
    -- Butcher (8,8) + Wu-Wang.2 (13,16)
    lp2 := crushedSet(lp2)$polsetpack
    lp2 := remove(zero?,lp2)
    any?(ground?,lp2) => return [lp2, lts]

```

```

lp3 := [p for p in lp2 | lin?(p)]
lp4 := [p for p in lp2 | not lin?(p)]
if clos?
  then
    lts := decompose(lp4,lts, clos?, info?)$regsetdecomppack
  else
    lp4 := sort(infRittWu?,lp4)
    for p in lp4 repeat
      lts := decompose([p],lts, clos?, info?)$regsetdecomppack
    lp2 := lp3
  else
    lp2 := crushedSet(lp2)$polsetpack
    lp2 := remove(zero?,lp2)
    any?(ground?,lp2) => return [lp2, lts]
if clos?
  then
    lts := decompose(lp2,lts, clos?, info?)$regsetdecomppack
  else
    lp2 := sort(infRittWu?,lp2)
    for p in lp2 repeat
      lts := decompose([p],lts, clos?, info?)$regsetdecomppack
    lp2 := []
    return [lp2,lts]
smallSystem?(lp) => [lp2,lts]
mediumSystem?(lp) => [crushedSet(lp2)$polsetpack,lts]
lp3 := [p for p in lp2 | lin?(p)]
lp4 := [p for p in lp2 | not lin?(p)]
if clos?
  then
    lts := decompose(lp4,lts, clos?, info?)$regsetdecomppack
  else
    lp4 := sort(infRittWu?,lp4)
    for p in lp4 repeat
      lts := decompose([p],lts, clos?, info?)$regsetdecomppack
if clos?
  then
    lts := decompose(lp3,lts, clos?, info?)$regsetdecomppack
  else
    lp3 := sort(infRittWu?,lp3)
    for p in lp3 repeat
      lts := decompose([p],lts, clos?, info?)$regsetdecomppack
lp2 := []
return [lp2,lts]

```

```
"SREGSET" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SREGSET"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"SREGSET" -> "SFRTCAT"
```

domain SQMATRIX SquareMatrix

— SquareMatrix.input —

```
)set break resume
)sys rm -f SquareMatrix.output
)spool SquareMatrix.output
)set message test on
)set message auto off
)clear all
--S 1 of 6
)set expose add constructor SquareMatrix
--R
--I SquareMatrix is now explicitly exposed in frame frame0
--E 1

--S 2 of 6
m := squareMatrix [ [1,-%i],[%i,4] ]
--R
--R
--R      +1   - %i+
--R  (1)  |      |
--R      +%i   4   +
--R
--R                                          Type: SquareMatrix(2,Complex(Integer))
--E 2

--S 3 of 6
m*m - m
--R
--R
--R      + 1   - 4%i+
--R  (2)  |      |
--R      +4%i  13   +
--R
--R                                          Type: SquareMatrix(2,Complex(Integer))
--E 3

--S 4 of 6
mm := squareMatrix [ [m, 1], [1-m, m**2] ]
--R
--R
```

```

--R      ++1  - %i+      +1  0+  +
--R      ||      |      |  |  |
--R      |+%i  4  +      +0  1+  |
--R  (3)  |      |      |
--R      |+ 0      %i +  + 2  - 5%i+|
--R      ||      |  |      ||
--R      +-+ %i  - 3+  +5%i  17  ++
--R                                     Type: SquareMatrix(2,SquareMatrix(2,Complex(Integer)))
--E 4

```

```

--S 5 of 6
p := (x + m)**2
--R
--R
--R      2  + 2  - 2%i+      + 2  - 5%i+
--R  (4)  x  + |      |x + |      |
--R      +2%i  8  +      +5%i  17  +
--R                                     Type: Polynomial(SquareMatrix(2,Complex(Integer)))
--E 5

```

```

--S 6 of 6
p::SquareMatrix(2, ?)
--R
--R
--R      + 2      +
--R      |x  + 2x + 2  - 2%i x - 5%i|
--R  (5)  |      |
--R      |      2      |
--R      +2%i x + 5%i  x  + 8x + 17  +
--R                                     Type: SquareMatrix(2,Polynomial(Complex(Integer)))
--E 6
)spool
)lisp (bye)

```

— SquareMatrix.help —

SquareMatrix examples

The top level matrix type in Axiom is `Matrix`, which provides basic arithmetic and linear algebra functions. However, since the matrices can be of any size it is not true that any pair can be added or multiplied. Thus `Matrix` has little algebraic structure.

Sometimes you want to use matrices as coefficients for polynomials or in other algebraic contexts. In this case, `SquareMatrix` should be

used. The domain `SquareMatrix(n,R)` gives the ring of n by n square matrices over R .

Since `SquareMatrix` is not normally exposed at the top level, you must expose it before it can be used.

```
)set expose add constructor SquareMatrix
```

Once `SQMATRIX` has been exposed, values can be created using the `squareMatrix` function.

```
m := squareMatrix [ [1,-%i],[%i,4] ]
+1 - %i+
|      |
+%i  4  +
                                         Type: SquareMatrix(2,Complex Integer)
```

The usual arithmetic operations are available.

```
m*m - m
+ 1 - 4%i+
|      |
+4%i  13 +
                                         Type: SquareMatrix(2,Complex Integer)
```

Square matrices can be used where ring elements are required. For example, here is a matrix with matrix entries.

```
mm := squareMatrix [ [m, 1], [1-m, m**2] ]
++1 - %i+ +1 0+ +
||      | |  |  |
|+%i  4  + +0 1+ |
|      | |  |  |
|+ 0 %i + + 2 - 5%i+|
||      | |  ||  ||
++- %i - 3+ +5%i 17 ++
                                         Type: SquareMatrix(2,SquareMatrix(2,Complex Integer))
```

Or you can construct a polynomial with square matrix coefficients.

```
p := (x + m)**2
  2 + 2 - 2%i+ + 2 - 5%i+
x + |      |x + |      |
    +2%i  8  + +5%i 17  +
                                         Type: Polynomial SquareMatrix(2,Complex Integer)
```

This value can be converted to a square matrix with polynomial coefficients.

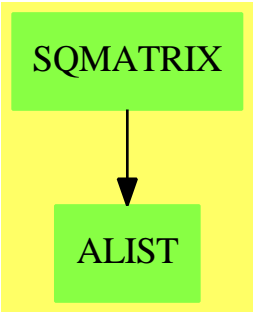
```
p::SquareMatrix(2, ?)
+ 2      +
```

```
|x  + 2x + 2  - 2%i x - 5%i|
|                               |
|               2             |
+2%i x + 5%i  x  + 8x + 17 +
                                Type: SquareMatrix(2,Polynomial Complex Integer)

See Also:
o )help Matrix
o )show SquareMatrix
```



SquareMatrix (SQMATRIX)



See

- ⇒ “IndexedMatrix” (IMATRIX) [10](#) on page [1238](#)
- ⇒ “Matrix” (MATRIX) [14](#) on page [1622](#)
- ⇒ “RectangularMatrix” (RMATRIX) [19](#) on page [2245](#)

Exports:

0	1	antisymmetric?	any?	characteristic
coerce	column	convert	copy	count
D	determinant	diagonal	diagonal?	diagonalMatrix
diagonalProduct	differentiate	elt	empty	empty?
eq?	eval	every?	exquo	hash
inverse	latex	less?	listOfLists	map
map!	matrix	maxColIndex	maxRowIndex	member?
members	minColIndex	minordet	minRowIndex	more?
ncols	nrows	nullSpace	nullity	one?
parts	qelt	rank	recip	reducedSystem
retract	retractIfCan	row	rowEchelon	sample
scalarMatrix	size?	square?	squareMatrix	subtractIfCan
symmetric?	trace	transpose	zero?	#?
?*?	?**?	?+?	?-?	-?
?=?	?^?	?~=?	?/?	

— domain SQMATRIX SquareMatrix —

```

)abbrev domain SQMATRIX SquareMatrix
++ Author: Grabmeier, Gschnitzer, Williamson
++ Date Created: 1987
++ Date Last Updated: July 1990
++ Description:
++ \spadtype{SquareMatrix} is a matrix domain of square matrices, where the
++ number of rows (= number of columns) is a parameter of the type.

SquareMatrix(ndim,R): Exports == Implementation where
  ndim : NonNegativeInteger
  R    : Ring
  Row ==> DirectProduct(ndim,R)
  Col ==> DirectProduct(ndim,R)
  MATLIN ==> MatrixLinearAlgebraFunctions(R,Row,Col,$)

Exports ==> Join(SquareMatrixCategory(ndim,R,Row,Col),_
  CoercibleTo Matrix R) with

  transpose: $ -> $
    ++ \spad{transpose(m)} returns the transpose of the matrix m.
  squareMatrix: Matrix R -> $
    ++ \spad{squareMatrix(m)} converts a matrix of type \spadtype{Matrix}
    ++ to a matrix of type \spadtype{SquareMatrix}.
  coerce: $ -> Matrix R
    ++ \spad{coerce(m)} converts a matrix of type \spadtype{SquareMatrix}
    ++ to a matrix of type \spadtype{Matrix}.
-- symdecomp : $ -> Record(sym:$,antisym:$)
-- ++ \spad{symdecomp(m)} decomposes the matrix m as a sum of a symmetric
-- ++ matrix \spad{m1} and an antisymmetric matrix \spad{m2}. The object
-- ++ returned is the Record \spad{[m1,m2]}

```

```

-- if R has commutative("*") then
--   minorsVect: -> Vector(Union(R,"uncomputed")) --range: 1..2**n-1
--   ++ \spad{minorsVect(m)} returns a vector of the minors of the matrix m
if R has commutative("*") then central
  ++ the elements of the Ring R, viewed as diagonal matrices, commute
  ++ with all matrices and, indeed, are the only matrices which commute
  ++ with all matrices.
if R has commutative("*") and R has unitsKnown then unitsKnown
  ++ the invertible matrices are simply the matrices whose determinants
  ++ are units in the Ring R.
if R has ConvertibleTo InputForm then ConvertibleTo InputForm

Implementation ==> Matrix R add
  minr ==> minRowIndex
  maxr ==> maxRowIndex
  minc ==> minColIndex
  maxc ==> maxColIndex
  mini ==> minIndex
  maxi ==> maxIndex

  ZERO := scalarMatrix 0
  0     == ZERO
  ONE  := scalarMatrix 1
  1     == ONE

  characteristic() == characteristic()$R

matrix(l: List List R) ==
  -- error check: this is a top level function
  #l ^= ndim => error "matrix: wrong number of rows"
  for ll in l repeat
    #ll ^= ndim => error "matrix: wrong number of columns"
  ans : Matrix R := new(ndim,ndim,0)
  for i in minr(ans)..maxr(ans) for ll in l repeat
    for j in minc(ans)..maxc(ans) for r in ll repeat
      qsetelt_!(ans,i,j,r)
  ans pretend $

row(x,i)    == directProduct row(x pretend Matrix(R),i)
column(x,j) == directProduct column(x pretend Matrix(R),j)
coerce(x:$):OutputForm == coerce(x pretend Matrix R)$Matrix(R)

scalarMatrix r == scalarMatrix(ndim,r)$Matrix(R) pretend $

diagonalMatrix l ==
  #l ^= ndim =>
    error "diagonalMatrix: wrong number of entries in list"
  diagonalMatrix(l)$Matrix(R) pretend $

coerce(x:$):Matrix(R) == copy(x pretend Matrix(R))

```



```

squareMatrix x ==
  (nrows(x) ^= ndim) or (ncols(x) ^= ndim) =>
    error "squareMatrix: matrix of bad dimensions"
  copy(x) pretend $

x:$ * v:Col ==
  directProduct((x pretend Matrix(R)) * (v :: Vector(R)))

v:Row * x:$ ==
  directProduct((v :: Vector(R)) * (x pretend Matrix(R)))

x:$ ** n:NonNegativeInteger ==
  ((x pretend Matrix(R)) ** n) pretend $

if R has commutative(" ") then

  determinant x == determinant(x pretend Matrix(R))
  minordet x    == minordet(x pretend Matrix(R))

if R has EuclideanDomain then

  rowEchelon x == rowEchelon(x pretend Matrix(R)) pretend $

if R has IntegralDomain then

  rank x      == rank(x pretend Matrix(R))
  nullity x   == nullity(x pretend Matrix(R))
  nullSpace x ==
    [directProduct c for c in nullSpace(x pretend Matrix(R))]

if R has Field then

  dimension() == (m * n) :: CardinalNumber

  inverse x ==
    (u := inverse(x pretend Matrix(R))) case "failed" => "failed"
    (u :: Matrix(R)) pretend $

  x:$ ** n:Integer ==
    ((x pretend Matrix(R)) ** n) pretend $

  recip x == inverse x

if R has ConvertibleTo InputForm then
  convert(x:$):InputForm ==
    convert [convert("squareMatrix"::Symbol)@InputForm,
             convert(x::Matrix(R))] $List(InputForm)

```

— SQMATRIX.dotabb —

```
"SQMATRIX" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SQMATRIX"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"SQMATRIX" -> "ALIST"
```

domain STACK Stack

— Stack.input —

```
)set break resume
)sys rm -f Stack.output
)spool Stack.output
)set message test on
)set message auto off
)clear all

--S 1 of 44
a:Stack INT:= stack [1,2,3,4,5]
--R
--R (1) [1,2,3,4,5]
--R
--R                                          Type: Stack(Integer)
--E 1

--S 2 of 44
pop! a
--R
--R (2) 1
--R
--R                                          Type: PositiveInteger
--E 2

--S 3 of 44
a
--R
--R (3) [2,3,4,5]
--R
--R                                          Type: Stack(Integer)
--E 3

--S 4 of 44
extract! a
--R
```

```

--R (4) 2
--R
--R                                         Type: PositiveInteger
--E 4

--S 5 of 44
a
--R
--R (5) [3,4,5]
--R
--R                                         Type: Stack(Integer)
--E 5

--S 6 of 44
push!(9,a)
--R
--R (6) 9
--R
--R                                         Type: PositiveInteger
--E 6

--S 7 of 44
a
--R
--R (7) [9,3,4,5]
--R
--R                                         Type: Stack(Integer)
--E 7

--S 8 of 44
insert!(8,a)
--R
--R (8) [8,9,3,4,5]
--R
--R                                         Type: Stack(Integer)
--E 8

--S 9 of 44
a
--R
--R (9) [8,9,3,4,5]
--R
--R                                         Type: Stack(Integer)
--E 9

--S 10 of 44
inspect a
--R
--R (10) 8
--R
--R                                         Type: PositiveInteger
--E 10

--S 11 of 44
empty? a
--R
--R (11) false

```



```
--E 18

--S 19 of 44
parts a
--R
--R (19) [8,9,3,4,5]
--R
--R Type: List(Integer)
--E 19

--S 20 of 44
bag([1,2,3,4,5])$Stack(INT)
--R
--R (20) [5,4,3,2,1]
--R
--R Type: Stack(Integer)
--E 20

--S 21 of 44
b:=empty()$(Stack INT)
--R
--R (21) []
--R
--R Type: Stack(Integer)
--E 21

--S 22 of 44
empty? b
--R
--R (22) true
--R
--R Type: Boolean
--E 22

--S 23 of 44
sample()$Stack(INT)
--R
--R (23) []
--R
--R Type: Stack(Integer)
--E 23

--S 24 of 44
c:=copy a
--R
--R (24) [8,9,3,4,5]
--R
--R Type: Stack(Integer)
--E 24

--S 25 of 44
eq?(a,c)
--R
--R (25) false
--R
--R Type: Boolean
--E 25
```

```

--S 26 of 44
eq?(a,a)
--R
--R (26) true
--R
--R Type: Boolean
--E 26

--S 27 of 44
(a=c)@Boolean
--R
--R (27) true
--R
--R Type: Boolean
--E 27

--S 28 of 44
(a=a)@Boolean
--R
--R (28) true
--R
--R Type: Boolean
--E 28

--S 29 of 44
a~c
--R
--R (29) false
--R
--R Type: Boolean
--E 29

--S 30 of 44
any?(x+-(x=4),a)
--R
--R (30) true
--R
--R Type: Boolean
--E 30

--S 31 of 44
any?(x+-(x=11),a)
--R
--R (31) false
--R
--R Type: Boolean
--E 31

--S 32 of 44
every?(x+-(x=11),a)
--R
--R (32) false
--R
--R Type: Boolean
--E 32

```

```

--S 33 of 44
count(4,a)
--R
--R (33) 1
--R
--R                                         Type: PositiveInteger
--E 33

--S 34 of 44
count(x+-(x>2),a)
--R
--R (34) 5
--R
--R                                         Type: PositiveInteger
--E 34

--S 35 of 44
map(x+>x+10,a)
--R
--R (35) [18,19,13,14,15]
--R
--R                                         Type: Stack(Integer)
--E 35

--S 36 of 44
a
--R
--R (36) [8,9,3,4,5]
--R
--R                                         Type: Stack(Integer)
--E 36

--S 37 of 44
map!(x+>x+10,a)
--R
--R (37) [18,19,13,14,15]
--R
--R                                         Type: Stack(Integer)
--E 37

--S 38 of 44
a
--R
--R (38) [18,19,13,14,15]
--R
--R                                         Type: Stack(Integer)
--E 38

--S 39 of 44
members a
--R
--R (39) [18,19,13,14,15]
--R
--R                                         Type: List(Integer)
--E 39

--S 40 of 44

```

```

member?(14,a)
--R
--R (40) true
--R
--R                                          Type: Boolean
--E 40

--S 41 of 44
coerce a
--R
--R
--R (41) [18,19,13,14,15]
--R
--R                                          Type: OutputForm
--E 41

--S 42 of 44
hash a
--R
--R
--R (42) 4999539
--R
--R                                          Type: SingleInteger
--E 42

--S 43 of 44
latex a
--R
--R
--R (43) "\mbox{\bf Unimplemented}"
--R
--R                                          Type: String
--E 43

--S 44 of 44
)show Stack
--R
--R Stack(S: SetCategory) is a domain constructor
--R Abbreviation for Stack is STACK
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for STACK
--R
--R----- Operations -----
--R bag : List(S) -> %                copy : % -> %
--R depth : % -> NonNegativeInteger    empty : () -> %
--R empty? : % -> Boolean              eq? : (%,% ) -> Boolean
--R extract! : % -> S                 insert! : (S,% ) -> %
--R inspect : % -> S                  map : ((S -> S),%) -> %
--R pop! : % -> S                     push! : (S,% ) -> S
--R sample : () -> %                  stack : List(S) -> %
--R top : % -> S
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?? : (%,% ) -> Boolean if S has SETCAT
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate

```



```

--R coerce : % -> OutputForm if S has SETCAT
--R count : (S,%) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R eval : (%,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R hash : % -> SingleInteger if S has SETCAT
--R latex : % -> String if S has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R member? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT
--R members : % -> List(S) if $ has finiteAggregate
--R more? : (%,NonNegativeInteger) -> Boolean
--R parts : % -> List(S) if $ has finiteAggregate
--R size? : (%,NonNegativeInteger) -> Boolean
--R ~=?: (%,%) -> Boolean if S has SETCAT
--R
--E 44
)spool
)lisp (bye)

```

— Stack.help —

=====

Stack examples

=====

A Stack object is represented as a list ordered by last-in, first-out. It operates like a pile of books, where the "next" book is the one on the top of the pile.

Here we create a stack of integers from a list. Notice that the order in the list is the order in the stack.

```

a:Stack INT:= stack [1,2,3,4,5]
[1,2,3,4,5]

```

We can remove the top of the stack using pop!:

```

pop! a
1

```

Notice that the use of pop! is destructive (destructive operations in Axiom usually end with ! to indicate that the underlying data

structure is changed).

```
a
  [2,3,4,5]
```

The `extract!` operation is another name for the `pop!` operation and has the same effect. This operation treats the stack as a `BagAggregate`:

```
extract! a
  2
```

and you can see that it also has destructively modified the stack:

```
a
  [3,4,5]
```

Next we push a new element on top of the stack:

```
push!(9,a)
  9
```

Again, the `push!` operation is destructive so the stack is changed:

```
a
  [9,2,3,4,5]
```

Another name for `push!` is `insert!`, which treats the stack as a `BagAggregate`:

```
insert!(8,a)
  [8,9,3,4,5]
```

and it modifies the stack:

```
a
  [8,9,3,4,5]
```

The `inspect` function returns the top of the stack without modification, viewed as a `BagAggregate`:

```
inspect a
  8
```

The `empty?` operation returns `true` only if there are no element on the stack, otherwise it returns `false`:

```
empty? a
  false
```

The `top` operation returns the top of stack without modification, viewed as a `Stack`:

```
top a
8
```

The depth operation returns the number of elements on the stack:

```
depth a
5
```

which is the same as the # (length) operation:

```
#a
5
```

The less? predicate will compare the stack length to an integer:

```
less?(a,9)
true
```

The more? predicate will compare the stack length to an integer:

```
more?(a,9)
false
```

The size? operation will compare the stack length to an integer:

```
size?(a,#a)
true
```

and since the last computation must always be true we try:

```
size?(a,9)
false
```

The parts function will return the stack as a list of its elements:

```
parts a
[8,9,3,4,5]
```

If we have a BagAggregate of elements we can use it to construct a stack. Notice that the elements are pushed in reverse order:

```
bag([1,2,3,4,5])$Stack(INT)
[5,4,3,2,1]
```

The empty function will construct an empty stack of a given type:

```
b:=empty()$(Stack INT)
[]
```

and the empty? predicate allows us to find out if a stack is empty:

```
empty? b
true
```

The sample function returns a sample, empty stack:

```
sample()$Stack(INT)
[]
```

We can copy a stack and it does not share storage so subsequent modifications of the original stack will not affect the copy:

```
c:=copy a
[8,9,3,4,5]
```

The eq? function is only true if the lists are the same reference, so even though c is a copy of a, they are not the same:

```
eq?(a,c)
false
```

However, a clearly shares a reference with itself:

```
eq?(a,a)
true
```

But we can compare a and c for equality:

```
(a=c)@Boolean
true
```

and clearly a is equal to itself:

```
(a=a)@Boolean
true
```

and since a and c are equal, they are clearly NOT not-equal:

```
a~=c
false
```

We can use the any? function to see if a predicate is true for any element:

```
any?(x+>(x=4),a)
true
```

or false for every element:

```
any?(x+>(x=11),a)
```

```
false
```

We can use the `every?` function to check every element satisfies a predicate:

```
every?(x+>(x=11),a)
false
```

We can count the elements that are equal to an argument of this type:

```
count(4,a)
1
```

or we can count against a boolean function:

```
count(x+>(x>2),a)
5
```

You can also map a function over every element, returning a new stack:

```
map(x+>x+10,a)
[18,19,13,14,15]
```

Notice that the original stack is unchanged:

```
a
[8,9,3,4,5]
```

You can use `map!` to map a function over every element and change the original stack since `map!` is destructive:

```
map!(x+>x+10,a)
[18,19,13,14,15]
```

Notice that the original stack has been changed:

```
a
[18,19,13,14,15]
```

The `member` function can also get the element of the stack as a list:

```
members a
[18,19,13,14,15]
```

and using `member?` we can test if the stack holds a given element:

```
member?(14,a)
true
```

See Also:

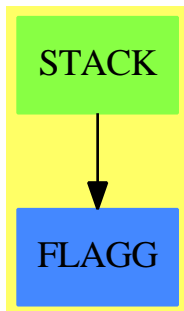
```
o )show Stack
```

```

o )show ArrayStack
o )show Queue
o )show Dequeue
o )show Heap
o )show BagAggregate

```

Stack (STACK)



See

```

⇒ “ArrayStack” (ASTACK) 2 on page 65
⇒ “Queue” (QUEUE) 18 on page 2184
⇒ “Dequeue” (DEQUEUE) 5 on page 512
⇒ “Heap” (HEAP) 9 on page 1134

```

Exports:

any?	bag	coerce	copy	count
depth	empty	empty?	eq?	eval
every?	extract!	hash	insert!	inspect
latex	less?	map	map!	member?
members	more?	parts	pop!	push!
sample	size?	stack	top	#?
?=?	?~=?			

— domain STACK Stack —

```

)abbrev domain STACK Stack
++ Author: Michael Monagan, Stephen Watt, Timothy Daly
++ Date Created: June 86 and July 87
++ Date Last Updated: Feb 09
++ Description:
++ Linked List implementation of a Stack
--% Dequeue and Heap data types

```

```

Stack(S:SetCategory): StackAggregate S with
  stack: List S -> %
  ++ stack([x,y,...,z]) creates a stack with first (top)
  ++ element x, second element y,...,and last element z.
  ++
  ++X a:Stack INT:= stack [1,2,3,4,5]

-- Inherited Signatures repeated for examples documentation

pop_! : % -> S
  ++
  ++X a:Stack INT:= stack [1,2,3,4,5]
  ++X pop! a
  ++X a
extract_! : % -> S
  ++
  ++X a:Stack INT:= stack [1,2,3,4,5]
  ++X extract! a
  ++X a
push_! : (S,%) -> S
  ++
  ++X a:Stack INT:= stack [1,2,3,4,5]
  ++X push!(9,a)
  ++X a
insert_! : (S,%) -> %
  ++
  ++X a:Stack INT:= stack [1,2,3,4,5]
  ++X insert!(8,a)
  ++X a
inspect : % -> S
  ++
  ++X a:Stack INT:= stack [1,2,3,4,5]
  ++X inspect a
top : % -> S
  ++
  ++X a:Stack INT:= stack [1,2,3,4,5]
  ++X top a
depth : % -> NonNegativeInteger
  ++
  ++X a:Stack INT:= stack [1,2,3,4,5]
  ++X depth a
less? : (% ,NonNegativeInteger) -> Boolean
  ++
  ++X a:Stack INT:= stack [1,2,3,4,5]
  ++X less?(a,9)
more? : (% ,NonNegativeInteger) -> Boolean
  ++
  ++X a:Stack INT:= stack [1,2,3,4,5]
  ++X more?(a,9)

```

```

size? : (% , NonNegativeInteger) -> Boolean
++
++X a:Stack INT:= stack [1,2,3,4,5]
++X size?(a,5)
bag : List S -> %
++
++X bag([1,2,3,4,5])$Stack(INT)
empty? : % -> Boolean
++
++X a:Stack INT:= stack [1,2,3,4,5]
++X empty? a
empty : () -> %
++
++X b:=empty()$(Stack INT)
sample : () -> %
++
++X sample()$Stack(INT)
copy : % -> %
++
++X a:Stack INT:= stack [1,2,3,4,5]
++X copy a
eq? : (% , %) -> Boolean
++
++X a:Stack INT:= stack [1,2,3,4,5]
++X b:=copy a
++X eq?(a,b)
map : ((S -> S) , %) -> %
++
++X a:Stack INT:= stack [1,2,3,4,5]
++X map(x+>x+10,a)
++X a
if $ has shallowlyMutable then
  map! : ((S -> S) , %) -> %
  ++
  ++X a:Stack INT:= stack [1,2,3,4,5]
  ++X map!(x+>x+10,a)
  ++X a
if S has SetCategory then
  latex : % -> String
  ++
  ++X a:Stack INT:= stack [1,2,3,4,5]
  ++X latex a
  hash : % -> SingleInteger
  ++
  ++X a:Stack INT:= stack [1,2,3,4,5]
  ++X hash a
  coerce : % -> OutputForm
  ++
  ++X a:Stack INT:= stack [1,2,3,4,5]
  ++X coerce a

```



```

"=: (%,% ) -> Boolean
++
++X a:Stack INT:= stack [1,2,3,4,5]
++X b:Stack INT:= stack [1,2,3,4,5]
++X (a=b)@Boolean
"~=" : (%,% ) -> Boolean
++
++X a:Stack INT:= stack [1,2,3,4,5]
++X b:=copy a
++X (a~b)
if % has finiteAggregate then
every? : ((S -> Boolean),%) -> Boolean
++
++X a:Stack INT:= stack [1,2,3,4,5]
++X every?(x+-(x=4),a)
any? : ((S -> Boolean),%) -> Boolean
++
++X a:Stack INT:= stack [1,2,3,4,5]
++X any?(x+-(x=4),a)
count : ((S -> Boolean),%) -> NonNegativeInteger
++
++X a:Stack INT:= stack [1,2,3,4,5]
++X count(x+-(x>2),a)
_# : % -> NonNegativeInteger
++
++X a:Stack INT:= stack [1,2,3,4,5]
++X #a
parts : % -> List S
++
++X a:Stack INT:= stack [1,2,3,4,5]
++X parts a
members : % -> List S
++
++X a:Stack INT:= stack [1,2,3,4,5]
++X members a
if % has finiteAggregate and S has SetCategory then
member? : (S,%) -> Boolean
++
++X a:Stack INT:= stack [1,2,3,4,5]
++X member?(3,a)
count : (S,%) -> NonNegativeInteger
++
++X a:Stack INT:= stack [1,2,3,4,5]
++X count(4,a)

== add
Rep := Reference List S
s = t == deref s = deref t
coerce(d:%): OutputForm == bracket [e::OutputForm for e in deref d]
copy s == ref copy deref s

```

```

depth s == # deref s
# s == depth s
pop_! (s:%):S ==
  empty? s => error "empty stack"
  e := first deref s
  setref(s,rest deref s)
  e
extract_! (s:%):S == pop_! s
top (s:%):S ==
  empty? s => error "empty stack"
  first deref s
inspect s == top s
push_!(e,s) == (setref(s,cons(e,deref s));e)
insert_!(e:S,s:%):% == (push_!(e,s);s)
empty() == ref nil()$List(S)
empty? s == null deref s
stack s == ref copy s
parts s == copy deref s
map(f,s) == ref map(f,deref s)
map!(f,s) == ref map!(f,deref s)

```

— STACK.dotabb —

```

"STACK" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STACK"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"STACK" -> "FLAGG"

```

domain SD StochasticDifferential

— StochasticDifferential.input —

```

)set break resume
)sys rm -f StochasticDifferential.output
)spool StochasticDifferential.output
)set message test on
)set message auto off
)clear all

--S 1 of 12
dt := introduce!(t,dt)

```

```

--R
--R
--R (1) dt
--R                                         Type: Union(BasicStochasticDifferential,...)
--E 1

--S 2 of 12
dX := introduce!(X,dX)
--R
--R
--R (2) dX
--R                                         Type: Union(BasicStochasticDifferential,...)
--E 2

--S 3 of 12
dY := introduce!(Y,dY)
--R
--R
--R (3) dY
--R                                         Type: Union(BasicStochasticDifferential,...)
--E 3

--S 4 of 12
copyBSD()
--R
--R
--R (4) [dX,dY,dt]
--R                                         Type: List(BasicStochasticDifferential)
--E 4

--S 5 of 12
copyIto()
--R
--R
--R (5) table(t= dt,Y= dY,X= dX)
--R                                         Type: Table(Symbol,BasicStochasticDifferential)
--E 5

--S 6 of 12
copyQuadVar()          -- display of multiplication table
--R
--R
--R (6) table()
--R Type: Table(StochasticDifferential(Integer),StochasticDifferential(Integer))
--E 6

--S 7 of 12
statusIto()
--R
--R

```

```

--R      +B S D      : dX dY dt+
--R      |
--R      |drift      : ? ? ? |
--R      |
--R      | *
--R      (7) |
--R      | dX      : ? ? ? |
--R      |
--R      | dY      : ? ? ? |
--R      |
--R      + dt      : ? ? ? +
--R
--R
--E 7

```

Type: OutputForm

```

--S 8 of 12
copyDrift()      -- display of drift list
--R
--R
--R      (8) table()
--R Type: Table(StochasticDifferential(Integer),StochasticDifferential(Integer))
--E 8

```

```

--S 9 of 12
nbsd := #copyBSD()
--R
--R
--R      (9) 3
--R
--R
--E 9

```

Type: PositiveInteger

```

--S 10 of 12
ItoMultArray:ARRAY2(SD INT) :=new(nbsd,nbsd,0$SD(INT))
--R
--R
--R      +0 0 0+
--R      |
--R      (10) |0 0 0|
--R      |
--R      +0 0 0+
--R
--R
--E 10

```

Type: TwoDimensionalArray(StochasticDifferential(Integer))

```

--S 11 of 12
ItoMultArray
--R
--R
--R      +0 0 0+
--R      |
--R      (11) |0 0 0|
--R      |

```

```

--R      +0  0  0+
--R                                     Type: TwoDimensionalArray(StochasticDifferential(Integer))
--E 11

```

```

--S 12 of 12
statusIto()
--R
--R
--R      +B S D   :  dX  dY  dt+
--R      |
--R      |drift   :  ?   ?   ? |
--R      |
--R      | *
--R      (12) |
--R      | dX     :  ?   ?   ? |
--R      |
--R      | dY     :  ?   ?   ? |
--R      |
--R      + dt     :  ?   ?   ? +
--R
--R                                     Type: OutputForm
--E 12

```

```

)spool
)lisp (bye)

```

— StochasticDifferential.help —

=====

StochasticDifferential examples

=====

A basic implementation of StochasticDifferential(R) using the associated domain BasicStochasticDifferential in the underlying representation as sparse multivariate polynomials. The domain is a module over Expression(R), and is a ring without identity (AXIOM term is "Rng"). Note that separate instances, for example using R=Integer and R=Float, have different hidden structure (multiplication and drift tables).

```

dt := introduce!(t,dt)
dt

```

```

dX := introduce!(X,dX)
dX

```

```

dY := introduce!(Y,dY)
dY

```

```

copyBSD()
  [dX,dY,dt]

copyIto()
  table(t= dt,Y= dY,X= dX)

copyQuadVar()          -- display of multiplication table
  table()

statusIto()
  +B S D   :  dX  dY  dt+
  |
  |drift   :  ?   ?   ? |
  |
  |  *
  |
  | dX     :  ?   ?   ? |
  |
  | dY     :  ?   ?   ? |
  |
  + dt     :  ?   ?   ? +

copyDrift()            -- display of drift list
  table()

nbsd := #copyBSD()
3

ItoMultArray:ARRAY2(SD INT) :=new(nbsd,nbsd,0$SD(INT))
+0  0  0+
|
|0  0  0|
|
+0  0  0+

ItoMultArray
+0  0  0+
|
|0  0  0|
|
+0  0  0+

statusIto()
  +B S D   :  dX  dY  dt+
  |
  |drift   :  ?   ?   ? |
  |
  |  *
  |

```

```

| dX      : ? ? ? |
|         |         |
| dY      : ? ? ? |
|         |         |
+ dt      : ? ? ? +

```

See Also:

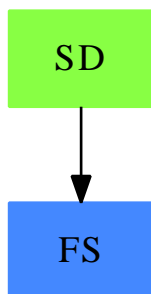
```

o )help BasicStochasticDifferential
o )show StochasticDifferential

```

—————

StochasticDifferential (SD)



See

← “BasicStochasticDifferential” (BSD) [3](#) on page [285](#)

Exports:

0	coerce	copyDrift	copyQuadVar
drift	hash	latex	sample
statusIto	uncorrelated?	zero?	alterDrift!
alterQuadVar!	coefficient	coerce	equation
freeOf?	listSD	retract	retractIfCan
subtractIfCan	?*?	?**?	?+?
?-?	-?	?/?	?=?
?^?	?~=?		

— domain SD StochasticDifferential —

```

)abbrev domain SD StochasticDifferential
++ Author: Wilfrid S. Kendall
++ Last Last Updated: July 26, 1999
++ Related Domains: BasicStochasticDifferential

```

```

++ References: Ito (1975), Kendall (1991a,b; 1993a,b).
++ Description:
++ A basic implementation of StochasticDifferential(R) using the
++ associated domain BasicStochasticDifferential in the underlying
++ representation as sparse multivariate polynomials. The domain is
++ a module over Expression(R), and is a ring without identity
++ (AXIOM term is "Rng"). Note that separate instances, for example
++ using R=Integer and R=Float, have different hidden structure
++ (multiplication and drift tables).

```

```

StochasticDifferential(R:Join(OrderedSet, IntegralDomain)):

```

```

  Category == Implementation where
    ER ==> Expression(R)
    PR ==> Polynomial(R)
    FR ==> Fraction(PR)
    BSD ==> BasicStochasticDifferential
    PI ==> PositiveInteger
    NNI ==> NonNegativeInteger
    OF ==> OutputForm
    Category ==> Join(Rng, Module(ER)) with

```

```

  RetractableTo(BSD)

```

```

alterQuadVar!:(BSD,BSD,%) -> Union(%, "failed")
++ alterQuadVar! adds multiplication formula for a
++ pair of stochastic differentials to a private table.
++ Failure occurs if
++ (a) either of first or second arguments is not basic
++ (b) third argument is not exactly of first degree

```

```

alterDrift!:(BSD,%) -> Union(%, "failed")
++ alterDrift! adds drift formula for a
++ stochastic differential to a private table.
++ Failure occurs if
++ (a) first arguments is not basic
++ (b) second argument is not exactly of first degree

```

```

drift:% -> %
++ drift(dx) returns the drift of \axiom{dx}

```

```

freeOf?:(%,BSD) -> Boolean
++ freeOf?(sd,dX) checks whether \axiom{dX} occurs in
++ \axiom{sd} as a module element

```

```

coefficient:(%,BSD) -> ER
++ coefficient(sd,dX) returns the coefficient of \axiom{dX}
++ in the stochastic differential \axiom{sd}

```

```

listSD:(%) -> List BSD
++ listSD(dx) returns a list of all \axiom{BSD} involved

```



```

++ in the generation of \axiom{dx} as a module element

equation:(%,R) -> Union(Equation %,"failed")
++ equation(dx,0) allows RHS of Equation % to be zero
equation:(R,%) -> Union(Equation %,"failed")
++ equation(0,dx) allows LHS of Equation % to be zero

copyDrift:() -> Table(%,%)
++ copyDrift returns private table of drifts
++ of basic stochastic differentials for inspection

copyQuadVar:() -> Table(%,%)
++ copyQuadVar returns private multiplication table
++ of basic stochastic differentials for inspection

"/"      : (%, ER) -> %
++ dx/y divides the stochastic differential dx
++ by the previsible function y.

"*)"      : (%, PI) -> %
++ dx**n is dx multiplied by itself n times.

"^"      : (%, PI) -> %
++ dx^n is dx multiplied by itself n times.

statusIto:() -> OF
++ statusIto() displays the current state of \axiom{setBSD},
++ \axiom{tableDrift}, and \axiom{tableQuadVar}. Question
++ marks are printed instead of undefined entries
++
++X dt:=introduce!(t,dt)
++X dX:=introduce!(X,dX)
++X dY:=introduce!(Y,dY)
++X copyBSD()
++X copyIto()
++X copyhQuadVar()
++X statusIto()

uncorrelated?: (%,%) -> Boolean
++ uncorrelated?(dx,dy) checks whether its two arguments
++ have zero quadratic co-variation.
uncorrelated?: (List %,List %) -> Boolean
++ uncorrelated?(l1,l2) checks whether its two arguments
++ are lists of stochastic differentials of zero inter-list
++ quadratic co-variation.
uncorrelated?: (List List %) -> Boolean
++ uncorrelated?(ll) checks whether its argument is a list
++ of lists of stochastic differentials of zero inter-list
++ quadratic co-variation.

```

```

Implementation ==> SparseMultivariatePolynomial(ER,BSD) add
Rep:=SparseMultivariatePolynomial(ER,BSD)

(v:% / s:ER):% == inv(s) * v

tableQuadVar:Table(%,% ) := table()
tableDrift:Table(%,% ) := table()

alterQuadVar!(da:BSD,db:BSD,dXdY:%):Union(%,"failed") ==
-- next two lines for security only!
1 < totalDegree(dXdY) => "failed"
0 ~= coefficient(dXdY,degree(1)$Rep) => "failed"
not(0::% = (dXdY*dXdY)::%) => "failed"
setelt(tableQuadVar,((da::Rep)*(db::Rep))$Rep,dXdY)$Table(%,% )
-- We have to take care here to avoid a bad
-- recursion on \axiom{*: (%,% )->%}

alterDrift!(da:BSD,dx:%):Union(%,"failed") ==
1 < totalDegree(dx) => "failed"
0 ~= coefficient(dx,degree(1)$Rep) => "failed"
not(0::% = (dx*dx)::%) => "failed"
setelt(tableDrift,da::Rep,dx)$Table(%,% )

multSDOrError(dm:%):% ==
c := leadingCoefficient dm
(dmm := search(dm/c,tableQuadVar))
case "failed" =>
print hconcat(message("ERROR IN ")$OF,(dm/c)::OF)
error "Above product of sd's is not defined"
c*dmm

(dx:% * dy:% ) : % ==
1 < totalDegree(dx) =>
print hconcat(message("ERROR IN ")$OF,dx::OF)
error "bad sd in lhs of sd product"
1 < totalDegree(dy) =>
print hconcat(message("ERROR IN ")$OF,dy::OF)
error "bad sd in rhs of sd product"
reduce("+",map(multSDOrError,monomials((dx*dy)$Rep)),0)
-- We have to take care here to avoid a bad
-- recursion on \axiom{*: (%,% )->%}

(dx:% ** n:PI) : % ==
n = 1 => dx
n = 2 => dx*dx
n > 2 => 0::%

(dx:% ^ n:PI) : % == dx**n

driftSDOrError(dm:%):% ==

```

```

c := leadingCoefficient dm
(dmm := search(dm/c,tableDrift))
case "failed" =>
  print hconcat(message("ERROR IN ")$OF,(dm/c)::OF)
  error "drift of sd is not defined"
c*dmm

drift(dx:~):~ ==
  reduce("+",map(driftSDOrError,monomials(dx)),0)

freeOf?(sd,dX) == (0 = coefficient(sd,dX,1))

coefficient(sd:~,dX:BSD):ER ==
  retract(coefficient(sd,dX,1))@ER

listSD(sd) ==
  [retract(dX)@BSD for dX in primitiveMonomials(sd)]

equation(dx:~,zero:R):Union(Equation ~,"failed") ==
  not(0 = zero) => "failed"
equation(dx,0::~~)
equation(zero:R,dx:~):Union(Equation ~,"failed") ==
  not(0 = zero) => "failed"
equation(0::~~,dx)

copyDrift() == tableDrift
copyQuadVar() == tableQuadVar

xDrift(dx:BSD):OF ==
  (xdx := search(dx::Rep,tableDrift)) case "failed" => "?":OF
  xdx::OF

xQV(dx:BSD,dy:BSD):OF ==
  (xdxdy := search((dx::~~ * dy::~~)$Rep,tableQuadVar))
  case "failed" => "?":Symbol::OF
  xdxdy::OF

statusIto():OF ==
  bsd := copyBSD()$BSD
  bsdo := [dx::OF for dx in bsd]
  blank:= "":Symbol::OF
  colon:= "":Symbol::OF
  bsdh := "B S D ":Symbol::OF
  dfth := "drift ":Symbol::OF
  qvh := " "*:Symbol::OF
  head := append([bsdh,colon],bsdo)
  drift:= append([dfth,colon],[xDrift dx for dx in bsd])
  space:= append([qvh,blank],[blank for dx in bsd])
  qv := [append([dy::OF,colon],[xQV(dx,dy) for dx in bsd])
        for dy in bsd]

```

```

matrix(append([head,drift,space],qv))$OF

uncorrelated?(dx:%,dy:%): Boolean == (0::% = dx*dy)

uncorrelated?(l1:List %,l2:List %): Boolean ==
  reduce("and", [
    reduce("and",[uncorrelated?(dx,dy) for dy in l2],true)
    for dx in l1 ],true)

uncorrelated1?(l1:List %,l1:List List %): Boolean ==
  reduce("and",[uncorrelated?(l1,l2) for l2 in l1],true)

uncorrelated?(l1:List List %): Boolean ==
  (0$Integer = # l1) => true
  (1 = # l1) => true
  uncorrelated1?(first l1,rest l1) and uncorrelated?(rest l1)

```

— SD.dotabb —

```

"SD" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SD"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"SD" -> "FS"

```

domain STREAM Stream

— Stream.input —

```

)set break resume
)sys rm -f Stream.output
)spool Stream.output
)set message test on
)set message auto off
)clear all
--S 1 of 12
ints := [i for i in 0..]
--R
--R
--R (1) [0,1,2,3,4,5,6,7,8,9,...]
--R
--R                                         Type: Stream(NonNegativeInteger)
--E 1

```

```

--S 2 of 12
f : List INT -> List INT
--R
--R
--R                                          Type: Void
--E 2

--S 3 of 12
f x == [x.1 + x.2, x.1]
--R
--R
--R                                          Type: Void
--E 3

--S 4 of 12
fibs := [i.2 for i in [generate(f,[1,1])]]
--R
--R   Compiling function f with type List(Integer) -> List(Integer)
--R
--R   (4)  [1,1,2,3,5,8,13,21,34,55,...]
--R
--R                                          Type: Stream(Integer)
--E 4

--S 5 of 12
[i for i in ints | odd? i]
--R
--R
--R   (5)  [1,3,5,7,9,11,13,15,17,19,...]
--R
--R                                          Type: Stream(NonNegativeInteger)
--E 5

--S 6 of 12
odds := [2*i+1 for i in ints]
--R
--R
--R   (6)  [1,3,5,7,9,11,13,15,17,19,...]
--R
--R                                          Type: Stream(NonNegativeInteger)
--E 6

--S 7 of 12
scan(0,+,odds)
--R
--R
--R   (7)  [1,4,9,16,25,36,49,64,81,100,...]
--R
--R                                          Type: Stream(NonNegativeInteger)
--E 7

--S 8 of 12
[i*j for i in ints for j in odds]
--R
--R
--R   (8)  [0,3,10,21,36,55,78,105,136,171,...]

```

```

--R                                                    Type: Stream(NonNegativeInteger)
--E 8

--S 9 of 12
map(*,ints,odds)
--R
--R
--R   (9)  [0,3,10,21,36,55,78,105,136,171,...]
--R                                                    Type: Stream(NonNegativeInteger)
--E 9

--S 10 of 12
first ints
--R
--R
--R   (10)  0
--R                                                    Type: NonNegativeInteger
--E 10

--S 11 of 12
rest ints
--R
--R
--R   (11)  [1,2,3,4,5,6,7,8,9,10,...]
--R                                                    Type: Stream(NonNegativeInteger)
--E 11

--S 12 of 12
fibs 20
--R
--R
--R   (12)  6765
--R                                                    Type: PositiveInteger
--E 12
)spool
)lisp (bye)

```

— Stream.help —

```

=====
Stream examples
=====

```

A Stream object is represented as a list whose last element contains the wherewithal to create the next element, should it ever be required.

Let ints be the infinite stream of non-negative integers.

```
ints := [i for i in 0..]
      [0,1,2,3,4,5,6,7,8,9,...]
                                     Type: Stream NonNegativeInteger
```

By default, ten stream elements are calculated. This number may be changed to something else by the system command
`)set streams calculate`

More generally, you can construct a stream by specifying its initial value and a function which, when given an element, creates the next element.

```
f : List INT -> List INT
                                     Type: Void

f x == [x.1 + x.2, x.1]
                                     Type: Void

fibs := [i.2 for i in [generate(f,[1,1])]]
      [1,1,2,3,5,8,13,21,34,55,...]
                                     Type: Stream Integer
```

You can create the stream of odd non-negative integers by either filtering them from the integers, or by evaluating an expression for each integer.

```
[i for i in ints | odd? i]
      [1,3,5,7,9,11,13,15,17,19,...]
                                     Type: Stream NonNegativeInteger

odds := [2*i+1 for i in ints]
      [1,3,5,7,9,11,13,15,17,19,...]
                                     Type: Stream NonNegativeInteger
```

You can accumulate the initial segments of a stream using the scan operation.

```
scan(0,+,odds)
      [1,4,9,16,25,36,49,64,81,100,...]
                                     Type: Stream NonNegativeInteger
```

The corresponding elements of two or more streams can be combined in this way.

```
[i*j for i in ints for j in odds]
      [0,3,10,21,36,55,78,105,136,171,...]
                                     Type: Stream NonNegativeInteger

map(*,ints,odds)
      [0,3,10,21,36,55,78,105,136,171,...]
                                     Type: Stream NonNegativeInteger
```

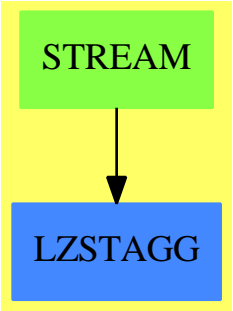
Many operations similar to those applicable to lists are available for

```
streams.  
  
  first ints  
    0  
                                     Type: NonNegativeInteger  
  
  rest ints  
    [1,2,3,4,5,6,7,8,9,10,...]  
                                     Type: Stream NonNegativeInteger  
  
  fibs 20  
    6765  
                                     Type: PositiveInteger
```

See Also:
o)help StreamFunctions1
o)help StreamFunctions2
o)help StreamFunctions3
o)show Stream



Stream (STREAM)



Exports:

any?	child?	children
coerce	complete	concat
concat!	cons	construct
convert	copy	count
cycleEntry	cycleLength	cycleSplit!
cycleTail	cyclic?	delay
delete	distance	elt
empty	empty?	entries
entry?	eq?	eval
eval	eval	eval
every?	explicitEntries?	explicitlyEmpty?
explicitlyFinite?	extend	fill!
filterUntil	filterWhile	find
findCycle	first	frst
generate	hash	index?
indices	insert	insert
last	latex	lazy?
lazyEvaluate	leaf?	leaves
less?	map	map!
maxIndex	member?	members
minIndex	more?	new
nodes	node?	numberOfComputedEntries
output	parts	possiblyInfinite?
qelt	qsetelt!	reduce
remove	removeDuplicates	repeating
repeating?	rest	rst
sample	second	select
setchildren!	setelt	setfirst!
setlast!	setrest!	setvalue!
showAll?	showAllElements	size?
split!	swap!	tail
third	value	#?
?=?	?~=?	?..?
?.last	?.rest	?.first
?.value		

— domain **STREAM** Stream —

```

)abbrev domain STREAM Stream
++ Implementation of streams via lazy evaluation
++ Authors: Burge, Watt; updated by Clifton J. Williamson
++ Date Created: July 1986
++ Date Last Updated: 30 March 1990
++ Description:
++ A stream is an implementation of an infinite sequence using
++ a list of terms that have been computed and a function closure

```

```

++ to compute additional terms when needed.

Stream(S): Exports == Implementation where
-- problems:
-- 1) dealing with functions which basically want a finite structure
-- 2) 'map' doesn't deal with cycles very well

S : Type
B  ==> Boolean
OUT ==> OutputForm
I  ==> Integer
L  ==> List
NNI ==> NonNegativeInteger
U  ==> UniversalSegment I

Exports ==> LazyStreamAggregate(S) with
  shallowlyMutable
  ++ one may destructively alter a stream by assigning new
  ++ values to its entries.

coerce: L S -> %
  ++ coerce(l) converts a list l to a stream.
  ++
  ++X m:=[1,2,3,4,5,6,7,8,9,10,11,12]
  ++X coerce(m)@Stream(Integer)
  ++X m::Stream(Integer)

repeating: L S -> %
  ++ repeating(l) is a repeating stream whose period is the list l.
  ++
  ++X m:=repeating([-1,0,1,2,3])

if S has SetCategory then
  repeating?: (L S,% ) -> B
  ++ repeating?(l,s) returns true if a stream s is periodic
  ++ with period l, and false otherwise.
  ++
  ++X m:=[1,2,3]
  ++X n:=repeating(m)
  ++X repeating?(m,n)

findCycle: (NNI,% ) -> Record(cycle?: B, prefix: NNI, period: NNI)
  ++ findCycle(n,st) determines if st is periodic within n.
  ++
  ++X m:=[1,2,3]
  ++X n:=repeating(m)
  ++X findCycle(3,n)
  ++X findCycle(2,n)

delay: (( ) -> % ) -> %

```

```

++ delay(f) creates a stream with a lazy evaluation defined by
++ function f.
++ Caution: This function can only be called in compiled code.
cons: (S,%) -> %
++ cons(a,s) returns a stream whose \spad{first} is \spad{a}
++ and whose \spad{rest} is s.
++ Note: \spad{cons(a,s) = concat(a,s)}.
++
++X m:=[1,2,3]
++X n:=repeating(m)
++X cons(4,n)

if S has SetCategory then
output: (I, %) -> Void
++ output(n,st) computes and displays the first n entries
++ of st.
++
++X m:=[1,2,3]
++X n:=repeating(m)
++X output(5,n)

showAllElements: % -> OUT
++ showAllElements(s) creates an output form which displays all
++ computed elements.
++
++X m:=[1,2,3,4,5,6,7,8,9,10,11,12]
++X n:=m::Stream(PositiveInteger)
++X showAllElements n

showAll?: () -> B
++ showAll?() returns true if all computed entries of streams
++ will be displayed.
--!! this should be a function of one argument
setrest_!: (%I,%) -> %
++ setrest!(x,n,y) sets rest(x,n) to y. The function will expand
++ cycles if necessary.
++
++X p:=[i for i in 1..]
++X q:=[i for i in 9..]
++X setrest!(p,4,q)
++X p

generate: ((() -> S) -> %
++ generate(f) creates an infinite stream all of whose elements are
++ equal to \spad{f()}.
++ Note: \spad{generate(f) = [f(),f(),f(),...]}
++
++X f():Integer == 1
++X generate(f)

```

```

generate: (S -> S,S) -> %
++ generate(f,x) creates an infinite stream whose first element is
++ x and whose nth element (\spad{n > 1}) is f applied to the previous
++ element. Note: \spad{generate(f,x) = [x,f(x),f(f(x)),...]}
++
++X f(x:Integer):Integer == x+10
++X generate(f,10)

filterWhile: (S -> Boolean,%) -> %
++ filterWhile(p,s) returns \spad{[x0,x1,...,x(n-1)]} where
++ \spad{s = [x0,x1,x2,...]} and
++ n is the smallest index such that \spad{p(xn) = false}.
++
++X m:=[i for i in 1..]
++X f(x:PositiveInteger):Boolean == x < 5
++X filterWhile(f,m)

filterUntil: (S -> Boolean,%) -> %
++ filterUntil(p,s) returns \spad{[x0,x1,...,x(n)]} where
++ \spad{s = [x0,x1,x2,...]} and
++ n is the smallest index such that \spad{p(xn) = true}.
++
++X m:=[i for i in 1..]
++X f(x:PositiveInteger):Boolean == x < 5
++X filterUntil(f,m)

-- if S has SetCategory then
--   map: ((S,S) -> S,%,%,S) -> %
--   ++ map(f,x,y,a) is equivalent to map(f,x,y)
--   ++ If z = map(f,x,y,a), then z = map(f,x,y) except if
--   ++ x.n = a and rest(rest(x,n)) = rest(x,n) in which case
--   ++ rest(z,n) = rest(y,n) or if y.m = a and rest(rest(y,m)) =
--   ++ rest(y,m) in which case rest(z,n) = rest(x,n).
--   ++ Think of the case where f(xi,yi) = xi + yi and a = 0.

Implementation ==> add
MIN ==> 1 -- minimal stream index; see also the defaults in LZSTAGG
x:%

import CyclicStreamTools(S,%)

--% representation

-- This description of the rep is not quite true.
-- The Rep is a pair of one of three forms:
--   [value: S,          rest: %]
--   [nullstream: Magic, NIL  ]
--   [nonnullstream: Magic, fun: () -> %]
-- Could use a record of unions if we could guarantee no tags.

```

```

NullStream:    S := _$NullStream$Lisp    pretend S
NonNullStream: S := _$NonNullStream$Lisp pretend S

Rep := Record(firstElt: S, restOfStream: %)

explicitlyEmpty? x == EQ(first x, NullStream)$Lisp
lazy? x           == EQ(first x, NonNullStream)$Lisp

--% signatures of local functions

setfirst_!      : (%,S) -> S
setrst_!        : (%,%) -> %
setToNil_!      : % -> %
setrestt_!      : (%,I,%) -> %
lazyEval        : % -> %
expand_!        : (%,I) -> %

--% functions to access or change record fields without lazy evaluation

first x == x.firstElt
rst x == x.restOfStream

setfirst_!(x,s) == x.firstElt := s
setrst_!(x,y) == x.restOfStream := y

setToNil_! x ==
-- destructively changes x to a null stream
  setfirst_!(x, NullStream); setrst_!(x, NIL$Lisp)
  x

--% SETCAT functions

if S has SetCategory then

  getm          : (%,L OUT,I) -> L OUT
  streamCountCoerce : % -> OUT
  listm         : (%,L OUT,I) -> L OUT

  getm(x,le,n) ==
    explicitlyEmpty? x => le
    lazy? x =>
      n > 0 =>
        empty? x => le
        getm(rst x, concat(first(x) :: OUT, le), n - 1)
        concat(message("..."), le)
    eq?(x, rst x) => concat(overbar(first(x) :: OUT), le)
    n > 0 => getm(rst x, concat(first(x) :: OUT, le), n - 1)
    concat(message("..."), le)

  streamCountCoerce x ==

```

```

-- this will not necessarily display all stream elements
-- which have been computed
count := _$streamCount$Lisp
-- compute count elements
y := x
for i in 1..count while not empty? y repeat y := rst y
fc := findCycle(count,x)
not fc.cycle? => bracket reverse_! getm(x,empty(),count)
le : L OUT := empty()
for i in 1..fc.prefix repeat
  le := concat(first(x) :: OUT,le)
  x := rest x
pp : OUT :=
  fc.period = 1 => overbar(frst(x) :: OUT)
  pl : L OUT := empty()
  for i in 1..fc.period repeat
    pl := concat(frst(x) :: OUT,pl)
    x := rest x
  overbar commaSeparate reverse_! pl
bracket reverse_! concat(pp,le)

listm(x,le,n) ==
  explicitlyEmpty? x => le
  lazy? x =>
    n > 0 =>
      empty? x => le
      listm(rst x, concat(frst(x) :: OUT,le),n-1)
      concat(message("..."),le)
  listm(rst x,concat(frst(x) :: OUT,le),n-1)

showAllElements x ==
-- this will display all stream elements which have been computed
-- and will display at least n elements with n = streamCount$Lisp
extend(x,_$streamCount$Lisp)
cycElt := cycleElt x
cycElt case "failed" =>
  le := listm(x,empty(),_$streamCount$Lisp)
  bracket reverse_! le
cycEnt := computeCycleEntry(x,cycElt :: %)
le : L OUT := empty()
while not eq?(x,cycEnt) repeat
  le := concat(frst(x) :: OUT,le)
  x := rst x
len := computeCycleLength(cycElt :: %)
pp : OUT :=
  len = 1 => overbar(frst(x) :: OUT)
  pl : L OUT := []
  for i in 1..len repeat
    pl := concat(frst(x) :: OUT,pl)
    x := rst x

```

```

        overbar commaSeparate reverse_! pl
        bracket reverse_! concat(pp,le)

showAll?() ==
    NULL(_$streamsShowAll$Lisp)$Lisp => false
    true

coerce(x):OUT ==
    showAll?() => showAllElements x
    streamCountCoerce x

--% AGG functions

lazyCopy:% -> %
lazyCopy x == delay
    empty? x => empty()
    concat(first x, copy rst x)

copy x ==
    cycElt := cycleElt x
    cycElt case "failed" => lazyCopy x
    ce := cycElt :: %
    len := computeCycleLength(ce)
    e := computeCycleEntry(x,ce)
    d := distance(x,e)
    cycle := complete first(e,len)
    setrst_!(tail cycle,cycle)
    d = 0 => cycle
    head := complete first(x,d::NNI)
    setrst_!(tail head,cycle)
    head

--% CNAGG functions

construct l ==
    -- copied from defaults to avoid loading defaults
    empty? l => empty()
    concat(first l, construct rest l)

--% ELTAGG functions

elt(x:%,n:I) ==
    -- copied from defaults to avoid loading defaults
    n < MIN or empty? x => error "elt: no such element"
    n = MIN => first x
    elt(rst x,n - 1)

seteltt:(%,I,S) -> S
seteltt(x,n,s) ==
    n = MIN => setfirst_!(x,s)

```

```

    seteltt(rst x,n - 1,s)

setelt(x,n:I,s:S) ==
  n < MIN or empty? x => error "setelt: no such element"
  x := expand_!(x,n - MIN + 1)
  seteltt(x,n,s)

--% IXAGG functions

removee: ((S -> Boolean),%) -> %
removee(p,x) == delay
  empty? x => empty()
  p(first x) => remove(p,rst x)
  concat(first x,remove(p,rst x))

remove(p,x) ==
  explicitlyEmpty? x => empty()
  eq?(x,rst x) =>
    p(first x) => empty()
  x
  removee(p,x)

selectt: ((S -> Boolean),%) -> %
selectt(p,x) == delay
  empty? x => empty()
  not p(first x) => select(p, rst x)
  concat(first x,select(p,rst x))

select(p,x) ==
  explicitlyEmpty? x => empty()
  eq?(x,rst x) =>
    p(first x) => x
  empty()
  selectt(p,x)

map(f,x) ==
  map(f,x pretend Stream(S))$StreamFunctions2(S,S) pretend %

map(g,x,y) ==
  xs := x pretend Stream(S); ys := y pretend Stream(S)
  map(g,xs,ys)$StreamFunctions3(S,S,S) pretend %

fill_!(x,s) ==
  setfirst_!(x,s)
  setrst_!(x,x)

map_!(f,x) ==
  -- too many problems with map_! on a lazy stream, so
  -- in this case, an error message is returned
  cyclic? x =>

```



```

    tail := cycleTail x ; y := x
    until y = tail repeat
        setfirst_!(y,f first y)
        y := rst y
    x
    explicitlyFinite? x =>
        y := x
        while not empty? y repeat
            setfirst_!(y,f first y)
            y := rst y
        x
    error "map!: stream with lazy evaluation"

swap_!(x,m,n) ==
    (not index?(m,x)) or (not index?(n,x)) =>
        error "swap!: no such elements"
    x := expand_!(x,max(m,n) - MIN + 1)
    xm := elt(x,m); xn := elt(x,n)
    setelt(x,m,xn); setelt(x,n,xm)
    x

--% LNAGG functions

concat(x:%,s:S) == delay
    empty? x => concat(s,empty())
    concat(first x,concat(rst x,s))

concat(x:%,y:%) == delay
    empty? x => copy y
    concat(first x,concat(rst x, y))

concat l == delay
    empty? l => empty()
    empty?(x := first l) => concat rest l
    concat(first x,concat(rst x,concat rest l))

setelt(x,seg:U,s:S) ==
    low := lo seg
    hasHi seg =>
        high := hi seg
        high < low => s
        (not index?(low,x)) or (not index?(high,x)) =>
            error "setelt: index out of range"
        x := expand_!(x,high - MIN + 1)
        y := rest(x,(low - MIN) :: NNI)
        for i in 0..(high-low) repeat
            setfirst_!(y,s)
            y := rst y
        s
    not index?(low,x) => error "setelt: index out of range"

```

```

    x := rest(x,(low - MIN) :: NNI)
    setrst_!(x,x)
    setfirst_!(x,s)

--% RCAGG functions

empty() == [NullStream, NIL$Lisp]

lazyEval x == (rst(x):(()-> %)) ()

lazyEvaluate x ==
  st := lazyEval x
  setfirst_!(x, first st)
  setrst_!(x,if EQ(rst st,st)$Lisp then x else rst st)
  x

-- empty? is the only function that explicitly causes evaluation
-- of a stream element
empty? x ==
  while lazy? x repeat
    st := lazyEval x
    setfirst_!(x, first st)
    setrst_!(x,if EQ(rst st,st)$Lisp then x else rst st)
  explicitlyEmpty? x

--setvalue(x,s) == setfirst_!(x,s)

--setchildren(x,l) ==
  --empty? l => error "setchildren: empty list of children"
  --not(empty? rest l) => error "setchildren: wrong number of children"
  --setrest_!(x,first l)

--% URAGG functions

first(x,n) == delay
-- former name: take
  n = 0 or empty? x => empty()
  (concat(first x, first(rst x,(n-1) :: NNI)))

concat(s:S,x:%) == [s,x]
cons(s,x) == concat(s,x)

cycleSplit! x ==
  cycElt := cycleElt x
  cycElt case "failed" =>
    error "cycleSplit_!: non-cyclic stream"
  y := computeCycleEntry(x,cycElt :: %)
  eq?(x,y) => (setToNil! x; return y)
  z := rst x
  repeat

```

```

eq?(y,z) => (setrest_!(x,empty()); return y)
x := z ; z := rst z

expand_!(x,n) ==
-- expands cycles (if necessary) so that the first n
-- elements of x will not be part of a cycle
n < 1 => x
y := x
for i in 1..n while not empty? y repeat y := rst y
cycElt := cycleElt x
cycElt case "failed" => x
e := computeCycleEntry(x,cycElt :: %)
d : I := distance(x,e)
d >= n => x
if d = 0 then
-- roll the cycle 1 entry
d := 1
t := cycleTail e
if eq?(t,e) then
t := concat(first t,empty())
e := setrst_!(t,t)
setrst_!(x,e)
else
setrst_!(t,concat(first e,rst e))
e := rst e
nLessD := (n-d) :: NNI
y := complete first(e,nLessD)
e := rest(e,nLessD)
setrst_!(tail y,e)
setrst_!(rest(x,(d-1) :: NNI),y)
x

first x ==
empty? x => error "Can't take the first of an empty stream."
first x

concat_!(x:%,y:%) ==
empty? x => y
setrst_!(tail x,y)

concat_!(x:%,s:S) ==
concat_!(x,concat(s,empty()))

setfirst_!(x,s) == setelt(x,0,s)
setelt(x,"first",s) == setfirst_!(x,s)
setrest_!(x,y) ==
empty? x => error "setrest!: empty stream"
setrst_!(x,y)
setelt(x,"rest",y) == setrest_!(x,y)

```

```

setlast_!(x,s) ==
  empty? x => error "setlast!: empty stream"
  setfirst_!(tail x, s)
setelt(x,"last",s) == setlast_!(x,s)

split_!(x,n) ==
  n < MIN => error "split!: index out of range"
  n = MIN =>
    y : % := empty()
    setfirst_!(y,first x)
    setrst_!(y,rst x)
    setToNil_! x
    y
  x := expand_!(x,n - MIN)
  x := rest(x,(n - MIN - 1) :: NNI)
  y := rest x
  setrst_!(x,empty())
  y

--% STREAM functions

coerce(l: L S) == construct l

repeating l ==
  empty? l =>
    error "Need a non-null list to make a repeating stream."
  x0 : % := x := construct l
  while not empty? rst x repeat x := rst x
  setrst_!(x,x0)

if S has SetCategory then

  repeating?(l, x) ==
    empty? l =>
      error "Need a non-empty? list to make a repeating stream."
    empty? rest l =>
      not empty? x and frst x = first l and x = rst x
    x0 := x
    for s in l repeat
      empty? x or s ^= frst x => return false
    x := rst x
    eq?(x,x0)

findCycle(n, x) ==
  hd := x
  -- Determine whether periodic within n.
  tl := rest(x, n)
  explicitlyEmpty? tl => [false, 0, 0]
  i := 0; while not eq?(x,tl) repeat (x := rst x; i := i + 1)
  i = n => [false, 0, 0]

```

```

-- Find period. Now x=tl, so step over and find it again.
x := rst x; per := 1
while not eq?(x,tl) repeat (x := rst x; per := per + 1)
-- Find non-periodic part.
x := hd; xp := rest(hd, per); npp := 0
while not eq?(x,xp) repeat (x := rst x; xp := rst xp; npp := npp+1)
[true, npp, per]

delay(fs:()->%) == [NonNullStream, fs pretend %]

--      explicitlyEmpty? x == markedNull? x

explicitEntries? x ==
  not explicitlyEmpty? x and not lazy? x

numberOfComputedEntries x ==
  explicitEntries? x => numberOfComputedEntries(rst x) + 1
  0

if S has SetCategory then

  output(n,x) ==
    (not(n>0))or empty? x => void()
    mathPrint(first(x)::OUT)$Lisp
    output(n-1, rst x)

  setrestt_!(x,n,y) ==
    n = 0 => setrst_!(x,y)
    setrestt_!(rst x,n-1,y)

  setrest_!(x,n,y) ==
    n < 0 or empty? x => error "setrest!: no such rest"
    x := expand_!(x,n+1)
    setrestt_!(x,n,y)

  generate f    == delay concat(f(), generate f)
  gen:(S -> S,S) -> %
  gen(f,s) == delay(ss:=f s; concat(ss, gen(f,ss)))
  generate(f,s)==concat(s,gen(f,s))

  concat(x:%,y:%) ==delay
    empty? x => y
    concat(first x,concat(rst x,y))

  swhilee:(S -> Boolean,%) -> %
  swhilee(p,x) == delay
    empty? x      => empty()
    not p(first x) => empty()
    concat(first x,filterWhile(p,rst x))
  filterWhile(p,x)==

```

```

    explicitlyEmpty? x => empty()
    eq?(x,rst x) =>
      p(first x) => x
      empty()
    swhilee(p,x)

suntill: (S -> Boolean,%) -> %
suntill(p,x) == delay
  empty? x => empty()
  p(first x) => concat(first x,empty())
  concat(first x, filterUntil(p, rst x))

filterUntil(p,x)==
  explicitlyEmpty? x => empty()
  eq?(x,rst x) =>
    p(first x) => concat(first x,empty())
    x
  suntill(p,x)

-- if S has SetCategory then
--   mapp: ((S,S) -> S,%,%,S) -> %
--   mapp(f,x,y,a) == delay
--     empty? x or empty? y => empty()
--     concat(f(first x,first y), map(f,rst x,rst y,a))
--   map(f,x,y,a) ==
--     explicitlyEmpty? x => empty()
--     eq?(x,rst x) =>
--       frst x=a => y
--       map(f(first x,#1),y)
--     explicitlyEmpty? y => empty()
--     eq?(y,rst y) =>
--       frst y=a => x
--       p(f(#1,first y),x)
--   mapp(f,x,y,a)

```

— STREAM.dotabb —

```

"STREAM" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STREAM"]
"LZSTAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LZSTAGG"]
"STREAM" -> "LZSTAGG"

```

— String.input —

[illegible]

--E 5

--S 6 of 35

hello.2

--R

--R

--R (6) e

--R

Type: Character

--E 6

--S 7 of 35

hello 2

--R

--R

--R (7) e

--R

Type: Character

--E 7

--S 8 of 35

hello(2)

--R

--R

--R (8) e

--R

Type: Character

--E 8

--S 9 of 35

hullo := copy hello

--R

--R

--R (9) "Hello, I'm AXIOM!"

--R

Type: String

--E 9

--S 10 of 35

hullo.2 := char "u"; [hello, hullo]

--R

--R

--R (10) ["Hello, I'm AXIOM!", "Hullo, I'm AXIOM!"]

--R

Type: List(String)

--E 10

--S 11 of 35

said saw := concat ["alpha", "---", "omega"]

--R

--R

--R (11) "alpha---omega"

--R

Type: String

--E 11


```
--S 12 of 35
concat("hello ", "goodbye")
--R
--R
--R (12) "hello goodbye"
--R
--R                                          Type: String
--E 12

--S 13 of 35
"This " "is " "several " "strings " "concatenated."
--R
--R
--R (13) "This is several strings concatenated."
--R
--R                                          Type: String
--E 13

--S 14 of 35
hello(1..5)
--R
--R
--R (14) "Hello"
--R
--R                                          Type: String
--E 14

--S 15 of 35
hello(8..)
--R
--R
--R (15) "I'm AXIOM!"
--R
--R                                          Type: String
--E 15

--S 16 of 35
split(hello, char " ")
--R
--R
--R (16) ["Hello,", "I'm", "AXIOM!"]
--R
--R                                          Type: List(String)
--E 16

--S 17 of 35
other := complement alphanumeric();
--R
--R
--R                                          Type: CharacterClass
--E 17

--S 18 of 35
split(said saw, other)
--R
--R
```

```
--R (18) ["alpha","omega"]
--R
--E 18
Type: List(String)

--S 19 of 35
trim("## ++ relax ++ ##", char "#")
--R
--R
--R (19) " ++ relax ++ "
--R
--E 19
Type: String

--S 20 of 35
trim("## ++ relax ++ ##", other)
--R
--R
--R (20) "relax"
--R
--E 20
Type: String

--S 21 of 35
leftTrim("## ++ relax ++ ##", other)
--R
--R
--R (21) "relax ++ ##"
--R
--E 21
Type: String

--S 22 of 35
rightTrim("## ++ relax ++ ##", other)
--R
--R
--R (22) "## ++ relax"
--R
--E 22
Type: String

--S 23 of 35
upperCase hello
--R
--R
--R (23) "HELLO, I'M AXIOM!"
--R
--E 23
Type: String

--S 24 of 35
lowerCase hello
--R
--R
--R (24) "hello, i'm axiom!"
--R
Type: String
```

--E 24

--S 25 of 35

prefix?("He", "Hello")

--R

--R

--R (25) true

--R

Type: Boolean

--E 25

--S 26 of 35

prefix?("Her", "Hello")

--R

--R

--R (26) false

--R

Type: Boolean

--E 26

--S 27 of 35

suffix?("", "Hello")

--R

--R

--R (27) true

--R

Type: Boolean

--E 27

--S 28 of 35

suffix?("L0", "Hello")

--R

--R

--R (28) false

--R

Type: Boolean

--E 28

--S 29 of 35

substring?("ll", "Hello", 3)

--R

--R

--R (29) true

--R

Type: Boolean

--E 29

--S 30 of 35

substring?("ll", "Hello", 4)

--R

--R

--R (30) false

--R

Type: Boolean

--E 30

```

--S 31 of 35
n := position("nd", "underground", 1)
--R
--R
--R (31)  2
--R
--R                                         Type: PositiveInteger
--E 31

--S 32 of 35
n := position("nd", "underground", n+1)
--R
--R
--R (32)  10
--R
--R                                         Type: PositiveInteger
--E 32

--S 33 of 35
n := position("nd", "underground", n+1)
--R
--R
--R (33)  0
--R
--R                                         Type: NonNegativeInteger
--E 33

--S 34 of 35
position(char "d", "underground", 1)
--R
--R
--R (34)  3
--R
--R                                         Type: PositiveInteger
--E 34

--S 35 of 35
position(hexDigit(), "underground", 1)
--R
--R
--R (35)  3
--R
--R                                         Type: PositiveInteger
--E 35
)spool
)lisp (bye)

```

— String.help —

```

=====
String examples
=====

```

The type `String` provides character strings. Character strings provide all the operations for a one-dimensional array of characters, plus additional operations for manipulating text.

String values can be created using double quotes.

```
hello := "Hello, I'm AXIOM!"
      "Hello, I'm AXIOM!"
                                     Type: String
```

Note, however, that double quotes and underscores must be preceded by an extra underscore.

```
said := "Jane said, \"Look!\""
      "Jane said, \"Look!\""
                                     Type: String
```

```
saw := "She saw exactly one underscore: \"_\"."
      "She saw exactly one underscore: \"_\"."
                                     Type: String
```

It is also possible to use `new` to create a string of any size filled with a given character. Since there are many new functions it is necessary to indicate the desired type.

```
gasp: String := new(32, char "x")
      "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
                                     Type: String
```

The length of a string is given by `#`.

```
#gasp
32
                                     Type: PositiveInteger
```

Indexing operations allow characters to be extracted or replaced in strings. For any string `s`, indices lie in the range `1..#s`.

```
hello.2
e
                                     Type: Character
```

Indexing is really just the application of a string to a subscript, so any application syntax works.

```
hello 2
e
                                     Type: Character
```

```
hello(2)
e
Type: Character
```

If it is important not to modify a given string, it should be copied before any updating operations are used.

```
hullo := copy hello
"Hello, I'm AXIOM!"
Type: String
```

```
hullo.2 := char "u"; [hello, hullo]
["Hello, I'm AXIOM!", "Hullo, I'm AXIOM!"]
Type: List String
```

Operations are provided to split and join strings. The concat operation allows several strings to be joined together.

```
said saw := concat ["alpha", "---", "omega"]
"alpha---omega"
Type: String
```

There is a version of concat that works with two strings.

```
concat("hello ", "goodbye")
"hello goodbye"
Type: String
```

Juxtaposition can also be used to concatenate strings.

```
"This " "is " "several " "strings " "concatenated."
"This is several strings concatenated."
Type: String
```

Substrings are obtained by giving an index range.

```
hello(1..5)
"Hello"
Type: String
```

```
hello(8..)
"I'm AXIOM!"
Type: String
```

A string can be split into several substrings by giving a separation character or character class.

```
split(hello, char " ")
["Hello", "I'm", "AXIOM!"]
Type: List String
```

```

other := complement alphanumeric();
                                     Type: CharacterClass

split(said saw, other)
["alpha", "omega"]
                                     Type: List String

```

Unwanted characters can be trimmed from the beginning or end of a string using the operations `trim`, `leftTrim` and `rightTrim`.

```

trim("## ++ relax ++ ##", char "#")
" ++ relax ++ "
                                     Type: String

```

Each of these functions takes a string and a second argument to specify the characters to be discarded.

```

trim("## ++ relax ++ ##", other)
"relax"
                                     Type: String

```

The second argument can be given either as a single character or as a character class.

```

leftTrim("## ++ relax ++ ##", other)
"relax ++ ##"
                                     Type: String

rightTrim("## ++ relax ++ ##", other)
"## ++ relax"
                                     Type: String

```

Strings can be changed to upper case or lower case using the operations `upperCase` and `lowerCase`.

```

upperCase hello
"HELLO, I'M AXIOM!"
                                     Type: String

```

The versions with the exclamation mark change the original string, while the others produce a copy.

```

lowerCase hello
"hello, i'm axiom!"
                                     Type: String

```

Some basic string matching is provided. The function `prefix?` tests whether one string is an initial prefix of another.

```

prefix?("He", "Hello")
  true
                                Type: Boolean

```

```

prefix?("Her", "Hello")
  false
                                Type: Boolean

```

A similar function, `suffix?`, tests for suffixes.

```

suffix?("", "Hello")
  true
                                Type: Boolean

```

```

suffix?("LO", "Hello")
  false
                                Type: Boolean

```

The function `substring?` tests for a substring given a starting position.

```

substring?("ll", "Hello", 3)
  true
                                Type: Boolean

```

```

substring?("ll", "Hello", 4)
  false
                                Type: Boolean

```

A number of position functions locate things in strings. If the first argument to `position` is a string, then `position(s,t,i)` finds the location of `s` as a substring of `t` starting the search at position `i`.

```

n := position("nd", "underground", 1)
  2
                                Type: PositiveInteger

```

```

n := position("nd", "underground", n+1)
  10
                                Type: PositiveInteger

```

If `s` is not found, then 0 is returned (`minIndex(s)-1` in `IndexedString`).

```

n := position("nd", "underground", n+1)
  0
                                Type: NonNegativeInteger

```

To search for a specific character or a member of a character class, a different first argument is used.

```

position(char "d", "underground", 1)

```


3

Type: PositiveInteger

position(hexDigit(), "underground", 1)

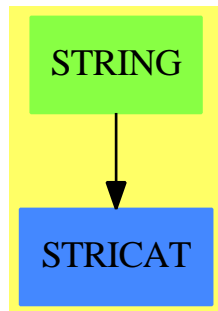
3

Type: PositiveInteger

See Also:

- o)help Character
- o)help CharacterClass
- o)show String

String (STRING)



See

- ⇒ “Character” (CHAR) [4](#) on page [373](#)
- ⇒ “CharacterClass” (CCLASS) [4](#) on page [381](#)
- ⇒ “IndexedString” (ISTRING) [10](#) on page [1249](#)

Exports:

any?	coerce	concat	construct	convert
copy	copyInto!	count	delete	elt
empty	empty?	entries	entry?	eq?
eval	every?	fill!	find	first
hash	index?	indices	insert	latex
leftTrim	less?	lowerCase	lowerCase!	map
map!	match	match?	max	maxIndex
member?	members	merge	min	minIndex
more?	new	OMwrite	parts	position
prefix?	qelt	qsetelt!	reduce	removeDuplicates
replace	reverse	reverse!	rightTrim	sample
select	setelt	size?	sort	sort!
sorted?	split	string	substring?	suffix?
swap!	trim	upperCase	upperCase!	#?
?=?	?.?	?~=?	?<?	?<=?
?>?	?>=?			

— domain STRING String —

```

)abbrev domain STRING String
++ Author: Mark Botch
++ Description:
++ This is the domain of character strings. Strings are 1 based.

String(): StringCategory == IndexedString(1) add
  string n == STRINGIMAGE(n)$Lisp

OMwrite(x: %): String ==
  s: String := ""
  sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
  dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML)
  OMputObject(dev)
  OMputString(dev, x pretend String)
  OMputEndObject(dev)
  OMclose(dev)
  s := OM_-STRINGPTRTOSTRING(sp)$Lisp pretend String
  s

OMwrite(x: %, wholeObj: Boolean): String ==
  s: String := ""
  sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
  dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML)
  if wholeObj then
    OMputObject(dev)
  OMputString(dev, x pretend String)
  if wholeObj then
    OMputEndObject(dev)
  OMclose(dev)

```

```

s := OM_-STRINGPTRTOSTRING(sp)$Lisp pretend String
s

OMwrite(dev: OpenMathDevice, x: %): Void ==
  OMputObject(dev)
  OMputString(dev, x pretend String)
  OMputEndObject(dev)

OMwrite(dev: OpenMathDevice, x: %, wholeObj: Boolean): Void ==
  if wholeObj then
    OMputObject(dev)
  OMputString(dev, x pretend String)
  if wholeObj then
    OMputEndObject(dev)

```

— STRING.dotabb —

```

"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"STRICAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=STRICAT"]
"STRING" -> "STRICAT"

```

domain STRTBL StringTable

— StringTable.input —

```

)set break resume
)sys rm -f StringTable.output
)spool StringTable.output
)set message test on
)set message auto off
)clear all
--S 1 of 3
t: StringTable(Integer) := table()
--R
--R
--R (1) table()
--R
--R                                          Type: StringTable(Integer)
--E 1

--S 2 of 3

```

```

for s in split("My name is Ian Watt.",char " ")
  repeat
    t.s := #s
--R
--R
--R                                          Type: Void
--E 2

--S 3 of 3
for key in keys t repeat output [key, t.key]
--R
--R  ["Watt.",5]
--R  ["Ian",3]
--R  ["is",2]
--R  ["name",4]
--R  ["My",2]
--R
--R                                          Type: Void
--E 3
)spool
)lisp (bye)

```

— StringType.help —

=====

StringTable examples

=====

This domain provides a table type in which the keys are known to be strings so special techniques can be used. Other than performance, the type StringTable(S) should behave exactly the same way as Table(String,S).

This creates a new table whose keys are strings.

```

t: StringTable(Integer) := table()
  table()
                                Type: StringTable Integer

```

The value associated with each string key is the number of characters in the string.

```

for s in split("My name is Ian Watt.",char " ")
  repeat
    t.s := #s
                                Type: Void

```

```

for key in keys t repeat output [key, t.key]
["Watt.",5]
["Ian",3]

```

```
["is",2]
["name",4]
["My",2]
```

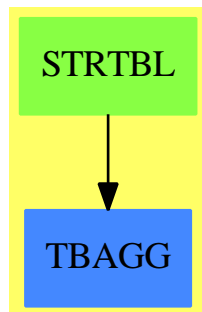
Type: Void

See Also:

- o)help Table
- o)show StringTable

—————→

StringTable (STRTBL)



See

- ⇒ “HashTable” (HASHTBL) [9](#) on page [1120](#)
- ⇒ “InnerTable” (INTABL) [10](#) on page [1335](#)
- ⇒ “Table” (TABLE) [21](#) on page [2663](#)
- ⇒ “EqTable” (EQTBL) [6](#) on page [698](#)
- ⇒ “GeneralSparseTable” (GSTBL) [8](#) on page [1078](#)
- ⇒ “SparseTable” (STBL) [20](#) on page [2450](#)

Exports:

any?	bag	coerce	construct	convert
copy	count	dictionary	elt	empty
empty?	entries	entry?	eq?	eval
every?	extract!	fill!	find	first
hash	index?	indices	insert!	inspect
key?	keys	latex	less?	map
map!	maxIndex	member?	members	minIndex
more?	parts	qelt	qsetelt!	reduce
remove	remove!	removeDuplicates	sample	search
select	select!	setelt	size?	swap!
table	#?	?=?	?~=?	???

— domain STRTBL StringTable —

```
)abbrev domain STRTBL StringTable
++ Author: Stephen M. Watt
++ Date Last Updated: June 21, 1991
++ Description:
++ This domain provides tables where the keys are strings.
++ A specialized hash function for strings is used.
```

```
StringTable(Entry: SetCategory) ==
  HashTable(String, Entry, "CVEC")
```

— STRTBL.dotabb —

```
"STRTBL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRTBL"]
"TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]
"STRTBL" -> "TBAGG"
```

domain SUBSPACE SubSpace

The first argument *n* is the dimension of the subSpace

The SubSpace domain is implemented as a tree. The root of the tree is the only node in which the field *dataList* - which points to a list of points over the ring, *R* - is defined. The children of the root are the top level components of the SubSpace (in 2D, these would be separate curves; in 3D, these would be separate surfaces).

The *pt* field is only defined in the leaves.

By way of example, consider a three dimensional subspace with two components - a three by three grid and a sphere. The internal representation of this subspace is a tree with a depth of three.

The root holds a list of all the points used in the subspace (so, if the grid and the sphere share points, the shared points would not be represented redundantly but would be referenced by index).

The root, in this case, has two children - the first points to the grid component and the second to the sphere component. The grid child has four children of its own - a 3x3 grid has 4 endpoints - and each of these point to a list of four points. To see it another way, the grid (child of the root) holds a list of line components which, when placed one above the next, forms a grid. Each of these line components is a list of points.

Points could be explicitly added to subspaces at any level. A path could be given as an argument to the `addPoint()` function. It is a list of `NonNegativeIntegers` and refers, in order, to the *n*-th child of the current node. For example,

```
addPoint(s,[2,3],p)
```

would add the point *p* to the subspace *s* by going to the second child of the root and then the third child of that node. If the path does extend to the full depth of the tree, nodes are automatically added so that the tree is of constant depth down any path. By not specifying the full path, new components could be added - e.g. for *s* from `SubSpace(3,Float)`

```
addPoint(s,[],p)
```

would create a new child to the root (a new component in *N*-space) and extend a path to a leaf of depth 3 that points to the data held in *p*. The subspace *s* would now have a new component which has one child which, in turn, has one child (the leaf). The new component is then a point.

— SubSpace.input —

```
)set break resume
)sys rm -f SubSpace.output
)spool SubSpace.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SubSpace
--R
--R SubSpace(n: PositiveInteger,R: Ring) is a domain constructor
--R Abbreviation for SubSpace is SUBSPACE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SUBSPACE
--R
--R----- Operations -----
--R ==? : (%,% ) -> Boolean          addPoint2 : (% ,Point(R)) -> %
--R birth : % -> %                  children : % -> List(%)
--R coerce : % -> OutputForm        deepCopy : % -> %
--R extractClosed : % -> Boolean    extractPoint : % -> Point(R)
--R hash : % -> SingleInteger       internal? : % -> Boolean
--R latex : % -> String             leaf? : % -> Boolean
--R level : % -> NonNegativeInteger merge : List(%) -> %
--R merge : (% ,%) -> %             new : () -> %
--R parent : % -> %                 pointData : % -> List(Point(R))
--R root? : % -> Boolean            separate : % -> List(%)
--R shallowCopy : % -> %           subspace : () -> %
--R ~=? : (% ,%) -> Boolean
```

```

--R addPoint : (% ,Point(R)) -> NonNegativeInteger
--R addPoint : (% ,List(NonNegativeInteger),NonNegativeInteger) -> %
--R addPoint : (% ,List(NonNegativeInteger),Point(R)) -> %
--R addPointLast : (% ,%,Point(R),NonNegativeInteger) -> %
--R child : (% ,NonNegativeInteger) -> %
--R closeComponent : (% ,List(NonNegativeInteger),Boolean) -> %
--R defineProperty : (% ,List(NonNegativeInteger),SubSpaceComponentProperty) -> %
--R extractIndex : % -> NonNegativeInteger
--R extractProperty : % -> SubSpaceComponentProperty
--R modifyPoint : (% ,NonNegativeInteger,Point(R)) -> %
--R modifyPoint : (% ,List(NonNegativeInteger),NonNegativeInteger) -> %
--R modifyPoint : (% ,List(NonNegativeInteger),Point(R)) -> %
--R numberOfChildren : % -> NonNegativeInteger
--R traverse : (% ,List(NonNegativeInteger)) -> %
--R
--E 1

```

```

)spool
)lisp (bye)

```

— SubSpace.help —

=====

SubSpace examples

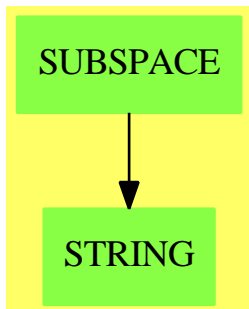
=====

This domain is not documented

See Also:

o)show SubSpace

SubSpace (SUBSPACE)



See

⇒ “Point” (POINT) [17](#) on page [2059](#)

⇒ “SubSpaceComponentProperty” (COMPPROP) [20](#) on page [2624](#)

Exports:

addPoint	addPointLast	addPoint2	birth	child
children	closeComponent	coerce	deepCopy	defineProperty
extractClosed	extractIndex	extractPoint	extractProperty	hash
internal?	latex	leaf?	level	merge
merge	modifyPoint	new	numberOfChildren	parent
pointData	root?	separate	shallowCopy	subspace
traverse	? =?	?=?		

— domain SUBSPACE SubSpace —

```

)abbrev domain SUBSPACE SubSpace
++ Author: Mark Botch
++ Description:
++ This domain is not documented
  
```

```

SubSpace(n:PI,R:Ring) : Exports == Implementation where
  I    ==> Integer
  PI   ==> PositiveInteger
  NNI  ==> NonNegativeInteger
  L    ==> List
  B    ==> Boolean
  POINT ==> Point(R)
  PROP ==> SubSpaceComponentProperty()
  S    ==> String
  O    ==> OutputForm
  empty ==> nil -- macro to ease conversion to new aggcats.spad
  
```

```

Exports ==> SetCategory with
  leaf?      : % -> B
    ++ leaf?(x) is not documented
  root?     : % -> B
  
```

```

    ++ root?(x) is not documented
internal?      : % -> B
    ++ internal?(x) is not documented
new           : () -> %
    ++ new() is not documented
subspace      : () -> %
    ++ subspace() is not documented
birth         : % -> %      -- returns a pointer to the baby
    ++ birth(x) is not documented
child         : (% , NNI) -> %
    ++ child(x,n) is not documented
children      : % -> List %
    ++ children(x) is not documented
numberOfChildren: % -> NNI
    ++ numberOfChildren(x) is not documented
shallowCopy   : % -> %
    ++ shallowCopy(x) is not documented
deepCopy      : % -> %
    ++ deepCopy(x) is not documented
merge         : (% , %) -> %
    ++ merge(s1,s2) the subspaces s1 and s2 into a single subspace.
merge         : List % -> %
    ++ merge(ls) a list of subspaces, ls, into one subspace.
separate      : % -> List %
    ++ separate(s) makes each of the components of the \spadtype{SubSpace},
    ++ s, into a list of separate and distinct subspaces and returns
    ++ the list.
addPoint      : (% , List NNI, POINT) -> %
    ++ addPoint(s,li,p) adds the 4 dimensional point, p, to the 3
    ++ dimensional subspace, s. The list of non negative integers, li,
    ++ dictates the path to follow, or, to look at it another way,
    ++ points to the component in which the point is to be added. It's
    ++ length should range from 0 to \spad{n - 1} where n is the dimension
    ++ of the subspace. If the length is \spad{n - 1}, then a specific
    ++ lowest level component is being referenced. If it is less than
    ++ \spad{n - 1}, then some higher level component (0 indicates top
    ++ level component) is being referenced and a component of that level
    ++ with the desired point is created. The subspace s is returned
    ++ with the additional point.
addPoint2     : (% , POINT) -> %
    ++ addPoint2(s,p) adds the 4 dimensional point, p, to the 3
    ++ dimensional subspace, s.
    ++ The subspace s is returned with the additional point.
addPointLast  : (% , %, POINT, NNI) -> %
    ++ addPointLast(s,s2,li,p) adds the 4 dimensional point, p, to the 3
    ++ dimensional subspace, s. s2 point to the end of the subspace
    ++ s. n is the path in the s2 component.
    ++ The subspace s is returned with the additional point.
modifyPoint   : (% , List NNI, POINT) -> %
    ++ modifyPoint(s,li,p) replaces an existing point in the 3 dimensional

```

```

++ subspace, s, with the 4 dimensional point, p. The list of non
++ negative integers, li, dictates the path to follow, or, to look at
++ it another way, points to the component in which the existing point
++ is to be modified. An error message occurs if s is empty, otherwise
++ the subspace s is returned with the point modification.
addPoint      : (%,List NNI,NNI) -> %
++ addPoint(s,li,i) adds the 4 dimensional point indicated by the
++ index location, i, to the 3 dimensional subspace, s. The list of
++ non negative integers, li, dictates the path to follow, or, to
++ look at it another way, points to the component in which the point
++ is to be added. It's length should range from 0 to \spad{n - 1}
++ where n is the dimension of the subspace. If the length is
++ \spad{n - 1}, then a specific lowest level component is being
++ referenced. If it is less than \spad{n - 1}, then some higher
++ level component (0 indicates top level component) is being
++ referenced and a component of that level with the desired point
++ is created. The subspace s is returned with the additional point.
modifyPoint   : (%,List NNI,NNI) -> %
++ modifyPoint(s,li,i) replaces an existing point in the 3 dimensional
++ subspace, s, with the 4 dimensional point indicated by the index
++ location, i. The list of non negative integers, li, dictates
++ the path to follow, or, to look at it another way, points to the
++ component in which the existing point is to be modified. An error
++ message occurs if s is empty, otherwise the subspace s is returned
++ with the point modification.
addPoint      : (%,POINT) -> NNI
++ addPoint(s,p) adds the point, p, to the 3 dimensional subspace, s,
++ and returns the new total number of points in s.
modifyPoint   : (%,NNI,POINT) -> %
++ modifyPoint(s,ind,p) modifies the point referenced by the index
++ location, ind, by replacing it with the point, p in the 3 dimensional
++ subspace, s. An error message occurs if s is empty, otherwise the
++ subspace s is returned with the point modification.

closeComponent : (%,List NNI,B)      -> %
++ closeComponent(s,li,b) sets the property of the component in the
++ 3 dimensional subspace, s, to be closed if b is true, or open if
++ b is false. The list of non negative integers, li, dictates the
++ path to follow, or, to look at it another way, points to the
++ component whose closed property is to be set. The subspace, s,
++ is returned with the component property modification.
defineProperty : (%,List NNI,PROP) -> %
++ defineProperty(s,li,p) defines the component property in the
++ 3 dimensional subspace, s, to be that of p, where p is of the
++ domain \spadtype{SubSpaceComponentProperty}. The list of non
++ negative integers, li, dictates the path to follow, or, to look
++ at it another way, points to the component whose property is
++ being defined. The subspace, s, is returned with the component
++ property definition.
traverse      : (%,List NNI) -> %

```

```

    ++ traverse(s,li) follows the branch list of the 3 dimensional
    ++ subspace, s, along the path dictated by the list of non negative
    ++ integers, li, which points to the component which has been
    ++ traversed to. The subspace, s, is returned, where s is now
    ++ the subspace pointed to by li.
extractPoint    : % -> POINT
    ++ extractPoint(s) returns the point which is given by the current
    ++ index location into the point data field of the 3 dimensional
    ++ subspace s.
extractIndex    : % -> NNI
    ++ extractIndex(s) returns a non negative integer which is the current
    ++ index of the 3 dimensional subspace s.
extractClosed   : % -> B
    ++ extractClosed(s) returns the \spadtype{Boolean} value of the closed
    ++ property for the indicated 3 dimensional subspace s. If the
    ++ property is closed, \spad{True} is returned, otherwise \spad{False}
    ++ is returned.
extractProperty : % -> PROP
    ++ extractProperty(s) returns the property of domain
    ++ \spadtype{SubSpaceComponentProperty} of the indicated 3 dimensional
    ++ subspace s.
level          : % -> NNI
    ++ level(s) returns a non negative integer which is the current
    ++ level field of the indicated 3 dimensional subspace s.
parent         : % -> %
    ++ parent(s) returns the subspace which is the parent of the indicated
    ++ 3 dimensional subspace s. If s is the top level subspace an error
    ++ message is returned.
pointData      : % -> L POINT
    ++ pointData(s) returns the list of points from the point data field
    ++ of the 3 dimensional subspace s.

Implementation ==> add
import String()

Rep := Record(pt:POINT, index:NNI, property:PROP, _
              childrenField:List %, _
              lastChild: List %, _
              levelField:NNI, _
              pointDataField:L POINT, _
              lastPoint: L POINT, _
              noPoints: NNI, _
              noChildren: NNI, _
              parentField:List %) -- needn't be list but...base case?

TELLWATT : String := "Non-null list: Please inform Stephen Watt"

leaf? space == empty? children space
root? space == (space.levelField = 0$NNI)
internal? space == ^(root? space and leaf? space)

```

```

new() ==
  [point(empty())$POINT,0,new()$PROP,empty(),empty(),0,_
    empty(),empty(),0,0,empty()]
subspace() == new()

birth momma ==
  baby := new()
  baby.levelField := momma.levelField+1
  baby.parentField := [momma]
  if not empty?(lastKid := momma.lastChild) then
    not empty? rest lastKid => error TELLWATT
  if empty? lastKid
  then
    momma.childrenField := [baby]
    momma.lastChild := momma.childrenField
    momma.noChildren := 1
  else
    setrest_!(lastKid,[baby])
    momma.lastChild := rest lastKid
    momma.noChildren := momma.noChildren + 1
  baby

child(space,num) ==
  space.childrenField.num

children space == space.childrenField
numberOfChildren space == space.noChildren

shallowCopy space ==
  node := new()
  node.pt      := space.pt
  node.index   := space.index
  node.property := copy(space.property)
  node.levelField := space.levelField
  node.parentField := nil()
  if root? space then
    node.pointDataField := copy(space.pointDataField)
    node.lastPoint := tail(node.pointDataField)
    node.noPoints := space.noPoints
  node

deepCopy space ==
  node := shallowCopy(space)
  leaf? space => node
  for c in children space repeat
    cc := deepCopy c
    cc.parentField := [node]
    node.childrenField := cons(cc,node.childrenField)
  node.childrenField := reverse_!(node.childrenField)

```

```

node.lastChild := tail node.childrenField
node

merge(s1,s2) ==
----- need to worry about reindexing s2 & parentField
n1 : Rep := deepCopy s1
n2 : Rep := deepCopy s2
n1.childrenField := append(children n1,children n2)
n1

merge listOfSpaces ==
----- need to worry about reindexing & parentField
empty? listOfSpaces => error "empty list passed as argument to merge"
-- notice that the properties of the first subspace on the
-- list are the ones that are inherited...hmmmm...
space := deepCopy first listOfSpaces
for s in rest listOfSpaces repeat
-- because of the initial deepCopy, above, everything is
-- deepCopied to be consistent...more hmmm...
space.childrenField := append(space.childrenField,[deepCopy c for c in s.childrenField])
space

separate space ==
----- need to worry about reindexing & parentField
spaceList := empty()
for s in space.childrenField repeat
spc:=shallowCopy space
spc.childrenField:=[deepCopy s]
spaceList := cons(spc,spaceList)
spaceList

addPoint(space:%,path:List NNI,point:POINT) ==
if not empty?(lastPt := space.lastPoint) then
not empty? rest lastPt => error TELLWATT
if empty? lastPt
then
space.pointDataField := [point]
space.lastPoint := space.pointDataField
else
setrest_!(lastPt,[point])
space.lastPoint := rest lastPt
space.noPoints := space.noPoints + 1
which := space.noPoints
node := space
depth : NNI := 0
for i in path repeat
node := child(node,i)
depth := depth + 1
for more in depth..(n-1) repeat
node := birth node

```

```

node.pt := point      -- will be obsolete field
node.index := which
space

addPoint2(space:%,point:POINT) ==
  if not empty?(lastPt := space.lastPoint) then
    not empty? rest lastPt => error TELLWATT
  if empty? lastPt
  then
    space.pointDataField := [point]
    space.lastPoint := space.pointDataField
  else
    setrest_!(lastPt,[point])
    space.lastPoint := rest lastPt
space.noPoints := space.noPoints + 1
which := space.noPoints
node := space
depth : NNI := 0
node := birth node
first := node
for more in 1..n-1 repeat
  node := birth node
node.pt := point      -- will be obsolete field
node.index := which
first

addPointLast(space:%,node:%, point:POINT, depth:NNI) ==
  if not empty?(lastPt := space.lastPoint) then
    not empty? rest lastPt => error TELLWATT
  if empty? lastPt
  then
    space.pointDataField := [point]
    space.lastPoint := space.pointDataField
  else
    setrest_!(lastPt,[point])
    space.lastPoint := rest lastPt
space.noPoints := space.noPoints + 1
which := space.noPoints
if depth = 2 then node := child(node, 2)
for more in depth..(n-1) repeat
  node := birth node
node.pt := point      -- will be obsolete field
node.index := which
node -- space

addPoint(space:%,path:List NNI,which:NNI) ==
  node := space
  depth : NNI := 0
  for i in path repeat
    node := child(node,i)

```

```

    depth := depth + 1
    for more in depth..(n-1) repeat
        node := birth node
    node.pt := space.pointDataField.which -- will be obsolete field
    node.index := which
    space

addPoint(space:%,point:POINT) ==
    root? space =>
        if not empty?(lastPt := space.lastPoint) then
            not empty? rest lastPt => error TELLWATT
        if empty? lastPt
            then
                space.pointDataField := [point]
                space.lastPoint := space.pointDataField
            else
                setrest_!(lastPt,[point])
                space.lastPoint := rest lastPt
        space.noPoints := space.noPoints + 1
        error "You need to pass a top level SubSpace (level should be zero)"

modifyPoint(space:%,path:List NNI,point:POINT) ==
    if not empty?(lastPt := space.lastPoint) then
        not empty? rest lastPt => error TELLWATT
    if empty? lastPt
        then
            space.pointDataField := [point]
            space.lastPoint := space.pointDataField
        else
            setrest_!(lastPt,[point])
            space.lastPoint := rest lastPt
    space.noPoints := space.noPoints + 1
    which := space.noPoints
    node := space
    for i in path repeat
        node := child(node,i)
    node.pt := point ----- will be obsolete field
    node.index := which
    space

modifyPoint(space:%,path:List NNI,which:NNI) ==
    node := space
    for i in path repeat
        node := child(node,i)
    node.pt := space.pointDataField.which ----- will be obsolete field
    node.index := which
    space

modifyPoint(space:%,which:NNI,point:POINT) ==
    root? space =>

```



```

    space.pointDataField.which := point
    space
    error "You need to pass a top level SubSpace (level should be zero)"

closeComponent(space,path,val) ==
  node := space
  for i in path repeat
    node := child(node,i)
  close(node.property,val)
  space

defineProperty(space,path,prop) ==
  node := space
  for i in path repeat
    node := child(node,i)
  node.property := prop
  space

traverse(space,path) ==
  for i in path repeat space := child(space,i)
  space

extractPoint space ==
  node := space
  while ^root? node repeat node := parent node
  (node.pointDataField).(space.index)
extractIndex space == space.index
extractClosed space == closed? space.property
extractProperty space == space.property

parent space ==
  empty? space.parentField => error "This is a top level SubSpace - it does not have a parent"
  first space.parentField
pointData space == space.pointDataField
level space == space.levelField
s1 = s2 ==
  ----- extra checks for list of point data
  (leaf? s1 and leaf? s2) =>
    (s1.pt = s2.pt) and (s1.property = s2.property) and (s1.levelField = s2.levelField)
  -- note that the ordering of children is important
  #s1.childrenField ^= #s2.childrenField => false
  and/[c1 = c2 for c1 in s1.childrenField for c2 in s2.childrenField]
  and (s1.property = s2.property) and (s1.levelField = s2.levelField)
coerce(space:):0 ==
  hconcat([n:0,"-Space with depth of ":0,
    (n - space.levelField):0," and ":0,(s:=(#space.childrenField)):0,
    (s=1 => " component":0;" components":0)])

```

— SUBSPACE.dotabb —

```
"SUBSPACE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SUBSPACE"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"SUBSPACE" -> "STRING"
```

domain COMPPROP SubSpaceComponentProperty

— SubSpaceComponentProperty.input —

```
)set break resume
)sys rm -f SubSpaceComponentProperty.output
)spool SubSpaceComponentProperty.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SubSpaceComponentProperty
--R SubSpaceComponentProperty is a domain constructor
--R Abbreviation for SubSpaceComponentProperty is COMPPROP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for COMPPROP
--R
--R----- Operations -----
--R ?=? : (%,%) -> Boolean          close : (%,Boolean) -> Boolean
--R closed? : % -> Boolean          coerce : % -> OutputForm
--R copy : % -> %                   hash : % -> SingleInteger
--R latex : % -> String             new : () -> %
--R solid : (%,Boolean) -> Boolean  solid? : % -> Boolean
--R ?~=? : (%,%) -> Boolean
--R
--E 1

)spool
)lisp (bye)
```

— SubSpaceComponentProperty.help —

=====

SubSpaceComponentProperty examples

=====

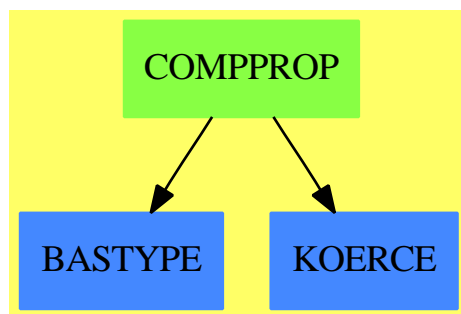
This domain implements some global properties of subspaces.

See Also:

o)show SubSpaceComponentProperty

—————

SubSpaceComponentProperty (COMPPROP)



See

⇒ “Point” (POINT) [17](#) on page [2059](#)

⇒ “SubSpace” (SUBSPACE) [20](#) on page [2614](#)

Exports:

close closed? coerce copy hash
 latex new solid solid? ?~=?
 ?=?

— domain COMPPROP SubSpaceComponentProperty —

```

)abbrev domain COMPPROP SubSpaceComponentProperty
++ Author: Mark Botch
++ Description:
++ This domain implements some global properties of subspaces.
  
```

SubSpaceComponentProperty() : Exports == Implementation where

```

O ==> OutputForm
I  ==> Integer
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
L  ==> List
B  ==> Boolean
  
```

```

Exports ==> SetCategory with
  new      : () -> %
      ++ new() is not documented
  closed? : % -> B
      ++ closed?(x) is not documented
  solid?  : % -> B
      ++ solid?(x) is not documented
  close   : (% , B) -> B
      ++ close(x,b) is not documented
  solid   : (% , B) -> B
      ++ solid(x,b) is not documented
  copy    : % -> %
      ++ copy(x) is not documented

Implementation ==> add
  Rep := Record(closed:B, solid:B)
  closed? p == p.closed
  solid? p == p.solid
  close(p,b) == p.closed := b
  solid(p,b) == p.solid := b
  new() == [false,false]
  copy p ==
    annuderOne := new()
    close(annuderOne,closed? p)
    solid(annuderOne,solid? p)
    annuderOne
  coerce p ==
    hconcat(["Component is ":":0,
      (closed? p => ":":0; "not ":":0),"closed, ":":0, _
      (solid? p => ":":0; "not ":":0),"solid"::0 ])

```

— COMPPROP.dotabb —

```

"COMPPROP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=COMPPROP"]
"BASTYPE"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE"   [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"COMPPROP" -> "BASTYPE"
"COMPPROP" -> "KOERCE"

```

domain SUCH SuchThat

— SuchThat.input —

```

)set break resume
)sys rm -f SuchThat.output
)spool SuchThat.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SuchThat
--R SuchThat(S1: SetCategory,S2: SetCategory) is a domain constructor
--R Abbreviation for SuchThat is SUCH
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SUCH
--R
--R----- Operations -----
--R ?=? : (%,%) -> Boolean          coerce : % -> OutputForm
--R construct : (S1,S2) -> %        hash : % -> SingleInteger
--R latex : % -> String             lhs : % -> S1
--R rhs : % -> S2                  ?~=? : (%,%) -> Boolean
--R
--E 1

)spool
)lisp (bye)

```

— SuchThat.help —

```

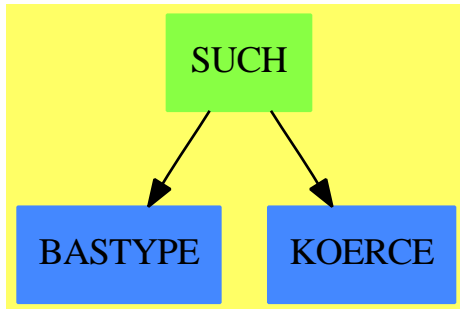
=====
SuchThat examples
=====

```

This domain implements "such that" forms

See Also:

o)show SuchThat

SuchThat (SUCH)**Exports:**

```

coerce  construct  hash  latex  lhs
rhs      ?=?      ?~=?

```

— domain SUCH SuchThat —

```

)abbrev domain SUCH SuchThat
++ Author: Mark Botch
++ Description:
++ This domain implements "such that" forms

SuchThat(S1, S2): Cat == Capsule where
  E ==> OutputForm
  S1, S2: SetCategory

  Cat == SetCategory with
    construct: (S1, S2) -> %
      ++ construct(s,t) makes a form s:t
    lhs: % -> S1
      ++ lhs(f) returns the left side of f
    rhs: % -> S2
      ++ rhs(f) returns the right side of f

  Capsule == add
    Rep := Record(obj: S1, cond: S2)
    construct(o, c) == [o, c]$Record(obj: S1, cond: S2)
    lhs st == st.obj
    rhs st == st.cond
    coerce(w):E == infix("|":E, w.obj::E, w.cond::E)

```

—————

— SUCH.dotabb —

```

"SUCH" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SUCH"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"SUCH" -> "BASTYPE"
"SUCH" -> "KOERCE"

```

domain SWITCH Switch

— Switch.input —

```

)set break resume
)sys rm -f Switch.output
)spool Switch.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Switch
--R
--R Switch is a domain constructor
--R Abbreviation for Switch is SWITCH
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SWITCH
--R
--R----- Operations -----
--R NOT : % -> %               coerce : Symbol -> %
--R coerce : % -> OutputForm
--R AND : (Union(I: Expression(Integer),F: Expression(Float),CF: Expression(Complex(Float)),switch)
--R EQ : (Union(I: Expression(Integer),F: Expression(Float),CF: Expression(Complex(Float)),switch)
--R GE : (Union(I: Expression(Integer),F: Expression(Float),CF: Expression(Complex(Float)),switch)
--R GT : (Union(I: Expression(Integer),F: Expression(Float),CF: Expression(Complex(Float)),switch)
--R LE : (Union(I: Expression(Integer),F: Expression(Float),CF: Expression(Complex(Float)),switch)
--R LT : (Union(I: Expression(Integer),F: Expression(Float),CF: Expression(Complex(Float)),switch)
--R NOT : Union(I: Expression(Integer),F: Expression(Float),CF: Expression(Complex(Float)),switch)
--R OR : (Union(I: Expression(Integer),F: Expression(Float),CF: Expression(Complex(Float)),switch)
--R
--E 1

)spool
)lisp (bye)

```

— Switch.help —

=====

Switch examples

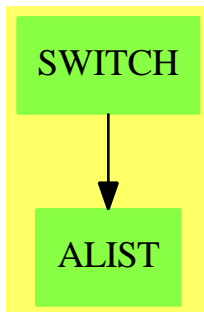
=====

This domain builds representations of boolean expressions for use with the FortranCode domain.

See Also:

o)show Switch

Switch (SWITCH)



See

- ⇒ “Result” (RESULT) [19](#) on page [2300](#)
- ⇒ “FortranCode” (FC) [7](#) on page [931](#)
- ⇒ “FortranProgram” (FORTRAN) [7](#) on page [955](#)
- ⇒ “ThreeDimensionalMatrix” (M3D) [21](#) on page [2702](#)
- ⇒ “SimpleFortranProgram” (SFORT) [20](#) on page [2405](#)
- ⇒ “FortranTemplate” (FTEM) [7](#) on page [967](#)
- ⇒ “FortranExpression” (FEXPR) [7](#) on page [946](#)

Exports:

coerce AND EQ GE GT LE LT NOT OR

— domain SWITCH Switch —

```

)abbrev domain SWITCH Switch
-- Because of a bug in the compiler:
)bo $noSubsumption:=false

```



```

++ Author: Mike Dewar
++ Date Created: April 1991
++ Date Last Updated: March 1994 30.6.94 Added coercion from Symbol MCD
++ Basic Operations:
++ Related Constructors: FortranProgram, FortranCode, FortranTypes
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This domain builds representations of boolean expressions for use with
++ the \axiomType{FortranCode} domain.

Switch():public == private where
  EXPR ==> Union(I:Expression Integer,F:Expression Float,
                 CF:Expression Complex Float,switch:%)

public == CoercibleTo OutputForm with
  coerce : Symbol -> $
    ++ coerce(s) is not documented
  LT : (EXPR,EXPR) -> $
    ++ LT(x,y) returns the \axiomType{Switch} expression representing \spad{x<y}.
  GT : (EXPR,EXPR) -> $
    ++ GT(x,y) returns the \axiomType{Switch} expression representing \spad{x>y}.
  LE : (EXPR,EXPR) -> $
    ++ LE(x,y) returns the \axiomType{Switch} expression representing \spad{x<=y}.
  GE : (EXPR,EXPR) -> $
    ++ GE(x,y) returns the \axiomType{Switch} expression representing \spad{x>=y}.
  OR : (EXPR,EXPR) -> $
    ++ OR(x,y) returns the \axiomType{Switch} expression representing \spad{x or y}.
  EQ : (EXPR,EXPR) -> $
    ++ EQ(x,y) returns the \axiomType{Switch} expression representing \spad{x = y}.
  AND : (EXPR,EXPR) -> $
    ++ AND(x,y) returns the \axiomType{Switch} expression representing \spad{x and y}.
  NOT : EXPR -> $
    ++ NOT(x) returns the \axiomType{Switch} expression representing \spad{\~x}.
  NOT : $ -> $
    ++ NOT(x) returns the \axiomType{Switch} expression representing \spad{\~x}.

private == add
  Rep := Record(op:BasicOperator,rands:List EXPR)

-- Public function definitions

nullOp : BasicOperator := operator NULL

coerce(s:):OutputForm ==
  rat := (s . op)::OutputForm
  ran := [u::OutputForm for u in s.rands]

```

```

(s . op) = nullOp => first ran
#ran = 1 =>
    prefix(rat,ran)
infix(rat,ran)

coerce(s:Symbol):$ == [nullOp,[[s::Expression(Integer)]$EXPR]$List(EXPR)]$Rep

NOT(r:EXPR):% ==
    [operator("~":Symbol),[r]$List(EXPR)]$Rep

NOT(r:%):% ==
    [operator("~":Symbol),[[r]$EXPR]$List(EXPR)]$Rep

LT(r1:EXPR,r2:EXPR):% ==
    [operator("<":Symbol),[r1,r2]$List(EXPR)]$Rep

GT(r1:EXPR,r2:EXPR):% ==
    [operator(">":Symbol),[r1,r2]$List(EXPR)]$Rep

LE(r1:EXPR,r2:EXPR):% ==
    [operator("<=":Symbol),[r1,r2]$List(EXPR)]$Rep

GE(r1:EXPR,r2:EXPR):% ==
    [operator(">=":Symbol),[r1,r2]$List(EXPR)]$Rep

AND(r1:EXPR,r2:EXPR):% ==
    [operator("and":Symbol),[r1,r2]$List(EXPR)]$Rep

OR(r1:EXPR,r2:EXPR):% ==
    [operator("or":Symbol),[r1,r2]$List(EXPR)]$Rep

EQ(r1:EXPR,r2:EXPR):% ==
    [operator("EQ":Symbol),[r1,r2]$List(EXPR)]$Rep

```

— SWITCH.dotabb —

```

"SWITCH" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SWITCH"]
"ALIST"  [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"SWITCH" -> "ALIST"

```

domain SYMBOL Symbol

— Symbol.input —

```

)set break resume
)sys rm -f Symbol.output
)spool Symbol.output
)set message test on
)set message auto off
)clear all
--S 1 of 24
X: Symbol := 'x
--R
--R
--R (1) x
--R
--R                                          Type: Symbol
--E 1

--S 2 of 24
XX: Symbol := x
--R
--R
--R (2) x
--R
--R                                          Type: Symbol
--E 2

--S 3 of 24
A := 'a
--R
--R
--R (3) a
--R
--R                                          Type: Variable(a)
--E 3

--S 4 of 24
B := b
--R
--R
--R (4) b
--R
--R                                          Type: Variable(b)
--E 4

--S 5 of 24
x**2 + 1
--R
--R
--R 2
--R (5) x  + 1

```

```

--R                                         Type: Polynomial(Integer)
--E 5

--S 6 of 24
"Hello":Symbol
--R
--R
--R   (6)  Hello
--R
--R                                         Type: Symbol
--E 6

--S 7 of 24
new()$Symbol
--R
--R
--R   (7)  %A
--R
--R                                         Type: Symbol
--E 7

--S 8 of 24
new()$Symbol
--R
--R
--R   (8)  %B
--R
--R                                         Type: Symbol
--E 8

--S 9 of 24
new("xyz")$Symbol
--R
--R
--R   (9)  %xyz0
--R
--R                                         Type: Symbol
--E 9

--S 10 of 24
X[i,j]
--R
--R
--R   (10)  x
--R         i,j
--R
--R                                         Type: Symbol
--E 10

--S 11 of 24
U := subscript(u, [1,2,1,2])
--R
--R
--R   (11)  u
--R         1,2,1,2

```

```
--R
--E 11
Type: Symbol

--S 12 of 24
V := superscript(v, [n])
--R
--R
--R      n
--R (12) v
--R
--R
--E 12
Type: Symbol

--S 13 of 24
P := argscript(p, [t])
--R
--R
--R (13) p(t)
--R
--R
--E 13
Type: Symbol

--S 14 of 24
scripted? U
--R
--R
--R (14) true
--R
--R
--E 14
Type: Boolean

--S 15 of 24
scripted? X
--R
--R
--R (15) false
--R
--R
--E 15
Type: Boolean

--S 16 of 24
string X
--R
--R
--R (16) "x"
--R
--R
--E 16
Type: String

--S 17 of 24
name U
--R
--R
--R (17) u
--R
--R
--E 17
Type: Symbol
```

--E 17

--S 18 of 24

scripts U

--R

--R

--R (18) [sub= [1,2,1,2],sup= [],presup= [],presub= [],args= []]

--R IType: Record(sub: List(OutputForm),sup: List(OutputForm),...

--E 18

--S 19 of 24

name X

--R

--R

--R (19) x

--R

Type: Symbol

--E 19

--S 20 of 24

scripts X

--R

--R

--R (20) [sub= [],sup= [],presup= [],presub= [],args= []]

--R IType: Record(sub: List(OutputForm),sup: List(OutputForm),...

--E 20

--S 21 of 24

M := script(Mammoth, [[i,j],[k,l],[0,1],[2],[u,v,w]])

--R

--R

--R 0,1 k,l

--R (21) Mammoth (u,v,w)

--R 2 i,j

--R

Type: Symbol

--E 21

--S 22 of 24

scripts M

--R

--R

--R (22) [sub= [i,j],sup= [k,l],presup= [0,1],presub= [2],args= [u,v,w]]

--R IType: Record(sub: List(OutputForm),sup: List(OutputForm),...

--E 22

--S 23 of 24

N := script(Nut, [[i,j],[k,l],[0,1]])

--R

--R

--R 0,1 k,l

--R (23) Nut

```

--R          i,j
--R
--R                                          Type: Symbol
--E 23

--S 24 of 24
scripts N
--R
--R
--R (24) [sub= [i,j],sup= [k,l],presup= [0,1],presub= [],args= []]
--R      --IType: Record(sub: List(OutputForm),sup: List(OutputForm),...
--E 24
)spool
)lisp (bye)

```

— Symbol.help —

=====

Symbol examples

=====

Symbols are one of the basic types manipulated by Axiom. The Symbol domain provides ways to create symbols of many varieties.

The simplest way to create a symbol is to "single quote" an identifier.

```

X: Symbol := 'x
x

```

Type: Symbol

This gives the symbol even if x has been assigned a value. If x has not been assigned a value, then it is possible to omit the quote.

```

XX: Symbol := x
x

```

Type: Symbol

Declarations must be used when working with symbols, because otherwise the interpreter tries to place values in a more specialized type Variable.

```

A := 'a
a

```

Type: Variable a

```

B := b
b

```

Type: Variable b

The normal way of entering polynomials uses this fact.

```
x**2 + 1
  2
  x  + 1
```

Type: Polynomial Integer

Another convenient way to create symbols is to convert a string. This is useful when the name is to be constructed by a program.

```
"Hello"::Symbol
Hello
```

Type: Symbol

Sometimes it is necessary to generate new unique symbols, for example, to name constants of integration. The expression `new()` generates a symbol starting with %.

```
new()$Symbol
%A
```

Type: Symbol

Successive calls to `new` produce different symbols.

```
new()$Symbol
%B
```

Type: Symbol

The expression `new("s")` produces a symbol starting with %s.

```
new("xyz")$Symbol
%xyz0
```

Type: Symbol

A symbol can be adorned in various ways. The most basic thing is applying a symbol to a list of subscripts.

```
X[i,j]
  x
  i,j
```

Type: Symbol

Somewhat less pretty is to attach subscripts, superscripts or arguments.

```
U := subscript(u, [1,2,1,2])
  u
  1,2,1,2
```

Type: Symbol

```
V := superscript(v, [n])
```



```

n
v
Type: Symbol

```

```
P := argscript(p, [t])
      p(t)
                                         Type: Symbol
```

It is possible to test whether a symbol has scripts using the `scripted?` test.

```
scripted? U
  true
Type: Boolean
```

```
scripted? X
  false
                                         Type: Boolean
```

If a symbol is not scripted, then it may be converted to a string.

```
string X
    "x"
                                     Type: String
```

The basic parts can always be extracted using the name and scripts operations.

```
name U
      u
                                     Type: Symbol
```

```
scripts U
  [sub= [1,2,1,2],sup= [],presup= [],presub= [],args= []]
    Type: Record(sub: List OutputForm,
                  sup: List OutputForm,
                  presup: List OutputForm,
                  presub: List OutputForm,
                  args: List OutputForm)
```

```
name X
  x
                                     Type: Symbol
```

```
scripts X
  [sub= [],sup= [],presup= [],presub= [],args= []]
  Type: Record(sub: List OutputForm,
               sup: List OutputForm,
               presup: List OutputForm,
               presub: List OutputForm,
               args: List OutputForm)
```

The most general form is obtained using the script operation. This operation takes an argument which is a list containing, in this order, lists of subscripts, superscripts, presupscripts, presubscripts and arguments to a symbol.

```
M := script(Mammoth, [ [i,j],[k,l],[0,1],[2],[u,v,w] ])
      0,1      k,l
      Mammoth (u,v,w)
      2      i,j
```

Type: Symbol

```
scripts M
[sub= [i,j],sup= [k,l],presup= [0,1],presub= [2],args= [u,v,w]]
Type: Record(sub: List OutputForm,
             sup: List OutputForm,
             presup: List OutputForm,
             presub: List OutputForm,
             args: List OutputForm)
```

If trailing lists of scripts are omitted, they are assumed to be empty.

```
N := script(Nut, [ [i,j],[k,l],[0,1] ])
      0,1      k,l
      Nut
      i,j
```

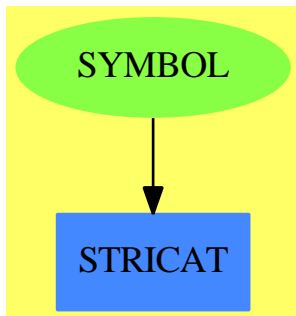
Type: Symbol

```
scripts N
[sub= [i,j],sup= [k,l],presup= [0,1],presub= [],args= []]
Type: Record(sub: List OutputForm,
             sup: List OutputForm,
             presup: List OutputForm,
             presub: List OutputForm,
             args: List OutputForm)
```

See Also:

o)show Symbol

Symbol (SYMBOL)

**Exports:**

argscript	coerce	convert	hash	latex
list	max	min	name	new
OMwrite	patternMatch	resetNew	sample	script
scripts	scripted?	string	subscript	superscript
?~=?	?.?	?<?	?<=?	?=?
?>?	?>=?			

— domain SYMBOL Symbol —

```

)abbrev domain SYMBOL Symbol
++ Author: Stephen Watt
++ Date Created: 1986
++ Date Last Updated: 7 Mar 1991, 29 Apr. 1994 (FDLL)
++ Description:
++ Basic and scripted symbols.

Symbol(): Exports == Implementation where
  L ==> List OutputForm
  Scripts ==> Record(sub:L,sup:L,presup:L,presub:L,args:L)

Exports ==> Join(OrderedSet, ConvertibleTo InputForm, OpenMath,
  ConvertibleTo Symbol,
  ConvertibleTo Pattern Integer, ConvertibleTo Pattern Float,
  PatternMatchable Integer, PatternMatchable Float) with
new: () -> %
  ++ new() returns a new symbol whose name starts with %.
new: % -> %
  ++ new(s) returns a new symbol whose name starts with %s.
resetNew: () -> Void
  ++ resetNew() resets the internal counters that new() and
  ++ new(s) use to return distinct symbols every time.
coerce: String -> %
  ++ coerce(s) converts the string s to a symbol.
name: % -> %

```

```

    ++ name(s) returns s without its scripts.
scripted?: % -> Boolean
    ++ scripted?(s) is true if s has been given any scripts.
scripts: % -> Scripts
    ++ scripts(s) returns all the scripts of s.
script: (% , List L) -> %
    ++ script(s, [a,b,c,d,e]) returns s with subscripts a,
    ++ superscripts b, pre-superscripts c, pre-subscripts d,
    ++ and argument-scripts e. Omitted components are taken to be empty.
    ++ For example, \spad{script(s, [a,b,c])} is equivalent to
    ++ \spad{script(s,[a,b,c,[],[]])}.
script: (% , Scripts) -> %
    ++ script(s, [a,b,c,d,e]) returns s with subscripts a,
    ++ superscripts b, pre-superscripts c, pre-subscripts d,
    ++ and argument-scripts e.
subscript: (% , L) -> %
    ++ subscript(s, [a1,...,an]) returns s
    ++ subscripted by \spad{[a1,...,an]}.
superscript: (% , L) -> %
    ++ superscript(s, [a1,...,an]) returns s
    ++ superscripted by \spad{[a1,...,an]}.
argscript: (% , L) -> %
    ++ argscript(s, [a1,...,an]) returns s
    ++ arg-scripted by \spad{[a1,...,an]}.
elt: (% , L) -> %
    ++ elt(s,[a1,...,an]) or s([a1,...,an]) returns s subscripted by \spad{[a1,...,an]}.
string: % -> String
    ++ string(s) converts the symbol s to a string.
    ++ Error: if the symbol is subscripted.
list: % -> List %
    ++ list(sy) takes a scripted symbol and produces a list
    ++ of the name followed by the scripts.
sample: constant -> %
    ++ sample() returns a sample of %

Implementation ==> add
count: Reference(Integer) := ref 0
xcount: AssociationList(% , Integer) := empty()
istrings: PrimitiveArray(String) :=
    construct ["0","1","2","3","4","5","6","7","8","9"]
-- the following 3 strings shall be of empty intersection
nums:String:="0123456789"
ALPHAS:String:="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
alphas:String:="abcdefghijklmnopqrstuvwxyz"

writeOMSym(dev: OpenMathDevice, x: %): Void ==
    scripted? x =>
        error "Cannot convert a scripted symbol to OpenMath"
    OMputVariable(dev, x pretend Symbol)

```

```

OMwrite(x: %): String ==
  s: String := ""
  sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
  dev: OpenMathDevice := OMopenString(sp @ String, OMencodingXML)
  OMputObject(dev)
  writeOMSym(dev, x)
  OMputEndObject(dev)
  OMclose(dev)
  s := OM_-STRINGPTRTOSTRING(sp)$Lisp @ String
  s

OMwrite(x: %, wholeObj: Boolean): String ==
  s: String := ""
  sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
  dev: OpenMathDevice := OMopenString(sp @ String, OMencodingXML)
  if wholeObj then
    OMputObject(dev)
  writeOMSym(dev, x)
  if wholeObj then
    OMputEndObject(dev)
  OMclose(dev)
  s := OM_-STRINGPTRTOSTRING(sp)$Lisp @ String
  s

OMwrite(dev: OpenMathDevice, x: %): Void ==
  OMputObject(dev)
  writeOMSym(dev, x)
  OMputEndObject(dev)

OMwrite(dev: OpenMathDevice, x: %, wholeObj: Boolean): Void ==
  if wholeObj then
    OMputObject(dev)
  writeOMSym(dev, x)
  if wholeObj then
    OMputEndObject(dev)

hd:String      := "*"
lhd            := #hd
ord0           := ord char("0")$Character

istring : Integer -> String
symprefix : Scripts -> String
syscripts: Scripts -> L

convert(s:%):InputForm == convert(s pretend Symbol)$InputForm
convert(s:%):Symbol    == s pretend Symbol
coerce(s:String):%     == VALUES(INTERN(s)$Lisp)$Lisp
x = y                == EQUAL(x,y)$Lisp
x < y                == GGREATERP(y, x)$Lisp
coerce(x:%):OutputForm == outputForm(x pretend Symbol)

```

```

subscript(sy, lx)      == script(sy, [lx, nil, nil(), nil(), nil()])
elt(sy, lx)           == subscript(sy, lx)
superscript(sy, lx)    == script(sy, [nil(), lx, nil(), nil(), nil()])
argscript(sy, lx)      == script(sy, [nil(), nil(), nil(), nil(), lx])

patternMatch(x:%, p:Pattern Integer, l:PatternMatchResult(Integer, %)) ==
  (patternMatch(x pretend Symbol, p, l pretend
    PatternMatchResult(Integer, Symbol))$PatternMatchSymbol(Integer))
    pretend PatternMatchResult(Integer, %)

patternMatch(x:%, p:Pattern Float, l:PatternMatchResult(Float, %)) ==
  (patternMatch(x pretend Symbol, p, l pretend
    PatternMatchResult(Float, Symbol))$PatternMatchSymbol(Float))
    pretend PatternMatchResult(Float, %)

convert(x:%):Pattern(Float) ==
  coerce(x pretend Symbol)$Pattern(Float)

convert(x:%):Pattern(Integer) ==
  coerce(x pretend Symbol)$Pattern(Integer)

syprefix sc ==
  ns: List Integer := [#sc.presub, #sc.presup, #sc.sup, #sc.sub]
  while #ns >= 2 and zero? first ns repeat ns := rest ns
  concat concat(concat(hd, istring(#sc.args)),
    [istring n for n in reverse_! ns])

syscripts sc ==
  all := sc.presub
  all := concat(sc.presup, all)
  all := concat(sc.sup, all)
  all := concat(sc.sub, all)
  concat(all, sc.args)

script(sy: %, ls: List L) ==
  sc: Scripts := [nil(), nil(), nil(), nil(), nil()]
  if not null ls then (sc.sub := first ls; ls := rest ls)
  if not null ls then (sc.sup := first ls; ls := rest ls)
  if not null ls then (sc.presup := first ls; ls := rest ls)
  if not null ls then (sc.presub := first ls; ls := rest ls)
  if not null ls then (sc.args := first ls; ls := rest ls)
  script(sy, sc)

script(sy: %, sc: Scripts) ==
  scripted? sy => error "Cannot add scripts to a scripted symbol"
  (concat(concat(syprefix sc, string name sy)::%::OutputForm,
    syscripts sc)) pretend %

string e ==
  not scripted? e => PNAME(e)$Lisp

```

```

error "Cannot form string from non-atomic symbols."

-- Scripts ==> Record(sub:L,sup:L,presup:L,presub:L,args:L)
latex e ==
  s : String := (PNAME(name e)$Lisp) @ String
  if #s > 1 and s.1 ^~= char "\"" then
    s := concat("\mbox{\it ", concat(s, "}")$String)$String
  not scripted? e => s
  ss : Scripts := scripts e
  lo : List OutputForm := ss.sub
  sc : String
  if not empty? lo then
    sc := "__{"
    while not empty? lo repeat
      sc := concat(sc, latex first lo)$String
      lo := rest lo
      if not empty? lo then sc := concat(sc, ", ")$String
    sc := concat(sc, "}")$String
    s := concat(s, sc)$String
  lo := ss.sup
  if not empty? lo then
    sc := "^{"
    while not empty? lo repeat
      sc := concat(sc, latex first lo)$String
      lo := rest lo
      if not empty? lo then sc := concat(sc, ", ")$String
    sc := concat(sc, "}")$String
    s := concat(s, sc)$String
  lo := ss.presup
  if not empty? lo then
    sc := "{}^{"
    while not empty? lo repeat
      sc := concat(sc, latex first lo)$String
      lo := rest lo
      if not empty? lo then sc := concat(sc, ", ")$String
    sc := concat(sc, "}")$String
    s := concat(sc, s)$String
  lo := ss.presub
  if not empty? lo then
    sc := "{}_{"
    while not empty? lo repeat
      sc := concat(sc, latex first lo)$String
      lo := rest lo
      if not empty? lo then sc := concat(sc, ", ")$String
    sc := concat(sc, "}")$String
    s := concat(sc, s)$String
  lo := ss.args
  if not empty? lo then
    sc := "\left( {"
    while not empty? lo repeat

```

```

        sc := concat(sc, latex first lo)$String
        lo := rest lo
        if not empty? lo then sc := concat(sc, ", ")$String
    sc := concat(sc, "}" \right)$String
    s := concat(s, sc)$String
s

anyRadix(n:Integer,s:String):String ==
    ns:String:=""
    repeat
        qr := divide(n,#s)
        n := qr.quotient
        ns := concat(s.(qr.remainder+minIndex s),ns)
        if zero?(n) then return ns

new() ==
    sym := anyRadix(count():Integer,ALPHAS)
    count() := count() + 1
    concat("%",sym)::%

new x ==
    n:Integer :=
        (u := search(x, xcount)) case "failed" => 0
        inc(u:Integer)
    xcount(x) := n
    xx :=
        not scripted? x => string x
        string name x
    xx := concat("%",xx)
    xx :=
        (position(xx.maxIndex(xx),nums)>=minIndex(nums)) =>
            concat(xx, anyRadix(n,alphas))
        concat(xx, anyRadix(n,nums))
    not scripted? x => xx::%
    script(xx::%,scripts x)

resetNew() ==
    count() := 0
    for k in keys xcount repeat remove_!(k, xcount)
    void

scripted? sy ==
    not ATOM(sy)$Lisp

name sy ==
    not scripted? sy => sy
    str := string first list sy
    for i in lhd+1..#str repeat
        not digit?(str.i) => return((str.(i..#str))::%)
    error "Improper scripted symbol"

```



```

scripts sy ==
  not scripted? sy => [nil(), nil(), nil(), nil(), nil()]
  nscripts: List NonNegativeInteger := [0, 0, 0, 0, 0]
  lscripts: List L := [nil(), nil(), nil(), nil(), nil()]
  str := string first list sy
  nstr := #str
  m := minIndex nscripts
  for i in m.. for j in lhd+1..nstr while digit?(str.j) repeat
    nscripts.i := (ord(str.j) - ord0)::NonNegativeInteger
  -- Put the number of function scripts at the end.
  nscripts := concat(rest nscripts, first nscripts)
  allscripts := rest list sy
  m := minIndex lscripts
  for i in m.. for n in nscripts repeat
    #allscripts < n => error "Improper script count in symbol"
    lscripts.i := [a::OutputForm for a in first(allscripts, n)]
    allscripts := rest(allscripts, n)
  [lscripts.m, lscripts.(m+1), lscripts.(m+2),
    lscripts.(m+3), lscripts.(m+4)]

istring n ==
  n > 9 => error "Can have at most 9 scripts of each kind"
  istrings.(n + minIndex istrings)

list sy ==
  not scripted? sy =>
    error "Cannot convert a symbol to a list if it is not subscripted"
  sy pretend List(%)

sample() == "aSymbol"::%

```

— SYMBOL.dotabb —

```

"SYMBOL" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SYMBOL",
  shape=ellipse]
"STRICAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=STRICAT"]
"SYMBOL" -> "STRICAT"

```

domain SYMTAB SymbolTable

— SymbolTable.input —

```

)set break resume
)sys rm -f SymbolTable.output
)spool SymbolTable.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SymbolTable
--R
--R SymbolTable is a domain constructor
--R Abbreviation for SymbolTable is SYMTAB
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SYMTAB
--R
--R----- Operations -----
--R coerce : % -> OutputForm          empty : () -> %
--R externalList : % -> List(Symbol)    newTypeLists : % -> SExpression
--R parametersOf : % -> List(Symbol)    printTypes : % -> Void
--R coerce : % -> Table(Symbol,FortranType)
--R declare! : (Symbol,FortranType,%) -> FortranType
--R declare! : (List(Symbol),FortranType,%) -> FortranType
--R fortranTypeOf : (Symbol,%) -> FortranType
--R symbolTable : List(Record(key: Symbol,entry: FortranType)) -> %
--R typeList : (FortranScalarType,%) -> List(Union(name: Symbol,bounds: List(Union(S: Symbol,P: Polynomial(Integer))))))
--R typeLists : % -> List(List(Union(name: Symbol,bounds: List(Union(S: Symbol,P: Polynomial(Integer))))))
--R
--E 1

)spool
)lisp (bye)

```

— SymbolTable.help —

```

=====
SymbolTable examples
=====

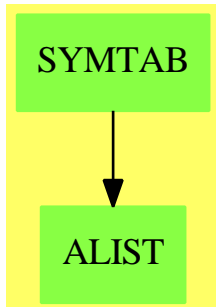
```

Create and manipulate a symbol table for generated FORTRAN code

See Also:

o)show SymbolTable

SymbolTable (SYMTAB)



See

- ⇒ “FortranScalarType” (FST) [7](#) on page [962](#)
- ⇒ “FortranType” (FT) [7](#) on page [971](#)
- ⇒ “TheSymbolTable” (SYMS) [21](#) on page [2696](#)

Exports:

coerce	declare!	empty	externalList	fortranTypeOf
newTypeLists	parametersOf	printTypes	symbolTable	typeList
typeLists				

— domain SYMTAB SymbolTable —

```

)abbrev domain SYMTAB SymbolTable
++ Author: Mike Dewar
++ Date Created:  October 1992
++ Date Last Updated: 12 July 1994
++ Description:
++ Create and manipulate a symbol table for generated FORTRAN code

```

```

SymbolTable() : exports == implementation where

```

```

T  ==> Union(S:Symbol,P:Polynomial Integer)
TL1 ==> List T
TU  ==> Union(name:Symbol,bounds:TL1)
TL  ==> List TU
SEX ==> SExpression
OFORM ==> OutputForm
L   ==> List
FSTU ==> Union(fst:FortranScalarType,void:"void")

exports ==> CoercibleTo OutputForm with
  coerce : $ -> Table(Symbol,FortranType)
    ++ coerce(x) returns a table view of x
  empty  : () -> $
    ++ empty() returns a new, empty symbol table
  declare! : (L Symbol,FortranType,$) -> FortranType

```

```

    ++ declare!(l,t,tab) creates new entrys in tab, declaring each of l
    ++ to be of type t
declare! : (Symbol,FortranType,$) -> FortranType
    ++ declare!(u,t,tab) creates a new entry in tab, declaring u to be of
    ++ type t
fortranTypeOf : (Symbol,$) -> FortranType
    ++ fortranTypeOf(u,tab) returns the type of u in tab
parametersOf: $ -> L Symbol
    ++ parametersOf(tab) returns a list of all the symbols declared in tab
typeList : (FortranScalarType,$) -> TL
    ++ typeList(t,tab) returns a list of all the objects of type t in tab
externalList : $ -> L Symbol
    ++ externalList(tab) returns a list of all the external symbols in tab
typeLists : $ -> L TL
    ++ typeLists(tab) returns a list of lists of types of objects in tab
newTypeLists : $ -> SEX
    ++ newTypeLists(x) is not documented
printTypes: $ -> Void
    ++ printTypes(tab) produces FORTRAN type declarations from tab, on the
    ++ current FORTRAN output stream
symbolTable: L Record(key:Symbol,entry:FortranType) -> $
    ++ symbolTable(l) creates a symbol table from the elements of l.

implementation ==> add

Rep := Table(Symbol,FortranType)

coerce(t:$):OFORM ==
    coerce(t)$Rep

coerce(t:$):Table(Symbol,FortranType) ==
    t pretend Table(Symbol,FortranType)

symbolTable(l:L Record(key:Symbol,entry:FortranType)):$ ==
    table(l)$Rep

empty():$ ==
    empty()$Rep

parametersOf(tab:$):L(Symbol) ==
    keys(tab)

declare!(name:Symbol,type:FortranType,tab:$):FortranType ==
    setelt(tab,name,type)$Rep
    type

declare!(names:L Symbol,type:FortranType,tab:$):FortranType ==
    for name in names repeat setelt(tab,name,type)$Rep
    type

```

```

fortranTypeOf(u:Symbol,tab:$):FortranType ==
  elt(tab,u)$Rep

externalList(tab:$):L(Symbol) ==
  [u for u in keys(tab) | external? fortranTypeOf(u,tab)]

typeList(type:FortranScalarType,tab:$):TL ==
  scalarList := []@TL
  arrayList  := []@TL
  for u in keys(tab)$Rep repeat
    uType : FortranType := fortranTypeOf(u,tab)
    sType : FSTU := scalarTypeOf(uType)
    if (sType case fst and (sType.fst)=type) then
      uDim : TL1 := [[v]$T for v in dimensionsOf(uType)]
      if empty? uDim then
        scalarList := cons([u]$TU,scalarList)
      else
        arrayList := cons([cons([u],uDim)$TL1]$TU,arrayList)
    -- Scalars come first in case they are integers which are later
    -- used as an array dimension.
  append(scalarList,arrayList)

typeList2(type:FortranScalarType,tab:$):TL ==
  t1 := []@TL
  symbolType : Symbol := coerce(type)$FortranScalarType
  for u in keys(tab)$Rep repeat
    uType : FortranType := fortranTypeOf(u,tab)
    sType : FSTU := scalarTypeOf(uType)
    if (sType case fst and (sType.fst)=type) then
      uDim : TL1 := [[v]$T for v in dimensionsOf(uType)]
      t1 := if empty? uDim then cons([u]$TU,t1)
            else cons([cons([u],uDim)$TL1]$TU,t1)
  empty? t1 => t1
  cons([symbolType]$TU,t1)

updateList(sType:SEX,name:SEX,lDims:SEX,t1:SEX):SEX ==
  l : SEX := ASSOC(sType,t1)$Lisp
  entry : SEX := if null?(lDims) then name else CONS(name,lDims)$Lisp
  null?(l) => CONS([sType,entry]$Lisp,t1)$Lisp
  RPLACD(l,CONS(entry,cdr l)$Lisp)$Lisp
  t1

newTypeLists(tab:$):SEX ==
  t1 := []$Lisp
  for u in keys(tab)$Rep repeat
    uType : FortranType := fortranTypeOf(u,tab)
    sType : FSTU := scalarTypeOf(uType)
    dims : L Polynomial Integer := dimensionsOf uType
    lDims : L SEX := [convert(convert(v)@InputForm)@SEX for v in dims]
    lType : SEX := if sType case void

```

```

        then convert(void::Symbol)@SEX
        else coerce(sType.fst)$FortranScalarType
    tl := updateList(lType,convert(u)@SEX,convert(lDims)@SEX,tl)
tl

typeLists(tab:$):L(TL) ==
    fortranTypes := ["real"::FortranScalarType, _
        "double precision"::FortranScalarType, _
        "integer"::FortranScalarType, _
        "complex"::FortranScalarType, _
        "logical"::FortranScalarType, _
        "character"::FortranScalarType]@L(FortranScalarType)
    tl := []@L TL
    for u in fortranTypes repeat
        types : TL := typeList2(u,tab)
        if (not null types) then
            tl := cons(types,tl)$(L TL)
    tl

oForm2(w:T):OFORM ==
    w case S => w.S::OFORM
    w case P => w.P::OFORM

oForm(v:TU):OFORM ==
    v case name => v.name::OFORM
    v case bounds =>
        ll : L OFORM := [oForm2(uu) for uu in v.bounds]
        ll :: OFORM

outForm(t:TL):L OFORM ==
    [oForm(u) for u in t]

printTypes(tab:$):Void ==
    -- It is important that INTEGER is the first element of this
    -- list since INTEGER symbols used in type declarations must
    -- be declared in advance.
    ft := ["integer"::FortranScalarType, _
        "real"::FortranScalarType, _
        "double precision"::FortranScalarType, _
        "complex"::FortranScalarType, _
        "logical"::FortranScalarType, _
        "character"::FortranScalarType]@L(FortranScalarType)
    for ty in ft repeat
        tl : TL := typeList(ty,tab)
        otl : L OFORM := outForm(tl)
        fortFormatTypes(ty::OFORM,otl)$Lisp
    el : L OFORM := [u::OFORM for u in externalList(tab)]
    fortFormatTypes("EXTERNAL"::OFORM,el)$Lisp
    void()$Void

```

— SYMTAB.dotabb —

```
"SYMTAB" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SYMTAB"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"SYMTAB" -> "ALIST"
```

domain SYMPOLY SymmetricPolynomial

— SymmetricPolynomial.input —

```
)set break resume
)sys rm -f SymmetricPolynomial.output
)spool SymmetricPolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SymmetricPolynomial
--R
--R SymmetricPolynomial(R: Ring) is a domain constructor
--R Abbreviation for SymmetricPolynomial is SYMPOLY
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SYMPOLY
--R
--R----- Operations -----
--R ?? : (R,%) -> %               ?? : (% ,R) -> %
--R ?? : (% ,%) -> %             ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> % ?? : (% ,PositiveInteger) -> %
--R ?? : (% ,%) -> %             ?-? : (% ,%) -> %
--R -? : % -> %                  ?? : (% ,%) -> Boolean
--R 1 : () -> %                  0 : () -> %
--R ?? : (% ,PositiveInteger) -> % coefficient : (% ,Partition) -> R
--R coefficients : % -> List(R)   coerce : R -> %
--R coerce : Integer -> %         coerce : % -> OutputForm
--R degree : % -> Partition       ground : % -> R
--R ground? : % -> Boolean        hash : % -> SingleInteger
--R latex : % -> String           leadingCoefficient : % -> R
--R leadingMonomial : % -> %      map : ((R -> R),%) -> %
--R minimumDegree : % -> Partition monomial : (R,Partition) -> %
--R monomial? : % -> Boolean      one? : % -> Boolean
```

```

--R recip : % -> Union(%, "failed")      reductum : % -> %
--R retract : % -> R                      sample : () -> %
--R zero? : % -> Boolean                  ?~=? : (%,%) -> Boolean
--R ?? : (%, Fraction(Integer)) -> % if R has ALGEBRA(FRAC(INT))
--R ?? : (Fraction(Integer), %) -> % if R has ALGEBRA(FRAC(INT))
--R ?? : (NonNegativeInteger, %) -> %
--R ***? : (%, NonNegativeInteger) -> %
--R ?/? : (%, R) -> % if R has FIELD
--R ?^? : (%, NonNegativeInteger) -> %
--R associates? : (%, %) -> Boolean if R has INTDOM
--R binomThmExpt : (%, %, NonNegativeInteger) -> % if R has COMRING
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if R has CHARNZ
--R coerce : Fraction(Integer) -> % if R has ALGEBRA(FRAC(INT)) or R has RETRACT(FRAC(INT))
--R coerce : % -> % if R has INTDOM
--R content : % -> R if R has GCDDOM
--R exquo : (%, R) -> Union(%, "failed") if R has INTDOM
--R exquo : (%, %) -> Union(%, "failed") if R has INTDOM
--R fmeq : (%, Partition, R, %) -> % if Partition has CABMON and R has INTDOM
--R mapExponents : ((Partition -> Partition), %) -> %
--R numberOfMonomials : % -> NonNegativeInteger
--R pomopo! : (%, R, Partition, %) -> %
--R primitivePart : % -> % if R has GCDDOM
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retract : % -> Integer if R has RETRACT(INT)
--R retractIfCan : % -> Union(R, "failed")
--R retractIfCan : % -> Union(Fraction(Integer), "failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(Integer, "failed") if R has RETRACT(INT)
--R subtractIfCan : (%, %) -> Union(%, "failed")
--R unit? : % -> Boolean if R has INTDOM
--R unitCanonical : % -> % if R has INTDOM
--R unitNormal : % -> Record(unit: %, canonical: %, associate: %) if R has INTDOM
--R
--E 1

```

```

)spool
)lisp (bye)

```

— SymmetricPolynomial.help —

```

=====
SymmetricPolynomial examples
=====

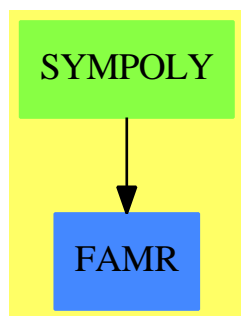
```

This domain implements symmetric polynomial

See Also:


```
o )show SymmetricPolynomial
```

SymmetricPolynomial (SYMPOLY)



See

⇒ “Partition” (PARTITION) [17](#) on page [1923](#)

Exports:

0	1	associates?	binomThmExpt
characteristic	charthRoot	coefficient	
coefficients	coerce	content	degree
exquo	exquo	fmeceg	ground
ground?	hash	latex	leadingCoefficient
leadingMonomial	map	mapExponents	minimumDegree
monomial	monomial?	numberOfMonomials	one?
pomopo!	primitivePart	recip	reductum
retract	retractIfCan	sample	subtractIfCan
unit?	unitCanonical	unitNormal	zero?
?^=?	?**?	?/?	?^?
?*?	?+?	?-?	-?
?=?			

— domain SYMPOLY SymmetricPolynomial —

```

)abbrev domain SYMPOLY SymmetricPolynomial
++ Author: Mark Botch
++ Description:
++ This domain implements symmetric polynomial

SymmetricPolynomial(R:Ring) == PolynomialRing(R,Partition) add
  Term:= Record(k:Partition,c:R)
  Rep:= List Term
  
```

```

-- override PR implementation because coeff. arithmetic too expensive (??)

    if R has EntireRing then
      (p1:%) * (p2:%) ==
        null p1 => 0
        null p2 => 0
        zero?(p1.first.k) => p1.first.c * p2
--      one? p2 => p1
      (p2 = 1) => p1
      +/[[t1.k+t2.k,t1.c*t2.c]$Term for t2 in p2]
        for t1 in reverse(p1)]
        -- This 'reverse' is an efficiency improvement:
        -- reduces both time and space [Abbott/Bradford/Davenport]
    else
      (p1:%) * (p2:%) ==
        null p1 => 0
        null p2 => 0
        zero?(p1.first.k) => p1.first.c * p2
--      one? p2 => p1
      (p2 = 1) => p1
      +/[[t1.k+t2.k,r]$Term for t2 in p2 | (r:=t1.c*t2.c) ^= 0]
        for t1 in reverse(p1)]
        -- This 'reverse' is an efficiency improvement:
        -- reduces both time and space [Abbott/Bradford/Davenport]



---



— SYMPOLY.dotabb —

"SYMPOLY" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SYMPOLY"]
"FAMR" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAMR"]
"SYMPOLY" -> "FAMR"



---



```


Chapter 21

Chapter T

domain TABLE Table

— Table.input —

```
)set break resume
)sys rm -f Table.output
)spool Table.output
)set message test on
)set message auto off
)clear all
--S 1 of 18
t: Table(Polynomial Integer, String) := table()
--R
--R
--R (1) table()
--R
--R                                         Type: Table(Polynomial(Integer),String)
--E 1

--S 2 of 18
setelt(t, x**2 - 1, "Easy to factor")
--R
--R
--R (2) "Easy to factor"
--R
--R                                         Type: String
--E 2

--S 3 of 18
t(x**3 + 1) := "Harder to factor"
--R
--R
--R (3) "Harder to factor"
```



```

--S 16 of 18
#t
--R
--R
--R (16) 2
--R
--R                                          Type: PositiveInteger
--E 16

--S 17 of 18
members t
--R
--R
--R (17) ["The easiest to factor","Harder to factor"]
--R
--R                                          Type: List(String)
--E 17

--S 18 of 18
count(s: String +-> prefix?("Hard", s), t)
--R
--R
--R (18) 1
--R
--R                                          Type: PositiveInteger
--E 18
)spool
)lisp (bye)

```

— Table.help —

```

=====
Table examples
=====

```

The Table constructor provides a general structure for associative storage. This type provides hash tables in which data objects can be saved according to keys of any type. For a given table, specific types must be chosen for the keys and entries.

In this example the keys to the table are polynomials with integer coefficients. The entries in the table are strings.

```

t: Table(Polynomial Integer, String) := table()
table()
                                          Type: Table(Polynomial Integer,String)

```

To save an entry in the table, the setelt operation is used. This can be called directly, giving the table a key and an entry.

```
setelt(t, x**2 - 1, "Easy to factor")
    "Easy to factor"
                                Type: String
```

Alternatively, you can use assignment syntax.

```
t(x**3 + 1) := "Harder to factor"
    "Harder to factor"
                                Type: String
```

```
t(x) := "The easiest to factor"
    "The easiest to factor"
                                Type: String
```

Entries are retrieved from the table by calling the elt operation.

```
elt(t, x)
    "The easiest to factor"
                                Type: String
```

This operation is called when a table is "applied" to a key using this or the following syntax.

```
t.x
    "The easiest to factor"
                                Type: String
```

```
t x
    "The easiest to factor"
                                Type: String
```

Parentheses are used only for grouping. They are needed if the key is an infix expression.

```
t.(x**2 - 1)
    "Easy to factor"
                                Type: String
```

Note that the elt operation is used only when the key is known to be in the table, otherwise an error is generated.

```
t (x**3 + 1)
    "Harder to factor"
                                Type: String
```

You can get a list of all the keys to a table using the keys operation.

```
keys t
      3      2
    [x,x  + 1,x  - 1]
```


Type: List Polynomial Integer

If you wish to test whether a key is in a table, the search operation is used. This operation returns either an entry or "failed".

```
search(x, t)
"The easiest to factor"
Type: Union(String,...)
```

```
search(x**2, t)
"failed"
Type: Union("failed",...)
```

The return type is a union so the success of the search can be tested using case.

```
search(x**2, t) case "failed"
true
Type: Boolean
```

The remove operation is used to delete values from a table.

```
remove!(x**2-1, t)
"Easy to factor"
Type: Union(String,...)
```

If an entry exists under the key, then it is returned. Otherwise remove returns "failed".

```
remove!(x-1, t)
"failed"
Type: Union("failed",...)
```

The number of key-entry pairs can be found using the # operation.

```
#t
2
Type: PositiveInteger
```

Just as keys returns a list of keys to the table, a list of all the entries can be obtained using the members operation.

```
members t
(17) ["The easiest to factor","Harder to factor"]
Type: List String
```

A number of useful operations take functions and map them on to the table to compute the result. Here we count the entries which have "Hard" as a prefix.

```
count(s: String +-> prefix?("Hard", s), t)
1
```

Type: PositiveInteger

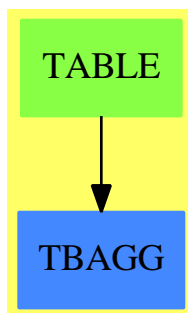
Other table types are provided to support various needs.

- o AssociationList gives a list with a table view. This allows new entries to be appended onto the front of the list to cover up old entries. This is useful when table entries need to be stacked or when frequent list traversals are required.
- o EqTable gives tables in which keys are considered equal only when they are in fact the same instance of a structure.
- o StringTable should be used when the keys are known to be strings.
- o SparseTable provides tables with default entries, so lookup never fails. The GeneralSparseTable constructor can be used to make any table type behave this way.
- o KeyedAccessFile allows values to be saved in a file, accessed as a table.

See Also:

- o)help AssociationList
- o)help EqTable
- o)help StringTable
- o)help SparseTable
- o)help GeneralSparseTable
- o)help KeyedAccessFile
- o)show Table

Table (TABLE)



See

- ⇒ “HashTable” (HASHTBL) [9](#) on page [1120](#)
- ⇒ “InnerTable” (INTABL) [10](#) on page [1335](#)
- ⇒ “EqTable” (EQTBL) [6](#) on page [698](#)

⇒ “StringTable” (STRTBL) [20](#) on page [2610](#)
 ⇒ “GeneralSparseTable” (GSTBL) [8](#) on page [1078](#)
 ⇒ “SparseTable” (STBL) [20](#) on page [2450](#)

Exports:

any?	bag	coerce	construct	convert
copy	count	dictionary	elt	empty
empty?	entries	entry?	eq?	eval
eval	every?	extract!	fill!	find
first	hash	index?	indices	insert!
inspect	key?	keys	latex	less?
map	map	map!	maxIndex	member?
members	minIndex	more?	parts	qelt
qsetelt!	reduce	remove	remove!	removeDuplicates
sample	search	select	select!	setelt
size?	swap!	table	#?	?=?
?~=?	??			

— domain TABLE Table —

```
)abbrev domain TABLE Table
++ Author: Stephen M. Watt, Barry Trager
++ Date Created: 1985
++ Date Last Updated: Sept 15, 1992
++ Description:
++ This is the general purpose table type.
++ The keys are hashed to look up the entries.
++ This creates a \spadtype{HashTable} if equal for the Key
++ domain is consistent with Lisp EQUAL otherwise an
++ \spadtype{AssociationList}
```

```
Table(Key: SetCategory, Entry: SetCategory):Exports == Implementation where
  Exports ==> TableAggregate(Key, Entry) with
    finiteAggregate
```

```
Implementation ==> InnerTable(Key, Entry,
  if hashable(Key)$Lisp then HashTable(Key, Entry, "UEQUAL")
  else AssociationList(Key, Entry))
```

— TABLE.dotabb —

```
"TABLE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=TABLE"]
"TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]
"TABLE" -> "TBAGG"
```

domain TABLEAU Tableau

— Tableau.input —

```
)set break resume
)sys rm -f Tableau.output
)spool Tableau.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Tableau
--R
--R Tableau(S: SetCategory) is a domain constructor
--R Abbreviation for Tableau is TABLEAU
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for TABLEAU
--R
--R----- Operations -----
--R coerce : % -> OutputForm          listOfLists : % -> List(List(S))
--R tableau : List(List(S)) -> %
--R
--E 1

)spool
)lisp (bye)
```

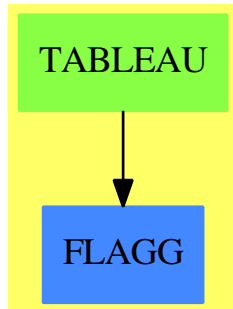
— Tableau.help —

Tableau examples

The tableau domain is for printing Young tableaux, and
coercions to and from List List S where S is a set.

See Also:
o)show Tableau

Tableau (TABLEAU)



Exports:

coerce listOfLists tableau

— domain TABLEAU Tableau —

```

)abbrev domain TABLEAU Tableau
++ Author: William H. Burge
++ Date Created: 1987
++ Date Last Updated: 23 Sept 1991
++ Description:
++ The tableau domain is for printing Young tableaux, and
++ coercions to and from List List S where S is a set.

Tableau(S:SetCategory):Exports == Implementation where
  L ==> List
  I ==> Integer
  NNI ==> NonNegativeInteger
  OUT ==> OutputForm
  V ==> Vector
  fm==>formMatrix$PrintableForm()
Exports ==> with
  tableau : L L S -> %
    ++ tableau(ll) converts a list of lists ll to a tableau.
  listOfLists : % -> L L S
    ++ listOfLists t converts a tableau t to a list of lists.
  coerce : % -> OUT
    ++ coerce(t) converts a tableau t to an output form.
Implementation ==> add

Rep := L L S

tableau(lls:(L L S)) == lls pretend %
listOfLists(x:%):(L L S) == x pretend (L L S)
makeupv : (NNI,L S) -> L OUT
makeupv(n,ls)==

```

```

v:=new(n,message " ")(List OUT)
for i in 1..#ls for s in ls repeat v.i:=box(s::OUT)
v
maketab : L L S -> OUT
maketab lls ==
  ll : L OUT :=
    empty? lls => [[empty()]]
    sz:NNI:=# first lls
    [blankSeparate makeupv(sz,i) for i in lls]
  pile ll

coerce(x:%):OUT == maketab listOfLists x

```

— TABLEAU.dotabb —

```

"TABLEAU" [color="#88FF44",href="bookvol10.3.pdf#nameddest=TABLEAU"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"TABLEAU" -> "FLAGG"

```

domain TS TaylorSeries

— TaylorSeries.input —

```

)set break resume
)sys rm -f TaylorSeries.output
)spool TaylorSeries.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TaylorSeries
--R
--R TaylorSeries(Coef: Ring) is a domain constructor
--R Abbreviation for TaylorSeries is TS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for TS
--R
--R----- Operations -----
--R ?? : (%,Coef) -> %          ?? : (Coef,%) -> %

```

```

--R ?? : (% , %) -> %
--R ?? : (PositiveInteger , %) -> %
--R +? : (% , %) -> %
--R -? : % -> %
--R D : (% , List(Symbol)) -> %
--R 1 : () -> %
--R ^? : (% , PositiveInteger) -> %
--R coerce : Symbol -> %
--R coerce : % -> OutputForm
--R differentiate : (% , Symbol) -> %
--R eval : (% , Equation(%)) -> %
--R hash : % -> SingleInteger
--R leadingCoefficient : % -> Coef
--R map : ((Coef -> Coef) , %) -> %
--R one? : % -> Boolean
--R recip : % -> Union(% , "failed")
--R sample : () -> %
--R zero? : % -> Boolean
--R ?? : (Fraction(Integer) , %) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (% , Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (NonNegativeInteger , %) -> %
--R ??? : (% , Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (% , %) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (% , NonNegativeInteger) -> %
--R ?/? : (% , Coef) -> % if Coef has FIELD
--R D : (% , List(Symbol) , List(NonNegativeInteger)) -> %
--R D : (% , Symbol , NonNegativeInteger) -> %
--R ^? : (% , NonNegativeInteger) -> %
--R acos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acoth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R associates? : (% , %) -> Boolean if Coef has INTDOM
--R atan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R atanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(% , "failed") if Coef has CHARNZ
--R coefficient : (% , NonNegativeInteger) -> Polynomial(Coef)
--R coefficient : (% , List(Symbol) , List(NonNegativeInteger)) -> %
--R coefficient : (% , Symbol , NonNegativeInteger) -> %
--R coefficient : (% , IndexedExponents(Symbol)) -> Coef
--R coerce : Fraction(Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R coerce : % -> % if Coef has INTDOM
--R coerce : Coef -> % if Coef has COMRING
?? : (Integer , %) -> %
??? : (% , PositiveInteger) -> %
?-? : (% , %) -> %
?=? : (% , %) -> Boolean
D : (% , Symbol) -> %
0 : () -> %
coerce : Polynomial(Coef) -> %
coerce : Integer -> %
complete : % -> %
eval : (% , % , %) -> %
eval : (% , Symbol , %) -> %
latex : % -> String
leadingMonomial : % -> %
monomial? : % -> Boolean
pole? : % -> Boolean
reductum : % -> %
variables : % -> List(Symbol)
?~=? : (% , %) -> Boolean

```

```

--R cos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R coth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R degree : % -> IndexedExponents(Symbol)
--R differentiate : (%,List(Symbol),List(NonNegativeInteger)) -> %
--R differentiate : (%,Symbol,NonNegativeInteger) -> %
--R differentiate : (%,List(Symbol)) -> %
--R eval : (%,List(%),List(%)) -> %
--R eval : (%,List(Equation(%))) -> %
--R eval : (%,List(Symbol),List(%)) -> %
--R exp : % -> % if Coef has ALGEBRA(FRAC(INT))
--R exquo : (%,%) -> Union(%, "failed") if Coef has INTDOM
--R extend : (%,NonNegativeInteger) -> %
--R fintegrate : ((() -> %),Symbol,Coef) -> % if Coef has ALGEBRA(FRAC(INT))
--R integrate : (%,Symbol,Coef) -> % if Coef has ALGEBRA(FRAC(INT))
--R integrate : (%,Symbol) -> % if Coef has ALGEBRA(FRAC(INT))
--R log : % -> % if Coef has ALGEBRA(FRAC(INT))
--R monomial : (%,List(Symbol),List(NonNegativeInteger)) -> %
--R monomial : (%,Symbol,NonNegativeInteger) -> %
--R monomial : (Coef,IndexedExponents(Symbol)) -> %
--R monomial : (%,Symbol,IndexedExponents(Symbol)) -> %
--R monomial : (%,List(Symbol),List(IndexedExponents(Symbol))) -> %
--R nthRoot : (%,Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R order : (%,Symbol,NonNegativeInteger) -> NonNegativeInteger
--R order : (%,Symbol) -> NonNegativeInteger
--R pi : () -> % if Coef has ALGEBRA(FRAC(INT))
--R polynomial : (%,NonNegativeInteger,NonNegativeInteger) -> Polynomial(Coef)
--R polynomial : (%,NonNegativeInteger) -> Polynomial(Coef)
--R sec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sqrt : % -> % if Coef has ALGEBRA(FRAC(INT))
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R tan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R tanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R unit? : % -> Boolean if Coef has INTDOM
--R unitCanonical : % -> % if Coef has INTDOM
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %) if Coef has INTDOM
--R
--E 1

)spool
)lisp (bye)

```

— TaylorSeries.help —

```
=====
TaylorSeries examples
=====
```

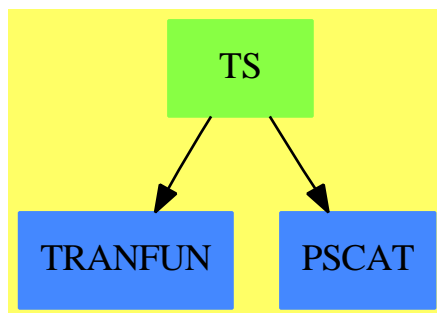
TaylorSeries is a general multivariate Taylor series domain over the ring Coef and with variables of type Symbol.

See Also:

o)show TaylorSeries

—————

TaylorSeries (TS)



See

⇒ “SparseMultivariateTaylorSeries” (SMTS) [20](#) on page [2440](#)

Exports:

0	1	acos	acosh	acot
acoth	acsc	acsch	asec	asech
asin	asinh	associates?	atan	atanh
characteristic	charthRoot	coefficient	coerce	complete
cos	cosh	cot	coth	csc
csch	D	degree	differentiate	eval
exp	exquo	extend	fintegrate	hash
integrate	latex	leadingCoefficient	leadingMonomial	log
map	monomial	monomial?	nthRoot	one?
order	pi	pole?	polynomial	recip
reductum	sample	sec	sech	sin
sinh	sqr	subtractIfCan	tan	tanh
unit?	unitCanonical	unitNormal	variables	zero?
?*?	***?	?+?	?-?	-?
?=?	?^?	?~=?	?/?	

— domain TS TaylorSeries —

```

)abbrev domain TS TaylorSeries
++ Authors: Burge, Watt, Williamson
++ Date Created: 15 August 1988
++ Date Last Updated: 18 May 1991
++ Description:
++ \spadtype{TaylorSeries} is a general multivariate Taylor series domain
++ over the ring Coef and with variables of type Symbol.

TaylorSeries(Coef): Exports == Implementation where
  Coef : Ring
  L ==> List
  NNI ==> NonNegativeInteger
  SMP ==> Polynomial Coef
  StS ==> Stream SMP

Exports ==> MultivariateTaylorSeriesCategory(Coef,Symbol) with
  coefficient: (%,NNI) -> SMP
    ++\spad{coefficient(s, n)} gives the terms of total degree n.
  coerce: Symbol -> %
    ++\spad{coerce(s)} converts a variable to a Taylor series
  coerce: SMP -> %
    ++\spad{coerce(s)} regroups terms of s by total degree
    ++ and forms a series.

if Coef has Algebra Fraction Integer then
  integrate: (%,Symbol,Coef) -> %
    ++\spad{integrate(s,v,c)} is the integral of s with respect
    ++ to v and having c as the constant of integration.
  fintegrate: (() -> %,Symbol,Coef) -> %
    ++\spad{fintegrate(f,v,c)} is the integral of \spad{f()} with respect
    ++ to v and having c as the constant of integration.

```

```

++ The evaluation of \spad{f()} is delayed.

Implementation ==> SparseMultivariateTaylorSeries(Coef,Symbol,SMP) add
Rep := StS -- Below we use the fact that Rep of PS is Stream SMP.

polynomial(s,n) ==
sum : SMP := 0
for i in 0..n while not empty? s repeat
  sum := sum + frst s
  s:= rst s
sum

```

— TS.dotabb —

```

"TS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=TS"]
"TRANFUN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TRANFUN"]
"PSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PSCAT"]
"TS" -> "PSCAT"
"TS" -> "TRANFUN"

```

domain TEX TexFormat

product(product(i*j,i=a..b),j=c..d) fix

The expression prints properly in ascii text but the tex output is incorrect. Originally the input

```
product(product(i*j,i=a..b),j=c..d)
```

prints as

$$(1) \quad PI2(j = c, d, PI2(i = a, b, i j))$$

but now says: The problem is in `[[src/algebra/tex.spad.pamphlet]]` in the list of constants. The code used to read

```

plexOps      : L S := ["SIGMA","SIGMA2","PI","INTSIGN","INDEFINTEGRAL"]$(L S)
plexPrecs    : L I := [ 700, 800,      700,      700]$(L I)

```

it now reads:

— product(product(i*j,i=a..b),j=c..d) fix —

```
plexOps      : L S := ["SIGMA","SIGMA2","PI","PI2","INTSIGN","INDEFINTEGRAL"]$(L S)
plexPrecs    : L I := [ 700, 800, 700, 800 , 700, 700]$(L I)
```

in addition we need to add a line defining `[[PI2]]` in `[[formatPlex]]`:

— **define PI2** —

```
op = "PI2"      => "\prod"
```

— **TexFormat.input** —

```
)set break resume
)sys rm -f TexFormat.output
)spool TexFormat.output
)set message test on
)set message auto off
)clear all
```

```
--S 1 of 11
```

```
(1/2)::TEX
```

```
--R
```

```
--R
```

```
--R (1) ["$$","\frac{1}{2} ","$$"]
```

```
--R
```

Type: TexFormat

```
--E 1
```

```
--S 2 of 11
```

```
(1/(x+5))::TEX
```

```
--R
```

```
--R
```

```
--R (2) ["$$","\frac{1}{{x+5}} ","$$"]
```

```
--R
```

Type: TexFormat

```
--E 2
```

```
--S 3 of 11
```

```
((x+3)/(y-5))::TEX
```

```
--R
```

```
--R
```

```
--R (3) ["$$","\frac{{x+3}}{{y -5}} ","$$"]
```

```
--R
```

Type: TexFormat

```
--E 3
```

```
--S 4 of 11
```

```
)set output fraction horizontal
```

```
--R
```

```
--E 4
```

```
--S 5 of 11
(1/2)::TEX
--R
--R
--R (4) [$$$,"SLASH ","\left(","{1, \: 2} ","\right)","$$"]
--R
--R                                          Type: TexFormat
--E 5
```

```
--S 6 of 11
(1/(x+5))::TEX
--R
--R
--R (5)
--R [$$$,"SLASH ","\left(","{1, \: {\left( x+5 ","\right)}}","\right)","$$"]
--R
--R                                          Type: TexFormat
--E 6
```

```
--S 7 of 11
)set output mathml on
--R
--E 7
```

```
--S 8 of 11
1/2
--R
--R
--R (6) 1/2
--R<math xmlns="http://www.w3.org/1998/Math/MathML" mathsize="big" display="block">
--R<mrow><mn>1</mn><mo>/</mo><mn>2</mn></mrow>
--R</math>
--R
--R                                          Type: Fraction(Integer)
--E 8
```

```
--S 9 of 11
1/(x+5)
--R
--R
--R (7) 1/(x + 5)
--R<math xmlns="http://www.w3.org/1998/Math/MathML" mathsize="big" display="block">
--R<mrow><mn>1</mn><mo>/</mo><mrow><mo>(</mo><mi>x</mi><mo>+</mo><mn>5</mn></mrow></mrow></math>
--R
--R
--R                                          Type: Fraction(Polynomial(Integer))
--E 9
```

```
--S 10 of 11
(x+3)/(y-5)
--R
```

```

--R
--R      (8)  (x + 3)/(y - 5)
--R<math xmlns="http://www.w3.org/1998/Math/MathML" mathsize="big" display="block">
--R<mrow><mrow><mo></mo><mi>x</mi><mo>+</mo><mn>3</mn><mo></mo></mrow><mo>/</mo><mrow><mo></mo><mi>y</mi><mo>-</mo><mn>5</mn><mo></mo></mrow></math>
--R
--R
--R                                          Type: Fraction(Polynomial(Integer))
--E 10

```

```

--S 11 of 11
)show TexFormat
--R
--R TexFormat  is a domain constructor
--R Abbreviation for TexFormat is TEX
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for TEX
--R
--R----- Operations -----
--R ?=? : (%,% ) -> Boolean          coerce : OutputForm -> %
--R coerce : % -> OutputForm          display : % -> Void
--R display : (%,Integer) -> Void      epilogue : % -> List(String)
--R hash : % -> SingleInteger          latex : % -> String
--R new : () -> %                      prologue : % -> List(String)
--R tex : % -> List(String)            ?~=? : (%,% ) -> Boolean
--R convert : (OutputForm,Integer,OutputForm) -> %
--R convert : (OutputForm,Integer) -> %
--R setEpilogue! : (%,List(String)) -> List(String)
--R setPrologue! : (%,List(String)) -> List(String)
--R setTex! : (%,List(String)) -> List(String)
--R
--E 11

)spool
)lisp (bye)

```

— TexFormat.help —

=====

TexFormat examples

=====

You can ask Axiom to show latex output. In particular, this can be used for complex output.

(1/2)::TEX

["\$\$", "\frac{1}{2} ", "\$\$"]

```
(1/(x+5))::TEX
```

```
["$${\frac{1}{{x+5}}} ", "$$"]
```

```
((x+3)/(y-5))::TEX
```

```
["$${\frac{{x+3}}{{y-5}}} ", "$$"]
```

We can change the fraction display so it is horizontal:

```
)set output fraction horizontal
```

```
(1/2)::TEX
```

```
["$${\left(1, \right)} ", "$$"]
```

```
(1/(x+5))::TEX
```

```
["$${\left(1, \right)} ", "$$"]
```

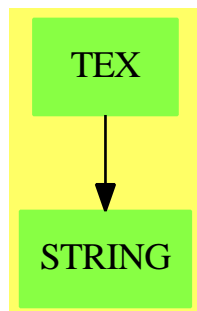
```
((x+3)/(y-5))::TEX
```

```
["$${\left({x+3}, \right)} ", "$$"]
```

See Also:

- o)show TexFormat

TexFormat (TEX)



Exports:

coerce	convert	display	epilogue	hash
latex	new	prologue	setEpilogue!	setPrologue!
setTex!	tex	convert	?=?	?~=?

— domain TEX TexFormat —

```
)abbrev domain TEX TexFormat
++ Author: Robert S. Sutor
++ Date Created: 1987 through 1992
++ Description:
++ \spadtype{TexFormat} provides a coercion from \spadtype{OutputForm} to
++ \TeX{} format. The particular dialect of \TeX{} used is \LaTeX{}.
++ The basic object consists of three parts: a prologue, a
++ tex part and an epilogue. The functions \spadfun{prologue},
++ \spadfun{tex} and \spadfun{epilogue} extract these parts,
++ respectively. The main guts of the expression go into the tex part.
++ The other parts can be set (\spadfun{setPrologue!},
++ \spadfun{setEpilogue!}) so that contain the appropriate tags for
++ printing. For example, the prologue and epilogue might simply
++ contain ‘\verb+[+’ and ‘\verb+]+’, respectively, so that
++ the TeX section will be printed in LaTeX display math mode.
```

```
TexFormat(): public == private where
E      ==> OutputForm
I      ==> Integer
L      ==> List
S      ==> String
US     ==> UniversalSegment(Integer)
```

```
public == SetCategory with
coerce: E -> $
++ coerce(o) changes o in the standard output format to TeX
++ format.
convert: (E,I) -> $
++ convert(o,step) changes o in standard output format to
++ TeX format and also adds the given step number. This is useful
++ if you want to create equations with given numbers or have the
++ equation numbers correspond to the interpreter step numbers.
convert: (E,I,E) -> $
++ convert(o,step,type) changes o in standard output format to
++ TeX format and also adds the given step number and type. This
++ is useful if you want to create equations with given numbers
++ or have the equation numbers correspond to the interpreter step
++ numbers.
display: ($, I) -> Void
++ display(t,width) outputs the TeX formatted code t so that each
++ line has length less than or equal to \spadvar{width}.
display: $ -> Void
++ display(t) outputs the TeX formatted code t so that each
```



```

++ line has length less than or equal to the value set by
++ the system command \spadsyscom{set output length}.
epilogue: $ -> L S
++ epilogue(t) extracts the epilogue section of a TeX form t.
tex:      $ -> L S
++ tex(t) extracts the TeX section of a TeX form t.
new:      () -> $
++ new() create a new, empty object. Use \spadfun{setPrologue!},
++ \spadfun{setTex!} and \spadfun{setEpilogue!} to set the various
++ components of this object.
prologue: $ -> L S
++ prologue(t) extracts the prologue section of a TeX form t.
setEpilogue!: ($, L S) -> L S
++ setEpilogue!(t,strings) sets the epilogue section of a TeX form t
++ to strings.
setTex!: ($, L S) -> L S
++ setTex!(t,strings) sets the TeX section of a TeX form t to strings.
setPrologue!: ($, L S) -> L S
++ setPrologue!(t,strings) sets the prologue section of a TeX form t
++ to strings.

private == add
import OutputForm
import Character
import Integer
import List OutputForm
import List String

Rep := Record(prolog : L S, TeX : L S, epilogs : L S)

-- local variables declarations and definitions

expr: E
prec,opPrec: I
str: S
blank      : S := " \ "

maxPrec    : I    := 1000000
minPrec    : I    := 0

unaryOps   : L S := ["-", "^"]$(L S)
unaryPrecs : L I := [700,260]$(L I)

-- the precedence of / in the following is relatively low because
-- the bar obviates the need for parentheses.
binaryOps  : L S := ["+>", "|", "**", "/", "<", ">", "=", "OVER"]$(L S)
binaryPrecs : L I := [0,0,900, 700,400,400,400, 700]$(L I)

naryOps    : L S := ["-", "+", "*", blank, ",", ";", " ", "ROW", "",
" \cr ", "&", " \\" ]$(L S)

```

```

naryPrecs      : L I := [700,700,800, 800,110,110, 0, 0, 0,
                        0, 0, 0]$(L I)
naryNGOps      : L S := ["ROW",&"]$(L S)

\getchunk{product(product(i*j,i=a..b),j=c..d) fix}

specialOps     : L S := ["MATRIX","BRACKET","BRACE","CONCATB","VCONCAT", _
                        "AGGLST","CONCAT","OVERBAR","ROOT","SUB","TAG", _
                        "SUPERSUB","ZAG","AGGSET","SC","PAREN", _
                        "SEGMENT","QUOTE","theMap" ]

-- the next two lists provide translations for some strings for
-- which TeX provides special macros.

specialStrings : L S :=
  ["cos", "cot", "csc", "log", "sec", "sin", "tan",
   "cosh", "coth", "csch", "sech", "sinh", "tanh",
   "acos","asin","atan","erf","...","$","infinity"]
specialStringsInTeX : L S :=
  ["\cos","\cot","\csc","\log","\sec","\sin","\tan",
   "\cosh","\coth","\csch","\sech","\sinh","\tanh",
   "\arccos","\arcsin","\arctan","\erf","\ldots","\$","\infty"]

-- local function signatures

addBraces:      S -> S
addBrackets:    S -> S
group:          S -> S
formatBinary:   (S,L E, I) -> S
formatFunction: (S,L E, I) -> S
formatMatrix:   L E -> S
formatNary:     (S,L E, I) -> S
formatNaryNoGroup: (S,L E, I) -> S
formatNullary:  S -> S
formatPlex:     (S,L E, I) -> S
formatSpecial:  (S,L E, I) -> S
formatUnary:    (S, E, I) -> S
formatTex:      (E,I) -> S
newWithNum:     I -> $
parenthesize:   S -> S
precondition:   E -> E
postcondition:  S -> S
splitLong:      (S,I) -> L S
splitLong1:     (S,I) -> L S
stringify:      E -> S
ungroup:        S -> S

-- public function definitions

new() : $ ==

```

```

--      [[\"\"$(L S), [\"\"]$(L S), [\"\\\"]$(L S)]$Rep
      [\"$$\"$(L S), [\"\"]$(L S), [\"$$\"$(L S)]$Rep

newWithNum(stepNum: I) : $ ==
--      num : S := concat(\"%AXIOM STEP NUMBER: \",string(stepNum)$S)
--      [[\"\"$(L S), [\"\"]$(L S), [\"\\\"]\",num]$(L S)]$Rep
      num : S := concat(concat(\"\\leqno(\" ,string(stepNum)$S),\")\")$S
      [\"$$\"$(L S), [\"\"]$(L S), [num,\"$$\"$(L S)]$Rep

coerce(expr : E): $ ==
      f : $ := new()$$
      f.TeX := [postcondition
        formatTex(precondition expr, minPrec)]$(L S)
      f

convert(expr : E, stepNum : I): $ ==
      f : $ := newWithNum(stepNum)
      f.TeX := [postcondition
        formatTex(precondition expr, minPrec)]$(L S)
      f

display(f : $, len : I) ==
      s,t : S
      for s in f.prolog repeat sayTeX$Lisp s
      for s in f.TeX repeat
        for t in splitLong(s, len) repeat sayTeX$Lisp t
      for s in f.epilog repeat sayTeX$Lisp s
      void()$Void

display(f : $) ==
      display(f, _$LINELENGTH$Lisp pretend I)

prologue(f : $) == f.prolog
tex(f : $) == f.TeX
epilogue(f : $) == f.epilog

setPrologue!(f : $, l : L S) == f.prolog := l
setTex!(f : $, l : L S) == f.TeX := l
setEpilogue!(f : $, l : L S) == f.epilog := l

coerce(f : $): E ==
      s,t : S
      l : L S := nil
      for s in f.prolog repeat l := concat(s,l)
      for s in f.TeX repeat
        for t in splitLong(s, (_$LINELENGTH$Lisp pretend Integer) - 4) repeat
          l := concat(t,l)
      for s in f.epilog repeat l := concat(s,l)
      (reverse l) :: E

```

```

-- local function definitions

ungroup(str: S): S ==
  len : I := #str
  len < 2 => str
  lbrace : Character := char "{"
  rbrace : Character := char "}"
  -- drop leading and trailing braces
  if (str.1 = $Character lbrace) and (str.len = $Character rbrace) then
    u : US := segment(2,len-1)$US
    str := str.u
  str

postcondition(str: S): S ==
  str := ungroup str
  len : I := #str
  plus : Character := char "+"
  minus: Character := char "-"
  len < 4 => str
  for i in 1..(len-1) repeat
    if (str.i = $Character plus) and (str.(i+1) = $Character minus)
      then setelt(str,i,char " ")$S
  str

stringify expr == (object2String$Lisp expr) pretend S

lineConcat( line : S, lines: L S ) : L S ==
  length := #line

  if ( length > 0 ) then
    -- If the last character is a backslash then split at "\".
    -- Reinstate the blank.

    if (line.length = char "\" ) then line := concat(line, " ")

    -- Remark: for some reason, "\" at the beginning
    -- of a line has the "\" erased when printed

    if ( line.1 = char "%" ) then line := concat(" \", line)
    else if ( line.1 = char "\" ) and length > 1 and ( line.2 = char "%" ) then
      line := concat(" ", line)

    lines := concat(line,lines)$List(S)
  lines

splitLong(str : S, len : I): L S ==
  -- this blocks into lines
  if len < 20 then len := _$LINELENGTH$Lisp
  splitLong1(str, len)

```

```

splitLong1(str : S, len : I) ==
-- We first build the list of lines backwards and then we
-- reverse it.

l : List S := nil
s : S := ""
ls : I := 0
ss : S
lss : I
for ss in split(str, char " ") repeat
  -- have the newline macro end a line (even if it means the line
  -- is slightly too long)

  ss = "\\\" =>
    l := lineConcat( concat(s,ss), l )
    s := ""
    ls := 0

  lss := #ss

-- place certain tokens on their own lines for clarity

ownLine : Boolean :=
  u : US := segment(1,4)$US
  (lss > 3) and ("end" = ss.u) => true
  u      := segment(1,5)$US
  (lss > 4) and ("left" = ss.u) => true
  u      := segment(1,6)$US
  (lss > 5) and (("right" = ss.u) or ("begin" = ss.u)) => true
  false

if ownLine or (ls + lss > len) then
  if not empty? s then l := lineConcat( s, l )
  s := ""
  ls := 0

ownLine or lss > len => l := lineConcat( ss, l )

(lss = 1) and (ss.1 = char "\") =>
  ls := ls + lss + 2
  s := concat(s,concat(ss," ")$S)$S

ls := ls + lss + 1
s := concat(s,concat(ss," ")$S)$S

if ls > 0 then l := lineConcat( s, l )

reverse l

group str ==

```

```

concat ["{",str,"}"]

addBraces str ==
  concat ["\left{ ",str," \right\}"]

addBrackets str ==
  concat ["\left[ ",str," \right]"]

parenthesize str ==
  concat ["\left( ",str," \right)"]

precondition expr ==
  outputTran$Lisp expr

formatSpecial(op : S, args : L E, prec : I) : S ==
  arg : E
  prescript : Boolean := false
  op = "theMap" => "\mbox{theMap(...)}"
  op = "AGGLST" =>
    formatNary(",",args,prec)
  op = "AGGSET" =>
    formatNary(";",args,prec)
  op = "TAG" =>
    group concat [formatTex(first args,prec),
                  "\rightarrow",
                  formatTex(second args,prec)]
  op = "VCONCAT" =>
    group concat("\begin{array}{c}",
                  concat(concat([concat(formatTex(u, minPrec),"\\")
                                for u in args]::L S),
                          "\end{array}"))
  op = "CONCATB" =>
    formatNary(" ",args,prec)
  op = "CONCAT" =>
    formatNary("",args,minPrec)
  op = "QUOTE" =>
    group concat("\tt ' ",formatTex(first args, minPrec))
  op = "BRACKET" =>
    group addBrackets ungroup formatTex(first args, minPrec)
  op = "BRACE" =>
    group addBraces ungroup formatTex(first args, minPrec)
  op = "PAREN" =>
    group parenthesize ungroup formatTex(first args, minPrec)
  op = "OVERBAR" =>
    null args => ""
    group concat ["\overline ",formatTex(first args, minPrec)]
  op = "ROOT" =>
    null args => ""
    tmp : S := group formatTex(first args, minPrec)
    null rest args => group concat ["\sqrt ",tmp]

```

```

group concat
  ["\root ",group formatTex(first rest args, minPrec)," \of ",tmp]
op = "SEGMENT" =>
  tmp : S := concat [formatTex(first args, minPrec),".."]
  group
    null rest args => tmp
    concat [tmp,formatTex(first rest args, minPrec)]
op = "SUB" =>
  group concat [formatTex(first args, minPrec)," \sb ",
    formatSpecial("AGGLST",rest args,minPrec)]
op = "SUPERSUB" =>
  -- variable name
  form : List S := [formatTex(first args, minPrec)]
  -- subscripts
  args := rest args
  null args => concat(form)$S
  tmp : S := formatTex(first args, minPrec)
  if (tmp ^="") and (tmp ^="{}") and (tmp ^=" ") then
    form := append(form,[" \sb ",group tmp])$(List S)
  -- superscripts
  args := rest args
  null args => group concat(form)$S
  tmp : S := formatTex(first args, minPrec)
  if (tmp ^="") and (tmp ^="{}") and (tmp ^=" ") then
    form := append(form,[" \sp ",group tmp])$(List S)
  -- presuperscripts
  args := rest args
  null args => group concat(form)$S
  tmp : S := formatTex(first args, minPrec)
  if (tmp ^="") and (tmp ^="{}") and (tmp ^=" ") then
    form := append([" \sp ",group tmp],form)$(List S)
    prescript := true
  -- presubscripts
  args := rest args
  null args =>
    group concat
      prescript => cons("{",form)
      form
  tmp : S := formatTex(first args, minPrec)
  if (tmp ^="") and (tmp ^="{}") and (tmp ^=" ") then
    form := append([" \sb ",group tmp],form)$(List S)
    prescript := true
  group concat
    prescript => cons("{",form)
    form
op = "SC" =>
  -- need to handle indentation someday
  null args => ""
  tmp := formatNaryNoGroup(" \\", args, minPrec)
  group concat ["\begin{array}{l} ",tmp," \end{array} "]

```

```

op = "MATRIX" => formatMatrix rest args
op = "ZAG" =>
  concat [" \zag{",formatTex(first args, minPrec),"}{",
    formatTex(first rest args,minPrec),"}"]
concat ["not done yet for ",op]

formatPlex(op : S, args : L E, prec : I) : S ==
  hold : S
  p : I := position(op,plexOps)
  p < 1 => error "unknown Tex unary op"
  opPrec := plexPrecs.p
  n : I := #args
  (n ^= 2) and (n ^= 3) => error "wrong number of arguments for plex"
  s : S :=
    op = "SIGMA"    => "\sum"
    op = "SIGMA2"   => "\sum"
    op = "PI"       => "\prod"
\getchunk{define PI2}
    op = "INTSIGN"  => "\int"
    op = "INDEFINTEGRAL" => "\int"
    "???"
  hold := formatTex(first args,minPrec)
  args := rest args
  if op ^= "INDEFINTEGRAL" then
    if hold ^= "" then
      s := concat [s," \sb",group concat ["\displaystyle ",hold]]
    if not null rest args then
      hold := formatTex(first args,minPrec)
      if hold ^= "" then
        s := concat [s," \sp",group concat ["\displaystyle ",hold]]
      args := rest args
      s := concat [s," ",formatTex(first args,minPrec)]
    else
      hold := group concat [hold," ",formatTex(first args,minPrec)]
      s := concat [s," ",hold]
  if opPrec < prec then s := parenthesize s
  group s

formatMatrix(args : L E) : S ==
  -- format for args is [[ROW ...],[ROW ...],[ROW ...]]
  -- generate string for formatting columns (centered)
  cols : S := "{"
  for i in 2..#(first(args) pretend L E) repeat
    cols := concat(cols,"c")
  cols := concat(cols,"} ")
  group addBrackets concat
    ["\begin{array}",cols,formatNaryNoGroup(" \\\ ",args,minPrec),
      " \end{array} "]

formatFunction(op : S, args : L E, prec : I) : S ==

```



```

group concat [op, " ", parenthesize formatNary(" ", args, minPrec)]

formatNullary(op : S) ==
  op = "NOTHING" => ""
  group concat [op, "()"]

formatUnary(op : S, arg : E, prec : I) ==
  p : I := position(op, unaryOps)
  p < 1 => error "unknown Tex unary op"
  opPrec := unaryPrecs.p
  s : S := concat [op, formatTex(arg, opPrec)]
  opPrec < prec => group parenthesize s
  op = "-" => s
  group s

formatBinary(op : S, args : L E, prec : I) : S ==
  p : I := position(op, binaryOps)
  p < 1 => error "unknown Tex binary op"
  op :=
    op = "|"      => " \mid "
    op = "**"     => " \sp "
    op = "/"      => " \over "
    op = "OVER"   => " \over "
    op = "+->"    => " \mapsto "
    op
  opPrec := binaryPrecs.p
  s : S := formatTex(first args, opPrec)
  if op = " \over " then
    s := concat [" \frac{", s, "}{", formatTex(first rest args, opPrec), "}"]
  else if op = " \sp " then
    s := concat [s, "^", formatTex(first rest args, opPrec)]
  else
    s := concat [s, op, formatTex(first rest args, opPrec)]
  group
  op = " \over " => s
  opPrec < prec => parenthesize s
  s

formatNary(op : S, args : L E, prec : I) : S ==
  group formatNaryNoGroup(op, args, prec)

formatNaryNoGroup(op : S, args : L E, prec : I) : S ==
  null args => ""
  p : I := position(op, naryOps)
  p < 1 => error "unknown Tex nary op"
  op :=
    op = ","      => " , \: "
    op = ";"      => " ; \: "
    op = "*"      => blank
    op = " "      => " \ "

```

```

    op = "ROW"    => " & "
    op
  l : L S := nil
  opPrec := naryPrecs.p
  for a in args repeat
    l := concat(op,concat(formatTex(a,opPrec),l)$L(S))$L(S)
  s : S := concat reverse rest l
  opPrec < prec => parenthesize s
  s

formatTex(expr,prec) ==
  i,len : Integer
  intSplitLen : Integer := 20
  ATOM(expr)$Lisp pretend Boolean =>
    str := stringify expr
    len := #str
    INTEGERP$Lisp expr =>
      i := expr pretend Integer
      if (i < 0) or (i > 9)
      then
        group
          nstr : String := ""
          -- insert some blanks into the string, if too long
          while ((len := #str) > intSplitLen) repeat
            nstr := concat [nstr," ",
              elt(str,segment(1,intSplitLen)$US)]
            str := elt(str,segment(intSplitLen+1)$US)
          empty? nstr => str
          nstr :=
            empty? str => nstr
            concat [nstr," ",str]
            elt(nstr,segment(2)$US)
        else str
      str = "%pi" => "\pi"
      str = "%e"  => "e"
      str = "%i"  => "i"
      len > 1 and str.1 = char "%" and str.2 = char "%" =>
        u : US := segment(3,len)$US
        concat(" \%\%",str.u)
      len > 0 and str.1 = char "%" => concat(" \",str)
      len > 1 and digit? str.1 => group str -- should handle floats
      len > 0 and str.1 = char "_" =>
        concat(concat(" \mbox{\tt ",str),"} ")
      len = 1 and str.1 = char " " => "{\ }"
      (i := position(str,specialStrings)) > 0 =>
        specialStringsInTeX.i
      (i := position(char " ",str)) > 0 =>
        -- We want to preserve spacing, so use a roman font.
        concat(concat(" \mbox{\rm ",str),"} ")
      str

```

```

l : L E := (expr pretend L E)
null l => blank
op : S := stringify first l
args : L E := rest l
nargs : I := #args

-- special cases
member?(op, specialOps) => formatSpecial(op,args,prec)
member?(op, plexOps)    => formatPlex(op,args,prec)

-- nullary case
0 = nargs => formatNullary op

-- unary case
(1 = nargs) and member?(op, unaryOps) =>
  formatUnary(op, first args, prec)

-- binary case
(2 = nargs) and member?(op, binaryOps) =>
  formatBinary(op, args, prec)

-- nary case
member?(op,naryNGOps) => formatNaryNoGroup(op,args, prec)
member?(op,naryOps) => formatNary(op,args, prec)
op := formatTex(first l,minPrec)
formatFunction(op,args,prec)

```

— TEX.dotabb —

```

"TEX" [color="#88FF44",href="bookvol10.3.pdf#nameddest=TEX"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"TEX" -> "STRING"

```

domain TEXTFILE TextFile

— TextFile.input —

```

)set break resume
)sys rm -f TextFile.output
)spool TextFile.output

```

```

)set message test on
)set message auto off
)clear all
--S 1 of 10
f1: TextFile := open("/etc/group", "input")
--R
--R
--R (1) "/etc/group"
--R
Type: TextFile
--E 1

--S 2 of 10
f2: TextFile := open("MOTD", "output")
--R
--R
--R (2) "MOTD"
--R
Type: TextFile
--E 2

--S 3 of 10
l := readLine! f1
--R
--R
--R (3) "ROOT:x:0:"
--R
Type: String
--E 3

--S 4 of 10
writeLine!(f2, upperCase l)
--R
--R
--R (4) "ROOT:X:0:"
--R
Type: String
--E 4

--S 5 of 10
while not endOfFile? f1 repeat
  s := readLine! f1
  writeLine!(f2, upperCase s)
--R
--R
Type: Void
--E 5

--S 6 of 10
close! f1
--R
--R
--R (6) "/etc/group"
--R
Type: TextFile
--E 6
```

```

--S 7 of 10
write!(f2, "-The-")
--R
--R
--R (7) "-The-"
--R
--R                                          Type: String
--E 7

--S 8 of 10
write!(f2, "-End-")
--R
--R
--R (8) "-End-"
--R
--R                                          Type: String
--E 8

--S 9 of 10
writeLine! f2
--R
--R
--R (9) ""
--R
--R                                          Type: String
--E 9

--S 10 of 10
close! f2
--R
--R
--R (10) "MOTD"
--R
--R                                          Type: TextFile
--E 10
)system rm -f MOTD
)spool
)lisp (bye)

```

— TextFile.help —

=====

TextFile examples

=====

The domain TextFile allows Axiom to read and write character data and exchange text with other programs. This type behaves in Axiom much like a File of strings, with additional operations to cause new lines. We give an example of how to produce an upper case copy of a file.

This is the file from which we read the text.

```
f1: TextFile := open("/etc/group", "input")
"/etc/group"
Type: TextFile
```

This is the file to which we write the text.

```
f2: TextFile := open("/tmp/MOTD", "output")
"MOTD"
Type: TextFile
```

Entire lines are handled using the readLine and writeLine operations.

```
l := readLine! f1
"root:x:0:root"
Type: String

writeLine!(f2, upperCase l)
"ROOT:X:0:ROOT"
Type: String
```

Use the endOfFile? operation to check if you have reached the end of the file.

```
while not endOfFile? f1 repeat
  s := readLine! f1
  writeLine!(f2, upperCase s)
Type: Void
```

The file f1 is exhausted and should be closed.

```
close! f1
"/etc/group"
Type: TextFile
```

It is sometimes useful to write lines a bit at a time. The write operation allows this.

```
write!(f2, "-The-")
"-The-"
Type: String

write!(f2, "-End-")
"-End-"
Type: String
```

This ends the line. This is done in a machine-dependent manner.

```
writeLine! f2
""
```

Type: String

```
close! f2
"MTD"
```

Type: TextFile

Finally, clean up.

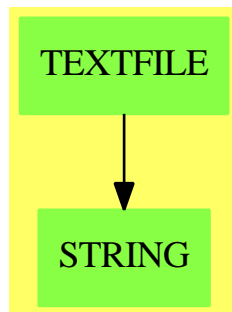
```
)system rm /tmp/MTD
```

See Also:

- o)help File
- o)help KeyedAccessFile
- o)help Library
- o)show TextFile

—————→

TextFile (TEXTFILE)



See

- ⇒ “File” (FILE) [7](#) on page [802](#)
- ⇒ “BinaryFile” (BINFILE) [3](#) on page [295](#)
- ⇒ “KeyedAccessFile” (KAFILE) [12](#) on page [1413](#)
- ⇒ “Library” (LIB) [13](#) on page [1428](#)

Exports:

close!	coerce	endOfFile?	hash	iomode
latex	name	open	read!	readIfCan!
readLine!	readLineIfCan!	reopen!	write!	writeLine!
?=?	?~=?			

— domain TEXTFILE TextFile —

```

)abbrev domain TEXTFILE TextFile
++ Author: Stephen M. Watt
++ Date Created: 1985
++ Date Last Updated: June 4, 1991
++ Description:
++ This domain provides an implementation of text files. Text is stored
++ in these files using the native character set of the computer.

TextFile: Cat == Def where
  StreamName ==> Union(FileName, "console")

  Cat == FileCategory(FileName, String) with
    writeLine_!: (%, String) -> String
      ++ writeLine!(f,s) writes the contents of the string s
      ++ and finishes the current line in the file f.
      ++ The value of s is returned.

    writeLine_!: % -> String
      ++ writeLine!(f) finishes the current line in the file f.
      ++ An empty string is returned. The call \spad{writeLine!(f)} is
      ++ equivalent to \spad{writeLine!(f,"")}.

    readLine_!: % -> String
      ++ readLine!(f) returns a string of the contents of a line from
      ++ the file f.

    readLineIfCan_!: % -> Union(String, "failed")
      ++ readLineIfCan!(f) returns a string of the contents of a line from
      ++ file f, if possible. If f is not readable or if it is
      ++ positioned at the end of file, then \spad{"failed"} is returned.

    readIfCan_!: % -> Union(String, "failed")
      ++ readIfCan!(f) returns a string of the contents of a line from
      ++ file f, if possible. If f is not readable or if it is
      ++ positioned at the end of file, then \spad{"failed"} is returned.

    endOfFile?: % -> Boolean
      ++ endOfFile?(f) tests whether the file f is positioned after the
      ++ end of all text. If the file is open for output, then
      ++ this test is always true.

  Def == File(String) add
    FileState ==> SExpression

    Rep := Record(fileName:  FileName,  _
                  fileState: FileState, _
                  fileIOmode: String)

    read_! f      == readLine_! f
    readIfCan_! f == readLineIfCan_! f

```



```

readLine_! f ==
  f.fileIOmode ^= "input" => error "File not in read state"
  s: String := read_line(f.fileState)$Lisp
  PLACEP(s)$Lisp => error "End of file"
  s
readLineIfCan_! f ==
  f.fileIOmode ^= "input" => error "File not in read state"
  s: String := read_line(f.fileState)$Lisp
  PLACEP(s)$Lisp => "failed"
  s
write_!(f, x) ==
  f.fileIOmode ^= "output" => error "File not in write state"
  PRINTEXP(x, f.fileState)$Lisp
  x
writeLine_! f ==
  f.fileIOmode ^= "output" => error "File not in write state"
  TERPRI(f.fileState)$Lisp
  ""
writeLine_!(f, x) ==
  f.fileIOmode ^= "output" => error "File not in write state"
  PRINTEXP(x, f.fileState)$Lisp
  TERPRI(f.fileState)$Lisp
  x
endOfFile? f ==
  f.fileIOmode = "output" => false
  (EOFP(f.fileState)$Lisp pretend Boolean) => true
  false

```

— TEXTFILE.dotabb —

```

"TEXTFILE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=TEXTFILE"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"TEXTFILE" -> "STRING"

```

domain SYMS TheSymbolTable

— TheSymbolTable.input —

```

)set break resume

```

```

)sys rm -f TheSymbolTable.output
)spool TheSymbolTable.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TheSymbolTable
--R
--R TheSymbolTable is a domain constructor
--R Abbreviation for TheSymbolTable is SYMS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SYMS
--R
--R----- Operations -----
--R clearTheSymbolTable : () -> Void      coerce : % -> OutputForm
--R currentSubProgram : () -> Symbol      empty : () -> %
--R endSubProgram : () -> Symbol          newSubProgram : Symbol -> Void
--R printHeader : () -> Void              printHeader : Symbol -> Void
--R printHeader : (Symbol,%) -> Void      printTypes : Symbol -> Void
--R showTheSymbolTable : () -> %
--R argumentList! : List(Symbol) -> Void
--R argumentList! : (Symbol,List(Symbol)) -> Void
--R argumentList! : (Symbol,List(Symbol),%) -> Void
--R argumentListOf : (Symbol,%) -> List(Symbol)
--R clearTheSymbolTable : Symbol -> Void
--R declare! : (Symbol,FortranType,Symbol) -> FortranType
--R declare! : (Symbol,FortranType) -> FortranType
--R declare! : (List(Symbol),FortranType,Symbol,%) -> FortranType
--R declare! : (Symbol,FortranType,Symbol,%) -> FortranType
--R returnType! : Union(fst: FortranScalarType,void: void) -> Void
--R returnType! : (Symbol,Union(fst: FortranScalarType,void: void)) -> Void
--R returnType! : (Symbol,Union(fst: FortranScalarType,void: void),%) -> Void
--R returnTypeOf : (Symbol,%) -> Union(fst: FortranScalarType,void: void)
--R symbolTableOf : (Symbol,%) -> SymbolTable
--R
--E 1

)spool
)lisp (bye)

```

— TheSymbolTable.help —

```

=====
TheSymbolTable examples
=====

```

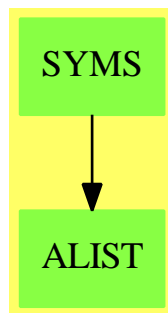
Creates and manipulates one global symbol table for FORTRAN code generation, containing details of types, dimensions, and argument lists.

See Also:

o)show TheSymbolTable

—————

TheSymbolTable (SYMS)



See

⇒ “FortranScalarType” (FST) [7](#) on page [962](#)
 ⇒ “FortranType” (FT) [7](#) on page [971](#)
 ⇒ “SymbolTable” (SYMTAB) [20](#) on page [2648](#)

Exports:

argumentList!	argumentListOf	clearTheSymbolTable	coerce	currentSubProgram
declare!	empty	endSubProgram	newSubProgram	printHeader
printTypes	returnType!	returnTypeOf	showTheSymbolTable	symbolTableOf

— domain SYMS TheSymbolTable —

```

)abbrev domain SYMS TheSymbolTable
++ Author: Mike Dewar
++ Date Created:  October 1992
++ Description:
++ Creates and manipulates one global symbol table for FORTRAN
++ code generation, containing details of types, dimensions, and argument
++ lists.
  
```

TheSymbolTable() : Exports == Implementation where

```

S    ==> Symbol
FST  ==> FortranScalarType
FSTU ==> Union(fst:FST,void:"void")
  
```

```

Exports == CoercibleTo OutputForm with
  showTheSymbolTable : () -> $
    ++ showTheSymbolTable() returns the current symbol table.
  clearTheSymbolTable : () -> Void
    ++ clearTheSymbolTable() clears the current symbol table.
  clearTheSymbolTable : Symbol -> Void
    ++ clearTheSymbolTable(x) removes the symbol x from the table
  declare! : (Symbol, FortranType, Symbol, $) -> FortranType
    ++ declare!(u,t,asp,tab) declares the parameter u of subprogram asp
    ++ to have type t in symbol table tab.
  declare! : (List Symbol, FortranType, Symbol, $) -> FortranType
    ++ declare!(u,t,asp,tab) declares the parameters u of subprogram asp
    ++ to have type t in symbol table tab.
  declare! : (Symbol, FortranType) -> FortranType
    ++ declare!(u,t) declares the parameter u to have type t in the
    ++ current level of the symbol table.
  declare! : (Symbol, FortranType, Symbol) -> FortranType
    ++ declare!(u,t,asp) declares the parameter u to have type t in asp.
  newSubProgram : Symbol -> Void
    ++ newSubProgram(f) asserts that from now on type declarations are part
    ++ of subprogram f.
  currentSubProgram : () -> Symbol
    ++ currentSubProgram() returns the name of the current subprogram being
    ++ processed
  endSubProgram : () -> Symbol
    ++ endSubProgram() asserts that we are no longer processing the current
    ++ subprogram.
  argumentList! : (Symbol, List Symbol, $) -> Void
    ++ argumentList!(f,l,tab) declares that the argument list for subprogram f
    ++ in symbol table tab is l.
  argumentList! : (Symbol, List Symbol) -> Void
    ++ argumentList!(f,l) declares that the argument list for subprogram f in
    ++ the global symbol table is l.
  argumentList! : List Symbol -> Void
    ++ argumentList!(l) declares that the argument list for the current
    ++ subprogram in the global symbol table is l.
  returnType! : (Symbol, FSTU, $) -> Void
    ++ returnType!(f,t,tab) declares that the return type of subprogram f in
    ++ symbol table tab is t.
  returnType! : (Symbol, FSTU) -> Void
    ++ returnType!(f,t) declares that the return type of subprogram f in
    ++ the global symbol table is t.
  returnType! : FSTU -> Void
    ++ returnType!(t) declares that the return type of the current subprogram
    ++ in the global symbol table is t.
  printHeader : (Symbol, $) -> Void
    ++ printHeader(f,tab) produces the FORTRAN header for subprogram f in
    ++ symbol table tab on the current FORTRAN output stream.
  printHeader : Symbol -> Void

```

```

    ++ printHeader(f) produces the FORTRAN header for subprogram f in
    ++ the global symbol table on the current FORTRAN output stream.
printHeader : () -> Void
    ++ printHeader() produces the FORTRAN header for the current subprogram in
    ++ the global symbol table on the current FORTRAN output stream.
printTypes: Symbol -> Void
    ++ printTypes(tab) produces FORTRAN type declarations from tab, on the
    ++ current FORTRAN output stream
empty : () -> $
    ++ empty() creates a new, empty symbol table.
returnTypeOf : (Symbol,$) -> FSTU
    ++ returnTypeOf(f,tab) returns the type of the object returned by f
argumentListOf : (Symbol,$) -> List(Symbol)
    ++ argumentListOf(f,tab) returns the argument list of f
symbolTableOf : (Symbol,$) -> SymbolTable
    ++ symbolTableOf(f,tab) returns the symbol table of f

Implementation == add

Entry : Domain := Record(symtab:SymbolTable, _
                        returnType:FSTU, _
                        argList:List Symbol)

Rep := Table(Symbol,Entry)

-- These are the global variables we want to update:
theSymbolTable : $ := empty()$Rep
currentSubProgramName : Symbol := MAIN

newEntry():Entry ==
    construct(empty()$SymbolTable,["void"]$FSTU,[]::List(Symbol))$Entry

checkIfEntryExists(name:Symbol,tab:$) : Void ==
    key?(name,tab) => void()$Void
    setelt(tab,name,newEntry())$Rep
    void()$Void

returnTypeOf(name:Symbol,tab:$):FSTU ==
    elt(elt(tab,name)$Rep,returnType)$Entry

argumentListOf(name:Symbol,tab:$):List(Symbol) ==
    elt(elt(tab,name)$Rep,argList)$Entry

symbolTableOf(name:Symbol,tab:$):SymbolTable ==
    elt(elt(tab,name)$Rep,symtab)$Entry

coerce(u:$):OutputForm ==
    coerce(u)$Rep

showTheSymbolTable():$ ==

```

```

theSymbolTable

clearTheSymbolTable():Void ==
  theSymbolTable := empty()$Rep
  void()$Void

clearTheSymbolTable(u:Symbol):Void ==
  remove!(u,theSymbolTable)$Rep
  void()$Void

empty():$ ==
  empty()$Rep

currentSubProgram():Symbol ==
  currentSubProgramName

endSubProgram():Symbol ==
-- If we want to support more complex languages then we should keep
-- a list of subprograms / blocks - but for the moment lets stick with
-- Fortran.
  currentSubProgramName := MAIN

newSubProgram(u:Symbol):Void ==
  setelt(theSymbolTable,u,newEntry())$Rep
  currentSubProgramName := u
  void()$Void

argumentList!(u:Symbol,args>List Symbol,symbols:$):Void ==
  checkIfEntryExists(u,symbols)
  setelt(elt(symbols,u)$Rep,argList,args)$Entry

argumentList!(u:Symbol,args>List Symbol):Void ==
  argumentList!(u,args,theSymbolTable)

argumentList!(args>List Symbol):Void ==
  checkIfEntryExists(currentSubProgramName,theSymbolTable)
  setelt(elt(theSymbolTable,currentSubProgramName)$Rep, _
    argList,args)$Entry

returnType!(u:Symbol,type:FSTU,symbols:$):Void ==
  checkIfEntryExists(u,symbols)
  setelt(elt(symbols,u)$Rep,returnType,type)$Entry

returnType!(u:Symbol,type:FSTU):Void ==
  returnType!(u,type,theSymbolTable)

returnType!(type:FSTU ):Void ==
  checkIfEntryExists(currentSubProgramName,theSymbolTable)
  setelt(elt(theSymbolTable,currentSubProgramName)$Rep, _
    returnType,type)$Entry

```

```

declare! (u:Symbol,type:FortranType):FortranType ==
  declare! (u,type,currentSubProgramName,theSymbolTable)

declare! (u:Symbol,type:FortranType,asp:Symbol,symbols:):FortranType ==
  checkIfEntryExists(asp,symbols)
  declare! (u,type, elt(elt(symbols,asp)$Rep,symtab)$Entry)$SymbolTable

declare! (u:List Symbol,type:FortranType,asp:Symbol,syms:):FortranType ==
  checkIfEntryExists(asp,syms)
  declare! (u,type, elt(elt(syms,asp)$Rep,symtab)$Entry)$SymbolTable

declare! (u:Symbol,type:FortranType,asp:Symbol):FortranType ==
  checkIfEntryExists(asp,theSymbolTable)
  declare! (u,type,elt(elt(theSymbolTable,asp)$Rep,symtab)$Entry)$SymbolTable

printHeader(u:Symbol,symbols:):Void ==
  entry := elt(symbols,u)$Rep
  fortFormatHead(elt(entry,returnType)$Entry::OutputForm,u::OutputForm, _
    elt(entry,argList)$Entry::OutputForm)$Lisp
  printTypes(elt(entry,symtab)$Entry)$SymbolTable

printHeader(u:Symbol):Void ==
  printHeader(u,theSymbolTable)

printHeader():Void ==
  printHeader(currentSubProgramName,theSymbolTable)

printTypes(u:Symbol):Void ==
  printTypes(elt(elt(theSymbolTable,u)$Rep,symtab)$Entry)$SymbolTable

```

— SYMS.dotabb —

```

"SYMS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SYMS"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"SYMS" -> "ALIST"

```

domain M3D ThreeDimensionalMatrix

— ThreeDimensionalMatrix.input —

```

)set break resume
)sys rm -f ThreeDimensionalMatrix.output
)spool ThreeDimensionalMatrix.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ThreeDimensionalMatrix
--R
--R ThreeDimensionalMatrix(R: SetCategory) is a domain constructor
--R Abbreviation for ThreeDimensionalMatrix is M3D
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for M3D
--R
--R----- Operations -----
--R copy : % -> %                                empty : () -> %
--R empty? : % -> Boolean                        eq? : (%,%) -> Boolean
--R map : ((R -> R),%) -> %                      sample : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?? : (%,%) -> Boolean if R has SETCAT
--R any? : ((R -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> PrimitiveArray(PrimitiveArray(PrimitiveArray(R)))
--R coerce : PrimitiveArray(PrimitiveArray(PrimitiveArray(R))) -> %
--R coerce : % -> OutputForm if R has SETCAT
--R construct : List(List(List(R))) -> %
--R count : (R,%) -> NonNegativeInteger if $ has finiteAggregate and R has SETCAT
--R count : ((R -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R elt : (%,NonNegativeInteger,NonNegativeInteger,NonNegativeInteger) -> R
--R eval : (%,List(R),List(R)) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (%,R,R) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (%,Equation(R)) -> % if R has EVALAB(R) and R has SETCAT
--R eval : (%,List(Equation(R))) -> % if R has EVALAB(R) and R has SETCAT
--R every? : ((R -> Boolean),%) -> Boolean if $ has finiteAggregate
--R hash : % -> SingleInteger if R has SETCAT
--R identityMatrix : NonNegativeInteger -> % if R has RING
--R latex : % -> String if R has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R map! : ((R -> R),%) -> % if $ has shallowlyMutable
--R matrixConcat3D : (Symbol,%,%) -> %
--R matrixDimensions : % -> Vector(NonNegativeInteger)
--R member? : (R,%) -> Boolean if $ has finiteAggregate and R has SETCAT
--R members : % -> List(R) if $ has finiteAggregate
--R more? : (%,NonNegativeInteger) -> Boolean
--R parts : % -> List(R) if $ has finiteAggregate
--R plus : (%,%) -> % if R has RING
--R setelt! : (%,NonNegativeInteger,NonNegativeInteger,NonNegativeInteger,R) -> R
--R size? : (%,NonNegativeInteger) -> Boolean
--R zeroMatrix : (NonNegativeInteger,NonNegativeInteger,NonNegativeInteger) -> % if R has RING
--R ?~=? : (%,%) -> Boolean if R has SETCAT

```



```
--R
--E 1

)spool
)lisp (bye)
```

— **ThreeDimensionalMatrix.help** —

=====

ThreeDimensionalMatrix examples

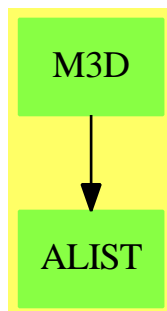
=====

This domain represents three dimensional matrices over a general object type

See Also:

o)show ThreeDimensionalMatrix

ThreeDimensionalMatrix (M3D)



See

- ⇒ “Result” (RESULT) [19](#) on page [2300](#)
- ⇒ “FortranCode” (FC) [7](#) on page [931](#)
- ⇒ “FortranProgram” (FORTRAN) [7](#) on page [955](#)
- ⇒ “SimpleFortranProgram” (SFORT) [20](#) on page [2405](#)
- ⇒ “Switch” (SWITCH) [20](#) on page [2629](#)
- ⇒ “FortranTemplate” (FTEM) [7](#) on page [967](#)
- ⇒ “FortranExpression” (FEXPR) [7](#) on page [946](#)

Exports:

any?	coerce	construct	copy	count
elt	empty	empty?	eq?	eval
every?	hash	identityMatrix	latex	less?
map	map!	matrixConcat3D	matrixDimensions	member?
members	more?	parts	plus	sample
setelt!	size?	zeroMatrix	#?	?=?
?~=?				

— domain M3D ThreeDimensionalMatrix —

```
)abbrev domain M3D ThreeDimensionalMatrix
++ Author: William Naylor
++ Date Created: 20 October 1993
++ Date Last Updated: 20 May 1994
++ Description:
++ This domain represents three dimensional matrices over a general object type
```

```
ThreeDimensionalMatrix(R) : Exports == Implementation where
```

```
R : SetCategory
L ==> List
NNI ==> NonNegativeInteger
A1AGG ==> OneDimensionalArrayAggregate
ARRAY1 ==> OneDimensionalArray
PA ==> PrimitiveArray
INT ==> Integer
PI ==> PositiveInteger
```

```
Exports ==> HomogeneousAggregate(R) with
```

```
if R has Ring then
  zeroMatrix : (NNI,NNI,NNI) -> $
    ++ zeroMatrix(i,j,k) create a matrix with all zero terms
  identityMatrix : (NNI) -> $
    ++ identityMatrix(n) create an identity matrix
    ++ we note that this must be square
  plus : ($,$) -> $
    ++ plus(x,y) adds two matrices, term by term
    ++ we note that they must be the same size
  construct : (L L L R) -> $
    ++ construct(l1l1) creates a 3-D matrix from a List List List R l1l1
  elt : ($,NNI,NNI,NNI) -> R
    ++ elt(x,i,j,k) extract an element from the matrix x
  setelt! : ($,NNI,NNI,NNI,R) -> R
    ++ setelt!(x,i,j,k,s) (or x.i.j.k:=s) sets a specific element of the array to some value of type R
  coerce : (PA PA PA R) -> $
    ++ coerce(p) moves from the representation type
    ++ (PrimitiveArray PrimitiveArray PrimitiveArray R)
```

```

    ++ to the domain
coerce : $ -> (PA PA PA R)
    ++ coerce(x) moves from the domain to the representation type
matrixConcat3D : (Symbol,$,$) -> $
    ++ matrixConcat3D(s,x,y) concatenates two 3-D matrices along a specified axis
matrixDimensions : $ -> Vector NNI
    ++ matrixDimensions(x) returns the dimensions of a matrix

Implementation ==> (PA PA PA R) add

import (PA PA PA R)
import (PA PA R)
import (PA R)
import R

matrix1,matrix2,resultMatrix : $

-- function to concatenate two matrices
-- the first argument must be a symbol, which is either i,j or k
-- to specify the direction in which the concatenation is to take place
matrixConcat3D(dir : Symbol,mat1 : $,mat2 : $) : $ ==
    ^((dir = (i::Symbol)) or (dir = (j::Symbol)) or (dir = (k::Symbol)))_
    => error "the axis of concatenation must be i,j or k"
mat1Dim := matrixDimensions(mat1)
mat2Dim := matrixDimensions(mat2)
iDim1 := mat1Dim.1
jDim1 := mat1Dim.2
kDim1 := mat1Dim.3
iDim2 := mat2Dim.1
jDim2 := mat2Dim.2
kDim2 := mat2Dim.3
matRep1 : (PA PA PA R) := copy(mat1 :: (PA PA PA R))$(PA PA PA R)
matRep2 : (PA PA PA R) := copy(mat2 :: (PA PA PA R))$(PA PA PA R)
retVal : $

if (dir = (i::Symbol)) then
    -- j,k dimensions must agree
    if (^((jDim1 = jDim2) and (kDim1=kDim2)))
    then
        error "jxk do not agree"
    else
        retVal := (coerce(concat(matRep1,matRep2)$(PA PA PA R))$$)@$

if (dir = (j::Symbol)) then
    -- i,k dimensions must agree
    if (^((iDim1 = iDim2) and (kDim1=kDim2)))
    then
        error "ixk do not agree"
    else
        for i in 0..(iDim1-1) repeat

```

```

        setelt(matRep1,i,(concat(elt(matRep1,i)$(PA PA PA R)_
        ,elt(matRep2,i)$(PA PA PA R))$(PA PA R))@(PA PA R))$(PA PA PA R)
    retVal := (coerce(matRep1)$)$@$

if (dir = (k::Symbol)) then
    temp : (PA PA R)
    -- i,j dimensions must agree
    if (^((iDim1 = iDim2) and (jDim1=jDim2)))
    then
        error "ixj do not agree"
    else
        for i in 0..(iDim1-1) repeat
            temp := copy(elt(matRep1,i)$(PA PA PA R))$(PA PA R)
            for j in 0..(jDim1-1) repeat
                setelt(temp,j,concat(elt(elt(matRep1,i)$(PA PA PA R)_
                ,j)$(PA PA R),elt(elt(matRep2,i)$(PA PA PA R),j)$(PA PA R)_
                )$(PA R))$(PA PA R)
            setelt(matRep1,i,temp)$(PA PA PA R)
        retVal := (coerce(matRep1)$)$@$

retVal

matrixDimensions(mat : $) : Vector NNI ==
    matRep : (PA PA PA R) := mat :: (PA PA PA R)
    iDim : NNI := (#matRep)$(PA PA PA R)
    matRep2 : PA PA R := elt(matRep,0)$(PA PA PA R)
    jDim : NNI := (#matRep2)$(PA PA R)
    matRep3 : (PA R) := elt(matRep2,0)$(PA PA R)
    kDim : NNI := (#matRep3)$(PA R)
    retVal : Vector NNI := new(3,0)$(Vector NNI)
    retVal.1 := iDim
    retVal.2 := jDim
    retVal.3 := kDim
    retVal

coerce(matrixRep : (PA PA PA R)) : $ == matrixRep pretend $

coerce(mat : $) : (PA PA PA R) == mat pretend (PA PA PA R)

-- i,j,k must be with in the bounds of the matrix
elt(mat : $,i : NNI,j : NNI,k : NNI) : R ==
    matDims := matrixDimensions(mat)
    iLength := matDims.1
    jLength := matDims.2
    kLength := matDims.3
    ((i > iLength) or (j > jLength) or (k > kLength) or (i=0) or (j=0) or_
(k=0)) => error "coordinates must be within the bounds of the matrix"
    matrixRep : PA PA PA R := mat :: (PA PA PA R)
    elt(elt(elt(matrixRep,i-1)$(PA PA PA R),j-1)$(PA PA R),k-1)$(PA R)

```

```

setelt!(mat : $, i : NNI, j : NNI, k : NNI, val : R) _
: R ==
matDims := matrixDimensions(mat)
iLength := matDims.1
jLength := matDims.2
kLength := matDims.3
((i > iLength) or (j > jLength) or (k > kLength) or (i=0) or (j=0) or_
(k=0)) => error "coordinates must be within the bounds of the matrix"
matrixRep : PA PA PA R := mat :: (PA PA PA R)
row2 : PA PA R := copy(elt(matrixRep, i-1)$(PA PA PA R))$(PA PA R)
row1 : PA R := copy(elt(row2, j-1)$(PA PA R))$(PA R)
setelt(row1, k-1, val)$(PA R)
setelt(row2, j-1, row1)$(PA PA R)
setelt(matrixRep, i-1, row2)$(PA PA PA R)
val

if R has Ring then
zeroMatrix(iLength:NNI, jLength:NNI, kLength:NNI) : $ ==
(new(iLength, new(jLength, new(kLength, (0$R)))$(PA R))$(PA PA R))$(PA PA PA R)) :: $

identityMatrix(iLength:NNI) : $ ==
retValueRep : PA PA PA R := zeroMatrix(iLength, iLength, iLength)$ :: (PA PA PA R)
row1 : PA R
row2 : PA PA R
row1empty : PA R := new(iLength, 0$R)$(PA R)
row2empty : PA PA R := new(iLength, copy(row1empty)$(PA R))$(PA PA R)
for count in 0..(iLength-1) repeat
  row1 := copy(row1empty)$(PA R)
  setelt(row1, count, 1$R)$(PA R)
  row2 := copy(row2empty)$(PA PA R)
  setelt(row2, count, copy(row1)$(PA R))$(PA PA R)
  setelt(retValueRep, count, copy(row2)$(PA PA R))$(PA PA PA R)
retValueRep :: $

plus(mat1 : $, mat2 : $) : $ ==

mat1Dims := matrixDimensions(mat1)
iLength1 := mat1Dims.1
jLength1 := mat1Dims.2
kLength1 := mat1Dims.3

mat2Dims := matrixDimensions(mat2)
iLength2 := mat2Dims.1
jLength2 := mat2Dims.2
kLength2 := mat2Dims.3

-- check that the dimensions are the same
^(iLength1 = iLength2) or ^(jLength1 = jLength2) or ^(kLength1 = kLength2))_
=> error "error the matrices are different sizes"

```

```

sum : R
row1 : (PA R) := new(kLength1,0$R)$(PA R)
row2 : (PA PA R) := new(jLength1,copy(row1)$(PA R))$(PA PA R)
row3 : (PA PA PA R) := new(iLength1,copy(row2)$(PA PA R))$(PA PA PA R)

for i in 1..iLength1 repeat
  for j in 1..jLength1 repeat
    for k in 1..kLength1 repeat
      sum := (elt(mat1,i,j,k)::R +$R_
              elt(mat2,i,j,k)::R)
      setelt(row1,k-1,sum)$(PA R)
      setelt(row2,j-1,copy(row1)$(PA R))$(PA PA R)
      setelt(row3,i-1,copy(row2)$(PA PA R))$(PA PA PA R)

resultMatrix := (row3 pretend $)

resultMatrix

construct(listRep : L L L R) : $ ==

(#listRep)$(L L L R) = 0 => error "empty list"
(#(listRep.1))$(L L R) = 0 => error "empty list"
(#((listRep.1).1))$(L R) = 0 => error "empty list"
iLength := (#listRep)$(L L L R)
jLength := (#(listRep.1))$(L L R)
kLength := (#((listRep.1).1))$(L R)

--first check that the matrix is in the correct form
for subList in listRep repeat
  ^((#subList)$(L L R) = jLength) => error_
"can not have an irregular shaped matrix"
  for subSubList in subList repeat
    ^((#(subSubList))$(L R) = kLength) => error_
"can not have an irregular shaped matrix"

row1 : (PA R) := new(kLength,((listRep.1).1).1)$(PA R)
row2 : (PA PA R) := new(jLength,copy(row1)$(PA R))$(PA PA R)
row3 : (PA PA PA R) := new(iLength,copy(row2)$(PA PA R))$(PA PA PA R)

for i in 1..iLength repeat
  for j in 1..jLength repeat
    for k in 1..kLength repeat

      element := elt(elt(elt(listRep,i)$(L L L R),j)$(L L R),k)$(L R)
      setelt(row1,k-1,element)$(PA R)
      setelt(row2,j-1,copy(row1)$(PA R))$(PA PA R)
      setelt(row3,i-1,copy(row2)$(PA PA R))$(PA PA PA R)

resultMatrix := (row3 pretend $)

```

```
resultMatrix
```

— M3D.dotabb —

```
"M3D" [color="#88FF44",href="bookvol10.3.pdf#nameddest=M3D"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"M3D" -> "ALIST"
```

domain VIEW3D ThreeDimensionalViewport

— ThreeDimensionalViewport.input —

```
)set break resume
)sys rm -f ThreeDimensionalViewport.output
)spool ThreeDimensionalViewport.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ThreeDimensionalViewport
--R
--R ThreeDimensionalViewport is a domain constructor
--R Abbreviation for ThreeDimensionalViewport is VIEW3D
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for VIEW3D
--R
--R----- Operations -----
--R ?? : (%,%) -> Boolean          axes : (%,String) -> Void
--R clipSurface : (%,String) -> Void  close : % -> Void
--R coerce : % -> OutputForm         controlPanel : (%,String) -> Void
--R diagonals : (%,String) -> Void    drawStyle : (%,String) -> Void
--R eyeDistance : (%,Float) -> Void  hash : % -> SingleInteger
--R hitherPlane : (%,Float) -> Void  intensity : (%,Float) -> Void
--R key : % -> Integer              latex : % -> String
--R makeViewport3D : % -> %          options : % -> List(DrawOption)
--R perspective : (%,String) -> Void  reset : % -> Void
--R rotate : (%,Float,Float) -> Void  showRegion : (%,String) -> Void
--R title : (%,String) -> Void       viewDeltaXDefault : Float -> Float
```

```

--R viewDeltaXDefault : () -> Float      viewDeltaYDefault : Float -> Float
--R viewDeltaYDefault : () -> Float      viewPhiDefault : Float -> Float
--R viewPhiDefault : () -> Float         viewThetaDefault : Float -> Float
--R viewThetaDefault : () -> Float       viewZoomDefault : Float -> Float
--R viewZoomDefault : () -> Float        viewport3D : () -> %
--R write : (%,String) -> String         zoom : (%,Float) -> Void
--R ~=? : (%,%) -> Boolean
--R colorDef : (%,Color,Color) -> Void
--R dimensions : (%,NonNegativeInteger,NonNegativeInteger,PositiveInteger,PositiveInteger) -> Void
--R lighting : (%,Float,Float,Float) -> Void
--R makeViewport3D : (ThreeSpace(DoubleFloat),List(DrawOption)) -> %
--R makeViewport3D : (ThreeSpace(DoubleFloat),String) -> %
--R modifyPointData : (%,NonNegativeInteger,Point(DoubleFloat)) -> Void
--R move : (%,NonNegativeInteger,NonNegativeInteger) -> Void
--R options : (%,List(DrawOption)) -> %
--R outlineRender : (%,String) -> Void
--R resize : (%,PositiveInteger,PositiveInteger) -> Void
--R rotate : (%,Integer,Integer) -> Void
--R showClipRegion : (%,String) -> Void
--R subspace : (%,ThreeSpace(DoubleFloat)) -> %
--R subspace : % -> ThreeSpace(DoubleFloat)
--R translate : (%,Float,Float) -> Void
--R viewpoint : (%,Float,Float,Float) -> Void
--R viewpoint : (%,Float,Float) -> Void
--R viewpoint : (%,Integer,Integer,Float,Float,Float) -> Void
--R viewpoint : (%,Record(theta: DoubleFloat,phi: DoubleFloat,scale: DoubleFloat,scaleX: DoubleFloat,scaleY: DoubleFloat)) -> Void
--R viewpoint : % -> Record(theta: DoubleFloat,phi: DoubleFloat,scale: DoubleFloat,scaleX: DoubleFloat,scaleY: DoubleFloat)
--R viewpoint : (%,Float,Float,Float,Float,Float) -> Void
--R write : (%,String,List(String)) -> String
--R write : (%,String,String) -> String
--R zoom : (%,Float,Float,Float) -> Void
--R
--E 1

)spool
)lisp (bye)

```

— ThreeDimensionalViewport.help —

```

=====
ThreeDimensionalViewport examples
=====

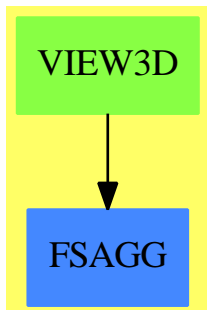
```

ThreeDimensionalViewport creates viewports to display graphs

See Also:

o)show ThreeDimensionalViewport

ThreeDimensionalViewport (VIEW3D)



Exports:

axes	clipSurface	colorDef	close	coerce
controlPanel	diagonals	dimensions	drawStyle	eyeDistance
hash	hitherPlane	intensity	key	latex
lighting	makeViewport3D	modifyPointData	move	options
outlineRender	perspective	reset	resize	rotate
showClipRegion	showRegion	subspace	title	translate
viewDeltaXDefault	viewDeltaYDefault	viewPhiDefault	viewport	viewThetaDefault
viewZoomDefault	viewport3D	write	zoom	?=?
?~=?				

— domain VIEW3D ThreeDimensionalViewport —

```

)abbrev domain VIEW3D ThreeDimensionalViewport
++ Author: Jim Wen
++ Date Created: 28 April 1989
++ Date Last Updated: 2 November 1991, Jim Wen
++ Description:
++ ThreeDimensionalViewport creates viewports to display graphs
  
```

```

ThreeDimensionalViewport(): Exports == Implementation where
  VIEW    ==> VIEWPORTSERVER$Lisp
  sendI   ==> SOCK_-SEND_-INT
  sendSF  ==> SOCK_-SEND_-FLOAT
  sendSTR ==> SOCK_-SEND_-STRING
  getI    ==> SOCK_-GET_-INT
  getSF   ==> SOCK_-GET_-FLOAT

  typeVIEW3D ==> 1$I
  
```

```

typeVIEWTube ==> 4

makeVIEW3D ==> (-1)$SingleInteger

I  ==> Integer
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
XY ==> Record( X:I, Y:I )
XYP ==> Record( X:PI, Y:PI )
XYNN ==> Record( X:NNI, Y:NNI )
SF ==> DoubleFloat
F  ==> Float
L  ==> List
Pt ==> ColoredThreeDimensionalPoint
SEG ==> Segment
S  ==> String
E  ==> OutputForm
PLOT3D ==> Plot3D
TUBE  ==> TubePlot
V  ==> Record( theta:SF, phi:SF, scale:SF, scaleX:SF, scaleY:SF, scaleZ:SF, deltaX:SF, deltaY:SF )
H  ==> Record( hueOffset:I, hueNumber:I)
FLAG ==> Record( showCP:I, style:I, axesOn:I, diagonalsOn:I, outlineRenderOn:I, showRegionField:I )
FR  ==> Record( fn:Fn2, fc: FnU, xmin:SF, xmax:SF, ymin:SF, ymax:SF, xnum:I, ynum:I )
FParamR ==> Record( theTube:TUBE )
LR  ==> Record( lightX:SF, lightY:SF, lightZ:SF, lightTheta:SF, lightPhi:SF , translucence:SF)
UFR ==> Union(FR,FParamR,"undefined")
PR  ==> Record( perspectiveField:I, eyeDistance:SF, hitherPlane:SF)
VR  ==> Record( clipXMin:SF, clipXMax:SF, clipYMin:SF, clipYMax:SF, clipZMin:SF, clipZMax:SF, clipRegionF
C  ==> Color()
B  ==> Boolean
POINT ==> Point(SF)
SUBSPACE ==> SubSpace(3,SF)
SPACE3 ==> ThreeSpace(SF)
DROP ==> DrawOption
COORDSYS ==> CoordinateSystems(SF)

-- the below macros correspond to the ones in include/actions.h
ROTATE      ==> 0$I -- rotate      in actions.h
ZOOM        ==> 1$I -- zoom        in actions.h
TRANSLATE   ==> 2      -- translate in actions.h
rendered    ==> 3      -- render    in actions.h
hideControl ==> 4
closeAll    ==> 5
axesOnOff   ==> 6
opaque      ==> 7      -- opaqueMesh in action.h
contour     ==> 24
RESET       ==> 8
wireMesh    ==> 9      -- transparent in actions.h
region3D    ==> 12
smooth      ==> 22

```

```

diagOnOff      ==> 26
outlineOnOff   ==> 13
zoomx          ==> 14
zoomy          ==> 15
zoomz          ==> 16
perspectiveOnOff ==> 27
clipRegionOnOff ==> 66
clipSurfaceOnOff ==> 67

SPADBUTTONPRESS ==> 100
COLORDEF        ==> 101
MOVE            ==> 102
RESIZE          ==> 103
TITLE           ==> 104
lightDef        ==> 108
translucenceDef ==> 109
writeView       ==> 110
eyeDistanceData ==> 111
modifyPOINT     ==> 114
-- printViewport ==> 115
hitherPlaneData ==> 116
queryVIEWPOINT  ==> 117
changeVIEWPOINT ==> 118

noControl ==> 0$I

yes          ==> 1$I
no           ==> 0$I

EYED        ==> 500::SF -- see draw.h, should be the same(?) as clipOffset
HITHER      ==> (-250)::SF -- see process.h in view3D/ (not yet passed to viewman)

openTube    ==> 1$I
closedTube  ==> 0$I

fun2Var3D ==> "    Three Dimensional Viewport: Function of Two Variables"
para1Var3D ==> "    Three Dimensional Viewport: Parametric Curve of One Variable"
undef3D    ==> "    Three Dimensional Viewport: No function defined for this viewport yet"

Exports ==> SetCategory with
  viewThetaDefault      : ()                                -> F
  ++ viewThetaDefault() returns the current default longitudinal
  ++ view angle in radians.
  viewThetaDefault      : F                                -> F
  ++ viewThetaDefault(t) sets the current default longitudinal
  ++ view angle in radians to the value t and returns t.
  viewPhiDefault        : ()                                -> F
  ++ viewPhiDefault() returns the current default latitudinal
  ++ view angle in radians.
  viewPhiDefault        : F                                -> F

```

```

    ++ viewPhiDefault(p) sets the current default latitudinal
    ++ view angle in radians to the value p and returns p.
viewZoomDefault      : ()                                -> F
    ++ viewZoomDefault() returns the current default graph scaling
    ++ value.
viewZoomDefault      : F                                -> F
    ++ viewZoomDefault(s) sets the current default graph scaling
    ++ value to s and returns s.
viewDeltaXDefault    : ()                                -> F
    ++ viewDeltaXDefault() returns the current default horizontal
    ++ offset from the center of the viewport window.
viewDeltaXDefault    : F                                -> F
    ++ viewDeltaXDefault(dx) sets the current default horizontal
    ++ offset from the center of the viewport window to be \spad{dx}
    ++ and returns \spad{dx}.
viewDeltaYDefault    : ()                                -> F
    ++ viewDeltaYDefault() returns the current default vertical
    ++ offset from the center of the viewport window.
viewDeltaYDefault    : F                                -> F
    ++ viewDeltaYDefault(dy) sets the current default vertical
    ++ offset from the center of the viewport window to be \spad{dy}
    ++ and returns \spad{dy}.
viewport3D           : ()                                -> %
    ++ viewport3D() returns an undefined three-dimensional viewport
    ++ of the domain \spadtype{ThreeDimensionalViewport} whose
    ++ contents are empty.
makeViewport3D       : %                                -> %
    ++ makeViewport3D(v) takes the given three-dimensional viewport,
    ++ v, of the domain \spadtype{ThreeDimensionalViewport} and
    ++ displays a viewport window on the screen which contains
    ++ the contents of v.
makeViewport3D       : (SPACE3,S)                       -> %
    ++ makeViewport3D(sp,s) takes the given space, \spad{sp} which is
    ++ of the domain \spadtype{ThreeSpace} and displays a viewport
    ++ window on the screen which contains the contents of \spad{sp},
    ++ and whose title is given by s.
makeViewport3D       : (SPACE3,L DROP)                   -> %
    ++ makeViewport3D(sp,lopt) takes the given space, \spad{sp} which is
    ++ of the domain \spadtype{ThreeSpace} and displays a viewport
    ++ window on the screen which contains the contents of \spad{sp},
    ++ and whose draw options are indicated by the list \spad{lopt}, which
    ++ is a list of options from the domain \spad{DrawOption}.
subspace             : %                                -> SPACE3
    ++ subspace(v) returns the contents of the viewport v, which is
    ++ of the domain \spadtype{ThreeDimensionalViewport}, as a subspace
    ++ of the domain \spad{ThreeSpace}.
subspace             : (% ,SPACE3)                       -> %
    ++ subspace(v,sp) places the contents of the viewport v, which is
    ++ of the domain \spadtype{ThreeDimensionalViewport}, in the subspace
    ++ \spad{sp}, which is of the domain \spad{ThreeSpace}.

```

```

modifyPointData      : (%,NNI,POINT)                                -> Void
++ modifyPointData(v,ind,pt) takes the viewport, v, which is of the
++ domain \spadtype{ThreeDimensionalViewport}, and places the data
++ point, \spad{pt} into the list of points database of v at the index
++ location given by \spad{ind}.
options              : %                                           -> L DROP
++ options(v) takes the viewport, v, which is of the domain
++ \spadtype{ThreeDimensionalViewport} and returns a list of all
++ the draw options from the domain \spad{DrawOption} which are
++ being used by v.
options              : (%,L DROP)                                   -> %
++ options(v,lopt) takes the viewport, v, which is of the domain
++ \spadtype{ThreeDimensionalViewport} and sets the draw options
++ being used by v to those indicated in the list, \spad{lopt},
++ which is a list of options from the domain \spad{DrawOption}.
move                 : (%,NNI,NNI)                                  -> Void
++ move(v,x,y) displays the three-dimensional viewport, v, which
++ is of domain \spadtype{ThreeDimensionalViewport}, with the upper
++ left-hand corner of the viewport window at the screen
++ coordinate position x, y.
resize               : (%,PI,PI)                                    -> Void
++ resize(v,w,h) displays the three-dimensional viewport, v, which
++ is of domain \spadtype{ThreeDimensionalViewport}, with a width
++ of w and a height of h, keeping the upper left-hand corner
++ position unchanged.
title                : (%,S)                                        -> Void
++ title(v,s) changes the title which is shown in the three-dimensional
++ viewport window, v of domain \spadtype{ThreeDimensionalViewport}.
dimensions           : (%,NNI,NNI,PI,PI)                           -> Void
++ dimensions(v,x,y,width,height) sets the position of the
++ upper left-hand corner of the three-dimensional viewport, v,
++ which is of domain \spadtype{ThreeDimensionalViewport}, to
++ the window coordinate x, y, and sets the dimensions of the
++ window to that of \spad{width}, \spad{height}. The new
++ dimensions are not displayed until the function
++ \spadfun{makeViewport3D} is executed again for v.
viewpoint            : (%,F,F,F,F,F)                               -> Void
++ viewpoint(v,th,phi,s,dx,dy) sets the longitudinal view angle
++ to \spad{th} radians, the latitudinal view angle to \spad{phi}
++ radians, the scale factor to \spad{s}, the horizontal viewport
++ offset to \spad{dx}, and the vertical viewport offset to \spad{dy}
++ for the viewport v, which is of the domain
++ \spadtype{ThreeDimensionalViewport}. The new viewpoint position
++ is not displayed until the function \spadfun{makeViewport3D} is
++ executed again for v.
viewpoint            : (%) -> V
++ viewpoint(v) returns the current viewpoint setting of the given
++ viewport, v. This function is useful in the situation where the
++ user has created a viewport, proceeded to interact with it via
++ the control panel and desires to save the values of the viewpoint

```

```

++ as the default settings for another viewport to be created using
++ the system.
viewpoint : (%,V) -> Void
++ viewpoint(v,viewpt) sets the viewpoint for the viewport. The
++ viewport record consists of the latitudinal and longitudinal angles,
++ the zoom factor, the x,y and z scales, and the x and y displacements.
viewpoint      : (%,I,I,F,F,F) -> Void
++ viewpoint(v,th,phi,s,dx,dy) sets the longitudinal view angle
++ to \spad{th} degrees, the latitudinal view angle to \spad{phi}
++ degrees, the scale factor to \spad{s}, the horizontal viewport
++ offset to \spad{dx}, and the vertical viewport offset to \spad{dy}
++ for the viewport v, which is of the domain
++ \spadtype{ThreeDimensionalViewport}. The new viewpoint position
++ is not displayed until the function \spadfun{makeViewport3D} is
++ executed again for v.
viewpoint      : (%,F,F,F) -> Void
++ viewpoint(v,th,phi) sets the longitudinal view angle to \spad{th}
++ radians and the latitudinal view angle to \spad{phi} radians
++ for the viewport v, which is of the domain
++ \spadtype{ThreeDimensionalViewport}. The new viewpoint position
++ is not displayed until the function \spadfun{makeViewport3D} is
++ executed again for v.
viewpoint      : (%,F,F,F,F) -> Void
++ viewpoint(v,rotx,roty,rotz) sets the rotation about the x-axis
++ to be \spad{rotx} radians, sets the rotation about the y-axis
++ to be \spad{roty} radians, and sets the rotation about the z-axis
++ to be \spad{rotz} radians, for the viewport v, which is of the
++ domain \spadtype{ThreeDimensionalViewport} and displays v with
++ the new view position.
controlPanel   : (%,S) -> Void
++ controlPanel(v,s) displays the control panel of the given
++ three-dimensional viewport, v, which is of domain
++ \spadtype{ThreeDimensionalViewport}, if s is "on", or hides
++ the control panel if s is "off".
axes           : (%,S) -> Void
++ axes(v,s) displays the axes of the given three-dimensional
++ viewport, v, which is of domain \spadtype{ThreeDimensionalViewport},
++ if s is "on", or does not display the axes if s is "off".
diagonals      : (%,S) -> Void
++ diagonals(v,s) displays the diagonals of the polygon outline
++ showing a triangularized surface instead of a quadrilateral
++ surface outline, for the given three-dimensional viewport v
++ which is of domain \spadtype{ThreeDimensionalViewport}, if s is
++ "on", or does not display the diagonals if s is "off".
outlineRender  : (%,S) -> Void
++ outlineRender(v,s) displays the polygon outline showing either
++ triangularized surface or a quadrilateral surface outline depending
++ on the whether the \spadfun{diagonals} function has been set, for
++ the given three-dimensional viewport v which is of domain
++ \spadtype{ThreeDimensionalViewport}, if s is "on", or does not

```

```

++ display the polygon outline if s is "off".
drawStyle      : (%,S)                                -> Void
++ drawStyle(v,s) displays the surface for the given three-dimensional
++ viewport v which is of domain \spadtype{ThreeDimensionalViewport}
++ in the style of drawing indicated by s. If s is not a valid
++ drawing style the style is wireframe by default. Possible
++ styles are \spad{"shade"}, \spad{"solid"} or \spad{"opaque"},
++ \spad{"smooth"}, and \spad{"wireMesh"}.
rotate        : (%,F,F)                                -> Void
++ rotate(v,th,phi) rotates the graph to the longitudinal view angle
++ \spad{th} radians and the latitudinal view angle \spad{phi} radians
++ for the viewport v, which is of the domain
++ \spadtype{ThreeDimensionalViewport}.
rotate        : (%,I,I)                                -> Void
++ rotate(v,th,phi) rotates the graph to the longitudinal view angle
++ \spad{th} degrees and the latitudinal view angle \spad{phi} degrees
++ for the viewport v, which is of the domain
++ \spadtype{ThreeDimensionalViewport}. The new rotation position
++ is not displayed until the function \spadfun{makeViewport3D} is
++ executed again for v.
zoom          : (%,F)                                    -> Void
++ zoom(v,s) sets the graph scaling factor to s, for the viewport v,
++ which is of the domain \spadtype{ThreeDimensionalViewport}.
zoom          : (%,F,F,F)                                -> Void
++ zoom(v,sx,sy,sz) sets the graph scaling factors for the x-coordinate
++ axis to \spad{sx}, the y-coordinate axis to \spad{sy} and the
++ z-coordinate axis to \spad{sz} for the viewport v, which is of
++ the domain \spadtype{ThreeDimensionalViewport}.
translate     : (%,F,F)                                    -> Void
++ translate(v,dx,dy) sets the horizontal viewport offset to \spad{dx}
++ and the vertical viewport offset to \spad{dy}, for the viewport v,
++ which is of the domain \spadtype{ThreeDimensionalViewport}.
perspective   : (%,S)                                    -> Void
++ perspective(v,s) displays the graph in perspective if s is "on",
++ or does not display perspective if s is "off" for the given
++ three-dimensional viewport, v, which is of domain
++ \spadtype{ThreeDimensionalViewport}.
eyeDistance   : (%,F)                                    -> Void
++ eyeDistance(v,d) sets the distance of the observer from the center
++ of the graph to d, for the viewport v, which is of the domain
++ \spadtype{ThreeDimensionalViewport}.
hitherPlane   : (%,F)                                    -> Void
++ hitherPlane(v,h) sets the hither clipping plane of the graph to h,
++ for the viewport v, which is of the domain
++ \spadtype{ThreeDimensionalViewport}.
showRegion    : (%,S)                                    -> Void
++ showRegion(v,s) displays the bounding box of the given
++ three-dimensional viewport, v, which is of domain
++ \spadtype{ThreeDimensionalViewport}, if s is "on", or does not
++ display the box if s is "off".

```

```

showClipRegion      : (%,S)                                -> Void
++ showClipRegion(v,s) displays the clipping region of the given
++ three-dimensional viewport, v, which is of domain
++ \spadtype{ThreeDimensionalViewport}, if s is "on", or does not
++ display the region if s is "off".
clipSurface         : (%,S)                                -> Void
++ clipSurface(v,s) displays the graph with the specified
++ clipping region removed if s is "on", or displays the graph
++ without clipping implemented if s is "off", for the given
++ three-dimensional viewport, v, which is of domain
++ \spadtype{ThreeDimensionalViewport}.
lighting            : (%,F,F,F)                            -> Void
++ lighting(v,x,y,z) sets the position of the light source to
++ the coordinates x, y, and z and displays the graph for the given
++ three-dimensional viewport, v, which is of domain
++ \spadtype{ThreeDimensionalViewport}.
intensity           : (%,F)                                -> Void
++ intensity(v,i) sets the intensity of the light source to i, for
++ the given three-dimensional viewport, v, which is of domain
++ \spadtype{ThreeDimensionalViewport}.
reset               : %                                     -> Void
++ reset(v) sets the current state of the graph characteristics
++ of the given three-dimensional viewport, v, which is of domain
++ \spadtype{ThreeDimensionalViewport}, back to their initial settings.
colorDef            : (%,C,C)                              -> Void
++ colorDef(v,c1,c2) sets the range of colors along the colormap so
++ that the lower end of the colormap is defined by \spad{c1} and the
++ top end of the colormap is defined by \spad{c2}, for the given
++ three-dimensional viewport, v, which is of domain
++ \spadtype{ThreeDimensionalViewport}.
write               : (%,S)                                -> S
++ write(v,s) takes the given three-dimensional viewport, v, which
++ is of domain \spadtype{ThreeDimensionalViewport}, and creates
++ a directory indicated by s, which contains the graph data
++ file for v.
write               : (%,S,S)                              -> S
++ write(v,s,f) takes the given three-dimensional viewport, v, which
++ is of domain \spadtype{ThreeDimensionalViewport}, and creates
++ a directory indicated by s, which contains the graph data
++ file for v and an optional file type f.
write               : (%,S,L S)                            -> S
++ write(v,s,lf) takes the given three-dimensional viewport, v, which
++ is of domain \spadtype{ThreeDimensionalViewport}, and creates
++ a directory indicated by s, which contains the graph data
++ file for v and the optional file types indicated by the list lf.
close               : %                                     -> Void
++ close(v) closes the viewport window of the given
++ three-dimensional viewport, v, which is of domain
++ \spadtype{ThreeDimensionalViewport}, and terminates the
++ corresponding process ID.

```



```

key          : %                                -> I
++ key(v) returns the process ID number of the given three-dimensional
++ viewport, v, which is of domain \spadtype{ThreeDimensionalViewport}.
-- print      : %                                -> Void

Implementation ==> add
import Color()
import ViewDefaultsPackage()
import Plot3D()
import TubePlot()
import POINT
import PointPackage(SF)
import SubSpaceComponentProperty()
import SPACE3
import MeshCreationRoutinesForThreeDimensions()
import DrawOptionFunctions0
import COORDSYS
import Set(PositiveInteger)

Rep := Record (key:I, fun:I, _
               title:S, moveTo:XYNN, size:XPY, viewpoint:V, colors:H, flags:FLAG, _
               lighting:LR, perspective:PR, volume:VR, _
               space3D:SPACE3, _
               optionsField:L DROP)

degrees := pi()$F / 180.0
degreesSF := pi()$SF / 180
defaultTheta : Reference(SF) := ref(convert(pi()$F/4.0)$SF)
defaultPhi   : Reference(SF) := ref(convert(-pi()$F/4.0)$SF)
defaultZoom  : Reference(SF) := ref(convert(1.2)$SF)
defaultDeltaX : Reference(SF) := ref 0
defaultDeltaY : Reference(SF) := ref 0

--%Local Functions
checkViewport (viewport:%):B ==
-- checks to see if this viewport still exists
-- by sending the key to the viewport manager and
-- waiting for its reply after it checks it against
-- the viewports in its list. a -1 means it doesn't
-- exist.
sendI(VIEW,viewport.key)$Lisp
i := getI(VIEW)$Lisp
(i < 0$I) =>
viewport.key := 0$I
error "This viewport has already been closed!"
true

arcsinTemp(x:SF):SF ==
-- the asin function doesn't exist in the SF domain currently

```

```

x >= 1 => (pi())$SF / 2) -- to avoid floating point error from SF (ie 1.0 -> 1.00001)
x <= -1 => 3 * pi()$SF / 2
convert(asin(convert(x)@Float)$Float)@SF

arctanTemp(x:SF):SF == convert(atan(convert(x)@Float)$Float)@SF

doOptions(v:Rep):Void ==
  v.title := title(v.optionsField,"AXIOM3D")
  st:S := style(v.optionsField,"wireMesh")
  if (st = "shade" or st = "render") then
    v.flags.style := rendered
  else if (st = "solid" or st = "opaque") then
    v.flags.style := opaque
  else if (st = "contour") then
    v.flags.style := contour
  else if (st = "smooth") then
    v.flags.style := smooth
  else v.flags.style := wireMesh
  v.viewpoint := viewpoint(v.optionsField,
    [deref defaultTheta,deref defaultPhi,deref defaultZoom, _
     1$SF,1$SF,1$SF,deref defaultDeltaX, deref defaultDeltaY])
  -- etc - 3D specific stuff...

--%Exported Functions : Default Settings
viewport3D() ==
  [0,typeVIEW3D,"AXIOM3D",[viewPosDefault().1,viewPosDefault().2], _
   [viewSizeDefault().1,viewSizeDefault().2], _
   [deref defaultTheta,deref defaultPhi,deref defaultZoom, _
    1$SF,1$SF,1$SF,deref defaultDeltaX, deref defaultDeltaY], [0,27], _
   [noControl,wireMesh,yes,no,no,no], [0$SF,0$SF,1$SF,0$SF,0$SF,1$SF], _
   [yes, EYED, HITHER], [0$SF,1$SF,0$SF,1$SF,0$SF,1$SF,no,yes], _
   create3Space()$SPACE3, [] ]

subspace viewport ==
  viewport.space3D

subspace(viewport,space) ==
  viewport.space3D := space
  viewport

options viewport ==
  viewport.optionsField

options(viewport,opts) ==
  viewport.optionsField := opts
  viewport

makeViewport3D(space:SPACE3,Title:S):% ==
  v := viewport3D()
  v.space3D := space

```

```

v.optionsField := [title(Title)]
makeViewport3D v

makeViewport3D(space:SPACE3,opts:L DROP):% ==
  v := viewport3D()
  v.space3D := space
  v.optionsField := opts
  makeViewport3D v

makeViewport3D viewport ==
  doOptions viewport --local function to extract and assign optional arguments for 3D viewport
  sayBrightly(["  Transmitting data...":E]$List(E))$Lisp
  transform := coord(viewport.optionsField,cartesian$COORDSYS)$DrawOptionFunctions0
  check(viewport.space3D)
  lpts := lp(viewport.space3D)
  lllipts := lllip(viewport.space3D)
  llprops := llprop(viewport.space3D)
  lprops := lprop(viewport.space3D)
  -- check for dimensionality of points
  -- if they are all 4D points, then everything is okay
  -- if they are all 3D points, then pad an extra constant
  -- coordinate for color
  -- if they have varying dimensionalities, give an error
  s := brace()$Set(PI)
  for pt in lpts repeat
    insert_!(dimension pt,s)
  #s > 1 => error "All points should have the same dimension"
  (n := first parts s) < 3 => error "Dimension of points should be greater than 2"
  sendI(VIEW,viewport.fun)$Lisp
  sendI(VIEW,makeVIEW3D)$Lisp
  sendSTR(VIEW,viewport.title)$Lisp
  sendSF(VIEW,viewport.viewpoint.deltaX)$Lisp
  sendSF(VIEW,viewport.viewpoint.deltaY)$Lisp
  sendSF(VIEW,viewport.viewpoint.scale)$Lisp
  sendSF(VIEW,viewport.viewpoint.scaleX)$Lisp
  sendSF(VIEW,viewport.viewpoint.scaleY)$Lisp
  sendSF(VIEW,viewport.viewpoint.scaleZ)$Lisp
  sendSF(VIEW,viewport.viewpoint.theta)$Lisp
  sendSF(VIEW,viewport.viewpoint.phi)$Lisp
  sendI(VIEW,viewport.moveTo.X)$Lisp
  sendI(VIEW,viewport.moveTo.Y)$Lisp
  sendI(VIEW,viewport.size.X)$Lisp
  sendI(VIEW,viewport.size.Y)$Lisp
  sendI(VIEW,viewport.flags.showCP)$Lisp
  sendI(VIEW,viewport.flags.style)$Lisp
  sendI(VIEW,viewport.flags.axesOn)$Lisp
  sendI(VIEW,viewport.flags.diagonalsOn)$Lisp
  sendI(VIEW,viewport.flags.outlineRenderOn)$Lisp
  sendI(VIEW,viewport.flags.showRegionField)$Lisp -- add to make3D.c
  sendI(VIEW,viewport.volume.clipRegionField)$Lisp -- add to make3D.c

```

```

sendI(VIEW,viewport.volume.clipSurfaceField)$Lisp -- add to make3D.c
sendI(VIEW,viewport.colors.hueOffset)$Lisp
sendI(VIEW,viewport.colors.hueNumber)$Lisp
sendSF(VIEW,viewport.lighting.lightX)$Lisp
sendSF(VIEW,viewport.lighting.lightY)$Lisp
sendSF(VIEW,viewport.lighting.lightZ)$Lisp
sendSF(VIEW,viewport.lighting.translucence)$Lisp
sendI(VIEW,viewport.perspective.perspectiveField)$Lisp
sendSF(VIEW,viewport.perspective.eyeDistance)$Lisp
-- new, crazy points domain stuff
-- first, send the point data list
sendI(VIEW,#lpts)$Lisp
for pt in lpts repeat
  aPoint := transform pt
  sendSF(VIEW,xCoord aPoint)$Lisp
  sendSF(VIEW,yCoord aPoint)$Lisp
  sendSF(VIEW,zCoord aPoint)$Lisp
  n = 3 => sendSF(VIEW,zCoord aPoint)$Lisp
  sendSF(VIEW,color aPoint)$Lisp -- change to c
  -- now, send the 3d subspace structure
sendI(VIEW,#lllipts)$Lisp
for allipts in lllipts for oneprop in lprops for onelprops in llprops repeat
  -- the following is false for f(x,y) and user-defined for [x(t),y(t),z(t)]
  -- this is temporary - until the generalized points stuff gets put in
  sendI(VIEW,(closed? oneprop => yes; no))$Lisp
  sendI(VIEW,(solid? oneprop => yes; no))$Lisp
  sendI(VIEW,#allipts)$Lisp
  for alipts in allipts for tinyprop in onelprops repeat
    -- the following is false for f(x,y) and true for [x(t),y(t),z(t)]
    -- this is temporary -- until the generalized points stuff gets put in
    sendI(VIEW,(closed? tinyprop => yes;no))$Lisp
    sendI(VIEW,(solid? tinyprop => yes;no))$Lisp
    sendI(VIEW,#alipts)$Lisp
    for oneIndexedPoint in alipts repeat
      sendI(VIEW,oneIndexedPoint)$Lisp
viewport.key := getI(VIEW)$Lisp
viewport
-- the key (now set to 0) should be what the viewport returns

viewThetaDefault == convert(defaultTheta())@F
viewThetaDefault t ==
  defaultTheta() := convert(t)@SF
  t
viewPhiDefault == convert(defaultPhi())@F
viewPhiDefault t ==
  defaultPhi() := convert(t)@SF
  t
viewZoomDefault == convert(defaultZoom())@F
viewZoomDefault t ==
  defaultZoom() := convert(t)@SF

```

```

t
viewDeltaXDefault == convert(defaultDeltaX())@F
viewDeltaXDefault t ==
    defaultDeltaX() := convert(t)@SF
t
viewDeltaYDefault == convert(defaultDeltaY())@F
viewDeltaYDefault t ==
    defaultDeltaY() := convert(t)@SF
t

--Exported Functions: Available features for 3D viewports
lighting(viewport,Xlight,Ylight,Zlight) ==
    viewport.lighting.lightX := convert(Xlight)@SF
    viewport.lighting.lightY := convert(Ylight)@SF
    viewport.lighting.lightZ := convert(Zlight)@SF
    (key(viewport) ^= 0$I) =>
        sendI(VIEW,typeVIEW3D)$Lisp
        sendI(VIEW,lightDef)$Lisp
        checkViewport viewport =>
            sendSF(VIEW,viewport.lighting.lightX)$Lisp
            sendSF(VIEW,viewport.lighting.lightY)$Lisp
            sendSF(VIEW,viewport.lighting.lightZ)$Lisp
            getI(VIEW)$Lisp -- acknowledge

axes (viewport,onOff) ==
    if onOff = "on" then viewport.flags.axesOn := yes
    else viewport.flags.axesOn := no
    (key(viewport) ^= 0$I) =>
        sendI(VIEW,typeVIEW3D)$Lisp
        sendI(VIEW,axesOnOff)$Lisp
        checkViewport viewport =>
            sendI(VIEW,viewport.flags.axesOn)$Lisp
            getI(VIEW)$Lisp -- acknowledge

diagonals (viewport,onOff) ==
    if onOff = "on" then viewport.flags.diagonalsOn := yes
    else viewport.flags.diagonalsOn := no
    (key(viewport) ^= 0$I) =>
        sendI(VIEW,typeVIEW3D)$Lisp
        sendI(VIEW,diagOnOff)$Lisp
        checkViewport viewport =>
            sendI(VIEW,viewport.flags.diagonalsOn)$Lisp
            getI(VIEW)$Lisp -- acknowledge

outlineRender (viewport,onOff) ==
    if onOff = "on" then viewport.flags.outlineRenderOn := yes
    else viewport.flags.outlineRenderOn := no
    (key(viewport) ^= 0$I) =>
        sendI(VIEW,typeVIEW3D)$Lisp
        sendI(VIEW,outlineOnOff)$Lisp

```

```

checkViewport viewport =>
    sendI(VIEW,viewport.flags.outlineRenderOn)$Lisp
    getI(VIEW)$Lisp          -- acknowledge

controlPanel (viewport,onOff) ==
    if onOff = "on" then viewport.flags.showCP := yes
    else viewport.flags.showCP := no
    (key(viewport) ^= 0$I) =>
        sendI(VIEW,typeVIEW3D)$Lisp
        sendI(VIEW,hideControl)$Lisp
        checkViewport viewport =>
            sendI(VIEW,viewport.flags.showCP)$Lisp
            getI(VIEW)$Lisp          -- acknowledge

drawStyle (viewport,how) ==
    if (how = "shade") then                -- render
        viewport.flags.style := rendered
    else if (how = "solid") then            -- opaque
        viewport.flags.style := opaque
    else if (how = "contour") then          -- contour
        viewport.flags.style := contour
    else if (how = "smooth") then           -- smooth
        viewport.flags.style := smooth
    else viewport.flags.style := wireMesh
    (key(viewport) ^= 0$I) =>
        sendI(VIEW,typeVIEW3D)$Lisp
        sendI(VIEW,viewport.flags.style)$Lisp
        checkViewport viewport =>
            getI(VIEW)$Lisp          -- acknowledge

reset viewport ==
    (key(viewport) ^= 0$I) =>
        sendI(VIEW,typeVIEW3D)$Lisp
        sendI(VIEW,SPADBUTTONPRESS)$Lisp
        checkViewport viewport =>
            sendI(VIEW,RESET)$Lisp
            getI(VIEW)$Lisp          -- acknowledge

close viewport ==
    (key(viewport) ^= 0$I) =>
        sendI(VIEW,typeVIEW3D)$Lisp
        sendI(VIEW,closeAll)$Lisp
        checkViewport viewport =>
            getI(VIEW)$Lisp          -- acknowledge
            viewport.key := 0$I

viewpoint (viewport:~):V ==
    (key(viewport) ^= 0$I) =>
        sendI(VIEW,typeVIEW3D)$Lisp
        sendI(VIEW,queryVIEWPOINT)$Lisp

```

```

checkViewport viewport =>
  deltaX_sf : SF := getSF(VIEW)$Lisp
  deltaY_sf : SF := getSF(VIEW)$Lisp
  scale_sf  : SF := getSF(VIEW)$Lisp
  scaleX_sf : SF := getSF(VIEW)$Lisp
  scaleY_sf : SF := getSF(VIEW)$Lisp
  scaleZ_sf : SF := getSF(VIEW)$Lisp
  theta_sf  : SF := getSF(VIEW)$Lisp
  phi_sf    : SF := getSF(VIEW)$Lisp
  getI(VIEW)$Lisp      -- acknowledge
  viewport.viewpoint :=
    [ theta_sf, phi_sf, scale_sf, scaleX_sf, scaleY_sf, scaleZ_sf,
      deltaX_sf, deltaY_sf ]
  viewport.viewpoint

viewpoint (viewport:%, viewpt:V):Void ==
  viewport.viewpoint := viewpt
  (key(viewport) ^= 0$I) =>
    sendI(VIEW,typeVIEW3D)$Lisp
    sendI(VIEW,changeVIEWPOINT)$Lisp
    checkViewport viewport =>
      sendSF(VIEW,viewport.viewpoint.deltaX)$Lisp
      sendSF(VIEW,viewport.viewpoint.deltaY)$Lisp
      sendSF(VIEW,viewport.viewpoint.scale)$Lisp
      sendSF(VIEW,viewport.viewpoint.scaleX)$Lisp
      sendSF(VIEW,viewport.viewpoint.scaleY)$Lisp
      sendSF(VIEW,viewport.viewpoint.scaleZ)$Lisp
      sendSF(VIEW,viewport.viewpoint.theta)$Lisp
      sendSF(VIEW,viewport.viewpoint.phi)$Lisp
      getI(VIEW)$Lisp      -- acknowledge

viewpoint (viewport:%,Theta:F,Phi:F,Scale:F,DeltaX:F,DeltaY:F):Void ==
  viewport.viewpoint :=
    [convert(Theta)$SF,convert(Phi)$SF,convert(Scale)$SF,1$SF,1$SF,1$SF,convert(DeltaX)$SF,convert(DeltaY)$SF]

viewpoint (viewport:%,Theta:I,Phi:I,Scale:F,DeltaX:F,DeltaY:F):Void ==
  viewport.viewpoint := [convert(Theta)$SF * degreesSF,convert(Phi)$SF * degreesSF,
    convert(Scale)$SF,1$SF,1$SF,1$SF,convert(DeltaX)$SF,convert(DeltaY)$SF]

viewpoint (viewport:%,Theta:F,Phi:F):Void ==
  viewport.viewpoint.theta := convert(Theta)$SF * degreesSF
  viewport.viewpoint.phi   := convert(Phi)$SF * degreesSF

viewpoint (viewport:%,X:F,Y:F,Z:F):Void ==
  Theta : F
  Phi : F
  if (X=0$F) and (Y=0$F) then
    Theta := 0$F
    if (Z>=0$F) then

```

```

        Phi := 0$F
    else
        Phi := 180.0
    else
        Theta := asin(Y/(R := sqrt(X*X+Y*Y)))
        if (Z=0$F) then
            Phi := 90.0
        else
            Phi := atan(Z/R)
    rotate(viewport, Theta * degrees, Phi * degrees)

title (viewport,Title) ==
viewport.title := Title
(key(viewport) ^= 0$I) =>
    sendI(VIEW,typeVIEW3D)$Lisp
    sendI(VIEW,TITLE)$Lisp
    checkViewport viewport =>
        sendSTR(VIEW,Title)$Lisp
        getI(VIEW)$Lisp          -- acknowledge

colorDef (viewport,HueOffset,HueNumber) ==
viewport.colors := [h := (hue HueOffset),(hue HueNumber) - h]
(key(viewport) ^= 0$I) =>
    sendI(VIEW,typeVIEW3D)$Lisp
    sendI(VIEW,COLORDEF)$Lisp
    checkViewport viewport =>
        sendI(VIEW,hue HueOffset)$Lisp
        sendI(VIEW,hue HueNumber)$Lisp
        getI(VIEW)$Lisp          -- acknowledge

dimensions (viewport,ViewX,ViewY,ViewWidth,ViewHeight) ==
viewport.moveTo := [ViewX,ViewY]
viewport.size := [ViewWidth,ViewHeight]

move(viewport,xLoc,yLoc) ==
viewport.moveTo := [xLoc,yLoc]
(key(viewport) ^= 0$I) =>
    sendI(VIEW,typeVIEW3D)$Lisp
    sendI(VIEW,MOVE)$Lisp
    checkViewport viewport =>
        sendI(VIEW,xLoc)$Lisp
        sendI(VIEW,yLoc)$Lisp
        getI(VIEW)$Lisp          -- acknowledge

resize(viewport,xSize,ySize) ==
viewport.size := [xSize,ySize]
(key(viewport) ^= 0$I) =>
    sendI(VIEW,typeVIEW3D)$Lisp
    sendI(VIEW,RESIZE)$Lisp
    checkViewport viewport =>

```



```

        sendI(VIEW,xSize)$Lisp
        sendI(VIEW,ySize)$Lisp
        getI(VIEW)$Lisp          -- acknowledge

coerce viewport ==
  (key(viewport) = 0$I) =>
    hconcat
      ["Closed or Undefined ThreeDimensionalViewport: "::E,
        (viewport.title)::E]
    hconcat ["ThreeDimensionalViewport: "::E, (viewport.title)::E]

key viewport == viewport.key

rotate(viewport:%,Theta:I,Phi:I) ==
  rotate(viewport,Theta::F * degrees,Phi::F * degrees)

rotate(viewport:%,Theta:F,Phi:F) ==
  viewport.viewpoint.theta := convert(Theta)@SF
  viewport.viewpoint.phi   := convert(Phi)@SF
  (key(viewport) ^= 0$I) =>
    sendI(VIEW,typeVIEW3D)$Lisp
    sendI(VIEW,ROTATE)$Lisp
    checkViewport viewport =>
      sendSF(VIEW,viewport.viewpoint.theta)$Lisp
      sendSF(VIEW,viewport.viewpoint.phi)$Lisp
      getI(VIEW)$Lisp          -- acknowledge

zoom(viewport:%,Scale:F) ==
  viewport.viewpoint.scale := convert(Scale)@SF
  (key(viewport) ^= 0$I) =>
    sendI(VIEW,typeVIEW3D)$Lisp
    sendI(VIEW,ZOOM)$Lisp
    checkViewport viewport =>
      sendSF(VIEW,viewport.viewpoint.scale)$Lisp
      getI(VIEW)$Lisp          -- acknowledge

zoom(viewport:%,ScaleX:F,ScaleY:F,ScaleZ:F) ==
  viewport.viewpoint.scaleX := convert(ScaleX)@SF
  viewport.viewpoint.scaleY := convert(ScaleY)@SF
  viewport.viewpoint.scaleZ := convert(ScaleZ)@SF
  (key(viewport) ^= 0$I) =>
    sendI(VIEW,typeVIEW3D)$Lisp
    sendI(VIEW,zoomx)$Lisp
    checkViewport viewport =>
      sendSF(VIEW,viewport.viewpoint.scaleX)$Lisp
      sendSF(VIEW,viewport.viewpoint.scaleY)$Lisp
      sendSF(VIEW,viewport.viewpoint.scaleZ)$Lisp
      getI(VIEW)$Lisp          -- acknowledge

translate(viewport,DeltaX,DeltaY) ==

```

```

viewport.viewpoint.deltaX := convert(DeltaX)@SF
viewport.viewpoint.deltaY := convert(DeltaY)@SF
(key(viewport) ^= 0$I) =>
    sendI(VIEW,typeVIEW3D)$Lisp
    sendI(VIEW,TRANSLATE)$Lisp
    checkViewport viewport =>
        sendSF(VIEW,viewport.viewpoint.deltaX)$Lisp
        sendSF(VIEW,viewport.viewpoint.deltaY)$Lisp
    getI(VIEW)$Lisp        -- acknowledge

intensity(viewport,Amount) ==
    if (Amount < 0$F) or (Amount > 1$F) then
        error "The intensity must be a value between 0 and 1, inclusively."
    viewport.lighting.translucence := convert(Amount)@SF
    (key(viewport) ^= 0$I) =>
        sendI(VIEW,typeVIEW3D)$Lisp
        sendI(VIEW,translucenceDef)$Lisp
        checkViewport viewport =>
            sendSF(VIEW,viewport.lighting.translucence)$Lisp
        getI(VIEW)$Lisp    -- acknowledge

write(viewport:%,Filename:S,aThingToWrite:S) ==
    write(viewport,Filename,[aThingToWrite])

write(viewport,Filename) ==
    write(viewport,Filename,viewWriteDefault())

write(viewport:%,Filename:S,thingsToWrite:L S) ==
    stringToSend : S := ""
    (key(viewport) ^= 0$I) =>
        sendI(VIEW,typeVIEW3D)$Lisp
        sendI(VIEW,writeView)$Lisp
        checkViewport viewport =>
            sendSTR(VIEW,Filename)$Lisp
            m := minIndex(avail := viewWriteAvailable())
            for aTypeOfFile in thingsToWrite repeat
                if (writeTypeInt:= position(upperCase aTypeOfFile,avail)-m) < 0 then
                    sayBrightly([" > "::E,(concat(aTypeOfFile, _
                        " is not a valid file type for writing a 3D viewport"))::E]$List(E))$Lisp
                else
                    sendI(VIEW,writeTypeInt+(1$I))$Lisp
            sendI(VIEW,0$I)$Lisp    -- no more types of things to write
        getI(VIEW)$Lisp        -- acknowledge
    Filename

perspective (viewport,onOff) ==
    if onOff = "on" then viewport.perspective.perspectiveField := yes
    else viewport.perspective.perspectiveField := no
    (key(viewport) ^= 0$I) =>
        sendI(VIEW,typeVIEW3D)$Lisp

```

```

sendI(VIEW,perspectiveOnOff)$Lisp
checkViewport viewport =>
    sendI(VIEW,viewport.perspective.perspectiveField)$Lisp
    getI(VIEW)$Lisp          -- acknowledge

showRegion (viewport,onOff) ==
    if onOff = "on" then viewport.flags.showRegionField := yes
    else viewport.flags.showRegionField := no
    (key(viewport) ~= 0$I) =>
        sendI(VIEW,typeVIEW3D)$Lisp
        sendI(VIEW,region3D)$Lisp
        checkViewport viewport =>
            sendI(VIEW,viewport.flags.showRegionField)$Lisp
            getI(VIEW)$Lisp          -- acknowledge

showClipRegion (viewport,onOff) ==
    if onOff = "on" then viewport.volume.clipRegionField := yes
    else viewport.volume.clipRegionField := no
    (key(viewport) ~= 0$I) =>
        sendI(VIEW,typeVIEW3D)$Lisp
        sendI(VIEW,clipRegionOnOff)$Lisp
        checkViewport viewport =>
            sendI(VIEW,viewport.volume.clipRegionField)$Lisp
            getI(VIEW)$Lisp          -- acknowledge

clipSurface (viewport,onOff) ==
    if onOff = "on" then viewport.volume.clipSurfaceField := yes
    else viewport.volume.clipSurfaceField := no
    (key(viewport) ~= 0$I) =>
        sendI(VIEW,typeVIEW3D)$Lisp
        sendI(VIEW,clipSurfaceOnOff)$Lisp
        checkViewport viewport =>
            sendI(VIEW,viewport.volume.clipSurfaceField)$Lisp
            getI(VIEW)$Lisp          -- acknowledge

eyeDistance(viewport:%,EyeDistance:F) ==
    viewport.perspective.eyeDistance := convert(EyeDistance)@SF
    (key(viewport) ~= 0$I) =>
        sendI(VIEW,typeVIEW3D)$Lisp
        sendI(VIEW,eyeDistanceData)$Lisp
        checkViewport viewport =>
            sendSF(VIEW,viewport.perspective.eyeDistance)$Lisp
            getI(VIEW)$Lisp          -- acknowledge

hitherPlane(viewport:%,HitherPlane:F) ==
    viewport.perspective.hitherPlane := convert(HitherPlane)@SF
    (key(viewport) ~= 0$I) =>
        sendI(VIEW,typeVIEW3D)$Lisp
        sendI(VIEW,hitherPlaneData)$Lisp
        checkViewport viewport =>

```

```

        sendSF(VIEW,viewport.perspective.hitherPlane)$Lisp
        getI(VIEW)$Lisp          -- acknowledge

modifyPointData(viewport,anIndex,aPoint) ==
(n := dimension aPoint) < 3 => error "The point should have dimension of at least 3"
viewport.space3D := modifyPointData(viewport.space3D,anIndex,aPoint)
(key(viewport) ^= 0$I) =>
    sendI(VIEW,typeVIEW3D)$Lisp
    sendI(VIEW,modifyPOINT)$Lisp
    checkViewport viewport =>
        sendI(VIEW,anIndex)$Lisp
        sendSF(VIEW,xCoord aPoint)$Lisp
        sendSF(VIEW,yCoord aPoint)$Lisp
        sendSF(VIEW,zCoord aPoint)$Lisp
        if (n = 3) then sendSF(VIEW,convert(0.5)@SF)$Lisp
        else sendSF(VIEW,color aPoint)$Lisp
        getI(VIEW)$Lisp          -- acknowledge

-- print viewport ==
-- (key(viewport) ^= 0$I) =>
--     sendI(VIEW,typeVIEW3D)$Lisp
--     sendI(VIEW,printViewport)$Lisp
--     checkViewport viewport =>
--         getI(VIEW)$Lisp          -- acknowledge

-----

— VIEW3D.dotabb —

"VIEW3D" [color="#88FF44",href="bookvol10.3.pdf#nameddest=VIEW3D"]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"VIEW3D" -> "FSAGG"

-----

```

domain SPACE3 ThreeSpace

```

— ThreeSpace.input —

)set break resume
)sys rm -f ThreeSpace.output
)spool ThreeSpace.output
)set message test on
)set message auto off

```

```
)clear all
```

```
--S 1 of 1
```

```
)show ThreeSpace
```

```
--R
```

```
--R ThreeSpace(R: Ring) is a domain constructor
```

```
--R Abbreviation for ThreeSpace is SPACE3
```

```
--R This constructor is exposed in this frame.
```

```
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for SPACE3
```

```
--R
```

```
--R----- Operations -----
```

```
--R ==? : (% , %) -> Boolean
```

```
check : % -> %
```

```
--R closedCurve : % -> List(Point(R))
```

```
closedCurve : List(Point(R)) -> %
```

```
--R closedCurve? : % -> Boolean
```

```
coerce : % -> OutputForm
```

```
--R components : % -> List(%)
```

```
composite : List(% ) -> %
```

```
--R composites : % -> List(%)
```

```
copy : % -> %
```

```
--R create3Space : SubSpace(3,R) -> %
```

```
create3Space : ( ) -> %
```

```
--R curve : % -> List(Point(R))
```

```
curve : List(Point(R)) -> %
```

```
--R curve : (% , List(List(R))) -> %
```

```
curve : (% , List(Point(R))) -> %
```

```
--R curve? : % -> Boolean
```

```
hash : % -> SingleInteger
```

```
--R latex : % -> String
```

```
lp : % -> List(Point(R))
```

```
--R merge : (% , %) -> %
```

```
merge : List(% ) -> %
```

```
--R mesh : % -> List(List(Point(R)))
```

```
mesh : List(List(Point(R))) -> %
```

```
--R mesh? : % -> Boolean
```

```
point : % -> Point(R)
```

```
--R point : Point(R) -> %
```

```
point : (% , List(R)) -> %
```

```
--R point : (% , Point(R)) -> %
```

```
point? : % -> Boolean
```

```
--R polygon : % -> List(Point(R))
```

```
polygon : List(Point(R)) -> %
```

```
--R polygon : (% , List(List(R))) -> %
```

```
polygon? : % -> Boolean
```

```
--R subspace : % -> SubSpace(3,R)
```

```
?~=? : (% , %) -> Boolean
```

```
--R closedCurve : (% , List(List(R))) -> %
```

```
--R closedCurve : (% , List(Point(R))) -> %
```

```
--R enterPointData : (% , List(Point(R))) -> NonNegativeInteger
```

```
--R lllip : % -> List(List(List(NonNegativeInteger)))
```

```
--R lllp : % -> List(List(List(Point(R))))
```

```
--R llprop : % -> List(List(SubSpaceComponentProperty))
```

```
--R lprop : % -> List(SubSpaceComponentProperty)
```

```
--R mesh : (List(List(Point(R))), Boolean, Boolean) -> %
```

```
--R mesh : (% , List(List(List(R))), Boolean, Boolean) -> %
```

```
--R mesh : (% , List(List(Point(R))), Boolean, Boolean) -> %
```

```
--R mesh : (% , List(List(List(R))), List(SubSpaceComponentProperty), SubSpaceComponentProperty) ->
```

```
--R mesh : (% , List(List(Point(R))), List(SubSpaceComponentProperty), SubSpaceComponentProperty) ->
```

```
--R modifyPointData : (% , NonNegativeInteger, Point(R)) -> %
```

```
--R numberOfComponents : % -> NonNegativeInteger
```

```
--R numberOfComposites : % -> NonNegativeInteger
```

```
--R objects : % -> Record(points: NonNegativeInteger, curves: NonNegativeInteger, polygons: NonNeg
```

```
--R point : (% , NonNegativeInteger) -> %
```

```
--R polygon : (% , List(Point(R))) -> %
```

```
--R
```

```
--E 1
```

```
)spool
)lisp (bye)
```

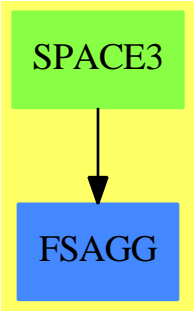
— ThreeSpace.help —

=====
ThreeSpace examples
=====

The domain ThreeSpace is used for creating three dimensional objects using functions for defining points, curves, polygons, constructs and the subspaces containing them.

See Also:
o)show ThreeSpace

ThreeSpace (SPACE3)



Exports:

check	closedCurve	closedCurve?	coerce	components
composite	composites	copy	create3Space	curve
curve?	enterPointData	hash	latex	lflip
lflip	lprop	lprop	lp	merge
mesh	mesh?	modifyPointData	numberOfComponents	objects
point	point?	polygon	polygon?	subspace
?=?	?~=?			

— domain SPACE3 ThreeSpace —

```
)abbrev domain SPACE3 ThreeSpace
```

```

++ Author: Mark Botch
++ Description:
++ The domain ThreeSpace is used for creating three dimensional
++ objects using functions for defining points, curves, polygons, constructs
++ and the subspaces containing them.

ThreeSpace(R:Ring):Exports == Implementation where
  -- m is the dimension of the point

  I    ==> Integer
  PI   ==> PositiveInteger
  NNI  ==> NonNegativeInteger
  L    ==> List
  B    ==> Boolean
  O    ==> OutputForm
  SUBSPACE ==> SubSpace(3,R)
  POINT ==> Point(R)
  PROP ==> SubSpaceComponentProperty()
  REP3D ==> Record(lp:L POINT, lllipt:L L L NNI, llprop:L L PROP, lprop:L PROP)
  OBJ3D ==> Record(points>NNI, curves>NNI, polygons>NNI, constructs>NNI)

Exports ==> ThreeSpaceCategory(R)
Implementation ==> add
  import COMPPROP
  import POINT
  import SUBSPACE
  import ListFunctions2(List(R),POINT)
  import Set(NNI)

  Rep := Record( subspaceField:SUBSPACE, compositesField:L SUBSPACE, _
                 rep3DField:REP3D, objectsField:OBJ3D, _
                 converted:B)

--% Local Functions
convertSpace : % -> %
convertSpace space ==
  space.converted => space
  space.converted := true
  lllipt : L L L NNI := []
  llprop : L L PROP := []
  lprop : L PROP := []
  for component in children space.subspaceField repeat
    lprop := cons(extractProperty component, lprop)
    tmpllipt : L L NNI := []
    tmplprop : L PROP := []
    for curve in children component repeat
      tmplprop := cons(extractProperty curve, tmplprop)
      tmplipt : L NNI := []
      for point in children curve repeat
        tmplipt := cons(extractIndex point, tmplipt)

```

```

        tmpllipt := cons(reverse_! tmplipt,tmpllipt)
        llprop := cons(reverse_! tmplprop, llprop)
        lllipt := cons(reverse_! tmpllipt, lllipt)
space.rep3DField := [pointData space.subspaceField,
                    reverse_! lllipt,reverse_! llprop,reverse_! lprop]
space

--% Exported Functions
polygon(space:%,points:L POINT) ==
  #points < 3 =>
    error "You need at least 3 points to define a polygon"
  pt := addPoint2(space.subspaceField,first points)
  points := rest points
  addPointLast(space.subspaceField, pt, first points, 1)
  for p in rest points repeat
    addPointLast(space.subspaceField, pt, p, 2)
  space.converted := false
  space
create3Space() == [ new()$SUBSPACE, [], [ [], [], [], [] ], [0,0,0,0], false ]
create3Space(s) == [ s, [], [ [], [], [], [] ], [0,0,0,0], false ]
numberOfComponents(space) == #(children((space::Rep).subspaceField))
numberOfComposites(space) == #((space::Rep).compositesField)
merge(listOfThreeSpaces) ==
  -- * -- we may want to remove duplicate components when that functionality exists in List
  newspace := create3Space(merge([ts.subspaceField for ts in listOfThreeSpaces]))
--   newspace.compositesField := [for cs in ts.compositesField for ts in listOfThreeSpaces]
  for ts in listOfThreeSpaces repeat
    newspace.compositesField := append(ts.compositesField,newspace.compositesField)
  newspace
merge(s1,s2) == merge([s1,s2])
composite(listOfThreeSpaces) ==
  space := create3Space()
  space.subspaceField := merge [s.subspaceField for s in listOfThreeSpaces]
  space.compositesField := [deepCopy space.subspaceField]
--   for aSpace in listOfThreeSpaces repeat
--     -- create a composite (which are supercomponents that group
--     -- separate components together) out of all possible components
--   space.compositesField := append(children aSpace.subspaceField,space.compositesField)
  space
components(space) == [create3Space(s) for s in separate space.subspaceField]
composites(space) == [create3Space(s) for s in space.compositesField]
copy(space) ==
  spc := create3Space(deepCopy(space.subspaceField))
  spc.compositesField := [deepCopy s for s in space.compositesField]
  spc

enterPointData(space,listOfPoints) ==
  for p in listOfPoints repeat
    addPoint(space.subspaceField,p)

```



```

    #(pointData space.subspaceField)
    modifyPointData(space,i,p) ==
        modifyPoint(space.subspaceField,i,p)
    space

    -- 3D primitives, each grouped in the following order
    --   xxx?(s) : query whether the threespace, s, holds an xxx
    --   xxx(s)  : extract xxx from threespace, s
    --   xxx(p)  : create a new three space with xxx, p
    --   xxx(s,p): add xxx, p, to a three space, s
    --   xxx(s,q): add an xxx, convertible from q, to a three space, s
    --   xxx(s,i): add an xxx, the data for xxx being indexed by reference *** complete th
point?(space:%) ==
    #(c:=children space.subspaceField) > 1$NNI =>
        error "This ThreeSpace has more than one component"
        -- our 3-space has one component, a list of list of points
    #(kid:=children first c) = 1$NNI => -- the component has one subcomponent (a list of point
        #(children first kid) = 1$NNI -- this list of points only has one entry, so it's a poin
    false
point(space:%) ==
    point? space => extractPoint(traverse(space.subspaceField,[1,1,1]::L NNI))
    error "This ThreeSpace holds something other than a single point - try the objects() comma
point(aPoint:POINT) == point(create3Space(),aPoint)
point(space:%,aPoint:POINT) ==
    addPoint(space.subspaceField,[],aPoint)
    space.converted := false
    space
point(space:%,l:L R) ==
    pt := point(l)
    point(space,pt)
point(space:%,i:NNI) ==
    addPoint(space.subspaceField,[],i)
    space.converted := false
    space

curve?(space:%) ==
    #(c:=children space.subspaceField) > 1$NNI =>
        error "This ThreeSpace has more than one component"
        -- our 3-space has one component, a list of list of points
    #(children first c) = 1$NNI -- there is only one subcomponent, so it's a list of points
curve(space:%) ==
    curve? space =>
        spc := first children first children space.subspaceField
        [extractPoint(s) for s in children spc]
        error "This ThreeSpace holds something other than a curve - try the objects() command"
    curve(points:L POINT) == curve(create3Space(),points)
    curve(space:%,points:L POINT) ==
        addPoint(space.subspaceField,[],first points)
    path : L NNI := [#(children space.subspaceField),1]
    for p in rest points repeat

```

```

    addPoint(space.subspaceField,path,p)
    space.converted := false
    space
curve(space:%,points:L L R) ==
    pts := map(point,points)
    curve(space,pts)

closedCurve?(space:%) ==
    #(c:=children space.subspaceField) > 1$NNI =>
        error "This ThreeSpace has more than one component"
        -- our 3-space has one component, a list of list of points
    #(kid := children first c) = 1$NNI => -- there is one subcomponent => it's a list of points
        extractClosed first kid -- is it a closed curve?
        false
closedCurve(space:%) ==
    closedCurve? space =>
        spc := first children first children space.subspaceField
        -- get the list of points
        [extractPoint(s) for s in children spc]
        -- for now, we are not repeating points...
        error "This ThreeSpace holds something other than a curve - try the objects() command"
closedCurve(points:L POINT) == closedCurve(create3Space(),points)
closedCurve(space:%,points:L POINT) ==
    addPoint(space.subspaceField,[],first points)
    path : L NNI := [(children space.subspaceField),1]
    closeComponent(space.subspaceField,path,true)
    for p in rest points repeat
        addPoint(space.subspaceField,path,p)
    space.converted := false
    space
closedCurve(space:%,points:L L R) ==
    pts := map(point,points)
    closedCurve(space,pts)

polygon?(space:%) ==
    #(c:=children space.subspaceField) > 1$NNI =>
        error "This ThreeSpace has more than one component"
        -- our 3-space has one component, a list of list of points
    #(kid:=children first c) = 2:$NNI =>
        -- there are two subcomponents
        -- the convention is to have one point in the first child and to put
        -- the remaining points (2 or more) in the second, and last, child
        #(children first kid) = 1$NNI and #(children second kid) > 2:$NNI
        false -- => returns Void...?
polygon(space:%) ==
    polygon? space =>
        listOfPoints : L POINT :=
            [extractPoint(first children first (cs := children first children space.subspaceField))],
            [extractPoint(s) for s in children second cs]
        error "This ThreeSpace holds something other than a polygon - try the objects() command"

```

```

polygon(points:L POINT) == polygon(create3Space(),points)
polygon(space:%,points:L L R) ==
  pts := map(point,points)
  polygon(space,pts)

mesh?(space:%) ==
  #(c:=children space.subspaceField) > 1$NNI =>
    error "This ThreeSpace has more than one component"
  -- our 3-space has one component, a list of list of points
  #(kid:=children first c) > 1$NNI =>
    -- there are two or more subcomponents (list of points)
    -- so this may be a definition of a mesh; if the size
    -- of each list of points is the same and they are all
    -- greater than 1(?) then we have an acceptable mesh
    -- use a set to hold the curve size info: if heterogenous
    -- curve sizes exist, then the set would hold all the sizes;
    -- otherwise it would just have the one element indicating
    -- the sizes for all the curves
    whatSizes := brace()$Set(NNI)
    for eachCurve in kid repeat
      insert_!(#children eachCurve,whatSizes)
    #whatSizes > 1 => error "Mesh defined with curves of different sizes"
    first parts whatSizes < 2 =>
      error "Mesh defined with single point curves (use curve())"
    true
  false
mesh(space:%) ==
  mesh? space =>
    llp : L L POINT := []
    for lpSpace in children first children space.subspaceField repeat
      llp := cons([extractPoint(s) for s in children lpSpace],llp)
    llp
    error "This ThreeSpace holds something other than a mesh - try the objects() command"
  mesh(points:L L POINT) == mesh(create3Space(),points,false,false)
  mesh(points:L L POINT,prop1:B,prop2:B) == mesh(create3Space(),points,prop1,prop2)
--+ old ones \/
mesh(space:%,llpoints:L L L R,lprops:L PROP,prop:PROP) ==
  pts := [map(point,points) for points in llpoints]
  mesh(space,pts,lprops,prop)
mesh(space:%,llp:L L POINT,lprops:L PROP,prop:PROP) ==
  addPoint(space.subspaceField,[],first first llp)
  defineProperty(space.subspaceField,path:L NNI:=[#children space.subspaceField],prop)
  path := append(path,[1])
  defineProperty(space.subspaceField,path,first lprops)
  for p in rest (first llp) repeat
    addPoint(space.subspaceField,path,p)
  for lp in rest llp for aProp in rest lprops for count in 2.. repeat
    addPoint(space.subspaceField,path := [first path],first lp)
    path := append(path,[count])
    defineProperty(space.subspaceField,path,aProp)

```

```

        for p in rest lp repeat
            addPoint(space.subspaceField,path,p)
        space.converted := false
    space
--+ old ones /\
mesh(space:%,llpoints:L L L R,prop1:B,prop2:B) ==
    pts := [map(point,points) for points in llpoints]
    mesh(space,pts,prop1,prop2)
mesh(space:%,llp:L L POINT,prop1:B,prop2:B) ==
    -- prop2 refers to property of the ends of a surface (list of lists of points)
    -- while prop1 refers to the individual curves (list of points)
    -- ** note we currently use Booleans for closed (rather than a pair
    -- ** of booleans for closed and solid)
    propA : PROP := new()
    close(propA,prop1)
    propB : PROP := new()
    close(propB,prop2)
    addPoint(space.subspaceField,[],first first llp)
    defineProperty(space.subspaceField,path:L NNI:=[#children space.subspaceField],propB)
    path := append(path,[1])
    defineProperty(space.subspaceField,path,propA)
    for p in rest (first llp) repeat
        addPoint(space.subspaceField,path,p)
    for lp in rest llp for count in 2.. repeat
        addPoint(space.subspaceField,path := [first path],first lp)
        path := append(path,[count])
        defineProperty(space.subspaceField,path,propA)
        for p in rest lp repeat
            addPoint(space.subspaceField,path,p)
    space.converted := false
    space

lp space ==
    if ^space.converted then space := convertSpace space
    space.rep3DField.lp
l1lip space ==
    if ^space.converted then space := convertSpace space
    space.rep3DField.l1liPt
-- l1lp space ==
--     if ^space.converted then space := convertSpace space
--     space.rep3DField.l1lPt
l1lprop space ==
    if ^space.converted then space := convertSpace space
    space.rep3DField.l1lProp
lprop space ==
    if ^space.converted then space := convertSpace space
    space.rep3DField.lProp

-- this function is just to see how this representation really
-- does work

```

```

objects space ==
  if ^space.converted then space := convertSpace space
  numPts      := 0$NNI
  numCurves   := 0$NNI
  numPolys     := 0$NNI
  numConstructs := 0$NNI
  for component in children space.subspaceField repeat
    #(kid:=children component) = 1 =>
      #(children first kid) = 1 => numPts := numPts + 1
      numCurves := numCurves + 1
    (#kid = 2) and _
      (#children first kid = 1) and _
      (#children first rest kid ^= 1) =>
        numPolys := numPolys + 1
      numConstructs := numConstructs + 1
    -- otherwise, a mathematical surface is assumed
    -- there could also be garbage representation
    -- since there are always more permutations that
    -- we could ever want, so the user should not
    -- fumble around too much with the structure
    -- as other applications need to interpret it
    [numPts,numCurves,numPolys,numConstructs]

check(s) ==
  ^s.converted => convertSpace s
  s

subspace(s) == s.subspaceField

coerce(s) ==
  if ^s.converted then s := convertSpace s
  hconcat(["3-Space with " : 0, _
    (size:=#(s.rep3DField.l1l1Pt)) : 0, _
    (size=1=>" component" : 0;" components" : 0)])

-----

— SPACE3.dotabb —

"SPACE3" [color="#88FF44",href="bookvol10.3.pdf#nameddest=SPACE3"]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"SPACE3" -> "FSAGG"

-----

```

domain TREE Tree

— Tree.input —

```

)set break resume
)sys rm -f Tree.output
)spool Tree.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Tree
--R
--R Tree(S: SetCategory) is a domain constructor
--R Abbreviation for Tree is TREE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for TREE
--R
--R----- Operations -----
--R children : % -> List(%)          copy : % -> %
--R cyclic? : % -> Boolean           cyclicCopy : % -> %
--R cyclicEntries : % -> List(%)     cyclicEqual? : (%,%) -> Boolean
--R cyclicParents : % -> List(%)     distance : (%,%) -> Integer
--R ?.value : (%,value) -> S         empty : () -> %
--R empty? : % -> Boolean            eq? : (%,%) -> Boolean
--R leaf? : % -> Boolean             leaves : % -> List(S)
--R map : ((S -> S),%) -> %          nodes : % -> List(%)
--R sample : () -> %                tree : S -> %
--R tree : List(S) -> %              tree : (S,List(%)) -> %
--R value : % -> S
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ==? : (%,%) -> Boolean if S has SETCAT
--R any? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R child? : (%,%) -> Boolean if S has SETCAT
--R coerce : % -> OutputForm if S has SETCAT
--R count : (S,%) -> NonNegativeInteger if $ has finiteAggregate and S has SETCAT
--R count : ((S -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R eval : (%,List(S),List(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,S,S) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,Equation(S)) -> % if S has EVALAB(S) and S has SETCAT
--R eval : (%,List(Equation(S))) -> % if S has EVALAB(S) and S has SETCAT
--R every? : ((S -> Boolean),%) -> Boolean if $ has finiteAggregate
--R hash : % -> SingleInteger if S has SETCAT
--R latex : % -> String if S has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R map! : ((S -> S),%) -> % if $ has shallowlyMutable
--R member? : (S,%) -> Boolean if $ has finiteAggregate and S has SETCAT

```

```

--R members : % -> List(S) if $ has finiteAggregate
--R more? : (% , NonNegativeInteger) -> Boolean
--R node? : (% , %) -> Boolean if S has SETCAT
--R parts : % -> List(S) if $ has finiteAggregate
--R setchildren! : (% , List(%)) -> % if $ has shallowlyMutable
--R setelt : (% , value, S) -> S if $ has shallowlyMutable
--R setvalue! : (% , S) -> S if $ has shallowlyMutable
--R size? : (% , NonNegativeInteger) -> Boolean
--R ?~=? : (% , %) -> Boolean if S has SETCAT
--R
--E 1

)spool
)lisp (bye)

```

— Tree.help —

```

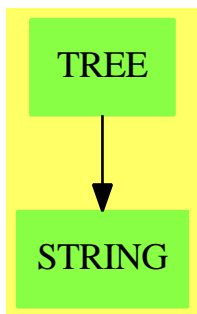
=====
Tree examples
=====

```

Tree(S) is a basic domains of tree structures. Each tree is either empty or else is a node consisting of a value and a list of (sub)trees.

See Also:
o)show Tree

Tree (TREE)



See

⇒ “BinaryTree” (BTREE) 3 on page 310

⇒ “BinarySearchTree” (BSTREE) [3](#) on page [302](#)
 ⇒ “BinaryTournament” (BTOURN) [3](#) on page [306](#)
 ⇒ “BalancedBinaryTree” (BBTREE) [3](#) on page [252](#)
 ⇒ “PendantTree” (PENDTREE) [17](#) on page [1944](#)

Exports:

any?	child?	children	coerce	copy
count	cyclic?	cyclicCopy	cyclicEntries	cyclicEqual?
cyclicParents	distance	empty	empty?	eq?
eval	every?	hash	latex	leaf?
leaves	less?	map	map!	member?
members	more?	node?	nodes	parts
sample	setchildren!	setelt	setvalue!	size?
tree	value	#?	?=?	?~=?
?.value				

— domain TREE Tree —

```
)abbrev domain TREE Tree
++ Author:W. H. Burge
++ Date Created:17 Feb 1992
++ Description:
++ \spadtype{Tree(S)} is a basic domains of tree structures.
++ Each tree is either empty or else is a node consisting of a value and
++ a list of (sub)trees.
```

```
Tree(S: SetCategory): T==C where
  T== RecursiveAggregate(S) with
    finiteAggregate
    shallowlyMutable
  tree: (S,List %) -> %
    ++ tree(nd,ls) creates a tree with value nd, and children ls.
    ++
    ++X t1:=tree [1,2,3,4]
    ++X tree(5,[t1])

  tree: List S -> %
    ++ tree(ls) creates a tree from a list of elements of s.
    ++
    ++X tree [1,2,3,4]

  tree: S -> %
    ++ tree(nd) creates a tree with value nd, and no children
    ++
    ++X tree 6

  cyclic?: % -> Boolean
    ++ cyclic?(t) tests if t is a cyclic tree.
    ++
```



```

++X t1:=tree [1,2,3,4]
++X cyclic? t1

cyclicCopy: % -> %
++ cyclicCopy(l) makes a copy of a (possibly) cyclic tree l.
++
++X t1:=tree [1,2,3,4]
++X cyclicCopy t1

cyclicEntries: % -> List %
++ cyclicEntries(t) returns a list of top-level cycles in tree t.
++
++X t1:=tree [1,2,3,4]
++X cyclicEntries t1

cyclicEqual?: (% , %) -> Boolean
++ cyclicEqual?(t1, t2) tests if two cyclic trees have
++ the same structure.
++
++X t1:=tree [1,2,3,4]
++X t2:=tree [1,2,3,4]
++X cyclicEqual?(t1,t2)

cyclicParents: % -> List %
++ cyclicParents(t) returns a list of cycles that are parents of t.
++
++X t1:=tree [1,2,3,4]
++X cyclicParents t1

C== add
cycleTreeMax ==> 5

Rep := Union(node:Record(value: S, args: List %),empty:"empty")
t:%
br:%
s: S
ls: List S
empty? t == t case empty
empty() == ["empty"]
children t ==
  t case empty => error "cannot take the children of an empty tree"
  (t.node.args)@List(%)
setchildren_(t,lt) ==
  t case empty => error "cannot set children of an empty tree"
  (t.node.args:=lt;t pretend %)
setvalue_(t,s) ==
  t case empty => error "cannot set value of an empty tree"
  (t.node.value:=s;s)
count(n, t) ==
  t case empty => 0

```

```

    i := +/[count(n, c) for c in children t]
    value t = n => i + 1
  i
count(fn: S -> Boolean, t: %): NonNegativeInteger ==
  t case empty => 0
  i := +/[count(fn, c) for c in children t]
  fn value t => i + 1
  i
map(fn, t) ==
  t case empty => t
  tree(fn value t, [map(fn, c) for c in children t])
map_!(fn, t) ==
  t case empty => t
  setvalue_!(t, fn value t)
  for c in children t repeat map_!(fn, c)
tree(s, lt) == [[s, lt]]
tree(s) == [[s, []]]
tree(ls) ==
  empty? ls => empty()
  tree(first ls, [tree s for s in rest ls])
value t ==
  t case empty => error "cannot take the value of an empty tree"
  t.node.value
child?(t1, t2) ==
  empty? t2 => false
  "or"/[t1 = t for t in children t2]
distance1(t1: %, t2: %): Integer ==
  t1 = t2 => 0
  t2 case empty => -1
  u := [n for t in children t2 | (n := distance1(t1, t)) >= 0]
  #u > 0 => 1 + "min"/u
  -1
distance(t1, t2) ==
  n := distance1(t1, t2)
  n >= 0 => n
  distance1(t2, t1)
node?(t1, t2) ==
  t1 = t2 => true
  t case empty => false
  "or"/[node?(t1, t) for t in children t2]
leaf? t ==
  t case empty => false
  empty? children t
leaves t ==
  t case empty => empty()
  leaf? t => [value t]
  "append"/[leaves c for c in children t]
less?(t, n) == # t < n
more?(t, n) == # t > n
nodes t ==      ---buggy

```

```

    t case empty => empty()
    nl := [nodes c for c in children t]
    nl = empty() => [t]
    cons(t,"append"/nl)
size? (t, n) == # t = n
any?(fn, t) == ---bug fixed
    t case empty => false
    fn value t or "or"/[any?(fn, c) for c in children t]
every?(fn, t) ==
    t case empty => true
    fn value t and "and"/[every?(fn, c) for c in children t]
member?(n, t) ==
    t case empty => false
    n = value t or "or"/[member?(n, c) for c in children t]
members t == parts t
parts t == --buggy?
    t case empty => empty()
    u := [parts c for c in children t]
    u = empty() => [value t]
    cons(value t,"append"/u)

---Functions that guard against cycles: =, #, copy-----

-----> =
equal?: (% , % , % , % , Integer) -> Boolean

t1 = t2 == equal?(t1, t2, t1, t2, 0)

equal?(t1, t2, ot1, ot2, k) ==
    k = cycleTreeMax and (cyclic? ot1 or cyclic? ot2) =>
        error "use cyclicEqual? to test equality on cyclic trees"
    t1 case empty => t2 case empty
    t2 case empty => false
    value t1 = value t2 and (c1 := children t1) = (c2 := children t2) and
        "and"/[equal?(x,y,ot1, ot2,k + 1) for x in c1 for y in c2]

-----> #
treeCount: (% , % , NonNegativeInteger) -> NonNegativeInteger
# t == treeCount(t, t, 0)
treeCount(t, origTree, k) ==
    k = cycleTreeMax and cyclic? origTree =>
        error "# is not defined on cyclic trees"
    t case empty => 0
    1 + +/[treeCount(c, origTree, k + 1) for c in children t]

-----> copy
copy1: (% , % , Integer) -> %
copy t == copy1(t, t, 0)
copy1(t, origTree, k) ==
    k = cycleTreeMax and cyclic? origTree =>

```

```

    error "use cyclicCopy to copy a cyclic tree"
  t case empty => t
  empty? children t => tree value t
  tree(value t, [copy1(x, origTree, k + 1) for x in children t])

-----Functions that allow cycles-----
--local utility functions:
eqUnion: (List %, List %) -> List %
eqMember?: (% , List %) -> Boolean
eqMemberIndex: (% , List %, Integer) -> Integer
lastNode: List % -> List %
insert: (% , List %) -> List %

-----> coerce to OutputForm
if S has SetCategory then
  multipleOverbar: (OutputForm, Integer, List %) -> OutputForm
  coerce1: (% , List %, List %) -> OutputForm

  coerce(t:%): OutputForm == coerce1(t, empty()$(List %), cyclicParents t)

  coerce1(t,parents, pl) ==
    t case empty => empty()@List(S)::OutputForm
    eqMember?(t, parents) =>
      multipleOverbar(("."::OutputForm,eqMemberIndex(t, pl,0),pl)
    empty? children t => value t::OutputForm
    nodeForm := (value t)::OutputForm
    if (k := eqMemberIndex(t, pl, 0)) > 0 then
      nodeForm := multipleOverbar(nodeForm, k, pl)
    prefix(nodeForm,
      [coerce1(br,cons(t,parents),pl) for br in children t])

  multipleOverbar(x, k, pl) ==
    k < 1 => x
    #pl = 1 => overbar x
    s : String := "abcdefghijklmnopqrstuvwxyz"
    c := s.(1 + ((k - 1) rem 26))
    overlable(c::OutputForm, x)

-----> cyclic?
cyclic2?: (% , List %) -> Boolean

cyclic? t == cyclic2?(t, empty()$(List %))

cyclic2?(x,parents) ==
  empty? x => false
  eqMember?(x, parents) => true
  for y in children x repeat
    cyclic2?(y,cons(x, parents)) => return true
  false

```

```

-----> cyclicCopy
cyclicCopy2: (% , List %) -> %
copyCycle2: (% , List %) -> %
copyCycle4: (% , % , % , List %) -> %

cyclicCopy(t) == cyclicCopy2(t, cyclicEntries t)

cyclicCopy2(t, cycles) ==
  eqMember?(t, cycles) => return copyCycle2(t, cycles)
  tree(value t, [cyclicCopy2(c, cycles) for c in children t])

copyCycle2(cycle, cycleList) ==
  newCycle := tree(value cycle, nil)
  setchildren!(newCycle,
    [copyCycle4(c, cycle, newCycle, cycleList) for c in children cycle])
  newCycle

copyCycle4(t, cycle, newCycle, cycleList) ==
  empty? cycle => empty()
  eq?(t, cycle) => newCycle
  eqMember?(t, cycleList) => copyCycle2(t, cycleList)
  tree(value t,
    [copyCycle4(c, cycle, newCycle, cycleList) for c in children t])

-----> cyclicEntries
cyclicEntries3: (% , List % , List %) -> List %

cyclicEntries(t) == cyclicEntries3(t, empty()$(List %), empty()$(List %))

cyclicEntries3(t, parents, cl) ==
  empty? t => cl
  eqMember?(t, parents) => insert(t, cl)
  parents := cons(t, parents)
  for y in children t repeat
    cl := cyclicEntries3(t, parents, cl)
  cl

-----> cyclicEqual?
cyclicEqual4?: (% , % , List % , List %) -> Boolean

cyclicEqual?(t1, t2) ==
  cp1 := cyclicParents t1
  cp2 := cyclicParents t2
  #cp1 ^= #cp2 or null cp1 => t1 = t2
  cyclicEqual4?(t1, t2, cp1, cp2)

cyclicEqual4?(t1, t2, cp1, cp2) ==
  t1 case empty => t2 case empty
  t2 case empty => false
  0 ^= (k := eqMemberIndex(t1, cp1, 0)) => eq?(t2, cp2 . k)

```

```

value t1 = value t2 and
  "and"/[cyclicEqual4?(x,y,cp1,cp2)
    for x in children t1 for y in children t2]

-----> cyclicParents t
cyclicParents3: (% , List % , List %) -> List %

cyclicParents t == cyclicParents3(t, empty()$(List %), empty()$(List %))

cyclicParents3(x, parents, pl) ==
  empty? x => pl
  eqMember?(x, parents) =>
    cycleMembers := [y for y in parents while not eq?(x,y)]
    eqUnion(cons(x, cycleMembers), pl)
  parents := cons(x, parents)
  for y in children x repeat
    pl := cyclicParents3(y, parents, pl)
  pl

insert(x, l) ==
  eqMember?(x, l) => l
  cons(x, l)

lastNode l ==
  empty? l => error "empty tree has no last node"
  while not empty? rest l repeat l := rest l
  l

eqMember?(y,l) ==
  for x in l repeat eq?(x,y) => return true
  false

eqMemberIndex(x, l, k) ==
  null l => k
  k := k + 1
  eq?(x, first l) => k
  eqMemberIndex(x, rest l, k)

eqUnion(u, v) ==
  null u => v
  x := first u
  newV :=
    eqMember?(x, v) => v
    cons(x, v)
  eqUnion(rest u, newV)

```

— TREE.dotabb —

```
"TREE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=TREE"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"TREE" -> "STRING"
```

domain TUBE TubePlot

— TubePlot.input —

```
)set break resume
)sys rm -f TubePlot.output
)spool TubePlot.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TubePlot
--R
--R TubePlot(Curve: PlottableSpaceCurveCategory) is a domain constructor
--R Abbreviation for TubePlot is TUBE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for TUBE
--R
--R----- Operations -----
--R closed? : % -> Boolean          getCurve : % -> Curve
--R open? : % -> Boolean
--R listLoops : % -> List(List(Point(DoubleFloat)))
--R setClosed : (%,Boolean) -> Boolean
--R tube : (Curve,List(List(Point(DoubleFloat))),Boolean) -> %
--R
--E 1

)spool
)lisp (bye)
```

— TubePlot.help —

=====

TubePlot examples

=====

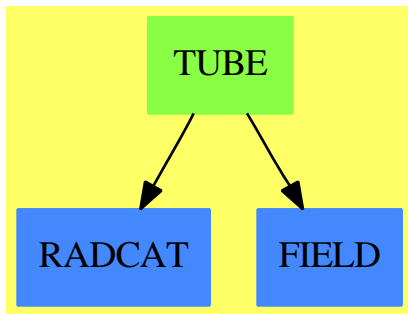
Package for constructing tubes around 3-dimensional parametric curves.
Domain of tubes around 3-dimensional parametric curves.

See Also:

o)show TubePlot

—————

TubePlot (TUBE)



Exports:

closed? getCurve listLoops open? setClosed tube

— domain TUBE TubePlot —

```

)abbrev domain TUBE TubePlot
++ Author: Clifton J. Williamson
++ Date Created: Bastille Day 1989
++ Date Last Updated: 5 June 1990
++ Description:
++ Package for constructing tubes around 3-dimensional parametric curves.
++ Domain of tubes around 3-dimensional parametric curves.
  
```

```

TubePlot(Curve): Exports == Implementation where
  Curve : PlottableSpaceCurveCategory
  B      ==> Boolean
  L      ==> List
  Pt     ==> Point DoubleFloat
  
```

```

Exports ==> with
  getCurve: % -> Curve
  
```



```

++ getCurve(t) returns the \spadtype{PlottableSpaceCurveCategory}
++ representing the parametric curve of the given tube plot t.
listLoops: % -> L L Pt
++ listLoops(t) returns the list of lists of points, or the 'loops',
++ of the given tube plot t.
closed?: % -> B
++ closed?(t) tests whether the given tube plot t is closed.
open?: % -> B
++ open?(t) tests whether the given tube plot t is open.
setClosed: (% ,B) -> B
++ setClosed(t,b) declares the given tube plot t to be closed if
++ b is true, or if b is false, t is set to be open.
tube: (Curve,L L Pt,B) -> %
++ tube(c,ll,b) creates a tube of the domain \spadtype{TubePlot} from a
++ space curve c of the category \spadtype{PlottableSpaceCurveCategory},
++ a list of lists of points (loops) ll and a boolean b which if
++ true indicates a closed tube, or if false an open tube.

Implementation ==> add

--% representation

Rep := Record(parCurve:Curve,loops:L L Pt,closedTube?:B)

getCurve plot == plot.parCurve

listLoops plot == plot.loops

closed? plot == plot.closedTube?
open? plot == not plot.closedTube?

setClosed(plot,flag) == plot.closedTube? := flag

tube(curve,ll,b) == [curve,ll,b]

_____

— TUBE.dotabb —

"TUBE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=TUBE"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"TUBE" -> "FIELD"
"TUBE" -> "RADCAT"

_____

```

domain TUPLE Tuple**— Tuple.input —**

```

)set break resume
)sys rm -f Tuple.output
)spool Tuple.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Tuple
--R
--R Tuple(S: Type) is a domain constructor
--R Abbreviation for Tuple is TUPLE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for TUPLE
--R
--R----- Operations -----
--R coerce : PrimitiveArray(S) -> %      coerce : % -> PrimitiveArray(S)
--R length : % -> NonNegativeInteger
--R ?=? : (%,% ) -> Boolean if S has SETCAT
--R coerce : % -> OutputForm if S has SETCAT
--R hash : % -> SingleInteger if S has SETCAT
--R latex : % -> String if S has SETCAT
--R select : (% , NonNegativeInteger) -> S
--R ?~=? : (% , %) -> Boolean if S has SETCAT
--R
--E 1

)spool
)lisp (bye)

```

— Tuple.help —

```

=====
Tuple examples
=====

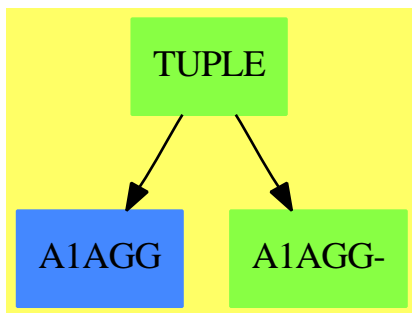
```

This domain is used to interface with the interpreter's notion of comma-delimited sequences of values.

See Also:

o)show Tuple

Tuple (TUPLE)



See

- ⇒ “PrimitiveArray” (PRIMARR) [17](#) on page [2109](#)
- ⇒ “IndexedFlexibleArray” (IFARRAY) [10](#) on page [1222](#)
- ⇒ “FlexibleArray” (FARRAY) [7](#) on page [885](#)
- ⇒ “IndexedOneDimensionalArray” (IARRAY1) [10](#) on page [1243](#)
- ⇒ “OneDimensionalArray” (ARRAY1) [16](#) on page [1774](#)

Exports:

coerce hash latex length select ?=? ?~=?

— domain TUPLE Tuple —

```

)abbrev domain TUPLE Tuple
++ Author: Mark Botch
++ Description:
++ This domain is used to interface with the interpreter's notion
++ of comma-delimited sequences of values.

```

```

Tuple(S:Type): CoercibleTo(PrimitiveArray S) with
  coerce: PrimitiveArray S -> %
  ++ coerce(a) makes a tuple from primitive array a
  ++
  ++X t1:PrimitiveArray(Integer):= [i for i in 1..10]
  ++X t2:=coerce(t1)$Tuple(Integer)

```

```

select: (% , NonNegativeInteger) -> S
  ++ select(x,n) returns the n-th element of tuple x.
  ++ tuples are 0-based
  ++
  ++X t1:PrimitiveArray(Integer):= [i for i in 1..10]
  ++X t2:=coerce(t1)$Tuple(Integer)
  ++X select(t2,3)

```

```

length: % -> NonNegativeInteger
++ length(x) returns the number of elements in tuple x
++
++X t1:PrimitiveArray(Integer):= [i for i in 1..10]
++X t2:=coerce(t1)$Tuple(Integer)
++X length(t2)

if S has SetCategory then SetCategory
== add
Rep := Record(len : NonNegativeInteger, elts : PrimitiveArray S)

coerce(x: PrimitiveArray S): % == [#x, x]
coerce(x:%): PrimitiveArray(S) == x.elts
length x == x.len

select(x, n) ==
  n >= x.len => error "Index out of bounds"
  x.elts.n

if S has SetCategory then
  x = y == (x.len = y.len) and (x.elts =$PrimitiveArray(S) y.elts)
  coerce(x : %): OutputForm ==
    paren [(x.elts.i)::OutputForm
      for i in minIndex x.elts .. maxIndex x.elts]$List(OutputForm)

```

— TUPLE.dotabb —

```

"TUPLE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=TUPLE"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"A1AGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=A1AGG"]
"TUPLE" -> "A1AGG"
"TUPLE" -> "A1AGG-"

```

domain ARRAY2 TwoDimensionalArray

— TwoDimensionalArray.input —

```

)set break resume
)sys rm -f TwoDimensionalArray.output

```

```

)spool TwoDimensionalArray.output
)set message test on
)set message auto off
)clear all
--S 1 of 20
arr : ARRAY2 INT := new(5,4,0)

```

```

--R
--R
--R      +0  0  0  0+
--R      |      |
--R      |0  0  0  0|
--R      |      |
--R  (1) |0  0  0  0|
--R      |      |
--R      |0  0  0  0|
--R      |      |
--R      +0  0  0  0+

```

Type: TwoDimensionalArray(Integer)

```

--E 1

```

```

--S 2 of 20
setelt(arr,1,1,17)

```

```

--R
--R
--R  (2)  17
--R
--E 2

```

Type: PositiveInteger

```

--S 3 of 20
arr

```

```

--R
--R
--R      +17  0  0  0+
--R      |      |
--R      |0  0  0  0|
--R      |      |
--R  (3) |0  0  0  0|
--R      |      |
--R      |0  0  0  0|
--R      |      |
--R      +0  0  0  0+

```

Type: TwoDimensionalArray(Integer)

```

--E 3

```

```

--S 4 of 20
elt(arr,1,1)

```

```

--R
--R
--R  (4)  17
--R

```

Type: PositiveInteger

--E 4

--S 5 of 20

arr(3,2) := 15

--R

--R

--R (5) 15

--R

Type: PositiveInteger

--E 5

--S 6 of 20

arr(3,2)

--R

--R

--R (6) 15

--R

Type: PositiveInteger

--E 6

--S 7 of 20

row(arr,1)

--R

--R

--R (7) [17,0,0,0]

--R

Type: OneDimensionalArray(Integer)

--E 7

--S 8 of 20

column(arr,1)

--R

--R

--R (8) [17,0,0,0,0]

--R

Type: OneDimensionalArray(Integer)

--E 8

--S 9 of 20

nrows(arr)

--R

--R

--R (9) 5

--R

Type: PositiveInteger

--E 9

--S 10 of 20

ncols(arr)

--R

--R

--R (10) 4

--R

Type: PositiveInteger

--E 10

--S 11 of 20

map(-,arr)

--R

--R

--R +- 17 0 0 0+

--R | |

--R | 0 0 0 0|

--R | |

--R (11) | 0 - 15 0 0|

--R | |

--R | 0 0 0 0|

--R | |

--R + 0 0 0 0+

--R

--E 11

Type: TwoDimensionalArray(Integer)

--S 12 of 20

map((x +-> x + x),arr)

--R

--R

--R +34 0 0 0+

--R | |

--R |0 0 0 0|

--R | |

--R (12) |0 30 0 0|

--R | |

--R |0 0 0 0|

--R | |

--R +0 0 0 0+

--R

--E 12

Type: TwoDimensionalArray(Integer)

--S 13 of 20

arrc := copy(arr)

--R

--R

--R +17 0 0 0+

--R | |

--R |0 0 0 0|

--R | |

--R (13) |0 15 0 0|

--R | |

--R |0 0 0 0|

--R | |

--R +0 0 0 0+

--R

--E 13

Type: TwoDimensionalArray(Integer)

--S 14 of 20

map!(-,arrc)

```

--R
--R
--R      +- 17  0  0  0+
--R      |
--R      | 0  0  0  0|
--R      |
--R      (14) | 0  - 15  0  0|
--R      |
--R      | 0  0  0  0|
--R      |
--R      + 0  0  0  0+
--R
--E 14

```

Type: TwoDimensionalArray(Integer)

```

--S 15 of 20
arrc
--R
--R
--R      +- 17  0  0  0+
--R      |
--R      | 0  0  0  0|
--R      |
--R      (15) | 0  - 15  0  0|
--R      |
--R      | 0  0  0  0|
--R      |
--R      + 0  0  0  0+
--R
--E 15

```

Type: TwoDimensionalArray(Integer)

```

--S 16 of 20
arr
--R
--R
--R      +17  0  0  0+
--R      |
--R      |0  0  0  0|
--R      |
--R      (16) |0  15  0  0|
--R      |
--R      |0  0  0  0|
--R      |
--R      +0  0  0  0+
--R
--E 16

```

Type: TwoDimensionalArray(Integer)

```

--S 17 of 20
member?(17,arr)
--R
--R

```



```

--R (17) true
--R
--R                                          Type: Boolean
--E 17

--S 18 of 20
member?(10317,arr)
--R
--R
--R (18) false
--R
--R                                          Type: Boolean
--E 18

--S 19 of 20
count(17,arr)
--R
--R
--R (19) 1
--R
--R                                          Type: PositiveInteger
--E 19

--S 20 of 20
count(0,arr)
--R
--R
--R (20) 18
--R
--R                                          Type: PositiveInteger
--E 20
)spool
)lisp (bye)

```

— TwoDimensionalArray.help —

```

=====
TwoDimensionalArray examples
=====

```

The TwoDimensionalArray domain is used for storing data in a two dimensional data structure indexed by row and by column. Such an array is a homogeneous data structure in that all the entries of the array must belong to the same Axiom domain.. Each array has a fixed number of rows and columns specified by the user and arrays are not extensible. In Axiom, the indexing of two-dimensional arrays is one-based. This means that both the "first" row of an array and the "first" column of an array are given the index 1. Thus, the entry in the upper left corner of an array is in position (1,1).

The operation new creates an array with a specified number of rows and

columns and fills the components of that array with a specified entry. The arguments of this operation specify the number of rows, the number of columns, and the entry.

This creates a five-by-four array of integers, all of whose entries are zero.

```
arr : ARRAY2 INT := new(5,4,0)
+0  0  0  0+
|
|0  0  0  0|
|
|0  0  0  0|
|
|0  0  0  0|
|
+0  0  0  0+
```

Type: TwoDimensionalArray Integer

The entries of this array can be set to other integers using `setelt`.

Issue this to set the element in the upper left corner of this array to 17.

```
setelt(arr,1,1,17)
17
```

Type: PositiveInteger

Now the first element of the array is 17.

```
arr
+17  0  0  0+
|
|0  0  0  0|
|
|0  0  0  0|
|
|0  0  0  0|
|
+0  0  0  0+
```

Type: TwoDimensionalArray Integer

Likewise, elements of an array are extracted using the operation `elt`.

```
elt(arr,1,1)
17
```

Type: PositiveInteger

Another way to use these two operations is as follows. This sets the element in position (3,2) of the array to 15.

```
arr(3,2) := 15
15
```

Type: PositiveInteger

This extracts the element in position (3,2) of the array.

```
arr(3,2)
15
```

Type: PositiveInteger

The operations `elt` and `setelt` come equipped with an error check which verifies that the indices are in the proper ranges. For example, the above array has five rows and four columns, so if you ask for the entry in position (6,2) with `arr(6,2)` Axiom displays an error message. If there is no need for an error check, you can call the operations `qelt` and `qsetelt` which provide the same functionality but without the error check. Typically, these operations are called in well-tested programs.

The operations `row` and `column` extract rows and columns, respectively, and return objects of `OneDimensionalArray` with the same underlying element type.

```
row(arr,1)
[17,0,0,0]
```

Type: OneDimensionalArray Integer

```
column(arr,1)
[17,0,0,0,0]
```

Type: OneDimensionalArray Integer

You can determine the dimensions of an array by calling the operations `nrows` and `ncols`, which return the number of rows and columns, respectively.

```
nrows(arr)
5
```

Type: PositiveInteger

```
ncols(arr)
4
```

Type: PositiveInteger

To apply an operation to every element of an array, use `map`. This creates a new array. This expression negates every element.

```
map(-,arr)
+- 17  0  0  0+
|      |
| 0    0  0  0|
|      |
| 0  - 15 0  0|
```

```

|
| 0    0    0 0|
|
+ 0    0    0 0+
Type: TwoDimensionalArray Integer

```

This creates an array where all the elements are doubled.

```

map((x +-> x + x),arr)
+34 0    0    0+
|
| 0    0    0 0|
|
| 0    30   0 0|
|
| 0    0    0 0|
|
+0   0    0    0+
Type: TwoDimensionalArray Integer

```

To change the array destructively, use `map` instead of `map`. If you need to make a copy of any array, use `copy`.

```

arrc := copy(arr)
+17 0    0    0+
|
| 0    0    0 0|
|
| 0    15   0 0|
|
| 0    0    0 0|
|
+0   0    0    0+
Type: TwoDimensionalArray Integer

```

```

map!(-,arrc)
+- 17 0    0    0+
|
| 0    0    0 0|
|
| 0    - 15  0 0|
|
| 0    0    0 0|
|
+ 0    0    0 0+
Type: TwoDimensionalArray Integer

```

```

arrc
+- 17 0    0    0+
|
|

```

```

| 0      0      0  0|
|          |
| 0      - 15    0  0|
|          |
| 0      0      0  0|
|          |
+ 0      0      0  0+
Type: TwoDimensionalArray Integer

```

```

arr
+17  0      0  0+
|          |
| 0      0      0  0|
|          |
| 0      15    0  0|
|          |
| 0      0      0  0|
|          |
+0    0      0  0+
Type: TwoDimensionalArray Integer

```

Use `member?` to see if a given element is in an array.

```

member?(17,arr)
true
Type: Boolean

```

```

member?(10317,arr)
false
Type: Boolean

```

To see how many times an element appears in an array, use `count`.

```

count(17,arr)
1
Type: PositiveInteger

```

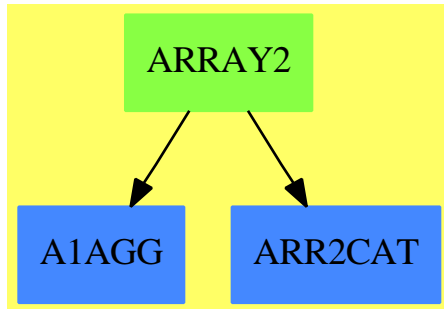
```

count(0,arr)
18
Type: PositiveInteger

```

See Also:

- o `)help Matrix`
- o `)help OneDimensionalArray`
- o `)show TwoDimensionalArray`

TwoDimensionalArray (ARRAY2)**See**⇒ “InnerIndexedTwoDimensionalArray” (IIARRAY2) [10](#) on page [1290](#)⇒ “IndexedTwoDimensionalArray” (IARRAY2) [10](#) on page [1256](#)**Exports:**

any?	coerce	column	copy	count
elt	empty	empty?	eq?	eval
every?	fill!	hash	latex	less?
map	map!	maxColIndex	maxRowIndex	member?
members	more?	minColIndex	minRowIndex	ncols
new	nrows	parts	qelt	qsetelt!
row	sample	setColumn!	setRow!	setelt
size?	#?	?=?	?~=?	

— domain **ARRAY2 TwoDimensionalArray** —

```

)abbrev domain ARRAY2 TwoDimensionalArray
++ Author: Mark Botch
++ Description:
++ A TwoDimensionalArray is a two dimensional array with
++ 1-based indexing for both rows and columns.

TwoDimensionalArray(R):Exports == Implementation where
  R : Type
  Row ==> OneDimensionalArray R
  Col ==> OneDimensionalArray R

Exports ==> TwoDimensionalArrayCategory(R,Row,Col) with
  shallowlyMutable
  ++ One may destructively alter TwoDimensionalArray's.

Implementation ==> InnerIndexedTwoDimensionalArray(R,1,1,Row,Col)

```

— ARRAY2.dotabb —

```
"ARRAY2" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ARRAY2"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"ARR2CAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ARR2CAT"]
"ARRAY2" -> "ARR2CAT"
"ARRAY2" -> "A1AGG"
```

domain VIEW2D TwoDimensionalViewport

— TwoDimensionalViewport.input —

```
)set break resume
)sys rm -f TwoDimensionalViewport.output
)spool TwoDimensionalViewport.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TwoDimensionalViewport
--E 1

)spool
)lisp (bye)
```

— TwoDimensionalViewport.help —

```
=====
TwoDimensionalViewport examples
=====
```

We want to graph $x^3 * (a+b*x)$ on the interval $x=-1..1$
 so we clear out the workspace

We assign values to the constants

```
a:=0.5
0.5
```

Type: Float

```
b:=0.5
0.5
```

```
Type: Float
```

We draw the first case of the graph

```
y1:=draw(x^3*(a+b*x),x=-1..1,title=="2.2.10 explicit")
TwoDimensionalViewport: "2.2.10 explicit"
Type: TwoDimensionalViewport
```

We fetch the graph of the first object

```
g1:=getGraph(y1,1)
Graph with 1 point list
Type: GraphImage
```

We extract its points

```
pointLists g1
[
  [[-1.,0.,1.,3.], [-0.9583333333333337,-1.8336166570216028E-2,1.,3.],
  [-0.91666666666666674,-3.2093942901234518E-2,1.,3.],
  [-0.87500000000000011,-4.18701171875E-2,1.,3.],
  [-0.83333333333333348,-4.8225308641975301E-2,1.,3.],
  [-0.79166666666666685,-5.1683967496141986E-2,1.,3.],
  [-0.75000000000000022,-5.2734375E-2,1.,3.],
  [-0.70833333333333359,-5.1828643422067916E-2,1.,3.],
  [-0.66666666666666696,-4.9382716049382741E-2,1.,3.],
  [-0.62500000000000033,-4.5776367187500042E-2,1.,3.],
  [-0.5833333333333337,-4.1353202160493867E-2,1.,3.],
  [-0.54166666666666707,-3.6420657310956832E-2,1.,3.],
  [-0.50000000000000044,-3.1250000000000056E-2,1.,3.],
  [-0.45833333333333376,-2.6076328607253136E-2,1.,3.],
  [-0.41666666666666707,-2.1098572530864244E-2,1.,3.],
  [-0.37500000000000039,-1.6479492187500042E-2,1.,3.],
  [-0.3333333333333337,-1.2345679012345713E-2,1.,3.],
  [-0.29166666666666702,-8.7875554591049648E-3,1.,3.],
  [-0.25000000000000033,-5.8593750000000208E-3,1.,3.],
  [-0.20833333333333368,-3.5792221257716214E-3,1.,3.],
  [-0.16666666666666702,-1.9290123456790237E-3,1.,3.],
  [-0.12500000000000036,-8.5449218750000705E-4,1.,3.],
  [-8.3333333333333703E-2,-2.6523919753086765E-4,1.,3.],
  [-4.1666666666667039E-2,-3.4661940586420673E-5,1.,3.],
  [-3.7470027081099033E-16,-2.6304013894372334E-47,1.,3.],
  [4.166666666666629E-2,3.7676022376542178E-5,1.,3.],
  [8.3333333333332954E-2,3.1346450617283515E-4,1.,3.],
  [0.1249999999999961,1.0986328124999894E-3,1.,3.],
  [0.16666666666666627,2.7006172839505972E-3,1.,3.],
  [0.20833333333333293,5.463023244598731E-3,1.,3.],
  [0.2499999999999958,9.76562499999948E-3,1.,3.],
```



```

[0.291666666666666624,1.6024365837191284E-2,1.,3.],
[0.333333333333333293,2.469135802469126E-2,1.,3.],
[0.37499999999999961,3.6254882812499882E-2,1.,3.],
[0.41666666666666663,5.1239390432098617E-2,1.,3.],
[0.45833333333333298,7.0205500096450435E-2,1.,3.],
[0.49999999999999967,9.374999999999792E-2,1.,3.],
[0.54166666666666663,0.12250584731867258,1.,3.],
[0.58333333333333293,0.15714216820987617,1.,3.],
[0.62499999999999956,0.1983642578124995,1.,3.],
[0.66666666666666619,0.24691358024691298,1.,3.],
[0.70833333333333282,0.30356776861496837,1.,3.],
[0.74999999999999944,0.369140624999999,1.,3.],
[0.79166666666666607,0.44448212046681984,1.,3.],
[0.8333333333333327,0.530478395061727,1.,3.],
[0.87499999999999933,0.62805175781249845,1.,3.],
[0.91666666666666596,0.73816068672839308,1.,3.],
[0.95833333333333259,0.86179982880015205,1.,3.], [1.,1.,1.,3.]]
]

```

Type: List List Point DoubleFloat

Now we create a second graph with a changed parameter

```

b:=1.0
1.0

```

Type: Float

We draw it

```

y2:=draw(x^3*(a+b*x),x=-1..1)
TwoDimensionalViewport: "AXIOM2D"
Type: TwoDimensionalViewport

```

We fetch this new graph

```

g2:=getGraph(y2,1)
Graph with 1 point list
Type: GraphImage

```

We get the points from this graph

```

pointLists g2
[
  [[-1.,0.5,1.,3.], [-0.9583333333333337,0.40339566454475323,1.,3.],
  [-0.91666666666666674,0.32093942901234584,1.,3.],
  [-0.87500000000000011,0.25122070312500017,1.,3.],
  [-0.83333333333333348,0.19290123456790137,1.,3.],
  [-0.79166666666666685,0.14471510898919768,1.,3.],
  [-0.75000000000000022,0.10546875000000019,1.,3.],
  [-0.70833333333333359,7.404091917438288E-2,1.,3.],
  [-0.66666666666666696,4.938271604938288E-2,1.,3.],

```

```

[-0.625000000000000033,3.0517578125000125E-2,1.,3.],
[-0.5833333333333337,1.6541280864197649E-2,1.,3.],
[-0.54166666666666707,6.6219376929013279E-3,1.,3.],
[-0.500000000000000044,5.5511151231257827E-17,1.,3.],
[-0.45833333333333376,-4.011742862654287E-3,1.,3.],
[-0.41666666666666707,-6.0281635802469057E-3,1.,3.],
[-0.375000000000000039,-6.5917968750000035E-3,1.,3.],
[-0.3333333333333337,-6.1728395061728461E-3,1.,3.],
[-0.29166666666666702,-5.1691502700617377E-3,1.,3.],
[-0.250000000000000033,-3.9062500000000104E-3,1.,3.],
[-0.20833333333333368,-2.6373215663580349E-3,1.,3.],
[-0.16666666666666702,-1.543209876543218E-3,1.,3.],
[-0.125000000000000036,-7.3242187500000564E-4,1.,3.],
[-8.3333333333333703E-2,-2.4112654320987957E-4,1.,3.],
[-4.1666666666667039E-2,-3.315489969135889E-5,1.,3.],
[-3.7470027081099033E-16,-2.6304013894372324E-47,1.,3.],
[4.16666666666629E-2,3.9183063271603852E-5,1.,3.],
[8.333333333332954E-2,3.3757716049382237E-4,1.,3.],
[0.1249999999999961,1.2207031249999879E-3,1.,3.],
[0.1666666666666627,3.0864197530863957E-3,1.,3.],
[0.2083333333333293,6.4049238040123045E-3,1.,3.],
[0.2499999999999958,1.171874999999934E-2,1.,3.],
[0.2916666666666624,1.9642771026234473E-2,1.,3.],
[0.3333333333333293,3.0864197530864071E-2,1.,3.],
[0.3749999999999961,4.6142578124999847E-2,1.,3.],
[0.416666666666663,6.6309799382715848E-2,1.,3.],
[0.4583333333333298,9.2270085841049135E-2,1.,3.],
[0.4999999999999967,0.1249999999999971,1.,3.],
[0.541666666666663,0.16554844232253049,1.,3.],
[0.5833333333333293,0.21503665123456736,1.,3.],
[0.6249999999999956,0.27465820312499928,1.,3.],
[0.6666666666666619,0.3456790123456781,1.,3.],
[0.7083333333333282,0.42943733121141858,1.,3.],
[0.7499999999999944,0.52734374999999845,1.,3.],
[0.7916666666666607,0.64088119695215873,1.,3.],
[0.833333333333327,0.77160493827160281,1.,3.],
[0.8749999999999933,0.92114257812499756,1.,3.],
[0.9166666666666596,1.0911940586419722,1.,3.],
[0.9583333333333259,1.2835316599151199,1.,3.], [1.,1.5,1.,3.]]
]

```

Type: List List Point DoubleFloat

and we put these points, g2 onto the first graph y1 as graph 2

```
putGraph(y1,g2,2)
```

Type: Void

And now we do the whole sequence again

```
b:=2.0
```

```

2.0
                                Type: Float

y3:=draw(x^3*(a+b*x),x=-1..1)
    TwoDimensionalViewport: "AXIOM2D"
                                Type: TwoDimensionalViewport

g3:=getGraph(y3,1)
    Graph with 1 point list
                                Type: GraphImage

pointLists g3
[
  [[-1.,1.5,1.,3.], [-0.9583333333333337,1.2468593267746917,1.,3.],
  [-0.9166666666666667,1.0270061728395066,1.,3.],
  [-0.8750000000000001,0.8374023437500004,1.,3.],
  [-0.8333333333333348,0.6751543209876547,1.,3.],
  [-0.7916666666666685,0.53751326195987703,1.,3.],
  [-0.7500000000000002,0.4218750000000056,1.,3.],
  [-0.7083333333333359,0.32578004436728447,1.,3.],
  [-0.6666666666666696,0.24691358024691412,1.,3.],
  [-0.6250000000000003,0.1831054687500004,1.,3.],
  [-0.5833333333333337,0.1323302469135807,1.,3.],
  [-0.541666666666667,9.2707127700617648E-2,1.,3.],
  [-0.5000000000000004,6.2500000000000278E-2,1.,3.],
  [-0.4583333333333376,4.0117428626543411E-2,1.,3.],
  [-0.416666666666667,2.4112654320987775E-2,1.,3.],
  [-0.3750000000000003,1.3183593750000073E-2,1.,3.],
  [-0.3333333333333337,6.1728395061728877E-3,1.,3.],
  [-0.291666666666667,2.0676601080247183E-3,1.,3.],
  [-0.2500000000000003,1.0408340855860843E-17,1.,3.],
  [-0.20833333333333368,-7.5352044753086191E-4,1.,3.],
  [-0.166666666666667,2.0676601080247183E-4,1.,3.],
  [-0.12500000000000036,-4.8828125000000282E-4,1.,3.],
  [-8.3333333333333703E-2,-1.9290123456790339E-4,1.,3.],
  [-4.1666666666667039E-2,-3.0140817901235325E-5,1.,3.],
  [-3.7470027081099033E-16,-2.6304013894372305E-47,1.,3.],
  [4.166666666666629E-2,4.21971450617272E-5,1.,3.],
  [8.3333333333332954E-2,3.8580246913579681E-4,1.,3.],
  [0.1249999999999961,1.4648437499999848E-3,1.,3.],
  [0.1666666666666627,3.8580246913579933E-3,1.,3.],
  [0.2083333333333293,8.2887249228394497E-3,1.,3.],
  [0.2499999999999958,1.56249999999991E-2,1.,3.],
  [0.2916666666666624,2.6879581404320851E-2,1.,3.],
  [0.3333333333333293,4.3209876543209694E-2,1.,3.],
  [0.3749999999999961,6.5917968749999764E-2,1.,3.],
  [0.416666666666663,9.6450617283950296E-2,1.,3.],
  [0.4583333333333298,0.13639925733024652,1.,3.],
  [0.4999999999999967,0.1874999999999956,1.,3.],
  [0.541666666666663,0.25163363233024633,1.,3.],

```

```

[0.58333333333333293,0.33082561728394977,1.,3.],
[0.62499999999999956,0.42724609374999883,1.,3.],
[0.66666666666666619,0.5432098765432084,1.,3.],
[0.70833333333333282,0.68117645640431912,1.,3.],
[0.74999999999999944,0.8437499999999756,1.,3.],
[0.79166666666666607,1.0336793499228365,1.,3.],
[0.8333333333333327,1.2538580246913544,1.,3.],
[0.87499999999999933,1.507324218749996,1.,3.],
[0.91666666666666596,1.7972608024691306,1.,3.],
[0.95833333333333259,2.1269953221450555,1.,3.], [1.,2.5,1.,3.]]
]

```

Type: List List Point DoubleFloat

and put the third graphs points g3 onto the first graph y1 as graph 3

```
putGraph(y1,g3,3)
```

Type: Void

Finally we show the combined result

```

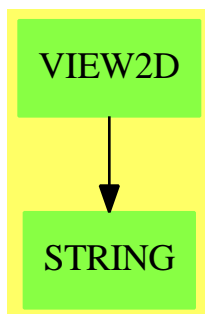
vp:=makeViewport2D(y1)
TwoDimensionalViewport: "2.2.10 explicit"
Type: TwoDimensionalViewport

```

See Also:

o)show TwoDimensionalViewport

TwoDimensionalViewport (VIEW2D)



Exports:

axes	close	coerce	connect	controlPanel
dimensions	getGraph	getPickedPoints	graphState	graphStates
graphs	hash	key	latex	makeViewport2D
move	options	points	putGraph	region
reset	resize	scale	show	title
translate	units	update	viewport2D	write
?=?	?~=?			

— domain VIEW2D TwoDimensionalViewport —

```
)abbrev domain VIEW2D TwoDimensionalViewport
++ Author: Jim Wen
++ Date Created: 28 April 1989
++ Date Last Updated: 29 October 1991, Jon Steinbach
++ Description:
++ TwoDimensionalViewport creates viewports to display graphs.
```

```
TwoDimensionalViewport ():Exports == Implementation where
```

```
VIEW    ==> VIEWPORTSERVER$Lisp
sendI    ==> SOCK_-SEND_-INT
sendSF   ==> SOCK_-SEND_-FLOAT
sendSTR  ==> SOCK_-SEND_-STRING
getI     ==> SOCK_-GET_-INT
getSF    ==> SOCK_-GET_-FLOAT

typeGRAPH ==> 2
typeVIEW2D ==> 3

makeGRAPH ==> (-1)$SingleInteger
makeVIEW2D ==> (-1)$SingleInteger

I    ==> Integer
PI   ==> PositiveInteger
NNI  ==> NonNegativeInteger
XY   ==> Record( X:I, Y:I )
XYP  ==> Record( X:PI, Y:PI )
XYNN ==> Record( X:NNI, Y:NNI )
F    ==> Float
SF   ==> DoubleFloat
STR  ==> String
L    ==> List
V    ==> Vector
E    ==> OutputForm
FLAG ==> Record( showCP:I )
PAL  ==> Palette()
B    ==> Boolean
G    ==> GraphImage
```

```

GS  ==> Record( scaleX:SF, scaleY:SF, deltaX:SF, deltaY:SF, _
              points:I, connect:I, spline:I, _
              axes:I, axesColor:PAL, units:I, unitsColor:PAL, _
              showing:I)
GU  ==> Union(G,"undefined")
DROP ==> DrawOption
POINT ==> Point(SF)

TRANSLATE2D ==> 0$I
SCALE2D ==> 1$I
pointsOnOff ==> 2
connectOnOff ==> 3
spline2D ==> 4 -- used for controlling regions, now
reset2D ==> 5
hideControl2D ==> 6
closeAll2D ==> 7
axesOnOff2D ==> 8
unitsOnOff2D ==> 9

SPADBUTTONPRESS ==> 100
MOVE ==> 102
RESIZE ==> 103
TITLE ==> 104
showing2D ==> 105 -- as defined in include/actions.h
putGraph2D ==> 106
writeView ==> 110
axesColor2D ==> 112
unitsColor2D ==> 113
getPickedPTS ==> 119

graphStart ==> 13 -- as defined in include/actions.h

noControl ==> 0$I

yes ==> 1$I
no ==> 0$I

maxGRAPHS ==> 9::I -- should be the same as maxGraphs in include/view2d.h

fileTypeDefs ==> ["PIXMAP"] -- see include/write.h for things to include

Exports ==> SetCategory with
  getPickedPoints : $ -> L POINT
  ++ getPickedPoints(x)
  ++ returns a list of small floats for the points the
  ++ user interactively picked on the viewport
  ++ for full integration into the system, some design
  ++ issues need to be addressed: e.g. how to go through
  ++ the GraphImage interface, how to default to graphs, etc.
viewport2D : () -> $

```

```

++ viewport2D() returns an undefined two-dimensional viewport
++ of the domain \spadtype{TwoDimensionalViewport} whose
++ contents are empty.
makeViewport2D : $                                -> $
++ makeViewport2D(v) takes the given two-dimensional viewport,
++ v, of the domain \spadtype{TwoDimensionalViewport} and
++ displays a viewport window on the screen which contains
++ the contents of v.
options      : $                                -> L DROP
++ options(v) takes the given two-dimensional viewport, v, of the
++ domain \spadtype{TwoDimensionalViewport} and returns a list
++ containing the draw options from the domain \spadtype{DrawOption}
++ for v.
options      : ($,L DROP)                        -> $
++ options(v,lopt) takes the given two-dimensional viewport, v,
++ of the domain \spadtype{TwoDimensionalViewport} and returns
++ v with it's draw options modified to be those which are indicated
++ in the given list, \spad{lopt} of domain \spadtype{DrawOption}.
makeViewport2D : (G,L DROP)                       -> $
++ makeViewport2D(gi,lopt) creates and displays a viewport window
++ of the domain \spadtype{TwoDimensionalViewport} whose graph
++ field is assigned to be the given graph, \spad{gi}, of domain
++ \spadtype{GraphImage}, and whose options field is set to be
++ the list of options, \spad{lopt} of domain \spadtype{DrawOption}.
graphState   : ($,PI,SF,SF,SF,SF,I,I,I,I,PAL,I,PAL,I) -> Void
++ graphState(v,num,sX,sY,dX,dY,pts,lns,box,axes,axesC,un,unC,cP)
++ sets the state of the characteristics for the graph indicated
++ by \spad{num} in the given two-dimensional viewport v, of domain
++ \spadtype{TwoDimensionalViewport}, to the values given as
++ parameters. The scaling of the graph in the x and y component
++ directions is set to be \spad{sX} and \spad{sY}; the window
++ translation in the x and y component directions is set to be
++ \spad{dX} and \spad{dY}; The graph points, lines, bounding box,
++ axes, or units will be shown in the viewport if their given
++ parameters \spad{pts}, \spad{lns}, \spad{box}, \spad{axes} or
++ \spad{un} are set to be \spad{1}, but will not be shown if they
++ are set to \spad{0}. The color of the axes and the color of the
++ units are indicated by the palette colors \spad{axesC} and
++ \spad{unC} respectively. To display the control panel when
++ the viewport window is displayed, set \spad{cP} to \spad{1},
++ otherwise set it to \spad{0}.
graphStates  : $                                -> V GS
++ graphStates(v) returns and shows a listing of a record containing
++ the current state of the characteristics of each of the ten graph
++ records in the given two-dimensional viewport, v, which is of
++ domain \spadtype{TwoDimensionalViewport}.
graphs      : $                                -> V GU
++ graphs(v) returns a vector, or list, which is a union of all
++ the graphs, of the domain \spadtype{GraphImage}, which are
++ allocated for the two-dimensional viewport, v, of domain

```

```

++ \spadtype{TwoDimensionalViewport}. Those graphs which have
++ no data are labeled "undefined", otherwise their contents
++ are shown.
title      : ($,STR)                                -> Void
++ title(v,s) changes the title which is shown in the two-dimensional
++ viewport window, v of domain \spadtype{TwoDimensionalViewport}.
putGraph   : ($,G,PI)                                -> Void
++ putGraph(v,gi,n) sets the graph field indicated by n, of the
++ indicated two-dimensional viewport, v, which is of domain
++ \spadtype{TwoDimensionalViewport}, to be the graph, \spad{gi}
++ of domain \spadtype{GraphImage}. The contents of viewport, v,
++ will contain \spad{gi} when the function \spadfun{makeViewport2D}
++ is called to create the an updated viewport v.
getGraph   : ($,PI)                                  -> G
++ getGraph(v,n) returns the graph which is of the domain
++ \spadtype{GraphImage} which is located in graph field n
++ of the given two-dimensional viewport, v, which is of the
++ domain \spadtype{TwoDimensionalViewport}.
axes       : ($,PI,STR)                              -> Void
++ axes(v,n,s) displays the axes of the graph in field n of
++ the given two-dimensional viewport, v, which is of domain
++ \spadtype{TwoDimensionalViewport}, if s is "on", or does
++ not display the axes if s is "off".
axes       : ($,PI,PAL)                              -> Void
++ axes(v,n,c) displays the axes of the graph in field n of
++ the given two-dimensional viewport, v, which is of domain
++ \spadtype{TwoDimensionalViewport}, with the axes color set to
++ the given palette color c.
units      : ($,PI,STR)                              -> Void
++ units(v,n,s) displays the units of the graph in field n of
++ the given two-dimensional viewport, v, which is of domain
++ \spadtype{TwoDimensionalViewport}, if s is "on", or does
++ not display the units if s is "off".
units      : ($,PI,PAL)                              -> Void
++ units(v,n,c) displays the units of the graph in field n of
++ the given two-dimensional viewport, v, which is of domain
++ \spadtype{TwoDimensionalViewport}, with the units color set to
++ the given palette color c.
points     : ($,PI,STR)                              -> Void
++ points(v,n,s) displays the points of the graph in field n of
++ the given two-dimensional viewport, v, which is of domain
++ \spadtype{TwoDimensionalViewport}, if s is "on", or does
++ not display the points if s is "off".
region     : ($,PI,STR)                              -> Void
++ region(v,n,s) displays the bounding box of the graph in
++ field n of the given two-dimensional viewport, v, which is
++ of domain \spadtype{TwoDimensionalViewport}, if s is "on",
++ or does not display the bounding box if s is "off".
connect    : ($,PI,STR)                              -> Void
++ connect(v,n,s) displays the lines connecting the graph

```



```

++ points in field n of the given two-dimensional viewport, v,
++ which is of domain \spadtype{TwoDimensionalViewport}, if s
++ is "on", or does not display the lines if s is "off".
controlPanel    : ($,STR)                                -> Void
++ controlPanel(v,s) displays the control panel of the given
++ two-dimensional viewport, v, which is of domain
++ \spadtype{TwoDimensionalViewport}, if s is "on", or hides
++ the control panel if s is "off".
close           : $                                       -> Void
++ close(v) closes the viewport window of the given
++ two-dimensional viewport, v, which is of domain
++ \spadtype{TwoDimensionalViewport}, and terminates the
++ corresponding process ID.
dimensions      : ($,NNI,NNI,PI,PI)                      -> Void
++ dimensions(v,x,y,width,height) sets the position of the
++ upper left-hand corner of the two-dimensional viewport, v,
++ which is of domain \spadtype{TwoDimensionalViewport}, to
++ the window coordinate x, y, and sets the dimensions of the
++ window to that of \spad{width}, \spad{height}. The new
++ dimensions are not displayed until the function
++ \spadfun{makeViewport2D} is executed again for v.
scale           : ($,PI,F,F)                             -> Void
++ scale(v,n,sx,sy) displays the graph in field n of the given
++ two-dimensional viewport, v, which is of domain
++ \spadtype{TwoDimensionalViewport}, scaled by the factor \spad{sx}
++ in the x-coordinate direction and by the factor \spad{sy} in
++ the y-coordinate direction.
translate       : ($,PI,F,F)                             -> Void
++ translate(v,n,dx,dy) displays the graph in field n of the given
++ two-dimensional viewport, v, which is of domain
++ \spadtype{TwoDimensionalViewport}, translated by \spad{dx} in
++ the x-coordinate direction from the center of the viewport, and
++ by \spad{dy} in the y-coordinate direction from the center.
++ Setting \spad{dx} and \spad{dy} to \spad{0} places the center
++ of the graph at the center of the viewport.
show           : ($,PI,STR)                              -> Void
++ show(v,n,s) displays the graph in field n of the given
++ two-dimensional viewport, v, which is of domain
++ \spadtype{TwoDimensionalViewport}, if s is "on", or does not
++ display the graph if s is "off".
move           : ($,NNI,NNI)                             -> Void
++ move(v,x,y) displays the two-dimensional viewport, v, which
++ is of domain \spadtype{TwoDimensionalViewport}, with the upper
++ left-hand corner of the viewport window at the screen
++ coordinate position x, y.
update         : ($,G,PI)                                -> Void
++ update(v,gr,n) drops the graph \spad{gr} in slot \spad{n}
++ of viewport \spad{v}. The graph gr must have been
++ transmitted already and acquired an integer key.
resize        : ($,PI,PI)                                -> Void

```

```

++ resize(v,w,h) displays the two-dimensional viewport, v, which
++ is of domain \spadtype{TwoDimensionalViewport}, with a width
++ of w and a height of h, keeping the upper left-hand corner
++ position unchanged.
write      : ($,STR)                                -> STR
++ write(v,s) takes the given two-dimensional viewport, v, which
++ is of domain \spadtype{TwoDimensionalViewport}, and creates
++ a directory indicated by s, which contains the graph data
++ files for v.
write      : ($,STR,STR)                            -> STR
++ write(v,s,f) takes the given two-dimensional viewport, v, which
++ is of domain \spadtype{TwoDimensionalViewport}, and creates
++ a directory indicated by s, which contains the graph data
++ files for v and an optional file type f.
write      : ($,STR,L STR)                          -> STR
++ write(v,s,lf) takes the given two-dimensional viewport, v, which
++ is of domain \spadtype{TwoDimensionalViewport}, and creates
++ a directory indicated by s, which contains the graph data
++ files for v and the optional file types indicated by the list lf.
reset      : $                                       -> Void
++ reset(v) sets the current state of the graph characteristics
++ of the given two-dimensional viewport, v, which is of domain
++ \spadtype{TwoDimensionalViewport}, back to their initial settings.
key        : $                                       -> I
++ key(v) returns the process ID number of the given two-dimensional
++ viewport, v, which is of domain \spadtype{TwoDimensionalViewport}.
coerce     : $                                       -> E
++ coerce(v) returns the given two-dimensional viewport, v, which
++ is of domain \spadtype{TwoDimensionalViewport} as output of
++ the domain \spadtype{OutputForm}.

```

Implementation ==> add

```

import GraphImage()
import Color()
import Palette()
import ViewDefaultsPackage()
import DrawOptionFunctions0
import POINT

Rep := Record (key:I, graphsField:V GU, graphStatesField:V GS, _
               title:STR, moveTo:XYNN, size:XYP, flags:FLAG, _
               optionsField:L DROP)

defaultGS : GS := [convert(0.9)@SF, convert(0.9)@SF, 0$SF, 0$SF, _
                  yes, yes, no, _
                  yes, axesColorDefault(), no, unitsColorDefault(), _
                  yes]

```

```

--% Local Functions
checkViewport (viewport:):B ==
  -- checks to see if this viewport still exists
  -- by sending the key to the viewport manager and
  -- waiting for its reply after it checks it against
  -- the viewports in its list. a -1 means it doesn't
  -- exist.
  sendI(VIEW,viewport.key)$Lisp
  i := getI(VIEW)$Lisp
  (i < 0$I) =>
    viewport.key := 0$I
    error "This viewport has already been closed!"
  true

doOptions(v:Rep):Void ==
  v.title := title(v.optionsField,"AXIOM2D")
  -- etc - 2D specific stuff...

--% Exported Functions

options viewport ==
  viewport.optionsField

options(viewport,opts) ==
  viewport.optionsField := opts
  viewport

putGraph (viewport,aGraph,which) ==
  if ((which > maxGRAPHS) or (which < 1)) then
    error "Trying to put a graph with a negative index or too big an index"
  viewport.graphsField.which := aGraph

getGraph (viewport,which) ==
  if ((which > maxGRAPHS) or (which < 1)) then
    error "Trying to get a graph with a negative index or too big an index"
  viewport.graphsField.which case "undefined" =>
    error "Graph is undefined!"
  viewport.graphsField.which::GraphImage

graphStates viewport == viewport.graphStatesField
graphs viewport == viewport.graphsField
key viewport == viewport.key

dimensions(viewport,ViewX,ViewY,ViewWidth,ViewHeight) ==
  viewport.moveTo := [ViewX,ViewY]
  viewport.size := [ViewWidth,ViewHeight]

move(viewport,xLoc,yLoc) ==
  viewport.moveTo := [xLoc,yLoc]

```

```

(key(viewport) ^= 0$I) =>
  sendI(VIEW,typeVIEW2D)$Lisp
  sendI(VIEW,MOVE)$Lisp
  checkViewport viewport =>
    sendI(VIEW,xLoc)$Lisp
    sendI(VIEW,yLoc)$Lisp
    getI(VIEW)$Lisp          -- acknowledge

update(viewport,graph,slot) ==
  (key(viewport) ^= 0$I) =>
    sendI(VIEW,typeVIEW2D)$Lisp
    sendI(VIEW,putGraph2D)$Lisp
    checkViewport viewport =>
      sendI(VIEW,key graph)$Lisp
      sendI(VIEW,slot)$Lisp
      getI(VIEW)$Lisp -- acknowledge

resize(viewport,xSize,ySize) ==
  viewport.size := [xSize,ySize]
  (key(viewport) ^= 0$I) =>
    sendI(VIEW,typeVIEW2D)$Lisp
    sendI(VIEW,RESIZE)$Lisp
    checkViewport viewport =>
      sendI(VIEW,xSize)$Lisp
      sendI(VIEW,ySize)$Lisp
      getI(VIEW)$Lisp          -- acknowledge

translate(viewport,graphIndex,xTranslateF,yTranslateF) ==
  xTranslate := convert(xTranslateF)@SF
  yTranslate := convert(yTranslateF)@SF
  if (graphIndex > maxGRAPHS) then
    error "Referring to a graph with too big an index"
  viewport.graphStatesField.graphIndex.deltaX := xTranslate
  viewport.graphStatesField.graphIndex.deltaY := yTranslate
  (key(viewport) ^= 0$I) =>
    sendI(VIEW,typeVIEW2D)$Lisp
    sendI(VIEW,TRANSLATE2D)$Lisp
    checkViewport viewport =>
      sendI(VIEW,graphIndex)$Lisp
      sendSF(VIEW,xTranslate)$Lisp
      sendSF(VIEW,yTranslate)$Lisp
      getI(VIEW)$Lisp          -- acknowledge

scale(viewport,graphIndex,xScaleF,yScaleF) ==
  xScale := convert(xScaleF)@SF
  yScale := convert(yScaleF)@SF
  if (graphIndex > maxGRAPHS) then
    error "Referring to a graph with too big an index"
  -- check union (undefined?)
  viewport.graphStatesField.graphIndex.scaleX := xScale

```

```

-- check union (undefined?)
viewport.graphStatesField.graphIndex.scaleY := yScale
(key(viewport) ^= 0$I) =>
  sendI(VIEW,typeVIEW2D)$Lisp
  sendI(VIEW,SCALE2D)$Lisp
  checkViewport viewport =>
    sendI(VIEW,graphIndex)$Lisp
    sendSF(VIEW,xScale)$Lisp
    sendSF(VIEW,yScale)$Lisp
    getI(VIEW)$Lisp          -- acknowledge

viewport2D ==
  [0,new(maxGRAPHS,"undefined"), _
  new(maxGRAPHS,copy defaultGS),"AXIOM2D", _
  [viewPosDefault().1,viewPosDefault().2], _
  [viewSizeDefault().1,viewSizeDefault().2], _
  [noControl], [] ]

makeViewport2D(g:G,opts:L DROP) ==
  viewport          := viewport2D()
  viewport.graphsField.1 := g
  viewport.optionsField := opts
  makeViewport2D viewport

makeViewport2D viewportDollar ==
  viewport := viewportDollar::Rep
--local function to extract and assign optional arguments for 2D viewports
doOptions viewport
sayBrightly(_
  ["  AXIOM2D data being transmitted to the viewport manager...":E]_
  $List(E))$Lisp
sendI(VIEW,typeVIEW2D)$Lisp
sendI(VIEW,makeVIEW2D)$Lisp
sendSTR(VIEW,viewport.title)$Lisp
sendI(VIEW,viewport.moveTo.X)$Lisp
sendI(VIEW,viewport.moveTo.Y)$Lisp
sendI(VIEW,viewport.size.X)$Lisp
sendI(VIEW,viewport.size.Y)$Lisp
sendI(VIEW,viewport.flags.showCP)$Lisp
for i in 1..maxGRAPHS repeat
  g := (graphs viewport).i
  if g case "undefined" then
    sendI(VIEW,0$I)$Lisp
  else
    sendI(VIEW,key(g::G))$Lisp
    gs := (graphStates viewport).i
    sendSF(VIEW,gs.scaleX)$Lisp
    sendSF(VIEW,gs.scaleY)$Lisp
    sendSF(VIEW,gs.deltaX)$Lisp
    sendSF(VIEW,gs.deltaY)$Lisp

```

```

    sendI(VIEW,gs.points)$Lisp
    sendI(VIEW,gs.connect)$Lisp
    sendI(VIEW,gs.spline)$Lisp
    sendI(VIEW,gs.axes)$Lisp
    hueShade:=hue hue gs.axesColor+shade gs.axesColor * numberOfHues()
    sendI(VIEW,hueShade)$Lisp
    sendI(VIEW,gs.units)$Lisp
    hueShade:=hue hue gs.unitsColor+shade gs.unitsColor * numberOfHues()
    sendI(VIEW,hueShade)$Lisp
    sendI(VIEW,gs.showing)$Lisp
viewport.key := getI(VIEW)$Lisp
viewport

graphState(viewport,num,sX,sY,dX,dY,Points,Lines,Spline, _
            Axes,AxesColor,Units,UnitsColor,Showing) ==
viewport.graphStatesField.num := [sX,sY,dX,dY,Points,Lines,Spline, _
                                   Axes,AxesColor,Units,UnitsColor,Showing]

title(viewport,Title) ==
viewport.title := Title
(key(viewport) ^= 0$I) =>
    sendI(VIEW,typeVIEW2D)$Lisp
    sendI(VIEW,TITLE)$Lisp
    checkViewport viewport =>
        sendSTR(VIEW,Title)$Lisp
        getI(VIEW)$Lisp      -- acknowledge

reset viewport ==
(key(viewport) ^= 0$I) =>
    sendI(VIEW,typeVIEW2D)$Lisp
    sendI(VIEW,SPADBUTTONPRESS)$Lisp
    checkViewport viewport =>
        sendI(VIEW,reset2D)$Lisp
        getI(VIEW)$Lisp      -- acknowledge

axes (viewport:$,graphIndex:PI,onOff:STR) : Void ==
    if (graphIndex > maxGRAPHS) then
        error "Referring to a graph with too big an index"
    if onOff = "on" then
        status := yes
    else
        status := no
    -- check union (undefined?)
viewport.graphStatesField.graphIndex.axes := status
(key(viewport) ^= 0$I) =>
    sendI(VIEW,typeVIEW2D)$Lisp
    sendI(VIEW,axesOnOff2D)$Lisp
    checkViewport viewport =>
        sendI(VIEW,graphIndex)$Lisp
        sendI(VIEW,status)$Lisp

```

```

getI(VIEW)$Lisp          -- acknowledge

axes (viewport:$,graphIndex:PI,color:PAL) : Void ==
  if (graphIndex > maxGRAPHS) then
    error "Referring to a graph with too big an index"
  viewport.graphStatesField.graphIndex.axesColor := color
  (key(viewport) ~= 0$I) =>
    sendI(VIEW,typeVIEW2D)$Lisp
    sendI(VIEW,axesColor2D)$Lisp
    checkViewport viewport =>
      sendI(VIEW,graphIndex)$Lisp
      hueShade := hue hue color + shade color * numberOfHues()
      sendI(VIEW,hueShade)$Lisp
      getI(VIEW)$Lisp      -- acknowledge

units (viewport:$,graphIndex:PI,onOff:STR) : Void ==
  if (graphIndex > maxGRAPHS) then
    error "Referring to a graph with too big an index"
  if onOff = "on" then
    status := yes
  else
    status := no
  -- check union (undefined?)
  viewport.graphStatesField.graphIndex.units := status
  (key(viewport) ~= 0$I) =>
    sendI(VIEW,typeVIEW2D)$Lisp
    sendI(VIEW,unitsOnOff2D)$Lisp
    checkViewport viewport =>
      sendI(VIEW,graphIndex)$Lisp
      sendI(VIEW,status)$Lisp
      getI(VIEW)$Lisp      -- acknowledge

units (viewport:$,graphIndex:PI,color:PAL) : Void ==
  if (graphIndex > maxGRAPHS) then
    error "Referring to a graph with too big an index"
  viewport.graphStatesField.graphIndex.unitsColor := color
  (key(viewport) ~= 0$I) =>
    sendI(VIEW,typeVIEW2D)$Lisp
    sendI(VIEW,unitsColor2D)$Lisp
    checkViewport viewport =>
      sendI(VIEW,graphIndex)$Lisp
      hueShade := hue hue color + shade color * numberOfHues()
      sendI(VIEW,hueShade)$Lisp
      getI(VIEW)$Lisp      -- acknowledge

connect (viewport:$,graphIndex:PI,onOff:STR) : Void ==
  if (graphIndex > maxGRAPHS) then
    error "Referring to a graph with too big an index"
  if onOff = "on" then
    status := 1$I

```

```

else
  status := 0$I
-- check union (undefined?)
viewport.graphStatesField.graphIndex.connect := status
(key(viewport) ^= 0$I) =>
  sendI(VIEW,typeVIEW2D)$Lisp
  sendI(VIEW,connectOnOff)$Lisp
  checkViewport viewport =>
    sendI(VIEW,graphIndex)$Lisp
    sendI(VIEW,status)$Lisp
    getI(VIEW)$Lisp      -- acknowledge

points (viewport:$,graphIndex:PI,onOff:STR) : Void ==
  if (graphIndex > maxGRAPHS) then
    error "Referring to a graph with too big an index"
  if onOff = "on" then
    status := 1$I
  else
    status := 0$I
-- check union (undefined?)
viewport.graphStatesField.graphIndex.points := status
(key(viewport) ^= 0$I) =>
  sendI(VIEW,typeVIEW2D)$Lisp
  sendI(VIEW,pointsOnOff)$Lisp
  checkViewport viewport =>
    sendI(VIEW,graphIndex)$Lisp
    sendI(VIEW,status)$Lisp
    getI(VIEW)$Lisp      -- acknowledge

region (viewport:$,graphIndex:PI,onOff:STR) : Void ==
  if (graphIndex > maxGRAPHS) then
    error "Referring to a graph with too big an index"
  if onOff = "on" then
    status := 1$I
  else
    status := 0$I
-- check union (undefined?)
viewport.graphStatesField.graphIndex.spline := status
(key(viewport) ^= 0$I) =>
  sendI(VIEW,typeVIEW2D)$Lisp
  sendI(VIEW,spline2D)$Lisp
  checkViewport viewport =>
    sendI(VIEW,graphIndex)$Lisp
    sendI(VIEW,status)$Lisp
    getI(VIEW)$Lisp      -- acknowledge

show (viewport,graphIndex,onOff) ==
  if (graphIndex > maxGRAPHS) then
    error "Referring to a graph with too big an index"
  if onOff = "on" then

```



```

        status := 1$I
    else
        status := 0$I
    -- check union (undefined?)
    viewport.graphStatesField.graphIndex.showing := status
    (key(viewport) ^= 0$I) =>
        sendI(VIEW,typeVIEW2D)$Lisp
        sendI(VIEW,showing2D)$Lisp
        checkViewport viewport =>
            sendI(VIEW,graphIndex)$Lisp
            sendI(VIEW,status)$Lisp
            getI(VIEW)$Lisp          -- acknowledge

controlPanel (viewport,onOff) ==
    if onOff = "on" then viewport.flags.showCP := yes
    else viewport.flags.showCP := no
    (key(viewport) ^= 0$I) =>
        sendI(VIEW,typeVIEW2D)$Lisp
        sendI(VIEW,hideControl2D)$Lisp
        checkViewport viewport =>
            sendI(VIEW,viewport.flags.showCP)$Lisp
            getI(VIEW)$Lisp          -- acknowledge

close viewport ==
    (key(viewport) ^= 0$I) =>
        sendI(VIEW,typeVIEW2D)$Lisp
        sendI(VIEW,closeAll2D)$Lisp
        checkViewport viewport =>
            getI(VIEW)$Lisp          -- acknowledge
        viewport.key := 0$I

coerce viewport ==
    (key(viewport) = 0$I) =>
        hconcat ["Closed or Undefined TwoDimensionalViewport: "::E,
            (viewport.title)::E]
        hconcat ["TwoDimensionalViewport: "::E, (viewport.title)::E]

write(viewport:$,Filename:STR,aThingToWrite:STR) ==
    write(viewport,Filename,[aThingToWrite])

write(viewport,Filename) ==
    write(viewport,Filename,viewWriteDefault())

write(viewport:$,Filename:STR,thingsToWrite:L STR) ==
    stringToSend : STR := ""
    (key(viewport) ^= 0$I) =>
        sendI(VIEW,typeVIEW2D)$Lisp
        sendI(VIEW,writeView)$Lisp
        checkViewport viewport =>
            sendSTR(VIEW,Filename)$Lisp

```

```

m := minIndex(avail := viewWriteAvailable())
for aTypeOfFile in thingsToWrite repeat
  if (writeTypeInt:=position(upperCase aTypeOfFile,avail)-m) < 0 then
    sayBrightly([" > "::E,(concat(aTypeOfFile, _
      " is not a valid file type for writing a 2D viewport"))::E]_
      $List(E))$Lisp
  else
    sendI(VIEW,writeTypeInt+(1$I))$Lisp
sendI(VIEW,0$I)$Lisp      -- no more types of things to write
getI(VIEW)$Lisp          -- acknowledge
Filename

```

— VIEW2D.dotabb —

```

"VIEW2D" [color="#88FF44",href="bookvol10.3.pdf#nameddest=VIEW2D"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"VIEW2D" -> "STRING"

```

Chapter 22

Chapter U

domain UFPS UnivariateFormalPowerSeries

— UnivariateFormalPowerSeries.input —

```
)set break resume
)sys rm -f UnivariateFormalPowerSeries.output
)spool UnivariateFormalPowerSeries.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariateFormalPowerSeries
--R
--R UnivariateFormalPowerSeries(Coef: Ring) is a domain constructor
--R Abbreviation for UnivariateFormalPowerSeries is UFPS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for UFPS
--R
--R----- Operations -----
--R ??? : (Coef,%) -> %           ??? : (%,Coef) -> %
--R ??? : (%,%) -> %             ??? : (Integer,%) -> %
--R ??? : (PositiveInteger,%) -> % ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %             ?-? : (%,%) -> %
--R -? : % -> %                   ?=? : (%,%) -> Boolean
--R 1 : () -> %                   0 : () -> %
--R ??? : (%,PositiveInteger) -> % center : % -> Coef
--R coefficients : % -> Stream(Coef) coerce : Variable(QUOTE(x)) -> %
--R coerce : Integer -> %         coerce : % -> OutputForm
--R complete : % -> %             degree : % -> NonNegativeInteger
--R evenlambert : % -> %          hash : % -> SingleInteger
```

```

--R lagrange : % -> %
--R latex : % -> String
--R leadingMonomial : % -> %
--R monomial? : % -> Boolean
--R one? : % -> Boolean
--R pole? : % -> Boolean
--R recip : % -> Union(%, "failed")
--R revert : % -> %
--R series : Stream(Coef) -> %
--R zero? : % -> Boolean
--R ?? : (%, Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (Fraction(Integer), %) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (NonNegativeInteger, %) -> %
--R ??? : (%, Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (%, %) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (%, Coef) -> % if Coef has FIELD
--R ??? : (%, NonNegativeInteger) -> %
--R ?/? : (%, Coef) -> % if Coef has FIELD
--R D : % -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef
--R D : (%, NonNegativeInteger) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef
--R D : (%, Symbol) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and Coef has PDRING(SYM
--R D : (%, List(Symbol)) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and Coef has PDRING(SYM
--R D : (%, Symbol, NonNegativeInteger) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and
--R D : (%, List(Symbol), List(NonNegativeInteger)) -> % if Coef has *: (NonNegativeInteger, Coef)
--R ?? : (%, NonNegativeInteger) -> %
--R acos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acoth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R approximate : (%, NonNegativeInteger) -> Coef if Coef has **: (Coef, NonNegativeInteger) -> Co
--R asec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R associates? : (%, %) -> Boolean if Coef has INTDOM
--R atan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R atanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if Coef has CHARNZ
--R coefficient : (%, NonNegativeInteger) -> Coef
--R coerce : UnivariatePolynomial(QUOTE(x), Coef) -> %
--R coerce : Coef -> % if Coef has COMRING
--R coerce : % -> % if Coef has INTDOM
--R coerce : Fraction(Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R cos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R coth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R lambert : % -> %
--R leadingCoefficient : % -> Coef
--R map : ((Coef -> Coef), %) -> %
--R oddlambert : % -> %
--R order : % -> NonNegativeInteger
--R quoByVar : % -> %
--R reductum : % -> %
--R sample : () -> %
--R variable : % -> Symbol
--R ~=? : (%, %) -> Boolean

```

```

--R csc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R differentiate : (% , Variable(QUOTE(x))) -> %
--R differentiate : % -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef
--R differentiate : (% , NonNegativeInteger) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef
--R differentiate : (% , Symbol) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and Coef has PDRING(SY
--R differentiate : (% , List(Symbol)) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and Coef has PDR
--R differentiate : (% , Symbol, NonNegativeInteger) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and
--R differentiate : (% , List(Symbol), List(NonNegativeInteger)) -> % if Coef has *: (NonNegativeInteger, Coef)
--R ?? : (% , %) -> % if NonNegativeInteger has SGROUP
--R ?? : (% , NonNegativeInteger) -> Coef
--R eval : (% , Coef) -> Stream(Coef) if Coef has **: (Coef, NonNegativeInteger) -> Coef
--R exp : % -> % if Coef has ALGEBRA(FRAC(INT))
--R exquo : (% , %) -> Union(%, "failed") if Coef has INTDOM
--R extend : (% , NonNegativeInteger) -> %
--R generalLambert : (% , Integer, Integer) -> %
--R integrate : (% , Variable(QUOTE(x))) -> % if Coef has ALGEBRA(FRAC(INT))
--R integrate : (% , Symbol) -> % if Coef has integrate: (Coef, Symbol) -> Coef and Coef has variables: Coef -
--R integrate : % -> % if Coef has ALGEBRA(FRAC(INT))
--R invmultisect : (Integer, Integer, %) -> %
--R log : % -> % if Coef has ALGEBRA(FRAC(INT))
--R monomial : (% , List(SingletonAsOrderedSet), List(NonNegativeInteger)) -> %
--R monomial : (% , SingletonAsOrderedSet, NonNegativeInteger) -> %
--R monomial : (Coef, NonNegativeInteger) -> %
--R multiplyCoefficients : ((Integer -> Coef), %) -> %
--R multiplyExponents : (% , PositiveInteger) -> %
--R multisect : (Integer, Integer, %) -> %
--R nthRoot : (% , Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R order : (% , NonNegativeInteger) -> NonNegativeInteger
--R pi : () -> % if Coef has ALGEBRA(FRAC(INT))
--R polynomial : (% , NonNegativeInteger, NonNegativeInteger) -> Polynomial(Coef)
--R polynomial : (% , NonNegativeInteger) -> Polynomial(Coef)
--R sec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R series : Stream(Record(k: NonNegativeInteger, c: Coef)) -> %
--R sin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sqrt : % -> % if Coef has ALGEBRA(FRAC(INT))
--R subtractIfCan : (% , %) -> Union(%, "failed")
--R tan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R tanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R terms : % -> Stream(Record(k: NonNegativeInteger, c: Coef))
--R truncate : (% , NonNegativeInteger, NonNegativeInteger) -> %
--R truncate : (% , NonNegativeInteger) -> %
--R unit? : % -> Boolean if Coef has INTDOM
--R unitCanonical : % -> % if Coef has INTDOM
--R unitNormal : % -> Record(unit: %, canonical: %, associate: %) if Coef has INTDOM
--R univariatePolynomial : (% , NonNegativeInteger) -> UnivariatePolynomial(QUOTE(x), Coef)
--R variables : % -> List(SingletonAsOrderedSet)
--R

```

```
--E 1  
  
)spool  
)lisp (bye)
```

— UnivariateFormalPowerSeries.help —

```
=====
```

```
UnivariateFormalPowerSeries examples
```

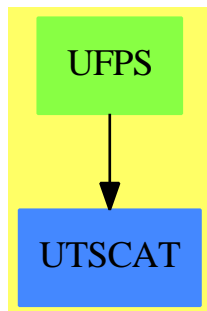
```
=====
```

This domain has no description

See Also:

o)show UnivariateFormalPowerSeries

UnivariateFormalPowerSeries (UFPS)



Exports:

0	1	acos	acosh
acot	acoth	acsc	acsch
approximate	asec	asech	asin
asinh	associates?	atan	atanh
center	characteristic	charthRoot	coefficient
coefficients	coerce	complete	cos
cosh	cot	coth	csc
csch	D	degree	differentiate
eval	evenlambert	exp	exquo
extend	generalLambert	hash	integrate
invmultisect	lagrange	lambert	latex
leadingCoefficient	leadingMonomial	log	map
monomial	monomial?	multiplyCoefficients	multiplyExponents
multisect	nthRoot	oddlambert	one?
order	pi	pole?	polynomial
quoByVar	recip	reductum	revert
sample	sec	sech	series
sin	sinh	sqrt	subtractIfCan
tan	tanh	terms	truncate
unit?	unitCanonical	unitNormal	univariatePolynomial
variable	variables	zero?	?*?
?**?	?+?	?-?	-?
?=?	?^?	?~=?	?/?
?..?			

— domain UFPS UnivariateFormalPowerSeries —

```
)abbrev domain UFPS UnivariateFormalPowerSeries
++ Author: Mark Botch
++ Description:
++ This domain has no description
```

```
UnivariateFormalPowerSeries(Coef: Ring) ==
  UnivariateTaylorSeries(Coef, 'x, 0$Coef)
```

— UFPS.dotabb —

```
"UFPS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=UFPS"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"UFPS" -> "UTSCAT"
```


domain ULS UnivariateLaurentSeries

— UnivariateLaurentSeries.input —

```

)set break resume
)sys rm -f UnivariateLaurentSeries.output
)spool UnivariateLaurentSeries.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariateLaurentSeries
--R
--R UnivariateLaurentSeries(Coef: Ring,var: Symbol,cen: Coef) is a domain constructor
--R Abbreviation for UnivariateLaurentSeries is ULS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ULS
--R
--R----- Operations -----
--R ?? : (Coef,%) -> %           ?? : (%,Coef) -> %
--R ?? : (%,%) -> %             ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %             ?-? : (%,%) -> %
--R -? : % -> %                  ?=? : (%,%) -> Boolean
--R 1 : () -> %                  0 : () -> %
--R ?? : (%,PositiveInteger) -> %   center : % -> Coef
--R coefficient : (%,Integer) -> Coef   coerce : Variable(var) -> %
--R coerce : Integer -> %              coerce : % -> OutputForm
--R complete : % -> %                 degree : % -> Integer
--R ?.? : (%,Integer) -> Coef          extend : (%,Integer) -> %
--R hash : % -> SingleInteger          latex : % -> String
--R leadingCoefficient : % -> Coef      leadingMonomial : % -> %
--R map : ((Coef -> Coef),%) -> %        monomial : (Coef,Integer) -> %
--R monomial? : % -> Boolean            one? : % -> Boolean
--R order : (%,Integer) -> Integer       order : % -> Integer
--R pole? : % -> Boolean                recip : % -> Union(%, "failed")
--R reductum : % -> %                  removeZeroes : (Integer,%) -> %
--R removeZeroes : % -> %              sample : () -> %
--R truncate : (%,Integer) -> %         variable : % -> Symbol
--R zero? : % -> Boolean                ?~=? : (%,%) -> Boolean
--R ?? : (%,Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (Fraction(Integer),%) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (UnivariateTaylorSeries(Coef,var,cen),%) -> % if Coef has FIELD
--R ?? : (%,UnivariateTaylorSeries(Coef,var,cen)) -> % if Coef has FIELD
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%,Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (%,%) -> % if Coef has ALGEBRA(FRAC(INT))

```

```

--R ***? : (%,Integer) -> % if Coef has FIELD
--R ***? : (%,NonNegativeInteger) -> %
--R ?/? : (UnivariateTaylorSeries(Coef,var,cen),UnivariateTaylorSeries(Coef,var,cen)) -> % if Coef has FIELD
--R ?/? : (%,% ) -> % if Coef has FIELD
--R ?/? : (% ,Coef) -> % if Coef has FIELD
--R ?<? : (%,% ) -> Boolean if UnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD or UnivariateTaylorSeries(Coef,var,cen) has OINTDOM
--R ?<=? : (%,% ) -> Boolean if UnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD or UnivariateTaylorSeries(Coef,var,cen) has OINTDOM
--R ?>? : (%,% ) -> Boolean if UnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD or UnivariateTaylorSeries(Coef,var,cen) has OINTDOM
--R ?>=? : (%,% ) -> Boolean if UnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD or UnivariateTaylorSeries(Coef,var,cen) has OINTDOM
--R D : (% ,Symbol) -> % if UnivariateTaylorSeries(Coef,var,cen) has PDRING(SYMBOL) and Coef has FIELD or UnivariateTaylorSeries(Coef,var,cen) has PDRING(SYMBOL)
--R D : (% ,List(Symbol)) -> % if UnivariateTaylorSeries(Coef,var,cen) has PDRING(SYMBOL) and Coef has FIELD or UnivariateTaylorSeries(Coef,var,cen) has PDRING(SYMBOL)
--R D : (% ,Symbol,NonNegativeInteger) -> % if UnivariateTaylorSeries(Coef,var,cen) has PDRING(SYMBOL) and Coef has FIELD or UnivariateTaylorSeries(Coef,var,cen) has PDRING(SYMBOL)
--R D : (% ,List(Symbol),List(NonNegativeInteger)) -> % if UnivariateTaylorSeries(Coef,var,cen) has PDRING(SYMBOL) and Coef has FIELD or UnivariateTaylorSeries(Coef,var,cen) has PDRING(SYMBOL)
--R D : % -> % if UnivariateTaylorSeries(Coef,var,cen) has DIFRING and Coef has FIELD or Coef has *: (Integer,Integer)
--R D : (% ,NonNegativeInteger) -> % if UnivariateTaylorSeries(Coef,var,cen) has DIFRING and Coef has FIELD or Coef has *: (Integer,Integer)
--R D : (% ,(UnivariateTaylorSeries(Coef,var,cen) -> UnivariateTaylorSeries(Coef,var,cen)),NonNegativeInteger) -> % if UnivariateTaylorSeries(Coef,var,cen) has DIFRING and Coef has FIELD or Coef has *: (Integer,Integer)
--R D : (% ,(UnivariateTaylorSeries(Coef,var,cen) -> UnivariateTaylorSeries(Coef,var,cen))) -> % if Coef has FIELD or Coef has *: (Integer,Integer)
--R ?? : (% ,Integer) -> % if Coef has FIELD
--R ?? : (% ,NonNegativeInteger) -> %
--R abs : % -> % if UnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD
--R acos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acoth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R approximate : (% ,Integer) -> Coef if Coef has **: (Coef,Integer) -> Coef and Coef has coerce: Symbol -> Coef
--R asec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R associates? : (%,% ) -> Boolean if UnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD or UnivariateTaylorSeries(Coef,var,cen) has OINTDOM
--R atan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R atanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R ceiling : % -> UnivariateTaylorSeries(Coef,var,cen) if UnivariateTaylorSeries(Coef,var,cen) has INS and Coef has FIELD
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%,"failed") if $ has CHARNZ and UnivariateTaylorSeries(Coef,var,cen) has PFECA
--R coerce : Fraction(Integer) -> % if UnivariateTaylorSeries(Coef,var,cen) has RETRACT(INT) and Coef has FIELD or UnivariateTaylorSeries(Coef,var,cen) has RETRACT(INT)
--R coerce : % -> % if UnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD or UnivariateTaylorSeries(Coef,var,cen) has OINTDOM
--R coerce : Symbol -> % if UnivariateTaylorSeries(Coef,var,cen) has RETRACT(SYMBOL) and Coef has FIELD or UnivariateTaylorSeries(Coef,var,cen) has RETRACT(SYMBOL)
--R coerce : UnivariateTaylorSeries(Coef,var,cen) -> %
--R coerce : Coef -> % if Coef has COMRING
--R conditionP : Matrix(%) -> Union(Vector(%),"failed") if $ has CHARNZ and UnivariateTaylorSeries(Coef,var,cen) has PFECA
--R convert : % -> Pattern(Integer) if UnivariateTaylorSeries(Coef,var,cen) has KONVERT(PATTERN(INT)) and Coef has FIELD or UnivariateTaylorSeries(Coef,var,cen) has KONVERT(PATTERN(INT))
--R convert : % -> Pattern(Float) if UnivariateTaylorSeries(Coef,var,cen) has KONVERT(PATTERN(FLOAT)) and Coef has FIELD or UnivariateTaylorSeries(Coef,var,cen) has KONVERT(PATTERN(FLOAT))
--R convert : % -> DoubleFloat if UnivariateTaylorSeries(Coef,var,cen) has REAL and Coef has FIELD or UnivariateTaylorSeries(Coef,var,cen) has REAL
--R convert : % -> Float if UnivariateTaylorSeries(Coef,var,cen) has REAL and Coef has FIELD or UnivariateTaylorSeries(Coef,var,cen) has REAL
--R convert : % -> InputForm if UnivariateTaylorSeries(Coef,var,cen) has KONVERT(INFORM) and Coef has FIELD or UnivariateTaylorSeries(Coef,var,cen) has KONVERT(INFORM)
--R cos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cosh : % -> % if Coef has ALGEBRA(FRAC(INT))

```

```

--R cot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R coth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R denom : % -> UnivariateTaylorSeries(Coef,var,cen) if Coef has FIELD
--R denominator : % -> % if Coef has FIELD
--R differentiate : (% ,Symbol) -> % if UnivariateTaylorSeries(Coef,var,cen) has PDRING(SYMBOL) a
--R differentiate : (% ,List(Symbol)) -> % if UnivariateTaylorSeries(Coef,var,cen) has PDRING(SYM
--R differentiate : (% ,Symbol,NonNegativeInteger) -> % if UnivariateTaylorSeries(Coef,var,cen) h
--R differentiate : (% ,List(Symbol),List(NonNegativeInteger)) -> % if UnivariateTaylorSeries(Coe
--R differentiate : % -> % if UnivariateTaylorSeries(Coef,var,cen) has DIFRING and Coef has FIEL
--R differentiate : (% ,NonNegativeInteger) -> % if UnivariateTaylorSeries(Coef,var,cen) has DIFR
--R differentiate : (% ,Variable(var)) -> %
--R differentiate : (% ,(UnivariateTaylorSeries(Coef,var,cen) -> UnivariateTaylorSeries(Coef,var,
--R differentiate : (% ,(UnivariateTaylorSeries(Coef,var,cen) -> UnivariateTaylorSeries(Coef,var,
--R divide : (% ,%) -> Record(quotient: %,remainder: %) if Coef has FIELD
--R ?.? : (% ,UnivariateTaylorSeries(Coef,var,cen)) -> % if UnivariateTaylorSeries(Coef,var,cen)
--R ?.? : (% ,%) -> % if Integer has SGROUP
--R euclideanSize : % -> NonNegativeInteger if Coef has FIELD
--R eval : (% ,List(UnivariateTaylorSeries(Coef,var,cen)),List(UnivariateTaylorSeries(Coef,var,ce
--R eval : (% ,UnivariateTaylorSeries(Coef,var,cen),UnivariateTaylorSeries(Coef,var,cen)) -> % if
--R eval : (% ,Equation(UnivariateTaylorSeries(Coef,var,cen))) -> % if UnivariateTaylorSeries(Coe
--R eval : (% ,List(Equation(UnivariateTaylorSeries(Coef,var,cen)))) -> % if UnivariateTaylorSeri
--R eval : (% ,List(Symbol),List(UnivariateTaylorSeries(Coef,var,cen))) -> % if UnivariateTaylorS
--R eval : (% ,Symbol,UnivariateTaylorSeries(Coef,var,cen)) -> % if UnivariateTaylorSeries(Coef,v
--R eval : (% ,Coef) -> Stream(Coef) if Coef has **: (Coef,Integer) -> Coef
--R exp : % -> % if Coef has ALGEBRA(FRAC(INT))
--R expressIdealMember : (List(%),%) -> Union(List(%),"failed") if Coef has FIELD
--R exquo : (% ,%) -> Union(%,"failed") if UnivariateTaylorSeries(Coef,var,cen) has OINTDOM and C
--R extendedEuclidean : (% ,%) -> Record(coef1: %,coef2: %,generator: %) if Coef has FIELD
--R extendedEuclidean : (% ,%,%) -> Union(Record(coef1: %,coef2: %),"failed") if Coef has FIELD
--R factor : % -> Factored(%) if Coef has FIELD
--R factorPolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%))
--R factorSquareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolyn
--R floor : % -> UnivariateTaylorSeries(Coef,var,cen) if UnivariateTaylorSeries(Coef,var,cen) ha
--R fractionPart : % -> % if UnivariateTaylorSeries(Coef,var,cen) has EUCDOM and Coef has FIELD
--R gcd : (% ,%) -> % if Coef has FIELD
--R gcd : List(%) -> % if Coef has FIELD
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUniva
--R init : () -> % if UnivariateTaylorSeries(Coef,var,cen) has STEP and Coef has FIELD
--R integrate : (% ,Variable(var)) -> % if Coef has ALGEBRA(FRAC(INT))
--R integrate : (% ,Symbol) -> % if Coef has integrate: (Coef,Symbol) -> Coef and Coef has variab
--R integrate : % -> % if Coef has ALGEBRA(FRAC(INT))
--R inv : % -> % if Coef has FIELD
--R laurent : (Integer,UnivariateTaylorSeries(Coef,var,cen)) -> %
--R lcm : (% ,%) -> % if Coef has FIELD
--R lcm : List(%) -> % if Coef has FIELD
--R log : % -> % if Coef has ALGEBRA(FRAC(INT))
--R map : ((UnivariateTaylorSeries(Coef,var,cen) -> UnivariateTaylorSeries(Coef,var,cen)),%) ->
--R max : (% ,%) -> % if UnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD or U

```

```

--R min : (% , %) -> % if UnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD or UnivariateTa
--R monomial : (% , List(SingletonAsOrderedSet), List(Integer)) -> %
--R monomial : (% , SingletonAsOrderedSet, Integer) -> %
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed") if Coef has FIELD
--R multiplyCoefficients : ((Integer -> Coef), %) -> %
--R multiplyExponents : (% , PositiveInteger) -> %
--R negative? : % -> Boolean if UnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD
--R nextItem : % -> Union(% , "failed") if UnivariateTaylorSeries(Coef,var,cen) has STEP and Coef has FIELD
--R nthRoot : (% , Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R numer : % -> UnivariateTaylorSeries(Coef,var,cen) if Coef has FIELD
--R numerator : % -> % if Coef has FIELD
--R patternMatch : (% , Pattern(Float), PatternMatchResult(Float, %)) -> PatternMatchResult(Float, %) if Univari
--R patternMatch : (% , Pattern(Integer), PatternMatchResult(Integer, %)) -> PatternMatchResult(Integer, %) if U
--R pi : () -> % if Coef has ALGEBRA(FRAC(INT))
--R positive? : % -> Boolean if UnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD
--R prime? : % -> Boolean if Coef has FIELD
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %) if Coef has FIELD
--R ?quo? : (% , %) -> % if Coef has FIELD
--R random : () -> % if UnivariateTaylorSeries(Coef,var,cen) has INS and Coef has FIELD
--R rationalFunction : (% , Integer, Integer) -> Fraction(Polynomial(Coef)) if Coef has INTDOM
--R rationalFunction : (% , Integer) -> Fraction(Polynomial(Coef)) if Coef has INTDOM
--R reducedSystem : Matrix(%) -> Matrix(Integer) if UnivariateTaylorSeries(Coef,var,cen) has LINEXP(INT) and
--R reducedSystem : (Matrix(%), Vector(%)) -> Record(mat: Matrix(Integer), vec: Vector(Integer)) if Univari
--R reducedSystem : (Matrix(%), Vector(%)) -> Record(mat: Matrix(UnivariateTaylorSeries(Coef,var,cen)), vec: V
--R reducedSystem : Matrix(%) -> Matrix(UnivariateTaylorSeries(Coef,var,cen)) if Coef has FIELD
--R ?rem? : (% , %) -> % if Coef has FIELD
--R retract : % -> Integer if UnivariateTaylorSeries(Coef,var,cen) has RETRACT(INT) and Coef has FIELD
--R retract : % -> Fraction(Integer) if UnivariateTaylorSeries(Coef,var,cen) has RETRACT(INT) and Coef has I
--R retract : % -> Symbol if UnivariateTaylorSeries(Coef,var,cen) has RETRACT(SYMBOL) and Coef has FIELD
--R retract : % -> UnivariateTaylorSeries(Coef,var,cen)
--R retractIfCan : % -> Union(Integer, "failed") if UnivariateTaylorSeries(Coef,var,cen) has RETRACT(INT) and
--R retractIfCan : % -> Union(Fraction(Integer), "failed") if UnivariateTaylorSeries(Coef,var,cen) has RETRA
--R retractIfCan : % -> Union(Symbol, "failed") if UnivariateTaylorSeries(Coef,var,cen) has RETRACT(SYMBOL) a
--R retractIfCan : % -> Union(UnivariateTaylorSeries(Coef,var,cen), "failed")
--R sec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R series : Stream(Record(k: Integer, c: Coef)) -> %
--R sign : % -> Integer if UnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD
--R sin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sizeLess? : (% , %) -> Boolean if Coef has FIELD
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(%)), SparseUnivariatePolynomial(%)) -> U
--R sqrt : % -> % if Coef has ALGEBRA(FRAC(INT))
--R squareFree : % -> Factored(%) if Coef has FIELD
--R squareFreePart : % -> % if Coef has FIELD
--R squareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%)) if Univ
--R subtractIfCan : (% , %) -> Union(% , "failed")
--R tan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R tanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R taylor : % -> UnivariateTaylorSeries(Coef,var,cen)

```

```

--R taylorIfCan : % -> Union(UnivariateTaylorSeries(Coef,var,cen),"failed")
--R taylorRep : % -> UnivariateTaylorSeries(Coef,var,cen)
--R terms : % -> Stream(Record(k: Integer,c: Coef))
--R truncate : (%,Integer,Integer) -> %
--R unit? : % -> Boolean if UnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD
--R unitCanonical : % -> % if UnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %) if UnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD
--R variables : % -> List(SingletonAsOrderedSet)
--R wholePart : % -> UnivariateTaylorSeries(Coef,var,cen) if UnivariateTaylorSeries(Coef,var,cen) has OINTDOM and Coef has FIELD
--R
--E 1

)spool
)lisp (bye)

```

— UnivariateLaurentSeries.help —

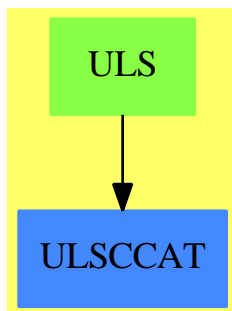
```

=====
UnivariateLaurentSeries examples
=====

```

UnivariateLaurentSeries is a domain representing Laurent series in one variable with coefficients in an arbitrary ring. The parameters of the type specify the coefficient ring, the power series variable, and the center of the power series expansion. For example, `UnivariateLaurentSeries(Integer,x,3)` represents Laurent series in $(x - 3)$ with integer coefficients.

See Also:
o `)show UnivariateLaurentSeries`

UnivariateLaurentSeries (ULS)

See

⇒ “UnivariateLaurentSeriesConstructor” (ULSCONS) [22](#) on page [2803](#)

Exports:

0	1	abs
acos	acosh	acot
acoth	acsc	acsch
approximate	asec	asech
asin	asinh	associates?
atan	atanh	ceiling
center	characteristic	charthRoot
coerce	coefficient	coerce
complete	conditionP	convert
cos	cosh	cot
coth	csc	csch
D	degree	denom
denominator	differentiate	divide
euclideanSize	eval	exp
expressIdealMember	exquo	extend
extendedEuclidean	factor	factorPolynomial
factorSquareFreePolynomial	floor	fractionPart
gcd	gcdPolynomial	hash
init	integrate	inv
latex	laurent	leadingCoefficient
leadingMonomial	lcm	log
map	max	min
monomial	monomial?	multiEuclidean
multiplyCoefficients	multiplyExponents	negative?
nextItem	nthRoot	numer
numerator	one?	order
patternMatch	pi	pole?
positive?	prime?	principalIdeal
random	rationalFunction	recip
reducedSystem	reductum	removeZeroes
retract	retractIfCan	sample
sec	sech	series
sign	sin	sinh
sizeLess?	solveLinearPolynomialEquation	sqrt
squareFree	squareFreePart	squareFreePolynomial
subtractIfCan	tan	tanh
taylor	taylorIfCan	taylorRep
terms	truncate	unit?
unitCanonical	unitNormal	variable
variables	wholePart	zero?
?*?	?**?	?+?
?-?	-?	?=?
?^?	?.	?~=?
?/?	?<?	?<=?
?>?	?>=?	?^?
?..?	?quo?	?rem?

— domain ULS UnivariateLaurentSeries —

```

)abbrev domain ULS UnivariateLaurentSeries
++ Author: Clifton J. Williamson
++ Date Created: 18 January 1990
++ Date Last Updated: 21 September 1993
++ Description:
++ Dense Laurent series in one variable
++ \spadtype{UnivariateLaurentSeries} is a domain representing Laurent
++ series in one variable with coefficients in an arbitrary ring. The
++ parameters of the type specify the coefficient ring, the power series
++ variable, and the center of the power series expansion. For example,
++ \spad{UnivariateLaurentSeries(Integer,x,3)} represents Laurent series in
++ \spad{(x - 3)} with integer coefficients.

UnivariateLaurentSeries(Coef,var,cen): Exports == Implementation where
  Coef : Ring
  var  : Symbol
  cen  : Coef
  I    ==> Integer
  UTS  ==> UnivariateTaylorSeries(Coef,var,cen)

Exports ==> UnivariateLaurentSeriesConstructorCategory(Coef,UTS) with
  coerce: Variable(var) -> %
    ++ \spad{coerce(var)} converts the series variable \spad{var} into a
    ++ Laurent series.
  differentiate: (% ,Variable(var)) -> %
    ++ \spad{differentiate(f(x),x)} returns the derivative of
    ++ \spad{f(x)} with respect to \spad{x}.
  if Coef has Algebra Fraction Integer then
    integrate: (% ,Variable(var)) -> %
      ++ \spad{integrate(f(x))} returns an anti-derivative of the power
      ++ series \spad{f(x)} with constant coefficient 0.
      ++ We may integrate a series when we can divide coefficients
      ++ by integers.

Implementation ==> UnivariateLaurentSeriesConstructor(Coef,UTS) add

  variable x == var
  center   x == cen

  coerce(v:Variable(var)) ==
    zero? cen => monomial(1,1)
    monomial(1,1) + monomial(cen,0)

  differentiate(x:% ,v:Variable(var)) == differentiate x

  if Coef has Algebra Fraction Integer then
    integrate(x:% ,v:Variable(var)) == integrate x

```

— ULS.dotabb —

```
"ULS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ULS"]
"ULSCCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ULSCCAT"]
"ULS" -> "ULSCCAT"
```

domain ULSCONS UnivariateLaurentSeriesConstructor

— UnivariateLaurentSeriesConstructor.input —

```
)set break resume
)sys rm -f UnivariateLaurentSeriesConstructor.output
)spool UnivariateLaurentSeriesConstructor.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariateLaurentSeriesConstructor
--R
--R UnivariateLaurentSeriesConstructor(Coef: Ring,UTS: UnivariateTaylorSeriesCategory(Coef)) is
--R Abbreviation for UnivariateLaurentSeriesConstructor is ULSCONS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for ULSCONS
--R
--R----- Operations -----
--R ?? : (Coef,%) -> %           ?? : (%,Coef) -> %
--R ?? : (%,%) -> %             ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %             ?-? : (%,%) -> %
--R -? : % -> %                  ?? : (%,%) -> Boolean
--R 1 : () -> %                  0 : () -> %
--R ?^? : (%,PositiveInteger) -> %   center : % -> Coef
--R coefficient : (%,Integer) -> Coef   coerce : UTS -> %
--R coerce : Integer -> %               coerce : % -> OutputForm
--R complete : % -> %                 degree : % -> Integer
--R ?.? : (%,Integer) -> Coef         extend : (%,Integer) -> %
--R hash : % -> SingleInteger         latex : % -> String
--R laurent : (Integer,UTS) -> %      leadingCoefficient : % -> Coef
```

```

--R leadingMonomial : % -> %
--R monomial : (Coef,Integer) -> %
--R one? : % -> Boolean
--R order : % -> Integer
--R recip : % -> Union(%, "failed")
--R removeZeroes : (Integer,%) -> %
--R retract : % -> UTS
--R taylor : % -> UTS
--R truncate : (%,Integer) -> %
--R zero? : % -> Boolean
--R ?? : (%,Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (Fraction(Integer),%) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (UTS,%) -> % if Coef has FIELD
--R ?? : (%,UTS) -> % if Coef has FIELD
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%,Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (%,%) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (%,Integer) -> % if Coef has FIELD
--R ??? : (%,NonNegativeInteger) -> %
--R ?/? : (UTS,UTS) -> % if Coef has FIELD
--R ?/? : (%,%) -> % if Coef has FIELD
--R ?/? : (%,Coef) -> % if Coef has FIELD
--R ?<? : (%,%) -> Boolean if UTS has ORDSET and Coef has FIELD
--R ?<=? : (%,%) -> Boolean if UTS has ORDSET and Coef has FIELD
--R ?>? : (%,%) -> Boolean if UTS has ORDSET and Coef has FIELD
--R ?>=? : (%,%) -> Boolean if UTS has ORDSET and Coef has FIELD
--R D : (%,Symbol) -> % if UTS has PDRING(SYMBOL) and Coef has FIELD or Coef has *: (Integer,Coef) -> Coef
--R D : (%,List(Symbol)) -> % if UTS has PDRING(SYMBOL) and Coef has FIELD or Coef has *: (Integer,Coef) ->
--R D : (%,Symbol,NonNegativeInteger) -> % if UTS has PDRING(SYMBOL) and Coef has FIELD or Coef has *: (Int
--R D : (%,List(Symbol),List(NonNegativeInteger)) -> % if UTS has PDRING(SYMBOL) and Coef has FIELD or Coef
--R D : % -> % if UTS has DIFRING and Coef has FIELD or Coef has *: (Integer,Coef) -> Coef
--R D : (%,NonNegativeInteger) -> % if UTS has DIFRING and Coef has FIELD or Coef has *: (Integer,Coef) ->
--R D : (%,(UTS -> UTS),NonNegativeInteger) -> % if Coef has FIELD
--R D : (%,(UTS -> UTS)) -> % if Coef has FIELD
--R ?? : (%,Integer) -> % if Coef has FIELD
--R ?? : (%,NonNegativeInteger) -> %
--R abs : % -> % if UTS has OINTDOM and Coef has FIELD
--R acos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acoth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R approximate : (%,Integer) -> Coef if Coef has **: (Coef,Integer) -> Coef and Coef has coerce: Symbol ->
--R asec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R associates? : (%,%) -> Boolean if Coef has INTDOM
--R atan : % -> % if Coef has ALGEBRA(FRAC(INT))

```

```

--R atanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R ceiling : % -> UTS if UTS has INS and Coef has FIELD
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if $ has CHARNZ and UTS has PFECAT and Coef has FIELD or
--R coerce : % -> % if Coef has INTDOM
--R coerce : Fraction(Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R coerce : Symbol -> % if UTS has RETRACT(SYMBOL) and Coef has FIELD
--R coerce : Coef -> % if Coef has COMRING
--R conditionP : Matrix(%) -> Union(Vector(%), "failed") if $ has CHARNZ and UTS has PFECAT and C
--R convert : % -> Pattern(Integer) if UTS has KONVERT(PATTERN(INT)) and Coef has FIELD
--R convert : % -> Pattern(Float) if UTS has KONVERT(PATTERN(FLOAT)) and Coef has FIELD
--R convert : % -> InputForm if UTS has KONVERT(INFORM) and Coef has FIELD
--R convert : % -> Float if UTS has REAL and Coef has FIELD
--R convert : % -> DoubleFloat if UTS has REAL and Coef has FIELD
--R cos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R coth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R denom : % -> UTS if Coef has FIELD
--R denominator : % -> % if Coef has FIELD
--R differentiate : (%, Symbol) -> % if UTS has PDRING(SYMBOL) and Coef has FIELD or Coef has *:
--R differentiate : (%, List(Symbol)) -> % if UTS has PDRING(SYMBOL) and Coef has FIELD or Coef h
--R differentiate : (%, Symbol, NonNegativeInteger) -> % if UTS has PDRING(SYMBOL) and Coef has FI
--R differentiate : (%, List(Symbol), List(NonNegativeInteger)) -> % if UTS has PDRING(SYMBOL) and
--R differentiate : % -> % if UTS has DIFRING and Coef has FIELD or Coef has *: (Integer, Coef) -
--R differentiate : (%, NonNegativeInteger) -> % if UTS has DIFRING and Coef has FIELD or Coef ha
--R differentiate : (%, (UTS -> UTS), NonNegativeInteger) -> % if Coef has FIELD
--R differentiate : (%, (UTS -> UTS)) -> % if Coef has FIELD
--R divide : (%, %) -> Record(quotient: %, remainder: %) if Coef has FIELD
--R ?.? : (%, UTS) -> % if UTS has ELTAB(UTS, UTS) and Coef has FIELD
--R ?.? : (%, %) -> % if Integer has SGROUP
--R euclideanSize : % -> NonNegativeInteger if Coef has FIELD
--R eval : (%, List(UTS), List(UTS)) -> % if UTS has EVALAB(UTS) and Coef has FIELD
--R eval : (%, UTS, UTS) -> % if UTS has EVALAB(UTS) and Coef has FIELD
--R eval : (%, Equation(UTS)) -> % if UTS has EVALAB(UTS) and Coef has FIELD
--R eval : (%, List(Equation(UTS))) -> % if UTS has EVALAB(UTS) and Coef has FIELD
--R eval : (%, List(Symbol), List(UTS)) -> % if UTS has IEVALAB(SYMBOL, UTS) and Coef has FIELD
--R eval : (%, Symbol, UTS) -> % if UTS has IEVALAB(SYMBOL, UTS) and Coef has FIELD
--R eval : (%, Coef) -> Stream(Coef) if Coef has **: (Coef, Integer) -> Coef
--R exp : % -> % if Coef has ALGEBRA(FRAC(INT))
--R expressIdealMember : (List(%), %) -> Union(List(%), "failed") if Coef has FIELD
--R exquo : (%, %) -> Union(%, "failed") if Coef has INTDOM
--R extendedEuclidean : (%, %) -> Record(coef1: %, coef2: %, generator: %) if Coef has FIELD
--R extendedEuclidean : (%, %, %) -> Union(Record(coef1: %, coef2: %), "failed") if Coef has FIELD
--R factor : % -> Factored(%) if Coef has FIELD
--R factorPolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%))
--R factorSquareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolyn
--R floor : % -> UTS if UTS has INS and Coef has FIELD

```

```

--R fractionPart : % -> % if UTS has EUCDOM and Coef has FIELD
--R gcd : (% , %) -> % if Coef has FIELD
--R gcd : List(%) -> % if Coef has FIELD
--R gcdPolynomial : (SparseUnivariatePolynomial(%), SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
--R init : () -> % if UTS has STEP and Coef has FIELD
--R integrate : (% , Symbol) -> % if Coef has integrate: (Coef, Symbol) -> Coef and Coef has variables: Coef -> %
--R integrate : % -> % if Coef has ALGEBRA(FRAC(INT))
--R inv : % -> % if Coef has FIELD
--R lcm : (% , %) -> % if Coef has FIELD
--R lcm : List(%) -> % if Coef has FIELD
--R log : % -> % if Coef has ALGEBRA(FRAC(INT))
--R map : ((UTS -> UTS), %) -> % if Coef has FIELD
--R max : (% , %) -> % if UTS has ORDSET and Coef has FIELD
--R min : (% , %) -> % if UTS has ORDSET and Coef has FIELD
--R monomial : (% , List(SingletonAsOrderedSet), List(Integer)) -> %
--R monomial : (% , SingletonAsOrderedSet, Integer) -> %
--R multiEuclidean : (List(%), %) -> Union(List(%), "failed") if Coef has FIELD
--R multiplyCoefficients : ((Integer -> Coef), %) -> %
--R multiplyExponents : (% , PositiveInteger) -> %
--R negative? : % -> Boolean if UTS has OINTDOM and Coef has FIELD
--R nextItem : % -> Union(% , "failed") if UTS has STEP and Coef has FIELD
--R nthRoot : (% , Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R numer : % -> UTS if Coef has FIELD
--R numerator : % -> % if Coef has FIELD
--R patternMatch : (% , Pattern(Integer), PatternMatchResult(Integer, %)) -> PatternMatchResult(Integer, %) if UTS has FIELD
--R patternMatch : (% , Pattern(Float), PatternMatchResult(Float, %)) -> PatternMatchResult(Float, %) if UTS has FIELD
--R pi : () -> % if Coef has ALGEBRA(FRAC(INT))
--R positive? : % -> Boolean if UTS has OINTDOM and Coef has FIELD
--R prime? : % -> Boolean if Coef has FIELD
--R principalIdeal : List(%) -> Record(coef: List(%), generator: %) if Coef has FIELD
--R ?quo? : (% , %) -> % if Coef has FIELD
--R random : () -> % if UTS has INS and Coef has FIELD
--R rationalFunction : (% , Integer, Integer) -> Fraction(Polynomial(Coef)) if Coef has INTDOM
--R rationalFunction : (% , Integer) -> Fraction(Polynomial(Coef)) if Coef has INTDOM
--R reducedSystem : Matrix(%) -> Matrix(Integer) if UTS has LINEXP(INT) and Coef has FIELD
--R reducedSystem : (Matrix(%), Vector(%)) -> Record(mat: Matrix(Integer), vec: Vector(Integer)) if UTS has LINEXP(INT)
--R reducedSystem : (Matrix(%), Vector(%)) -> Record(mat: Matrix(UTS), vec: Vector(UTS)) if Coef has FIELD
--R reducedSystem : Matrix(%) -> Matrix(UTS) if Coef has FIELD
--R ?rem? : (% , %) -> % if Coef has FIELD
--R retract : % -> Symbol if UTS has RETRACT(SYMBOL) and Coef has FIELD
--R retract : % -> Fraction(Integer) if UTS has RETRACT(INT) and Coef has FIELD
--R retract : % -> Integer if UTS has RETRACT(INT) and Coef has FIELD
--R retractIfCan : % -> Union(Symbol, "failed") if UTS has RETRACT(SYMBOL) and Coef has FIELD
--R retractIfCan : % -> Union(Fraction(Integer), "failed") if UTS has RETRACT(INT) and Coef has FIELD
--R retractIfCan : % -> Union(Integer, "failed") if UTS has RETRACT(INT) and Coef has FIELD
--R retractIfCan : % -> Union(UTS, "failed")
--R sec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R series : Stream(Record(k: Integer, c: Coef)) -> %
--R sign : % -> Integer if UTS has OINTDOM and Coef has FIELD

```

```

--R sin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sizeLess? : (%,%) -> Boolean if Coef has FIELD
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(%)),SparseUnivariatePolynomial(%)) -> List(SparseUnivariatePolynomial(%))
--R sqrt : % -> % if Coef has ALGEBRA(FRAC(INT))
--R squareFree : % -> Factored(%) if Coef has FIELD
--R squareFreePart : % -> % if Coef has FIELD
--R squareFreePolynomial : SparseUnivariatePolynomial(%) -> Factored(SparseUnivariatePolynomial(%))
--R subtractIfCan : (%,%) -> Union(%, "failed")
--R tan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R tanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R taylorIfCan : % -> Union(UTS, "failed")
--R terms : % -> Stream(Record(k: Integer, c: Coef))
--R truncate : (%, Integer, Integer) -> %
--R unit? : % -> Boolean if Coef has INTDOM
--R unitCanonical : % -> % if Coef has INTDOM
--R unitNormal : % -> Record(unit: %, canonical: %, associate: %) if Coef has INTDOM
--R variables : % -> List(SingletonAsOrderedSet)
--R wholePart : % -> UTS if UTS has EUCDOM and Coef has FIELD
--R
--E 1

```

```

)spool
)lisp (bye)

```

— UnivariateLaurentSeriesConstructor.help —

```

=====
UnivariateLaurentSeriesConstructor examples
=====

```

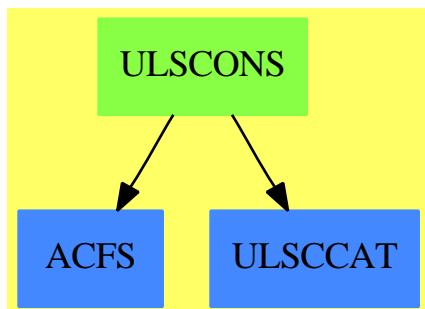
This package enables one to construct a univariate Laurent series domain from a univariate Taylor series domain. Univariate Laurent series are represented by a pair $[n, f(x)]$, where n is an arbitrary integer and $f(x)$ is a Taylor series. This pair represents the Laurent series $x^n * f(x)$.

See Also:

```

o )show UnivariateLaurentSeriesConstructor

```

UnivariateLaurentSeriesConstructor (ULSCONS)

See

⇒ “UnivariateLaurentSeries” (ULS) [22](#) on page [2795](#)

Exports:

0	1	abs
acos	acosh	acot
acoth	acsc	acsch
approximate	asec	asech
asin	asinh	associates?
atan	atanh	ceiling
center	characteristic	charthRoot
coefficient	coerce	complete
conditionP	convert	cos
cosh	cot	coth
csc	csch	D
degree	denom	denominator
differentiate	divide	extend
euclideanSize	eval	exp
expressIdealMember	exquo	extendedEuclidean
factor	factorPolynomial	factorSquareFreePolynomial
floor	fractionPart	gcd
gcdPolynomial	hash	init
integrate	inv	latex
laurent	lcm	leadingCoefficient
leadingMonomial	log	map
max	min	monomial
monomial?	multiEuclidean	multiplyCoefficients
multiplyExponents	negative?	nextItem
nthRoot	numer	numerator
one?	order	patternMatch
pi	pole?	positive?
prime?	principalIdeal	random
rationalFunction	recip	reducedSystem
reductum	removeZeroes	retract
retractIfCan	sample	sec
sech	series	sign
sin	sinh	sizeLess?
solveLinearPolynomialEquation	sqrt	squareFree
squareFreePart	squareFreePolynomial	subtractIfCan
tan	tanh	taylor
taylorIfCan	taylorRep	terms
truncate	unit?	unitCanonical
unitNormal	variable	variables
wholePart	zero?	?*?
?**?	?+?	?-?
-?	?=?	?^?
?..?	?~=?	?/?
?<?	?<=?	?>?
?>=?	?quo?	?rem?

— domain ULSCONS UnivariateLaurentSeriesConstructor —

```

)abbrev domain ULSCONS UnivariateLaurentSeriesConstructor
++ Authors: Bill Burge, Clifton J. Williamson
++ Date Created: August 1988
++ Date Last Updated: 17 June 1996
++ Description:
++ This package enables one to construct a univariate Laurent series
++ domain from a univariate Taylor series domain. Univariate
++ Laurent series are represented by a pair \spad{[n,f(x)]}, where n is
++ an arbitrary integer and \spad{f(x)} is a Taylor series. This pair
++ represents the Laurent series \spad{x**n * f(x)}.

```

```

UnivariateLaurentSeriesConstructor(Coef,UTS):_
Exports == Implementation where
  Coef      : Ring
  UTS       : UnivariateTaylorSeriesCategory Coef
  I         ==> Integer
  L         ==> List
  NNI       ==> NonNegativeInteger
  OUT       ==> OutputForm
  P         ==> Polynomial Coef
  RF        ==> Fraction Polynomial Coef
  RN        ==> Fraction Integer
  ST        ==> Stream Coef
  TERM      ==> Record(k:I,c:Coef)
  monom     ==> monomial$UTS
  EFULS     ==> ElementaryFunctionsUnivariateLaurentSeries(Coef,UTS,%)
  STTAYLOR  ==> StreamTaylorSeriesOperations Coef

```

```

Exports ==> UnivariateLaurentSeriesConstructorCategory(Coef,UTS)

```

```

Implementation ==> add

```

```

--% representation

```

```

  Rep := Record(expon:I,ps:UTS)

```

```

  getExpon : % -> I
  getUTS   : % -> UTS

```

```

  getExpon x == x.expon
  getUTS    x == x.ps

```

```

--% creation and destruction

```

```

  laurent(n,psr) == [n,psr]
  taylorRep x    == getUTS x
  degree x       == getExpon x

```



```

0 == laurent(0,0)
1 == laurent(0,1)

monomial(s,e) == laurent(e,s::UTS)

coerce(uts:UTS):% == laurent(0,uts)
coerce(r:Coef):% == r :: UTS :: %
coerce(i:I):% == i :: Coef :: %

taylorIfCan uls ==
  n := getExpon uls
  n < 0 =>
    uls := removeZeroes(-n,uls)
    getExpon(uls) < 0 => "failed"
    getUTS uls
  n = 0 => getUTS uls
  getUTS(uls) * monom(1,n :: NNI)

taylor uls ==
  (uts := taylorIfCan uls) case "failed" =>
    error "taylor: Laurent series has a pole"
  uts :: UTS

termExpon: TERM -> I
termExpon term == term.k
termCoef: TERM -> Coef
termCoef term == term.c
rec: (I,Coef) -> TERM
rec(exponent,coef) == [exponent,coef]

recs: (ST,I) -> Stream TERM
recs(st,n) == delay
  empty? st => empty()
  zero? (coef := frst st) => recs(rst st,n + 1)
  concat(rec(n,coef),recs(rst st,n + 1))

terms x == recs(coefficients getUTS x,getExpon x)

recsToCoefs: (Stream TERM,I) -> ST
recsToCoefs(st,n) == delay
  empty? st => empty()
  term := frst st; ex := termExpon term
  n = ex => concat(termCoef term,recsToCoefs(rst st,n + 1))
  concat(0,recsToCoefs(rst st,n + 1))

series st ==
  empty? st => 0
  ex := termExpon frst st
  laurent(ex,series recsToCoefs(st,ex))

```

--% normalizations

```

removeZeroes x ==
  empty? coefficients(xUTS := getUTS x) => 0
  coefficient(xUTS,0) = 0 =>
    removeZeroes laurent(getExpon(x) + 1,quoByVar xUTS)
  x

removeZeroes(n,x) ==
  n <= 0 => x
  empty? coefficients(xUTS := getUTS x) => 0
  coefficient(xUTS,0) = 0 =>
    removeZeroes(n - 1,laurent(getExpon(x) + 1,quoByVar xUTS))
  x

```

--% predicates

```

x = y ==
  EQ(x,y)$Lisp => true
  (expDiff := getExpon(x) - getExpon(y)) = 0 =>
    getUTS(x) = getUTS(y)
  abs(expDiff) > _$streamCount$Lisp => false
  expDiff > 0 =>
    getUTS(x) * monom(1,expDiff :: NNI) = getUTS(y)
    getUTS(y) * monom(1,(- expDiff) :: NNI) = getUTS(x)

pole? x ==
  (n := degree x) >= 0 => false
  x := removeZeroes(-n,x)
  degree x < 0

```

--% arithmetic

```

x + y ==
  n := getExpon(x) - getExpon(y)
  n >= 0 =>
    laurent(getExpon y,getUTS(y) + getUTS(x) * monom(1,n::NNI))
    laurent(getExpon x,getUTS(x) + getUTS(y) * monom(1,(-n)::NNI))

x - y ==
  n := getExpon(x) - getExpon(y)
  n >= 0 =>
    laurent(getExpon y,getUTS(x) * monom(1,n::NNI) - getUTS(y))
    laurent(getExpon x,getUTS(x) - getUTS(y) * monom(1,(-n)::NNI))

x:% * y:% == laurent(getExpon x + getExpon y,getUTS x * getUTS y)

x:% ** n:NNI ==
  zero? n =>

```

```

    zero? x => error "0 ** 0 is undefined"
    1
    laurent(n * getExpon(x),getUTS(x) ** n)

recip x ==
  x := removeZeroes(1000,x)
  zero? coefficient(x,d := degree x) => "failed"
  (uts := recip getUTS x) case "failed" => "failed"
  laurent(-d,uts :: UTS)

elt(uls1:%,uls2:%) ==
  (uts := taylorIfCan uls2) case "failed" =>
    error "elt: second argument must have positive order"
  uts2 := uts :: UTS
  not zero? coefficient(uts2,0) =>
    error "elt: second argument must have positive order"
  if (deg := getExpon uls1) < 0 then uls1 := removeZeroes(-deg,uls1)
  (deg := getExpon uls1) < 0 =>
    (recipr := recip(uts2 :: %)) case "failed" =>
      error "elt: second argument not invertible"
    uls1 := taylor(uls1 * monomial(1,-deg))
    (elt(uls1,uts2) :: %) * (recipr :: %) ** ((-deg) :: NNI)
  elt(taylor uls1,uts2) :: %

eval(uls:%,r:Coef) ==
  if (n := getExpon uls) < 0 then uls := removeZeroes(-n,uls)
  uts := getUTS uls
  (n := getExpon uls) < 0 =>
    zero? r => error "eval: 0 raised to negative power"
    (recipr := recip r) case "failed" =>
      error "eval: non-unit raised to negative power"
    (recipr :: Coef) ** ((-n) :: NNI) * $$STAYLOR eval(uts,r)
  zero? n => eval(uts,r)
  r ** (n :: NNI) * $$STAYLOR eval(uts,r)

--% values

variable x == variable getUTS x
center x == center getUTS x

coefficient(x,n) ==
  a := n - getExpon(x)
  a >= 0 => coefficient(getUTS x,a :: NNI)
  0

elt(x:%,n:I) == coefficient(x,n)

--% other functions

order x == getExpon x + order getUTS x

```

```

order(x,n) ==
  (m := n - (e := getExpon x)) < 0 => n
  e + order(getUTS x,m :: NNI)

truncate(x,n) ==
  (m := n - (e := getExpon x)) < 0 => 0
  laurent(e,truncate(getUTS x,m :: NNI))

truncate(x,n1,n2) ==
  if n2 < n1 then (n1,n2) := (n2,n1)
  (m1 := n1 - (e := getExpon x)) < 0 => truncate(x,n2)
  laurent(e,truncate(getUTS x,m1 :: NNI,(n2 - e) :: NNI))

if Coef has IntegralDomain then
  rationalFunction(x,n) ==
    (m := n - (e := getExpon x)) < 0 => 0
    poly := polynomial(getUTS x,m :: NNI) :: RF
    zero? e => poly
    v := variable(x) :: RF; c := center(x) :: P :: RF
    positive? e => poly * (v - c) ** (e :: NNI)
    poly / (v - c) ** ((-e) :: NNI)

  rationalFunction(x,n1,n2) ==
    if n2 < n1 then (n1,n2) := (n2,n1)
    (m1 := n1 - (e := getExpon x)) < 0 => rationalFunction(x,n2)
    poly := polynomial(getUTS x,m1 :: NNI,(n2 - e) :: NNI) :: RF
    zero? e => poly
    v := variable(x) :: RF; c := center(x) :: P :: RF
    positive? e => poly * (v - c) ** (e :: NNI)
    poly / (v - c) ** ((-e) :: NNI)

-- La fonction < exquo > manque dans laurent.spad,
--les lignes suivantes le mettent en evidence :
--
--ls := laurent(0,series [i for i in 1..])$ULS(INT,x,0)
---- missing function in laurent.spad of Axiom 2.0a version of
---- Friday March 10, 1995 at 04:15:22 on 615:
--exquo(ls,ls)
--
-- Je l'ai ajoutee a laurent.spad.
--
--Frederic Lehobey
x exquo y ==
  x := removeZeroes(1000,x)
  y := removeZeroes(1000,y)
  zero? coefficient(y, d := degree y) => "failed"
  (uts := (getUTS x) exquo (getUTS y)) case "failed" => "failed"
  laurent(degree x-d,uts :: UTS)

if Coef has coerce: Symbol -> Coef then

```

```

if Coef has "**": (Coef,I) -> Coef then

  approximate(x,n) ==
    (m := n - (e := getExpon x)) < 0 => 0
    app := approximate(getUTS x,m :: NNI)
    zero? e => app
    app * ((variable(x) :: Coef) - center(x)) ** e

complete x == laurent(getExpon x,complete getUTS x)
extend(x,n) ==
  e := getExpon x
  (m := n - e) < 0 => x
  laurent(e,extend(getUTS x,m :: NNI))

map(f:Coef -> Coef,x:%) == laurent(getExpon x,map(f,getUTS x))

multiplyCoefficients(f,x) ==
  e := getExpon x
  laurent(e,multiplyCoefficients((z1:I):Coef +-> f(e + z1),getUTS x))

multiplyExponents(x,n) ==
  laurent(n * getExpon x,multiplyExponents(getUTS x,n))

differentiate x ==
  e := getExpon x
  laurent(e - 1,
    multiplyCoefficients((z1:I):Coef +-> (e + z1)::Coef,getUTS x))

if Coef has PartialDifferentialRing(Symbol) then
  differentiate(x:%,s:Symbol) ==
    (s = variable(x)) => differentiate x
    map((z1:Coef):Coef +-> differentiate(z1,s),x)
    - differentiate(center x,s)*differentiate(x)

characteristic() == characteristic()$Coef

if Coef has Field then

  retract(x:%):UTS == taylor x
  retractIfCan(x:%):Union(UTS,"failed") == taylorIfCan x

  (x:%) ** (n:I) ==
    zero? n =>
      zero? x => error "0 ** 0 is undefined"
      1
    n > 0 => laurent(n * getExpon(x),getUTS(x) ** (n :: NNI))
  xInv := inv x; minusN := (-n) :: NNI
  laurent(minusN * getExpon(xInv),getUTS(xInv) ** minusN)

  (x:UTS) * (y:%) == (x :: %) * y

```

```

(x:%) * (y:UTS) == x * (y :: %)

inv x ==
  (xInv := recip x) case "failed" =>
    error "multiplicative inverse does not exist"
  xInv :: %

(x:%) / (y:%) ==
  (yInv := recip y) case "failed" =>
    error "inv: multiplicative inverse does not exist"
  x * (yInv :: %)

(x:UTS) / (y:UTS) == (x :: %) / (y :: %)

numer x ==
  (n := degree x) >= 0 => taylor x
  x := removeZeroes(-n,x)
  (n := degree x) = 0 => taylor x
  getUTS x

denom x ==
  (n := degree x) >= 0 => 1
  x := removeZeroes(-n,x)
  (n := degree x) = 0 => 1
  monom(1,(-n) :: NNI)

--% algebraic and transcendental functions

if Coef has Algebra Fraction Integer then

  coerce(r:RN) == r :: Coef :: %

  if Coef has Field then
    (x:%) ** (r:RN) == x **$EFULS r

  exp x == exp(x)$EFULS
  log x == log(x)$EFULS
  sin x == sin(x)$EFULS
  cos x == cos(x)$EFULS
  tan x == tan(x)$EFULS
  cot x == cot(x)$EFULS
  sec x == sec(x)$EFULS
  csc x == csc(x)$EFULS
  asin x == asin(x)$EFULS
  acos x == acos(x)$EFULS
  atan x == atan(x)$EFULS
  acot x == acot(x)$EFULS
  asec x == asec(x)$EFULS
  acsc x == acsc(x)$EFULS
  sinh x == sinh(x)$EFULS

```

```

cosh x == cosh(x)$EFULS
tanh x == tanh(x)$EFULS
coth x == coth(x)$EFULS
sech x == sech(x)$EFULS
csch x == csch(x)$EFULS
asinh x == asinh(x)$EFULS
acosh x == acosh(x)$EFULS
atanh x == atanh(x)$EFULS
acoth x == acoth(x)$EFULS
asech x == asech(x)$EFULS
acsch x == acsch(x)$EFULS

ratInv: I -> Coef
ratInv n ==
  zero? n => 1
  inv(n :: RN) :: Coef

integrate x ==
  not zero? coefficient(x,-1) =>
    error "integrate: series has term of order -1"
  e := getExpon x
  laurent(e+1,multiplyCoefficients((z:I):Coef+-->ratInv(e+1+z),getUTS x))

if Coef has integrate: (Coef,Symbol) -> Coef and _
  Coef has variables: Coef -> List Symbol then
    integrate(x:%,s:Symbol) ==
      (s = variable(x)) => integrate x
      not entry?(s,variables center x)
      => map((z1:Coef):Coef+-->integrate(z1,s),x)
      error "integrate: center is a function of variable of integration"

if Coef has TranscendentalFunctionCategory and _
  Coef has PrimitiveFunctionCategory and _
  Coef has AlgebraicallyClosedFunctionSpace Integer then

  integrateWithOneAnswer: (Coef,Symbol) -> Coef
  integrateWithOneAnswer(f,s) ==
    res := integrate(f,s)$FunctionSpaceIntegration(I,Coef)
    res case Coef => res :: Coef
    first(res :: List Coef)

  integrate(x:%,s:Symbol) ==
    (s = variable(x)) => integrate x
    not entry?(s,variables center x) =>
      map((z1:Coef):Coef +--> integrateWithOneAnswer(z1,s),x)
      error "integrate: center is a function of variable of integration"

termOutput:(I,Coef,OUT) -> OUT
termOutput(k,c,vv) ==
-- creates a term c * vv ** k

```

```

k = 0 => c :: OUT
mon :=
  k = 1 => vv
  vv ** (k :: OUT)
c = 1 => mon
c = -1 => -mon
(c :: OUT) * mon

showAll?:() -> Boolean
-- check a global Lisp variable
showAll?() == true

termsToOutputForm:(I,ST,OUT) -> OUT
termsToOutputForm(m,uu,xxx) ==
  l : L OUT := empty()
  empty? uu => (0$Coef) :: OUT
  n : NNI ; count : NNI := _$streamCount$Lisp
  for n in 0..count while not empty? uu repeat
    if frst(uu) ^= 0 then
      l := concat(termOutput((n :: I) + m,frst(uu),xxx),l)
      uu := rst uu
  if showAll?() then
    for n in (count + 1).. while explicitEntries? uu and _
      not eq?(uu,rst uu) repeat
        if frst(uu) ^= 0 then
          l := concat(termOutput((n::I) + m,frst(uu),xxx),l)
          uu := rst uu
  l :=
    explicitlyEmpty? uu => l
    eq?(uu,rst uu) and frst uu = 0 => l
    concat(prefix("0" :: OUT,[xxx ** ((n :: I) + m) :: OUT]),l)
  empty? l => (0$Coef) :: OUT
  reduce("+",reverse_! l)

coerce(x:):OUT ==
  x := removeZeroes(_$streamCount$Lisp,x)
  m := degree x
  uts := getUTS x
  p := coefficients uts
  var := variable uts; cen := center uts
  xxx :=
    zero? cen => var :: OUT
    paren(var :: OUT - cen :: OUT)
  termsToOutputForm(m,p,xxx)

```

```

"ULSCONS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ULSCONS"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"ULSCCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ULSCCAT"]
"ULSCONS" -> "ULSCCAT"
"ULSCONS" -> "ACFS"

```

domain UP UnivariatePolynomial

— UnivariatePolynomial.input —

```

)set break resume
)sys rm -f UnivariatePolynomial.output
)spool UnivariatePolynomial.output
)set message test on
)set message auto off
)clear all
--S 1 of 35
(p,q) : UP(x,INT)
--R
--R
--E 1
Type: Void

--S 2 of 35
p := (3*x-1)**2 * (2*x + 8)
--R
--R
--R      3      2
--R      (2)  18x  + 60x  - 46x + 8
--R
--E 2
Type: UnivariatePolynomial(x,Integer)

--S 3 of 35
q := (1 - 6*x + 9*x**2)**2
--R
--R
--R      4      3      2
--R      (3)  81x  - 108x  + 54x  - 12x + 1
--R
--E 3
Type: UnivariatePolynomial(x,Integer)

--S 4 of 35
p**2 + p*q
--R
--R

```

```

--R      7      6      5      4      3      2
--R  (4) 1458x + 3240x - 7074x + 10584x - 9282x + 4120x - 878x + 72
--R                                         Type: UnivariatePolynomial(x,Integer)
--E 4

```

```

--S 5 of 35
leadingCoefficient p
--R
--R
--R  (5) 18
--R                                         Type: PositiveInteger
--E 5

```

```

--S 6 of 35
degree p
--R
--R
--R  (6) 3
--R                                         Type: PositiveInteger
--E 6

```

```

--S 7 of 35
reductum p
--R
--R
--R      2
--R  (7) 60x - 46x + 8
--R                                         Type: UnivariatePolynomial(x,Integer)
--E 7

```

```

--S 8 of 35
gcd(p,q)
--R
--R
--R      2
--R  (8) 9x - 6x + 1
--R                                         Type: UnivariatePolynomial(x,Integer)
--E 8

```

```

--S 9 of 35
lcm(p,q)
--R
--R
--R      5      4      3      2
--R  (9) 162x + 432x - 756x + 408x - 94x + 8
--R                                         Type: UnivariatePolynomial(x,Integer)
--E 9

```

```

--S 10 of 35
resultant(p,q)

```

```

--R
--R
--R (10)  0
--R
--R                                         Type: NonNegativeInteger
--E 10

```

```

--S 11 of 35
D p
--R
--R
--R
--R      2
--R (11)  54x  + 120x - 46
--R
--R                                         Type: UnivariatePolynomial(x,Integer)
--E 11

```

```

--S 12 of 35
p(2)
--R
--R
--R (12)  300
--R
--R                                         Type: PositiveInteger
--E 12

```

```

--S 13 of 35
p(q)
--R
--R
--R (13)
--R      12      11      10      9      8
--R 9565938x  - 38263752x  + 70150212x  - 77944680x  + 58852170x
--R +
--R      7      6      5      4      3      2
--R - 32227632x  + 13349448x  - 4280688x  + 1058184x  - 192672x  + 23328x
--R +
--R - 1536x + 40
--R
--R                                         Type: UnivariatePolynomial(x,Integer)
--E 13

```

```

--S 14 of 35
q(p)
--R
--R
--R (14)
--R      12      11      10      9      8
--R 8503056x  + 113374080x  + 479950272x  + 404997408x  - 1369516896x
--R +
--R      7      6      5      4      3
--R - 626146848x  + 2939858712x  - 2780728704x  + 1364312160x  - 396838872x
--R +
--R      2

```



```

--S 21 of 35
p
--R
--R
--R      3      2
--R (21) 18x  + 60x  - 46x + 8
--R
--R                                          Type: UnivariatePolynomial(x,Integer)
--E 21

```

```

--S 22 of 35
squareTerms p
--R
--R   Compiling function squareTerms with type UnivariatePolynomial(x,
--R   Integer) -> UnivariatePolynomial(x,Integer)
--R
--R      6      4      2
--R (22) 324x  + 3600x  + 2116x  + 64
--R
--R                                          Type: UnivariatePolynomial(x,Integer)
--E 22

```

```

--S 23 of 35
(r,s) : UP(a1,FRAC INT)
--R
--R
--R                                          Type: Void
--E 23

```

```

--S 24 of 35
r := a1**2 - 2/3
--R
--R
--R      2      2
--R (24) a1  - -
--R      3
--R
--R                                          Type: UnivariatePolynomial(a1,Fraction(Integer))
--E 24

```

```

--S 25 of 35
s := a1 + 4
--R
--R
--R (25) a1 + 4
--R
--R                                          Type: UnivariatePolynomial(a1,Fraction(Integer))
--E 25

```

```

--S 26 of 35
r quo s
--R
--R
--R (26) a1 - 4

```

```

--R                                     Type: UnivariatePolynomial(a1,Fraction(Integer))
--E 26

--S 27 of 35
r rem s
--R
--R
--R      46
--R  (27)  --
--R      3
--R                                     Type: UnivariatePolynomial(a1,Fraction(Integer))
--E 27

--S 28 of 35
d := divide(r, s)
--R
--R
--R      46
--R  (28)  [quotient= a1 - 4,remainder= --]
--R      3
--R  IType: Record(quotient: UnivariatePolynomial(a1,Fraction(Integer)),...
--E 28

--S 29 of 35
r - (d.quotient * s + d.remainder)
--R
--R
--R  (29)  0
--R                                     Type: UnivariatePolynomial(a1,Fraction(Integer))
--E 29

--S 30 of 35
integrate r
--R
--R
--R      1  3  2
--R  (30)  - a1 - - a1
--R      3    3
--R                                     Type: UnivariatePolynomial(a1,Fraction(Integer))
--E 30

--S 31 of 35
integrate s
--R
--R
--R      1  2
--R  (31)  - a1 + 4a1
--R      2
--R                                     Type: UnivariatePolynomial(a1,Fraction(Integer))
--E 31

```

```

--S 32 of 35
t : UP(a1,FRAC POLY INT)
--R
--R
--R                                          Type: Void
--E 32

--S 33 of 35
t := a1**2 - a1/b2 + (b1**2-b1)/(b2+3)
--R
--R
--R
--R              2
--R      2      1      b1 - b1
--R  (33)  a1 - -- a1 + -----
--R          b2      b2 + 3
--R
--R      Type: UnivariatePolynomial(a1,Fraction(Polynomial(Integer)))
--E 33

--S 34 of 35
u : FRAC POLY INT := t
--R
--R
--R
--R      2 2      2      2
--R      a1 b2 + (b1 - b1 + 3a1 - a1)b2 - 3a1
--R  (34) -----
--R              2
--R            b2 + 3b2
--R
--R                                          Type: Fraction(Polynomial(Integer))
--E 34

--S 35 of 35
u :: UP(b1,?)
--R
--R
--R
--R              2
--R      1      2      1      a1 b2 - a1
--R  (35) ----- b1 - ----- b1 + -----
--R      b2 + 3      b2 + 3      b2
--R
--R      Type: UnivariatePolynomial(b1,Fraction(Polynomial(Integer)))
--E 35
)spool
)lisp (bye)

```

— UnivariatePolynomial.help —

=====

UnivariatePolynomial examples

=====

The domain constructor `UnivariatePolynomial` (abbreviated `UP`) creates domains of univariate polynomials in a specified variable. For example, the domain `UP(a1,POLY FRAC INT)` provides polynomials in the single variable `a1` whose coefficients are general polynomials with rational number coefficients.

Restriction: Axiom does not allow you to create types where `UnivariatePolynomial` is contained in the coefficient type of `Polynomial`. Therefore, `UP(x,POLY INT)` is legal but `POLY UP(x,INT)` is not.

`UP(x,INT)` is the domain of polynomials in the single variable `x` with integer coefficients.

```
(p,q) : UP(x,INT)
                                Type: Void

p := (3*x-1)**2 * (2*x + 8)
      3      2
    18x  + 60x  - 46x + 8
                                Type: UnivariatePolynomial(x,Integer)

q := (1 - 6*x + 9*x**2)**2
      4      3      2
    81x  - 108x  + 54x  - 12x + 1
                                Type: UnivariatePolynomial(x,Integer)
```

The usual arithmetic operations are available for univariate polynomials.

```
p**2 + p*q
      7      6      5      4      3      2
    1458x  + 3240x  - 7074x  + 10584x  - 9282x  + 4120x  - 878x + 72
                                Type: UnivariatePolynomial(x,Integer)
```

The operation `leadingCoefficient` extracts the coefficient of the term of highest degree.

```
leadingCoefficient p
    18
                                Type: PositiveInteger
```

The operation `degree` returns the degree of the polynomial. Since the polynomial has only one variable, the variable is not supplied to operations like `degree`.

```
degree p
    3
                                Type: PositiveInteger
```


The reductum of the polynomial, the polynomial obtained by subtracting the term of highest order, is returned by reductum.

```
reductum p
      2
    60x  - 46x + 8
                                     Type: UnivariatePolynomial(x,Integer)
```

The operation gcd computes the greatest common divisor of two polynomials.

```
gcd(p,q)
      2
    9x  - 6x + 1
                                     Type: UnivariatePolynomial(x,Integer)
```

The operation lcm computes the least common multiple.

```
lcm(p,q)
      5      4      3      2
    162x  + 432x  - 756x  + 408x  - 94x + 8
                                     Type: UnivariatePolynomial(x,Integer)
```

The operation resultant computes the resultant of two univariate polynomials. In the case of p and q, the resultant is 0 because they share a common root.

```
resultant(p,q)
      0
                                     Type: NonNegativeInteger
```

To compute the derivative of a univariate polynomial with respect to its variable, use the function D.

```
D p
      2
    54x  + 120x - 46
                                     Type: UnivariatePolynomial(x,Integer)
```

Univariate polynomials can also be used as if they were functions. To evaluate a univariate polynomial at some point, apply the polynomial to the point.

```
p(2)
    300
                                     Type: PositiveInteger
```

The same syntax is used for composing two univariate polynomials, i.e. substituting one polynomial for the variable in another. This substitutes q for the variable in p.

```

p(q)
      12      11      10      9      8
      9565938x - 38263752x + 70150212x - 77944680x + 58852170x
+
      7      6      5      4      3      2
      - 32227632x + 13349448x - 4280688x + 1058184x - 192672x + 23328x
+
      - 1536x + 40
                                     Type: UnivariatePolynomial(x,Integer)

```

This substitutes p for the variable in q.

```

q(p)
      12      11      10      9      8
      8503056x + 113374080x + 479950272x + 404997408x - 1369516896x
+
      7      6      5      4      3
      - 626146848x + 2939858712x - 2780728704x + 1364312160x - 396838872x
+
      2
      69205896x - 6716184x + 279841
                                     Type: UnivariatePolynomial(x,Integer)

```

To obtain a list of coefficients of the polynomial, use coefficients.

```

l := coefficients p
    [18,60,- 46,8]
                                     Type: List Integer

```

From this you can use gcd and reduce to compute the content of the polynomial.

```

reduce(gcd,l)
      2
                                     Type: PositiveInteger

```

Alternatively (and more easily), you can just call content.

```

content p
      2
                                     Type: PositiveInteger

```

Note that the operation coefficients omits the zero coefficients from the list. Sometimes it is useful to convert a univariate polynomial to a vector whose i-th position contains the degree i-1 coefficient of the polynomial.

```

ux := (x**4+2*x+3)::UP(x,INT)
      4
      x + 2x + 3

```

Type: UnivariatePolynomial(x,Integer)

To get a complete vector of coefficients, use the operation `vectorise`, which takes a univariate polynomial and an integer denoting the length of the desired vector.

```
vectorise(ux,5)
[3,2,0,0,1]
```

Type: Vector Integer

It is common to want to do something to every term of a polynomial, creating a new polynomial in the process.

This is a function for iterating across the terms of a polynomial, squaring each term.

```
squareTerms(p) == reduce(+,[t**2 for t in monomials p])
Type: Void
```

Recall what `p` looked like.

```
p
      3      2
    18x  + 60x  - 46x + 8
Type: UnivariatePolynomial(x,Integer)
```

We can demonstrate `squareTerms` on `p`.

```
squareTerms p
      6      4      2
    324x  + 3600x  + 2116x  + 64
Type: UnivariatePolynomial(x,Integer)
```

When the coefficients of the univariate polynomial belong to a field, it is possible to compute quotients and remainders. For example, when the coefficients are rational numbers, as opposed to integers. The important property of a field is that non-zero elements can be divided and produce another element. The quotient of the integers 2 and 3 is not another integer.

```
(r,s) : UP(a1,FRAC INT)
Type: Void
```

```
r := a1**2 - 2/3
      2      2
    a1  - -
      3
Type: UnivariatePolynomial(a1,Fraction Integer)
```

```
s := a1 + 4
```

$$a1 + 4$$

Type: UnivariatePolynomial(a1,Fraction Integer)

When the coefficients are rational numbers or rational expressions,
the operation quo computes the quotient of two polynomials.

$$\frac{r}{a1 - 4}$$

Type: UnivariatePolynomial(a1,Fraction Integer)

The operation rem computes the remainder.

$$\frac{r \text{ rem } s}{46}$$

Type: UnivariatePolynomial(a1,Fraction Integer)

The operation divide can be used to return a record of both components.

$$d := \text{divide}(r, s)$$

$$\left[\text{quotient} = \frac{46}{a1 - 4}, \text{remainder} = \frac{--}{3} \right]$$

Type: Record(quotient: UnivariatePolynomial(a1,Fraction Integer),
remainder: UnivariatePolynomial(a1,Fraction Integer))

Now we check the arithmetic!

$$r - (d.\text{quotient} * s + d.\text{remainder})$$

Type: UnivariatePolynomial(a1,Fraction Integer)

It is also possible to integrate univariate polynomials when the
coefficients belong to a field.

$$\int \left(\frac{1}{3} a1^3 - \frac{2}{3} a1 \right) da1$$

Type: UnivariatePolynomial(a1,Fraction Integer)

$$\int \left(\frac{1}{2} a1^2 + 4a1 \right) da1$$

Type: UnivariatePolynomial(a1,Fraction Integer)

One application of univariate polynomials is to see expressions in terms
of a specific variable.

We start with a polynomial in a_1 whose coefficients are quotients of polynomials in b_1 and b_2 .

```
t : UP(a1,FRAC POLY INT)
      Type: Void
```

Since in this case we are not talking about using multivariate polynomials in only two variables, we use Polynomial. We also use Fraction because we want fractions.

```
t := a1**2 - a1/b2 + (b1**2-b1)/(b2+3)
      2
      2    1    b1 - b1
a1  - -- a1 + -----
      b2      b2 + 3
      Type: UnivariatePolynomial(a1,Fraction Polynomial Integer)
```

We push all the variables into a single quotient of polynomials.

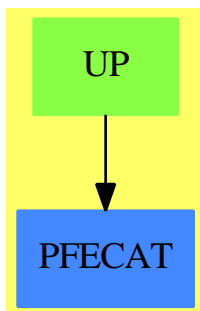
```
u : FRAC POLY INT := t
      2 2    2    2
a1 b2  + (b1 - b1 + 3a1 - a1)b2 - 3a1
-----
      2
      b2  + 3b2
      Type: Fraction Polynomial Integer
```

Alternatively, we can view this as a polynomial in the variable b_1 . This is a mode-directed conversion: you indicate as much of the structure as you care about and let Axiom decide on the full type and how to do the transformation.

```
u :: UP(b1,?)
      1    2    1    2
----- b1  - ----- b1 + -----
b2 + 3    b2 + 3    b2
      Type: UnivariatePolynomial(b1,Fraction Polynomial Integer)
```

See Also:

```
o )help MultivariatePolynomial
o )help DistributedMultivariatePolynomial
o )show UnivariatePolynomial
```

UnivariatePolynomial (UP)**See**

- ⇒ “FreeModule” (FM) [7](#) on page [1013](#)
- ⇒ “PolynomialRing” (PR) [17](#) on page [2092](#)
- ⇒ “SparseUnivariatePolynomial” (SUP) [20](#) on page [2466](#)

Exports:

0	1
associates?	binomThmExpt
characteristic	charthRoot
coefficient	coefficients
coerce	composite
conditionP	content
convert	D
degree	differentiate
discriminant	divide
divideExponents	elt
euclideanSize	eval
expressIdealMember	exquo
extendedEuclidean	factor
factorPolynomial	factorSquareFreePolynomial
fmecg	gcd
gcdPolynomial	ground
ground?	hash
init	integrate
isExpt	isPlus
isTimes	karatsubaDivide
latex	lcm
leadingCoefficient	leadingMonomial
mainVariable	makeSUP
mapExponents	map
max	min
minimumDegree	monicDivide
monomial	monomial?
monomials	multiEuclidean
multiplyExponents	multivariate
nextItem	numberOfMonomials
one?	order
patternMatch	popo!
prime?	primitivePart
primitiveMonomials	principalIdeal
pseudoDivide	pseudoQuotient
pseudoRemainder	recip
reducedSystem	reductum
resultant	retract
retractIfCan	sample
separate	shiftLeft
shiftRight	sizeLess?
solveLinearPolynomialEquation	squareFree
squareFreePart	squareFreePolynomial
subResultantGcd	subtractIfCan
totalDegree	totalDegree
unit?	unitCanonical
unitNormal	univariate
unmakeSUP	variables
vectorise	zero?
?*?	?**?
?+?	?-?
-?	?=?
?^?	?..?
?~=?	?/?
?<?	?<=?

— domain UP UnivariatePolynomial —

```

)abbrev domain UP UnivariatePolynomial
++ Author: Mark Botch
++ Description:
++ This domain represents univariate polynomials in some symbol
++ over arbitrary (not necessarily commutative) coefficient rings.
++ The representation is sparse
++ in the sense that only non-zero terms are represented.
++ Note that if the coefficient ring is a field, then this domain
++ forms a euclidean domain.

UnivariatePolynomial(x:Symbol, R:Ring):
  UnivariatePolynomialCategory(R) with
    coerce: Variable(x) -> %
      ++ coerce(x) converts the variable x to a univariate polynomial.
    fmeq: (% , NonNegativeInteger, R, %) -> %
      ++ fmeq(p1, e, r, p2) finds x : p1 - r * x**e * p2
  == SparseUnivariatePolynomial(R) add
  Rep:=SparseUnivariatePolynomial(R)
  coerce(p:%):OutputForm == outputForm(p, outputForm x)
  coerce(v:Variable(x)):% == monomial(1, 1)

```

— UP.dotabb —

```

"UP" [color="#88FF44", href="bookvol10.3.pdf#nameddest=UP"]
"PFECAT" [color="#4488FF", href="bookvol10.2.pdf#nameddest=PFECAT"]
"UP" -> "PFECAT"

```

domain UPXS UnivariatePuisseuxSeries

— UnivariatePuisseuxSeries.input —

```

)set break resume
)sys rm -f UnivariatePuisseuxSeries.output
)spool UnivariatePuisseuxSeries.output
)set message test on
)set message auto off
)clear all

```



```

--S 1 of 1
)show UnivariatePuisseuxSeries
--R
--R UnivariatePuisseuxSeries(Coef: Ring,var: Symbol,cen: Coef) is a domain constructor
--R Abbreviation for UnivariatePuisseuxSeries is UPXS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for UPXS
--R
--R----- Operations -----
--R ?? : (Coef,%) -> %           ?? : (%,Coef) -> %
--R ?? : (%,%) -> %             ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %             ?-? : (%,%) -> %
--R -? : % -> %                 ?=? : (%,%) -> Boolean
--R 1 : () -> %                 0 : () -> %
--R ?? : (%,PositiveInteger) -> %   center : % -> Coef
--R coerce : Variable(var) -> %     coerce : Integer -> %
--R coerce : % -> OutputForm         complete : % -> %
--R degree : % -> Fraction(Integer) hash : % -> SingleInteger
--R latex : % -> String             leadingCoefficient : % -> Coef
--R leadingMonomial : % -> %         map : ((Coef -> Coef),%) -> %
--R monomial? : % -> Boolean         one? : % -> Boolean
--R order : % -> Fraction(Integer)   pole? : % -> Boolean
--R recip : % -> Union(%, "failed")  reductum : % -> %
--R sample : () -> %               variable : % -> Symbol
--R zero? : % -> Boolean            ?~=? : (%,%) -> Boolean
--R ??? : (%,Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (Fraction(Integer),%) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (NonNegativeInteger,%) -> %
--R ??? : (%,Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (%,%) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (%,Integer) -> % if Coef has FIELD
--R ??? : (%,NonNegativeInteger) -> %
--R ?/? : (%,%) -> % if Coef has FIELD
--R ?/? : (%,Coef) -> % if Coef has FIELD
--R D : % -> % if Coef has *: (Fraction(Integer),Coef) -> Coef
--R D : (%,NonNegativeInteger) -> % if Coef has *: (Fraction(Integer),Coef) -> Coef
--R D : (%,Symbol) -> % if Coef has *: (Fraction(Integer),Coef) -> Coef and Coef has PDRING(SYMB
--R D : (%,List(Symbol)) -> % if Coef has *: (Fraction(Integer),Coef) -> Coef and Coef has PDRIN
--R D : (%,Symbol,NonNegativeInteger) -> % if Coef has *: (Fraction(Integer),Coef) -> Coef and C
--R D : (%,List(Symbol),List(NonNegativeInteger)) -> % if Coef has *: (Fraction(Integer),Coef) -
--R ?? : (%,Integer) -> % if Coef has FIELD
--R ?? : (%,NonNegativeInteger) -> %
--R acos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acoth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsch : % -> % if Coef has ALGEBRA(FRAC(INT))

```

```

--R approximate : (% , Fraction(Integer)) -> Coef if Coef has **: (Coef, Fraction(Integer)) -> Coef and Coef has
--R asec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R associates? : (% , %) -> Boolean if Coef has INTDOM
--R atan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R atanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(% , "failed") if Coef has CHARNZ
--R coefficient : (% , Fraction(Integer)) -> Coef
--R coerce : % -> % if Coef has INTDOM
--R coerce : Fraction(Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R coerce : UnivariateTaylorSeries(Coef, var, cen) -> %
--R coerce : UnivariateLaurentSeries(Coef, var, cen) -> %
--R coerce : Coef -> % if Coef has COMRING
--R cos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R coth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R differentiate : (% , Variable(var)) -> %
--R differentiate : % -> % if Coef has *: (Fraction(Integer), Coef) -> Coef
--R differentiate : (% , NonNegativeInteger) -> % if Coef has *: (Fraction(Integer), Coef) -> Coef
--R differentiate : (% , Symbol) -> % if Coef has *: (Fraction(Integer), Coef) -> Coef and Coef has PDRING(SYM)
--R differentiate : (% , List(Symbol)) -> % if Coef has *: (Fraction(Integer), Coef) -> Coef and Coef has PDRING(SYM)
--R differentiate : (% , Symbol, NonNegativeInteger) -> % if Coef has *: (Fraction(Integer), Coef) -> Coef and Coef has PDRING(SYM)
--R differentiate : (% , List(Symbol), List(NonNegativeInteger)) -> % if Coef has *: (Fraction(Integer), Coef) -> Coef and Coef has PDRING(SYM)
--R divide : (% , %) -> Record(quotient: % , remainder: %) if Coef has FIELD
--R ?? : (% , %) -> % if Fraction(Integer) has SGROUP
--R ?? : (% , Fraction(Integer)) -> Coef
--R euclideanSize : % -> NonNegativeInteger if Coef has FIELD
--R eval : (% , Coef) -> Stream(Coef) if Coef has **: (Coef, Fraction(Integer)) -> Coef
--R exp : % -> % if Coef has ALGEBRA(FRAC(INT))
--R expressIdealMember : (List(% , %) -> Union(List(% , "failed") if Coef has FIELD
--R exquo : (% , %) -> Union(% , "failed") if Coef has INTDOM
--R extend : (% , Fraction(Integer)) -> %
--R extendedEuclidean : (% , %) -> Record(coef1: % , coef2: % , generator: %) if Coef has FIELD
--R extendedEuclidean : (% , % , %) -> Union(Record(coef1: % , coef2: % , "failed") if Coef has FIELD
--R factor : % -> Factored(%) if Coef has FIELD
--R gcd : (% , %) -> % if Coef has FIELD
--R gcd : List(%) -> % if Coef has FIELD
--R gcdPolynomial : (SparseUnivariatePolynomial(% , SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%))
--R integrate : (% , Variable(var)) -> % if Coef has ALGEBRA(FRAC(INT))
--R integrate : (% , Symbol) -> % if Coef has integrate: (Coef, Symbol) -> Coef and Coef has variables: Coef -> %
--R integrate : % -> % if Coef has ALGEBRA(FRAC(INT))
--R inv : % -> % if Coef has FIELD
--R laurent : % -> UnivariateLaurentSeries(Coef, var, cen)
--R laurentIfCan : % -> Union(UnivariateLaurentSeries(Coef, var, cen) , "failed")

```

```

--R laurentRep : % -> UnivariateLaurentSeries(Coef,var,cen)
--R lcm : (%,% ) -> % if Coef has FIELD
--R lcm : List(% ) -> % if Coef has FIELD
--R log : % -> % if Coef has ALGEBRA(FRAC(INT))
--R monomial : (% ,List(SingletonAsOrderedSet),List(Fraction(Integer))) -> %
--R monomial : (% ,SingletonAsOrderedSet,Fraction(Integer)) -> %
--R monomial : (Coef,Fraction(Integer)) -> %
--R multiEuclidean : (List(% ),%) -> Union(List(% ),"failed") if Coef has FIELD
--R multiplyExponents : (% ,Fraction(Integer)) -> %
--R multiplyExponents : (% ,PositiveInteger) -> %
--R nthRoot : (% ,Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R order : (% ,Fraction(Integer)) -> Fraction(Integer)
--R pi : () -> % if Coef has ALGEBRA(FRAC(INT))
--R prime? : % -> Boolean if Coef has FIELD
--R principalIdeal : List(% ) -> Record(coef: List(% ),generator: % ) if Coef has FIELD
--R puioux : (Fraction(Integer),UnivariateLaurentSeries(Coef,var,cen)) -> %
--R ?quo? : (% ,%) -> % if Coef has FIELD
--R rationalPower : % -> Fraction(Integer)
--R ?rem? : (% ,%) -> % if Coef has FIELD
--R retract : % -> UnivariateTaylorSeries(Coef,var,cen)
--R retract : % -> UnivariateLaurentSeries(Coef,var,cen)
--R retractIfCan : % -> Union(UnivariateTaylorSeries(Coef,var,cen),"failed")
--R retractIfCan : % -> Union(UnivariateLaurentSeries(Coef,var,cen),"failed")
--R sec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R series : (NonNegativeInteger,Stream(Record(k: Fraction(Integer),c: Coef))) -> %
--R sin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sizeLess? : (% ,%) -> Boolean if Coef has FIELD
--R sqrt : % -> % if Coef has ALGEBRA(FRAC(INT))
--R squareFree : % -> Factored(% ) if Coef has FIELD
--R squareFreePart : % -> % if Coef has FIELD
--R subtractIfCan : (% ,%) -> Union(% ,"failed")
--R tan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R tanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R terms : % -> Stream(Record(k: Fraction(Integer),c: Coef))
--R truncate : (% ,Fraction(Integer),Fraction(Integer)) -> %
--R truncate : (% ,Fraction(Integer)) -> %
--R unit? : % -> Boolean if Coef has INTDOM
--R unitCanonical : % -> % if Coef has INTDOM
--R unitNormal : % -> Record(unit: % ,canonical: % ,associate: % ) if Coef has INTDOM
--R variables : % -> List(SingletonAsOrderedSet)
--R
--E 1

)spool
)lisp (bye)

```

— UnivariatePuisseuxSeries.help —

```
=====
UnivariatePuisseuxSeries examples
=====
```

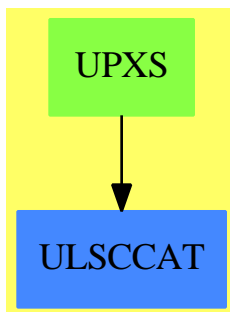
UnivariatePuisseuxSeries is a domain representing Puiseux series in one variable with coefficients in an arbitrary ring. The parameters of the type specify the coefficient ring, the power series variable, and the center of the power series expansion. For example, `UnivariatePuisseuxSeries(Integer,x,3)` represents Puiseux series in $(x - 3)$ with Integer coefficients.

See Also:

o `)show UnivariatePuisseuxSeries`

—————→

UnivariatePuisseuxSeries (UPXS)



See

⇒ “UnivariatePuisseuxSeriesConstructor” (UPXSCONS) [22](#) on page [2841](#)

Exports:

0	1	acos	acosh
acot	acoth	acsc	acsch
approximate	asec	asech	asin
asinh	associates?	atan	atanh
center	characteristic	charthRoot	coefficient
coerce	complete	cos	cosh
cot	coth	csc	csch
D	degree	differentiate	divide
euclideanSize	eval	exp	expressIdealMember
exquo	extend	extendedEuclidean	factor
gcd	gcdPolynomial	hash	integrate
inv	latex	laurent	laurentIfCan
laurentRep	lcm	leadingCoefficient	leadingMonomial
log	map	monomial	monomial?
multiEuclidean	multiplyExponents	nthRoot	one?
order	pi	pole?	prime?
principalIdeal	puiseux	rationalPower	recip
reductum	retract	retractIfCan	sample
sec	sech	series	sin
sinh	sizeLess?	sqrt	squareFree
squareFreePart	subtractIfCan	tan	tanh
terms	truncate	unit?	unitCanonical
unitNormal	variable	variables	zero?
?*?	?**?	?+?	?-?
-?	?=?	?^?	?~=?
?/?	?..?	?quo?	?rem?

— domain UPXS UnivariatePuisseuxSeries —

```
)abbrev domain UPXS UnivariatePuisseuxSeries
```

```
++ Author: Clifton J. Williamson
```

```
++ Date Created: 28 January 1990
```

```
++ Date Last Updated: 21 September 1993
```

```
++ Description:
```

```
++ Dense Puiseux series in one variable
```

```
UnivariatePuisseuxSeries(Coef,var,cen): Exports == Implementation where
```

```
Coef : Ring
```

```
var : Symbol
```

```
cen : Coef
```

```
I ==> Integer
```

```
L ==> List
```

```
NNI ==> NonNegativeInteger
```

```
OUT ==> OutputForm
```

```
RN ==> Fraction Integer
```

```
ST ==> Stream Coef
```

```
UTS ==> UnivariateTaylorSeries(Coef,var,cen)
```

```

ULS ==> UnivariateLaurentSeries(Coef,var,cen)

Exports ==> Join(UnivariatePuisseuxSeriesConstructorCategory(Coef,ULS),_
    RetractableTo UTS) with
coerce: Variable(var) -> %
    ++ coerce(var) converts the series variable \spad{var} into a
    ++ Puisseux series.
differentiate: (% ,Variable(var)) -> %
    ++ \spad{differentiate(f(x),x)} returns the derivative of
    ++ \spad{f(x)} with respect to \spad{x}.
if Coef has Algebra Fraction Integer then
    integrate: (% ,Variable(var)) -> %
        ++ \spad{integrate(f(x))} returns an anti-derivative of the power
        ++ series \spad{f(x)} with constant coefficient 0.
        ++ We may integrate a series when we can divide coefficients
        ++ by integers.

Implementation ==> UnivariatePuisseuxSeriesConstructor(Coef,ULS) add

Rep := Record(expon:RN,lSeries:ULS)

getExpon: % -> RN
getExpon pxs == pxs.expon

variable upxs == var
center   upxs == cen

coerce(uts:UTS) == uts :: ULS :: %

retractIfCan(upxs:%):Union(UTS,"failed") ==
    (ulsIfCan := retractIfCan(upxs)@Union(ULS,"failed")) case "failed" =>
        "failed"
    retractIfCan(ulsIfCan :: ULS)

--retract(upxs:%):UTS ==
--(ulsIfCan := retractIfCan(upxs)@Union(ULS,"failed")) case "failed" =>
--error "retractIfCan: series has fractional exponents"
--utsIfCan := retractIfCan(ulsIfCan :: ULS)@Union(UTS,"failed")
--utsIfCan case "failed" =>
--error "retractIfCan: series has negative exponents"
--utsIfCan :: UTS

coerce(v:Variable(var)) ==
    zero? cen => monomial(1,1)
    monomial(1,1) + monomial(cen,0)

if Coef has "*": (Fraction Integer, Coef) -> Coef then
    differentiate(upxs:% ,v:Variable(var)) == differentiate upxs

if Coef has Algebra Fraction Integer then

```

```

integrate(upxs:%,v:Variable(var)) == integrate upxs

if Coef has coerce: Symbol -> Coef then
  if Coef has "**": (Coef,RN) -> Coef then

    roundDown: RN -> I
    roundDown rn ==
      -- returns the largest integer <= rn
      (den := denom rn) = 1 => numer rn
      n := (num := numer rn) quo den
      positive?(num) => n
      n - 1

    stToCoef: (ST,Coef,NNI,NNI) -> Coef
    stToCoef(st,term,n,n0) ==
      (n > n0) or (empty? st) => 0
      first(st) * term ** n + stToCoef(rst st,term,n + 1,n0)

    approximateLaurent: (ULS,Coef,I) -> Coef
    approximateLaurent(x,term,n) ==
      (m := n - (e := degree x)) < 0 => 0
      app := stToCoef(coefficients taylorRep x,term,0,m :: NNI)
      zero? e => app
      app * term ** (e :: RN)

    approximate(x,r) ==
      e := rationalPower(x)
      term := ((variable(x) :: Coef) - center(x)) ** e
      approximateLaurent(laurentRep x,term,roundDown(r / e))

  termOutput: (RN,Coef,OUT) -> OUT
  termOutput(k,c,vv) ==
    -- creates a term c * vv ** k
    k = 0 => c :: OUT
    mon :=
      k = 1 => vv
      vv ** (k :: OUT)
    c = 1 => mon
    c = -1 => -mon
    (c :: OUT) * mon

  showAll?:() -> Boolean
  -- check a global Lisp variable
  showAll?() == true

  termsToOutputForm: (RN,RN,ST,OUT) -> OUT
  termsToOutputForm(m,rat,uu,xxx) ==
    l : L OUT := empty()
    empty? uu => 0 :: OUT
    n : NNI; count : NNI := _$streamCount$Lisp

```

```

for n in 0..count while not empty? uu repeat
  if frst(uu) ^= 0 then
    l := concat(termOutput((n :: I) * rat + m, frst uu, xxx), l)
    uu := rst uu
  if showAll?() then
    for n in (count + 1).. while explicitEntries? uu and _
      not eq?(uu, rst uu) repeat
        if frst(uu) ^= 0 then
          l := concat(termOutput((n :: I) * rat + m, frst uu, xxx), l)
          uu := rst uu
    l :=
      explicitlyEmpty? uu => l
      eq?(uu, rst uu) and frst uu = 0 => l
      concat(prefix("0" :: OUT, [xxx ** ((n::I) * rat + m) :: OUT])), l)
    empty? l => 0 :: OUT
    reduce("+", reverse_! l)

coerce(upxs:%):OUT ==
  rat := getExpon upxs; uls := laurentRep upxs
  count : I := _$streamCount$Lisp
  uls := removeZeroes(_$streamCount$Lisp, uls)
  m : RN := (degree uls) * rat
  p := coefficients taylorRep uls
  xxx :=
    zero? cen => var :: OUT
    paren(var :: OUT - cen :: OUT)
  termsToOutputForm(m, rat, p, xxx)

```

— UPXS.dotabb —

```

"UPXS" [color="#88FF44", href="bookvol10.3.pdf#nameddest=UPXS"]
"ULSCCAT" [color="#4488FF", href="bookvol10.2.pdf#nameddest=ULSCCAT"]
"UPXS" -> "ULSCCAT"

```

domain UPXSCONS UnivariatePuisseuxSeriesConstructor

— UnivariatePuisseuxSeriesConstructor.input —

```
)set break resume
```



```

)sys rm -f UnivariatePuisseuxSeriesConstructor.output
)spool UnivariatePuisseuxSeriesConstructor.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariatePuisseuxSeriesConstructor
--R
--R UnivariatePuisseuxSeriesConstructor(Coef: Ring, ULS: UnivariateLaurentSeriesCategory(Coef))
--R Abbreviation for UnivariatePuisseuxSeriesConstructor is UPXSCONS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for UPXSCONS
--R
--R----- Operations -----
--R ?? : (Coef, %) -> %           ?? : (%, Coef) -> %
--R ?? : (%, %) -> %             ?? : (Integer, %) -> %
--R ?? : (PositiveInteger, %) -> %   ??? : (%, PositiveInteger) -> %
--R ?+? : (%, %) -> %             ?-? : (%, %) -> %
--R -? : % -> %                   ?=? : (%, %) -> Boolean
--R 1 : () -> %                   0 : () -> %
--R ?? : (%, PositiveInteger) -> %   center : % -> Coef
--R coerce : ULS -> %               coerce : Integer -> %
--R coerce : % -> OutputForm         complete : % -> %
--R degree : % -> Fraction(Integer)  hash : % -> SingleInteger
--R latex : % -> String              laurent : % -> ULS
--R laurentRep : % -> ULS            leadingCoefficient : % -> Coef
--R leadingMonomial : % -> %         map : ((Coef -> Coef), %) -> %
--R monomial? : % -> Boolean         one? : % -> Boolean
--R order : % -> Fraction(Integer)   pole? : % -> Boolean
--R recip : % -> Union(%, "failed")  reductum : % -> %
--R retract : % -> ULS              sample : () -> %
--R variable : % -> Symbol           zero? : % -> Boolean
--R ~=? : (%, %) -> Boolean
--R ?? : (%, Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (Fraction(Integer), %) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (NonNegativeInteger, %) -> %
--R ??? : (%, Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (%, %) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (%, Integer) -> % if Coef has FIELD
--R ??? : (%, NonNegativeInteger) -> %
--R ?/? : (%, %) -> % if Coef has FIELD
--R ?/? : (%, Coef) -> % if Coef has FIELD
--R D : % -> % if Coef has *: (Fraction(Integer), Coef) -> Coef
--R D : (%, NonNegativeInteger) -> % if Coef has *: (Fraction(Integer), Coef) -> Coef
--R D : (%, Symbol) -> % if Coef has *: (Fraction(Integer), Coef) -> Coef and Coef has PDRING(SYMB
--R D : (%, List(Symbol)) -> % if Coef has *: (Fraction(Integer), Coef) -> Coef and Coef has PDRIN
--R D : (%, Symbol, NonNegativeInteger) -> % if Coef has *: (Fraction(Integer), Coef) -> Coef and C
--R D : (%, List(Symbol), List(NonNegativeInteger)) -> % if Coef has *: (Fraction(Integer), Coef) -
--R ?? : (%, Integer) -> % if Coef has FIELD

```

```

--R ?? : (%,NonNegativeInteger) -> %
--R acos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acoth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R approximate : (%,Fraction(Integer)) -> Coef if Coef has **: (Coef,Fraction(Integer)) -> Coef and Coef has
--R asec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R associates? : (%,%) -> Boolean if Coef has INTDOM
--R atan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R atanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if Coef has CHARNZ
--R coefficient : (%,Fraction(Integer)) -> Coef
--R coerce : % -> % if Coef has INTDOM
--R coerce : Fraction(Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R coerce : Coef -> % if Coef has COMRING
--R cos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R coth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R differentiate : % -> % if Coef has *: (Fraction(Integer),Coef) -> Coef
--R differentiate : (%,NonNegativeInteger) -> % if Coef has *: (Fraction(Integer),Coef) -> Coef
--R differentiate : (%,Symbol) -> % if Coef has *: (Fraction(Integer),Coef) -> Coef and Coef has PDRING(SYM
--R differentiate : (%,List(Symbol)) -> % if Coef has *: (Fraction(Integer),Coef) -> Coef and Coef has PDRIN
--R differentiate : (%,Symbol,NonNegativeInteger) -> % if Coef has *: (Fraction(Integer),Coef) -> Coef and
--R differentiate : (%,List(Symbol),List(NonNegativeInteger)) -> % if Coef has *: (Fraction(Integer),Coef)
--R divide : (%,%) -> Record(quotient: %,remainder: %) if Coef has FIELD
--R ?? : (%,%) -> % if Fraction(Integer) has SGROUP
--R ?? : (%,Fraction(Integer)) -> Coef
--R euclideanSize : % -> NonNegativeInteger if Coef has FIELD
--R eval : (%,Coef) -> Stream(Coef) if Coef has **: (Coef,Fraction(Integer)) -> Coef
--R exp : % -> % if Coef has ALGEBRA(FRAC(INT))
--R expressIdealMember : (List(%),%) -> Union(List(%), "failed") if Coef has FIELD
--R exquo : (%,%) -> Union(%, "failed") if Coef has INTDOM
--R extend : (%,Fraction(Integer)) -> %
--R extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %) if Coef has FIELD
--R extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %), "failed") if Coef has FIELD
--R factor : % -> Factored(%) if Coef has FIELD
--R gcd : (%,%) -> % if Coef has FIELD
--R gcd : List(%) -> % if Coef has FIELD
--R gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynom
--R integrate : (%,Symbol) -> % if Coef has integrate: (Coef,Symbol) -> Coef and Coef has variables: Coef -
--R integrate : % -> % if Coef has ALGEBRA(FRAC(INT))

```

```

--R inv : % -> % if Coef has FIELD
--R laurentIfCan : % -> Union(ULS,"failed")
--R lcm : (%,% ) -> % if Coef has FIELD
--R lcm : List(%) -> % if Coef has FIELD
--R log : % -> % if Coef has ALGEBRA(FRAC(INT))
--R monomial : (% ,List(SingletonAsOrderedSet),List(Fraction(Integer))) -> %
--R monomial : (% ,SingletonAsOrderedSet,Fraction(Integer)) -> %
--R monomial : (Coef,Fraction(Integer)) -> %
--R multiEuclidean : (List(%),%) -> Union(List(%),"failed") if Coef has FIELD
--R multiplyExponents : (% ,Fraction(Integer)) -> %
--R multiplyExponents : (% ,PositiveInteger) -> %
--R nthRoot : (% ,Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R order : (% ,Fraction(Integer)) -> Fraction(Integer)
--R pi : () -> % if Coef has ALGEBRA(FRAC(INT))
--R prime? : % -> Boolean if Coef has FIELD
--R principalIdeal : List(%) -> Record(coef: List(%),generator: %) if Coef has FIELD
--R puioux : (Fraction(Integer),ULS) -> %
--R ?quo? : (% ,%) -> % if Coef has FIELD
--R rationalPower : % -> Fraction(Integer)
--R ?rem? : (% ,%) -> % if Coef has FIELD
--R retractIfCan : % -> Union(ULS,"failed")
--R sec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R series : (NonNegativeInteger,Stream(Record(k: Fraction(Integer),c: Coef))) -> %
--R sin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sizeLess? : (% ,%) -> Boolean if Coef has FIELD
--R sqrt : % -> % if Coef has ALGEBRA(FRAC(INT))
--R squareFree : % -> Factored(%) if Coef has FIELD
--R squareFreePart : % -> % if Coef has FIELD
--R subtractIfCan : (% ,%) -> Union(%,"failed")
--R tan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R tanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R terms : % -> Stream(Record(k: Fraction(Integer),c: Coef))
--R truncate : (% ,Fraction(Integer),Fraction(Integer)) -> %
--R truncate : (% ,Fraction(Integer)) -> %
--R unit? : % -> Boolean if Coef has INTDOM
--R unitCanonical : % -> % if Coef has INTDOM
--R unitNormal : % -> Record(unit: % ,canonical: % ,associate: %) if Coef has INTDOM
--R variables : % -> List(SingletonAsOrderedSet)
--R
--E 1

```

```

)spool
)lisp (bye)

```

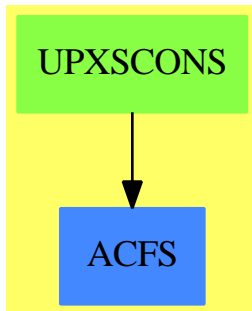
```
=====
UnivariatePuisseuxSeriesConstructor examples
=====
```

This package enables one to construct a univariate Puiseux series domain from a univariate Laurent series domain. Univariate Puiseux series are represented by a pair $[r, f(x)]$, where r is a positive rational number and $f(x)$ is a Laurent series. This pair represents the Puiseux series $f(x^r)$.

See Also:

o)show UnivariatePuisseuxSeriesConstructor

UnivariatePuisseuxSeriesConstructor (UPXSCONS)



See

⇒ “UnivariatePuisseuxSeries” (UPXS) [22](#) on page [2833](#)

Exports:

0	1	acos	acosh
acot	acoth	acsc	acsch
approximate	asec	asech	asin
asinh	associates?	atan	atanh
center	characteristic	charthRoot	coefficient
coerce	complete	cos	cosh
cot	coth	csc	csch
D	degree	differentiate	divide
euclideanSize	eval	exp	expressIdealMember
exquo	extend	extendedEuclidean	factor
gcd	gcdPolynomial	hash	integrate
inv	latex	laurent	laurentIfCan
laurentRep	lcm	leadingCoefficient	leadingMonomial
log	map	monomial	monomial?
multiEuclidean	multiplyExponents	nthRoot	one?
order	pi	pole?	prime?
principalIdeal	puiseux	rationalPower	recip
reductum	retract	retractIfCan	sample
sec	sech	series	sin
sinh	sizeLess?	sqrt	squareFree
squareFreePart	subtractIfCan	tan	tanh
terms	truncate	unit?	unitCanonical
unitNormal	variable	variables	zero?
?*?	?**?	?+?	?-?
-?	?=?	?^?	?~=?
?/?	?..?	?quo?	?rem?

— domain UPXSCONS UnivariatePuisseuxSeriesConstructor —

```
)abbrev domain UPXSCONS UnivariatePuisseuxSeriesConstructor
++ Author: Clifton J. Williamson
++ Date Created: 9 May 1989
++ Date Last Updated: 30 November 1994
++ Description:
++ This package enables one to construct a univariate Puiseux series
++ domain from a univariate Laurent series domain. Univariate
++ Puiseux series are represented by a pair \spad{[r,f(x)]}, where r is
++ a positive rational number and \spad{f(x)} is a Laurent series.
++ This pair represents the Puiseux series \spad{f(x^r)}.
```

```
UnivariatePuisseuxSeriesConstructor(Coef,ULS):_
Exports == Implementation where
Coef : Ring
ULS : UnivariateLaurentSeriesCategory Coef
I ==> Integer
L ==> List
NNI ==> NonNegativeInteger
```

```

OUT ==> OutputForm
PI  ==> PositiveInteger
RN  ==> Fraction Integer
ST  ==> Stream Coef
LTerm ==> Record(k:I,c:Coef)
PTerm ==> Record(k:RN,c:Coef)
ST2LP ==> StreamFunctions2(LTerm,PTerm)
ST2PL ==> StreamFunctions2(PTerm,LTerm)

Exports ==> UnivariatePuisseuxSeriesConstructorCategory(Coef,Uls)

Implementation ==> add

--% representation

Rep := Record(expon:RN,lSeries:Uls)

getExpon: % -> RN
getUls : % -> Uls

getExpon pxs == pxs.expon
getUls pxs == pxs.lSeries

--% creation and destruction

puisseux(n,ls) == [n,ls]
laurentRep x == getUls x
rationalPower x == getExpon x
degree x == getExpon(x) * degree(getUls(x))

0 == puisseux(1,0)
1 == puisseux(1,1)

monomial(c,k) ==
  k = 0 => c :: %
  k < 0 => puisseux(-k,monomial(c,-1))
  puisseux(k,monomial(c,1))

coerce(ls:Uls) == puisseux(1,ls)
coerce(r:Coef) == r :: Uls :: %
coerce(i:I) == i :: Coef :: %

laurentIfCan upxs ==
  r := getExpon upxs
  one? denom r =>
    (denom r) = 1 =>
      multiplyExponents(getUls upxs,numer(r) :: PI)
    "failed"

laurent upxs ==

```

```

(uls := laurentIfCan upxs) case "failed" =>
  error "laurent: Puiseux series has fractional powers"
uls :: ULS

multExp: (RN,LTerm) -> PTerm
multExp(r,lTerm) == [r * lTerm.k,lTerm.c]

terms upxs ==
  map((t1:LTerm):PTerm+>multExp(getExpon upxs,t1),terms getULS upxs)$ST2LP

clearDen: (I,PTerm) -> LTerm
clearDen(n,lTerm) ==
  (int := retractIfCan(n * lTerm.k@Union(I,"failed"))) case "failed" =>
    error "series: inappropriate denominator"
  [int :: I,lTerm.c]

series(n,stream) ==
  str := map((t1:PTerm):LTerm +> clearDen(n,t1),stream)$ST2PL
  puiseux(1/n,series str)

--% normalizations

rewrite:(%,PI) -> %
rewrite(upxs,m) ==
  -- rewrites a series in x**r as a series in x**(r/m)
  puiseux((getExpon upxs)*(1/m),multiplyExponents(getULS upxs,m))

ratGcd: (RN,RN) -> RN
ratGcd(r1,r2) ==
  -- if r1 = prod(p prime,p ** ep(r1)) and
  -- if r2 = prod(p prime,p ** ep(r2)), then
  -- ratGcd(r1,r2) = prod(p prime,p ** min(ep(r1),ep(r2)))
  gcd(numer r1,numer r2) / lcm(denom r1,denom r2)

withNewExpon:(%,RN) -> %
withNewExpon(upxs,r) ==
  rewrite(upxs,numer(getExpon(upxs)/r) pretend PI)

--% predicates

upxs1 = upxs2 ==
  r1 := getExpon upxs1; r2 := getExpon upxs2
  ls1 := getULS upxs1; ls2 := getULS upxs2
  (r1 = r2) => (ls1 = ls2)
  r := ratGcd(r1,r2)
  m1 := numer(getExpon(upxs1)/r) pretend PI
  m2 := numer(getExpon(upxs2)/r) pretend PI
  multiplyExponents(ls1,m1) = multiplyExponents(ls2,m2)

pole? upxs == pole? getULS upxs

```

```

--% arithmetic

applyFcn:((ULS,ULS) -> ULS,%,%) -> %
applyFcn(op,pxs1,pxs2) ==
  r1 := getExpon pxs1; r2 := getExpon pxs2
  ls1 := getULS pxs1; ls2 := getULS pxs2
  (r1 = r2) => puioux(r1,op(ls1,ls2))
  r := ratGcd(r1,r2)
  m1 := numer(getExpon(pxs1)/r) pretend PI
  m2 := numer(getExpon(pxs2)/r) pretend PI
  puioux(r,op(multiplyExponents(ls1,m1),multiplyExponents(ls2,m2)))

pxs1 + pxs2      == applyFcn((z1:ULS,z2:ULS):ULS+>z1 +$ULS z2,pxs1,pxs2)
pxs1 - pxs2      == applyFcn((z1:ULS,z2:ULS):ULS+>z1 -$ULS z2,pxs1,pxs2)
pxs1:% * pxs2:% == applyFcn((z1:ULS,z2:ULS):ULS+>z1 *$ULS z2,pxs1,pxs2)

pxs:% ** n:NNI == puioux(getExpon pxs,getULS(pxs)**n)

recip pxs ==
  rec := recip getULS pxs
  rec case "failed" => "failed"
  puioux(getExpon pxs,rec :: ULS)

RATALG : Boolean := Coef has Algebra(Fraction Integer)

elt(upxs1:%,upxs2:%) ==
  uls1 := laurentRep upxs1; uls2 := laurentRep upxs2
  r1 := rationalPower upxs1; r2 := rationalPower upxs2
  (n := retractIfCan(r1)@Union(Integer,"failed")) case Integer =>
    puioux(r2,uls1(uls2 ** r1))
  RATALG =>
    if zero? (coef := coefficient(uls2,deg := degree uls2)) then
      deg := order(uls2,deg + 1000)
      zero? (coef := coefficient(uls2,deg)) =>
        error "elt: series with many leading zero coefficients"
    -- a fractional power of a Laurent series may not be defined:
    -- if f(x) = c * x**n + ..., then f(x) ** (p/q) will be defined
    -- only if q divides n
    b := lcm(denom r1,deg); c := b quo deg
    mon : ULS := monomial(1,c)
    uls2 := elt(uls2,mon) ** r1
    puioux(r2*(1/c),elt(uls1,uls2))
  error "elt: rational powers not available for this coefficient domain"

if Coef has "***": (Coef,Integer) -> Coef and
  Coef has "***": (Coef, RN) -> Coef then
  eval(upxs:%,a:Coef) == eval(getULS upxs,a ** getExpon(upxs))

if Coef has Field then

```



```

pxs1:% / pxs2:% == applyFcn((z1:ULS,z2:ULS):ULS+>z1 /$ULS z2,pxs1,pxs2)

inv upxs ==
  (invUpxs := recip upxs) case "failed" =>
    error "inv: multiplicative inverse does not exist"
  invUpxs :: %

--% values

variable upxs == variable getULS upxs
center upxs == center getULS upxs

coefficient(upxs,rn) ==
--   one? denom(n := rn / getExpon upxs) =>
     (denom(n := rn / getExpon upxs)) = 1 =>
       coefficient(getULS upxs,numer n)
     0

elt(upxs:%,rn:RN) == coefficient(upxs,rn)

--% other functions

roundDown: RN -> I
roundDown rn ==
  -- returns the largest integer <= rn
  (den := denom rn) = 1 => numer rn
  n := (num := numer rn) quo den
  positive?(num) => n
  n - 1

roundUp: RN -> I
roundUp rn ==
  -- returns the smallest integer >= rn
  (den := denom rn) = 1 => numer rn
  n := (num := numer rn) quo den
  positive?(num) => n + 1
  n

order upxs == getExpon upxs * order getULS upxs
order(upxs,r) ==
  e := getExpon upxs
  ord := order(getULS upxs, n := roundDown(r / e))
  ord = n => r
  ord * e

truncate(upxs,r) ==
  e := getExpon upxs
  puioux(e,truncate(getULS upxs,roundDown(r / e)))

```

```

truncate(upxs,r1,r2) ==
  e := getExpon upxs
  puioux(e,truncate(getULS upxs,roundUp(r1 / e),roundDown(r2 / e)))

complete upxs == puioux(getExpon upxs,complete getULS upxs)
extend(upxs,r) ==
  e := getExpon upxs
  puioux(e,extend(getULS upxs,roundDown(r / e)))

map(fcn,upxs) == puioux(getExpon upxs,map(fcn,getULS upxs))

characteristic() == characteristic()$Coef

-- multiplyCoefficients(f,upxs) ==
-- r := getExpon upxs
-- puioux(r,multiplyCoefficients(f(#1 * r),getULS upxs))

multiplyExponents(upxs:%,n:RN) ==
  puioux(n * getExpon(upxs),getULS upxs)
multiplyExponents(upxs:%,n:PI) ==
  puioux(n * getExpon(upxs),getULS upxs)

if Coef has "*" : (Fraction Integer, Coef) -> Coef then

  differentiate upxs ==
    r := getExpon upxs
    puioux(r,differentiate getULS upxs) * monomial(r :: Coef,r-1)

  if Coef has PartialDifferentialRing(Symbol) then

    differentiate(upxs:%,s:Symbol) ==
      (s = variable(upxs)) => differentiate upxs
      dcds := differentiate(center upxs,s)
      map((z1:Coef):Coef+>->differentiate(z1,s),upxs)
        - dcds*differentiate(upxs)

  if Coef has Algebra Fraction Integer then

    coerce(r:RN) == r :: Coef :: %

    ratInv: RN -> Coef
    ratInv r ==
      zero? r => 1
      inv(r) :: Coef

    integrate upxs ==
      not zero? coefficient(upxs,-1) =>
        error "integrate: series has term of order -1"
      r := getExpon upxs
      uls := getULS upxs

```

```

uls := multiplyCoefficients((z1:Integer):Coef+>ratInv(z1*r+1),uls)
monomial(1,1) * puioux(r,uls)

if Coef has integrate: (Coef,Symbol) -> Coef and _
  Coef has variables: Coef -> List Symbol then

  integrate(upxs:%,s:Symbol) ==
    (s = variable(upxs)) => integrate upxs
    not entry?(s,variables center upxs)
    => map((z1:Coef):Coef +> integrate(z1,s),upxs)
    error "integrate: center is a function of variable of integration"

if Coef has TranscendentalFunctionCategory and _
  Coef has PrimitiveFunctionCategory and _
  Coef has AlgebraicallyClosedFunctionSpace Integer then

  integrateWithOneAnswer: (Coef,Symbol) -> Coef
  integrateWithOneAnswer(f,s) ==
    res := integrate(f,s)$FunctionSpaceIntegration(I,Coef)
    res case Coef => res :: Coef
    first(res :: List Coef)

  integrate(upxs:%,s:Symbol) ==
    (s = variable(upxs)) => integrate upxs
    not entry?(s,variables center upxs) =>
      map((z1:Coef):Coef +> integrateWithOneAnswer(z1,s),upxs)
      error "integrate: center is a function of variable of integration"

if Coef has Field then
  (upxs:%) ** (q:RN) ==
    num := numer q; den := denom q
    one? den => upxs ** num
    den = 1 => upxs ** num
    r := rationalPower upxs; uls := laurentRep upxs
    deg := degree uls
    if zero?(coef := coefficient(uls,deg)) then
      deg := order(uls,deg + 1000)
      zero?(coef := coefficient(uls,deg)) =>
        error "power of series with many leading zero coefficients"
    ulsPow := (uls * monomial(1,-deg)$ULS) ** q
    puioux(r,ulsPow) * monomial(1,deg*q*r)

applyUnary: (ULS -> ULS,%) -> %
applyUnary(fcn,upxs) ==
  puioux(rationalPower upxs,fcn laurentRep upxs)

exp upxs == applyUnary(exp,upxs)
log upxs == applyUnary(log,upxs)
sin upxs == applyUnary(sin,upxs)
cos upxs == applyUnary(cos,upxs)

```

```

tan upxs == applyUnary(tan,upxs)
cot upxs == applyUnary(cot,upxs)
sec upxs == applyUnary(sec,upxs)
csc upxs == applyUnary(csc,upxs)
asin upxs == applyUnary(asin,upxs)
acos upxs == applyUnary(acos,upxs)
atan upxs == applyUnary(atan,upxs)
acot upxs == applyUnary(acot,upxs)
asec upxs == applyUnary(asec,upxs)
acsc upxs == applyUnary(acsc,upxs)
sinh upxs == applyUnary(sinh,upxs)
cosh upxs == applyUnary(cosh,upxs)
tanh upxs == applyUnary(tanh,upxs)
coth upxs == applyUnary(coth,upxs)
sech upxs == applyUnary(sech,upxs)
csch upxs == applyUnary(csch,upxs)
asinh upxs == applyUnary(asinh,upxs)
acosh upxs == applyUnary(acosh,upxs)
atanh upxs == applyUnary(atanh,upxs)
acoth upxs == applyUnary(acoth,upxs)
asech upxs == applyUnary(asech,upxs)
acsch upxs == applyUnary(acsch,upxs)

```

— UPXSCONS.dotabb —

```

"UPXSCONS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=UPXSCONS"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"UPXSCONS" -> "ACFS"

```

domain UPXSSING UnivariatePuisseuxSeriesWithExponentialSingularity

— UnivariatePuisseuxSeriesWithExponentialSingularity.input —

```

)set break resume
)sys rm -f UnivariatePuisseuxSeriesWithExponentialSingularity.output
)spool UnivariatePuisseuxSeriesWithExponentialSingularity.output
)set message test on
)set message auto off

```

```

)clear all

--S 1 of 1
)show UnivariatePuisseuxSeriesWithExponentialSingularity
--R
--R UnivariatePuisseuxSeriesWithExponentialSingularity(R: Join(OrderedSet,RetractableTo(Integer),
--R Abbreviation for UnivariatePuisseuxSeriesWithExponentialSingularity is UPXSSING
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for UPXSSING
--R
--R----- Operations -----
--R ?? : (%,%) -> %                ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %                ?-? : (%,%) -> %
--R -? : % -> %                     ?=? : (%,%) -> Boolean
--R 1 : () -> %                     0 : () -> %
--R ?? : (%,PositiveInteger) -> %   associates? : (%,%) -> Boolean
--R coerce : % -> %                 coerce : Integer -> %
--R coerce : % -> OutputForm         ground? : % -> Boolean
--R hash : % -> SingleInteger        latex : % -> String
--R leadingMonomial : % -> %         monomial? : % -> Boolean
--R one? : % -> Boolean              recip : % -> Union(%, "failed")
--R reductum : % -> %                sample : () -> %
--R unit? : % -> Boolean              unitCanonical : % -> %
--R zero? : % -> Boolean              ?~=? : (%,%) -> Boolean
--R ?? : (%,Fraction(Integer)) -> % if UnivariatePuisseuxSeries(FE,var,cen) has ALGEBRA(FRAC(INT
--R ?? : (Fraction(Integer),%) -> % if UnivariatePuisseuxSeries(FE,var,cen) has ALGEBRA(FRAC(INT
--R ?? : (UnivariatePuisseuxSeries(FE,var,cen),%) -> %
--R ?? : (%,UnivariatePuisseuxSeries(FE,var,cen)) -> %
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R ?/? : (%,UnivariatePuisseuxSeries(FE,var,cen)) -> % if UnivariatePuisseuxSeries(FE,var,cen) ha
--R ?? : (%,NonNegativeInteger) -> %
--R binomThmExpt : (%,%,NonNegativeInteger) -> % if UnivariatePuisseuxSeries(FE,var,cen) has COMR
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if UnivariatePuisseuxSeries(FE,var,cen) has CHARNZ
--R coefficient : (%,ExponentialOfUnivariatePuisseuxSeries(FE,var,cen)) -> UnivariatePuisseuxSerie
--R coefficients : % -> List(UnivariatePuisseuxSeries(FE,var,cen))
--R coerce : Fraction(Integer) -> % if UnivariatePuisseuxSeries(FE,var,cen) has ALGEBRA(FRAC(INT
--R coerce : UnivariatePuisseuxSeries(FE,var,cen) -> %
--R content : % -> UnivariatePuisseuxSeries(FE,var,cen) if UnivariatePuisseuxSeries(FE,var,cen) ha
--R degree : % -> ExponentialOfUnivariatePuisseuxSeries(FE,var,cen)
--R dominantTerm : % -> Union(Record(%term: Record(%coef: UnivariatePuisseuxSeries(FE,var,cen),%e
--R exquo : (%,%) -> Union(%, "failed")
--R exquo : (%,UnivariatePuisseuxSeries(FE,var,cen)) -> Union(%, "failed") if UnivariatePuisseuxSer
--R ground : % -> UnivariatePuisseuxSeries(FE,var,cen)
--R leadingCoefficient : % -> UnivariatePuisseuxSeries(FE,var,cen)
--R limitPlus : % -> Union(OrderedCompletion(FE), "failed")
--R map : ((UnivariatePuisseuxSeries(FE,var,cen) -> UnivariatePuisseuxSeries(FE,var,cen)),%) -> %
--R mapExponents : ((ExponentialOfUnivariatePuisseuxSeries(FE,var,cen) -> ExponentialOfUnivariate

```

```

--R minimumDegree : % -> ExponentialOfUnivariatePuisseuxSeries(FE,var,cen)
--R monomial : (UnivariatePuisseuxSeries(FE,var,cen),ExponentialOfUnivariatePuisseuxSeries(FE,var,cen)) -> %
--R numberOfMonomials : % -> NonNegativeInteger
--R pomopo! : (% ,UnivariatePuisseuxSeries(FE,var,cen),ExponentialOfUnivariatePuisseuxSeries(FE,var,cen),%) ->
--R primitivePart : % -> % if UnivariatePuisseuxSeries(FE,var,cen) has GCDDOM
--R retract : % -> UnivariatePuisseuxSeries(FE,var,cen)
--R retract : % -> Fraction(Integer) if UnivariatePuisseuxSeries(FE,var,cen) has RETRACT(FRAC(INT))
--R retract : % -> Integer if UnivariatePuisseuxSeries(FE,var,cen) has RETRACT(INT)
--R retractIfCan : % -> Union(UnivariatePuisseuxSeries(FE,var,cen),"failed")
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if UnivariatePuisseuxSeries(FE,var,cen) has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(Integer,"failed") if UnivariatePuisseuxSeries(FE,var,cen) has RETRACT(INT)
--R subtractIfCan : (% ,%) -> Union(%,"failed")
--R unitNormal : % -> Record(unit: %,canonical: %,associate: %)
--R
--E 1

)spool
)lisp (bye)

```

— UnivariatePuisseuxSeriesWithExponentialSingularity.help —

=====

UnivariatePuisseuxSeriesWithExponentialSingularity examples

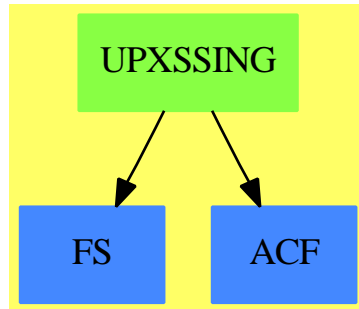
=====

UnivariatePuisseuxSeriesWithExponentialSingularity is a domain used to represent functions with essential singularities. Objects in this domain are sums, where each term in the sum is a univariate Puiseux series times the exponential of a univariate Puiseux series. Thus, the elements of this domain are sums of expressions of the form $g(x) * \exp(f(x))$, where $g(x)$ is a univariate Puiseux series and $f(x)$ is a univariate Puiseux series with no terms of non-negative degree.

See Also:

o)show UnivariatePuisseuxSeriesWithExponentialSingularity

UnivariatePuisseuxSeriesWithExponentialSingularity (UPXSSING)



See

⇒ “ExponentialOfUnivariatePuisseuxSeries” (EXPUPXS) 6 on page 739

⇒ “ExponentialExpansion” (EXPEXPAN) 6 on page 711

Exports:

0	1	associates?	binomThmExpt	characteristic
charthRoot	coefficient	coefficients	coerce	content
degree	dominantTerm	exquo	ground	ground?
hash	latex	leadingCoefficient	leadingMonomial	limitPlus
map	mapExponents	minimumDegree	monomial	monomial?
numberOfMonomials	one?	pomopo!	primitivePart	recip
reductum	retract	retractIfCan	sample	subtractIfCan
unit?	unitCanonical	unitNormal	zero?	?*?
?**?	?+?	?-?	-?	?=?
?^?	?~=?	?/?		

— domain UPXSSING UnivariatePuisseuxSeriesWithExponentialSingularity —

```

)abbrev domain UPXSSING UnivariatePuisseuxSeriesWithExponentialSingularity
++ Author: Clifton J. Williamson
++ Date Created: 4 August 1992
++ Date Last Updated: 27 August 1992
++ Description:
++ UnivariatePuisseuxSeriesWithExponentialSingularity is a domain used to
++ represent functions with essential singularities. Objects in this
++ domain are sums, where each term in the sum is a univariate Puisseux
++ series times the exponential of a univariate Puisseux series. Thus,
++ the elements of this domain are sums of expressions of the form
++ \spad{g(x) * exp(f(x))}, where g(x) is a univariate Puisseux series
++ and f(x) is a univariate Puisseux series with no terms of non-negative
++ degree.

```

```

UnivariatePuisseuxSeriesWithExponentialSingularity(R,FE,var,cen):_
  Exports == Implementation where

```

```

R      : Join(OrderedSet,RetractableTo Integer,_,
              LinearlyExplicitRingOver Integer,GcdDomain)
FE     : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,_,
              FunctionSpace R)
var    : Symbol
cen    : FE
B      ==> Boolean
I      ==> Integer
L      ==> List
RN     ==> Fraction Integer
UPXS   ==> UnivariatePuisseuxSeries(FE,var,cen)
EXPUPXS ==> ExponentialOfUnivariatePuisseuxSeries(FE,var,cen)
OFE    ==> OrderedCompletion FE
Result ==> Union(OFE,"failed")
PxRec  ==> Record(k: Fraction Integer,c:FE)
Term   ==> Record(%coef:UPXS,%expon:EXPUPXS,%expTerms:List PxRec)
-- the %expTerms field is used to record the list of the terms (a 'term'
-- records an exponent and a coefficient) in the exponent %expon
TypedTerm ==> Record(%term:Term,%type:String)
-- a term together with a String which tells whether it has an infinite,
-- zero, or unknown limit as var -> cen+
TRec   ==> Record(%zeroTerms: List Term,_,
                  %infiniteTerms: List Term,_,
                  %failedTerms: List Term,_,
                  %puisseuxSeries: UPXS)
SIGNEF ==> ElementaryFunctionSign(R,FE)

Exports ==> Join(FiniteAbelianMonoidRing(UPXS,EXPUPXS),IntegralDomain) with
  limitPlus : % -> Union(OFE,"failed")
  ++ limitPlus(f(var)) returns \spad{limit(var -> cen+,f(var))}.
  dominantTerm : % -> Union(TypedTerm,"failed")
  ++ dominantTerm(f(var)) returns the term that dominates the limiting
  ++ behavior of \spad{f(var)} as \spad{var -> cen+} together with a
  ++ \spadtype{String} which briefly describes that behavior. The
  ++ value of the \spadtype{String} will be \spad{"zero"} (resp.
  ++ \spad{"infinity"}) if the term tends to zero (resp. infinity)
  ++ exponentially and will \spad{"series"} if the term is a
  ++ Puiseux series.

Implementation ==> PolynomialRing(UPXS,EXPUPXS) add
  makeTerm : (UPXS,EXPUPXS) -> Term
  coeff : Term -> UPXS
  exponent : Term -> EXPUPXS
  exponentTerms : Term -> List PxRec
  setExponentTerms_! : (Term,List PxRec) -> List PxRec
  computeExponentTerms_! : Term -> List PxRec
  terms : % -> List Term
  sortAndDiscardTerms: List Term -> TRec
  termsWithExtremeLeadingCoef : (L Term,RN,I) -> Union(L Term,"failed")
  filterByOrder: (L Term,(RN,RN) -> B) -> Record(%list:L Term,%order:RN)

```



```

dominantTermOnList : (L Term,RN,I) -> Union(Term,"failed")
iDominantTerm : L Term -> Union(Record(%term:Term,%type:String),"failed")

retractIfCan f ==
  (numberOfMonomials f = 1) and (zero? degree f) => leadingCoefficient f
  "failed"

recip f ==
  numberOfMonomials f = 1 =>
    monomial(inv leadingCoefficient f,- degree f)
  "failed"

makeTerm(coef,expon) == [coef,expon,empty()]
coeff term == term.%coef
exponent term == term.%expon
exponentTerms term == term.%expTerms
setExponentTerms_!(term,list) == term.%expTerms := list
computeExponentTerms_! term ==
  setExponentTerms_!(term,entries complete terms exponent term)

terms f ==
  -- terms with a higher order singularity will appear closer to the
  -- beginning of the list because of the ordering in EXPPUPXS;
  -- no "exponent terms" are computed by this function
  zero? f => empty()
  concat(makeTerm(leadingCoefficient f,degree f),terms reductum f)

sortAndDiscardTerms termList ==
  -- 'termList' is the list of terms of some function f(var), ordered
  -- so that terms with a higher order singularity occur at the
  -- beginning of the list.
  -- This function returns lists of candidates for the "dominant
  -- term" in 'termList', i.e. the term which describes the
  -- asymptotic behavior of f(var) as var -> cen+.
  -- 'zeroTerms' will contain terms which tend to zero exponentially
  -- and contains only those terms with the lowest order singularity.
  -- 'zeroTerms' will be non-empty only when there are no terms of
  -- infinite or series type.
  -- 'infiniteTerms' will contain terms which tend to infinity
  -- exponentially and contains only those terms with the highest
  -- order singularity.
  -- 'failedTerms' will contain terms which have an exponential
  -- singularity, where we cannot say whether the limiting value
  -- is zero or infinity. Only terms with a higher order singularity
  -- than the terms on 'infiniteList' are included.
  -- 'pSeries' will be a Puiseux series representing a term without an
  -- exponential singularity. 'pSeries' will be non-zero only when no
  -- other terms are known to tend to infinity exponentially
  zeroTerms : List Term := empty()
  infiniteTerms : List Term := empty()

```

```

failedTerms : List Term := empty()
-- we keep track of whether or not we've found an infinite term
-- if so, 'infTermOrd' will be set to a negative value
infTermOrd : RN := 0
-- we keep track of whether or not we've found a zero term
-- if so, 'zeroTermOrd' will be set to a negative value
zeroTermOrd : RN := 0
ord : RN := 0; pSeries : UPXS := 0 -- dummy values
while not empty? termList repeat
  -- 'expon' is a Puiseux series
  expon := exponent(term := first termList)
  -- quit if there is an infinite term with a higher order singularity
  (ord := order(expon,0)) > infTermOrd => leave "infinite term dominates"
  -- if ord = 0, we've hit the end of the list
  (ord = 0) =>
    -- since we have a series term, don't bother with zero terms
    leave(pSeries := coeff(term); zeroTerms := empty())
  coef := coefficient(expon,ord)
  -- if we can't tell if the lowest order coefficient is positive or
  -- negative, we have a "failed term"
  (signum := sign(coef)$SIGNEF) case "failed" =>
    failedTerms := concat(term,failedTerms)
    termList := rest termList
  -- if the lowest order coefficient is positive, we have an
  -- "infinite term"
  (sig := signum :: Integer) = 1 =>
    infTermOrd := ord
    infiniteTerms := concat(term,infiniteTerms)
    -- since we have an infinite term, don't bother with zero terms
    zeroTerms := empty()
    termList := rest termList
  -- if the lowest order coefficient is negative, we have a
  -- "zero term" if there are no infinite terms and no failed
  -- terms, add the term to 'zeroTerms'
  if empty? infiniteTerms then
    zeroTerms :=
      ord = zeroTermOrd => concat(term,zeroTerms)
      zeroTermOrd := ord
      list term
    termList := rest termList
  -- reverse "failed terms" so that higher order singularities
  -- appear at the beginning of the list
  [zeroTerms,infiniteTerms,reverse_! failedTerms,pSeries]

termsWithExtremeLeadingCoef(termList,ord,signum) ==
  -- 'termList' consists of terms of the form [g(x),exp(f(x)),...];
  -- when 'signum' is +1 (resp. -1), this function filters 'termList'
  -- leaving only those terms such that coefficient(f(x),ord) is
  -- maximal (resp. minimal)
  while (coefficient(exponent first termList,ord) = 0) repeat

```

```

    termList := rest termList
empty? termList => error "UPXSING: can't happen"
coefExtreme := coefficient(exponent first termList,ord)
outList := list first termList; termList := rest termList
for term in termList repeat
    (coefDiff := coefficient(exponent term,ord) - coefExtreme) = 0 =>
        outList := concat(term,outList)
    (sig := sign(coefDiff)$SIGNEF) case "failed" => return "failed"
    (sig :: Integer) = signum => outList := list term
outList

filterByOrder(termList,predicate) ==
-- 'termList' consists of terms of the form [g(x),exp(f(x)),expTerms],
-- where 'expTerms' is a list containing some of the terms in the
-- series f(x).
-- The function filters 'termList' and, when 'predicate' is < (resp. >),
-- leaves only those terms with the lowest (resp. highest) order term
-- in 'expTerms'
while empty? exponentTerms first termList repeat
    termList := rest termList
    empty? termList => error "UPXSING: can't happen"
ordExtreme := (first exponentTerms first termList).k
outList := list first termList
for term in rest termList repeat
    not empty? exponentTerms term =>
        (ord := (first exponentTerms term).k) = ordExtreme =>
            outList := concat(term,outList)
        predicate(ord,ordExtreme) =>
            ordExtreme := ord
            outList := list term
-- advance pointers on "exponent terms" on terms on 'outList'
for term in outList repeat
    setExponentTerms_!(term,rest exponentTerms term)
[outList,ordExtreme]

dominantTermOnList(termList,ord0,signum) ==
-- finds dominant term on 'termList'
-- it is known that "exponent terms" of order < 'ord0' are
-- the same for all terms on 'termList'
newList := termsWithExtremeLeadingCoef(termList,ord0,signum)
newList case "failed" => "failed"
termList := newList :: List Term
empty? rest termList => first termList
filtered :=
    signum = 1 => filterByOrder(termList,(x,y) +-> x < y)
    filterByOrder(termList,(x,y) +-> x > y)
termList := filtered.%list
empty? rest termList => first termList
dominantTermOnList(termList,filtered.%order,signum)

```

```

iDominantTerm termList ==
  termRecord := sortAndDiscardTerms termList
  zeroTerms := termRecord.%zeroTerms
  infiniteTerms := termRecord.%infiniteTerms
  failedTerms := termRecord.%failedTerms
  pSeries := termRecord.%puiseuxSeries
  -- in future versions, we will deal with "failed terms"
  -- at present, if any occur, we cannot determine the limit
  not empty? failedTerms => "failed"
  not zero? pSeries => [makeTerm(pSeries,0),"series"]
  not empty? infiniteTerms =>
    empty? rest infiniteTerms => [first infiniteTerms,"infinity"]
    for term in infiniteTerms repeat computeExponentTerms_! term
    ord0 := order exponent first infiniteTerms
    (dTerm := dominantTermOnList(infiniteTerms,ord0,1)) case "failed" =>
      return "failed"
    [dTerm :: Term,"infinity"]
  empty? rest zeroTerms => [first zeroTerms,"zero"]
  for term in zeroTerms repeat computeExponentTerms_! term
  ord0 := order exponent first zeroTerms
  (dTerm := dominantTermOnList(zeroTerms,ord0,-1)) case "failed" =>
    return "failed"
  [dTerm :: Term,"zero"]

dominantTerm f == iDominantTerm terms f

limitPlus f ==
  -- list the terms occurring in 'f'; if there are none, then f = 0
  empty?(termList := terms f) => 0
  -- compute dominant term
  (tInfo := iDominantTerm termList) case "failed" => "failed"
  termInfo := tInfo :: Record(%term:Term,%type:String)
  domTerm := termInfo.%term
  (type := termInfo.%type) = "series" =>
    -- find limit of series term
    (ord := order(pSeries := coeff domTerm,1)) > 0 => 0
    coef := coefficient(pSeries,ord)
    member?(var,variables coef) => "failed"
    ord = 0 => coef :: OFE
    -- in the case of an infinite limit, we need to know the sign
    -- of the first non-zero coefficient
    (signum := sign(coef)$SIGNEF) case "failed" => "failed"
    (signum :: Integer) = 1 => plusInfinity()
    minusInfinity()
  type = "zero" => 0
  -- examine lowest order coefficient in series part of 'domTerm'
  ord := order(pSeries := coeff domTerm)
  coef := coefficient(pSeries,ord)
  member?(var,variables coef) => "failed"
  (signum := sign(coef)$SIGNEF) case "failed" => "failed"

```

```
(signum :: Integer) = 1 => plusInfinity()
minusInfinity()
```

— UPXSSING.dotabb —

```
"UPXSSING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=UPXSSING"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"UPXSSING" -> "ACF"
"UPXSSING" -> "FS"
```

domain OREUP UnivariateSkewPolynomial

— UnivariateSkewPolynomial.input —

```
)set break resume
)sys rm -f UnivariateSkewPolynomial.output
)spool UnivariateSkewPolynomial.output
)set message test on
)set message auto off
)set message type off
)clear all

--S 1 of 33
F:=EXPR(FRAC(INT))
--R
--R
--R (1) Expression(Fraction(Integer))
--E 1

--S 2 of 33
Dx:F->F:=f+-->D(f,['x])
--R
--R
--R (2) theMap(Closure)
--E 2

--S 3 of 33
DO:=OREUP('d,F,1,Dx);
--R
```

--R
--E 3

--S 4 of 33
u:D0:=(operator 'u)(x)
--R
--R
--R (4) $u(x)$
--E 4

--S 5 of 33
d:D0:='d
--R
--R
--R (5) d
--E 5

--S 6 of 33
a:D0:=u^3*d^3+u^2*d^2+u*d+1
--R
--R
--R (6) $u(x)^3 d^3 + u(x)^2 d^2 + u(x) d + 1$
--E 6

--S 7 of 33
b:D0:=(u+1)*d^2+2*d
--R
--R
--R (7) $(u(x) + 1) d^2 + 2d$
--E 7

--S 8 of 33
r:=rightDivide(a,b)
--R
--R
--R (8)
--R (3) , (3) 2
--R $- u(x)^3 u(x) - u(x)^3 + u(x)^2$
--R $u(x)$
--R [quotient= $\frac{u(x)^3}{u(x) + 1} d + \frac{- u(x)^3 u(x) - u(x)^3 + u(x)^2}{u(x)^2 + 2u(x) + 1}$,
--R (3) , (3)
--R $2u(x)^3 u(x) + 3u(x)^3 + u(x)$
--R remainder= $\frac{2u(x)^3 u(x) + 3u(x)^3 + u(x)}{u(x)^2 + 2u(x) + 1} d + 1]$
--R
--R

--E 8

--S 9 of 33

r.quotient

--R

--R

--R
$$^3, \quad ^3 \quad ^2$$

--R
$$^3 \quad - u(x) u^3(x) - u^3(x) + u^2(x)$$

--R
$$u(x)$$

--R (9)
$$\frac{u(x)}{u(x) + 1} d + \frac{u^2(x) + 2u(x) + 1}{u(x) + 1}$$

--R
$$u(x) + 1 \quad u^2(x) + 2u(x) + 1$$

--E 9

--S 10 of 33

r.remainder

--R

--R

--R
$$^3, \quad ^3$$

--R
$$2u(x) u^3(x) + 3u^3(x) + u(x)$$

--R

--R (10)
$$\frac{2u(x) u^3(x) + 3u^3(x) + u(x)}{u(x) + 1} d + 1$$

--R
$$u^2(x) + 2u(x) + 1$$

--E 10

)clear all

--S 11 of 33

R:=UP('t,INT)

--R

--R

--R (1) UnivariatePolynomial(t,Integer)

--E 11

--S 12 of 33

W:=OREUP('x,R,1,D)

--R

--R

--R (2)

--R UnivariateSkewPolynomial(x,UnivariatePolynomial(t,Integer),R -> R,theMap(DIFR

--I ING-;D;2S;1,0))

--E 12

--S 13 of 33

t:W:='t

--R

--R

--R (3) t

--E 13

--S 14 of 33

x:W:='x

--R

--R

--R (4) x

--E 14

--S 15 of 33

a:W:=(t-1)*x^4+(t^3+3*t+1)*x^2+2*t*x+t^3

--R

--R

--R (5) $(t^4 - 1)x^4 + (t^3 + 3t + 1)x^2 + 2tx + t^3$

--E 15

--S 16 of 33

b:W:=(6*t^4+2*t^2)*x^3+3*t^2*x^2

--R

--R

--R (6) $(6t^4 + 2t^2)x^3 + 3t^2x^2$

--E 16

--S 17 of 33

a*b

--R

--R

--R (7)

--R $(6t^5 - 6t^4 + 2t^3 - 2t^2)x^7 + (96t^4 - 93t^3 + 13t^2 - 16t)x^6$

--R +

--R $(6t^7 + 20t^5 + 6t^4 + 438t^3 - 406t^2 - 24t)x^5$

--R +

--R $(48t^6 + 15t^5 + 152t^4 + 61t^3 + 603t^2 - 532t - 36)x^4$

--R +

--R $(6t^7 + 74t^5 + 60t^4 + 226t^3 + 116t^2 + 168t - 140)x^3$

--R +

--R $(3t^5 + 6t^3 + 12t^2 + 18t + 6)x^2$

--E 17

--S 18 of 33

a^3

--R

--R

--R (8)


```

--R      3      2      12      5      4      3      2      10
--R      (t  - 3t  + 3t - 1)x  + (3t  - 6t  + 12t  - 15t  + 3t + 3)x
--R      +
--R      3      2      9      7      6      5      4      3      2      8
--R      (6t  - 12t  + 6t)x  + (3t  - 3t  + 21t  - 18t  + 24t  - 9t  - 15t - 3)x
--R      +
--R      5      4      3      2      7
--R      (12t  - 12t  + 36t  - 24t  - 12t)x
--R      +
--R      9      7      6      5      4      3      2      6
--R      (t  + 15t  - 3t  + 45t  + 6t  + 36t  + 15t  + 9t + 1)x
--R      +
--R      7      5      3      2      5
--R      (6t  + 48t  + 54t  + 36t  + 6t)x
--R      +
--R      9      7      6      5      4      3      2      4
--R      (3t  + 21t  + 3t  + 39t  + 18t  + 39t  + 12t )x
--R      +
--R      7      5      4      3      3      9      7      6      5      2      7      9
--R      (12t  + 36t  + 12t  + 8t )x  + (3t  + 9t  + 3t  + 12t )x  + 6t x + t
--E 18

```

```

)clear all

```

```

--S 19 of 33
S:EXPR(INT)->EXPR(INT):=e+-->eval(e,[n],[n+1])
--R
--R
--R      (1)  theMap(Closure)
--E 19

```

```

--S 20 of 33
DF:EXPR(INT)->EXPR(INT):=e+-->eval(e,[n],[n+1])-e
--R
--R
--R      (2)  theMap(Closure)
--E 20

```

```

--S 21 of 33
DO:=OREUP('D,EXPR(INT),morphism S,DF);
--R
--R
--E 21

```

```

--S 22 of 33
u:=(operator 'u)[n]
--R
--R
--R      (4)  u(n)
--E 22

```

```

--S 23 of 33
L:=D0:='D+u
--R
--R
--R (5)  $D + u(n)$ 
--E 23

--S 24 of 33
L^2
--R
--R
--R (6)  $D^2 + 2u(n)D + u(n)^2$ 
--E 24

)clear all

--S 25 of 33
)set expose add constructor SquareMatrix
--R
--R SquareMatrix is now explicitly exposed in frame frame0
--E 25

--S 26 of 33
R:=SQMATRIX(2,INT)
--R
--R
--R (1) SquareMatrix(2,Integer)
--E 26

--S 27 of 33
y:=matrix [[1,1],[0,1]]
--R
--R
--R (2)  $\begin{pmatrix} +1 & 1+ \\ | & | \\ +0 & 1+ \end{pmatrix}$ 
--E 27

--S 28 of 33
delta:R->R:=r+>y*r-r*y
--R
--R
--R (3) theMap(Closure)
--E 28

--S 29 of 33
S:=OREUP('x,R,1,delta);
--R

```

```
--R
--E 29
```

```
--S 30 of 33
```

```
x:S:='x
```

```
--R
```

```
--R
```

```
--R (5) x
```

```
--E 30
```

```
--S 31 of 33
```

```
a:S:=matrix [[2,3],[1,1]]
```

```
--R
```

```
--R
```

```
--R +2 3+
```

```
--R (6) | |
```

```
--R +1 1+
```

```
--E 31
```

```
--S 32 of 33
```

```
x^2*a
```

```
--R
```

```
--R
```

```
--R +2 3+ 2 +2 - 2+ +0 - 2+
```

```
--R (7) | |x + | |x + | |
```

```
--R +1 1+ +0 - 2+ +0 0 +
```

```
--E 32
```

```
--S 33 of 33
```

```
)show UnivariateSkewPolynomial
```

```
--R
```

```
--R UnivariateSkewPolynomial(x: Symbol,R: Ring,sigma: Automorphism(R),delta: (R -> R)) is a dom
```

```
--R Abbreviation for UnivariateSkewPolynomial is OREUP
```

```
--R This constructor is not exposed in this frame.
```

```
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for OREUP
```

```
--R
```

```
--R----- Operations -----
```

```
--R ?? : (R,%) -> %
```

```
?? : (%,R) -> %
```

```
--R ?? : (%,%) -> %
```

```
?? : (Integer,%) -> %
```

```
--R ?? : (PositiveInteger,%) -> %
```

```
??? : (%,PositiveInteger) -> %
```

```
--R +? : (%,%) -> %
```

```
?-? : (%,%) -> %
```

```
--R -? : % -> %
```

```
?=? : (%,%) -> Boolean
```

```
--R 1 : () -> %
```

```
0 : () -> %
```

```
--R ^? : (%,PositiveInteger) -> %
```

```
apply : (%,R,R) -> R
```

```
--R coefficients : % -> List(R)
```

```
coerce : Variable(x) -> %
```

```
--R coerce : R -> %
```

```
coerce : Integer -> %
```

```
--R coerce : % -> OutputForm
```

```
degree : % -> NonNegativeInteger
```

```
--R hash : % -> SingleInteger
```

```
latex : % -> String
```

```
--R leadingCoefficient : % -> R
```

```
one? : % -> Boolean
```

```
--R recip : % -> Union(%, "failed")
```

```
reductum : % -> %
```

```

--R retract : % -> R                                sample : () -> %
--R zero? : % -> Boolean                            ?~=? : (%,% ) -> Boolean
--R ??? : (NonNegativeInteger,% ) -> %
--R ??? : (% ,NonNegativeInteger) -> %
--R ?? : (% ,NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R coefficient : (% ,NonNegativeInteger) -> R
--R coerce : Fraction(Integer) -> % if R has RETRACT(FRAC(INT))
--R content : % -> R if R has GCDDOM
--R exquo : (% ,R) -> Union(% ,"failed") if R has INTDOM
--R leftDivide : (% ,%) -> Record(quotient: %,remainder: %) if R has FIELD
--R leftExactQuotient : (% ,%) -> Union(% ,"failed") if R has FIELD
--R leftExtendedGcd : (% ,%) -> Record(coef1: %,coef2: %,generator: %) if R has FIELD
--R leftGcd : (% ,%) -> % if R has FIELD
--R leftLcm : (% ,%) -> % if R has FIELD
--R leftQuotient : (% ,%) -> % if R has FIELD
--R leftRemainder : (% ,%) -> % if R has FIELD
--R minimumDegree : % -> NonNegativeInteger
--R monicLeftDivide : (% ,%) -> Record(quotient: %,remainder: %) if R has INTDOM
--R monicRightDivide : (% ,%) -> Record(quotient: %,remainder: %) if R has INTDOM
--R monomial : (R,NonNegativeInteger) -> %
--R primitivePart : % -> % if R has GCDDOM
--R retract : % -> Fraction(Integer) if R has RETRACT(FRAC(INT))
--R retract : % -> Integer if R has RETRACT(INT)
--R retractIfCan : % -> Union(R,"failed")
--R retractIfCan : % -> Union(Fraction(Integer),"failed") if R has RETRACT(FRAC(INT))
--R retractIfCan : % -> Union(Integer,"failed") if R has RETRACT(INT)
--R rightDivide : (% ,%) -> Record(quotient: %,remainder: %) if R has FIELD
--R rightExactQuotient : (% ,%) -> Union(% ,"failed") if R has FIELD
--R rightExtendedGcd : (% ,%) -> Record(coef1: %,coef2: %,generator: %) if R has FIELD
--R rightGcd : (% ,%) -> % if R has FIELD
--R rightLcm : (% ,%) -> % if R has FIELD
--R rightQuotient : (% ,%) -> % if R has FIELD
--R rightRemainder : (% ,%) -> % if R has FIELD
--R subtractIfCan : (% ,%) -> Union(% ,"failed")
--R
--E 33
)set expose drop constructor SquareMatrix

)spool
)lisp (bye)

```

— UnivariateSkewPolynomial.help —

=====

UnivariateSkewPolynomial examples

=====

Skew or Ore polynomial rings provide a unified framework to compute with differential and difference equations.

In the following, let A be an integral domain, equipped with two endomorphisms σ and δ where:

$\sigma: A \rightarrow A$ is an injective ring endomorphism
 $\delta: A \rightarrow A$, the pseudo-derivation with respect to σ ,
 is an additive endomorphism with

$$\delta(ab) = \sigma(a)\delta(b) + \delta(a)b$$

for all a, b in A

Note that in the domains and categories below, these properties are not checked.

The skew polynomial ring $[\Delta; \sigma, \delta]$ is the ring of polynomials in Δ with coefficients in A , with the usual addition, while the product is given by

$$\Delta a = \sigma(a)\Delta + \delta(a) \text{ for } a \text{ in } A$$

The two most important examples of skew polynomial rings are:

$K(x)[D, 1, \delta]$, where 1 is the identity on K and δ is the usual derivative, is the ring of differential polynomials

$K[E, n, \text{mapsto } n+1, 0]$ is the ring of linear recurrence operators with polynomial coefficients

For example,

The `UnivariateSkewPolynomialCategory` (OREPCAT) provides a unified framework for polynomial rings in a non-central indeterminate over some coefficient ring R . The commutation relations between the indeterminate x and the coefficient t is given by

$$x r = \sigma(r) x + \delta(r)$$

where σ is a ring endomorphism of R
 and δ is a σ -derivation of R
 which is an additive map from R to R such that

$$\backslash\delta(rs) = \backslash\sigma(r) \backslash\delta(s) + \backslash\delta(r) s$$

In case $\backslash\sigma$ is the identity map on R , a $\backslash\sigma$ -derivation of R is just called a derivation. Here are some examples

We start with a linear ordinary differential operator. First, we define the coefficient ring to be expressions in one variable x with fractional coefficients:

```
F:=EXPR(FRAC(INT))
```

Define Dx to be a derivative d/dx :

```
Dx:F->F:=f+-->D(f,['x])
```

Define a skew polynomial ring over F with identity endomorphism as $\backslash\sigma$ and derivation d/dx as $\backslash\delta$:

```
D0:=OREUP('d,F,1,Dx)
```

```
u:D0:=(operator 'u)(x)
```

```
d:D0:='d
```

```
a:D0:=u^3*d^3+u^2*d^2+u*d+1
```

$$u(x)^3 d^3 + u(x)^2 d^2 + u(x) d + 1$$

```
b:D0:=(u+1)*d^2+2*d
```

$$(u(x) + 1) d^2 + 2d$$

```
r:=rightDivide(a,b)
```

$$\begin{aligned} \text{[quotient]} &= \frac{u(x)^3}{u(x) + 1} d + \frac{-u(x)^3 u(x) - u(x)^3 + u(x)^2}{u(x)^2 + 2u(x) + 1}, \\ \text{remainder} &= \frac{2u(x)^3 u(x) + 3u(x)^3 + u(x)}{u(x)^2 + 2u(x) + 1} d + 1 \end{aligned}$$

r.quotient

$$\frac{u(x)^3 - u(x)u'(x)^3 - u(x)^3 + u(x)^2}{u(x)^3 + 1} d + \frac{u(x)^2}{u(x)^3 + 2u(x) + 1}$$

r.remainer

$$\frac{2u(x)^3 u'(x)^3 + 3u(x)^3 + u(x)^3}{u(x)^2 + 2u(x) + 1} d + 1$$

)clear all

As a second example, we consider the so-called Weyl algebra.

Define the coefficient ring to be an ordinary polynomial over integers in one variable t

R:=UP('t,INT)

Define a skew polynomial ring over R with identity map as \sigma and derivation d/dt as \delta. The resulting algebra is then called a Weyl algebra. This is a simple ring over a division ring that is non-commutative, similar to the ring of matrices.

W:=OREUP('x,R,1,D)

t:W:='t

x:W:='x

Let

a:W:=(t-1)*x^4+(t^3+3*t+1)*x^2+2*t*x+t^3

$$(t-1)x^4 + (t^3 + 3t + 1)x^2 + 2tx + t^3$$

b:W:=(6*t^4+2*t^2)*x^3+3*t^2*x^2

$$(6t^4 + 2t^2)x^3 + 3t^2x^2$$

Then

a*b

$$\begin{aligned} & (6t^5 - 6t^4 + 2t^3 - 2t^2)x^7 + (96t^4 - 93t^3 + 13t^2 - 16t)x^6 \\ & + (6t^7 + 20t^5 + 6t^4 + 438t^3 - 406t^2 - 24t)x^5 \\ & + (48t^6 + 15t^5 + 152t^4 + 61t^3 + 603t^2 - 532t - 36)x^4 \\ & + (6t^7 + 74t^5 + 60t^4 + 226t^3 + 116t^2 + 168t - 140)x^3 \\ & + (3t^5 + 6t^3 + 12t^2 + 18t + 6)x^2 \end{aligned}$$

a^3

$$\begin{aligned} & (t^3 - 3t^2 + 3t - 1)x^{12} + (3t^5 - 6t^4 + 12t^3 - 15t^2 + 3t + 3)x^{10} \\ & + (6t^3 - 12t^2 + 6t)x^9 + (3t^7 - 3t^6 + 21t^5 - 18t^4 + 24t^3 - 9t^2 - 15t - 3)x^8 \\ & + (12t^5 - 12t^4 + 36t^3 - 24t^2 - 12t)x^7 \\ & + (t^9 + 15t^7 - 3t^6 + 45t^5 + 6t^4 + 36t^3 + 15t^2 + 9t + 1)x^6 \\ & + (6t^7 + 48t^5 + 54t^3 + 36t^2 + 6t)x^5 \\ & + (3t^9 + 21t^7 + 3t^6 + 39t^5 + 18t^4 + 39t^3 + 12t^2)x^4 \\ & + (12t^7 + 36t^5 + 12t^4 + 8t^3)x^3 + (3t^9 + 9t^7 + 3t^6 + 12t^5)x^2 + 6t^2x + t \end{aligned}$$

)clear all

As a third example, we construct a difference operator algebra over

the ring of $\text{EXPR}(\text{INT})$ by using an automorphism S defined by a "shift" operation $S:\text{EXPR}(\text{INT}) \rightarrow \text{EXPR}(\text{INT})$

$$s(e)(n) = e(n+1)$$

and an S -derivation defined by $DF:\text{EXPR}(\text{INT}) \rightarrow \text{EXPR}(\text{INT})$ as

$$DF(e)(n) = e(n+1) - e(n)$$

Define S to be a "shift" operator, which acts on expressions with the discrete variable n :

$$S:\text{EXPR}(\text{INT}) \rightarrow \text{EXPR}(\text{INT}) := e \mapsto \text{eval}(e, [n], [n+1])$$

Define DF to be a "difference" operator, which acts on expressions with a discrete variable n :

$$DF:\text{EXPR}(\text{INT}) \rightarrow \text{EXPR}(\text{INT}) := e \mapsto \text{eval}(e, [n], [n+1]) - e$$

Then define the difference operator algebra $D0$:

$$D0 := \text{OREUP}('D, \text{EXPR}(\text{INT}), \text{morphism } S, DF)$$

$$u := (\text{operator } 'u)[n]$$

$$L:D0 := 'D + u$$

$$D + u(n)$$

$$L^2$$

$$D^2 + 2u(n)D + u(n)^2$$

)clear all

As a fourth example, we construct a skew polynomial ring by using an inner derivation δ induced by a fixed y in R :

$$\delta(r) = yr - ry$$

First we should expose the constructor `SquareMatrix` so it is visible in the interpreter:

)set expose add constructor SquareMatrix

Define R to be the square matrix with integer entries:

```

R:=SQMATRIX(2,INT)

y:R:=matrix [[1,1],[0,1]]
      +1  1+
      |   |
      +0  1+

Define the inner derivative \delta:

delta:R->R:=r+>y*r-r*y

Define S to be a skew polynomial determined by \sigma = 1
and \delta as an inner derivative:

S:=OREUP('x,R,1,delta)

x:S:='x

a:S:=matrix [[2,3],[1,1]]
      +2  3+
      |   |
      +1  1+

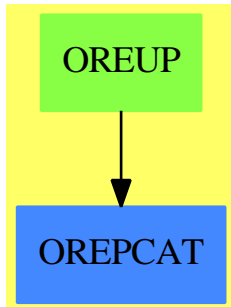
x^2*a
      +2  3+ 2  +2  - 2+  +0  - 2+
      |   |x  + |   |x  + |   |
      +1  1+  +0  - 2+  +0  0  +

```

See Also:

- o)show UnivariateSkewPolynomial
- o)show UnivariateSkewPolynomialCategory
- o)show SquareMatrix

UnivariateSkewPolynomial (OREUP)



See

⇒ “Automorphism” (AUTOMOR) [2](#) on page [245](#)

⇒ “SparseUnivariateSkewPolynomial” (ORESUP) [20](#) on page [2492](#)

Exports:

0	1	apply
characteristic	coefficient	coefficients
coerce	content	degree
exquo	hash	latex
leadingCoefficient	leftDivide	leftExactQuotient
leftExtendedGcd	leftGcd	leftLcm
leftQuotient	leftRemainder	minimumDegree
monicLeftDivide	monicRightDivide	monomial
one?	primitivePart	recip
reductum	retract	retractIfCan
rightDivide	rightExactQuotient	rightExtendedGcd
rightGcd	rightLcm	rightQuotient
rightRemainder	sample	subtractIfCan
zero?	?*?	?**?
?+?	?-?	-?
?=?	?^?	?^=?

— domain OREUP UnivariateSkewPolynomial —

```
)abbrev domain OREUP UnivariateSkewPolynomial
++ Author: Manuel Bronstein
++ Date Created: 19 October 1993
++ Date Last Updated: 1 February 1994
++ Description:
++ This is the domain of univariate skew polynomials over an Ore
++ coefficient field in a named variable.
++ The multiplication is given by \spad{x a = \sigma(a) x + \delta a}.

UnivariateSkewPolynomial(x:Symbol,R:Ring,sigma:Automorphism R,delta: R -> R):
  UnivariateSkewPolynomialCategory R with
```

```

coerce: Variable x -> %
  ++ coerce(x) returns x as a skew-polynomial.
== SparseUnivariateSkewPolynomial(R, sigma, delta) add
Rep := SparseUnivariateSkewPolynomial(R, sigma, delta)
coerce(v:Variable(x)):% == monomial(1, 1)
coerce(p:%):OutputForm == outputForm(p, outputForm x)$Rep

```

— OREUP.dotabb —

```

"OREUP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=OREUP"]
"OREPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OREPCAT"]
"OREUP" -> "OREPCAT"

```

domain UTS UnivariateTaylorSeries

— UnivariateTaylorSeries.input —

```

)set break resume
)sys rm -f UnivariateTaylorSeries.output
)spool UnivariateTaylorSeries.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariateTaylorSeries
--R
--R UnivariateTaylorSeries(Coef: Ring,var: Symbol,cen: Coef) is a domain constructor
--R Abbreviation for UnivariateTaylorSeries is UTS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for UTS
--R
--R----- Operations -----
--R ??? : (Coef,%) -> %          ??? : (%,Coef) -> %
--R ??? : (%,%) -> %           ??? : (Integer,%) -> %
--R ??? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %           ?-? : (%,%) -> %
--R -? : % -> %                ?? : (%,%) -> Boolean
--R 1 : () -> %                0 : () -> %
--R ?? : (%,PositiveInteger) -> %   center : % -> Coef

```

```

--R coefficients : % -> Stream(Coef)
--R coerce : Integer -> %
--R complete : % -> %
--R evenlambert : % -> %
--R lagrange : % -> %
--R latex : % -> String
--R leadingMonomial : % -> %
--R monomial? : % -> Boolean
--R one? : % -> Boolean
--R pole? : % -> Boolean
--R recip : % -> Union(%, "failed")
--R revert : % -> %
--R series : Stream(Coef) -> %
--R zero? : % -> Boolean
--R ?? : (%, Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (Fraction(Integer), %) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (NonNegativeInteger, %) -> %
--R ??? : (%, Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (%, %) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (%, Coef) -> % if Coef has FIELD
--R ??? : (%, NonNegativeInteger) -> %
--R ?/? : (%, Coef) -> % if Coef has FIELD
--R D : % -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef
--R D : (%, NonNegativeInteger) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef
--R D : (%, Symbol) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and Coef has PDRING(SYM
--R D : (%, List(Symbol)) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and Coef has PDRI
--R D : (%, Symbol, NonNegativeInteger) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and
--R D : (%, List(Symbol), List(NonNegativeInteger)) -> % if Coef has *: (NonNegativeInteger, Coef)
--R ?? : (%, NonNegativeInteger) -> %
--R acos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acoth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R approximate : (%, NonNegativeInteger) -> Coef if Coef has **: (Coef, NonNegativeInteger) -> Co
--R asec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R associates? : (%, %) -> Boolean if Coef has INTDOM
--R atan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R atanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(%, "failed") if Coef has CHARNZ
--R coefficient : (%, NonNegativeInteger) -> Coef
--R coerce : UnivariatePolynomial(var, Coef) -> %
--R coerce : Coef -> % if Coef has COMRING
--R coerce : % -> % if Coef has INTDOM
--R coerce : Fraction(Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R coerce : Variable(var) -> %
--R coerce : % -> OutputForm
--R degree : % -> NonNegativeInteger
--R hash : % -> SingleInteger
--R lambert : % -> %
--R leadingCoefficient : % -> Coef
--R map : ((Coef -> Coef), %) -> %
--R oddlambert : % -> %
--R order : % -> NonNegativeInteger
--R quoByVar : % -> %
--R reductum : % -> %
--R sample : () -> %
--R variable : % -> Symbol
--R ~=? : (%, %) -> Boolean

```

```

--R cos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R coth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R differentiate : (% , Variable(var)) -> %
--R differentiate : % -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef
--R differentiate : (% , NonNegativeInteger) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef
--R differentiate : (% , Symbol) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and Coef has PDRING(SY
--R differentiate : (% , List(Symbol)) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and Coef has PDR
--R differentiate : (% , Symbol, NonNegativeInteger) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and
--R differentiate : (% , List(Symbol), List(NonNegativeInteger)) -> % if Coef has *: (NonNegativeInteger, Coef)
--R ?.? : (% , %) -> % if NonNegativeInteger has SGROUP
--R ?.? : (% , NonNegativeInteger) -> Coef
--R eval : (% , Coef) -> Stream(Coef) if Coef has **: (Coef, NonNegativeInteger) -> Coef
--R exp : % -> % if Coef has ALGEBRA(FRAC(INT))
--R exquo : (% , %) -> Union(% , "failed") if Coef has INTDOM
--R extend : (% , NonNegativeInteger) -> %
--R generalLambert : (% , Integer, Integer) -> %
--R integrate : (% , Variable(var)) -> % if Coef has ALGEBRA(FRAC(INT))
--R integrate : (% , Symbol) -> % if Coef has integrate: (Coef, Symbol) -> Coef and Coef has variables: Coef -
--R integrate : % -> % if Coef has ALGEBRA(FRAC(INT))
--R invmultisect : (Integer, Integer, %) -> %
--R log : % -> % if Coef has ALGEBRA(FRAC(INT))
--R monomial : (% , List(SingletonAsOrderedSet), List(NonNegativeInteger)) -> %
--R monomial : (% , SingletonAsOrderedSet, NonNegativeInteger) -> %
--R monomial : (Coef, NonNegativeInteger) -> %
--R multiplyCoefficients : ((Integer -> Coef), %) -> %
--R multiplyExponents : (% , PositiveInteger) -> %
--R multisect : (Integer, Integer, %) -> %
--R nthRoot : (% , Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R order : (% , NonNegativeInteger) -> NonNegativeInteger
--R pi : () -> % if Coef has ALGEBRA(FRAC(INT))
--R polynomial : (% , NonNegativeInteger, NonNegativeInteger) -> Polynomial(Coef)
--R polynomial : (% , NonNegativeInteger) -> Polynomial(Coef)
--R sec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R series : Stream(Record(k: NonNegativeInteger, c: Coef)) -> %
--R sin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sqrt : % -> % if Coef has ALGEBRA(FRAC(INT))
--R subtractIfCan : (% , %) -> Union(% , "failed")
--R tan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R tanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R terms : % -> Stream(Record(k: NonNegativeInteger, c: Coef))
--R truncate : (% , NonNegativeInteger, NonNegativeInteger) -> %
--R truncate : (% , NonNegativeInteger) -> %
--R unit? : % -> Boolean if Coef has INTDOM
--R unitCanonical : % -> % if Coef has INTDOM

```

```

--R unitNormal : % -> Record(unit: %,canonical: %,associate: %) if Coef has INTDOM
--R univariatePolynomial : (% ,NonNegativeInteger) -> UnivariatePolynomial(var,Coef)
--R variables : % -> List(SingletonAsOrderedSet)
--R
--E 1

)spool
)lisp (bye)

```

— UnivariateTaylorSeries.help —

=====

UnivariateTaylorSeries examples

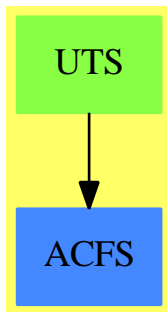
=====

Dense Taylor series in one variable. UnivariateTaylorSeries is a domain representing Taylor series in one variable with coefficients in an arbitrary ring. The parameters of the type specify the coefficient ring, the power series variable, and the center of the power series expansion. For example, UnivariateTaylorSeries(Integer,x,3) represents Taylor series in $(x - 3)$ with Integer coefficients.

See Also:

o)show UnivariateTaylorSeries

UnivariateTaylorSeries (UTS)



See

⇒ “InnerTaylorSeries” (ITAYLOR) [10](#) on page [1338](#)

Exports:

0	1	acos	acosh
acot	acoth	acsc	acsch
approximate	asec	asech	asin
asinh	associates?	atan	atanh
center	characteristic	charthRoot	coefficient
coefficients	coerce	complete	cos
cosh	cot	coth	csc
csch	D	degree	differentiate
eval	evenlambert	exp	exquo
extend	generalLambert	hash	integrate
invmultisect	lagrange	lambert	latex
leadingCoefficient	leadingMonomial	log	map
monomial	monomial?	multiplyCoefficients	multiplyExponents
multisect	nthRoot	oddlambert	one?
order	pi	pole?	polynomial
quoByVar	recip	reductum	revert
sample	sec	sech	series
sin	sinh	sqrt	subtractIfCan
tan	tanh	terms	truncate
unit?	unitCanonical	unitNormal	univariatePolynomial
variable	variables	zero?	?*?
?**?	?+?	?-?	-?
?=?	?^?	?~=?	?..?

— domain UTS UnivariateTaylorSeries —

```
)abbrev domain UTS UnivariateTaylorSeries
++ Author: Clifton J. Williamson
++ Date Created: 21 December 1989
++ Date Last Updated: 21 September 1993
++ Description:
++ Dense Taylor series in one variable
++ \spadtype{UnivariateTaylorSeries} is a domain representing Taylor
++ series in
++ one variable with coefficients in an arbitrary ring. The parameters
++ of the type specify the coefficient ring, the power series variable,
++ and the center of the power series expansion. For example,
++ \spadtype{UnivariateTaylorSeries}(Integer,x,3) represents
++ Taylor series in
++ \spad{(x - 3)} with \spadtype{Integer} coefficients.
```

```
UnivariateTaylorSeries(Coef,var,cen): Exports == Implementation where
  Coef : Ring
  var : Symbol
  cen : Coef
  I ==> Integer
  NNI ==> NonNegativeInteger
```



```

P      ==> Polynomial Coef
RN      ==> Fraction Integer
ST      ==> Stream
STT     ==> StreamTaylorSeriesOperations Coef
TERM    ==> Record(k:NNI,c:Coef)
UP      ==> UnivariatePolynomial(var,Coef)
Exports ==> UnivariateTaylorSeriesCategory(Coef) with
  coerce: UP -> %
    ++\spad{coerce(p)} converts a univariate polynomial p in the variable
    ++\spad{var} to a univariate Taylor series in \spad{var}.
  univariatePolynomial: (%,NNI) -> UP
    ++\spad{univariatePolynomial(f,k)} returns a univariate polynomial
    ++ consisting of the sum of all terms of f of degree \spad{<= k}.
  coerce: Variable(var) -> %
    ++\spad{coerce(var)} converts the series variable \spad{var} into a
    ++ Taylor series.
  differentiate: (%,Variable(var)) -> %
    ++ \spad{differentiate(f(x),x)} computes the derivative of
    ++ \spad{f(x)} with respect to \spad{x}.
  lagrange: % -> %
    ++\spad{lagrange(g(x))} produces the Taylor series for \spad{f(x)}
    ++ where \spad{f(x)} is implicitly defined as \spad{f(x) = x*g(f(x))}.
  lambert: % -> %
    ++\spad{lambert(f(x))} returns \spad{f(x) + f(x^2) + f(x^3) + ...}.
    ++ This function is used for computing infinite products.
    ++ \spad{f(x)} should have zero constant coefficient.
    ++ If \spad{f(x)} is a Taylor series with constant term 1, then
    ++ \spad{product(n = 1..infinity,f(x^n)) = exp(log(lambert(f(x))))}.
  oddlambert: % -> %
    ++\spad{oddlambert(f(x))} returns \spad{f(x) + f(x^3) + f(x^5) + ...}.
    ++ \spad{f(x)} should have a zero constant coefficient.
    ++ This function is used for computing infinite products.
    ++ If \spad{f(x)} is a Taylor series with constant term 1, then
    ++ \spad{product(n=1..infinity,f(x^(2*n-1)))=exp(log(oddlambert(f(x))))}.
  evenlambert: % -> %
    ++\spad{evenlambert(f(x))} returns \spad{f(x^2) + f(x^4) + f(x^6) + ...}.
    ++ \spad{f(x)} should have a zero constant coefficient.
    ++ This function is used for computing infinite products.
    ++ If \spad{f(x)} is a Taylor series with constant term 1, then
    ++ \spad{product(n=1..infinity,f(x^(2*n))) = exp(log(evenlambert(f(x))))}.
  generalLambert: (%,I,I) -> %
    ++\spad{generalLambert(f(x),a,d)} returns \spad{f(x^a) + f(x^(a + d)) +
    ++ f(x^(a + 2 d)) + ... }. \spad{f(x)} should have zero constant
    ++ coefficient and \spad{a} and d should be positive.
  revert: % -> %
    ++ \spad{revert(f(x))} returns a Taylor series \spad{g(x)} such that
    ++ \spad{f(g(x)) = g(f(x)) = x}. Series \spad{f(x)} should have constant
    ++ coefficient 0 and 1st order coefficient 1.
  multisect: (I,I,%) -> %
    ++\spad{multisect(a,b,f(x))} selects the coefficients of

```

```

    ++ \spad{x^((a+b)*n+a)}, and changes this monomial to \spad{x^n}.
invmultisect: (I,I,%) -> %
    ++\spad{invmultisect(a,b,f(x))} substitutes \spad{x^((a+b)*n)}
    ++ for \spad{x^n} and multiplies by \spad{x^b}.
if Coef has Algebra Fraction Integer then
    integrate: (%,Variable(var)) -> %
        ++ \spad{integrate(f(x),x)} returns an anti-derivative of the power
        ++ series \spad{f(x)} with constant coefficient 0.
        ++ We may integrate a series when we can divide coefficients
        ++ by integers.

Implementation ==> InnerTaylorSeries(Coef) add

Rep := Stream Coef

--% creation and destruction of series

stream: % -> Stream Coef
stream x == x pretend Stream(Coef)

coerce(v:Variable(var)) ==
    zero? cen => monomial(1,1)
    monomial(1,1) + monomial(cen,0)

coerce(n:I) == n :: Coef :: %
coerce(r:Coef) == coerce(r)$STT
monomial(c,n) == monom(c,n)$STT

getExpon: TERM -> NNI
getExpon term == term.k
getCoef: TERM -> Coef
getCoef term == term.c
rec: (NNI,Coef) -> TERM
rec(expon,coef) == [expon,coef]

recs: (ST Coef,NNI) -> ST TERM
recs(st,n) == delay$ST(TERM)
    empty? st => empty()
    zero? (coef := first st) => recs(rst st,n + 1)
    concat(rec(n,coef),recs(rst st,n + 1))

terms x == recs(stream x,0)

recsToCoefs: (ST TERM,NNI) -> ST Coef
recsToCoefs(st,n) == delay
    empty? st => empty()
    term := first st; expon := getExpon term
    n = expon => concat(getCoef term,recsToCoefs(rst st,n + 1))
    concat(0,recsToCoefs(st,n + 1))

```

```

series(st: ST TERM) == recsToCoefs(st,0)

stToPoly: (ST Coef,P,NNI,NNI) -> P
stToPoly(st,term,n,n0) ==
  (n > n0) or (empty? st) => 0
  first(st) * term ** n + stToPoly(rst st,term,n + 1,n0)

polynomial(x,n) == stToPoly(stream x,(var :: P) - (cen :: P),0,n)

polynomial(x,n1,n2) ==
  if n1 > n2 then (n1,n2) := (n2,n1)
  stToPoly(rest(stream x,n1),(var :: P) - (cen :: P),n1,n2)

stToUPoly: (ST Coef,UP,NNI,NNI) -> UP
stToUPoly(st,term,n,n0) ==
  (n > n0) or (empty? st) => 0
  first(st) * term ** n + stToUPoly(rst st,term,n + 1,n0)

univariatePolynomial(x,n) ==
  stToUPoly(stream x,monomial(1,1)$UP - monomial(cen,0)$UP,0,n)

coerce(p:UP) ==
  zero? p => 0
  if not zero? cen then
    p := p(monomial(1,1)$UP + monomial(cen,0)$UP)
  st : ST Coef := empty()
  oldDeg : NNI := degree(p) + 1
  while not zero? p repeat
    deg := degree p
    delta := (oldDeg - deg - 1) :: NNI
    for i in 1..delta repeat st := concat(0$Coef,st)
    st := concat(leadingCoefficient p,st)
    oldDeg := deg; p := reductum p
  for i in 1..oldDeg repeat st := concat(0$Coef,st)
  st

if Coef has coerce: Symbol -> Coef then
  if Coef has "***": (Coef,NNI) -> Coef then

    stToCoef: (ST Coef,Coef,NNI,NNI) -> Coef
    stToCoef(st,term,n,n0) ==
      (n > n0) or (empty? st) => 0
      first(st) * term ** n + stToCoef(rst st,term,n + 1,n0)

    approximate(x,n) ==
      stToCoef(stream x,(var :: Coef) - cen,0,n)

--% values

variable x == var

```

```

center    s == cen

coefficient(x,n) ==
  -- Cannot use elt!  Should return 0 if stream doesn't have it.
  u := stream x
  while not empty? u and n > 0 repeat
    u := rst u
    n := (n - 1) :: NNI
  empty? u or n ^= 0 => 0
  first u

elt(x:%,n:NNI) == coefficient(x,n)

--% functions

map(f,x) == map(f,x)$Rep
eval(x:%,r:Coef) == eval(stream x,r-cen)$STT
differentiate x == deriv(stream x)$STT
differentiate(x:%,v:Variable(var)) == differentiate x
if Coef has PartialDifferentialRing(Symbol) then
  differentiate(x:%,s:Symbol) ==
    (s = variable(x)) => differentiate x
  map(y +-> differentiate(y,s),x)
    - differentiate(center x,s)*differentiate(x)
multiplyCoefficients(f,x) == gderiv(f,stream x)$STT
lagrange x == lagrange(stream x)$STT
lambert x == lambert(stream x)$STT
oddlambert x == oddlambert(stream x)$STT
evenlambert x == evenlambert(stream x)$STT
generalLambert(x:%,a:I,d:I) == generalLambert(stream x,a,d)$STT
extend(x,n) == extend(x,n+1)$Rep
complete x == complete(x)$Rep
truncate(x,n) == first(stream x,n + 1)$Rep
truncate(x,n1,n2) ==
  if n2 < n1 then (n1,n2) := (n2,n1)
  m := (n2 - n1) :: NNI
  st := first(rest(stream x,n1)$Rep,m + 1)$Rep
  for i in 1..n1 repeat st := concat(0$Coef,st)
  st
elt(x:%,y:%) == compose(stream x,stream y)$STT
revert x == revert(stream x)$STT
multisect(a,b,x) == multisect(a,b,stream x)$STT
invmultisect(a,b,x) == invmultisect(a,b,stream x)$STT
multiplyExponents(x,n) == invmultisect(n,0,x)
quoByVar x == (empty? x => 0; rst x)
if Coef has IntegralDomain then
  unit? x == unit? coefficient(x,0)
if Coef has Field then
  if Coef is RN then
    (x:%) ** (s:Coef) == powern(s,stream x)$STT

```

```

else
  (x:%) ** (s:Coef) == power(s,stream x)$STT

if Coef has Algebra Fraction Integer then
  coerce(r:RN) == r :: Coef :: %

  integrate x == integrate(0,stream x)$STT
  integrate(x:%,v:Variable(var)) == integrate x

if Coef has integrate: (Coef,Symbol) -> Coef and _
  Coef has variables: Coef -> List Symbol then
  integrate(x:%,s:Symbol) ==
    (s = variable(x)) => integrate x
    not entry?(s,variables center x) => map(y +-> integrate(y,s),x)
    error "integrate: center is a function of variable of integration"

if Coef has TranscendentalFunctionCategory and _
  Coef has PrimitiveFunctionCategory and _
  Coef has AlgebraicallyClosedFunctionSpace Integer then

  integrateWithOneAnswer: (Coef,Symbol) -> Coef
  integrateWithOneAnswer(f,s) ==
    res := integrate(f,s)$FunctionSpaceIntegration(I,Coef)
    res case Coef => res :: Coef
    first(res :: List Coef)

  integrate(x:%,s:Symbol) ==
    (s = variable(x)) => integrate x
    not entry?(s,variables center x) =>
      map(y +-> integrateWithOneAnswer(y,s),x)
      error "integrate: center is a function of variable of integration"

--% OutputForms
-- We use the default coerce: % -> OutputForm in UTSCAT&



---



— UTS.dotabb —

"UTS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=UTS"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"UTS" -> "ACFS"



---



```

domain UTSZ UnivariateTaylorSeriesCZero

— UnivariateTaylorSeriesCZero.input —

```

)set break resume
)sys rm -f UnivariateTaylorSeriesCZero.output
)spool UnivariateTaylorSeriesCZero.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariateTaylorSeriesCZero
--R
--R UnivariateTaylorSeriesCZero(Coef: Ring,var: Symbol) is a domain constructor
--R Abbreviation for UnivariateTaylorSeriesCZero is UTSZ
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for UTSZ
--R
--R----- Operations -----
--R ?? : (Coef,%) -> %          ?? : (%,Coef) -> %
--R ?? : (%,%) -> %            ?? : (Integer,%) -> %
--R ?? : (PositiveInteger,%) -> %   ??? : (%,PositiveInteger) -> %
--R ?+? : (%,%) -> %            ?-? : (%,%) -> %
--R -? : % -> %                 ?=? : (%,%) -> Boolean
--R 1 : () -> %                 0 : () -> %
--R ??? : (%,PositiveInteger) -> % center : % -> Coef
--R coefficients : % -> Stream(Coef) coerce : Variable(var) -> %
--R coerce : Integer -> %        coerce : % -> OutputForm
--R complete : % -> %            degree : % -> NonNegativeInteger
--R evenlambert : % -> %         hash : % -> SingleInteger
--R lagrange : % -> %           lambert : % -> %
--R latex : % -> String          leadingCoefficient : % -> Coef
--R leadingMonomial : % -> %     map : ((Coef -> Coef),%) -> %
--R monomial? : % -> Boolean     oddlambert : % -> %
--R one? : % -> Boolean          order : % -> NonNegativeInteger
--R pole? : % -> Boolean         quoByVar : % -> %
--R recip : % -> Union(%, "failed") reductum : % -> %
--R revert : % -> %             sample : () -> %
--R series : Stream(Coef) -> %    variable : % -> Symbol
--R zero? : % -> Boolean         ?~=? : (%,%) -> Boolean
--R ?? : (%,Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (Fraction(Integer),%) -> % if Coef has ALGEBRA(FRAC(INT))
--R ?? : (NonNegativeInteger,%) -> %
--R ??? : (%,Fraction(Integer)) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (%,%) -> % if Coef has ALGEBRA(FRAC(INT))
--R ??? : (%,Coef) -> % if Coef has FIELD
--R ??? : (%,NonNegativeInteger) -> %

```

```

--R ?/? : (% , Coef) -> % if Coef has FIELD
--R D : % -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef
--R D : (% , NonNegativeInteger) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef
--R D : (% , Symbol) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and Coef has PDRING(SYM
--R D : (% , List(Symbol)) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and Coef has PDRI
--R D : (% , Symbol, NonNegativeInteger) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and
--R D : (% , List(Symbol), List(NonNegativeInteger)) -> % if Coef has *: (NonNegativeInteger, Coef)
--R ?^? : (% , NonNegativeInteger) -> %
--R acos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acoth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R acsch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R approximate : (% , NonNegativeInteger) -> Coef if Coef has **: (Coef, NonNegativeInteger) -> Co
--R asec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R asinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R associates? : (% , %) -> Boolean if Coef has INTDOM
--R atan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R atanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R characteristic : () -> NonNegativeInteger
--R charthRoot : % -> Union(% , "failed") if Coef has CHARNZ
--R coefficient : (% , NonNegativeInteger) -> Coef
--R coerce : UnivariatePolynomial(var, Coef) -> %
--R coerce : Coef -> % if Coef has COMRING
--R coerce : % -> % if Coef has INTDOM
--R coerce : Fraction(Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R cos : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cosh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R cot : % -> % if Coef has ALGEBRA(FRAC(INT))
--R coth : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csc : % -> % if Coef has ALGEBRA(FRAC(INT))
--R csch : % -> % if Coef has ALGEBRA(FRAC(INT))
--R differentiate : (% , Variable(var)) -> %
--R differentiate : % -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef
--R differentiate : (% , NonNegativeInteger) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef
--R differentiate : (% , Symbol) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and Coef ha
--R differentiate : (% , List(Symbol)) -> % if Coef has *: (NonNegativeInteger, Coef) -> Coef and C
--R differentiate : (% , Symbol, NonNegativeInteger) -> % if Coef has *: (NonNegativeInteger, Coef)
--R differentiate : (% , List(Symbol), List(NonNegativeInteger)) -> % if Coef has *: (NonNegativeIn
--R ?.? : (% , %) -> % if NonNegativeInteger has SGROUP
--R ?.? : (% , NonNegativeInteger) -> Coef
--R eval : (% , Coef) -> Stream(Coef) if Coef has **: (Coef, NonNegativeInteger) -> Coef
--R exp : % -> % if Coef has ALGEBRA(FRAC(INT))
--R exquo : (% , %) -> Union(% , "failed") if Coef has INTDOM
--R extend : (% , NonNegativeInteger) -> %
--R generalLambert : (% , Integer, Integer) -> %
--R integrate : (% , Variable(var)) -> % if Coef has ALGEBRA(FRAC(INT))

```

```

--R integrate : (% ,Symbol) -> % if Coef has integrate: (Coef,Symbol) -> Coef and Coef has variables: Coef ->
--R integrate : % -> % if Coef has ALGEBRA(FRAC(INT))
--R invmultisect : (Integer,Integer,%) -> %
--R log : % -> % if Coef has ALGEBRA(FRAC(INT))
--R monomial : (% ,List(SingletonAsOrderedSet),List(NonNegativeInteger)) -> %
--R monomial : (% ,SingletonAsOrderedSet,NonNegativeInteger) -> %
--R monomial : (Coef,NonNegativeInteger) -> %
--R multiplyCoefficients : ((Integer -> Coef),%) -> %
--R multiplyExponents : (% ,PositiveInteger) -> %
--R multisect : (Integer,Integer,%) -> %
--R nthRoot : (% ,Integer) -> % if Coef has ALGEBRA(FRAC(INT))
--R order : (% ,NonNegativeInteger) -> NonNegativeInteger
--R pi : () -> % if Coef has ALGEBRA(FRAC(INT))
--R polynomial : (% ,NonNegativeInteger,NonNegativeInteger) -> Polynomial(Coef)
--R polynomial : (% ,NonNegativeInteger) -> Polynomial(Coef)
--R sec : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sech : % -> % if Coef has ALGEBRA(FRAC(INT))
--R series : Stream(Record(k: NonNegativeInteger,c: Coef)) -> %
--R sin : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sinh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R sqrt : % -> % if Coef has ALGEBRA(FRAC(INT))
--R subtractIfCan : (% ,%) -> Union(% ,"failed")
--R tan : % -> % if Coef has ALGEBRA(FRAC(INT))
--R tanh : % -> % if Coef has ALGEBRA(FRAC(INT))
--R terms : % -> Stream(Record(k: NonNegativeInteger,c: Coef))
--R truncate : (% ,NonNegativeInteger,NonNegativeInteger) -> %
--R truncate : (% ,NonNegativeInteger) -> %
--R unit? : % -> Boolean if Coef has INTDOM
--R unitCanonical : % -> % if Coef has INTDOM
--R unitNormal : % -> Record(unit: % ,canonical: % ,associate: % ) if Coef has INTDOM
--R univariatePolynomial : (% ,NonNegativeInteger) -> UnivariatePolynomial(var,Coef)
--R variables : % -> List(SingletonAsOrderedSet)
--R
--E 1

```

```

)spool
)lisp (bye)

```

— UnivariateTaylorSeriesCZero.help —

```

=====
UnivariateTaylorSeriesCZero examples
=====

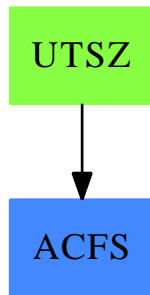
```

Part of the Package for Algebraic Function Fields in one variable PAFF

See Also:


```
o )show UnivariateTaylorSeriesCZero
```

UnivariateTaylorSeriesCZero (UTSZ)



See

⇒ “InnerTaylorSeries” (ITAYLOR) [10](#) on page [1338](#)

Exports:

0	1	acos	acosh
acot	acoth	acsc	acsch
approximate	asec	asech	asin
asinh	associates?	atan	atanh
center	characteristic	charthRoot	coefficient
coefficients	coerce	complete	cos
cosh	cot	coth	csc
csch	D	degree	differentiate
eval	evenlambert	exp	exquo
extend	generalLambert	hash	integrate
invmultisect	lagrange	lambert	latex
leadingCoefficient	leadingMonomial	log	map
monomial	monomial?	multiplyCoefficients	multiplyExponents
multisect	nthRoot	oddlambert	one?
order	pi	pole?	polynomial
quoByVar	recip	reductum	revert
sample	sec	sech	series
sin	sinh	sqrt	subtractIfCan
tan	tanh	terms	truncate
unit?	unitCanonical	unitNormal	univariatePolynomial
variable	variables	zero?	?.?
?~=?	?/?	?*?	?**?
?+?	?-?	-?	?=?
?^?			

— domain UTSZ UnivariateTaylorSeriesCZero —

```
)abbrev domain UTSZ UnivariateTaylorSeriesCZero
++ Author: Gaetan Hache
++ Date Created: September 1996
++ Date Last Updated: April, 2010, by Tim Daly
++ Description:
++ Part of the Package for Algebraic Function Fields in one variable PAFF
```

```
UnivariateTaylorSeriesCZero(Coef,var): Exports == Implementation where
```

```
  Coef : Ring
  var  : Symbol
  I    ==> Integer
  NNI  ==> NonNegativeInteger
  P    ==> Polynomial Coef
  RN   ==> Fraction Integer
  ST   ==> Stream
  STT  ==> StreamTaylorSeriesOperations Coef
  TERM ==> Record(k:NNI,c:Coef)
  UP   ==> UnivariatePolynomial(var,Coef)
Exports ==> UnivariateTaylorSeriesCategory(Coef) with
  coerce: UP -> %
```

```

++\spad{coerce(p)} converts a univariate polynomial p in the variable
++\spad{var} to a univariate Taylor series in \spad{var}.
univariatePolynomial: (%,NNI) -> UP
++\spad{univariatePolynomial(f,k)} returns a univariate polynomial
++ consisting of the sum of all terms of f of degree \spad{<= k}.
coerce: Variable(var) -> %
++\spad{coerce(var)} converts the series variable \spad{var} into a
++ Taylor series.
differentiate: (%,Variable(var)) -> %
++ \spad{differentiate(f(x),x)} computes the derivative of
++ \spad{f(x)} with respect to \spad{x}.
lagrange: % -> %
++\spad{lagrange(g(x))} produces the Taylor series for \spad{f(x)}
++ where \spad{f(x)} is implicitly defined as \spad{f(x) = x*g(f(x))}.
lambert: % -> %
++\spad{lambert(f(x))} returns \spad{f(x) + f(x^2) + f(x^3) + ...}.
++ This function is used for computing infinite products.
++ \spad{f(x)} should have zero constant coefficient.
++ If \spad{f(x)} is a Taylor series with constant term 1, then
++ \spad{product(n = 1..infinity,f(x^n)) = exp(log(lambert(f(x))))}.
oddlambert: % -> %
++\spad{oddlambert(f(x))} returns \spad{f(x) + f(x^3) + f(x^5) + ...}.
++ \spad{f(x)} should have a zero constant coefficient.
++ This function is used for computing infinite products.
++ If \spad{f(x)} is a Taylor series with constant term 1, then
++ \spad{product(n=1..infinity,f(x^(2*n-1)))=exp(log(oddlambert(f(x))))}.
evenlambert: % -> %
++\spad{evenlambert(f(x))} returns \spad{f(x^2) + f(x^4) + f(x^6) + ...}.
++ \spad{f(x)} should have a zero constant coefficient.
++ This function is used for computing infinite products.
++ If \spad{f(x)} is a Taylor series with constant term 1, then
++ \spad{product(n=1..infinity,f(x^(2*n)))=exp(log(evenlambert(f(x))))}.
generalLambert: (%,I,I) -> %
++\spad{generalLambert(f(x),a,d)} returns \spad{f(x^a) + f(x^(a + d)) +
++ f(x^(a + 2 d)) + ... }. \spad{f(x)} should have zero constant
++ coefficient and \spad{a} and d should be positive.
revert: % -> %
++ \spad{revert(f(x))} returns a Taylor series \spad{g(x)} such that
++ \spad{f(g(x)) = g(f(x)) = x}. Series \spad{f(x)} should have constant
++ coefficient 0 and 1st order coefficient 1.
multisect: (I,I,%) -> %
++\spad{multisect(a,b,f(x))} selects the coefficients of
++ \spad{x^((a+b)*n+a)}, and changes this monomial to \spad{x^n}.
invmultisect: (I,I,%) -> %
++\spad{invmultisect(a,b,f(x))} substitutes \spad{x^((a+b)*n)}
++ for \spad{x^n} and multiplies by \spad{x^b}.
if Coef has Algebra Fraction Integer then
integrate: (%,Variable(var)) -> %
++ \spad{integrate(f(x),x)} returns an anti-derivative of the power
++ series \spad{f(x)} with constant coefficient 0.

```

```

    ++ We may integrate a series when we can divide coefficients
    ++ by integers.

Implementation ==> InnerTaylorSeries(Coef) add

Rep := Stream Coef

--% creation and destruction of series

stream: % -> Stream Coef
stream x == x pretend Stream(Coef)

coerce(v:Variable(var)) ==
  monomial(1,1)

coerce(n:I) == n :: Coef :: %
coerce(r:Coef) == coerce(r)$STT
monomial(c,n) == monom(c,n)$STT

getExpon: TERM -> NNI
getExpon term == term.k
getCoef: TERM -> Coef
getCoef term == term.c
rec: (NNI,Coef) -> TERM
rec(expon,coef) == [expon,coef]

recs: (ST Coef,NNI) -> ST TERM
recs(st,n) == delay$ST(TERM)
  empty? st => empty()
  zero? (coef := first st) => recs(rst st,n + 1)
  concat(rec(n,coef),recs(rst st,n + 1))

terms x == recs(stream x,0)

recsToCoefs: (ST TERM,NNI) -> ST Coef
recsToCoefs(st,n) == delay
  empty? st => empty()
  term := first st; expon := getExpon term
  n = expon => concat(getCoef term,recsToCoefs(rst st,n + 1))
  concat(0,recsToCoefs(st,n + 1))

series(st: ST TERM) == recsToCoefs(st,0)

stToPoly: (ST Coef,P,NNI,NNI) -> P
stToPoly(st,term,n,n0) ==
  (n > n0) or (empty? st) => 0
  first(st) * term ** n + stToPoly(rst st,term,n + 1,n0)

polynomial(x,n) == stToPoly(stream x,(var :: P),0,n)

```

```

polynomial(x,n1,n2) ==
  if n1 > n2 then (n1,n2) := (n2,n1)
  stToPoly(rest(stream x,n1),(var :: P),n1,n2)

stToUPoly: (ST Coef,UP,NNI,NNI) -> UP
stToUPoly(st,term,n,n0) ==
  (n > n0) or (empty? st) => 0
  frst(st) * term ** n + stToUPoly(rst st,term,n + 1,n0)

univariatePolynomial(x,n) ==
  stToUPoly(stream x,monomial(1,1)$UP,0,n)

coerce(p:UP) ==
  zero? p => 0
  st : ST Coef := empty()
  oldDeg : NNI := degree(p) + 1
  while not zero? p repeat
    deg := degree p
    delta := (oldDeg - deg - 1) :: NNI
    for i in 1..delta repeat st := concat(0$Coef,st)
    st := concat(leadingCoefficient p,st)
    oldDeg := deg; p := reductum p
  for i in 1..oldDeg repeat st := concat(0$Coef,st)
  st

if Coef has coerce: Symbol -> Coef then
  if Coef has "***": (Coef,NNI) -> Coef then

    stToCoef: (ST Coef,Coef,NNI,NNI) -> Coef
    stToCoef(st,term,n,n0) ==
      (n > n0) or (empty? st) => 0
      frst(st) * term ** n + stToCoef(rst st,term,n + 1,n0)

    approximate(x,n) ==
      stToCoef(stream x,(var :: Coef),0,n)

--% values

variable x == var
center x == 0$Coef

coefficient(x,n) ==
  -- Cannot use elt! Should return 0 if stream doesn't have it.
  u := stream x
  while not empty? u and n > 0 repeat
    u := rst u
    n := (n - 1) :: NNI
  empty? u or n ^= 0 => 0
  frst u

```

```

elt(x:%,n:NNI) == coefficient(x,n)

--% functions

map(f,x) == map(f,x)$Rep
eval(x:%,r:Coef) == eval(stream x,r)$STT
differentiate x == deriv(stream x)$STT
differentiate(x:%,v:Variable(var)) == differentiate x
if Coef has PartialDifferentialRing(Symbol) then
  differentiate(x:%,s:Symbol) ==
    (s = variable(x)) => differentiate x
    map(differentiate(#1,s),x) - differentiate(0,s)*differentiate(x)
multiplyCoefficients(f,x) == gderiv(f,stream x)$STT
lagrange x == lagrange(stream x)$STT
lambert x == lambert(stream x)$STT
oddlambert x == oddlambert(stream x)$STT
evenlambert x == evenlambert(stream x)$STT
generalLambert(x:%,a:I,d:I) == generalLambert(stream x,a,d)$STT
extend(x,n) == extend(x,n+1)$Rep
complete x == complete(x)$Rep
truncate(x,n) == first(stream x,n + 1)$Rep
truncate(x,n1,n2) ==
  if n2 < n1 then (n1,n2) := (n2,n1)
  m := (n2 - n1) :: NNI
  st := first(rest(stream x,n1)$Rep,m + 1)$Rep
  for i in 1..n1 repeat st := concat(0$Coef,st)
  st
elt(x:%,y:%) == compose(stream x,stream y)$STT
revert x == revert(stream x)$STT
multisect(a,b,x) == multisect(a,b,stream x)$STT
invmultisect(a,b,x) == invmultisect(a,b,stream x)$STT
multiplyExponents(x,n) == invmultisect(n,0,x)
quoByVar x == (empty? x => 0; rst x)
if Coef has IntegralDomain then
  unit? x == unit? coefficient(x,0)
if Coef has Field then
  if Coef is RN then
    (x:%) ** (s:Coef) == powern(s,stream x)$STT
  else
    (x:%) ** (s:Coef) == power(s,stream x)$STT

if Coef has Algebra Fraction Integer then
  coerce(r:RN) == r :: Coef :: %

integrate x == integrate(0,stream x)$STT
integrate(x:%,v:Variable(var)) == integrate x

if Coef has integrate: (Coef,Symbol) -> Coef and _
  Coef has variables: Coef -> List Symbol then
  integrate(x:%,s:Symbol) ==

```

```

(s = variable(x)) => integrate x
map(integrate(#1,s),x)

if Coef has TranscendentalFunctionCategory and _
  Coef has PrimitiveFunctionCategory and _
  Coef has AlgebraicallyClosedFunctionSpace Integer then

integrateWithOneAnswer: (Coef,Symbol) -> Coef
integrateWithOneAnswer(f,s) ==
  res := integrate(f,s)$FunctionSpaceIntegration(I,Coef)
  res case Coef => res :: Coef
  first(res :: List Coef)

integrate(x:%,s:Symbol) ==
  (s = variable(x)) => integrate x
  map(integrateWithOneAnswer(#1,s),x)

```

— UTSZ.dotabb —

```

"UTSZ" [color="#88FF44",href="bookvol10.3.pdf#nameddest=UTSZ"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"UTSZ" -> "ACFS"

```

domain UNISEG UniversalSegment

— UniversalSegment.input —

```

)set break resume
)sys rm -f UniversalSegment.output
)spool UniversalSegment.output
)set message test on
)set message auto off
)clear all
--S 1 of 9
pints := 1..
--R
--R
--R (1)  1..
--R
--R                                         Type: UniversalSegment(PositiveInteger)
--E 1

```

```

--S 2 of 9
nevens := (0..) by -2
--R
--R
--R (2)  0.. by - 2
--R
--R                                         Type: UniversalSegment(NonNegativeInteger)
--E 2

--S 3 of 9
useg: UniversalSegment(Integer) := 3..10
--R
--R
--R (3)  3..10
--R
--R                                         Type: UniversalSegment(Integer)
--E 3

--S 4 of 9
hasHi pints
--R
--R
--R (4)  false
--R
--R                                         Type: Boolean
--E 4

--S 5 of 9
hasHi nevens
--R
--R
--R (5)  false
--R
--R                                         Type: Boolean
--E 5

--S 6 of 9
hasHi useg
--R
--R
--R (6)  true
--R
--R                                         Type: Boolean
--E 6

--S 7 of 9
expand pints
--R
--R
--R (7)  [1,2,3,4,5,6,7,8,9,10,...]
--R
--R                                         Type: Stream(Integer)
--E 7

--S 8 of 9

```

```
=====
UniversalSegment examples
=====
```

```
hasHi pints
false
```

Type: Boolean

```
hasHi nevens  
false
```

Type: Boolean

```
hasHi useg  
true
```

Type: Boolean

All operations available on type Segment apply to UniversalSegment, with the proviso that expansions produce streams rather than lists. This is to accommodate infinite expansions.

```
expand pints  
[1,2,3,4,5,6,7,8,9,10,...]
```

Type: Stream Integer

```
expand nevens  
[0,- 2,- 4,- 6,- 8,- 10,- 12,- 14,- 16,- 18,...]
```

Type: Stream Integer

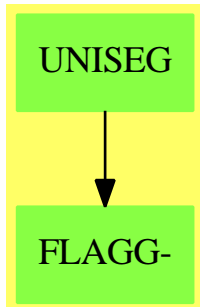
```
expand [1, 3, 10..15, 100..]  
[1,3,10,11,12,13,14,15,100,101,...]
```

Type: Stream Integer

See Also:

- o)help Segment
- o)help SegmentBinding
- o)help List
- o)help Stream
- o)show UniversalSegment

UniversalSegment (UNISEG)



See

⇒ “Segment” (SEG) [20](#) on page [2359](#)

⇒ “SegmentBinding” (SEGBIND) [20](#) on page [2364](#)

Exports:

BY	coerce	convert	expand	hasHi
hash	hi	high	incr	latex
lo	low	map	segment	segment
?=?	?SEGMENT	?..?	?~=?	

— domain UNISEG UniversalSegment —

```

)abbrev domain UNISEG UniversalSegment
++ Author:  Robert S. Sutor
++ Date Created: 1987
++ Date Last Updated: June 4, 1991
++ Description:
++ This domain provides segments which may be half open.
++ That is, ranges of the form \spad{a..} or \spad{a..b}.

UniversalSegment(S: Type): SegmentCategory(S) with
  SEGMENT: S -> %
    ++ \spad{1..} produces a half open segment,
    ++ that is, one with no upper bound.
  segment: S -> %
    ++ segment(1) is an alternate way to construct the segment \spad{1..}.
  coerce : Segment S -> %
    ++ coerce(x) allows \spadtype{Segment} values to be used as %.
  hasHi: % -> Boolean
    ++ hasHi(s) tests whether the segment s has an upper bound.

if S has SetCategory then SetCategory

if S has OrderedRing then
  SegmentExpansionCategory(S, Stream S)
--  expand : (List %, S) -> Stream S
  
```

```

--      expand : (% , S) -> Stream S

== add
  Rec ==> Record(low: S, high: S, incr: Integer)
  Rec2 ==> Record(low: S, incr: Integer)
  SEG ==> Segment S

  Rep := Union(Rec2, Rec)
  a,b : S
  s : %
  i: Integer
  ls : List %

  segment a == [a, 1]$Rec2 :: Rep
  segment(a,b) == [a,b,1]$Rec :: Rep
  BY(s,i) ==
    s case Rec => [lo s, hi s, i]$Rec ::Rep
    [lo s, i]$Rec2 :: Rep

  lo s ==
    s case Rec2 => (s :: Rec2).low
    (s :: Rec).low

  low s ==
    s case Rec2 => (s :: Rec2).low
    (s :: Rec).low

  hasHi s == s case Rec

  hi s ==
    not hasHi(s) => error "hi: segment has no upper bound"
    (s :: Rec).high

  high s ==
    not hasHi(s) => error "high: segment has no upper bound"
    (s :: Rec).high

  incr s ==
    s case Rec2 => (s :: Rec2).incr
    (s :: Rec).incr

  SEGMENT(a) == segment a
  SEGMENT(a,b) == segment(a,b)

  coerce(sg : SEG): % == segment(lo sg, hi sg)

  convert a == [a,a,1]

  if S has SetCategory then

```

```

(s1:%) = (s2:%) ==
  s1 case Rec2 =>
    s2 case Rec2 =>
      s1.low = s2.low and s1.incr = s2.incr
    false
  s1 case Rec =>
    s2 case Rec =>
      s2.low = s1.low and s1.high=s2.high and s1.incr=s2.incr
    false
  false

coerce(s: %): OutputForm ==
  seg :=
    e := (lo s)::OutputForm
    hasHi s => SEGMENT(e, (hi s)::OutputForm)
  SEGMENT e
  inc := incr s
  inc = 1 => seg
  infix(" by " :: OutputForm, seg, inc :: OutputForm)

if S has OrderedRing then
  expand(s:%) == expand([s])
  map(f:S->S, s:%) == map(f, expand s)

plusInc(t: S, a: S): S == t + a

expand(ls: List %):Stream S ==
  st:Stream S := empty()
  null ls => st

  lb:List(Segment S) := nil()
  while not null ls and hasHi first ls repeat
    s := first ls
    ls := rest ls
    ns := BY(SEGMENT(lo s, hi s), incr s)$Segment(S)
    lb := concat_!(lb,ns)
  if not null ls then
    s := first ls
    st: Stream S := generate(x +-> x+incr(s)::S, lo s)
  else
    st: Stream S := empty()
  concat(construct expand(lb), st)

```

— UNISEG.dotabb —

"UNISEG" [color="#88FF44",href="bookvol10.3.pdf#nameddest=UNISEG"]

"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
 "UNISEG" -> "FLAGG-"

domain U8MAT U8Matrix

— U8Matrix.input —

```
)set break resume
)sys rm -f U8Matrix.output
)spool U8Matrix.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show U8Matrix
--R
--R U8Matrix is a domain constructor
--R Abbreviation for U8Matrix is U8MAT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for U8MAT
--R
--R----- Operations -----
--R ??? : (U8Vector,%) -> U8Vector      ??? : (%,U8Vector) -> U8Vector
--R ??? : (Integer,%) -> %              ??? : (%,Integer) -> %
--R ??? : (Integer,%) -> %              ??? : (%,%) -> %
--R ?+? : (%,%) -> %                   -? : % -> %
--R ?-? : (%,%) -> %                   antisymmetric? : % -> Boolean
--R coerce : U8Vector -> %              column : (%,Integer) -> U8Vector
--R copy : % -> %                       diagonal? : % -> Boolean
--R diagonalMatrix : List(%) -> %        empty : () -> %
--R empty? : % -> Boolean                eq? : (%,%) -> Boolean
--R fill! : (%,Integer) -> %             horizConcat : (%,%) -> %
--R matrix : List(List(Integer)) -> %    maxColIndex : % -> Integer
--R maxRowIndex : % -> Integer            minColIndex : % -> Integer
--R minRowIndex : % -> Integer            ncols : % -> NonNegativeInteger
--R nrows : % -> NonNegativeInteger       parts : % -> List(Integer)
--R qnew : (Integer,Integer) -> %         row : (%,Integer) -> U8Vector
--R sample : () -> %                     square? : % -> Boolean
--R squareTop : % -> %                   symmetric? : % -> Boolean
--R transpose : % -> %                   transpose : U8Vector -> %
--R vertConcat : (%,%) -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?**? : (%,Integer) -> % if Integer has FIELD
```

```

--R ??? : (% , NonNegativeInteger) -> %
--R ?/? : (% , Integer) -> % if Integer has FIELD
--R ?=? : (% , %) -> Boolean if Integer has SETCAT
--R any? : ((Integer -> Boolean), %) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if Integer has SETCAT
--R columnSpace : % -> List(U8Vector) if Integer has EUCDOM
--R count : (Integer, %) -> NonNegativeInteger if $ has finiteAggregate and Integer has SETCAT
--R count : ((Integer -> Boolean), %) -> NonNegativeInteger if $ has finiteAggregate
--R determinant : % -> Integer if Integer has commutative(*)
--R diagonalMatrix : List(Integer) -> %
--R elt : (% , List(Integer), List(Integer)) -> %
--R elt : (% , Integer, Integer, Integer) -> Integer
--R elt : (% , Integer, Integer) -> Integer
--R eval : (% , List(Integer), List(Integer)) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (% , Integer, Integer) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (% , Equation(Integer)) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (% , List(Equation(Integer))) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R every? : ((Integer -> Boolean), %) -> Boolean if $ has finiteAggregate
--R exquo : (% , Integer) -> Union(%, "failed") if Integer has INTDOM
--R hash : % -> SingleInteger if Integer has SETCAT
--R inverse : % -> Union(%, "failed") if Integer has FIELD
--R latex : % -> String if Integer has SETCAT
--R less? : (% , NonNegativeInteger) -> Boolean
--R listOfLists : % -> List(List(Integer))
--R map : (((Integer, Integer) -> Integer), %, %, Integer) -> %
--R map : (((Integer, Integer) -> Integer), %, %) -> %
--R map : ((Integer -> Integer), %) -> %
--R map! : ((Integer -> Integer), %) -> %
--R matrix : (NonNegativeInteger, NonNegativeInteger, ((Integer, Integer) -> Integer)) -> %
--R member? : (Integer, %) -> Boolean if $ has finiteAggregate and Integer has SETCAT
--R members : % -> List(Integer) if $ has finiteAggregate
--R minordet : % -> Integer if Integer has commutative(*)
--R more? : (% , NonNegativeInteger) -> Boolean
--R new : (NonNegativeInteger, NonNegativeInteger, Integer) -> %
--R nullSpace : % -> List(U8Vector) if Integer has INTDOM
--R nullity : % -> NonNegativeInteger if Integer has INTDOM
--R pfaffian : % -> Integer if Integer has COMRING
--R qelt : (% , Integer, Integer) -> Integer
--R qsetelt! : (% , Integer, Integer, Integer) -> Integer
--R rank : % -> NonNegativeInteger if Integer has INTDOM
--R rowEchelon : % -> % if Integer has EUCDOM
--R scalarMatrix : (NonNegativeInteger, Integer) -> %
--R setColumn! : (% , Integer, U8Vector) -> %
--R setRow! : (% , Integer, U8Vector) -> %
--R setelt : (% , List(Integer), List(Integer), %) -> %
--R setelt : (% , Integer, Integer, Integer) -> Integer
--R setsubMatrix! : (% , Integer, Integer, %) -> %
--R size? : (% , NonNegativeInteger) -> Boolean
--R subMatrix : (% , Integer, Integer, Integer, Integer) -> %
--R swapColumns! : (% , Integer, Integer) -> %

```

```

--R swapRows! : (%,Integer,Integer) -> %
--R zero : (NonNegativeInteger,NonNegativeInteger) -> %
--R ?~=? : (%,% ) -> Boolean if Integer has SETCAT
--R
--E 1

```

```

)spool
)lisp (bye)

```

— U8Matrix.help —

=====

U8Matrix examples

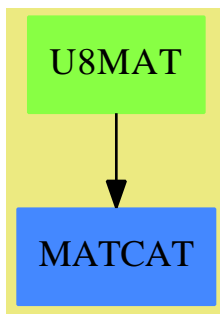
=====

This is a low-level domain which implements matrices (two dimensional arrays) of 8-bit integers. Indexing is 0 based, there is no bound checking (unless provided by lower level).

See Also:

- o)show U8Matrix
- o)show U16Matrix
- o)show U32Matrix

U8Matrix (U8MAT)



See

Exports:

#?	-?	***?	?*?
?+?	?-?	?/?	?=?
?^=?	antisymmetric?	any?	coerce
column	columnSpace	copy	count
determinant	diagonal?	diagonalMatrix	elt
empty	empty?	eq?	eval
every?	exquo	fill!	hash
horizConcat	inverse	latex	less?
listOfLists	map	map!	matrix
maxColIndex	maxRowIndex	member?	members
minColIndex	minRowIndex	minordet	more?
ncols	new	nrows	nullSpace
nullity	parts	pfaffian	qelt
qnew	qsetelt!	rank	row
rowEchelon	sample	scalarMatrix	setColumn!
setRow!	setelt	setsubMatrix!	size?
square?	squareTop	subMatrix	swapColumns!
swapRows!	symmetric?	transpose	vertConcat
zero			

— domain U8MAT U8Matrix —

```

)abbrev domain U8MAT U8Matrix
++ Author: Waldek Hebisch
++ Description:
++ This is a low-level domain which implements matrices
++ (two dimensional arrays) of 8-bit integers.
++ Indexing is 0 based, there is no bound checking (unless
++ provided by lower level).
U8Matrix : MatrixCategory(Integer,
                           U8Vector,
                           U8Vector) with
  qnew : (Integer, Integer) -> %
    ++ qnew(n, m) creates a new n by m matrix of zeros.
    ++
    ++X qnew(3,4)$U8Matrix()
== add

R ==> Integer

Qelt2    ==> AREF2U8$Lisp
Qsetelt2 ==> SETAREF2U8$Lisp
Qnrows   ==> ANROWSU8$Lisp
Qncols   ==> ANCOLSU8$Lisp
Qnew     ==> MAKEMATRIXU8$Lisp
Qnew1    ==> MAKEMATRIX1U8$Lisp

```

```

minRowIndex x == 0
minColIndex x == 0
nrows x == Qnrows(x)
ncols x == Qncols(x)
maxRowIndex x == Qnrows(x) - 1
maxColIndex x == Qncols(x) - 1

qelt(m, i, j) == Qelt2(m, i, j)
elt(m : %, i : Integer, j : Integer) : R == Qelt2(m, i, j)
qsetelt!(m, i, j, r) == Qsetelt2(m, i, j, r)
setelt(m : %, i : Integer, j : Integer, r : R) == Qsetelt2(m, i, j, r)

empty() == Qnew(0$Integer, 0$Integer)
qnew(rows, cols) == Qnew(rows, cols)
new(rows, cols, a) == Qnew1(rows, cols, a)

```

— U8MAT.dotabb —

```

"U8MAT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=U8MAT"]
"MATCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MATCAT"]
"U8MAT" -> "MATCAT"

```

domain U16MAT U16Matrix

— U16Matrix.input —

```

)set break resume
)sys rm -f U16Matrix.output
)spool U16Matrix.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show U16Matrix
--R U16Matrix is a domain constructor
--R Abbreviation for U16Matrix is U16MAT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for U16MAT
--R

```

```

--R----- Operations -----
--R ?? : (U16Vector,%) -> U16Vector      ?? : (%,U16Vector) -> U16Vector
--R ?? : (Integer,%) -> %                ?? : (%,Integer) -> %
--R ?? : (Integer,%) -> %                ?? : (%,%) -> %
--R ?? : (%,%) -> %                      -? : % -> %
--R ?? : (%,%) -> %                      antisymmetric? : % -> Boolean
--R coerce : U16Vector -> %              column : (%,Integer) -> U16Vector
--R copy : % -> %                        diagonal? : % -> Boolean
--R diagonalMatrix : List(%) -> %        empty : () -> %
--R empty? : % -> Boolean                 eq? : (%,%) -> Boolean
--R fill! : (%,Integer) -> %             horizConcat : (%,%) -> %
--R matrix : List(List(Integer)) -> %    maxColIndex : % -> Integer
--R maxRowIndex : % -> Integer           minColIndex : % -> Integer
--R minRowIndex : % -> Integer           ncols : % -> NonNegativeInteger
--R nrows : % -> NonNegativeInteger      parts : % -> List(Integer)
--R qnew : (Integer,Integer) -> %        row : (%,Integer) -> U16Vector
--R sample : () -> %                    square? : % -> Boolean
--R squareTop : % -> %                  symmetric? : % -> Boolean
--R transpose : % -> %                  transpose : U16Vector -> %
--R vertConcat : (%,%) -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ??? : (%,Integer) -> % if Integer has FIELD
--R ??? : (%,NonNegativeInteger) -> %
--R ??/? : (%,Integer) -> % if Integer has FIELD
--R ??=? : (%,%) -> Boolean if Integer has SETCAT
--R any? : ((Integer -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if Integer has SETCAT
--R columnSpace : % -> List(U16Vector) if Integer has EUCDOM
--R count : (Integer,%) -> NonNegativeInteger if $ has finiteAggregate and Integer has SETCAT
--R count : ((Integer -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R determinant : % -> Integer if Integer has commutative(*)
--R diagonalMatrix : List(Integer) -> %
--R elt : (%,List(Integer),List(Integer)) -> %
--R elt : (%,Integer,Integer,Integer) -> Integer
--R elt : (%,Integer,Integer) -> Integer
--R eval : (%,List(Integer),List(Integer)) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (%,Integer,Integer) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (%,Equation(Integer)) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (%,List(Equation(Integer))) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R every? : ((Integer -> Boolean),%) -> Boolean if $ has finiteAggregate
--R exquo : (%,Integer) -> Union(%,"failed") if Integer has INTDOM
--R hash : % -> SingleInteger if Integer has SETCAT
--R inverse : % -> Union(%,"failed") if Integer has FIELD
--R latex : % -> String if Integer has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R listOfLists : % -> List(List(Integer))
--R map : (((Integer,Integer) -> Integer),%,%,Integer) -> %
--R map : (((Integer,Integer) -> Integer),%,%) -> %
--R map : ((Integer -> Integer),%) -> %
--R map! : ((Integer -> Integer),%) -> %

```

```

--R matrix : (NonNegativeInteger,NonNegativeInteger,((Integer,Integer) -> Integer)) -> %
--R member? : (Integer,%) -> Boolean if $ has finiteAggregate and Integer has SETCAT
--R members : % -> List(Integer) if $ has finiteAggregate
--R minordet : % -> Integer if Integer has commutative(*)
--R more? : (%,NonNegativeInteger) -> Boolean
--R new : (NonNegativeInteger,NonNegativeInteger,Integer) -> %
--R nullSpace : % -> List(U16Vector) if Integer has INTDOM
--R nullity : % -> NonNegativeInteger if Integer has INTDOM
--R pfaffian : % -> Integer if Integer has COMRING
--R qelt : (%,Integer,Integer) -> Integer
--R qsetelt! : (%,Integer,Integer,Integer) -> Integer
--R rank : % -> NonNegativeInteger if Integer has INTDOM
--R rowEchelon : % -> % if Integer has EUCDOM
--R scalarMatrix : (NonNegativeInteger,Integer) -> %
--R setColumn! : (%,Integer,U16Vector) -> %
--R setRow! : (%,Integer,U16Vector) -> %
--R setelt : (%,List(Integer),List(Integer),%) -> %
--R setelt : (%,Integer,Integer,Integer) -> Integer
--R setsubMatrix! : (%,Integer,Integer,%) -> %
--R size? : (%,NonNegativeInteger) -> Boolean
--R subMatrix : (%,Integer,Integer,Integer,Integer) -> %
--R swapColumns! : (%,Integer,Integer) -> %
--R swapRows! : (%,Integer,Integer) -> %
--R zero : (NonNegativeInteger,NonNegativeInteger) -> %
--R ~=? : (%,%) -> Boolean if Integer has SETCAT
--R
--E 1

)spool
)lisp (bye)

```

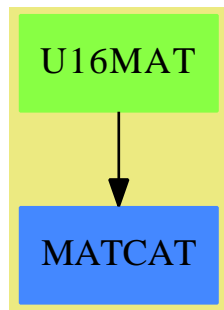
— U16Matrix.help —

U16Matrix examples

This is a low-level domain which implements matrices (two dimensional arrays) of 16-bit integers. Indexing is 0 based, there is no bound checking (unless provided by lower level).

See Also:

- o)show U8Matrix
 - o)show U16Matrix
 - o)show U32Matrix
-

U16Matrix (U16MAT)

See

Exports:

#?	-?	?**?	?*?
?+?	?-?	?/?	?=?
?~=?	antisymmetric?	any?	coerce
column	columnSpace	copy	count
determinant	diagonal?	diagonalMatrix	elt
empty	empty?	eq?	eval
every?	exquo	fill!	hash
horizConcat	inverse	latex	less?
listOfLists	map	map!	matrix
maxColIndex	maxRowIndex	member?	members
minColIndex	minRowIndex	minordet	more?
ncols	new	nrows	nullSpace
nullity	parts	pfaffian	qelt
qnew	qsetelt!	rank	row
rowEchelon	sample	scalarMatrix	setColumn!
setRow!	setelt	setsubMatrix!	size?
square?	squareTop	subMatrix	swapColumns!
swapRows!	symmetric?	transpose	vertConcat
zero			

— domain U16MAT U16Matrix —

```

)abbrev domain U16MAT U16Matrix
++ Author: Waldek Hebisch
++ Description:
++ This is a low-level domain which implements matrices
++ (two dimensional arrays) of 16-bit integers.
++ Indexing is 0 based, there is no bound checking (unless
++ provided by lower level).

```

```

U16Matrix : MatrixCategory(Integer,
                             U16Vector,
                             U16Vector) with
  qnew : (Integer, Integer) -> %
    ++ qnew(n, m) creates a new n by m matrix of zeros.
    ++
    ++X qnew(3,4)$U16Matrix()
== add

R ==> Integer

Qelt2    ==> AREF2U16$Lisp
Qsetelt2 ==> SETAREF2U16$Lisp
Qnrows   ==> ANROWSU16$Lisp
Qncols    ==> ANCOLSU16$Lisp
Qnew      ==> MAKEMATRIXU16$Lisp
Qnew1     ==> MAKEMATRIX1U16$Lisp

minRowIndex x == 0
minColIndex x == 0
nrows x == Qnrows(x)
ncols x == Qncols(x)
maxRowIndex x == Qnrows(x) - 1
maxColIndex x == Qncols(x) - 1

qelt(m, i, j) == Qelt2(m, i, j)
elt(m : %, i : Integer, j : Integer) : R == Qelt2(m, i, j)
qsetelt!(m, i, j, r) == Qsetelt2(m, i, j, r)
setelt(m : %, i : Integer, j : Integer, r : R) == Qsetelt2(m, i, j, r)

empty() == Qnew(0$Integer, 0$Integer)
qnew(rows, cols) == Qnew(rows, cols)
new(rows, cols, a) == Qnew1(rows, cols, a)

```

— U16MAT.dotabb —

```

"U16MAT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=U16MAT"]
"MATCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MATCAT"]
"U16MAT" -> "MATCAT"

```

domain U32MAT U32Matrix

— U32Matrix.input —

```
)set break resume
)sys rm -f U32Matrix.output
)spool U32Matrix.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show U32Matrix
--R U32Matrix is a domain constructor
--R Abbreviation for U32Matrix is U32MAT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for U32MAT
--R
--R----- Operations -----
--R ?? : (U32Vector,%) -> U32Vector      ?? : (%,U32Vector) -> U32Vector
--R ?? : (Integer,%) -> %                ?? : (%,Integer) -> %
--R ?? : (Integer,%) -> %                ?? : (%,%) -> %
--R +? : (%,%) -> %                      -? : % -> %
--R -? : (%,%) -> %                      antisymmetric? : % -> Boolean
--R coerce : U32Vector -> %              column : (%,Integer) -> U32Vector
--R copy : % -> %                        diagonal? : % -> Boolean
--R diagonalMatrix : List(%) -> %        empty : () -> %
--R empty? : % -> Boolean                 eq? : (%,%) -> Boolean
--R fill! : (%,Integer) -> %             horizConcat : (%,%) -> %
--R matrix : List(List(Integer)) -> %    maxColIndex : % -> Integer
--R maxRowIndex : % -> Integer            minColIndex : % -> Integer
--R minRowIndex : % -> Integer            ncols : % -> NonNegativeInteger
--R nrows : % -> NonNegativeInteger       parts : % -> List(Integer)
--R qnew : (Integer,Integer) -> %         row : (%,Integer) -> U32Vector
--R sample : () -> %                     square? : % -> Boolean
--R squareTop : % -> %                   symmetric? : % -> Boolean
--R transpose : % -> %                   transpose : U32Vector -> %
--R vertConcat : (%,%) -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ***? : (%,Integer) -> % if Integer has FIELD
--R ***? : (%,NonNegativeInteger) -> %
--R ?/? : (%,Integer) -> % if Integer has FIELD
--R ?=? : (%,%) -> Boolean if Integer has SETCAT
--R any? : ((Integer -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if Integer has SETCAT
--R columnSpace : % -> List(U32Vector) if Integer has EUCDOM
--R count : (Integer,%) -> NonNegativeInteger if $ has finiteAggregate and Integer has SETCAT
--R count : ((Integer -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
```

```

--R determinant : % -> Integer if Integer has commutative(*)
--R diagonalMatrix : List(Integer) -> %
--R elt : (%,List(Integer),List(Integer)) -> %
--R elt : (%,Integer,Integer,Integer) -> Integer
--R elt : (%,Integer,Integer) -> Integer
--R eval : (%,List(Integer),List(Integer)) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (%,Integer,Integer) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (%,Equation(Integer)) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (%,List(Equation(Integer))) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R every? : ((Integer -> Boolean),%) -> Boolean if $ has finiteAggregate
--R exquo : (%,Integer) -> Union(%, "failed") if Integer has INTDOM
--R hash : % -> SingleInteger if Integer has SETCAT
--R inverse : % -> Union(%, "failed") if Integer has FIELD
--R latex : % -> String if Integer has SETCAT
--R less? : (%,NonNegativeInteger) -> Boolean
--R listOfLists : % -> List(List(Integer))
--R map : (((Integer,Integer) -> Integer),%,%,Integer) -> %
--R map : (((Integer,Integer) -> Integer),%,%) -> %
--R map : ((Integer -> Integer),%) -> %
--R map! : ((Integer -> Integer),%) -> %
--R matrix : (NonNegativeInteger,NonNegativeInteger,((Integer,Integer) -> Integer)) -> %
--R member? : (Integer,%) -> Boolean if $ has finiteAggregate and Integer has SETCAT
--R members : % -> List(Integer) if $ has finiteAggregate
--R minordet : % -> Integer if Integer has commutative(*)
--R more? : (%,NonNegativeInteger) -> Boolean
--R new : (NonNegativeInteger,NonNegativeInteger,Integer) -> %
--R nullSpace : % -> List(U32Vector) if Integer has INTDOM
--R nullity : % -> NonNegativeInteger if Integer has INTDOM
--R pfaffian : % -> Integer if Integer has COMRING
--R qelt : (%,Integer,Integer) -> Integer
--R qsetelt! : (%,Integer,Integer,Integer) -> Integer
--R rank : % -> NonNegativeInteger if Integer has INTDOM
--R rowEchelon : % -> % if Integer has EUCDOM
--R scalarMatrix : (NonNegativeInteger,Integer) -> %
--R setColumn! : (%,Integer,U32Vector) -> %
--R setRow! : (%,Integer,U32Vector) -> %
--R setelt : (%,List(Integer),List(Integer),%) -> %
--R setelt : (%,Integer,Integer,Integer) -> Integer
--R setsubMatrix! : (%,Integer,Integer,%) -> %
--R size? : (%,NonNegativeInteger) -> Boolean
--R subMatrix : (%,Integer,Integer,Integer,Integer) -> %
--R swapColumns! : (%,Integer,Integer) -> %
--R swapRows! : (%,Integer,Integer) -> %
--R zero : (NonNegativeInteger,NonNegativeInteger) -> %
--R ?~=? : (%,%) -> Boolean if Integer has SETCAT
--R
--E 1

)spool
)lisp (bye)

```

— U32Matrix.help —

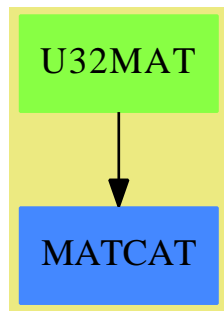
```
=====
U32Matrix examples
=====
```

This is a low-level domain which implements matrices (two dimensional arrays) of 32-bit integers. Indexing is 0 based, there is no bound checking (unless provided by lower level).

See Also:

```
o )show U8Matrix
o )show U16Matrix
o )show U32Matrix
```

U32Matrix (U32MAT)



See

Exports:

#?	-?	***?	?*?
?+?	?-?	?/?	?=?
?^=?	antisymmetric?	any?	coerce
column	columnSpace	copy	count
determinant	diagonal?	diagonalMatrix	elt
empty	empty?	eq?	eval
every?	exquo	fill!	hash
horizConcat	inverse	latex	less?
listOfLists	map	map!	matrix
maxColIndex	maxRowIndex	member?	members
minColIndex	minRowIndex	minordet	more?
ncols	new	nrows	nullSpace
nullity	parts	pfaffian	qelt
qnew	qsetelt!	rank	row
rowEchelon	sample	scalarMatrix	setColumn!
setRow!	setelt	setsubMatrix!	size?
square?	squareTop	subMatrix	swapColumns!
swapRows!	symmetric?	transpose	vertConcat
zero			

— domain U32MAT U32Matrix —

```

)abbrev domain U32MAT U32Matrix
++ Author: Waldek Hebisch
++ Description:
++ This is a low-level domain which implements matrices
++ (two dimensional arrays) of 32-bit integers.
++ Indexing is 0 based, there is no bound checking (unless
++ provided by lower level).
U32Matrix : MatrixCategory(Integer,
                             U32Vector,
                             U32Vector) with
  qnew : (Integer, Integer) -> %
    ++ qnew(n, m) creates a new n by m matrix of zeros.
    ++
    ++X qnew(3,4)$U32Matrix()
== add

R ==> Integer

Qelt2    ==> AREF2U32$Lisp
Qsetelt2 ==> SETAREF2U32$Lisp
Qnrows   ==> ANROWSU32$Lisp
Qncols   ==> ANCOLSU32$Lisp
Qnew     ==> MAKEMATRIXU32$Lisp
Qnew1    ==> MAKEMATRIX1U32$Lisp

```

```

minRowIndex x == 0
minColIndex x == 0
nrows x == Qnrows(x)
ncols x == Qncols(x)
maxRowIndex x == Qnrows(x) - 1
maxColIndex x == Qncols(x) - 1

qelt(m, i, j) == Qelt2(m, i, j)
elt(m : %, i : Integer, j : Integer) : R == Qelt2(m, i, j)
qsetelt!(m, i, j, r) == Qsetelt2(m, i, j, r)
setelt(m : %, i : Integer, j : Integer, r : R) == Qsetelt2(m, i, j, r)

empty() == Qnew(0$Integer, 0$Integer)
qnew(rows, cols) == Qnew(rows, cols)
new(rows, cols, a) == Qnew1(rows, cols, a)

```

— U32MAT.dotabb —

```

"U32MAT" [color="#88FF44",href="bookvol10.3.pdf#nameddest=U32MAT"]
"MATCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MATCAT"]
"U32MAT" -> "MATCAT"

```

domain U8VEC U8Vector

— U8Vector.input —

```

)set break resume
)sys rm -f U8Vector.output
)spool U8Vector.output
)set message test on
)set message auto off
)clear all

```

```

--S 1 of 12
t1:=empty()$U8Vector
--R
--R (1) []
--R
--E 1

```

Type: U8Vector

```

--S 2 of 12
t2:=new(10,10)$U8Vector
--R
--R (2) [10,10,10,10,10,10,10,10,10,10]
--R
--R                                         Type: U8Vector
--E 2

--S 3 of 12
t3:=qelt(t2,2)
--R
--R (3) 10
--R
--R                                         Type: PositiveInteger
--E 3

--S 4 of 12
t4:=elt(t2,2)
--R
--R (4) 10
--R
--R                                         Type: PositiveInteger
--E 4

--S 5 of 12
t5:=t2.2
--R
--R (5) 10
--R
--R                                         Type: PositiveInteger
--E 5

--S 6 of 12
t6:=qsetelt!(t2,2,5)
--R
--R (6) 5
--R
--R                                         Type: PositiveInteger
--E 6

--S 7 of 12
t2.2
--R
--R (7) 5
--R
--R                                         Type: PositiveInteger
--E 7

--S 8 of 12
t7:=setelt(t2,3,6)
--R
--R (8) 6
--R
--R                                         Type: PositiveInteger
--E 8

--S 9 of 12

```

```

t2.3
--R
--R (9) 6
--R
--R                                         Type: PositiveInteger
--E 9

--S 10 of 12
#t2
--R
--R (10) 10
--R
--R                                         Type: PositiveInteger
--E 10

--S 11 of 12
t8:=fill!(t2,7)
--R
--R (11) [7,7,7,7,7,7,7,7,7,7]
--R
--R                                         Type: U8Vector
--E 11

--S 12 of 12
)show U8Vector
--R U8Vector is a domain constructor
--R Abbreviation for U8Vector is U8VEC
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for U8VEC
--R
--R----- Operations -----
--R concat : List(%) -> %                concat : (%,%) -> %
--R concat : (Integer,%) -> %            concat : (%,Integer) -> %
--R construct : List(Integer) -> %       copy : % -> %
--R delete : (%,Integer) -> %            ?? : (%,Integer) -> Integer
--R empty : () -> %                      empty? : % -> Boolean
--R entries : % -> List(Integer)          eq? : (%,%) -> Boolean
--R index? : (Integer,%) -> Boolean       indices : % -> List(Integer)
--R insert : (%,%,Integer) -> %          qelt : (%,Integer) -> Integer
--R reverse : % -> %                     sample : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?<? : (%,%) -> Boolean if Integer has ORDSET
--R ?<=? : (%,%) -> Boolean if Integer has ORDSET
--R ?=? : (%,%) -> Boolean if Integer has SETCAT
--R ?>? : (%,%) -> Boolean if Integer has ORDSET
--R ?>=? : (%,%) -> Boolean if Integer has ORDSET
--R any? : ((Integer -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if Integer has SETCAT
--R convert : % -> InputForm if Integer has KONVERT(INFORM)
--R copyInto! : (%,%,Integer) -> % if $ has shallowlyMutable
--R count : (Integer,%) -> NonNegativeInteger if $ has finiteAggregate and Integer has SETCAT
--R count : ((Integer -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R delete : (%,UniversalSegment(Integer)) -> %

```

```

--R ?.? : (% , UniversalSegment(Integer)) -> %
--R elt : (% , Integer, Integer) -> Integer
--R entry? : (Integer, %) -> Boolean if $ has finiteAggregate and Integer has SETCAT
--R eval : (% , List(Integer), List(Integer)) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (% , Integer, Integer) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (% , Equation(Integer)) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (% , List(Equation(Integer))) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R every? : ((Integer -> Boolean), %) -> Boolean if $ has finiteAggregate
--R fill! : (% , Integer) -> % if $ has shallowlyMutable
--R find : ((Integer -> Boolean), %) -> Union(Integer, "failed")
--R first : % -> Integer if Integer has ORDSET
--R hash : % -> SingleInteger if Integer has SETCAT
--R insert : (Integer, %, Integer) -> %
--R latex : % -> String if Integer has SETCAT
--R less? : (% , NonNegativeInteger) -> Boolean
--R map : (((Integer, Integer) -> Integer), %, %) -> %
--R map : (Integer -> Integer, %) -> %
--R map! : ((Integer -> Integer), %) -> % if $ has shallowlyMutable
--R max : (% , %) -> % if Integer has ORDSET
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (Integer, %) -> Boolean if $ has finiteAggregate and Integer has SETCAT
--R members : % -> List(Integer) if $ has finiteAggregate
--R merge : (% , %) -> % if Integer has ORDSET
--R merge : (((Integer, Integer) -> Boolean), %, %) -> %
--R min : (% , %) -> % if Integer has ORDSET
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (% , NonNegativeInteger) -> Boolean
--R new : (NonNegativeInteger, Integer) -> %
--R parts : % -> List(Integer) if $ has finiteAggregate
--R position : (Integer, %, Integer) -> Integer if Integer has SETCAT
--R position : (Integer, %) -> Integer if Integer has SETCAT
--R position : ((Integer -> Boolean), %) -> Integer
--R qsetelt! : (% , Integer, Integer) -> Integer if $ has shallowlyMutable
--R reduce : (((Integer, Integer) -> Integer), %) -> Integer if $ has finiteAggregate
--R reduce : (((Integer, Integer) -> Integer), %, Integer) -> Integer if $ has finiteAggregate
--R reduce : (((Integer, Integer) -> Integer), %, Integer, Integer) -> Integer if $ has finiteAggregate and Integer has SETCAT
--R remove : ((Integer -> Boolean), %) -> % if $ has finiteAggregate
--R remove : (Integer, %) -> % if $ has finiteAggregate and Integer has SETCAT
--R removeDuplicates : % -> % if $ has finiteAggregate and Integer has SETCAT
--R reverse! : % -> % if $ has shallowlyMutable
--R select : ((Integer -> Boolean), %) -> % if $ has finiteAggregate
--R setelt : (% , UniversalSegment(Integer), Integer) -> Integer if $ has shallowlyMutable
--R setelt : (% , Integer, Integer) -> Integer if $ has shallowlyMutable
--R size? : (% , NonNegativeInteger) -> Boolean
--R sort : % -> % if Integer has ORDSET
--R sort : (((Integer, Integer) -> Boolean), %) -> %
--R sort! : % -> % if $ has shallowlyMutable and Integer has ORDSET
--R sort! : (((Integer, Integer) -> Boolean), %) -> % if $ has shallowlyMutable
--R sorted? : % -> Boolean if Integer has ORDSET
--R sorted? : (((Integer, Integer) -> Boolean), %) -> Boolean

```

```
--R swap! : (%,Integer,Integer) -> Void if $ has shallowlyMutable
--R ?^=? : (%,% ) -> Boolean if Integer has SETCAT
--R
--E 12
```

```
)spool
)lisp (bye)
```

— U8Vector.help —

```
=====
U8Vector examples
=====
```

This is a low-level domain which implements vectors (one dimensional arrays) of unsigned 8-bit numbers. Indexing is 0 based, there is no bound checking (unless provided by lower level).

```
t1:=empty()$U8Vector
[]

t2:=new(10,10)$U8Vector
[10,10,10,10,10,10,10,10,10,10]

t3:=qelt(t2,2)
10

t4:=elt(t2,2)
10

t5:=t2.2
10

t6:=qsetelt!(t2,2,5)
5

t2.2
5

t7:=setelt(t2,3,6)
6

t2.3
6

#t2
10
```

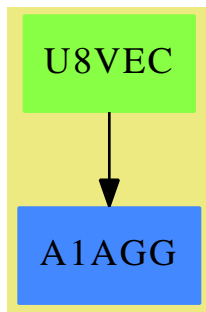
```
t8:=fill!(t2,7)
      [7,7,7,7,7,7,7,7,7]
```

See Also:

- o)show U8Vector
- o)show U16Vector
- o)show U32Vector

—————

U8Vector (U8VEC)



See

⇒ “Segment” (SEG) [20](#) on page [2359](#)

⇒ “SegmentBinding” (SEGBIND) [20](#) on page [2364](#)

Exports:

#?	?.?	??	?<=?	?<?
?=?	?>=?	?>?	?~=?	any?
coerce	concat	construct	convert	copy
copyInto!	count	delete	elt	empty
empty?	entries	entry?	eq?	eval
every?	fill!	find	first	hash
index?	indices	insert	latex	less?
map	map!	max	maxIndex	member?
members	merge	min	minIndex	more?
new	parts	position	qelt	qsetelt!
reduce	remove	removeDuplicates	reverse	reverse!
sample	select	setelt	size?	sort
sort!	sorted?	swap!		

— domain U8VEC U8Vector —


```

)abbrev domain U8VEC U8Vector
++ Author: Waldek Hebisch
++ Description:
++ This is a low-level domain which implements vectors
++ (one dimensional arrays) of unsigned 8-bit numbers. Indexing
++ is 0 based, there is no bound checking (unless provided by
++ lower level).
U8Vector() : OneDimensionalArrayAggregate Integer == add
  Qsize ==> QVLENU8$Lisp
  Qelt ==> ELTU8$Lisp
  Qsetelt ==> SETELTU8$Lisp
  Qnew ==> GETREFVU8$Lisp

#x == Qsize x
++ returns the length of the vector
++
++X t1:=new(10,10)$U8Vector
++X #t1

minIndex x == 0
++ minIndex returns the minimum index of the vector
++
++X t1:=new(10,10)$U8Vector
++X minIndex t1

empty() == Qnew(0$Lisp, 0$Lisp)
++ empty() returns a new vector of length 0
++
++X t1:=empty()$U8Vector

new(n, x) == Qnew (n, x)
++ new(n, x) returns a new vector of length n filled with x
++
++X t1:=new(10,7)$U8Vector

qelt(x, i) == Qelt(x, i)
++ qelt(x, i) returns the i-th element of x
++
++X t1:=new(10,7)$U8Vector
++X qelt(t1,3)

elt(x:%, i:Integer) == Qelt(x, i)
++ elt(x, i) returns the i-th element of x
++
++X t1:=new(10,7)$U8Vector
++X elt(t1,3)

qsetelt_!(x, i, s) == Qsetelt(x, i, s)
++ qsetelt(x, i, s) modifies the i-th element of x to be s
++

```

```

++X t1:=new(10,7)$U8Vector
++X qsetelt!(t1,3,9)

setelt(x:%, i:Integer, s:Integer) == Qsetelt(x, i, s)
++ setelt(x, i, s) modifies the i-th element of x to be s
++
++X t1:=new(10,7)$U8Vector
++X setelt(t1,3,9)

fill_!(x, s) == (for i in 0..(Qsize x) - 1) repeat Qsetelt(x, i, s); x)
++ fill_!(x, s) modifies a vector x so every element has value s
++
++X t1:=new(10,7)$U8Vector
++X fill!(t1,9)

```

— U8VEC.dotabb —

```

"U8VEC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=U8VEC"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"U8VEC" -> "A1AGG"

```

domain U16VEC U16Vector

— U16Vector.input —

```

)set break resume
)sys rm -f U16Vector.output
)spool U16Vector.output
)set message test on
)set message auto off
)clear all

```

```
--S 1 of 12
```

```
t1:=empty()$U16Vector
```

```
--R
```

```
--R (1) []
```

```
--R
```

```
--E 1
```

Type: U16Vector

```
--S 2 of 12
```

```

t2:=new(10,10)$U16Vector
--R
--R (2) [10,10,10,10,10,10,10,10,10,10]
--R
--R                                          Type: U16Vector
--E 2

--S 3 of 12
t3:=qelt(t2,2)
--R
--R (3) 10
--R
--R                                          Type: PositiveInteger
--E 3

--S 4 of 12
t4:=elt(t2,2)
--R
--R (4) 10
--R
--R                                          Type: PositiveInteger
--E 4

--S 5 of 12
t5:=t2.2
--R
--R (5) 10
--R
--R                                          Type: PositiveInteger
--E 5

--S 6 of 12
t6:=qsetelt!(t2,2,5)
--R
--R (6) 5
--R
--R                                          Type: PositiveInteger
--E 6

--S 7 of 12
t2.2
--R
--R (7) 5
--R
--R                                          Type: PositiveInteger
--E 7

--S 8 of 12
t7:=setelt(t2,3,6)
--R
--R (8) 6
--R
--R                                          Type: PositiveInteger
--E 8

--S 9 of 12
t2.3

```

```

--R
--R   (9)  6
--R
--R                                          Type: PositiveInteger
--E 9

--S 10 of 12
#t2
--R
--R   (10)  10
--R
--R                                          Type: PositiveInteger
--E 10

--S 11 of 12
t8:=fill!(t2,7)
--R
--R   (11)  [7,7,7,7,7,7,7,7,7,7]
--R
--R                                          Type: U16Vector
--E 11

--S 12 of 12
)show U16Vector
--R U16Vector is a domain constructor
--R Abbreviation for U16Vector is U16VEC
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for U16VEC
--R
--R----- Operations -----
--R concat : List(%) -> %                concat : (%,%) -> %
--R concat : (Integer,%) -> %            concat : (%,Integer) -> %
--R construct : List(Integer) -> %       copy : % -> %
--R delete : (%,Integer) -> %            ?.? : (%,Integer) -> Integer
--R empty : () -> %                      empty? : % -> Boolean
--R entries : % -> List(Integer)          eq? : (%,%) -> Boolean
--R index? : (Integer,%) -> Boolean        indices : % -> List(Integer)
--R insert : (%,%,Integer) -> %           qelt : (%,Integer) -> Integer
--R reverse : % -> %                     sample : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?<? : (%,%) -> Boolean if Integer has ORDSET
--R ?<=? : (%,%) -> Boolean if Integer has ORDSET
--R ?=? : (%,%) -> Boolean if Integer has SETCAT
--R ?>? : (%,%) -> Boolean if Integer has ORDSET
--R ?>=? : (%,%) -> Boolean if Integer has ORDSET
--R any? : ((Integer -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if Integer has SETCAT
--R convert : % -> InputForm if Integer has KONVERT(INFORM)
--R copyInto! : (%,%,Integer) -> % if $ has shallowlyMutable
--R count : (Integer,%) -> NonNegativeInteger if $ has finiteAggregate and Integer has SETCAT
--R count : ((Integer -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R delete : (%,UniversalSegment(Integer)) -> %
--R ?.? : (%,UniversalSegment(Integer)) -> %

```

```

--R elt : (%,Integer,Integer) -> Integer
--R entry? : (Integer,%) -> Boolean if $ has finiteAggregate and Integer has SETCAT
--R eval : (%,List(Integer),List(Integer)) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (%,Integer,Integer) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (%,Equation(Integer)) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (%,List(Equation(Integer))) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R every? : ((Integer -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (%,Integer) -> % if $ has shallowlyMutable
--R find : ((Integer -> Boolean),%) -> Union(Integer,"failed")
--R first : % -> Integer if Integer has ORDSET
--R hash : % -> SingleInteger if Integer has SETCAT
--R insert : (Integer,%,Integer) -> %
--R latex : % -> String if Integer has SETCAT
--R less? : (%NonNegativeInteger) -> Boolean
--R map : (((Integer,Integer) -> Integer),%,%) -> %
--R map : ((Integer -> Integer),%) -> %
--R map! : ((Integer -> Integer),%) -> % if $ has shallowlyMutable
--R max : (%,% ) -> % if Integer has ORDSET
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (Integer,%) -> Boolean if $ has finiteAggregate and Integer has SETCAT
--R members : % -> List(Integer) if $ has finiteAggregate
--R merge : (%,% ) -> % if Integer has ORDSET
--R merge : (((Integer,Integer) -> Boolean),%,%) -> %
--R min : (%,% ) -> % if Integer has ORDSET
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (%NonNegativeInteger) -> Boolean
--R new : (NonNegativeInteger,Integer) -> %
--R parts : % -> List(Integer) if $ has finiteAggregate
--R position : (Integer,%,Integer) -> Integer if Integer has SETCAT
--R position : (Integer,%) -> Integer if Integer has SETCAT
--R position : ((Integer -> Boolean),%) -> Integer
--R qsetelt! : (%Integer,Integer) -> Integer if $ has shallowlyMutable
--R reduce : (((Integer,Integer) -> Integer),%) -> Integer if $ has finiteAggregate
--R reduce : (((Integer,Integer) -> Integer),%,Integer) -> Integer if $ has finiteAggregate
--R reduce : (((Integer,Integer) -> Integer),%,Integer,Integer) -> Integer if $ has finiteAggregate
--R remove : ((Integer -> Boolean),%) -> % if $ has finiteAggregate
--R remove : (Integer,%) -> % if $ has finiteAggregate and Integer has SETCAT
--R removeDuplicates : % -> % if $ has finiteAggregate and Integer has SETCAT
--R reverse! : % -> % if $ has shallowlyMutable
--R select : ((Integer -> Boolean),%) -> % if $ has finiteAggregate
--R setelt : (%UniversalSegment(Integer),Integer) -> Integer if $ has shallowlyMutable
--R setelt : (%Integer,Integer) -> Integer if $ has shallowlyMutable
--R size? : (%NonNegativeInteger) -> Boolean
--R sort : % -> % if Integer has ORDSET
--R sort : (((Integer,Integer) -> Boolean),%) -> %
--R sort! : % -> % if $ has shallowlyMutable and Integer has ORDSET
--R sort! : (((Integer,Integer) -> Boolean),%) -> % if $ has shallowlyMutable
--R sorted? : % -> Boolean if Integer has ORDSET
--R sorted? : (((Integer,Integer) -> Boolean),%) -> Boolean
--R swap! : (%Integer,Integer) -> Void if $ has shallowlyMutable

```

```
--R ?~=? : (%,% ) -> Boolean if Integer has SETCAT
```

```
--R
```

```
--E 12
```

```
)spool
```

```
)lisp (bye)
```

— U16Vector.help —

```
=====
U16Vector examples
=====
```

This is a low-level domain which implements vectors (one dimensional arrays) of unsigned 16-bit numbers. Indexing is 0 based, there is no bound checking (unless provided by lower level).

```
t1:=empty()$U16Vector
```

```
 []
```

```
t2:=new(10,10)$U16Vector
```

```
 [10,10,10,10,10,10,10,10,10,10]
```

```
t3:=qelt(t2,2)
```

```
 10
```

```
t4:=elt(t2,2)
```

```
 10
```

```
t5:=t2.2
```

```
 10
```

```
t6:=qsetelt!(t2,2,5)
```

```
 5
```

```
t2.2
```

```
 5
```

```
t7:=setelt(t2,3,6)
```

```
 6
```

```
t2.3
```

```
 6
```

```
#t2
```

```
 10
```

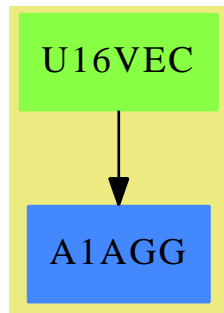
```
t8:=fill!(t2,7)
      [7,7,7,7,7,7,7,7]
```

See Also:

```
o )show U8Vector
o )show U16Vector
o )show U32Vector
```

—————→

U16Vector (U16VEC)



See

⇒ “Segment” (SEG) [20](#) on page [2359](#)

⇒ “SegmentBinding” (SEGBIND) [20](#) on page [2364](#)

Exports:

#?	?.?	?.?	?<=?	?<?
?=?	?>=?	?>?	?~=?	any?
coerce	concat	construct	convert	copy
copyInto!	count	delete	elt	empty
empty?	entries	entry?	eq?	eval
every?	fill!	find	first	hash
index?	indices	insert	latex	less?
map	map!	max	maxIndex	member?
members	merge	min	minIndex	more?
new	parts	position	qelt	qsetelt!
reduce	remove	removeDuplicates	reverse	reverse!
sample	select	setelt	size?	sort
sort!	sorted?	swap!		

— domain U16VEC U16Vector —

```
)abbrev domain U16VEC U16Vector
```

```

++ Author: Waldek Hebisch
++ Description:
++ This is a low-level domain which implements vectors
++ (one dimensional arrays) of unsigned 16-bit numbers. Indexing
++ is 0 based, there is no bound checking (unless provided by
++ lower level).
U16Vector() : OneDimensionalArrayAggregate Integer == add
  Qsize  ==> QVLENU16$Lisp
  Qelt   ==> ELTU16$Lisp
  Qsetelt ==> SETELTU16$Lisp
  Qnew   ==> GETREFVU16$Lisp

#x                                == Qsize x
++ returns the length of the vector
++
++X t1:=new(10,10)$U16Vector
++X #t1

minIndex x                        == 0
++ minIndex returns the minimum index of the vector
++
++X t1:=new(10,10)$U16Vector
++X minIndex t1

empty()                           == Qnew(0$Lisp, 0$Lisp)
++ empty() returns a new vector of length 0
++
++X t1:=empty()$U16Vector

new(n, x)                        == Qnew (n, x)
++ new(n, x) returns a new vector of length n filled with x
++
++X t1:=new(10,7)$U16Vector

qelt(x, i)                       == Qelt(x, i)
++ qelt(x, i) returns the i-th element of x
++
++X t1:=new(10,7)$U16Vector
++X qelt(t1,3)

elt(x:%, i:Integer)              == Qelt(x, i)
++ elt(x, i) returns the i-th element of x
++
++X t1:=new(10,7)$U16Vector
++X elt(t1,3)

qsetelt_!(x, i, s)               == Qsetelt(x, i, s)
++ qsetelt(x, i, s) modifies the i-th element of x to be s
++
++X t1:=new(10,7)$U16Vector

```



```

++X qsetelt!(t1,3,9)

setelt(x:%, i:Integer, s:Integer) == Qsetelt(x, i, s)
++ setelt(x, i, s) modifies the i-th element of x to be s
++
++X t1:=new(10,7)$U16Vector
++X setelt(t1,3,9)

fill_!(x, s) == (for i in 0..(Qsize x) - 1) repeat Qsetelt(x, i, s); x)
++ fill_!(x, s) modifies a vector x so every element has value s
++
++X t1:=new(10,7)$U16Vector
++X fill!(t1,9)

```

— U16VEC.dotabb —

```

"U16VEC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=U16VEC"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"U16VEC" -> "A1AGG"

```

domain U32VEC U32Vector

— U32Vector.input —

```

)set break resume
)sys rm -f U32Vector.output
)spool U32Vector.output
)set message test on
)set message auto off
)clear all

--S 1 of 12
t1:=empty()$U32Vector
--R
--R (1) []
--R
--R Type: U32Vector
--E 1

--S 2 of 12
t2:=new(10,10)$U32Vector

```

```

--R
--R (2) [10,10,10,10,10,10,10,10,10,10,10]
--R
--R                                         Type: U32Vector
--E 2

--S 3 of 12
t3:=qelt(t2,2)
--R
--R (3) 10
--R
--R                                         Type: PositiveInteger
--E 3

--S 4 of 12
t4:=elt(t2,2)
--R
--R (4) 10
--R
--R                                         Type: PositiveInteger
--E 4

--S 5 of 12
t5:=t2.2
--R
--R (5) 10
--R
--R                                         Type: PositiveInteger
--E 5

--S 6 of 12
t6:=qsetelt!(t2,2,5)
--R
--R (6) 5
--R
--R                                         Type: PositiveInteger
--E 6

--S 7 of 12
t2.2
--R
--R (7) 5
--R
--R                                         Type: PositiveInteger
--E 7

--S 8 of 12
t7:=setelt(t2,3,6)
--R
--R (8) 6
--R
--R                                         Type: PositiveInteger
--E 8

--S 9 of 12
t2.3
--R

```

```

--R (9) 6
--R
--R                                         Type: PositiveInteger
--E 9

--S 10 of 12
#t2
--R
--R (10) 10
--R
--R                                         Type: PositiveInteger
--E 10

--S 11 of 12
t8:=fill!(t2,7)
--R
--R (11) [7,7,7,7,7,7,7,7,7,7]
--R
--R                                         Type: U32Vector
--E 11

--S 12 of 12
)show U32Vector
--R
--R U32Vector is a domain constructor
--R Abbreviation for U32Vector is U32VEC
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for U32VEC
--R
--R----- Operations -----
--R concat : List(%) -> %                concat : (%,%) -> %
--R concat : (Integer,%) -> %            concat : (%,Integer) -> %
--R construct : List(Integer) -> %       copy : % -> %
--R delete : (%,Integer) -> %            ?..? : (%,Integer) -> Integer
--R empty : () -> %                      empty? : % -> Boolean
--R entries : % -> List(Integer)          eq? : (%,%) -> Boolean
--R index? : (Integer,%) -> Boolean       indices : % -> List(Integer)
--R insert : (%,%,Integer) -> %          qelt : (%,Integer) -> Integer
--R reverse : % -> %                     sample : () -> %
--R #? : % -> NonNegativeInteger if $ has finiteAggregate
--R ?<? : (%,%) -> Boolean if Integer has ORDSET
--R ?<=? : (%,%) -> Boolean if Integer has ORDSET
--R ?=? : (%,%) -> Boolean if Integer has SETCAT
--R ?>? : (%,%) -> Boolean if Integer has ORDSET
--R ?>=? : (%,%) -> Boolean if Integer has ORDSET
--R any? : ((Integer -> Boolean),%) -> Boolean if $ has finiteAggregate
--R coerce : % -> OutputForm if Integer has SETCAT
--R convert : % -> InputForm if Integer has KONVERT(INFORM)
--R copyInto! : (%,%,Integer) -> % if $ has shallowlyMutable
--R count : (Integer,%) -> NonNegativeInteger if $ has finiteAggregate and Integer has SETCAT
--R count : ((Integer -> Boolean),%) -> NonNegativeInteger if $ has finiteAggregate
--R delete : (%,UniversalSegment(Integer)) -> %
--R ?..? : (%,UniversalSegment(Integer)) -> %

```

```

--R elt : (%,Integer,Integer) -> Integer
--R entry? : (Integer,%) -> Boolean if $ has finiteAggregate and Integer has SETCAT
--R eval : (%,List(Integer),List(Integer)) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (%,Integer,Integer) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (%,Equation(Integer)) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R eval : (%,List(Equation(Integer))) -> % if Integer has EVALAB(INT) and Integer has SETCAT
--R every? : ((Integer -> Boolean),%) -> Boolean if $ has finiteAggregate
--R fill! : (%,Integer) -> % if $ has shallowlyMutable
--R find : ((Integer -> Boolean),%) -> Union(Integer,"failed")
--R first : % -> Integer if Integer has ORDSET
--R hash : % -> SingleInteger if Integer has SETCAT
--R insert : (Integer,%,Integer) -> %
--R latex : % -> String if Integer has SETCAT
--R less? : (%NonNegativeInteger) -> Boolean
--R map : (((Integer,Integer) -> Integer),%,%) -> %
--R map : ((Integer -> Integer),%) -> %
--R map! : ((Integer -> Integer),%) -> % if $ has shallowlyMutable
--R max : (%,% ) -> % if Integer has ORDSET
--R maxIndex : % -> Integer if Integer has ORDSET
--R member? : (Integer,%) -> Boolean if $ has finiteAggregate and Integer has SETCAT
--R members : % -> List(Integer) if $ has finiteAggregate
--R merge : (%,% ) -> % if Integer has ORDSET
--R merge : (((Integer,Integer) -> Boolean),%,%) -> %
--R min : (%,% ) -> % if Integer has ORDSET
--R minIndex : % -> Integer if Integer has ORDSET
--R more? : (%NonNegativeInteger) -> Boolean
--R new : (NonNegativeInteger,Integer) -> %
--R parts : % -> List(Integer) if $ has finiteAggregate
--R position : (Integer,%,Integer) -> Integer if Integer has SETCAT
--R position : (Integer,%) -> Integer if Integer has SETCAT
--R position : ((Integer -> Boolean),%) -> Integer
--R qsetelt! : (%,Integer,Integer) -> Integer if $ has shallowlyMutable
--R reduce : (((Integer,Integer) -> Integer),%) -> Integer if $ has finiteAggregate
--R reduce : (((Integer,Integer) -> Integer),%,Integer) -> Integer if $ has finiteAggregate
--R reduce : (((Integer,Integer) -> Integer),%,Integer,Integer) -> Integer if $ has finiteAggregate and Int
--R remove : ((Integer -> Boolean),%) -> % if $ has finiteAggregate
--R remove : (Integer,%) -> % if $ has finiteAggregate and Integer has SETCAT
--R removeDuplicates : % -> % if $ has finiteAggregate and Integer has SETCAT
--R reverse! : % -> % if $ has shallowlyMutable
--R select : ((Integer -> Boolean),%) -> % if $ has finiteAggregate
--R setelt : (%UniversalSegment(Integer),Integer) -> Integer if $ has shallowlyMutable
--R setelt : (%Integer,Integer) -> Integer if $ has shallowlyMutable
--R size? : (%NonNegativeInteger) -> Boolean
--R sort : % -> % if Integer has ORDSET
--R sort : (((Integer,Integer) -> Boolean),%) -> %
--R sort! : % -> % if $ has shallowlyMutable and Integer has ORDSET
--R sort! : (((Integer,Integer) -> Boolean),%) -> % if $ has shallowlyMutable
--R sorted? : % -> Boolean if Integer has ORDSET
--R sorted? : (((Integer,Integer) -> Boolean),%) -> Boolean
--R swap! : (%Integer,Integer) -> Void if $ has shallowlyMutable

```

```
--R ?~=? : (%,%) -> Boolean if Integer has SETCAT
--R
--E 12
```

```
)spool
)lisp (bye)
```

— U32Vector.help —

```
=====
U32Vector examples
=====
```

This is a low-level domain which implements vectors (one dimensional arrays) of unsigned 32-bit numbers. Indexing is 0 based, there is no bound checking (unless provided by lower level).

```
t1:=empty()$U32Vector
[]

t2:=new(10,10)$U32Vector
[10,10,10,10,10,10,10,10,10,10]

t3:=qelt(t2,2)
10

t4:=elt(t2,2)
10

t5:=t2.2
10

t6:=qsetelt!(t2,2,5)
5

t2.2
5

t7:=setelt(t2,3,6)
6

t2.3
6

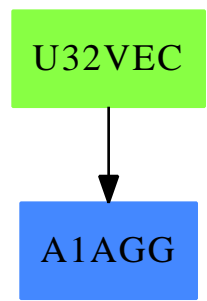
#t2
10
```

```
t8:=fill!(t2,7)
      [7,7,7,7,7,7,7,7,7]

)spool
)lisp (bye)

See Also:
o )show U8Vector
o )show U16Vector
o )show U32Vector
```

U32Vector (U32VEC)



See

- ⇒ “Segment” (SEG) [20](#) on page [2359](#)
- ⇒ “SegmentBinding” (SEGBIND) [20](#) on page [2364](#)

Exports:

#?	?.?	?.?	?<=?	?<?
?=?	?>=?	?>?	?~=?	any?
coerce	concat	construct	convert	copy
copyInto!	count	delete	elt	empty
empty?	entries	entry?	eq?	eval
every?	fill!	find	first	hash
index?	indices	insert	latex	less?
map	map!	max	maxIndex	member?
members	merge	min	minIndex	more?
new	parts	position	qelt	qsetelt!
reduce	remove	removeDuplicates	reverse	reverse!
sample	select	setelt	size?	sort
sort!	sorted?	swap!		

— domain U32VEC U32Vector —

```

)abbrev domain U32VEC U32Vector
++ Author: Waldek Hebisch
++ Description:
++ This is a low-level domain which implements vectors
++ (one dimensional arrays) of unsigned 32-bit numbers. Indexing
++ is 0 based, there is no bound checking (unless provided by
++ lower level).
U32Vector() : OneDimensionalArrayAggregate Integer == add
  Qsize    ==> QVLENU32$Lisp
  Qelt      ==> ELTU32$Lisp
  Qsetelt   ==> SETELTU32$Lisp
  Qnew      ==> GETREFVU32$Lisp

#x                                == Qsize x
++ returns the length of the vector
++
++X t1:=new(10,10)$U32Vector
++X #t1

minIndex x                        == 0
++ minIndex returns the minimum index of the vector
++
++X t1:=new(10,10)$U32Vector
++X minIndex t1

empty()                           == Qnew(0$Lisp, 0$Lisp)
++ empty() returns a new vector of length 0
++
++X t1:=empty()$U32Vector

new(n, x)                         == Qnew (n, x)
++ new(n, x) returns a new vector of length n filled with x
++
++X t1:=new(10,7)$U32Vector

qelt(x, i)                        == Qelt(x, i)
++ qelt(x, i) returns the i-th element of x
++
++X t1:=new(10,7)$U32Vector
++X qelt(t1,3)

elt(x:%, i:Integer)               == Qelt(x, i)
++ elt(x, i) returns the i-th element of x
++
++X t1:=new(10,7)$U32Vector
++X elt(t1,3)

```

```

qsetelt!(x, i, s) == Qsetelt(x, i, s)
++ qsetelt(x, i, s) modifies the i-th element of x to be s
++
++X t1:=new(10,7)$U32Vector
++X qsetelt!(t1,3,9)

setelt(x:%, i:Integer, s:Integer) == Qsetelt(x, i, s)
++ setelt(x, i, s) modifies the i-th element of x to be s
++
++X t1:=new(10,7)$U32Vector
++X setelt(t1,3,9)

fill_(x, s) == (for i in 0..(Qsize x) - 1) repeat Qsetelt(x, i, s); x)
++ fill!(x, s) modifies a vector x so every element has value s
++
++X t1:=new(10,7)$U32Vector
++X fill!(t1,9)

```

— U32VEC.dotabb —

```

"U32VEC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=U32VEC"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"U32VEC" -> "A1AGG"

```

Chapter 23

Chapter V

domain VARIABLE Variable

— Variable.input —

```
)set break resume
)sys rm -f Variable.output
)spool Variable.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Variable
--R
--R Variable(sym: Symbol) is a domain constructor
--R Abbreviation for Variable is VARIABLE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for VARIABLE
--R
--R----- Operations -----
--R ==? : (% ,%) -> Boolean          coerce : % -> Symbol
--R coerce : % -> OutputForm        hash : % -> SingleInteger
--R latex : % -> String             variable : () -> Symbol
--R ~=? : (% ,%) -> Boolean
--R
--E 1

)spool
)lisp (bye)
```

— Variable.help —

```
=====
Variable examples
=====
```

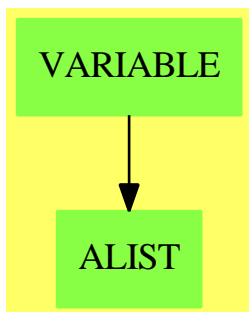
```
This domain implements variables
```

```
See Also:
```

```
o )show Variable
```

```
—————
```

Variable (VARIABLE)

**Exports:**

```
coerce  hash  latex  variable  ??  ?~=?
```

— domain VARIABLE Variable —

```
)abbrev domain VARIABLE Variable
```

```
++ Author: Mark Botch
```

```
++ Description:
```

```
++ This domain implements variables
```

```
Variable(sym:Symbol): Join(SetCategory, CoercibleTo Symbol) with
```

```
  coerce  : % -> Symbol
```

```
    ++ coerce(x) returns the symbol
```

```
  variable: () -> Symbol
```

```
    ++ variable() returns the symbol
```

```
== add
```

```
  coerce(x:%):Symbol      == sym
```

```
  coerce(x:%):OutputForm == sym::OutputForm
```

```

variable()      == sym
x = y           == true
latex(x:%) :String == latex sym

```

— VARIABLE.dotabb —

```

"VARIABLE" [color="#88FF44",href="bookvol10.3.pdf#nameddest=VARIABLE"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"VARIABLE" -> "ALIST"

```

domain VECTOR Vector

— Vector.input —

```

)set break resume
)sys rm -f Vector.output
)spool Vector.output
)set message test on
)set message auto off
)clear all
--S 1 of 11
u : VECTOR INT := new(5,12)
--R
--R
--R (1) [12,12,12,12,12]
--R
--R                                          Type: Vector(Integer)
--E 1

--S 2 of 11
v : VECTOR INT := vector([1,2,3,4,5])
--R
--R
--R (2) [1,2,3,4,5]
--R
--R                                          Type: Vector(Integer)
--E 2

--S 3 of 11
#(v)
--R
--R

```

[illegible]

```

--E 9

--S 10 of 11
v + w
--R
--R
--R (10) [3,5,103,9,11]
--R
--R                                          Type: Vector(Integer)
--E 10

--S 11 of 11
v - w
--R
--R
--R (11) [- 1,- 1,95,- 1,- 1]
--R
--R                                          Type: Vector(Integer)
--E 11
)spool
)lisp (bye)

```

— Vector.help —

=====

Vector examples

=====

The Vector domain is used for storing data in a one-dimensional indexed data structure. A vector is a homogeneous data structure in that all the components of the vector must belong to the same Axiom domain. Each vector has a fixed length specified by the user; vectors are not extensible. This domain is similar to the OneDimensionalArray domain, except that when the components of a Vector belong to a Ring, arithmetic operations are provided.

As with the OneDimensionalArray domain, a Vector can be created by calling the operation new, its components can be accessed by calling the operations elt and qelt, and its components can be reset by calling the operations setelt and qsetelt.

This creates a vector of integers of length 5 all of whose components are 12.

```

u : VECTOR INT := new(5,12)
[12,12,12,12,12]

```

Type: Vector Integer

This is how you create a vector from a list of its components.

```
v : VECTOR INT := vector([1,2,3,4,5])
[1,2,3,4,5]
```

Type: Vector Integer

Indexing for vectors begins at 1. The last element has index equal to the length of the vector, which is computed by #.

```
 #(v)
  5
```

Type: PositiveInteger

This is the standard way to use elt to extract an element. Functionally, it is the same as if you had typed elt(v,2).

```
 v.2
    2
```

Type: PositiveInteger

This is the standard way to use setelt to change an element. It is the same as if you had typed setelt(v,3,99).

```
 v.3 := 99
    99
```

Type: PositiveInteger

Now look at v to see the change. You can use qelt and qsetelt (instead of elt and setelt, respectively) but only when you know that the index is within the valid range.

```
 v
[1,2,99,4,5]
```

Type: Vector Integer

When the components belong to a Ring, Axiom provides arithmetic operations for Vector. These include left and right scalar multiplication.

```
 5 * v
[5,10,495,20,25]
```

Type: Vector Integer

```
 v * 7
[7,14,693,28,35]
```

Type: Vector Integer

```
 w : VECTOR INT := vector([2,3,4,5,6])
[2,3,4,5,6]
```

Type: Vector Integer

Addition and subtraction are also available.

```
v + w  
[3,5,103,9,11]
```

Type: Vector Integer

Of course, when adding or subtracting, the two vectors must have the same length or an error message is displayed.

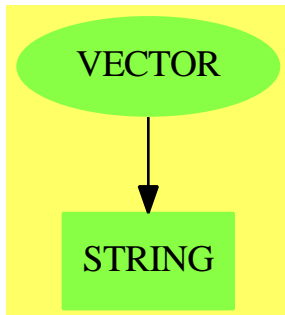
```
v - w  
[- 1,- 1,95,- 1,- 1]
```

Type: Vector Integer

See Also:

- o)help List
- o)help Matrix
- o)help OneDimensionalArray
- o)help Set
- o)help Table
- o)help TwoDimensionalArray
- o)show Vector

Vector (VECTOR)



Exports:

any?	coerce	concat	construct	convert
copy	copyInto!	count	cross	delete
dot	elt	empty	empty?	entries
entry?	eq?	eval	every?	fill!
find	first	hash	index?	indices
insert	latex	length	less?	magnitude
map	map!	max	maxIndex	member?
members	merge	min	minIndex	more?
new	outerProduct	parts	position	qelt
qsetelt!	reduce	remove	removeDuplicates	reverse
reverse!	sample	select	setelt	size?
sort	sort!	sorted?	swap!	vector
zero	#?	?*?	?+?	?-?
?<?	?<=?	?=?	?>?	?>=?
?..?	?~=?	-?	?..?	

— domain VECTOR Vector —

```
)abbrev domain VECTOR Vector
++ Author: Mark Botch
++ Description:
++ This type represents vector like objects with varying lengths
++ and indexed by a finite segment of integers starting at 1.

Vector(R:Type): Exports == Implementation where
  VECTORMININDEX ==> 1      -- if you want to change this, be my guest

Exports ==> VectorCategory R with
  vector: List R -> %
  ++ vector(l) converts the list l to a vector.
Implementation ==>
  IndexedVector(R, VECTORMININDEX) add
  vector l == construct l
  if R has ConvertibleTo InputForm then
    convert(x:%):InputForm ==
      convert [convert("vector"::Symbol)@InputForm,
               convert(parts x)@InputForm]
```

— VECTOR.dotabb —

```
"VECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=VECTOR",
          shape=ellipse]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"VECTOR" -> "STRING"
```



```

--S 5 of 5
% :: PositiveInteger
--R
--R
--RDaly Bug
--R   Cannot convert from type Void to PositiveInteger for value
--R   "()"
--R
--E 5
)spool
)lisp (bye)

```

— Void.help —

=====

Void examples

=====

When an expression is not in a value context, it is given type Void.
For example, in the expression

```
r := (a; b; if c then d else e; f)
```

values are used only from the subexpressions c and f: all others are thrown away. The subexpressions a, b, d and e are evaluated for side-effects only and have type Void. There is a unique value of type Void.

You will most often see results of type Void when you declare a variable.

```
a : Integer
```

Type: Void

Usually no output is displayed for Void results. You can force the display of a rather ugly object by issuing

```
)set message void on
```

```
b : Fraction Integer
```

Type: Void

```
)set message void off
```

All values can be converted to type Void.

```
3::Void
```

Type: Void

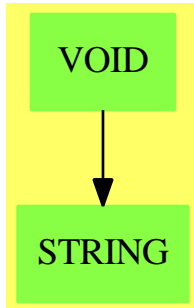
Once a value has been converted to Void, it cannot be recovered.

```
% :: PositiveInteger
  Cannot convert from type Void to PositiveInteger for value "()"
```

See Also:

- o)show Void

Void (VOID)



Exports:

coerce void

— domain VOID Void —

```
)abbrev domain VOID Void
-- These types act as the top and bottom of the type lattice
-- and are known to the compiler and interpreter for type resolution.
++ Author: Stephen M. Watt
++ Date Created: 1986
++ Date Last Updated: May 30, 1991
++ Description:
++ This type is used when no value is needed, e.g., in the \spad{then}
++ part of a one armed \spad{if}.
++ All values can be coerced to type Void. Once a value has been coerced
++ to Void, it cannot be recovered.
```

```
Void: with
  void: () -> %
        ++ void() produces a void object.
  coerce: % -> OutputForm
```

```

++ coerce(v) coerces void object to outputForm.
== add
Rep := String
void()      == voidValue()$Lisp
coerce(v:%) == coerce(v)$Rep

```

— VOID.dotabb —

```

"VOID" [color="#88FF44",href="bookvol10.3.pdf#nameddest=VOID"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"VOID" -> "STRING"

```

Chapter 24

Chapter W

domain WP WeightedPolynomials

— WeightedPolynomials.input —

```
)set break resume
)sys rm -f WeightedPolynomials.output
)spool WeightedPolynomials.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show WeightedPolynomials
--R
--R WeightedPolynomials(R: Ring, VarSet: OrderedSet, E: OrderedAbelianMonoidSup, P: PolynomialCategory(R, E, VarSet))
--R Abbreviation for WeightedPolynomials is WP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for WP
--R
--R----- Operations -----
--R ?? : (% , %) -> %
--R ?? : (PositiveInteger, %) -> %
--R ?? : (% , %) -> %
--R -? : % -> %
--R 1 : () -> %
--R ?? : (% , PositiveInteger) -> %
--R coerce : % -> P
--R coerce : % -> OutputForm
--R latex : % -> String
--R recip : % -> Union(% , "failed")
--R zero? : % -> Boolean
--R ?? : (Integer, %) -> %
--R ??? : (% , PositiveInteger) -> %
--R ?-? : (% , %) -> %
--R ?=? : (% , %) -> Boolean
--R 0 : () -> %
--R coerce : P -> %
--R coerce : Integer -> %
--R hash : % -> SingleInteger
--R one? : % -> Boolean
--R sample : () -> %
--R ?~=? : (% , %) -> Boolean
```

```

--R ??? : (% , R) -> % if R has COMRING
--R ??? : (R , %) -> % if R has COMRING
--R ??? : (NonNegativeInteger , %) -> %
--R ??? : (% , NonNegativeInteger) -> %
--R ?/? : (% , %) -> Union(%, "failed") if R has FIELD
--R ^? : (% , NonNegativeInteger) -> %
--R changeWeightLevel : NonNegativeInteger -> Void
--R characteristic : () -> NonNegativeInteger
--R coerce : R -> % if R has COMRING
--R subtractIfCan : (% , %) -> Union(%, "failed")
--R
--E 1

)spool
)lisp (bye)

```

— WeightedPolynomials.help —

=====

WeightedPolynomials examples

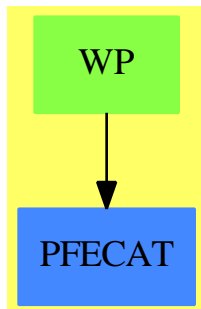
=====

This domain represents truncated weighted polynomials over a general (not necessarily commutative) polynomial type. The variables must be specified, as must the weights. The representation is sparse in the sense that only non-zero terms are represented.

See Also:

o)show WeightedPolynomials

WeightedPolynomials (WP)

**Exports:**

0	1	changeWeightLevel	characteristic
coerce	hash	latex	one?
recip	sample	subtractIfCan	zero?
?~=?	?*?	?**?	?/?
?^?	?*?	?**?	?+?
?-?	-?	?=?	

— domain WP WeightedPolynomials —

```

)abbrev domain WP WeightedPolynomials
++ Author: James Davenport
++ Date Created: 17 April 1992
++ Date Last Updated: 12 July 1992
++ Description:
++ This domain represents truncated weighted polynomials over a general
++ (not necessarily commutative) polynomial type. The variables must be
++ specified, as must the weights.
++ The representation is sparse
++ in the sense that only non-zero terms are represented.

WeightedPolynomials(R:Ring, VarSet: OrderedSet, E:OrderedAbelianMonoidSup,
                    P:PolynomialCategory(R,E,VarSet),
                    vl:List VarSet, wl:List NonNegativeInteger,
                    wtlevel:NonNegativeInteger):

Ring with
  if R has CommutativeRing then Algebra(R)
  coerce: $ -> P
    ++ convert back into a "P", ignoring weights
  if R has Field then "/" : ($,$) -> Union($,"failed")
    ++ x/y division (only works if minimum weight
    ++ of divisor is zero, and if R is a Field)
  coerce: P -> $
    ++ coerce(p) coerces p into Weighted form,
    ++ applying weights and ignoring terms

```



```

changeWeightLevel: NonNegativeInteger -> Void
  ++ changeWeightLevel(n) changes the weight level to
  ++ the new value given:
  ++ NB: previously calculated terms are not affected
==
add
--representations
Rep := PolynomialRing(P,NonNegativeInteger)
p:P
w,x1,x2:$
n:NonNegativeInteger
z:Integer
changeWeightLevel(n) ==
  wtlevel:=n
lookupList:List Record(var:VarSet, weight:NonNegativeInteger)
if #v1 ^= #w1 then error "incompatible length lists in WeightedPolynomial"
lookupList:=[[v,n] for v in v1 for n in w1]
-- local operation
innercoerce:(p,z) -> $
lookup:VarSet -> NonNegativeInteger
lookup v ==
  l:=lookupList
  while l ^= [] repeat
    v = l.first.var => return l.first.weight
    l:=l.rest
  0
innercoerce(p,z) ==
  z<0 => 0
  zero? p => 0
  mv:= mainVariable p
  mv case "failed" => monomial(p,0)
  n:=lookup(mv)
  up:=univariate(p,mv)
  ans:$
  ans:=0
  while not zero? up repeat
    d:=degree up
    f:=n*d
    lcup:=leadingCoefficient up
    up:=up-leadingMonomial up
    mon:=monomial(1,mv,d)
    f<=z =>
      tmp:= innercoerce(lcup,z-f)
      while not zero? tmp repeat
        ans:=ans+ monomial(mon*leadingCoefficient(tmp),degree(tmp)+f)
        tmp:=reductum tmp
  ans
coerce(p):$ == innercoerce(p,wtlevel)
coerce(w):P == "+"/[c for c in coefficients w]
coerce(p:$):OutputForm ==

```

```

zero? p => (0$Integer)::OutputForm
degree p = 0 => leadingCoefficient(p):: OutputForm
reduce("+", (reverse [paren(c::OutputForm) for c in coefficients p])
      ::List OutputForm)
0 == 0$Rep
1 == 1$Rep
x1 = x2 ==
  -- Note that we must strip out any terms greater than wtlevel
  while degree x1 > wtlevel repeat
    x1 := reductum x1
  while degree x2 > wtlevel repeat
    x2 := reductum x2
  x1 =$Rep x2
x1 + x2 == x1 +$Rep x2
-x1 == -(x1::$Rep)
x1 * x2 ==
  -- Note that this is probably an extremely inefficient definition
  w:=x1 *$Rep x2
  while degree(w) > wtlevel repeat
    w:=reductum w
w

```

— WP.dotabb —

```

"WP" [color="#88FF44",href="bookvol10.3.pdf#nameddest=WP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"WP" -> "PFECAT"

```

domain WUTSET WuWenTsunTriangularSet

— WuWenTsunTriangularSet.input —

```

)set break resume
)sys rm -f WuWenTsunTriangularSet.output
)spool WuWenTsunTriangularSet.output
)set message test on
)set message auto off
)clear all
--S 1 of 16
R := Integer

```

```

--R
--R
--R (1) Integer
--R
--R                                          Type: Domain
--E 1

--S 2 of 16
ls : List Symbol := [x,y,z,t]
--R
--R
--R (2) [x,y,z,t]
--R
--R                                          Type: List(Symbol)
--E 2

--S 3 of 16
V := OVAR(ls)
--R
--R
--R (3) OrderedVariableList([x,y,z,t])
--R
--R                                          Type: Domain
--E 3

--S 4 of 16
E := IndexedExponents V
--R
--R
--R (4) IndexedExponents(OrderedVariableList([x,y,z,t]))
--R
--R                                          Type: Domain
--E 4

--S 5 of 16
P := NSMP(R, V)
--R
--R
--R (5)
--R NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--R
--R                                          Type: Domain
--E 5

--S 6 of 16
x: P := 'x
--R
--R
--R (6) x
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--E 6

--S 7 of 16
y: P := 'y
--R

```

```

--R
--R (7) y
--RType: NewSparseMultivariatePolynomial(Integer, OrderedVariableList([x,y,z,t]))
--E 7

--S 8 of 16
z: P := 'z
--R
--R
--R (8) z
--RType: NewSparseMultivariatePolynomial(Integer, OrderedVariableList([x,y,z,t]))
--E 8

--S 9 of 16
t: P := 't
--R
--R
--R (9) t
--RType: NewSparseMultivariatePolynomial(Integer, OrderedVariableList([x,y,z,t]))
--E 9

--S 10 of 16
T := WUTSET(R,E,V,P)
--R
--R
--R (10)
--R WuWenTsunTriangularSet(Integer, IndexedExponents(OrderedVariableList([x,y,z,t]
--R )), OrderedVariableList([x,y,z,t]), NewSparseMultivariatePolynomial(Integer, Ord
--R eredVariableList([x,y,z,t])))
--R
--R Type: Domain
--E 10

--S 11 of 16
p1 := x ** 31 - x ** 6 - x - y
--R
--R
--R (11) 
$$x^{31} - x^6 - x - y$$

--RType: NewSparseMultivariatePolynomial(Integer, OrderedVariableList([x,y,z,t]))
--E 11

--S 12 of 16
p2 := x ** 8 - z
--R
--R
--R (12) 
$$x^8 - z$$

--RType: NewSparseMultivariatePolynomial(Integer, OrderedVariableList([x,y,z,t]))
--E 12

```

```

--S 13 of 16
p3 := x ** 10 - t
--R
--R
--R      10
--R (13) x  - t
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t]))
--E 13

--S 14 of 16
lp := [p1, p2, p3]
--R
--R
--R      31      6      8      10
--R (14) [x  - x  - x - y, x  - z, x  - t]
--RType: List(NewSparseMultivariatePolynomial(Integer,OrderedVariableList([x,y,z,t])))
--E 14

--S 15 of 16
characteristicSet(lp)$T
--R
--R
--R (15)
--R      5      4      4 2 2      3 4      7      4      6      6      3      3      3      3
--R {z  - t , t z y  + 2t z y + (- t  + 2t  - t)z  + t z, (t  - 1)z x - z y - t }
--RType: Union(WuWenTsunTriangularSet(Integer,IndexedExponents(...
--E 15

--S 16 of 16
zeroSetSplit(lp)$T
--R
--R
--R (16)
--R      3      5      4 3      3      2
--R [{t,z,y,x}, {t  - 1, z  - t , z y + t , z x  - t},
--R      5      4      4 2 2      3 4      7      4      6      6      3      3      3      3
--R {z  - t , t z y  + 2t z y + (- t  + 2t  - t)z  + t z, (t  - 1)z x - z y - t }]
--RType: List(WuWenTsunTriangularSet(Integer,IndexedExponents(...
--E 16
)spool
)lisp (bye)

```

— WuWenTsunTriangularSet.help —

```

=====
WuWenTsunTriangularSet examples
=====

```

The WuWenTsunTriangularSet domain constructor implements the characteristic set method of Wu Wen Tsun. This algorithm computes a list of triangular sets from a list of polynomials such that the algebraic variety defined by the given list of polynomials decomposes into the union of the regular-zero sets of the computed triangular sets. The constructor takes four arguments. The first one, R, is the coefficient ring of the polynomials; it must belong to the category IntegralDomain. The second one, E, is the exponent monoid of the polynomials; it must belong to the category OrderedAbelianMonoidSup. The third one, V, is the ordered set of variables; it must belong to the category OrderedSet. The last one is the polynomial ring; it must belong to the category RecursivePolynomialCategory(R,E,V). The abbreviation for WuWenTsunTriangularSet is WUTSET.

Let us illustrate the facilities by an example.

Define the coefficient ring.

```
R := Integer
Integer
Type: Domain
```

Define the list of variables,

```
ls : List Symbol := [x,y,z,t]
[x,y,z,t]
Type: List Symbol
```

and make it an ordered set;

```
V := OVAR(ls)
OrderedVariableList [x,y,z,t]
Type: Domain
```

then define the exponent monoid.

```
E := IndexedExponents V
IndexedExponents OrderedVariableList [x,y,z,t]
Type: Domain
```

Define the polynomial ring.

```
P := NSMP(R, V)
NewSparseMultivariatePolynomial(Integer,OrderedVariableList [x,y,z,t])
Type: Domain
```

Let the variables be polynomial.

```
x: P := 'x
```

```

      x
      Type: NewSparseMultivariatePolynomial(Integer,
        OrderedVariableList [x,y,z,t])

y: P := 'y
  y
      Type: NewSparseMultivariatePolynomial(Integer,
        OrderedVariableList [x,y,z,t])

z: P := 'z
  z
      Type: NewSparseMultivariatePolynomial(Integer,
        OrderedVariableList [x,y,z,t])

t: P := 't
  t
      Type: NewSparseMultivariatePolynomial(Integer,
        OrderedVariableList [x,y,z,t])

```

Now call the WuWenTsunTriangularSet domain constructor.

```

T := WUTSET(R,E,V,P)
WuWenTsunTriangularSet(Integer,IndexedExponents OrderedVariableList [x,y,z,t]
,OrderedVariableList [x,y,z,t],NewSparseMultivariatePolynomial(Integer,Ordere
dVariableList [x,y,z,t]))
      Type: Domain

```

Define a polynomial system.

```

p1 := x ** 31 - x ** 6 - x - y
      31      6
      x  - x  - x - y
      Type: NewSparseMultivariatePolynomial(Integer,
        OrderedVariableList [x,y,z,t])

p2 := x ** 8 - z
      8
      x  - z
      Type: NewSparseMultivariatePolynomial(Integer,
        OrderedVariableList [x,y,z,t])

p3 := x ** 10 - t
      10
      x  - t
      Type: NewSparseMultivariatePolynomial(Integer,
        OrderedVariableList [x,y,z,t])

lp := [p1, p2, p3]
      31      6      8      10
      [x  - x  - x - y, x  - z, x  - t]

```

```
Type: List NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t])
```

Compute a characteristic set of the system.

```
characteristicSet(lp)$T
      5      4 4 2 2      3 4      7      4      6      6      3      3      3      3
{z - t , t z y + 2t z y + (- t + 2t - t)z + t z, (t - 1)z x - z y - t }
      Type: Union(WuWenTsunTriangularSet(Integer,
      IndexedExponents OrderedVariableList [x,y,z,t],
      OrderedVariableList [x,y,z,t],
      NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t])),...)
```

Solve the system.

```
zeroSetSplit(lp)$T
      3      5      4 3      3      2
[{t,z,y,x}, {t - 1,z - t , z y + t , z x - t},
      5      4 4 2 2      3 4      7      4      6      6      3      3      3      3
{z - t , t z y + 2t z y + (- t + 2t - t)z + t z, (t - 1)z x - z y - t }]
      Type: List WuWenTsunTriangularSet(Integer,
      IndexedExponents OrderedVariableList [x,y,z,t],
      OrderedVariableList [x,y,z,t],
      NewSparseMultivariatePolynomial(Integer,
      OrderedVariableList [x,y,z,t]))
```

The RegularTriangularSet and SquareFreeRegularTriangularSet domain constructors, the LazardSetSolvingPackage package constructors as well as, SquareFreeRegularTriangularSet and ZeroDimensionalSolvePackage package constructors also provide operations to compute triangular decompositions of algebraic varieties. These five constructor use a special kind of characteristic sets, called regular triangular sets. These special characteristic sets have better properties than the general ones. Regular triangular sets and their related concepts are presented in the paper "On the Theories of Triangular sets" By P. Aubry, D. Lazard and M. Moreno Maza (to appear in the Journal of Symbolic Computation). The decomposition algorithm (due to the third author) available in the four above constructors provide generally better timings than the characteristic set method. In fact, the WUTSET constructor remains interesting for the purpose of manipulating characteristic sets whereas the other constructors are more convenient for solving polynomial systems.

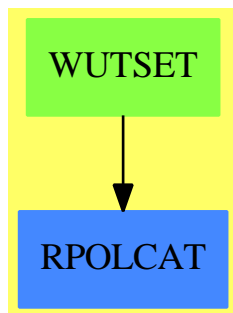
Note that the way of understanding triangular decompositions is detailed in the example of the RegularTriangularSet constructor.

See Also:

- o)help RecursivePolynomialCategory
- o)help RegularTriangularSet


```
o )help SquareFreeRegularTriangularSet
o )help LazardSetSolvingPackage
o )help ZeroDimensionalSolvePackage
o )show WuWenTsunTriangularSet
```

WuWenTsunTriangularSet (WUTSET)



See

⇒ “GeneralTriangularSet” (GTSET) [8](#) on page [1082](#)

Exports:

algebraic?	algebraicVariables
any?	autoReduced?
basicSet	characteristicSerie
characteristicSet	coerce
coHeight	collect
collectQuasiMonic	collectUnder
collectUpper	construct
convert	copy
count	degree
empty	empty?
eq?	eval
every?	extend
extendIfCan	find
first	hash
headReduce	headReduced?
headRemainder	inFRittWu?
initiallyReduce	initiallyReduced?
initials	last
latex	less?
mainVariable?	mainVariables
map	map!
medialSet	member?
members	more?
mvar	normalized?
normalized?	parts
quasiComponent	reduce
reduceByQuasiMonic	reduced?
remainder	remove
removeDuplicates	removeZero
rest	retract
retractIfCan	rewriteIdealWithHeadRemainder
rewriteIdealWithRemainder	rewriteSetWithReduction
roughBase?	roughEqualIdeals?
roughSubIdeal?	roughUnitIdeal?
sample	select
size?	sort
stronglyReduce	stronglyReduced?
triangular?	trivialIdeal?
variables	zeroSetSplit
zeroSetSplitIntoTriangularSystems	#?
?=?	?~=?

— domain WUTSET WuWenTsunTriangularSet —

)abbrev domain WUTSET WuWenTsunTriangularSet

```

++ Author: Marc Moreno Maza (marc@nag.co.uk)
++ Date Created: 11/18/1995
++ Date Last Updated: 12/15/1998
++ References :
++ [1] W. T. WU "A Zero Structure Theorem for polynomial equations solving"
++     MM Research Preprints, 1987.
++ [2] D. M. WANG "An implementation of the characteristic set method in Maple"
++     Proc. DISCO'92. Bath, England.
++ Description:
++ A domain constructor of the category \axiomType{GeneralTriangularSet}.
++ The only requirement for a list of polynomials to be a member of such
++ a domain is the following: no polynomial is constant and two distinct
++ polynomials have distinct main variables. Such a triangular set may
++ not be auto-reduced or consistent. The construct operation
++ does not check the previous requirement. Triangular sets are stored
++ as sorted lists w.r.t. the main variables of their members.
++ Furthermore, this domain exports operations dealing with the
++ characteristic set method of Wu Wen Tsun and some optimizations
++ mainly proposed by Dong Ming Wang.

WuWenTsunTriangularSet(R,E,V,P) : Exports == Implementation where

  R : IntegralDomain
  E : OrderedAbelianMonoidSup
  V : OrderedSet
  P : RecursivePolynomialCategory(R,E,V)
  N ==> NonNegativeInteger
  Z ==> Integer
  B ==> Boolean
  LP ==> List P
  A ==> FiniteEdge P
  H ==> FiniteSimpleHypergraph P
  GPS ==> GeneralPolynomialSet(R,E,V,P)
  RBT ==> Record(bas:$,top:LP)
  RUL ==> Record(chs:Union($,"failed"),rfs:LP)
  pa ==> PolynomialSetUtilitiesPackage(R,E,V,P)
  NLpT ==> SplittingNode(LP,$)
  ALpT ==> SplittingTree(LP,$)
  O ==> OutputForm
  OP ==> OutputPackage

Exports == TriangularSetCategory(R,E,V,P) with

  medialSet : (LP,((P,P)->B),((P,P)->P)) -> Union($,"failed")
  ++ \axiom{medialSet(ps,redOp?,redOp)} returns \axiom{bs} a basic set
  ++ (in Wu Wen Tsun sense w.r.t the reduction-test \axiom{redOp?})
  ++ of some set generating the same ideal as \axiom{ps} (with
  ++ rank not higher than any basic set of \axiom{ps}), if no non-zero
  ++ constant polynomials appear during the computations, else
  ++ \axiom{"failed"} is returned. In the former case, \axiom{bs} has to be

```

```

    ++ understood as a candidate for being a characteristic set of \axiom{ps}.
    ++ In the original algorithm, \axiom{bs} is simply a basic set of \axiom{ps}.
medialSet: LP -> Union($,"failed")
    ++ \axiom{medial(ps)} returns the same as
    ++ \axiom{medialSet(ps,initiallyReduced?,initiallyReduce)}.
characteristicSet : (LP,((P,P)->B),((P,P)->P)) -> Union($,"failed")
    ++ \axiom{characteristicSet(ps,redOp?,redOp)} returns a non-contradictory
    ++ characteristic set of \axiom{ps} in Wu Wen Tsun sense w.r.t the
    ++ reduction-test \axiom{redOp?} (using \axiom{redOp} to reduce
    ++ polynomials w.r.t a \axiom{redOp?} basic set), if no
    ++ non-zero constant polynomial appear during those reductions,
    ++ else \axiom{"failed"} is returned.
    ++ The operations \axiom{redOp} and \axiom{redOp?} must satisfy
    ++ the following conditions: \axiom{redOp?(redOp(p,q),q)} holds
    ++ for every polynomials \axiom{p,q} and there exists an integer
    ++ \axiom{e} and a polynomial \axiom{f} such that we have
    ++ \axiom{init(q)^e*p = f*q + redOp(p,q)}.
characteristicSet: LP -> Union($,"failed")
    ++ \axiom{characteristicSet(ps)} returns the same as
    ++ \axiom{characteristicSet(ps,initiallyReduced?,initiallyReduce)}.
characteristicSerie : (LP,((P,P)->B),((P,P)->P)) -> List $
    ++ \axiom{characteristicSerie(ps,redOp?,redOp)} returns a list \axiom{lbs}
    ++ of triangular sets such that the zero set of \axiom{ps} is the
    ++ union of the regular zero sets of the members of \axiom{lbs}.
    ++ This is made by the Ritt and Wu Wen Tsun process applying
    ++ the operation \axiom{characteristicSet(ps,redOp?,redOp)}
    ++ to compute characteristic sets in Wu Wen Tsun sense.
characteristicSerie: LP -> List $
    ++ \axiom{characteristicSerie(ps)} returns the same as
    ++ \axiom{characteristicSerie(ps,initiallyReduced?,initiallyReduce)}.

Implementation == GeneralTriangularSet(R,E,V,P) add

removeSquares: $ -> Union($,"failed")

Rep ==> LP

rep(s:$):Rep == s pretend Rep
per(l:Rep):$ == l pretend $

removeAssociates (lp:LP):LP ==
  removeDuplicates [primPartElseUnitCanonical(p) for p in lp]

medialSetWithTrace (ps:LP,redOp?:((P,P)->B),redOp:((P,P)->P)):Union(RBT,"failed") ==
  qs := rewriteIdealWithQuasiMonicGenerators(ps,redOp?,redOp)$pa
  contradiction : B := any?(ground?,ps)
  contradiction => "failed":Union(RBT,"failed")
  rs : LP := qs
  bs : $
  while (not empty? rs) and (not contradiction) repeat

```

```

rec := basicSet(rs,redOp?)
contradiction := (rec case "failed")@B
if not contradiction
  then
    bs := (rec::RBT).bas
    rs := (rec::RBT).top
    rs := rewriteIdealWithRemainder(rs,bs)
--    contradiction := ((not empty? rs) and (one? first(rs)))
    contradiction := ((not empty? rs) and (first(rs) = 1))
    if (not empty? rs) and (not contradiction)
      then
        rs := rewriteSetWithReduction(rs,bs,redOp,redOp?)
--        contradiction := ((not empty? rs) and (one? first(rs)))
        contradiction := ((not empty? rs) and (first(rs) = 1))
    if (not empty? rs) and (not contradiction)
      then
        rs := removeDuplicates concat(rs,members(bs))
        rs := rewriteIdealWithQuasiMonicGenerators(rs,redOp?,redOp)$pa
--        contradiction := ((not empty? rs) and (one? first(rs)))
        contradiction := ((not empty? rs) and (first(rs) = 1))
    contradiction => "failed"::Union(RBT,"failed")
    ([bs,qs]$RBT)::Union(RBT,"failed")

medialSet(ps:LP,redOp?:((P,P)->B),redOp:((P,P)->P)) ==
  foo: Union(RBT,"failed") := medialSetWithTrace(ps,redOp?,redOp)
  (foo case "failed") => "failed" :: Union($,"failed")
  ((foo::RBT).bas) :: Union($,"failed")

medialSet(ps:LP) == medialSet(ps,initiallyReduced?,initiallyReduce)

characteristicSetUsingTrace(ps:LP,redOp?:((P,P)->B),redOp:((P,P)->P)):Union($,"failed") ==
  ps := removeAssociates ps
  ps := remove(zero?,ps)
  contradiction : B := any?(ground?,ps)
  contradiction => "failed"::Union($,"failed")
  rs : LP := ps
  qs : LP := ps
  ms : $
  while (not empty? rs) and (not contradiction) repeat
    rec := medialSetWithTrace (qs,redOp?,redOp)
    contradiction := (rec case "failed")@B
    if not contradiction
      then
        ms := (rec::RBT).bas
        qs := (rec::RBT).top
        qs := rewriteIdealWithRemainder(qs,ms)
--        contradiction := ((not empty? qs) and (one? first(qs)))
        contradiction := ((not empty? qs) and (first(qs) = 1))
        if not contradiction
          then

```

```

--      rs := rewriteSetWithReduction(qs,ms,lazyPrem,reduced?)
      contradiction := ((not empty? rs) and (one? first(rs)))
      contradiction := ((not empty? rs) and (first(rs) = 1))
      if (not contradiction) and (not empty? rs)
      then
        qs := removeDuplicates(concat(rs,concat(members(ms),qs)))
      contradiction => "failed":Union($,"failed")
      ms::Union($,"failed")

characteristicSet(ps:LP,redOp?:((P,P)->B),redOp:((P,P)->P)) ==
  characteristicSetUsingTrace(ps,redOp?,redOp)

characteristicSet(ps:LP) == characteristicSet(ps,initiallyReduced?,initiallyReduce)

characteristicSerie(ps:LP,redOp?:((P,P)->B),redOp:((P,P)->P)) ==
  a := [[ps,empty()$$]$NLpT]$ALpT
  while ((esl := extractSplittingLeaf(a)) case ALpT) repeat
    ps := value(value(esl::ALpT)$ALpT)$NLpT
    charSet? := characteristicSetUsingTrace(ps,redOp?,redOp)
    if not (charSet? case $)
    then
      setvalue!(esl::ALpT,[nil()$LP,empty()$$,true]$NLpT)
      updateStatus!(a)
    else
      cs := (charSet?)::$
      lics := initials(cs)
      lics := removeRedundantFactors(lics)$pa
      lics := sort(infRittWu?,lics)
      if empty? lics
      then
        setvalue!(esl::ALpT,[ps,cs,true]$NLpT)
        updateStatus!(a)
      else
        ln : List NLpT := [[nil()$LP,cs,true]$NLpT]
        while not empty? lics repeat
          newps := cons(first(lics),concat(cs::LP,ps))
          lics := rest lics
          newps := removeDuplicates newps
          newps := sort(infRittWu?,newps)
          ln := cons([newps,empty()$$,false]$NLpT,ln)
        splitNodeOf!(esl::ALpT,a,ln)
      remove(empty()$$,conditions(a))

characteristicSerie(ps:LP) == characteristicSerie (ps,initiallyReduced?,initiallyReduce)

if R has GcdDomain
then

  removeSquares (ts:$):Union($,"failed") ==
    empty?(ts)$$ => ts::Union($,"failed")

```

```

p := (first ts)::P
rsts : Union($,"failed")
rsts := removeSquares((rest ts)::P)
not(rsts case $) => "failed"::Union($,"failed")
newts := rsts::P
empty? newts =>
  p := squareFreePart(p)
  (per([primitivePart(p)]$LP))::Union($,"failed")
zero? initiallyReduce(init(p),newts) => "failed"::Union($,"failed")
p := primitivePart(removeZero(p,newts))
ground? p => "failed"::Union($,"failed")
not (mvar(newts) < mvar(p)) => "failed"::Union($,"failed")
p := squareFreePart(p)
(per(cons(unitCanonical(p),rep(newts))))::Union($,"failed")

zeroSetSplit lp ==
  lts : List $ := characteristicSerie(lp,initiallyReduced?,initiallyReduce)
  lts := removeDuplicates(lts)$(List $)
  newlts : List $ := []
  while not empty? lts repeat
    ts := first lts
    lts := rest lts
    iic := removeSquares(ts)
    if iic case $
      then
        newlts := cons(iic::P,newlts)
  newlts := removeDuplicates(newlts)$(List $)
  sort(infRittWu?, newlts)

else

zeroSetSplit lp ==
  lts : List $ := characteristicSerie(lp,initiallyReduced?,initiallyReduce)
  sort(infRittWu?, removeDuplicates lts)

```

— WUTSET.dotabb —

"WUTSET" [color="#88FF44",href="bookvol10.3.pdf#nameddest=WUTSET"]
 "RPOLCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RPOLCAT"]
 "WUTSET" -> "RPOLCAT"

Chapter 25

Chapter X

domain XDPOLY XDistributedPolynomial

Polynomial arithmetic with non-commutative variables has been improved by a contribution of Michel Petitot (University of Lille I, France). The domain constructor **XDistributedPolynomial** provide a distributed representation for these polynomials. It is the non-commutative equivalent for the **DistributedMultivariatePolynomial** constructor.

— XDistributedPolynomial.input —

```
)set break resume
)sys rm -f XDistributedPolynomial.output
)spool XDistributedPolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show XDistributedPolynomial
--R
--R XDistributedPolynomial(v1: OrderedSet,R: Ring) is a domain constructor
--R Abbreviation for XDistributedPolynomial is XDPOLY
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for XDPOLY
--R
--R----- Operations -----
--R ??? : (%,%) -> %          ??? : (v1,%) -> %
--R ??? : (%,R) -> %          ??? : (R,%) -> %
--R ??? : (Integer,%) -> %     ??? : (PositiveInteger,%) -> %
--R ??? : (%,PositiveInteger) -> %  ?+? : (%,%) -> %
--R ?-? : (%,%) -> %          -? : % -> %
--R ?=? : (%,%) -> Boolean     1 : () -> %
```



```

--R 0 : () -> %
--R coef : (%,% ) -> R
--R coerce : Integer -> %
--R coerce : vl -> %
--R constant : % -> R
--R degree : % -> NonNegativeInteger
--R latex : % -> String
--R lquo : (% , vl) -> %
--R map : ((R -> R), %) -> %
--R monomial? : % -> Boolean
--R one? : % -> Boolean
--R quasiRegular? : % -> Boolean
--R reductum : % -> %
--R rquo : (% , %) -> %
--R varList : % -> List(vl)
--R ~=? : (% , %) -> Boolean
--R ?? : (R, OrderedFreeMonoid(vl)) -> %
--R ?? : (NonNegativeInteger, %) -> %
--R ??? : (% , NonNegativeInteger) -> %
--R ?? : (% , NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R coef : (% , OrderedFreeMonoid(vl)) -> R
--R coefficient : (% , OrderedFreeMonoid(vl)) -> R
--R coerce : OrderedFreeMonoid(vl) -> %
--R leadingMonomial : % -> OrderedFreeMonoid(vl)
--R leadingTerm : % -> Record(k: OrderedFreeMonoid(vl), c: R)
--R listOfTerms : % -> List(Record(k: OrderedFreeMonoid(vl), c: R))
--R lquo : (% , OrderedFreeMonoid(vl)) -> %
--R maxdeg : % -> OrderedFreeMonoid(vl)
--R mindeg : % -> OrderedFreeMonoid(vl)
--R mindegTerm : % -> Record(k: OrderedFreeMonoid(vl), c: R)
--R monom : (OrderedFreeMonoid(vl), R) -> %
--R numberOfMonomials : % -> NonNegativeInteger
--R retract : % -> OrderedFreeMonoid(vl)
--R retractIfCan : % -> Union(OrderedFreeMonoid(vl), "failed")
--R rquo : (% , OrderedFreeMonoid(vl)) -> %
--R sh : (% , %) -> % if R has COMRING
--R sh : (% , NonNegativeInteger) -> % if R has COMRING
--R subtractIfCan : (% , %) -> Union(% , "failed")
--R trunc : (% , NonNegativeInteger) -> %
--R
--E 1

)spool
)lisp (bye)

```

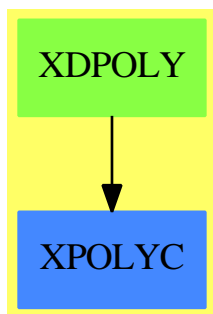
```
=====
XDistributedPolynomial examples
=====
```

This type supports distributed multivariate polynomials whose variables do not commute. The coefficient ring may be non-commutative too. However, coefficients and variables commute.

See Also:

o)show XDistributedPolynomial

XDistributedPolynomial (XDPOLY)



Exports:

0	1	characteristic	coef
coefficient	coefficients	coerce	constant
constant?	degree	hash	latex
leadingCoefficient	listOfTerms	leadingMonomial	leadingTerm
lquo	map	mirror	monomial?
monomials	maxdeg	mindeg	mindegTerm
monom	numberOfMonomials	one?	quasiRegular
quasiRegular?	recip	reductum	retract
retractIfCan	rquo	sample	sh
subtractIfCan	trunc	varList	zero?
?*?	?**?	?+?	?-?
-?	?=?	?^?	?~=?

— domain XDPOLY XDistributedPolynomial —

```
)abbrev domain XDPOLY XDistributedPolynomial
++ Author: Michel Petitot petitot@lifl.fr
++ Date Created: 91
```

```

++ Date Last Updated: 7 Juillet 92
++ Description:
++ This type supports distributed multivariate polynomials
++ whose variables do not commute.
++ The coefficient ring may be non-commutative too.
++ However, coefficients and variables commute.

XDistributedPolynomial(vl:OrderedSet,R:Ring): XDPcat == XDPdef where

WORD ==> OrderedFreeMonoid(vl)
I     ==> Integer
NNI   ==> NonNegativeInteger
TERM  ==> Record(k:WORD, c:R)

XDPcat == Join(FreeModuleCat(R, WORD), XPolynomialsCat(vl,R))

XDPdef == XPolynomialRing(R,WORD) add

import( WORD, TERM)

-- Representation
Rep := List TERM

-- local functions
shw: (WORD , WORD) -> %    -- shuffle de 2 mots

-- definitions

mindegTerm p == last(p)$Rep

if R has CommutativeRing then
  sh(p:%, n:NNI):% ==
    n=0 => 1
    n=1 => p
    n1: NNI := (n-$I 1)::NNI
    sh(p, sh(p,n1))

  sh(p1:%, p2:%) ==
    p:% := 0
    for t1 in p1 repeat
      for t2 in p2 repeat
        p := p + (t1.c * t2.c) * shw(t1.k,t2.k)
    p

  coerce(v: vl):% == coerce(v::WORD)
  v:vl * p:% ==
    [[v * t.k , t.c]$TERM for t in p]

  mirror p ==

```

```

null p => p
monom(mirror$WORD leadingMonomial p, leadingCoefficient p) + _
    mirror reductum p

degree(p) == length(maxdeg(p))$WORD

trunc(p, n) ==
  p = 0 => p
  degree(p) > n => trunc( reductum p , n)
  p

varList p ==
  constant? p => []
  le : List vl := "setUnion"/[varList(t.k) for t in p]
  sort_!(le)

rquo(p:% , w: WORD) ==
  [[r::WORD,t.c]$TERM for t in p | not (r:= rquo(t.k,w)) case "failed" ]
lquo(p:% , w: WORD) ==
  [[r::WORD,t.c]$TERM for t in p | not (r:= lquo(t.k,w)) case "failed" ]
rquo(p:% , v: vl) ==
  [[r::WORD,t.c]$TERM for t in p | not (r:= rquo(t.k,v)) case "failed" ]
lquo(p:% , v: vl) ==
  [[r::WORD,t.c]$TERM for t in p | not (r:= lquo(t.k,v)) case "failed" ]

shw(w1,w2) ==
  w1 = 1$WORD => w2::%
  w2 = 1$WORD => w1::%
  x: vl := first w1 ; y: vl := first w2
  x * shw(rest w1,w2) + y * shw(w1,rest w2)

lquo(p:%,q:%):% ==
  +/ [r * t.c for t in q | (r := lquo(p,t.k)) ^= 0]

rquo(p:%,q:%):% ==
  +/ [r * t.c for t in q | (r := rquo(p,t.k)) ^= 0]

coef(p:%,q:%):R ==
  p = 0 => 0$R
  q = 0 => 0$R
  p.first.k > q.first.k => coef(p.rest,q)
  p.first.k < q.first.k => coef(p,q.rest)
  return p.first.c * q.first.c + coef(p.rest,q.rest)

```

— XPBWPolynomial.input —

```
)set break resume
)sys rm -f XPBWPolynomial.output
)spool XPBWPolynomial.output
)set message test on
)set message auto off
)clear all
--S 1 of 39
a:Symbol := 'a
--R
--R
--R (1) a
--R
--R                                          Type: Symbol
--E 1

--S 2 of 39
b:Symbol := 'b
--R
--R
--R (2) b
--R
--R                                          Type: Symbol
--E 2

--S 3 of 39
RN := Fraction(Integer)
--R
--R
--R (3) Fraction(Integer)
--R
--R                                          Type: Domain
--E 3

--S 4 of 39
word := OrderedFreeMonoid Symbol
--R
--R
--R (4) OrderedFreeMonoid(Symbol)
--R
--R                                          Type: Domain
```

--E 4

--S 5 of 39

lword := LyndonWord(Symbol)

--R

--R

--R (5) LyndonWord(Symbol)

--R

Type: Domain

--E 5

--S 6 of 39

base := PoincareBirkhoffWittLyndonBasis Symbol

--R

--R

--R (6) PoincareBirkhoffWittLyndonBasis(Symbol)

--R

Type: Domain

--E 6

--S 7 of 39

dpoly := XDistributedPolynomial(Symbol, RN)

--R

--R

--R (7) XDistributedPolynomial(Symbol, Fraction(Integer))

--R

Type: Domain

--E 7

--S 8 of 39

rpoly := XRecursivePolynomial(Symbol, RN)

--R

--R

--R (8) XRecursivePolynomial(Symbol, Fraction(Integer))

--R

Type: Domain

--E 8

--S 9 of 39

lpoly := LiePolynomial(Symbol, RN)

--R

--R

--R (9) LiePolynomial(Symbol, Fraction(Integer))

--R

Type: Domain

--E 9

--S 10 of 39

poly := XPBWPolynomial(Symbol, RN)

--R

--R

--R (10) XPBWPolynomial(Symbol, Fraction(Integer))

--R

Type: Domain

--E 10

```

--S 11 of 39
liste : List lword := LyndonWordsList([a,b], 6)
--R
--R
--R (11)
--R      2      2      3      2 2      3      4      3 2
--R [[a], [b], [a b], [a b], [a b ], [a b], [a b ], [a b ], [a b ], [a b ],
--R      2      2 3      2      4      5      4 2      3      3 3
--R [a b a b], [a b ], [a b a b ], [a b ], [a b], [a b ], [a b a b], [a b ],
--R      2      2      2 2      2 4      3      5
--R [a b a b ], [a b a b], [a b ], [a b a b ], [a b ]]
--R
--R                                          Type: List(LyndonWord(Symbol))
--E 11

--S 12 of 39
0$poly
--R
--R
--R (12)  0
--R
--R                                          Type: XPBWPolynomial(Symbol,Fraction(Integer))
--E 12

--S 13 of 39
1$poly
--R
--R
--R (13)  1
--R
--R                                          Type: XPBWPolynomial(Symbol,Fraction(Integer))
--E 13

--S 14 of 39
p : poly := a
--R
--R
--R (14)  [a]
--R
--R                                          Type: XPBWPolynomial(Symbol,Fraction(Integer))
--E 14

--S 15 of 39
q : poly := b
--R
--R
--R (15)  [b]
--R
--R                                          Type: XPBWPolynomial(Symbol,Fraction(Integer))
--E 15

--S 16 of 39
pq: poly := p*q
--R
--R

```

[illegible]

--E 22

--S 23 of 39

leadingTerm pq

--R

--R

--R (23) $[k = [b][a], c = 1]$

--RType: Record(k: PoincareBirkhoffWittLyndonBasis(Symbol), c: Fraction(Integer))

--E 23

--S 24 of 39

degree pq

--R

--R

--R (24) 2

--R

Type: PositiveInteger

--E 24

--S 25 of 39

pq4:=exp(pq,4)

--R

--R

--R (25)

--R
$$1 + [a b] + [b][a] + \frac{1}{2} [a b][a b] + \frac{1}{2} [a b]^2 [a] + \frac{1}{2} [b][a b]^2$$

--R +

--R
$$\frac{3}{2} [b][a b][a] + \frac{1}{2} [b][b][a][a]$$

--R Type: XPBWPolynomial(Symbol, Fraction(Integer))

--E 25

--S 26 of 39

log(pq4,4) - pq

--R

--R

--R (26) 0

--R

Type: XPBWPolynomial(Symbol, Fraction(Integer))

--E 26

--S 27 of 39

lp1 :lpoly := LiePoly liste.10

--R

--R

--R (27) $[a b]^3$

--R

Type: LiePolynomial(Symbol, Fraction(Integer))

--E 27

```

--S 28 of 39
lp2 :lpoly := LiePoly liste.11
--R
--R
--R      2
--R (28) [a b a b]
--R
--R                                          Type: LiePolynomial(Symbol,Fraction(Integer))
--E 28

--S 29 of 39
lp :lpoly := [lp1, lp2]
--R
--R
--R      3 2 2
--R (29) [a b a b a b]
--R
--R                                          Type: LiePolynomial(Symbol,Fraction(Integer))
--E 29

--S 30 of 39
lpd1: dpoly := lp1
--R
--R
--R      3 2      2      2 2      2 2      2      2 2 3
--R (30) a b - 2a b a b - a b a + 4a b a b a - a b a - 2b a b a + b a
--R
--R                                          Type: XDistributedPolynomial(Symbol,Fraction(Integer))
--E 30

--S 31 of 39
lpd2: dpoly := lp2
--R
--R
--R (31)
--R      2      2 2      2      2 2      3      2
--R a b a b - a b a - 3a b a b + 4a b a b a - a b a + 2b a b - 3b a b a
--R +
--R      2
--R b a b a
--R
--R                                          Type: XDistributedPolynomial(Symbol,Fraction(Integer))
--E 31

--S 32 of 39
lpd : dpoly := lpd1 * lpd2 - lpd2 * lpd1
--R
--R
--R (32)
--R      3 2 2      3 2 2 2      3 2      2      3 2      3 2      2 2
--R a b a b a b - a b a b a - 3a b a b a b + 4a b a b a b a - a b a b a
--R +
--R      3 3 3      3 3 2      3 3      2      2      3 2      2      2 2
--R 2a b a b - 3a b a b a + a b a b a - a b a b a b + 3a b a b a b a

```

Type: XDistributedPolynomial(Symbol,Fraction(Integer))

```

--S 33 of 39
lp :: dpoly - lpd
--R
--R
--R (33)  0
--R
--R                                         Type: XDistributedPolynomial(Symbol,Fraction(Integer))
--E 33

--S 34 of 39
p := 3 * lp
--R
--R
--R          3 2 2
--R (34)  3[a b a b a b]
--R
--R                                         Type: XPBWPolynomial(Symbol,Fraction(Integer))
--E 34

--S 35 of 39
q := lp1
--R
--R
--R          3 2
--R (35)  [a b ]
--R
--R                                         Type: XPBWPolynomial(Symbol,Fraction(Integer))
--E 35

--S 36 of 39
pq:= p * q
--R
--R
--R          3 2 2          3 2
--R (36)  3[a b a b a b][a b ]
--R
--R                                         Type: XPBWPolynomial(Symbol,Fraction(Integer))
--E 36

--S 37 of 39
pr:rpoly := p :: rpoly
--R
--R
--R (37)
--R      a
--R      *
--R      a
--R      *
--R      a b b
--R      *
--R      a(a b(a b 3 + b a(- 3)) + b(a(a b(- 9) + b a 12) + b a a(- 3)))
--R      +
--R      b a(a(a b 6 + b a(- 9)) + b a a 3)

```

```

--R      +
--R      b
--R      *
--R      a b
--R      *
--R      a
--R      *
--R      a(a b b(- 3) + b b a 9)
--R      +
--R      b(a(a b 18 + b a(- 36)) + b a a 9)
--R      +
--R      b(a a(a b(- 12) + b a 18) + b a a a(- 3))
--R      +
--R      b a
--R      *
--R      a(a(a b b 3 + b a b(- 9)) + b a a b 9)
--R      +
--R      b(a(a(a b(- 6) + b a 9) + b a a(- 9)) + b a a a 3)
--R      +
--R      b
--R      *
--R      a
--R      *
--R      a b
--R      *
--R      a
--R      *
--R      a(a b b 9 + b(a b(- 18) + b a(- 9)))
--R      +
--R      b(a b a 36 + b a a(- 9))
--R      +
--R      b(a b a a(- 18) + b a a a 9)
--R      +
--R      b a
--R      *
--R      a(a(a b b(- 12) + b a b 36) + b a a b(- 36))
--R      +
--R      b(a(a(a b 24 + b a(- 36)) + b a a 36) + b a a a(- 12))
--R      +
--R      b a a
--R      *
--R      a(a(a b b 3 + b a b(- 9)) + b a a b 9)
--R      +
--R      b(a(a(a b(- 6) + b a 9) + b a a(- 9)) + b a a a 3)
--R      +
--R      b
--R      *
--R      a
--R      *
--R      a

```

```

--R      *
--R      a b
--R      *
--R      a
--R      *
--R      a(a b b(- 6) + b(a b b 12 + b a 6))
--R      +
--R      b(a b a(- 24) + b a a 6)
--R      +
--R      b(a b a a 12 + b a a a(- 6))
--R      +
--R      b a
--R      *
--R      a
--R      *
--R      a(a b b 9 + b(a b(- 18) + b a(- 9)))
--R      +
--R      b(a b a 36 + b a a(- 9))
--R      +
--R      b(a b a a(- 18) + b a a a 9)
--R      +
--R      b a a
--R      *
--R      a(a(a b b(- 3) + b b a 9) + b(a(a b 18 + b a(- 36)) + b a a 9))
--R      +
--R      b(a a(a b(- 12) + b a 18) + b a a a(- 3))
--R      +
--R      b a a a
--R      *
--R      a(a b(a b 3 + b a(- 3)) + b(a(a b(- 9) + b a 12) + b a a(- 3)))
--R      +
--R      b a(a(a b 6 + b a(- 9)) + b a a 3)
--R      Type: XRecursivePolynomial(Symbol,Fraction(Integer))
--E 37

```

--S 38 of 39

qr:rpoly := q :: rpoly

```

--R
--R
--R      (38)
--R      a(a(a b b 1 + b(a b(- 2) + b a(- 1))) + b(a b a 4 + b a a(- 1)))
--R      +
--R      b(a b a a(- 2) + b a a a 1)
--R      Type: XRecursivePolynomial(Symbol,Fraction(Integer))
--E 38

```

--S 39 of 39

pq :: rpoly - pr*qr

```

--R
--R

```

— XPBWPolynomial.help —

```
dpoly := XDistributedPolynomial(Symbol, RN)
XDistributedPolynomial(Symbol, Fraction Integer)
Type: Domain
```

```

rpoly := XRecursivePolynomial(Symbol, RN)
      XRecursivePolynomial(Symbol, Fraction Integer)
                        Type: Domain

lpoly := LiePolynomial(Symbol, RN)
      LiePolynomial(Symbol, Fraction Integer)
                        Type: Domain

poly  := XPBWPolynomial(Symbol, RN)
      XPBWPolynomial(Symbol, Fraction Integer)
                        Type: Domain

liste : List lword := LyndonWordsList([a,b], 6)
      2      2      3      2 2      3      4      3 2
[[a], [b], [a b], [a b], [a b ], [a b ], [a b ], [a b ], [a b ], [a b ],
      2      2 3      2      4      5      4 2      3      3 3
[a b a b], [a b ], [a b a b ], [a b ], [a b ], [a b ], [a b a b], [a b ],
      2      2      2 2      2 4      3      5
[a b a b ], [a b a b], [a b ], [a b a b ], [a b ]]
                        Type: List LyndonWord Symbol

Let's make some polynomials

0$poly
  0
                        Type: XPBWPolynomial(Symbol, Fraction Integer)

1$poly
  1
                        Type: XPBWPolynomial(Symbol, Fraction Integer)

p : poly := a
  [a]
                        Type: XPBWPolynomial(Symbol, Fraction Integer)

q : poly := b
  [b]
                        Type: XPBWPolynomial(Symbol, Fraction Integer)

pq: poly := p*q
  [a b] + [b][a]
                        Type: XPBWPolynomial(Symbol, Fraction Integer)

Coerce to distributed polynomial

pq :: dpoly
  a b
                        Type: XDistributedPolynomial(Symbol, Fraction Integer)

```


Check some polynomial operations

```

mirror pq
  [b][a]
                                     Type: XPBWPolynomial(Symbol,Fraction Integer)

listOfTerms pq
  [[k= [b][a],c= 1],[k= [a b],c= 1]]
                                     Type: List Record(k: PoincareBirkhoffWittLyndonBasis Symbol,
                                                         c: Fraction Integer)

reductum pq
  [a b]
                                     Type: XPBWPolynomial(Symbol,Fraction Integer)

leadingMonomial pq
  [b][a]
                                     Type: PoincareBirkhoffWittLyndonBasis Symbol

coefficients pq
  [1,1]
                                     Type: List Fraction Integer

leadingTerm pq
  [k= [b][a],c= 1]
                                     Type: Record(k: PoincareBirkhoffWittLyndonBasis Symbol,
                                                         c: Fraction Integer)

degree pq
  2
                                     Type: PositiveInteger

pq4:=exp(pq,4)
      1      1      2      1      2
      1 + [a b] + [b][a] + - [a b][a b] + - [a b ][a] + - [b][a b]
      2      2      2      2
+
      3      1
      - [b][a b][a] + - [b][b][a][a]
      2      2
                                     Type: XPBWPolynomial(Symbol,Fraction Integer)

log(pq4,4) - pq
(26) 0
                                     Type: XPBWPolynomial(Symbol,Fraction Integer)

```

Calculations with verification in XDistributedPolynomial.

```

lp1 :lpoly := LiePoly liste.10
      3 2

```

```
[a b ]
Type: LiePolynomial(Symbol,Fraction Integer)

lp2 :lpoly := LiePoly liste.11
      2
[a b a b]
Type: LiePolynomial(Symbol,Fraction Integer)

lp :lpoly := [lp1, lp2]
     3 2 2
[a b a b a b]
Type: LiePolynomial(Symbol,Fraction Integer)

lpd1: dpoly := lp1
      3 2    2        2 2          2 2          2    2 3
a b - 2a b a b - a b a + 4a b a b a - a b a - 2b a b a + b a
Type: XDistributedPolynomial(Symbol,Fraction Integer)

lpd2: dpoly := lp2
      2        2 2          2 2          3        2
a b a b - a b a - 3a b a b + 4a b a b a - a b a + 2b a b - 3b a b a
+
      2
b a b a
Type: XDistributedPolynomial(Symbol,Fraction Integer)

lpd : dpoly := lpd1 * lpd2 - lpd2 * lpd1
      3 2 2       3 2 2 2       3 2    2       3 2           3 2   2 2
a b a b a b - a b a b a - 3a b a b a b + 4a b a b a b a - a b a b a
+
      3 3 3       3 3 2         3 3    2    2       3 2    2       2 2
2a b a b - 3a b a b a + a b a b a - a b a b a b + 3a b a b a b a
+
      2        2        2            2        2 2    2    2 3
6a b a b a b a b - 12a b a b a b a b a + 3a b a b a b a - 4a b a b a b
+
      2    2 2       2    3 3       2 2 4 2       2 2 3           2 2 2    2
6a b a b a b a - a b a b a + a b a b - 3a b a b a b + 3a b a b a b
+
      2 2    3       2 2    2       2 2          2    2 2    2 3
- 2a b a b a b + 3a b a b a b a - 3a b a b a b a + a b a b a
+
      2    3 2       2    2             2    2 2          2
3a b a b a b - 6a b a b a b a b - 3a b a b a b a + 12a b a b a b a b a
+
      2    2 2       2 2    2       2 3 3           4 2
- 3a b a b a b a - 6a b a b a b a + 3a b a b a - 4a b a b a b
+
      3              2    2                3
12a b a b a b a b - 12a b a b a b a b + 8a b a b a b a b
```

```

+
      2      2      2 3      2 5 2
- 12a b a b a b a b a + 12a b a b a b a b a - 4a b a b a b a b a + a b a b
+
      2 4      2 3 2      2 2 3      2 2 2
- 3a b a b a b a b + 3a b a b a b a b - 2a b a b a b a b + 3a b a b a b a b
+
      2 2      2      2 2 2 3      3 3 2      3 2
- 3a b a b a b a b a + a b a b a b a b - 2b a b a b a b a + 4b a b a b a b a
+
      3 2 2      3      3 2 2      3 2 2      3 3 3
2b a b a b a b a - 8b a b a b a b a b a + 2b a b a b a b a b a + 4b a b a b a b a b a - 2b a b a b a b a b a
+
      2 4 2      2 3      2 3 2      2 2
3b a b a b a b a b - 6b a b a b a b a b a b - 3b a b a b a b a b a b + 12b a b a b a b a b a b a
+
      2 2 2 2      2      2      2 2 3      5 2
- 3b a b a b a b a b a - 6b a b a b a b a b a b a + 3b a b a b a b a b a b a - b a b a b a b a b a
+
      4 2      3 2      3      3 2 2
3b a b a b a b a b a + 6b a b a b a b a b a b - 12b a b a b a b a b a b a + 3b a b a b a b a b a b a
+
      2 3      2 2      2 2 3      2 5      2 5 2
- 4b a b a b a b a b a b + 6b a b a b a b a b a b a - b a b a b a b a b a b a + b a b a b a b a b a b a - b a b a b a b a b a b a
+
      2 4 2      2 4      2 4 2 2      2 3 3      2 3 2
- 3b a b a b a b a b a b + 4b a b a b a b a b a b a b - b a b a b a b a b a b a + 2b a b a b a b a b a b a b - 3b a b a b a b a b a b a b
+
      2 3      2
b a b a b a b a b a

```

Type: XDistributedPolynomial(Symbol,Fraction Integer)

```

lp :: dpoly - lpd
0

```

Type: XDistributedPolynomial(Symbol,Fraction Integer)

Calculations with verification in XRecursivePolynomial.

```

p := 3 * lp
      3 2 2
3[a b a b a b]

```

Type: XPBWPolynomial(Symbol,Fraction Integer)

```

q := lp1
      3 2
[a b ]

```

Type: XPBWPolynomial(Symbol,Fraction Integer)

```

pq:= p * q
      3 2 2      3 2

```

```

3[a b a b a b][a b ]
Type: XPBWPolynomial(Symbol,Fraction Integer)

pr:rpoly := p :: rpoly
a
*
  a
  *
    a b b
    *
      a(a b(a b 3 + b a(- 3)) + b(a(a b(- 9) + b a 12) + b a a(- 3)))
      +
        b a(a(a b 6 + b a(- 9)) + b a a 3)
    +
      b
      *
        a b
        *
          a
          *
            a(a b b(- 3) + b b a 9)
            +
              b(a(a b 18 + b a(- 36)) + b a a 9)
          +
            b(a a(a b(- 12) + b a 18) + b a a a(- 3))
        +
          b a
          *
            a(a(a b b 3 + b a b(- 9)) + b a a b 9)
            +
              b(a(a(a b(- 6) + b a 9) + b a a(- 9)) + b a a a 3)
      +
        b
        *
          a
          *
            a b
            *
              a
              *
                a(a b b 9 + b(a b(- 18) + b a(- 9)))
                +
                  b(a b a 36 + b a a(- 9))
              +
                b(a b a a(- 18) + b a a a 9)
            +
              b a
              *
                a(a(a b b(- 12) + b a b 36) + b a a b(- 36))
          +

```

```

      b(a(a(a b 24 + b a(- 36)) + b a a 36) + b a a a(- 12))
+
      b a a
*
      a(a(a b b 3 + b a b(- 9)) + b a a b 9)
+
      b(a(a(a b(- 6) + b a 9) + b a a(- 9)) + b a a a 3)
+
      b
*
      a
*
      a
*
      a b
*
      a
*
      a(a b b(- 6) + b(a b 12 + b a 6))
+
      b(a b a(- 24) + b a a 6)
+
      b(a b a a 12 + b a a a(- 6))
+
      b a
*
      a
*
      a(a b b 9 + b(a b(- 18) + b a(- 9)))
+
      b(a b a 36 + b a a(- 9))
+
      b(a b a a(- 18) + b a a a 9)
+
      b a a
*
      a(a(a b b(- 3) + b b a 9) + b(a(a b 18 + b a(- 36)) + b a a 9))
+
      b(a a(a b(- 12) + b a 18) + b a a a(- 3))
+
      b a a a
*
      a(a b(a b 3 + b a(- 3)) + b(a(a b(- 9) + b a 12) + b a a(- 3)))
+
      b a(a(a b 6 + b a(- 9)) + b a a 3)
Type: XRecursivePolynomial(Symbol,Fraction Integer)

qr:rpoly := q :: rpoly
a(a(a b b 1 + b(a b(- 2) + b a(- 1))) + b(a b a 4 + b a a(- 1)))
+

```

```

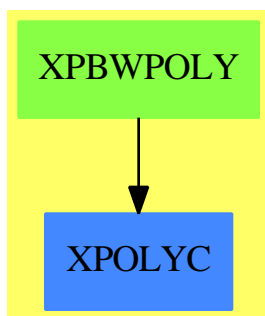
b(a b a a(- 2) + b a a a 1)
Type: XRecursivePolynomial(Symbol,Fraction Integer)

pq :: rpoly - pr*qr
0
Type: XRecursivePolynomial(Symbol,Fraction Integer)

See Also:
o )show XPBWPolynomial

```

XPBWPolynomial (XPBWPOLY)



Exports:

0	1	characteristic	coef
coefficient	coefficients	coerce	constant
constant?	degree	exp	hash
latex	leadingCoefficient	leadingMonomial	leadingTerm
LiePolyIfCan	listOfTerms	log	lquo
map	maxdeg	mindeg	mindegTerm
mirror	monom	monomial?	monomials
numberOfMonomials	one?	product	quasiRegular
quasiRegular?	recip	reductum	retract
retractIfCan	rquo	sample	sh
subtractIfCan	trunc	varList	zero?
?*?	?**?	?+?	?-?
~?	?=?	?^?	?~=?

— domain XPBWPOLY XPBWPolynomial —

```

)abbrev domain XPBWPOLY XPBWPolynomial
++ Author: Michel Petitot (petitot@lifl.fr).

```

```

++ Date Created: 91
++ Date Last Updated: 7 Juillet 92
++ Description:
++ This domain constructor implements polynomials in non-commutative
++ variables written in the Poincare-Birkhoff-Witt basis from the
++ Lyndon basis.
++ These polynomials can be used to compute Baker-Campbell-Hausdorff
++ relations.

XPBWPolynomial(VarSet:OrderedSet,R:CommutativeRing): XDPcat == XDPdef where

WORD    ==> OrderedFreeMonoid(VarSet)
LWORD   ==> LyndonWord(VarSet)
LWORDS  ==> List LWORD
BASIS   ==> PoincareBirkhoffWittLyndonBasis(VarSet)
TERM    ==> Record(k:BASIS, c:R)
LTERMS  ==> List(TERM)
LPOLY   ==> LiePolynomial(VarSet,R)
EX      ==> OutputForm
XDPOLY  ==> XDistributedPolynomial(VarSet,R)
XRPOLY  ==> XRecursivePolynomial(VarSet,R)
TERM1   ==> Record(k:LWORD, c:R)
NNI     ==> NonNegativeInteger
I       ==> Integer
RN      ==> Fraction(Integer)

XDPcat == Join(XPolynomialsCat(VarSet,R), FreeModuleCat(R, BASIS)) with
  coerce      : LPOLY -> $
  ++ \axiom{coerce(p)} returns \axiom{p}.
  coerce      : $ -> XDPOLY
  ++ \axiom{coerce(p)} returns \axiom{p} as a distributed polynomial.
  coerce      : $ -> XRPOLY
  ++ \axiom{coerce(p)} returns \axiom{p} as a recursive polynomial.
  LiePolyIfCan: $ -> Union(LPOLY,"failed")
  ++ \axiom{LiePolyIfCan(p)} return \axiom{p} if \axiom{p} is a Lie polynomial.
  product     : ($,$,NNI) -> $ -- produit tronque a l'ordre n
  ++ \axiom{product(a,b,n)} returns \axiom{a*b} (truncated up to order \axiom{n}).

if R has Module(RN) then
  exp      : ($,NNI) -> $
  ++ \axiom{exp(p,n)} returns the exponential of \axiom{p}
  ++ (truncated up to order \axiom{n}).
  log      : ($,NNI) -> $
  ++ \axiom{log(p,n)} returns the logarithm of \axiom{p}
  ++ (truncated up to order \axiom{n}).

XDPdef == FreeModule1(R,BASIS) add
  import(TERM)

-- Representation

```

```

Rep:= LTERMS

-- local functions
prod1: (BASIS, $) -> $
prod2: ($, BASIS) -> $
prod : (BASIS, BASIS) -> $

prod11: (BASIS, $, NNI) -> $
prod22: ($, BASIS, NNI) -> $

outForm : TERM -> EX
Dexpand : BASIS -> XDPOLY
Rexpand : BASIS -> XRPOLY
process : (List LWORD, LWORD, List LWORD) -> $
mirror1 : BASIS -> $

-- functions locales
outForm t ==
  t.c =$R 1 => t.k :: EX
  t.k =$BASIS 1 => t.c :: EX
  t.c::EX * t.k ::EX

prod1(b:BASIS, p:$):$ ==
  +/ [t.c * prod(b, t.k) for t in p]

prod2(p:$, b:BASIS):$ ==
  +/ [t.c * prod(t.k, b) for t in p]

prod11(b,p,n) ==
  limit: I := n -$I length b
  +/ [t.c * prod(b, t.k) for t in p | length(t.k) :: I <= limit]

prod22(p,b,n) ==
  limit: I := n -$I length b
  +/ [t.c * prod(t.k, b) for t in p | length(t.k) :: I <= limit]

prod(g,d) ==
  d = 1 => monom(g,1)
  g = 1 => monom(d,1)
  process(reverse listOfTerms g, first d, rest listOfTerms d)

Dexpand b ==
  b = 1 => 1$XDPOLY
  */ [LiePoly(1)$LPOLY :: XDPOLY for l in listOfTerms b]

Rexpand b ==
  b = 1 => 1$XRPOLY
  */ [LiePoly(1)$LPOLY :: XRPOLY for l in listOfTerms b]

mirror1(b:BASIS):$ ==

```



```

b = 1 => 1
lp: LPOLY := LiePoly first b
lp := mirror lp
mirror1(rest b) * lp :: $

process(gauche, x, droite) == -- algo du "collect process"
  null gauche => monom( cons(x, droite) pretend BASIS, 1$R)
  r1, r2 : $
  not lexico(first gauche, x) => -- cas facile !!!
    monom(append(reverse gauche, cons(x, droite)) pretend BASIS , 1$R)

p: LPOLY := [first gauche , x] -- on crochete !!!
null droite =>
  r1 := +/ [t.c * process(rest gauche, t.k, droite) for t in _
    listOfTerms p]
  r2 := process( rest gauche, x, list first gauche)
  r1 + r2
rd: List LWORD := rest droite; fd: LWORD := first droite
r1 := +/ [t.c * process(list t.k, fd, rd) for t in listOfTerms p]
r1 := +/ [t.c * process(rest gauche, first t.k, rest listOfTerms(t.k))_
  for t in r1]
r2 := process([first gauche, x], fd, rd)
r2 := +/ [t.c * process(rest gauche, first t.k, rest listOfTerms(t.k))_
  for t in r2]
r1 + r2

-- definitions
1 == monom(1$BASIS, 1$R)

coerce(r:R):$ == [[1$BASIS , r]$TERM ]

coerce(p:$):EX ==
  null p => (0$R) :: EX
  le : List EX := nil
  for rec in p repeat le := cons(outForm rec, le)
  reduce(_+, le)$List(EX)

coerce(v: VarSet):$ == monom(v::$BASIS , 1$R)
coerce(p: LPOLY):$ ==
  [[t.k :: BASIS , t.c ]$TERM for t in listOfTerms p]

coerce(p:$):XDPOLY ==
  +/ [t.c * Dexpand t.k for t in p]

coerce(p:$):XRPOLY ==
  p = 0 => 0$XRPOLY
  +/ [t.c * Rexpand t.k for t in p]

constant? p == (null p) or (leadingMonomial(p) =$BASIS 1)
constant p ==

```

```

null p => 0$R
p.last.k = 1$BASIS => p.last.c
0$R

quasiRegular? p == (p=0) or (p.last.k ^= 1$BASIS)
quasiRegular p ==
  p = 0 => p
  p.last.k = 1$BASIS => delete(p, maxIndex p)
  p

x:$ * y:$ ==
  y = 0$$ => 0
  +/ [t.c * prod1(t.k, y) for t in x]

--      listOfTerms p == p pretend LTERMS

varList p ==
  lv: List VarSet := "setUnion"/ [varList(b.k)$BASIS for b in p]
  sort(lv)

degree(p) ==
  p=0 => error "null polynomial"
  length(leadingMonomial p)

trunc(p, n) ==
  p = 0 => p
  degree(p) > n => trunc( reductum p , n)
  p

product(x,y,n) ==
  x = 0 => 0
  y = 0 => 0
  +/ [t.c * prod11(t.k, y, n) for t in x]

if R has Module(RN) then
  exp (p,n) ==
    p = 0 => 1
    not quasiRegular? p =>
      error "a proper polynomial is required"
    s : $ := 1 ; r: $ := 1 -- resultat
    for i in 1..n repeat
      k1 :RN := 1/i
      k2 : R := k1 * 1$R
      s := k2 * product(p, s, n)
      r := r + s
    r

log (p,n) ==
  p = 1 => 0
  p1: $ := 1 - p

```

```

not quasiRegular? p1 =>
  error "constant term <> 1, impossible log "
s : $ := - 1 ; r: $ := 0 -- resultat
for i in 1..n repeat
  k1 :RN := 1/i
  k2 : R := k1 * 1$R
  s := product(p1, s, n)
  r := k2 * s + r
r

LiePolyIfCan p ==
p = 0 => 0$LPOLY
"and"/ [retractable?(t.k)$BASIS for t in p] =>
  lt : List TERM1 := _
    [[retract(t.k)$BASIS, t.c]$TERM1 for t in p]
  lt pretend LPOLY
"failed"

mirror p ==
+ / [t.c * mirror1(t.k) for t in p]

-----

— XPBWPOLY.dotabb —

"XPBWPOLY" [color="#88FF44",href="bookvol10.3.pdf#nameddest=XPBWPOLY"]
"XPOLYC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=XPOLYC"]
"XPBWPOLY" -> "XPOLYC"

-----

```

domain XPOLY XPolynomial

```

— XPolynomial.input —

)set break resume
)sys rm -f XPolynomial.output
)spool XPolynomial.output
)set message test on
)set message auto off
)clear all
--S 1 of 14
poly := XPolynomial(Integer)
--R

```

```

--R
--R (1) XPolynomial(Integer)
--R
--R                                          Type: Domain
--E 1

--S 2 of 14
pr: poly := 2*x + 3*y-5
--R
--R
--R (2) - 5 + x 2 + y 3
--R
--R                                          Type: XPolynomial(Integer)
--E 2

--S 3 of 14
pr2: poly := pr*pr
--R
--R
--R (3) 25 + x(- 20 + x 4 + y 6) + y(- 30 + x 6 + y 9)
--R
--R                                          Type: XPolynomial(Integer)
--E 3

--S 4 of 14
pd := expand pr
--R
--R
--R (4) - 5 + 2x + 3y
--R
--R                                          Type: XDistributedPolynomial(Symbol,Integer)
--E 4

--S 5 of 14
pd2 := pd*pd
--R
--R
--R (5) 25 - 20x - 30y + 4x2 + 6x y + 6y x + 9y2
--R
--R                                          Type: XDistributedPolynomial(Symbol,Integer)
--E 5

--S 6 of 14
expand(pr2) - pd2
--R
--R
--R (6) 0
--R
--R                                          Type: XDistributedPolynomial(Symbol,Integer)
--E 6

--S 7 of 14
qr := pr**3
--R
--R

```

```

--R (7)
--R      - 125 + x(150 + x(- 60 + x 8 + y 12) + y(- 90 + x 12 + y 18))
--R      +
--R      y(225 + x(- 90 + x 12 + y 18) + y(- 135 + x 18 + y 27))
--R                                          Type: XPolynomial(Integer)
--E 7

--S 8 of 14
qd := pd**3
--R
--R
--R (8)
--R      2      2      3      2
--R      - 125 + 150x + 225y - 60x  - 90x y - 90y x - 135y  + 8x  + 12x y + 12x y x
--R      +
--R      2      2      2      3
--R      18x y  + 12y x  + 18y x y + 18y x + 27y
--R                                          Type: XDistributedPolynomial(Symbol,Integer)
--E 8

--S 9 of 14
trunc(qd,2)
--R
--R
--R      2      2
--R (9) - 125 + 150x + 225y - 60x  - 90x y - 90y x - 135y
--R                                          Type: XDistributedPolynomial(Symbol,Integer)
--E 9

--S 10 of 14
trunc(qr,2)
--R
--R
--R (10) - 125 + x(150 + x(- 60) + y(- 90)) + y(225 + x(- 90) + y(- 135))
--R                                          Type: XPolynomial(Integer)
--E 10

--S 11 of 14
Word := OrderedFreeMonoid Symbol
--R
--R
--R (11) OrderedFreeMonoid(Symbol)
--R
--R                                          Type: Domain
--E 11

--S 12 of 14
w: Word := x*y**2
--R
--R
--R      2

```

XPolynomial examples

```
poly := XPolynomial(Integer)
      XPolynomial Integer
                                     Type: Domain
```

Define a first polynomial,

```
pr: poly := 2*x + 3*y-5
      - 5 + x 2 + y 3
                                         Type: XPolynomial Integer
```

and a second one.

```
pr2: poly := pr*pr
      25 + x(- 20 + x 4 + y 6) + y(- 30 + x 6 + y 9)
                                         Type: XPolynomial Integer
```

Rewrite pr in a distributive way,

```
pd := expand pr
      - 5 + 2x + 3y
                                         Type: XDistributedPolynomial(Symbol,Integer)
```

compute its square,

```
pd2 := pd*pd
      25 - 20x - 30y + 4x2 + 6x y + 6y x + 9y2
                                         Type: XDistributedPolynomial(Symbol,Integer)
```

and checks that:

```
expand(pr2) - pd2
      0
                                         Type: XDistributedPolynomial(Symbol,Integer)
```

We define:

```
qr := pr**3
      - 125 + x(150 + x(- 60 + x 8 + y 12) + y(- 90 + x 12 + y 18))
      +
      y(225 + x(- 90 + x 12 + y 18) + y(- 135 + x 18 + y 27))
                                         Type: XPolynomial Integer
```

and:

```
qd := pd**3
      - 125 + 150x + 225y - 60x2 - 90x y - 90y x - 135y2 + 8x3 + 12x2 y + 12x y x
      +
      18x y2 + 12y x2 + 18y x y + 18y x2 + 27y3
                                         Type: XDistributedPolynomial(Symbol,Integer)
```

We truncate qd at degree 3.

```
trunc(qd,2)
      2      2
- 125 + 150x + 225y - 60x - 90x y - 90y x - 135y
      Type: XDistributedPolynomial(Symbol,Integer)
```

The same for qr:

```
trunc(qr,2)
- 125 + x(150 + x(- 60) + y(- 90)) + y(225 + x(- 90) + y(- 135))
      Type: XPolynomial Integer
```

We define:

```
Word := OrderedFreeMonoid Symbol
      OrderedFreeMonoid Symbol
      Type: Domain
```

and:

```
w: Word := x*y**2
      2
      x y
      Type: OrderedFreeMonoid Symbol
```

We can compute the right-quotient of qr by r:

```
rquo(qr,w)
      18
      Type: XPolynomial Integer
```

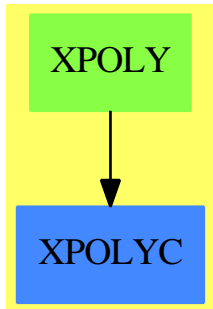
and the shuffle-product of pr by r:

```
sh(pr,w::poly)
x(x y y 4 + y(x y 2 + y(- 5 + x 2 + y 9))) + y x y y 3
      Type: XPolynomial Integer
```

See Also:

```
o )help XPBWPolynomial
o )help LiePolynomial
o )help XDistributedPolynomial
o )help XRecursivePolynomial
o )help XPolynomialRing
o )show XPolynomial
```

XPolynomial (XPOLY)



Exports:

0	1	characteristic	coef	coerce
constant	constant?	degree	expand	hash
latex	lquo	map	maxdeg	mindeg
mindegTerm	mirror	monom	monomial?	RemainderList
one?	quasiRegular	quasiRegular?	recip	retract
retractIfCan	rquo	sample	sh	subtractIfCan
trunc	unexpand	varList	zero?	?*?
?**?	?+?	?-?	-?	?=?
?^?	?~=?			

— domain XPOLY XPolynomial —

```

)abbrev domain XPOLY XPolynomial
++ Author: Michel Petitot petitot@lifl.fr
++ Date Created: 91
++ Date Last Updated: 7 Juillet 92
++ Description:
++ This type supports multivariate polynomials whose set of variables
++ is \spadtype{Symbol}. The representation is recursive.
++ The coefficient ring may be non-commutative and the variables
++ do not commute. However, coefficients and variables commute.

```

```

XPolynomial(R:Ring) == XRecursivePolynomial(Symbol, R)

```

—————

— XPOLY.dotabb —

```

"XPOLY" [color="#88FF44",href="bookvol10.3.pdf#nameddest=XPOLY"]
"XPOLYC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=XPOLYC"]
"XPOLY" -> "XPOLYC"

```

domain XPR XPolynomialRing

— XPolynomialRing.input —

```

)set break resume
)sys rm -f XPolynomialRing.output
)spool XPolynomialRing.output
)set message test on
)set message auto off
)clear all
--S 1 of 15
Word := OrderedFreeMonoid(Symbol)
--R
--R
--R (1) OrderedFreeMonoid(Symbol)
--R
--R                                          Type: Domain
--E 1

--S 2 of 15
poly:= XPR(Integer,Word)
--R
--R
--R (2) XPolynomialRing(Integer,OrderedFreeMonoid(Symbol))
--R
--R                                          Type: Domain
--E 2

--S 3 of 15
p:poly := 2 * x - 3 * y + 1
--R
--R
--R (3) 1 + 2x - 3y
--R
--R                                          Type: XPolynomialRing(Integer,OrderedFreeMonoid(Symbol))
--E 3

--S 4 of 15
q:poly := 2 * x + 1
--R
--R
--R (4) 1 + 2x
--R
--R                                          Type: XPolynomialRing(Integer,OrderedFreeMonoid(Symbol))
--E 4

--S 5 of 15
p + q
--R

```

```

--R
--R (5)  $x^2 + 4x - 3y$ 
--R                                         Type: XPolynomialRing(Integer,OrderedFreeMonoid(Symbol))
--E 5

--S 6 of 15
p * q
--R
--R
--R
--R (6)  $x^2 + 4x - 3y + 4x^2 - 6y x$ 
--R                                         Type: XPolynomialRing(Integer,OrderedFreeMonoid(Symbol))
--E 6

--S 7 of 15
(p+q)**2-p**2-q**2-2*p*q
--R
--R
--R (7)  $-6x y + 6y x$ 
--R                                         Type: XPolynomialRing(Integer,OrderedFreeMonoid(Symbol))
--E 7

--S 8 of 15
M := SquareMatrix(2,Fraction Integer)
--R
--R
--R (8) SquareMatrix(2,Fraction(Integer))
--R                                         Type: Domain
--E 8

--S 9 of 15
poly1:= XPR(M,Word)
--R
--R
--R (9)
--R XPolynomialRing(SquareMatrix(2,Fraction(Integer)),OrderedFreeMonoid(Symbol))
--R                                         Type: Domain
--E 9

--S 10 of 15
m1:M := matrix [ [i*j**2 for i in 1..2] for j in 1..2]
--R
--R
--R
--R (10)  $\begin{array}{cc} +1 & 2+ \\ | & | \\ +4 & 8+ \end{array}$ 
--R                                         Type: SquareMatrix(2,Fraction(Integer))
--E 10

--S 11 of 15

```

```

m2:M := m1 - 5/4
--R
--R
--R      + 1 +
--R      | - - 2 |
--R      | 4 |
--R      (11) | |
--R      | 27|
--R      | 4 --|
--R      + 4+
--R
--R                                          Type: SquareMatrix(2,Fraction(Integer))
--E 11

--S 12 of 15
m3: M := m2**2
--R
--R
--R      +129 +
--R      |--- 13 |
--R      | 16 |
--R      (12) | |
--R      | 857|
--R      |26 ---|
--R      + 16+
--R
--R                                          Type: SquareMatrix(2,Fraction(Integer))
--E 12

--S 13 of 15
pm:poly1 := m1*x + m2*y + m3*z - 2/3
--R
--R
--R      + 2 + + 1 + +129 +
--R      | - - 0 | | - - 2 | | --- 13 |
--R      | 3 | +1 2+ | 4 | | 16 |
--R      (13) | | + | | x + | | y + | | z
--R      | 2| +4 8+ | 27| | 857|
--R      | 0 - -| | 4 --| |26 ---|
--R      + 3+ + 4+ + 16+
--R
--R                                          Type: XPolynomialRing(SquareMatrix(2,Fraction(Integer)),OrderedFreeMonoid(Symbol))
--E 13

--S 14 of 15
qm:poly1 := pm - m1*x
--R
--R
--R      + 2 + + 1 + +129 +
--R      | - - 0 | | - - 2 | | --- 13 |
--R      | 3 | | 4 | | 16 |
--R      (14) | | + | | y + | | z
--R      | 2| | 27| | 857|

```

```

--R      | 0  - -|  | 4  --|  |26  ---|
--R      +      3+  +      4+  +      16+
--RType: XPolynomialRing(SquareMatrix(2,Fraction(Integer)),OrderedFreeMonoid(Symbol))
--E 14

--S 15 of 15
qm**3
--R
--R
--R (15)
--R      + 8      + + 1 8+      +43 52 +      + 129      +
--R      |- -- 0 |  |- - -|  |- -- -- |  |- --- - 26 |
--R      | 27      |  | 3 3|  | 4 3 |  | 8      | 2
--R      |      | + |  |y + |  |z + |  |y
--R      |      8|  |16 |  |104 857|  | 857|
--R      | 0  - --|  |- 9|  |--- ---|  |- 52 - ---|
--R      +      27+  + 3  +      + 3 12+  +      8 +
--R      +
--R      + 3199      831 +      + 3199      831 +      + 103169      6409 +
--R      |- ---- - --- |  |- ---- - --- |  |- ---- - ---- |
--R      | 32      4 |  | 32      4 |  | 128      4 | 2
--R      |      |y z + |  |z y + |  |z
--R      | 831      26467|  | 831      26467|  | 6409      820977|
--R      |- --- - ----|  |- --- - ----|  |- --- - ----|
--R      + 2      32 +      + 2      32 +      + 2      128 +
--R      +
--R      +3199 831 +      +103169 6409 +      +103169 6409 +
--R      |---- --- |  |----- ---- |  |----- ---- |
--R      | 64 8 | 3 | 256 8 | 2 | 256 8 |
--R      |      |y + |  |y z + |  |y z y
--R      |831 26467|  | 6409 820977|  | 6409 820977|
--R      |--- ----|  | ---- - ----|  | ---- - ----|
--R      + 4      64 +      + 4      256 +      + 4      256 +
--R      +
--R      +3178239 795341 +      +103169 6409 +      +3178239 795341 +
--R      |----- ---- |  |----- ---- |  |----- ---- |
--R      | 1024      128 | 2 | 256 8 | 2 | 1024      128 |
--R      |      |y z + |  |z y + |  |z y z
--R      |795341 25447787|  | 6409 820977|  |795341 25447787|
--R      |----- ----|  | ---- - ----|  |----- ----|
--R      + 64      1024 +      + 4      256 +      + 64      1024 +
--R      +
--R      +3178239 795341 +      +98625409 12326223 +
--R      |----- ---- |  |----- ---- |
--R      | 1024      128 | 2 | 4096 256 | 3
--R      |      |z y + |  |z
--R      |795341 25447787|  |12326223 788893897|
--R      |----- ----|  |----- ----|
--R      + 64      1024 +      + 128      4096 +
--RType: XPolynomialRing(SquareMatrix(2,Fraction(Integer)),OrderedFreeMonoid(Symbol))

```

```
--E 15
)spool
)lisp (bye)
```

— XPolynomialRing.help —

=====

XPolynomialRing examples

=====

The XPolynomialRing domain constructor implements generalized polynomials with coefficients from an arbitrary Ring (not necessarily commutative) and whose exponents are words from an arbitrary OrderedMonoid (not necessarily commutative too). Thus these polynomials are (finite) linear combinations of words.

This constructor takes two arguments. The first one is a Ring and the second is an OrderedMonoid. The abbreviation for XPolynomialRing is XPR.

Other constructors like XPolynomial, XRecursivePolynomial, XDistributedPolynomial, LiePolynomial and XPBWPolynomial implement multivariate polynomials in non-commutative variables.

We illustrate now some of the facilities of the XPR domain constructor.

Define the free ordered monoid generated by the symbols.

```
Word := OrderedFreeMonoid(Symbol)
      OrderedFreeMonoid Symbol
                                     Type: Domain
```

Define the linear combinations of these words with integer coefficients.

```
poly:= XPR(Integer,Word)
      XPolynomialRing(Integer,OrderedFreeMonoid Symbol)
                                     Type: Domain
```

Then we define a first element from poly.

```
p:poly := 2 * x - 3 * y + 1
      1 + 2x - 3y
                                     Type: XPolynomialRing(Integer,OrderedFreeMonoid Symbol)
```

And a second one.

```
q:poly := 2 * x + 1
      1 + 2x
```

```
Type: XPolynomialRing(Integer,OrderedFreeMonoid Symbol)
```

We compute their sum,

$$\begin{array}{r} p + q \\ 2 + 4x - 3y \end{array}$$

```
Type: XPolynomialRing(Integer,OrderedFreeMonoid Symbol)
```

their product,

$$\begin{array}{r} p * q \\ 1 + 4x - 3y + 4x^2 - 6y x \end{array}$$

```
Type: XPolynomialRing(Integer,OrderedFreeMonoid Symbol)
```

and see that variables do not commute.

$$\begin{array}{r} (p+q)**2-p**2-q**2-2*p*q \\ - 6x y + 6y x \end{array}$$

```
Type: XPolynomialRing(Integer,OrderedFreeMonoid Symbol)
```

Now we define a ring of square matrices,

```
M := SquareMatrix(2,Fraction Integer)
SquareMatrix(2,Fraction Integer)
Type: Domain
```

and the linear combinations of words with these matrices as coefficients.

```
poly1:= XPR(M,Word)
XPolynomialRing(SquareMatrix(2,Fraction Integer),OrderedFreeMonoid Symbol)
Type: Domain
```

Define a first matrix,

```
m1:M := matrix [ [i*j**2 for i in 1..2] for j in 1..2]
+1 2+
|  |
+4 8+
Type: SquareMatrix(2,Fraction Integer)
```

a second one,

```
m2:M := m1 - 5/4
+ 1 +
|- - 2 |
| 4 |
|  |
| 27|
| 4 --|
```

+ 4+

Type: SquareMatrix(2,Fraction Integer)

and a third one.

```
m3: M := m2**2
      +129      +
      |--- 13 |
      | 16      |
      |          |
      |      857|
      |26  ---|
      +      16+
```

Type: SquareMatrix(2,Fraction Integer)

Define a polynomial,

```
pm:poly1 := m1*x + m2*y + m3*z - 2/3
      + 2      +      + 1      +      +129      +
      |- - 0 |      |- - 2 |      |--- 13 |
      | 3      |      +1 2+      | 4      |      | 16      |
      |          | + |      |x + |      |y + |      |z
      |          2|      +4 8+      | 27|      | 857|
      | 0      - -|      | 4      --|      |26  ---|
      +      3+      +      4+      +      16+
```

Type: XPolynomialRing(SquareMatrix(2,Fraction Integer),
OrderedFreeMonoid Symbol)

a second one,

```
qm:poly1 := pm - m1*x
      + 2      +      + 1      +      +129      +
      |- - 0 |      |- - 2 |      |--- 13 |
      | 3      |      | 4      |      | 16      |
      |          | + |      |y + |      |z
      |          2|      | 27|      | 857|
      | 0      - -|      | 4      --|      |26  ---|
      +      3+      +      4+      +      16+
```

Type: XPolynomialRing(SquareMatrix(2,Fraction Integer),
OrderedFreeMonoid Symbol)

and the following power.

```
qm**3
      + 8      +      + 1 8+      +43 52 +      + 129      +
      |- -- 0 |      |- - -|      |-- --|      |- --- - 26 |
      | 27      |      | 3 3|      | 4      3 |      | 8      | 2
      |          | + |      |y + |      |z + |      |y
      |          8|      |16      |      |104 857|      | 857|
      | 0      - --|      |-- 9|      |--- ---|      |- 52 - ---|
```



```

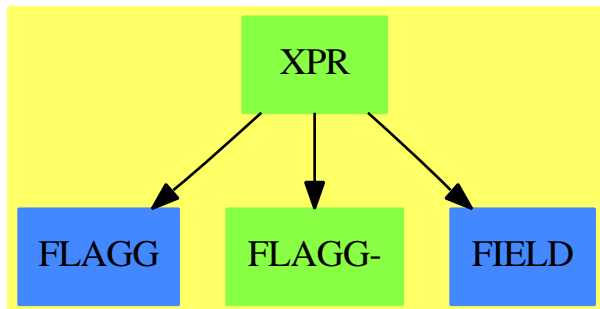
+      27+  + 3  +  + 3  12+  +      8 +
+
+ 3199      831 +      + 3199      831 +      + 103169      6409 +
|- ---- - --- |      |- ---- - --- |      |- ---- - ---- |
| 32      4 |      | 32      4 |      | 128      4 | 2
|      |y z + |      |      |z y + |      |      |z
| 831      26467|      | 831      26467|      | 6409      820977|
|- --- - ----|      |- --- - ----|      | - ---- - ----|
+ 2      32 +      + 2      32 +      + 2      128 +
+
+3199 831 +      +103169 6409 +      +103169 6409 +
|---- --- |      |----- --- |      |----- --- |
| 64 8 | 3 | 256 8 | 2 | 256 8 |
|      |y + |      |      |y z + |      |      |y z y
|831 26467|      | 6409 820977|      | 6409 820977|
|--- ----|      | --- ----|      | --- ----|
+ 4      64 +      + 4      256 +      + 4      256 +
+
+3178239 795341 +      +103169 6409 +      +3178239 795341 +
|----- ---- |      |----- ---- |      |----- ---- |
| 1024 128 | 2 | 256 8 | 2 | 1024 128 |
|      |y z + |      |      |z y + |      |      |z y z
|795341 25447787|      | 6409 820977|      |795341 25447787|
|----- ----|      | --- ----|      | --- ----|
+ 64      1024 +      + 4      256 +      + 64      1024 +
+
+3178239 795341 +      +98625409 12326223 +
|----- ---- |      |----- ---- |
| 1024 128 | 2 | 4096 256 | 3
|      |z y + |      |      |z
|795341 25447787|      |12326223 788893897|
|----- ----|      |----- ----|
+ 64      1024 +      + 128      4096 +
Type: XPolynomialRing(SquareMatrix(2,Fraction Integer),
OrderedFreeMonoid Symbol)

```

See Also:

- o)help XPBWPolynomial
- o)help LiePolynomial
- o)help XDistributedPolynomial
- o)help XRecursivePolynomial
- o)help XPolynomial
- o)show XPolynomialRing

XPolynomialRing (XPR)



Exports:

0	1	characteristic	coef
coefficient	coefficients	coerce	constant
constant?	hash	latex	leadingCoefficient
leadingMonomial	leadingTerm	listOfTerms	map
maxdeg	mindeg	monom	monomial?
monomials	numberOfMonomials	one?	quasiRegular
quasiRegular?	recip	reductum	retract
retractIfCan	sample	subtractIfCan	zero?
#?	?*?	?**?	?+?
?-?	-?	?=?	?^?
?~=?			

— domain XPR XPolynomialRing —

```

)abbrev domain XPR XPolynomialRing
++ Author: Michel Petitot petitot@lifl.fr
++ Date Created: 91
++ Date Last Updated: 7 Juillet 92
++ Description:
++ This domain represents generalized polynomials with coefficients
++ (from a not necessarily commutative ring), and words
++ belonging to an arbitrary \spadtype{OrderedMonoid}.
++ This type is used, for instance, by the \spadtype{XDistributedPolynomial}
++ domain constructor where the Monoid is free.

```

```

XPolynomialRing(R:Ring,E:OrderedMonoid): T == C where
  TERM ==> Record(k: E, c: R)
  EX    ==> OutputForm
  NNI   ==> NonNegativeInteger

```

```

T == Join(Ring, XAlgebra(R), FreeModuleCat(R,E)) with
--operations
  "*": (% ,R) -> %

```

```

    ++ \spad{p*r} returns the product of \spad{p} by \spad{r}.
"#": % -> NonNegativeInteger
    ++ \spad{# p} returns the number of terms in \spad{p}.
coerce: E -> %
    ++ \spad{coerce(e)} returns \spad{1*e}
maxdeg: % -> E
    ++ \spad{maxdeg(p)} returns the greatest word occurring in the polynomial \spad{p}
    ++ with a non-zero coefficient. An error is produced if \spad{p} is zero.
mindeg: % -> E
    ++ \spad{mindeg(p)} returns the smallest word occurring in the polynomial \spad{p}
    ++ with a non-zero coefficient. An error is produced if \spad{p} is zero.
reductum : % -> %
    ++ \spad{reductum(p)} returns \spad{p} minus its leading term.
    ++ An error is produced if \spad{p} is zero.
coef : (% ,E) -> R
    ++ \spad{coef(p,e)} extracts the coefficient of the monomial \spad{e}.
    ++ Returns zero if \spad{e} is not present.
constant?:% -> Boolean
    ++ \spad{constant?(p)} tests whether the polynomial \spad{p} belongs to the
    ++ coefficient ring.
constant: % -> R
    ++ \spad{constant(p)} return the constant term of \spad{p}.
quasiRegular? : % -> Boolean
    ++ \spad{quasiRegular?(x)} return true if \spad{constant(p)} is zero.
quasiRegular : % -> %
    ++ \spad{quasiRegular(x)} return \spad{x} minus its constant term.
map : (R -> R, %) -> %
    ++ \spad{map(fn,x)} returns \spad{Sum(fn(r_i) w_i)} if \spad{x} writes \spad{Sum(r_i w_i)}
if R has Field then "/" : (% ,R) -> %
    ++ \spad{p/r} returns \spad{p*(1/r)}.

--assertions
if R has noZeroDivisors then noZeroDivisors
if R has unitsKnown then unitsKnown
if R has canonicalUnitNormal then canonicalUnitNormal
    ++ canonicalUnitNormal guarantees that the function
    ++ unitCanonical returns the same representative for all
    ++ associates of any particular element.

C == FreeModule1(R,E) add
--representations
Rep:= List TERM
--uses
repeatMultExpt: (% ,NonNegativeInteger) -> %
--define
1 == [[1$E,1$R]]

characteristic == characteristic$R
#x == #$Rep x

```

```

maxdeg p == if null p then error " polynome nul !"
              else p.first.k
mindeg p == if null p then error " polynome nul !"
              else (last p).k

coef(p,e) ==
  for tm in p repeat
    tm.k=e => return tm.c
    tm.k < e => return 0$R
  0$R

constant? p == (p = 0) or (maxdeg(p) = 1$E)
constant p == coef(p,1$E)

quasiRegular? p == (p=0) or (last p).k ^= 1$E
quasiRegular p ==
  quasiRegular?(p) => p
  [t for t in p | not(t.k = 1$E)]

recip(p) ==
  p=0 => "failed"
  p.first.k > 1$E => "failed"
  (u:=recip(p.first.c)) case "failed" => "failed"
  (u::R)::%

coerce(r:R) == if r=0$R then 0$% else [[1$E,r]]
coerce(n:Integer) == (n::R)::%

if R has noZeroDivisors then
  p1:% * p2:% ==
    null p1 => 0
    null p2 => 0
    p1.first.k = 1$E => p1.first.c * p2
    p2 = 1 => p1
    -- +/[[[t1.k*t2.k,t1.c*t2.c]$TERM for t2 in p2]
    -- for t1 in reverse(p1)]
    +/[[[t1.k*t2.k,t1.c*t2.c]$TERM for t2 in p2]
      for t1 in p1]
else
  p1:% * p2:% ==
    null p1 => 0
    null p2 => 0
    p1.first.k = 1$E => p1.first.c * p2
    p2 = 1 => p1
    -- +/[[[t1.k*t2.k,r]$TERM for t2 in p2 | not (r:=t1.c*t2.c) =$R 0]
    -- for t1 in reverse(p1)]
    +/[[[t1.k*t2.k,r]$TERM for t2 in p2 | not (r:=t1.c*t2.c) =$R 0]
      for t1 in p1]
p:% ** nn:NNI == repeatMultExpt(p,nn)
repeatMultExpt(x,nn) ==

```

```

nn = 0 => 1
y:% := x
for i in 2..nn repeat y:= x * y
y

outTerm(r:R, m:E):EX ==
  r=1 => m::EX
  m=1 => r::EX
  r::EX * m::EX

--  coerce(x:%) : EX ==
--  null x => (0$R) :: EX
--  le : List EX := nil
--  for rec in x repeat
--    rec.c = 1$R => le := cons(rec.k :: EX, le)
--    rec.k = 1$E => le := cons(rec.c :: EX, le)
--    le := cons(mkBinary("":EX,rec.c :: EX,
--      rec.k :: EX), le)
--  1 = #le => first le
--  mkNary("+ " :: EX,le)

coerce(a:%):EX ==
  empty? a => (0$R)::EX
  reduce(_+, reverse_! [outTerm(t.c, t.k) for t in a])$List(EX)

if R has Field then
  x/r == inv(r)*x

-----

— XPR.dotabb —

"XPR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=XPR"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"XPR" -> "FLAGG"
"XPR" -> "FLAGG-"
"XPR" -> "FIELD"

-----

```

domain XRPOLY XRecursivePolynomial

— XRecursivePolynomial.input —

```

)set break resume
)sys rm -f XRecursivePolynomial.output
)spool XRecursivePolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show XRecursivePolynomial
--R
--R XRecursivePolynomial(VarSet: OrderedSet,R: Ring) is a domain constructor
--R Abbreviation for XRecursivePolynomial is XRPOLY
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.3.pamphlet to see algebra source code for XRPOLY
--R
--R----- Operations -----
--R ??? : (VarSet,%) -> %           ??? : (%,R) -> %
--R ??? : (R,%) -> %               ??? : (%,%) -> %
--R ??? : (Integer,%) -> %         ??? : (PositiveInteger,%) -> %
--R ??? : (%,PositiveInteger) -> % ?+? : (%,%) -> %
--R ?-? : (%,%) -> %               -? : % -> %
--R ?? : (%,%) -> Boolean          1 : () -> %
--R 0 : () -> %                    ?? : (%,PositiveInteger) -> %
--R coef : (%,%) -> R              coerce : VarSet -> %
--R coerce : R -> %                coerce : Integer -> %
--R coerce : % -> OutputForm       constant : % -> R
--R constant? : % -> Boolean        degree : % -> NonNegativeInteger
--R hash : % -> SingleInteger       latex : % -> String
--R lquo : (%,%) -> %              lquo : (%,VarSet) -> %
--R map : ((R -> R),%) -> %         mirror : % -> %
--R monomial? : % -> Boolean        one? : % -> Boolean
--R quasiRegular : % -> %          quasiRegular? : % -> Boolean
--R recip : % -> Union(%, "failed") rquo : (%,%) -> %
--R rquo : (%,VarSet) -> %          sample : () -> %
--R varList : % -> List(VarSet)     zero? : % -> Boolean
--R ?~=? : (%,%) -> Boolean
--R ??? : (NonNegativeInteger,%) -> %
--R ??? : (%,NonNegativeInteger) -> %
--R RemainderList : % -> List(Record(k: VarSet,c: %))
--R ??? : (%,NonNegativeInteger) -> %
--R characteristic : () -> NonNegativeInteger
--R coef : (%,OrderedFreeMonoid(VarSet)) -> R
--R coerce : OrderedFreeMonoid(VarSet) -> %
--R expand : % -> XDistributedPolynomial(VarSet,R)

```

```

--R lquo : (% , OrderedFreeMonoid(VarSet)) -> %
--R maxdeg : % -> OrderedFreeMonoid(VarSet)
--R mindeg : % -> OrderedFreeMonoid(VarSet)
--R mindegTerm : % -> Record(k: OrderedFreeMonoid(VarSet), c: R)
--R monom : (OrderedFreeMonoid(VarSet), R) -> %
--R retract : % -> OrderedFreeMonoid(VarSet)
--R retractIfCan : % -> Union(OrderedFreeMonoid(VarSet), "failed")
--R rquo : (% , OrderedFreeMonoid(VarSet)) -> %
--R sh : (% , NonNegativeInteger) -> % if R has COMRING
--R sh : (% , %) -> % if R has COMRING
--R subtractIfCan : (% , %) -> Union(%, "failed")
--R trunc : (% , NonNegativeInteger) -> %
--R unexpand : XDistributedPolynomial(VarSet, R) -> %
--R
--E 1

)spool
)lisp (bye)

```

— XRecursivePolynomial.help —

```

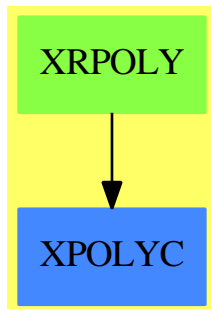
=====
XRecursivePolynomial examples
=====

```

This type supports multivariate polynomials whose variables do not commute. The representation is recursive. The coefficient ring may be non-commutative. Coefficients and variables commute.

See Also:
o)show XRecursivePolynomial

Polynomial arithmetic with non-commutative variables has been improved by a contribution of Michel Petitot (University of Lille I, France). The domain constructors **XRecursivePolynomial** provides a recursive for these polynomials. It is the non-commutative equivalents for the **SparseMultivariatePolynomial** constructor.

XRecursivePolynomial (XRPOLY)**Exports:**

0	1	characteristic	coef	coerce
constant	constant?	degree	expand	hash
latex	lquo	map	maxdeg	mindeg
mindegTerm	mirror	monom	monomial?	one?
quasiRegular	quasiRegular?	recip	RemainderList	retract
retractIfCan	rquo	sample	sh	subtractIfCan
trunc	unexpand	varList	zero?	?*?
?**?	?+?	?-?	-?	?=?
?^?	?~=?			

— domain XRPOLY XRecursivePolynomial —

```

)abbrev domain XRPOLY XRecursivePolynomial
++ Author: Michel Petitot petitot@lifl.fr
++ Date Created: 91
++ Date Last Updated: 7 Juillet 92
++ Description:
++ This type supports multivariate polynomials whose variables do not commute.
++ The representation is recursive. The coefficient ring may be
++ non-commutative. Coefficients and variables commute.

```

```

XRecursivePolynomial(VarSet:OrderedSet,R:Ring): Xcat == Xdef where
  I      ==> Integer
  NNI    ==> NonNegativeInteger
  XDPOLY ==> XDistributedPolynomial(VarSet, R)
  EX     ==> OutputForm
  WORD   ==> OrderedFreeMonoid(VarSet)
  TERM   ==> Record(k:VarSet , c:%)
  LTERMS ==> List(TERM)
  REGPOLY==> FreeModule1(%, VarSet)
  VPOLY  ==> Record(c0:R, reg:REGPOLY)

  Xcat == XPolynomialsCat(VarSet,R) with
    expand: % -> XDPOLY

```



```

    ++ \spad{expand(p)} returns \spad{p} in distributed form.
unexpand : XDPOLY -> %
    ++ \spad{unexpand(p)} returns \spad{p} in recursive form.
RemainderList: % -> LTERMS
    ++ \spad{RemainderList(p)} returns the regular part of \spad{p}
    ++ as a list of terms.

Xdef == add
import(VPOLY)

-- representation
Rep      := Union(R,VPOLY)

-- local functions
construct: LTERMS -> REGPOLY
simplifie: VPOLY -> %
lquo1: (LTERMS,LTERMS) -> %      -- a ajouter
coef1: (LTERMS,LTERMS) -> R      -- a ajouter
outForm: REGPOLY -> EX

--define
construct(lt) == lt pretend REGPOLY
p1:% = p2:% ==
  p1 case R =>
    p2 case R => p1 =$R p2
    false
  p2 case R => false
  p1.c0 =$R p2.c0 and p1.reg =$REGPOLY p2.reg

monom(w, r) ==
  r =0 => 0
  r * w::%

--      if R has Field then      -- Bug non resolu !!!!!!!
--      p:% / r: R == inv(r) * p

rquo(p1:%, p2:%):% ==
  p2 case R => p1 * p2::R
  p1 case R => p1 * p2.c0
  x:REGPOLY := construct [[t.k, a]$TERM for t in listOfTerms(p1.reg) _
    | (a:= rquo(t.c,p2)) ^= 0$% ]$LTERMS
  simplifie [coef(p1,p2) , x]$VPOLY

trunc(p,n) ==
  n = 0 or (p case R) => (constant p)::%
  n1: NNI := (n-1)::NNI
  lt: LTERMS := [[t.k, r]$TERM for t in listOfTerms p.reg _
    | (r := trunc(t.c, n1)) ^= 0]$LTERMS
  x: REGPOLY := construct lt
  simplifie [constant p, x]$VPOLY

```

```

unexpand p ==
  constant? p => (constant p)::%
  vl: List VarSet := sort((y,z) +-> y > z, varList p)
  x : REGPOLY := _
    construct [[v, unexpand r]$TERM for v in vl | (r:=lquo(p,v)) ^= 0]
  [constant p, x]$VPOLY

if R has CommutativeRing then
  sh(p:%, n:NNI):% ==
    n = 0 => 1
    p case R => (p::R)** n
    n1: NNI := (n-1)::NNI
    p1: % := n * sh(p, n1)
    lt: LTERMS := [[t.k, sh(t.c, p1)]$TERM for t in listOfTerms p.reg]
    [p.c0 ** n, construct lt]$VPOLY

  sh(p1:%, p2:%) ==
    p1 case R => p1::R * p2
    p2 case R => p1 * p2::R
    lt1:LTERMS := listOfTerms p1.reg ; lt2:LTERMS := listOfTerms p2.reg
    x: REGPOLY := construct [[t.k,sh(t.c,p2)]$TERM for t in lt1]
    y: REGPOLY := construct [[t.k,sh(p1,t.c)]$TERM for t in lt2]
    [p1.c0*p2.c0,x + y]$VPOLY

RemainderList p ==
  p case R => []
  listOfTerms( p.reg)$REGPOLY

lquo(p1:%,p2:%):% ==
  p2 case R => p1 * p2
  p1 case R => p1 *$R p2.c0
  p1 * p2.c0 +$% lquo1(listOfTerms p1.reg, listOfTerms p2.reg)

lquo1(x:LTERMS,y:LTERMS):% ==
  null x => 0$%
  null y => 0$%
  x.first.k < y.first.k => lquo1(x,y.rest)
  x.first.k = y.first.k =>
    lquo(x.first.c,y.first.c) + lquo1(x.rest,y.rest)
  return lquo1(x.rest,y)

coef(p1:%, p2:%):R ==
  p1 case R => p1::R * constant p2
  p2 case R => p1.c0 * p2::R
  p1.c0 * p2.c0 +$R coef1(listOfTerms p1.reg, listOfTerms p2.reg)

coef1(x:LTERMS,y:LTERMS):R ==
  null x => 0$R
  null y => 0$R

```

```

x.first.k < y.first.k => coef1(x,y.rest)
x.first.k = y.first.k =>
    coef(x.first.c,y.first.c) + coef1(x.rest,y.rest)
return coef1(x.rest,y)

-----
outForm(p:REGPOLY): EX ==
  le : List EX := [t.k::EX * t.c::EX for t in listOfTerms p]
  reduce(+, reverse_! le)$List(EX)

coerce(p:$): EX ==
  p case R => (p::R)::EX
  p.c0 = 0 => outForm p.reg
  p.c0::EX + outForm p.reg

0 == 0$R::%
1 == 1$R::%
constant? p == p case R
constant p ==
  p case R => p
  p.c0

simplifie p ==
  p.reg = 0$REGPOLY => (p.c0)::%
  p

coerce (v:VarSet):% ==
  [0$R,coerce(v)$REGPOLY]$VPOLY

coerce (r:R):% == r::%
coerce (n:Integer) == n::R::%
coerce (w:WORD) ==
  w = 1 => 1$R
  (first w) * coerce(rest w)

expand p ==
  p case R => p::R::XDPOLY
  lt:LTERMS := listOfTerms(p.reg)
  ep:XDPOLY := (p.c0)::XDPOLY
  for t in lt repeat
    ep:= ep + t.k * expand(t.c)
  ep

- p:% ==
  p case R => -$R p
  [- p.c0, - p.reg]$VPOLY

p1 + p2 ==
  p1 case R and p2 case R => p1 +$R p2
  p1 case R => [p1 + p2.c0 , p2.reg]$VPOLY

```

```

p2 case R => [p2 + p1.c0 , p1.reg]$VPOLY
simplifie [p1.c0 + p2.c0 , p1.reg +$REGPOLY p2.reg]$VPOLY

p1 - p2 ==
p1 case R and p2 case R => p1 -$R p2
p1 case R => [p1 - p2.c0 , -p2.reg]$VPOLY
p2 case R => [p1.c0 - p2 , p1.reg]$VPOLY
simplifie [p1.c0 - p2.c0 , p1.reg -$REGPOLY p2.reg]$VPOLY

n:Integer * p:% ==
n=0 => 0$%
p case R => n *$R p
-- [ n*p.c0,n*p.reg]$VPOLY
simplifie [ n*p.c0,n*p.reg]$VPOLY

r:R * p:% ==
r=0 => 0$%
p case R => r *$R p
-- [ r*p.c0,r*p.reg]$VPOLY
simplifie [ r*p.c0,r*p.reg]$VPOLY

p:% * r:R ==
r=0 => 0$%
p case R => p *$R r
-- [ p.c0 * r,p.reg * r]$VPOLY
simplifie [ r*p.c0,r*p.reg]$VPOLY

v:VarSet * p:% ==
p = 0 => 0$%
[0$R, v *$REGPOLY p]$VPOLY

p1:% * p2:% ==
p1 case R => p1::R * p2
p2 case R => p1 * p2::R
x:REGPOLY := p1.reg *$REGPOLY p2
y:REGPOLY := (p1.c0)::% *$REGPOLY p2.reg -- maladroite:(p1.c0)::% !!
-- [ p1.c0 * p2.c0 , x+y ]$VPOLY
simplifie [ p1.c0 * p2.c0 , x+y ]$VPOLY

lquo(p:%, v:VarSet):% ==
p case R => 0
coefficient(p.reg,v)$REGPOLY

lquo(p:%, w:WORD):% ==
w = 1$WORD => p
lquo(lquo(p,first w),rest w)

rquo(p:%, v:VarSet):% ==
p case R => 0
x:REGPOLY := construct [[t.k, a]$TERM for t in listOfTerms(p.reg)

```

```

      | (a:= rquo(t.c,v)) ^= 0 ]
    simplifie [constant(coefficient(p.reg,v)) , x]$VPOLY

rquo(p:%, w:WORD):% ==
  w = 1$WORD => p
  rquo(rquo(p,rest w),first w)

coef(p:%, w:WORD):R ==
  constant lquo(p,w)

quasiRegular? p ==
  p case R => p = 0$R
  p.c0 = 0$R

quasiRegular p ==
  p case R => 0$%
  [0$R,p.reg]$VPOLY

characteristic == characteristic()$R
recip p ==
  p case R => recip(p::R)
  "failed"

mindeg p ==
  p case R =>
    p = 0 => error "XRPOLY.mindeg: polynome nul !!"
    1$WORD
  p.c0 ^= 0 => 1$WORD
  "min"/[(t.k) *$WORD mindeg(t.c) for t in listOfTerms p.reg]

maxdeg p ==
  p case R =>
    p = 0 => error "XRPOLY.maxdeg: polynome nul !!"
    1$WORD
  "max"/[(t.k) *$WORD maxdeg(t.c) for t in listOfTerms p.reg]

degree p ==
  p = 0 => error "XRPOLY.degree: polynome nul !!"
  length(maxdeg p)

map(fn,p) ==
  p case R => fn(p::R)
  x:REGPOLY := construct [[t.k,a]$TERM for t in listOfTerms p.reg
    | (a := map(fn,t.c)) ^= 0$R]
  simplifie [fn(p.c0),x]$VPOLY

varList p ==
  p case R => []
  lv: List VarSet := "setUnion"/[varList(t.c) for t in listOfTerms p.reg]
  lv:= setUnion(lv,[t.k for t in listOfTerms p.reg])

```

sort_!(lv)

— XRPOLY.dotabb —

"XRPOLY" [color="#88FF44",href="bookvol10.3.pdf#nameddest=XRPOLY"]

"XPOLYC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=XPOLYC"]

"XRPOLY" -> "XPOLYC"

Chapter 26

Chapter Y

Chapter 27

Chapter Z

Chapter 28

The bootstrap code

BOOLEAN.lsp

BOOLEAN depends on **ORDSET** which depends on **SETCAT** which depends on **BASTYPE** which depends on **BOOLEAN**. We need to break this cycle to build the algebra. So we keep a cached copy of the translated **BOOLEAN** domain which we can write into the **MID** directory. We compile the lisp code and copy the **BOOLEAN.o** file to the **OUT** directory. This is eventually forcibly replaced by a recompiled version.

— **BOOLEAN.lsp BOOTSTRAP** —

```
(|/VERSIONCHECK| 2)

(PUT
  (QUOTE |BOOLEAN;test;2$;1|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|a|) |a|)))

(DEFUN |BOOLEAN;test;2$;1| (|a| |$|) |a|)

(DEFUN |BOOLEAN;nt| (|b| |$|)
  (COND (|b| (QUOTE NIL))
  ((QUOTE T) (QUOTE T))))

(PUT
  (QUOTE |BOOLEAN>true;$;3|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM NIL (QUOTE T))))

(DEFUN |BOOLEAN>true;$;3| (|a|)
  (QUOTE T))

(PUT
```

```

(QUOTE |BOOLEAN;false;$;4|)
(QUOTE |SPADreplace|)
(QUOTE (XLAM NIL NIL)))

(DEFUN |BOOLEAN;false;$;4| (|b|) NIL)

(DEFUN |BOOLEAN;not;2$;5| (|b| |$|)
  (COND
    (|b| (QUOTE NIL))
    ((QUOTE T) (QUOTE T))))

(DEFUN |BOOLEAN;~;2$;6| (|b| |$|)
  (COND
    (|b| (QUOTE NIL))
    ((QUOTE T) (QUOTE T))))

(DEFUN |BOOLEAN;~;2$;7| (|b| |$|)
  (COND
    (|b| (QUOTE NIL))
    ((QUOTE T) (QUOTE T))))

(DEFUN |BOOLEAN;and;3$;8| (|a| |b| |$|)
  (COND
    (|a| |b|)
    ((QUOTE T) (QUOTE NIL))))

(DEFUN |BOOLEAN;/\;3$;9| (|a| |b| |$|)
  (COND
    (|a| |b|)
    ((QUOTE T) (QUOTE NIL))))

(DEFUN |BOOLEAN;or;3$;10| (|a| |b| |$|)
  (COND
    (|a| (QUOTE T))
    ((QUOTE T) |b|)))

(DEFUN |BOOLEAN;\|/;3$;11| (|a| |b| |$|)
  (COND
    (|a| (QUOTE T))
    ((QUOTE T) |b|)))

(DEFUN |BOOLEAN;xor;3$;12| (|a| |b| |$|)
  (COND
    (|a| (|BOOLEAN;nt| |b| |$|))
    ((QUOTE T) |b|)))

(DEFUN |BOOLEAN;nor;3$;13| (|a| |b| |$|)
  (COND
    (|a| (QUOTE NIL))
    ((QUOTE T) (|BOOLEAN;nt| |b| |$|))))

```

```

(DEFUN |BOOLEAN;nand;3$;14| (|a| |b| |$|)
  (COND
    (|a| (|BOOLEAN;nt| |b| |$|))
    ((QUOTE T) (QUOTE T))))

(PUT
  (QUOTE |BOOLEAN;=;3$;15|)
  (QUOTE |SPADreplace|)
  (QUOTE |BooleanEquality|))

(DEFUN |BOOLEAN;=;3$;15| (|a| |b| |$|)
  (|BooleanEquality| |a| |b|))

(DEFUN |BOOLEAN;implies;3$;16| (|a| |b| |$|)
  (COND
    (|a| |b|)
    ((QUOTE T) (QUOTE T))))

(DEFUN |BOOLEAN;<;3$;17| (|a| |b| |$|)
  (COND
    (|b|
      (COND
        (|a| (QUOTE NIL))
        ((QUOTE T) (QUOTE T))))
    ((QUOTE T) (QUOTE NIL))))

(PUT
  (QUOTE |BOOLEAN;size;Nni;18|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM NIL 2)))

(DEFUN |BOOLEAN;size;Nni;18| (|a|) 2)

(DEFUN |BOOLEAN;index;Pi$;19| (|i| |$|)
  (COND
    ((SPADCALL |i| (QREFELT |$| 26)) (QUOTE NIL))
    ((QUOTE T) (QUOTE T))))

(DEFUN |BOOLEAN;lookup;$Pi;20| (|a| |$|)
  (COND
    (|a| 1)
    ((QUOTE T) 2)))

(DEFUN |BOOLEAN;random;$;21| (|a|)
  (COND
    ((SPADCALL (|random|) (QREFELT |$| 26)) (QUOTE NIL))
    ((QUOTE T) (QUOTE T))))

(DEFUN |BOOLEAN;convert;$If;22| (|x| |$|)

```

```

(COND
  (|x| (SPADCALL (SPADCALL "true" (QREFELT |$| 33)) (QREFELT |$| 35)))
  ((QUOTE T)
    (SPADCALL (SPADCALL "false" (QREFELT |$| 33)) (QREFELT |$| 35))))))

(DEFUN |BOOLEAN;coerce;$Of;23| (|x| |$|)
  (COND
    (|x| (SPADCALL "true" (QREFELT |$| 38)))
    ((QUOTE T) (SPADCALL "false" (QREFELT |$| 38)))))

(DEFUN |Boolean| ()
  (PROG ()
    (RETURN
      (PROG (G82461)
        (RETURN
          (COND
            ((LETT G82461 (HGET |$ConstructorCache| '|Boolean|)
              |Boolean|)
              (|CDRwithIncrement| (CDAR G82461)))
            ('T
              (UNWIND-PROTECT
                (PROG1 (CDDAR (HPUT |$ConstructorCache| '|Boolean|
                  (LIST
                    (CONS NIL (CONS 1 (|Boolean;|))))))
                  (LETT G82461 T |Boolean|))
                (COND
                  ((NOT G82461) (HREM |$ConstructorCache| '|Boolean|))))))))))

(DEFUN |Boolean;| ()
  (PROG (|dv$| $ |pv$|)
    (RETURN
      (PROGN
        (LETT |dv$| '(|Boolean|) |Boolean|)
        (LETT $ (make-array 41) |Boolean|)
        (QSETREFV $ 0 |dv$|)
        (QSETREFV $ 3
          (LETT |pv$| (|buildPredVector| 0 0 NIL) |Boolean|))
        (|haddProp| |$ConstructorCache| '|Boolean| NIL (CONS 1 $))
        (|stuffDomainSlots| $)
        $))))

(setf (get
  (QUOTE |Boolean|)
  (QUOTE |infovec|))
  (LIST
    (QUOTE
      #(NIL NIL NIL NIL NIL NIL
        (|Boolean|
          |BOOLEAN;test;2$;1|
          (CONS IDENTITY

```

```

      (FUNCALL (|dispatchFunction| |BOOLEAN;true;$;3|) |$|))
(CONS IDENTITY
  (FUNCALL (|dispatchFunction| |BOOLEAN>false;$;4|) |$|))
|BOOLEAN;not;2$;5|
|BOOLEAN;~;2$;6|
|BOOLEAN;~;2$;7|
|BOOLEAN;and;3$;8|
|BOOLEAN;/\;3$;9|
|BOOLEAN;or;3$;10|
|BOOLEAN;\|;3$;11|
|BOOLEAN;xor;3$;12|
|BOOLEAN;nor;3$;13|
|BOOLEAN;nand;3$;14|
|BOOLEAN;=;3$;15|
|BOOLEAN;implies;3$;16|
|BOOLEAN;<;3$;17|
(|NonNegativeInteger|)
|BOOLEAN;size;Nni;18|
(|Integer|)
(0 . |even?|)
(|PositiveInteger|)
|BOOLEAN;index;Pi$;19|
|BOOLEAN;lookup;$Pi;20|
|BOOLEAN;random;$;21|
(|String|)
(|Symbol|)
(5 . |coerce|)
(|InputForm|)
(10 . |convert|)
|BOOLEAN;convert;$If;22|
(|OutputForm|)
(15 . |message|)
|BOOLEAN;coerce;$Of;23|
(|SingleInteger|)))
(QUOTE
  #(|~| 20 |~| 26 |xor| 31 |true| 37 |test| 41 |size| 46 |random| 50
    |or| 54 |not| 60 |nor| 65 |nand| 71 |min| 77 |max| 83 |lookup| 89
    |latex| 94 |index| 99 |implies| 104 |hash| 110 |false| 115
    |convert| 119 |coerce| 124 |and| 129 |~| 135 |\/| 140 |>=| 146
    |>| 152 |=| 158 |<=| 164 |<| 170 |/\| 176))
(QUOTE NIL)
(CONS
  (|makeByteWordVec2| 1 (QUOTE (0 0 0 0 0 0)))
(CONS
  (QUOTE
    #(|OrderedSet&| NIL |Logic&| |SetCategory&| NIL |BasicType&| NIL))
(CONS
  (QUOTE
    #((|OrderedSet|)
      (|Finite|)

```



```

(|Logic|)
(|SetCategory|)
(|ConvertibleTo| 34)
(|BasicType|)
(|CoercibleTo| 37)))
(|makeByteWordVec2|
40
(QUOTE
  (1 25 6 0 26 1 32 0 31 33 1 34 0 32 35 1 37 0 31 38 2 0 6 0 0
   1 1 0 0 0 12 2 0 0 0 0 17 0 0 0 8 1 0 6 0 7 0 0 23 24 0 0 0
   30 2 0 0 0 0 15 1 0 0 0 10 2 0 0 0 0 18 2 0 0 0 0 19 2 0 0 0
   0 1 2 0 0 0 0 1 1 0 27 0 29 1 0 31 0 1 1 0 0 27 28 2 0 0 0 0
   21 1 0 40 0 1 0 0 0 9 1 0 34 0 36 1 0 37 0 39 2 0 0 0 0 13 1
   0 0 0 11 2 0 0 0 0 16 2 0 6 0 0 1 2 0 6 0 0 1 2 0 6 0 0 20 2
   0 6 0 0 1 2 0 6 0 0 22 2 0 0 0 0 14))))))
(QUOTE |lookupComplete|)))

(setf (get (QUOTE |Boolean|) (QUOTE NILADIC)) T)

```

—————→

CHAR.lsp BOOTSTRAP

CHAR depends on a chain of files. We need to break this cycle to build the algebra. So we keep a cached copy of the translated **CHAR** category which we can write into the **MID** directory. We compile the lisp code and copy the **CHAR.o** file to the **OUT** directory. This is eventually forcibly replaced by a recompiled version.

Note that this code is not included in the generated catdef.spad file.

— CHAR.lsp BOOTSTRAP —

```

(|/VERSIONCHECK| 2)

(PUT (QUOTE |CHAR;=;2$B;1|) (QUOTE |SPADreplace|) (QUOTE EQL))

(DEFUN |CHAR;=;2$B;1| (|a| |b| |$|) (EQL |a| |b|))

(PUT (QUOTE |CHAR;<;2$B;2|) (QUOTE |SPADreplace|) (QUOTE QSLESSP))

(DEFUN |CHAR;<;2$B;2| (|a| |b| |$|) (QSLESSP |a| |b|))

(PUT (QUOTE |CHAR;size;Nni;3|) (QUOTE |SPADreplace|) (QUOTE (XLAM NIL 256)))

(DEFUN |CHAR;size;Nni;3| (|$|) 256)

```

```

(DEFUN |CHAR;index;Pi$;4| (|n| |$|) (SPADCALL (|-| |n| 1) (QREFELT |$| 18)))

(DEFUN |CHAR;lookup;Pi$;5| (|c| $)
  (PROG (G90919)
    (RETURN
      (PROG1 (LETT G90919 (+ 1 (SPADCALL |c| (QREFELT $ 21)))
        |CHAR;lookup;Pi$;5|)
        (|check-subtype| (> G90919 0) '(|PositiveInteger| G90919))))))

(DEFUN |CHAR;char;I$;6| (|n| |$|) (SPADCALL |n| (QREFELT |$| 23)))

(PUT (QUOTE |CHAR;ord;$I;7|) (QUOTE |SPADreplace|) (QUOTE (XLAM (|c|) |c|)))

(DEFUN |CHAR;ord;$I;7| (|c| |$|) |c|)

(DEFUN |CHAR;random;$;8| (|$|)
  (SPADCALL (REMAINDER2 (|random|) (SPADCALL (QREFELT |$| 16)))
    (QREFELT |$| 18)))

(PUT (QUOTE |CHAR;space;$;9|)
  (QUOTE |SPADreplace|) (QUOTE (XLAM NIL (QENUM " " 0))))

(DEFUN |CHAR;space;$;9| (|$|) (QENUM " " 0))

(PUT (QUOTE |CHAR;quote;$;10|)
  (QUOTE |SPADreplace|) (QUOTE (XLAM NIL (QENUM "\" " 0))))

(DEFUN |CHAR;quote;$;10| (|$|) (QENUM "\" " 0))

(PUT (QUOTE |CHAR;escape;$;11|)
  (QUOTE |SPADreplace|) (QUOTE (XLAM NIL (QENUM "_" " 0))))

(DEFUN |CHAR;escape;$;11| (|$|) (QENUM "_" " 0))

(DEFUN |CHAR;coerce;$Of;12| (|c| |$|)
  (ELT (QREFELT |$| 10)
    (|+| (QREFELT |$| 11) (SPADCALL |c| (QREFELT |$| 21)))))

(DEFUN |CHAR;digit?;$B;13| (|c| |$|)
  (SPADCALL |c| (|spadConstant| |$| 31) (QREFELT |$| 33)))

(DEFUN |CHAR;hexDigit?;$B;14| (|c| |$|)
  (SPADCALL |c| (|spadConstant| |$| 35) (QREFELT |$| 33)))

(DEFUN |CHAR;upperCase?;$B;15| (|c| |$|)
  (SPADCALL |c| (|spadConstant| |$| 37) (QREFELT |$| 33)))

(DEFUN |CHAR;lowerCase?;$B;16| (|c| |$|)
  (SPADCALL |c| (|spadConstant| |$| 39) (QREFELT |$| 33)))

```

```

(DEFUN |CHAR;alphabetic?;$B;17| (|c| |$|)
  (SPADCALL |c| (|spadConstant| |$| 41) (QREFELT |$| 33)))

(DEFUN |CHAR;alphanumeric?;$B;18| (|c| |$|)
  (SPADCALL |c| (|spadConstant| |$| 43) (QREFELT |$| 33)))

(DEFUN |CHAR;latex;$S;19| (|c| |$|)
  (STRCONC "\mbox{'" (STRCONC (|MAKE-FULL-CVEC| 1 |c|) "'}"))))

(DEFUN |CHAR;char;$S;20| (|s| |$|)
  (COND
    ((EQL (QCSIZE |s|) 1)
      (SPADCALL |s| (SPADCALL |s| (QREFELT |$| 47)) (QREFELT |$| 48)))
    ((QUOTE T) (|error| "String is not a single character"))))

(DEFUN |CHAR;upperCase;2$;21| (|c| |$|)
  (QENUM (PNAME (UPCASE (code-char (SPADCALL |c| (QREFELT |$| 21))))) 0))

(DEFUN |CHAR;lowerCase;2$;22| (|c| |$|)
  (QENUM (PNAME (DOWNCASE (code-char (SPADCALL |c| (QREFELT |$| 21))))) 0))

(DEFUN |Character| ()
  (PROG ()
    (RETURN
      (PROG (G90941)
        (RETURN
          (COND
            ((LETT G90941 (HGET |$ConstructorCache| '|Character|)
              |Character|)
              (|CDRwithIncrement| (CDAR G90941)))
            ('T
              (UNWIND-PROTECT
                (PROG1 (CDDAR (HPUT |$ConstructorCache| '|Character|
                  (LIST
                    (CONS NIL (CONS 1 (|Character;|))))))
                  (LETT G90941 T |Character|))
                (COND
                  ((NOT G90941) (HREM |$ConstructorCache| '|Character|))))))))))

(DEFUN |Character;| ()
  (PROG (|dv$| $ |pv$| G90939 |i|)
    (RETURN
      (SEQ (PROGN
        (LETT |dv$| '(|Character|) |Character|)
        (LETT $ (make-array 53) |Character|)
        (QSETREFV $ 0 |dv$|)
        (QSETREFV $ 3
          (LETT |pv$| (|buildPredVector| 0 0 NIL) |Character|))
        (|haddProp| |$ConstructorCache| '|Character| NIL
          (CONS 1 $))

```

```

(|stuffDomainSlots| $)
(QSETREFV $ 6 (|SingleInteger|))
(QSETREFV $ 10
  (SPADCALL
    (PROGN
      (LETT G90939 NIL |Character|)
      (SEQ (LETT |i| 0 |Character|) G190
        (COND ((QSGREATERP |i| 255) (GO G191)))
        (SEQ (EXIT (LETT G90939
          (CONS (code-char |i|) G90939)
          |Character|)))
          (LETT |i| (QSADD1 |i|) |Character|)
          (GO G190) G191 (EXIT (NREVERSEO G90939))))
      (QREFELT $ 9)))
    (QSETREFV $ 11 0)
    $))))))

(setf (get (QUOTE |Character|) (QUOTE |infovec|))
  (LIST (QUOTE
    #(NIL NIL NIL NIL NIL NIL (QUOTE |Rep|) (|List| 28) (|PrimitiveArray| 28)
      (0 . |construct|) (QUOTE |OutChars|) (QUOTE |minChar|) (|Boolean|)
      |CHAR;|=;2$B;1| |CHAR;<;2$B;2| (|NonNegativeInteger|) |CHAR;size;Nni;3|
      (|Integer|) |CHAR;char;I$;6| (|PositiveInteger|) |CHAR;index;Pi$;4|
      |CHAR;ord;$I;7| |CHAR;lookup;$Pi;5| (5 . |coerce|) |CHAR;random;$;8|
      |CHAR;space;$;9| |CHAR;quote;$;10| |CHAR;escape;$;11| (|OutputForm|)
      |CHAR;coerce;$Of;12| (|CharacterClass|) (10 . |digit|) (|Character|)
      (14 . |member?|) |CHAR;digit?;$B;13| (20 . |hexDigit|)
      |CHAR;hexDigit?;$B;14| (24 . |upperCase|) |CHAR;upperCase?;$B;15|
      (28 . |lowerCase|) |CHAR;lowerCase?;$B;16| (32 . |alphabetic|)
      |CHAR;alphabetic?;$B;17| (36 . |alphanumeric|) |CHAR;alphanumeric?;$B;18|
      (|String|) |CHAR;latex;$S;19| (40 . |minIndex|) (45 . |elt|)
      |CHAR;char;$S;20| |CHAR;upperCase;2$;21| |CHAR;lowerCase;2$;22|
      (|SingleInteger|))) (QUOTE #(|~|=| 51 |upperCase?| 57 |upperCase| 62
      |space| 67 |size| 71 |random| 75 |quote| 79 |ord| 83 |min| 88 |max| 94
      |lowerCase?| 100 |lowerCase| 105 |lookup| 110 |latex| 115 |index| 120
      |hexDigit?| 125 |hash| 130 |escape| 135 |digit?| 139 |coerce| 144 |char|
      149 |alphanumeric?| 159 |alphabetic?| 164 |>=| 169 |>| 175 |=| 181 |<=|
      187 |<| 193)) (QUOTE NIL)
    (CONS
      (|makeByteWordVec2| 1 (QUOTE (0 0 0 0 0 0)))
      (CONS
        (QUOTE #(NIL |OrderedSet&| NIL |SetCategory&| |BasicType&| NIL))
        (CONS
          (QUOTE #((|OrderedFinite|) (|OrderedSet|) (|Finite|) (|SetCategory|)
            (|BasicType|) (|CoercibleTo| 28)))
          (|makeByteWordVec2| 52
            (QUOTE (1 8 0 7 9 1 6 0 17 23 0 30 0 31 2 30 12 32 0 33 0 30 0 35
              0 30 0 37 0 30 0 39 0 30 0 41 0 30 0 43 1 45 17 0 47 2 45
              32 0 17 48 2 0 12 0 0 1 1 0 12 0 38 1 0 0 0 50 0 0 0 25 0
              0 15 16 0 0 0 24 0 0 0 26 1 0 17 0 21 2 0 0 0 0 1 2 0 0 0

```

```

0 1 1 0 12 0 40 1 0 0 0 51 1 0 19 0 22 1 0 45 0 46 1 0 0 19
20 1 0 12 0 36 1 0 52 0 1 0 0 0 27 1 0 12 0 34 1 0 28 0 29
1 0 0 45 49 1 0 0 17 18 1 0 12 0 44 1 0 12 0 42 2 0 12 0 0
1 2 0 12 0 0 1 2 0 12 0 0 13 2 0 12 0 0 1 2 0 12 0 0 14))))))
(QUOTE |lookupComplete|)))

(setf (get (QUOTE |Character|) (QUOTE NILADIC)) T)

```

DFLOAT.lsp BOOTSTRAP

DFLOAT depends on itself. We need to break this cycle to build the algebra. So we keep a cached copy of the translated **DFLOAT** category which we can write into the **MID** directory. We compile the lisp code and copy the **DFLOAT.o** file to the **OUT** directory. This is eventually forcibly replaced by a recompiled version.

Note that this code is not included in the generated catdef.spad file.

— DFLOAT.lsp BOOTSTRAP —

```

(|/VERSIONCHECK| 2)

(DEFUN |DFLOAT;OMwrite;$S;1| (|x| |$|)
  (PROG (|sp| |dev| |s|)
    (RETURN
      (SEQ
        (LETT |s| "" |DFLOAT;OMwrite;$S;1|)
        (LETT |sp| (|OM-STRINGTOSTRINGPTR| |s|) |DFLOAT;OMwrite;$S;1|)
        (LETT |dev|
          (SPADCALL |sp| (SPADCALL (QREFELT |$| 7)) (QREFELT |$| 10))
          |DFLOAT;OMwrite;$S;1|)
        (SPADCALL |dev| (QREFELT |$| 12))
        (SPADCALL |dev| |x| (QREFELT |$| 14))
        (SPADCALL |dev| (QREFELT |$| 15))
        (SPADCALL |dev| (QREFELT |$| 16))
        (LETT |s| (|OM-STRINGPTRTOSTRING| |sp|) |DFLOAT;OMwrite;$S;1|)
        (EXIT |s|))))))

(DEFUN |DFLOAT;OMwrite;$BS;2| (|x| |wholeObj| |$|)
  (PROG (|sp| |dev| |s|)
    (RETURN
      (SEQ
        (LETT |s| "" |DFLOAT;OMwrite;$BS;2|)
        (LETT |sp| (|OM-STRINGTOSTRINGPTR| |s|) |DFLOAT;OMwrite;$BS;2|)
        (LETT |dev|
          (SPADCALL |sp| (SPADCALL (QREFELT |$| 7)) (QREFELT |$| 10))
          |DFLOAT;OMwrite;$BS;2|)

```

```

(COND (|wholeObj| (SPADCALL |dev| (QREFELT |$| 12))))
(SPADCALL |dev| |x| (QREFELT |$| 14))
(COND (|wholeObj| (SPADCALL |dev| (QREFELT |$| 15))))
(SPADCALL |dev| (QREFELT |$| 16))
(LETT |s| (|OM-STRINGPTRTOSTRING| |sp|) |DFLOAT;OMwrite;$BS;2|)
(EXIT |s|))))))

(DEFUN |DFLOAT;OMwrite;Omd$V;3| (|dev| |x| |$|)
  (SEQ
    (SPADCALL |dev| (QREFELT |$| 12))
    (SPADCALL |dev| |x| (QREFELT |$| 14))
    (EXIT (SPADCALL |dev| (QREFELT |$| 15)))))

(DEFUN |DFLOAT;OMwrite;Omd$BV;4| (|dev| |x| |wholeObj| |$|)
  (SEQ
    (COND (|wholeObj| (SPADCALL |dev| (QREFELT |$| 12))))
    (SPADCALL |dev| |x| (QREFELT |$| 14))
    (EXIT (COND (|wholeObj| (SPADCALL |dev| (QREFELT |$| 15)))))))

(PUT (QUOTE |DFLOAT;checkComplex|) (QUOTE |SPADreplace|) (QUOTE |C-TO-R|))

(DEFUN |DFLOAT;checkComplex| (|x| |$|) (|C-TO-R| |x|))

(PUT
  (QUOTE |DFLOAT;base;Pi;6|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM NIL (|FLOAT-RADIX| 0.0))))

(DEFUN |DFLOAT;base;Pi;6| (|$|) (|FLOAT-RADIX| 0.0))

(DEFUN |DFLOAT;mantissa;$I;7| (|x| |$|) (QCAR (|DFLOAT;manexp| |x| |$|)))

(DEFUN |DFLOAT;exponent;$I;8| (|x| |$|) (QCDR (|DFLOAT;manexp| |x| |$|)))

(PUT
  (QUOTE |DFLOAT;precision;Pi;9|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM NIL (|FLOAT-DIGITS| 0.0))))

(DEFUN |DFLOAT;precision;Pi;9| (|$|) (|FLOAT-DIGITS| 0.0))

(DEFUN |DFLOAT;bits;Pi;10| ($)
  (PROG (G105705)
    (RETURN
      (COND
        ((EQL (FLOAT-RADIX 0.0) 2) (FLOAT-DIGITS 0.0))
        ((EQL (FLOAT-RADIX 0.0) 16) (* 4 (FLOAT-DIGITS 0.0)))
        ('T
          (PROG1 (LETT G105705
            (truncate (SPADCALL (FLOAT-DIGITS 0.0)

```

```

                                (SPADCALL
                                (FLOAT (FLOAT-RADIX 0.0)
                                MOST-POSITIVE-LONG-FLOAT)
                                (QREFELT $ 28))
                                (QREFELT $ 29)))
                                |DFLOAT;bits;Pi;10|)
(|check-subtype| (> G105705 0) '(|PositiveInteger| G105705))))))

(PUT
 (QUOTE |DFLOAT;max;$;11|)
 (QUOTE |SPADreplace|)
 (QUOTE (XLAM NIL |MOST-POSITIVE-LONG-FLOAT|)))

(DEFUN |DFLOAT;max;$;11| (|$|) |MOST-POSITIVE-LONG-FLOAT|)

(PUT
 (QUOTE |DFLOAT;min;$;12|)
 (QUOTE |SPADreplace|)
 (QUOTE (XLAM NIL |MOST-NEGATIVE-LONG-FLOAT|)))

(DEFUN |DFLOAT;min;$;12| (|$|) |MOST-NEGATIVE-LONG-FLOAT|)

(DEFUN |DFLOAT;order;$I;13| (|a| |$|)
 (|-| (|+| (|FLOAT-DIGITS| 0.0) (SPADCALL |a| (QREFELT |$| 26))) 1))

(PUT
 (QUOTE |DFLOAT;Zero;$;14|)
 (QUOTE |SPADreplace|)
 (QUOTE (XLAM NIL (FLOAT 0 |MOST-POSITIVE-LONG-FLOAT|))))

(DEFUN |DFLOAT;Zero;$;14| (|$|) (FLOAT 0 |MOST-POSITIVE-LONG-FLOAT|))

(PUT
 (QUOTE |DFLOAT;One;$;15|)
 (QUOTE |SPADreplace|)
 (QUOTE (XLAM NIL (FLOAT 1 |MOST-POSITIVE-LONG-FLOAT|))))

(DEFUN |DFLOAT;One;$;15| (|$|) (FLOAT 1 |MOST-POSITIVE-LONG-FLOAT|))

(DEFUN |DFLOAT;exp1;$;16| (|$|)
 (|/|
 (FLOAT 534625820200 |MOST-POSITIVE-LONG-FLOAT|)
 (FLOAT 196677847971 |MOST-POSITIVE-LONG-FLOAT|)))

(PUT (QUOTE |DFLOAT;pi;$;17|) (QUOTE |SPADreplace|) (QUOTE (XLAM NIL PI)))

(DEFUN |DFLOAT;pi;$;17| (|$|) PI)

(DEFUN |DFLOAT;coerce;$Of;18| (|x| |$|) (SPADCALL |x| (QREFELT |$| 39)))

```

```

(DEFUN |DFLOAT;convert;$If;19| (|x| |$|) (SPADCALL |x| (QREFELT |$| 42)))

(PUT (QUOTE |DFLOAT;<;2$B;20|) (QUOTE |SPADreplace|) (QUOTE |<|))

(DEFUN |DFLOAT;<;2$B;20| (|x| |y| |$|) (|<| |x| |y|))

(PUT (QUOTE |DFLOAT;-;2$;21|) (QUOTE |SPADreplace|) (QUOTE |-|))

(DEFUN |DFLOAT;-;2$;21| (|x| |$|) (|-| |x|))

(PUT (QUOTE |DFLOAT;+;3$;22|) (QUOTE |SPADreplace|) (QUOTE |+|))

(DEFUN |DFLOAT;+;3$;22| (|x| |y| |$|) (|+| |x| |y|))

(PUT (QUOTE |DFLOAT;-;3$;23|) (QUOTE |SPADreplace|) (QUOTE |-|))

(DEFUN |DFLOAT;-;3$;23| (|x| |y| |$|) (|-| |x| |y|))

(PUT (QUOTE |DFLOAT;*;3$;24|) (QUOTE |SPADreplace|) (QUOTE |*|))

(DEFUN |DFLOAT;*;3$;24| (|x| |y| |$|) (|*| |x| |y|))

(PUT (QUOTE |DFLOAT;*;I2$;25|) (QUOTE |SPADreplace|) (QUOTE |*|))

(DEFUN |DFLOAT;*;I2$;25| (|i| |x| |$|) (|*| |i| |x|))

(PUT (QUOTE |DFLOAT;max;3$;26|) (QUOTE |SPADreplace|) (QUOTE MAX))

(DEFUN |DFLOAT;max;3$;26| (|x| |y| |$|) (MAX |x| |y|))

(PUT (QUOTE |DFLOAT;min;3$;27|) (QUOTE |SPADreplace|) (QUOTE MIN))

(DEFUN |DFLOAT;min;3$;27| (|x| |y| |$|) (MIN |x| |y|))

(PUT (QUOTE |DFLOAT;=;2$B;28|) (QUOTE |SPADreplace|) (QUOTE |=|))

(DEFUN |DFLOAT;=;2$B;28| (|x| |y| |$|) (|=| |x| |y|))

(PUT (QUOTE |DFLOAT;/;$I$;29|) (QUOTE |SPADreplace|) (QUOTE |/|))

(DEFUN |DFLOAT;/;$I$;29| (|x| |i| |$|) (|/| |x| |i|))

(DEFUN |DFLOAT;sqrt;2$;30| (|x| |$|) (|DFLOAT;checkComplex| (SQRT |x|) |$|))

(DEFUN |DFLOAT;log10;2$;31| (|x| |$|) (|DFLOAT;checkComplex| (|log| |x|) |$|))

(PUT (QUOTE |DFLOAT;**;$I$;32|) (QUOTE |SPADreplace|) (QUOTE EXPT))

(DEFUN |DFLOAT;**;$I$;32| (|x| |i| |$|) (EXPT |x| |i|))

```



```

(DEFUN |DFLOAT;**;3$;33| (|x| |y| |$|)
  (|DFLOAT;checkComplex| (EXPT |x| |y|) |$|))

(PUT
  (QUOTE |DFLOAT;coerce;I$;34|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|i|) (FLOAT |i| |MOST-POSITIVE-LONG-FLOAT|))))

(DEFUN |DFLOAT;coerce;I$;34| (|i| |$|) (FLOAT |i| |MOST-POSITIVE-LONG-FLOAT|))

(PUT (QUOTE |DFLOAT;exp;2$;35|) (QUOTE |SPADreplace|) (QUOTE EXP))

(DEFUN |DFLOAT;exp;2$;35| (|x| |$|) (EXP |x|))

(DEFUN |DFLOAT;log;2$;36| (|x| |$|) (|DFLOAT;checkComplex| (LN |x|) |$|))

(DEFUN |DFLOAT;log2;2$;37| (|x| |$|) (|DFLOAT;checkComplex| (LOG2 |x|) |$|))

(PUT (QUOTE |DFLOAT;sin;2$;38|) (QUOTE |SPADreplace|) (QUOTE SIN))

(DEFUN |DFLOAT;sin;2$;38| (|x| |$|) (SIN |x|))

(PUT (QUOTE |DFLOAT;cos;2$;39|) (QUOTE |SPADreplace|) (QUOTE COS))

(DEFUN |DFLOAT;cos;2$;39| (|x| |$|) (COS |x|))

(PUT (QUOTE |DFLOAT;tan;2$;40|) (QUOTE |SPADreplace|) (QUOTE TAN))

(DEFUN |DFLOAT;tan;2$;40| (|x| |$|) (TAN |x|))

(PUT (QUOTE |DFLOAT;cot;2$;41|) (QUOTE |SPADreplace|) (QUOTE COT))

(DEFUN |DFLOAT;cot;2$;41| (|x| |$|) (COT |x|))

(PUT (QUOTE |DFLOAT;sec;2$;42|) (QUOTE |SPADreplace|) (QUOTE SEC))

(DEFUN |DFLOAT;sec;2$;42| (|x| |$|) (SEC |x|))

(PUT (QUOTE |DFLOAT;csc;2$;43|) (QUOTE |SPADreplace|) (QUOTE CSC))

(DEFUN |DFLOAT;csc;2$;43| (|x| |$|) (CSC |x|))

(DEFUN |DFLOAT;asin;2$;44| (|x| |$|) (|DFLOAT;checkComplex| (ASIN |x|) |$|))

(DEFUN |DFLOAT;acos;2$;45| (|x| |$|) (|DFLOAT;checkComplex| (ACOS |x|) |$|))

(PUT (QUOTE |DFLOAT;atan;2$;46|) (QUOTE |SPADreplace|) (QUOTE ATAN))

(DEFUN |DFLOAT;atan;2$;46| (|x| |$|) (ATAN |x|))

```

```

(DEFUN |DFLOAT;acsc;2$;47| (|x| |$|) (|DFLOAT;checkComplex| (ACSC |x|) |$|))

(PUT (QUOTE |DFLOAT;acot;2$;48|) (QUOTE |SPADreplace|) (QUOTE ACOT))

(DEFUN |DFLOAT;acot;2$;48| (|x| |$|) (ACOT |x|))

(DEFUN |DFLOAT;asec;2$;49| (|x| |$|) (|DFLOAT;checkComplex| (ASEC |x|) |$|))

(PUT (QUOTE |DFLOAT;sinh;2$;50|) (QUOTE |SPADreplace|) (QUOTE SINH))

(DEFUN |DFLOAT;sinh;2$;50| (|x| |$|) (SINH |x|))

(PUT (QUOTE |DFLOAT;cosh;2$;51|) (QUOTE |SPADreplace|) (QUOTE COSH))

(DEFUN |DFLOAT;cosh;2$;51| (|x| |$|) (COSH |x|))

(PUT (QUOTE |DFLOAT;tanh;2$;52|) (QUOTE |SPADreplace|) (QUOTE TANH))

(DEFUN |DFLOAT;tanh;2$;52| (|x| |$|) (TANH |x|))

(PUT (QUOTE |DFLOAT;csch;2$;53|) (QUOTE |SPADreplace|) (QUOTE CSCH))

(DEFUN |DFLOAT;csch;2$;53| (|x| |$|) (CSCH |x|))

(PUT (QUOTE |DFLOAT;coth;2$;54|) (QUOTE |SPADreplace|) (QUOTE COTH))

(DEFUN |DFLOAT;coth;2$;54| (|x| |$|) (COTH |x|))

(PUT (QUOTE |DFLOAT;sech;2$;55|) (QUOTE |SPADreplace|) (QUOTE SECH))

(DEFUN |DFLOAT;sech;2$;55| (|x| |$|) (SECH |x|))

(PUT (QUOTE |DFLOAT;asinh;2$;56|) (QUOTE |SPADreplace|) (QUOTE ASINH))

(DEFUN |DFLOAT;asinh;2$;56| (|x| |$|) (ASINH |x|))

(DEFUN |DFLOAT;acosh;2$;57| (|x| |$|) (|DFLOAT;checkComplex| (ACOSH |x|) |$|))

(DEFUN |DFLOAT;atanh;2$;58| (|x| |$|) (|DFLOAT;checkComplex| (ATANH |x|) |$|))

(PUT (QUOTE |DFLOAT;acsch;2$;59|) (QUOTE |SPADreplace|) (QUOTE ACSCH))

(DEFUN |DFLOAT;acsch;2$;59| (|x| |$|) (ACSCH |x|))

(DEFUN |DFLOAT;acoth;2$;60| (|x| |$|) (|DFLOAT;checkComplex| (ACOTH |x|) |$|))

(DEFUN |DFLOAT;asech;2$;61| (|x| |$|) (|DFLOAT;checkComplex| (ASECH |x|) |$|))

(PUT (QUOTE |DFLOAT;/;3$;62|) (QUOTE |SPADreplace|) (QUOTE /|))

```

```

(DEFUN |DFLOAT;/;3$;62| (|x| |y| |$|) (|/| |x| |y|))

(PUT (QUOTE |DFLOAT;negative?;$B;63|) (QUOTE |SPADreplace|) (QUOTE MINUSP))

(DEFUN |DFLOAT;negative?;$B;63| (|x| |$|) (MINUSP |x|))

(PUT (QUOTE |DFLOAT;zero?;$B;64|) (QUOTE |SPADreplace|) (QUOTE ZEROP))

(DEFUN |DFLOAT;zero?;$B;64| (|x| |$|) (ZEROP |x|))

(PUT (QUOTE |DFLOAT;hash;$I;65|) (QUOTE |SPADreplace|) (QUOTE SXHASH))

(DEFUN |DFLOAT;hash;$I;65| (|x| |$|) (SXHASH |x|))

(DEFUN |DFLOAT;recip;$U;66| (|x| |$|)
  (COND
    ((ZEROP |x|) (CONS 1 "failed"))
    ((QUOTE T) (CONS 0 (|/| 1.0 |x|)))))

(PUT
  (QUOTE |DFLOAT;differentiate;2$;67|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|x|) 0.0)))

(DEFUN |DFLOAT;differentiate;2$;67| (|x| |$|) 0.0)

(DEFUN |DFLOAT;Gamma;2$;68| (|x| |$|) (SPADCALL |x| (QREFELT |$| 93)))

(DEFUN |DFLOAT;Beta;3$;69| (|x| |y| |$|) (SPADCALL |x| |y| (QREFELT |$| 95)))

(PUT (QUOTE |DFLOAT;wholePart;$I;70|) (QUOTE |SPADreplace|) (QUOTE truncate))

(DEFUN |DFLOAT;wholePart;$I;70| (|x| |$|) (truncate |x|))

(DEFUN |DFLOAT;float;2IPi$;71| (|ma| |ex| |b| |$|)
  (|*| |ma| (EXPT (FLOAT |b| |MOST-POSITIVE-LONG-FLOAT|) |ex|)))

(PUT
  (QUOTE |DFLOAT;convert;2$;72|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|x|) |x|)))

(DEFUN |DFLOAT;convert;2$;72| (|x| |$|) |x|)

(DEFUN |DFLOAT;convert;$F;73| (|x| |$|) (SPADCALL |x| (QREFELT |$| 101)))

(DEFUN |DFLOAT;rationalApproximation;$NniF;74| (|x| |d| |$|)
  (SPADCALL |x| |d| 10 (QREFELT |$| 105)))

(DEFUN |DFLOAT;atan;3$;75| (|x| |y| |$|)

```

```

(PROG (|theta|)
  (RETURN
    (SEQ
      (COND
        ((|=| |x| 0.0)
          (COND
            ((|<| 0.0 |y|) (|/| PI 2))
            ((|<| |y| 0.0) (|-| (|/| PI 2)))
            ((QUOTE T) 0.0)))
        ((QUOTE T)
          (SEQ
            (LETT |theta|
              (ATAN (|FLOAT-SIGN| 1.0 (|/| |y| |x|)))
              |DFLOAT;atan;3$;75|)
            (COND
              ((|<| |x| 0.0) (LETT |theta| (|-| PI |theta|) |DFLOAT;atan;3$;75|)))
            (COND ((|<| |y| 0.0) (LETT |theta| (|-| |theta|) |DFLOAT;atan;3$;75|)))
            (EXIT |theta|))))))

(DEFUN |DFLOAT;retract;$F;76| (|x| $)
  (PROG (G105780)
    (RETURN
      (SPADCALL |x|
        (PROG1 (LETT G105780 (- (FLOAT-DIGITS 0.0) 1)
          |DFLOAT;retract;$F;76|)
          (|check-subtype| (>= G105780 0) '(|NonNegativeInteger|
            G105780))
          (FLOAT-RADIX 0.0) (QREFELT $ 105))))))

(DEFUN |DFLOAT;retractIfCan;$U;77| (|x| $)
  (PROG (G105785)
    (RETURN
      (CONS 0
        (SPADCALL |x|
          (PROG1 (LETT G105785 (- (FLOAT-DIGITS 0.0) 1)
            |DFLOAT;retractIfCan;$U;77|)
            (|check-subtype| (>= G105785 0)
              '(|NonNegativeInteger| G105785))
            (FLOAT-RADIX 0.0) (QREFELT $ 105))))))

(DEFUN |DFLOAT;retract;$I;78| (|x| |$|)
  (PROG (|n|)
    (RETURN
      (SEQ
        (LETT |n| (truncate |x|) |DFLOAT;retract;$I;78|)
        (EXIT
          (COND
            ((|=| |x| (FLOAT |n| |MOST-POSITIVE-LONG-FLOAT|)) |n|)
            ((QUOTE T) (|error| "Not an integer"))))))))

```

```

(DEFUN |DFLOAT;retractIfCan;$U;79| (|x| |$|)
  (PROG (|n|)
    (RETURN
      (SEQ
        (LETT |n| (truncate |x|) |DFLOAT;retractIfCan;$U;79|)
        (EXIT
          (COND
            ((|=| |x| (FLOAT |n| |MOST-POSITIVE-LONG-FLOAT|)) (CONS 0 |n|))
            ((QUOTE T) (CONS 1 "failed"))))))))

(DEFUN |DFLOAT;sign;$I;80| (|x| |$|)
  (SPADCALL (|FLOAT-SIGN| |x| 1.0) (QREFELT |$| 111)))

(PUT
  (QUOTE |DFLOAT;abs;2$;81|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|x|) (|FLOAT-SIGN| 1.0 |x|))))

(DEFUN |DFLOAT;abs;2$;81| (|x| |$|) (|FLOAT-SIGN| 1.0 |x|))

(DEFUN |DFLOAT;manexp| (|x| $)
  (PROG (|s| G105806 |me| |two53|)
    (RETURN
      (SEQ (EXIT (COND
        ((ZEROP |x|) (CONS 0 0))
        ('T
          (SEQ (LETT |s| (SPADCALL |x| (QREFELT $ 114))
            |DFLOAT;manexp|)
            (LETT |x| (FLOAT-SIGN 1.0 |x|)
              |DFLOAT;manexp|)
            (COND
              ((< MOST-POSITIVE-LONG-FLOAT |x|)
                (PROGN
                  (LETT G105806
                    (CONS
                      (+
                        (* |s|
                          (SPADCALL
                            MOST-POSITIVE-LONG-FLOAT
                            (QREFELT $ 25)))
                        1)
                    (SPADCALL MOST-POSITIVE-LONG-FLOAT
                      (QREFELT $ 26)))
                  |DFLOAT;manexp|)
                  (GO G105806))))
            (LETT |me| (MANEXP |x|) |DFLOAT;manexp|)
            (LETT |two53|
              (EXPT (FLOAT-RADIX 0.0)
                (FLOAT-DIGITS 0.0))
              |DFLOAT;manexp|)

```

```

        (EXIT (CONS (* |s|
                    (truncate (* |two53| (QCAR |me|))))
                    (- (QCDR |me|) (FLOAT-DIGITS 0.0))))))
    G105806 (EXIT G105806))))))

(DEFUN |DFLOAT;rationalApproximation;$2NniF;83| (|f| |d| |b| $)
  (PROG (|#G102| |nu| |ex| BASE G105809 |de| |tol| |#G103| |q| |r| |p2|
        |q2| G105827 |#G104| |#G105| |p0| |p1| |#G106| |#G107|
        |q0| |q1| |#G108| |#G109| |s| |t| G105825)
    (RETURN
      (SEQ (EXIT (SEQ
        (PROGN
          (LETT |#G102| (|DFLOAT;manexp| |f| $)
                |DFLOAT;rationalApproximation;$2NniF;83|)
          (LETT |nu| (QCAR |#G102|)
                |DFLOAT;rationalApproximation;$2NniF;83|)
          (LETT |ex| (QCDR |#G102|)
                |DFLOAT;rationalApproximation;$2NniF;83|)
          |#G102|)
        (LETT BASE (FLOAT-RADIX 0.0)
                |DFLOAT;rationalApproximation;$2NniF;83|)
      (EXIT
        (COND
          ((< |ex| 0)
            (SEQ
              (LETT |de|
                (EXPT BASE
                  (PROG1
                    (LETT G105809 (- |ex|) |DFLOAT;rationalApproximation;$2NniF;83|)
                    (|check-subtype| (>= G105809 0) '(|NonNegativeInteger|
                      G105809))))
                |DFLOAT;rationalApproximation;$2NniF;83|)
            (EXIT
              (COND
                ((< |b| 2) (|error| "base must be > 1"))
                (t
                  (SEQ
                    (LETT |tol| (EXPT |b| |d|)
                      |DFLOAT;rationalApproximation;$2NniF;83|)
                    (LETT |s| |nu|
                      |DFLOAT;rationalApproximation;$2NniF;83|)
                    (LETT |t| |de|
                      |DFLOAT;rationalApproximation;$2NniF;83|)
                    (LETT |p0| 0
                      |DFLOAT;rationalApproximation;$2NniF;83|)
                    (LETT |p1| 1
                      |DFLOAT;rationalApproximation;$2NniF;83|)
                    (LETT |q0| 1
                      |DFLOAT;rationalApproximation;$2NniF;83|)
                    (LETT |q1| 0

```

```

      |DFLOAT;rationalApproximation;$2NniF;83|)
(EXIT
  (SEQ G190 NIL
    (SEQ
      (PROGN
        (LETT |#G103| (DIVIDE2 |s| |t|)
          |DFLOAT;rationalApproximation;$2NniF;83|)
        (LETT |q| (QCAR |#G103|)
          |DFLOAT;rationalApproximation;$2NniF;83|)
        (LETT |r| (QCDR |#G103|)
          |DFLOAT;rationalApproximation;$2NniF;83|)
        |#G103|)
      (LETT |p2| (+ (* |q| |p1|) |p0|)
        |DFLOAT;rationalApproximation;$2NniF;83|)
      (LETT |q2| (+ (* |q| |q1|) |q0|)
        |DFLOAT;rationalApproximation;$2NniF;83|)
      (COND
        ((OR (EQL |r| 0)
          (< (SPADCALL |tol| (ABS (- (* |nul| |q2|) (* |del| |p2|)))
            (QREFELT $ 118))
            (* |del| (ABS |p2|)))))
        (EXIT
          (PROGN
            (LETT G105827
              (SPADCALL |p2| |q2| (QREFELT $ 117))
              |DFLOAT;rationalApproximation;$2NniF;83|)
            (GO G105827))))
        (PROGN
          (LETT |#G104| |p1|
            |DFLOAT;rationalApproximation;$2NniF;83|)
          (LETT |#G105| |p2|
            |DFLOAT;rationalApproximation;$2NniF;83|)
          (LETT |p0| |#G104|
            |DFLOAT;rationalApproximation;$2NniF;83|)
          (LETT |p1| |#G105|
            |DFLOAT;rationalApproximation;$2NniF;83|))
          (PROGN
            (LETT |#G106| |q1|
              |DFLOAT;rationalApproximation;$2NniF;83|)
            (LETT |#G107| |q2|
              |DFLOAT;rationalApproximation;$2NniF;83|)
            (LETT |q0| |#G106|
              |DFLOAT;rationalApproximation;$2NniF;83|)
            (LETT |q1| |#G107|
              |DFLOAT;rationalApproximation;$2NniF;83|))
            (EXIT
              (PROGN
                (LETT |#G108| |t|
                  |DFLOAT;rationalApproximation;$2NniF;83|)
                (LETT |#G109| |r|

```

```

      |DFLOAT;rationalApproximation;$2NniF;83|)
    (LETT |s| |#G108|
      |DFLOAT;rationalApproximation;$2NniF;83|)
    (LETT |t| |#G109|
      |DFLOAT;rationalApproximation;$2NniF;83|))))
    NIL (GO G190) G191
    (EXIT NIL)))))))))
('T
  (SPADCALL
    (* |nu|
      (EXPT BASE
        (PROG1
          (LETT G105825 |ex| |DFLOAT;rationalApproximation;$2NniF;83|)
            (|check-subtype| (>= G105825 0) '(|NonNegativeInteger|
              G105825))))
          (QREFELT $ 119))))))
  G105827
  (EXIT G105827))))))

(DEFUN |DFLOAT;**;$F$;84| (|x| |r| $)
  (PROG (|n| |d| G105837)
    (RETURN
      (SEQ (EXIT (COND
        ((ZEROP |x|)
          (COND
            ((SPADCALL |r| (QREFELT $ 120))
              (|error| "0**0 is undefined"))
            ((SPADCALL |r| (QREFELT $ 121))
              (|error| "division by 0"))
            ('T 0.0)))
          ((OR (SPADCALL |r| (QREFELT $ 120))
              (SPADCALL |x| (QREFELT $ 122)))
            1.0)
          ('T
            (COND
              ((SPADCALL |r| (QREFELT $ 123)) |x|)
              ('T
                (SEQ (LETT |n| (SPADCALL |r| (QREFELT $ 124))
                  |DFLOAT;**;$F$;84|)
                  (LETT |d| (SPADCALL |r| (QREFELT $ 125))
                    |DFLOAT;**;$F$;84|)
                  (EXIT (COND
                    ((MINUSP |x|)
                      (COND
                        ((ODDP |d|)
                          (COND
                            ((ODDP |n|)
                              (PROGN
                                (LETT G105837
                                  (-

```



```

        (SPADCALL (- |x|) |r|
         (QREFELT $ 126)))
      |DFLOAT;**;$F$;84|)
    (GO G105837)))
('T
 (PROGN
  (LETT G105837
   (SPADCALL (- |x|) |r|
    (QREFELT $ 126))
   |DFLOAT;**;$F$;84|)
  (GO G105837))))
('T (|error| "negative root"))))
((EQL |d| 2)
 (EXPT
  (SPADCALL |x| (QREFELT $ 54))
  |n|))
('T
 (SPADCALL |x|
  (/
   (FLOAT |n|
    MOST-POSITIVE-LONG-FLOAT)
   (FLOAT |d|
    MOST-POSITIVE-LONG-FLOAT))
  (QREFELT $ 57)))))))))
G105837 (EXIT G105837))))))

(DEFUN |DoubleFloat| ()
 (PROG ()
  (RETURN
   (PROG (G105850)
    (RETURN
     (COND
      ((LETT G105850 (HGET |$ConstructorCache| '|DoubleFloat|)
       |DoubleFloat|)
       (|CDRwithIncrement| (CDAR G105850)))
      ('T
       (UNWIND-PROTECT
        (PROG1 (CDDAR (HPUT |$ConstructorCache| '|DoubleFloat|
         (LIST
          (CONS NIL
           (CONS 1 (|DoubleFloat;|))))))
         (LETT G105850 T |DoubleFloat|))
        (COND
         ((NOT G105850)
          (HREM |$ConstructorCache| '|DoubleFloat|))))))))))

(DEFUN |DoubleFloat;| ()
 (PROG (|dv$| $ |pv$|)
  (RETURN
   (PROGN
    (SPADCALL (- |x|) |r|
     (QREFELT $ 126)))
    |DFLOAT;**;$F$;84|)
  (GO G105837))))

```

```

(LETT |dv$| '(|DoubleFloat|) |DoubleFloat|)
(LETT $ (make-array 140) |DoubleFloat|)
(QSETREFV $ 0 |dv$|)
(QSETREFV $ 3
  (LETT |pv$| (|buildPredVector| 0 0 NIL) |DoubleFloat|))
(|haddProp| |$ConstructorCache| ' |DoubleFloat| NIL (CONS 1 $))
(|stuffDomainSlots| $)
$)))

(setf (get
  (QUOTE |DoubleFloat|)
  (QUOTE |infovec|))
  (LIST
    (QUOTE #(NIL NIL NIL NIL NIL NIL (|OpenMathEncoding|) (0 . |OMencodingXML|)
      (|String|) (|OpenMathDevice|) (4 . |OMopenString|) (|Void|)
      (10 . |OMputObject|) (|DoubleFloat|) (15 . |OMputFloat|)
      (21 . |OMputEndObject|) (26 . |OMclose|) |DFLOAT;OMwrite;$S;1|
      (|Boolean|) |DFLOAT;OMwrite;$BS;2| |DFLOAT;OMwrite;Omd$V;3|
      |DFLOAT;OMwrite;Omd$BV;4| (|PositiveInteger|) |DFLOAT;base;Pi;6|
      (|Integer|) |DFLOAT;mantissa;$I;7| |DFLOAT;exponent;$I;8|
      |DFLOAT;precision;Pi;9| |DFLOAT;log2;2$;37| (31 . |*|)
      |DFLOAT;bits;Pi;10| |DFLOAT;max;$;11| |DFLOAT;min;$;12|
      |DFLOAT;order;$I;13|
      (CONS IDENTITY (FUNCALL (|dispatchFunction| |DFLOAT;Zero;$;14|) |$|))
      (CONS IDENTITY (FUNCALL (|dispatchFunction| |DFLOAT;One;$;15|) |$|))
      |DFLOAT;exp1;$;16| |DFLOAT;pi;$;17| (|OutputForm|) (37 . |outputForm|)
      |DFLOAT;coerce;$Of;18| (|InputForm|) (42 . |convert|)
      |DFLOAT;convert;$If;19| |DFLOAT;<;2$B;20| |DFLOAT;-;2$;21| | |
      |DFLOAT;+;3$;22| |DFLOAT;-;3$;23| |DFLOAT;*;3$;24| |DFLOAT;*;I2$;25|
      |DFLOAT;max;3$;26| |DFLOAT;min;3$;27| |DFLOAT;=;2$B;28|
      |DFLOAT;/;$I$;29| |DFLOAT;sqrt;2$;30| |DFLOAT;log10;2$;31|
      |DFLOAT;**;$I$;32| |DFLOAT;**;3$;33| |DFLOAT;coerce;I$;34|
      |DFLOAT;exp;2$;35| |DFLOAT;log;2$;36| |DFLOAT;sin;2$;38|
      |DFLOAT;cos;2$;39| |DFLOAT;tan;2$;40| |DFLOAT;cot;2$;41|
      |DFLOAT;sec;2$;42| |DFLOAT;csc;2$;43| |DFLOAT;asin;2$;44|
      |DFLOAT;acos;2$;45| |DFLOAT;atan;2$;46| |DFLOAT;acsc;2$;47|
      |DFLOAT;acot;2$;48| |DFLOAT;asec;2$;49| |DFLOAT;sinh;2$;50|
      |DFLOAT;cosh;2$;51| |DFLOAT;tanh;2$;52| |DFLOAT;csch;2$;53|
      |DFLOAT;coth;2$;54| |DFLOAT;sech;2$;55| |DFLOAT;asinh;2$;56|
      |DFLOAT;acosh;2$;57| |DFLOAT;atanh;2$;58| |DFLOAT;acsch;2$;59|
      |DFLOAT;acoth;2$;60| |DFLOAT;asech;2$;61| |DFLOAT;/;3$;62|
      |DFLOAT;negative?;$B;63| |DFLOAT;zero?;$B;64| |DFLOAT;hash;$I;65|
      (|Union| |$| (QUOTE "failed")) |DFLOAT;recip;$U;66|
      |DFLOAT;differentiate;2$;67| (|DoubleFloatSpecialFunctions|)
      (47 . |Gamma|) |DFLOAT;Gamma;2$;68| (52 . |Beta|) |DFLOAT;Beta;3$;69|
      |DFLOAT;wholePart;$I;70| |DFLOAT;float;2IPi$;71| |DFLOAT;convert;2$;72|
      (|Float|) (58 . |convert|) |DFLOAT;convert;$F;73| (|Fraction| 24)
      (|NonNegativeInteger|) |DFLOAT;rationalApproximation;$2NniF;83|
      |DFLOAT;rationalApproximation;$NniF;74| |DFLOAT;atan;3$;75|
      |DFLOAT;retract;$F;76| (|Union| 103 (QUOTE "failed"))

```

[illegible]

```

|ElementaryFunctionCategory&| NIL |HyperbolicFunctionCategory&|
|ArcTrigonometricFunctionCategory&| |TrigonometricFunctionCategory&|
NIL NIL |RadicalCategory&| |RetractableTo&| |RetractableTo&| NIL
NIL |BasicType&| NIL)) (CONS (QUOTE #(|FloatingPointSystem|)
(|RealNumberSystem|) (|Field|) (|EuclideanDomain|)
(|PrincipalIdealDomain|) (|UniqueFactorizationDomain|)
(|GcdDomain|) (|DivisionRing|) (|IntegralDomain|) (|Algebra| 103)
(|Algebra| |$$|) (|DifferentialRing|) (|CharacteristicZero|)
(|OrderedRing|) (|Module| 103) (|EntireRing|) (|CommutativeRing|)
(|Module| |$$|) (|OrderedAbelianGroup|) (|BiModule| 103 103)
(|BiModule| |$$| |$$|) (|Ring|) (|OrderedCancellationAbelianMonoid|)
(|RightModule| 103) (|LeftModule| 103) (|LeftModule| |$$|) (|Rng|)
(|RightModule| |$$|) (|OrderedAbelianMonoid|) (|AbelianGroup|)
(|OrderedAbelianSemiGroup|) (|CancellationAbelianMonoid|)
(|AbelianMonoid|) (|Monoid|) (|PatternMatchable| 100) (|OrderedSet|)
(|AbelianSemiGroup|) (|SemiGroup|) (|TranscendentalFunctionCategory|)
(|RealConstant|) (|SetCategory|) (|ConvertibleTo| 41)
(|ElementaryFunctionCategory|) (|ArcHyperbolicFunctionCategory|)
(|HyperbolicFunctionCategory|) (|ArcTrigonometricFunctionCategory|)
(|TrigonometricFunctionCategory|) (|OpenMath|) (|ConvertibleTo| 127)
(|RadicalCategory|) (|RetractableTo| 103) (|RetractableTo| 24)
(|ConvertibleTo| 100) (|ConvertibleTo| 13) (|BasicType|)
(|CoercibleTo| 38)))
(|makeByteWordVec2| 139
(QUOTE (0 6 0 7 2 9 0 8 6 10 1 9 11 0 12 2 9 11 0 13 14 1 9 11 0 15
1 9 11 0 16 2 0 0 22 0 29 1 38 0 13 39 1 41 0 13 42 1 92 13 13 93 2 92
13 13 13 95 1 100 0 13 101 0 103 0 116 2 103 0 24 24 117 2 24 0 104 0
118 1 103 0 24 119 1 103 18 0 120 1 103 18 0 121 1 0 18 0 122 1 103 18
0 123 1 103 24 0 124 1 103 24 0 125 2 0 18 0 0 1 1 0 18 0 87 1 0 24 0
97 1 0 138 0 1 1 0 0 0 1 1 0 18 0 1 1 0 0 0 1 1 0 0 0 75 1 0 0 0 63 2
0 89 0 0 1 1 0 0 0 1 1 0 129 0 1 1 0 0 0 54 2 0 18 0 0 1 1 0 0 0 73 1
0 0 0 61 1 0 24 0 114 1 0 0 0 78 1 0 0 0 65 0 0 0 1 1 0 0 0 1 1 0 109
0 110 1 0 112 0 113 1 0 103 0 108 1 0 24 0 111 2 0 0 0 0 1 1 0 89 0
90 2 0 103 0 104 106 3 0 103 0 104 104 105 2 0 0 0 0 1 1 0 136 131 1
1 0 18 0 1 0 0 22 27 1 0 18 0 1 0 0 0 37 3 0 128 0 127 128 1 1 0 24
0 33 1 0 18 0 122 2 0 0 0 24 1 1 0 0 0 1 1 0 18 0 86 2 0 130 131 0 1
0 0 0 32 2 0 0 0 0 51 0 0 0 31 2 0 0 0 0 50 1 0 24 0 25 1 0 0 0 28 1
0 0 0 55 1 0 0 0 60 1 0 0 131 1 2 0 0 0 0 1 1 0 8 0 1 1 0 0 0 1 1 0
24 0 88 1 0 139 0 1 2 0 137 137 137 1 1 0 0 131 1 2 0 0 0 0 1 1 0 0
0 1 1 0 0 0 1 3 0 0 24 24 22 98 2 0 0 24 24 1 1 0 129 0 1 2 0 132 0
0 1 3 0 134 0 0 0 1 2 0 89 0 0 1 2 0 130 131 0 1 1 0 24 0 26 0 0 0
36 1 0 0 0 59 1 0 104 0 1 2 0 135 0 0 1 0 0 22 1 1 0 0 0 91 2 0 0 0
104 1 1 0 0 0 76 1 0 0 0 66 1 0 0 0 77 1 0 0 0 64 1 0 0 0 74 1 0 0 0
62 1 0 41 0 43 1 0 127 0 1 1 0 13 0 99 1 0 100 0 102 1 0 0 103 1 1 0
0 24 58 1 0 0 103 1 1 0 0 24 58 1 0 0 0 1 1 0 38 0 40 0 0 104 1 1 0
0 0 1 0 0 22 30 0 0 22 23 1 0 0 0 81 2 0 0 0 0 107 1 0 0 0 69 2 0 18
0 0 1 1 0 0 0 79 1 0 0 0 67 1 0 0 0 84 1 0 0 0 72 1 0 0 0 82 1 0 0 0
70 1 0 0 0 83 1 0 0 0 71 1 0 0 0 80 1 0 0 0 68 1 0 0 0 115 2 0 0 0
24 1 2 0 0 0 104 1 2 0 0 0 22 1 0 0 0 34 0 0 0 35 2 0 11 9 0 20 3 0
11 9 0 18 21 1 0 8 0 17 2 0 8 0 18 19 1 0 0 0 94 1 0 0 0 1 2 0 0 0

```

```

104 1 2 0 0 0 0 96 2 0 18 0 0 1 2 0 18 0 0 1 2 0 18 0 0 52 2 0 18 0
0 1 2 0 18 0 0 44 2 0 0 0 24 53 2 0 0 0 0 85 2 0 0 0 0 47 1 0 0 0
45 2 0 0 0 0 46 2 0 0 0 0 57 2 0 0 0 103 126 2 0 0 0 24 56 2 0 0 0
104 1 2 0 0 0 22 1 2 0 0 0 103 1 2 0 0 103 0 1 2 0 0 0 0 48 2 0 0
24 0 49 2 0 0 104 0 1 2 0 0 22 0 29))))))
(QUOTE |lookupComplete|))

```

```
(setf (get (QUOTE |DoubleFloat|) (QUOTE NILADIC)) T)
```

—

ILIST.lsp BOOTSTRAP

ILIST depends on a chain of files. We need to break this cycle to build the algebra. So we keep a cached copy of the translated **ILIST** category which we can write into the **MID** directory. We compile the lisp code and copy the **ILIST.o** file to the **OUT** directory. This is eventually forcibly replaced by a recompiled version.

Note that this code is not included in the generated catdef.spad file.

— ILIST.lsp BOOTSTRAP —

```

(|/VERSIONCHECK| 2)

(PUT (QUOTE |ILIST;#;$Nni;1|) (QUOTE |SPADreplace|) (QUOTE LENGTH))

(DEFUN |ILIST;#;$Nni;1| (|x| |$|) (LENGTH |x|))

(PUT (QUOTE |ILIST;concat;S2$;2|) (QUOTE |SPADreplace|) (QUOTE CONS))

(DEFUN |ILIST;concat;S2$;2| (|s| |x| |$|) (CONS |s| |x|))

(PUT (QUOTE |ILIST;eq?;2$B;3|) (QUOTE |SPADreplace|) (QUOTE EQ))

(DEFUN |ILIST;eq?;2$B;3| (|x| |y| |$|) (EQ |x| |y|))

(PUT (QUOTE |ILIST;first;$S;4|) (QUOTE |SPADreplace|) (QUOTE |SPADfirst|))

(DEFUN |ILIST;first;$S;4| (|x| |$|) (|SPADfirst| |x|))

(PUT
  (QUOTE |ILIST;elt;$firstS;5|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|x| "first") (|SPADfirst| |x|))))

(DEFUN |ILIST;elt;$firstS;5| (|x| G101995 |$|) (|SPADfirst| |x|))

(PUT (QUOTE |ILIST;empty;$;6|) (QUOTE |SPADreplace|) (QUOTE (XLAM NIL NIL)))

```

```

(DEFUN |ILIST;empty;$;6| (|$|) NIL)

(PUT (QUOTE |ILIST;empty?;$B;7|) (QUOTE |SPADreplace|) (QUOTE NULL))

(DEFUN |ILIST;empty?;$B;7| (|x| |$|) (NULL |x|))

(PUT (QUOTE |ILIST;rest;2$;8|) (QUOTE |SPADreplace|) (QUOTE CDR))

(DEFUN |ILIST;rest;2$;8| (|x| |$|) (CDR |x|))

(PUT
  (QUOTE |ILIST;elt;$rest$;9|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|x| "rest") (CDR |x|))))

(DEFUN |ILIST;elt;$rest$;9| (|x| G102000 |$|) (CDR |x|))

(DEFUN |ILIST;setfirst!;$2S;10| (|x| |s| |$|)
  (COND
    ((SPADCALL |x| (QREFELT |$| 17)) (|error| "Cannot update an empty list"))
    ((QUOTE T) (QCAR (RPLACA |x| |s|)))))

(DEFUN |ILIST;setelt;$first2S;11| (|x| G102005 |s| |$|)
  (COND
    ((SPADCALL |x| (QREFELT |$| 17)) (|error| "Cannot update an empty list"))
    ((QUOTE T) (QCAR (RPLACA |x| |s|)))))

(DEFUN |ILIST;setrest!;$3$;12| (|x| |y| |$|)
  (COND
    ((SPADCALL |x| (QREFELT |$| 17)) (|error| "Cannot update an empty list"))
    ((QUOTE T) (QCDR (RPLACD |x| |y|)))))

(DEFUN |ILIST;setelt;$rest2$;13| (|x| G102010 |y| |$|)
  (COND
    ((SPADCALL |x| (QREFELT |$| 17)) (|error| "Cannot update an empty list"))
    ((QUOTE T) (QCDR (RPLACD |x| |y|)))))

(PUT
  (QUOTE |ILIST;construct;L$;14|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|l| |l|))))

(DEFUN |ILIST;construct;L$;14| (|l| |$|) |l|)

(PUT
  (QUOTE |ILIST;parts;$L;15|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|s| |s|))))

```

```

(DEFUN |ILIST;parts;$L;15| (|s| |$|) |s|)

(PUT (QUOTE |ILIST;reverse!;2$;16|) (QUOTE |SPADreplace|) (QUOTE NREVERSE))

(DEFUN |ILIST;reverse!;2$;16| (|x| |$|) (NREVERSE |x|))

(PUT (QUOTE |ILIST;reverse;2$;17|) (QUOTE |SPADreplace|) (QUOTE REVERSE))

(DEFUN |ILIST;reverse;2$;17| (|x| |$|) (REVERSE |x|))

(DEFUN |ILIST;minIndex;$I;18| (|x| |$|) (QREFELT |$| 7))

(DEFUN |ILIST;rest;$Nni$;19| (|x| |n| |$|)
  (PROG (|i|)
    (RETURN
      (SEQ
        (SEQ
          (LETT |i| 1 |ILIST;rest;$Nni$;19|)
          G190
          (COND ((QSGREATERP |i| |n|) (GO G191)))
          (SEQ
            (COND ((NULL |x|) (|error| "index out of range")))
            (EXIT (LETT |x| (QCDR |x|) |ILIST;rest;$Nni$;19|)))
          (LETT |i| (QSADD1 |i|) |ILIST;rest;$Nni$;19|)
          (GO G190)
          G191
          (EXIT NIL))
          (EXIT |x|))))))

(DEFUN |ILIST;copy;2$;20| (|x| |$|)
  (PROG (|i| |y|)
    (RETURN
      (SEQ
        (LETT |y| (SPADCALL (QREFELT |$| 16)) |ILIST;copy;2$;20|)
        (SEQ
          (LETT |i| 0 |ILIST;copy;2$;20|)
          G190
          (COND
            ((NULL (COND ((NULL |x|) (QUOTE NIL)) ((QUOTE T) (QUOTE T)))) (GO G191)))
          (SEQ
            (COND
              ((EQ |i| 1000)
                (COND ((SPADCALL |x| (QREFELT |$| 33)) (|error| "cyclic list"))))
              (LETT |y| (CONS (QCAR |x|) |y|) |ILIST;copy;2$;20|)
              (EXIT (LETT |x| (QCDR |x|) |ILIST;copy;2$;20|)))
            (LETT |i| (QSADD1 |i|) |ILIST;copy;2$;20|)
            (GO G190)
            G191
            (EXIT NIL))
            (EXIT (NREVERSE |y|))))))

```

```

(DEFUN |ILIST;coerce;$0f;21| (|x| |$|)
  (PROG (|s| |y| |z|)
    (RETURN
      (SEQ
        (LETT |y| NIL |ILIST;coerce;$0f;21|)
        (LETT |s| (SPADCALL |x| (QREFELT |$| 35)) |ILIST;coerce;$0f;21|)
        (SEQ
          G190
          (COND ((NULL (NEQ |x| |s|)) (GO G191)))
          (SEQ
            (LETT |y|
              (CONS (SPADCALL (SPADCALL |x| (QREFELT |$| 13)) (QREFELT |$| 37)) |y|)
              |ILIST;coerce;$0f;21|)
            (EXIT (LETT |x| (SPADCALL |x| (QREFELT |$| 18)) |ILIST;coerce;$0f;21|)))
          NIL
          (GO G190)
          G191
          (EXIT NIL))
        (LETT |y| (NREVERSE |y|) |ILIST;coerce;$0f;21|)
        (EXIT
          (COND
            ((SPADCALL |s| (QREFELT |$| 17)) (SPADCALL |y| (QREFELT |$| 39)))
            ((QUOTE T)
              (SEQ
                (LETT |z|
                  (SPADCALL
                    (SPADCALL (SPADCALL |x| (QREFELT |$| 13)) (QREFELT |$| 37))
                    (QREFELT |$| 41))
                  |ILIST;coerce;$0f;21|)
                (SEQ
                  G190
                  (COND ((NULL (NEQ |s| (SPADCALL |x| (QREFELT |$| 18)))) (GO G191)))
                  (SEQ
                    (LETT |x| (SPADCALL |x| (QREFELT |$| 18)) |ILIST;coerce;$0f;21|)
                    (EXIT
                      (LETT |z|
                        (CONS
                          (SPADCALL (SPADCALL |x| (QREFELT |$| 13)) (QREFELT |$| 37))
                          |z|)
                        |ILIST;coerce;$0f;21|)))
                    NIL
                    (GO G190)
                    G191
                    (EXIT NIL))
                  (EXIT
                    (SPADCALL
                      (SPADCALL |y|
                        (SPADCALL
                          (SPADCALL

```



```

(NREVERSE |z|)
(QREFELT |$| 42))
(QREFELT |$| 43))
(QREFELT |$| 44))
(QREFELT |$| 39)))))))))

(DEFUN |ILIST;=;2$B;22| (|x| |y| $)
  (PROG (G102042)
    (RETURN
      (SEQ (EXIT (COND
        ((EQ |x| |y|) 'T)
        ('T
          (SEQ (SEQ G190
            (COND
              ((NULL (COND
                ((OR (NULL |x|) (NULL |y|))
                'NIL)
                ('T 'T)))
              (GO G191)))
            (SEQ (EXIT
              (COND
                ((NULL
                  (SPADCALL (QCAR |x|) (QCAR |y|)
                  (QREFELT $ 46)))
                (PROGN
                  (LETT G102042 'NIL
                    |ILIST;=;2$B;22|)
                  (GO G102042)))
                ('T
                  (SEQ
                    (LETT |x| (QCDR |x|)
                    |ILIST;=;2$B;22|)
                    (EXIT
                      (LETT |y| (QCDR |y|)
                      |ILIST;=;2$B;22|))))))
                NIL (GO G190) G191 (EXIT NIL))
              (EXIT (COND
                ((NULL |x|) (NULL |y|))
                ('T 'NIL))))))
          G102042 (EXIT G102042))))))

(DEFUN |ILIST;latex;$S;23| (|x| |$|)
  (PROG (|s|)
    (RETURN
      (SEQ
        (LETT |s| "\\left[" |ILIST;latex;$S;23|)
        (SEQ
          G190
          (COND
            ((NULL (COND ((NULL |x|) (QUOTE NIL)) ((QUOTE T) (QUOTE T))))

```

```

      (GO G191)))
    (SEQ
      (LETT |s|
        (STRCONC |s| (SPADCALL (QCAR |x|) (QREFELT |$| 49)))
        |ILIST;latex;$S;23|)
      (LETT |x| (QCDR |x|) |ILIST;latex;$S;23|)
      (EXIT
        (COND
          ((NULL (NULL |x|))
            (LETT |s| (STRCONC |s| ", ") |ILIST;latex;$S;23|))))))
    NIL
    (GO G190)
  G191
  (EXIT NIL))
(EXIT (STRCONC |s| " \right|")))))))

(DEFUN |ILIST;member?;$B;24| (|s| |x| $)
  (PROG (G102052)
    (RETURN
      (SEQ (EXIT (SEQ (SEQ G190
        (COND
          ((NULL (COND ((NULL |x|) 'NIL) ('T 'T)))
            (GO G191)))
        (SEQ (EXIT (COND
          ((SPADCALL |s| (QCAR |x|)
            (QREFELT $ 46))
          (PROGN
            (LETT G102052 'T
              |ILIST;member?;$B;24|)
            (GO G102052)))
          ('T
            (LETT |x| (QCDR |x|)
              |ILIST;member?;$B;24|))))))
        NIL (GO G190) G191 (EXIT NIL))
        (EXIT 'NIL)))
      G102052 (EXIT G102052))))))

(DEFUN |ILIST;concat!;3$;25| (|x| |y| |$|)
  (PROG (|z|)
    (RETURN
      (SEQ
        (COND
          ((NULL |x|)
            (COND
              ((NULL |y|) |x|)
              ((QUOTE T)
                (SEQ
                  (PUSH (SPADCALL |y| (QREFELT |$| 13)) |x|)
                  (QRPLACD |x| (SPADCALL |y| (QREFELT |$| 18)) (EXIT |x|))))))
          ((QUOTE T)
            (PUSH (SPADCALL |y| (QREFELT |$| 13)) |x|)
            (QRPLACD |x| (SPADCALL |y| (QREFELT |$| 18)) (EXIT |x|))))))
      ((QUOTE T)
        (PUSH (SPADCALL |y| (QREFELT |$| 13)) |x|)
        (QRPLACD |x| (SPADCALL |y| (QREFELT |$| 18)) (EXIT |x|))))))

```

```
(SEQ
(LETT |z| |x| |ILIST;concat!;3$;25|)
(SEQ
G190
(COND
((NULL (COND ((NULL (QCDR |z|)) (QUOTE NIL)) ((QUOTE T) (QUOTE T))))
(GO G191)))
(SEQ (EXIT (LETT |z| (QCDR |z|) |ILIST;concat!;3$;25|)))
NIL
(GO G190)
G191
(EXIT NIL))
(QRPLACD |z| |y|)
(EXIT |x|)))))))))

(DEFUN |ILIST;removeDuplicates!;2$;26| (|l| |$|)
(PROG (|f| |p| |pr| |pp|)
(RETURN
(SEQ
(LETT |p| |l| |ILIST;removeDuplicates!;2$;26|)
(SEQ
G190
(COND
((NULL (COND ((NULL |p|) (QUOTE NIL)) ((QUOTE T) (QUOTE T)))) (GO G191)))
(SEQ
(LETT |pp| |p| |ILIST;removeDuplicates!;2$;26|)
(LETT |f| (QCAR |p|) |ILIST;removeDuplicates!;2$;26|)
(LETT |p| (QCDR |p|) |ILIST;removeDuplicates!;2$;26|)
(EXIT
(SEQ
G190
(COND
((NULL
(COND
((NULL (LETT |pr| (QCDR |pp|) |ILIST;removeDuplicates!;2$;26|))
(QUOTE NIL))
((QUOTE T) (QUOTE T))))
(GO G191)))
(SEQ
(EXIT
(COND
((SPADCALL (QCAR |pr|) |f| (QREFELT |$| 46))
(QRPLACD |pp| (QCDR |pr|)))
((QUOTE T) (LETT |pp| |pr| |ILIST;removeDuplicates!;2$;26|))))))
NIL
(GO G190)
G191
(EXIT NIL))))))
NIL
(GO G190)
```

```

G191
(EXIT NIL))
(EXIT |1|))))))

(DEFUN |ILIST;sort!;M2$;27| (|f| |l| |$|)
(|ILIST;mergeSort| |f| |l| (SPADCALL |l| (QREFELT |$| 9)) |$|))

(DEFUN |ILIST;merge!;M3$;28| (|f| |p| |q| |$|)
(PROG (|r| |t|)
(RETURN
(SEQ
(COND
((NULL |p|) |q|)
((NULL |q|) |p|)
((EQ |p| |q|) (|error| "cannot merge a list into itself"))
((QUOTE T)
(SEQ
(COND
((SPADCALL (QCAR |p|) (QCAR |q|) |f|)
(SEQ
(LETT |r| (LETT |t| |p| |ILIST;merge!;M3$;28|) |ILIST;merge!;M3$;28|)
(EXIT (LETT |p| (QCDR |p|) |ILIST;merge!;M3$;28|))))
((QUOTE T)
(SEQ
(LETT |r| (LETT |t| |q| |ILIST;merge!;M3$;28|) |ILIST;merge!;M3$;28|)
(EXIT (LETT |q| (QCDR |q|) |ILIST;merge!;M3$;28|))))))
(SEQ
G190
(COND
((NULL
(COND
((OR (NULL |p|) (NULL |q|)) (QUOTE NIL))
((QUOTE T) (QUOTE T))))
(GO G191)))
(SEQ
(EXIT
(COND
((SPADCALL (QCAR |p|) (QCAR |q|) |f|)
(SEQ
(QRPLACD |t| |p|)
(LETT |t| |p| |ILIST;merge!;M3$;28|)
(EXIT (LETT |p| (QCDR |p|) |ILIST;merge!;M3$;28|))))
((QUOTE T)
(SEQ
(QRPLACD |t| |q|)
(LETT |t| |q| |ILIST;merge!;M3$;28|)
(EXIT (LETT |q| (QCDR |q|) |ILIST;merge!;M3$;28|)))))))
NIL
(GO G190)
G191

```

```

(EXIT NIL))
(QRPLACD |t| (COND ((NULL |p|) |q|) ((QUOTE T) |p|)))
(EXIT |r|))))))

(DEFUN |ILIST;split!;$I$;29| (|p| |n| $)
  (PROG (G102085 |q|)
    (RETURN
      (SEQ (COND
        ((< |n| 1) (|error| "index out of range"))
        ('T
          (SEQ (LETT |p|
            (SPADCALL |p|
              (PROG1 (LETT G102085 (- |n| 1)
                |ILIST;split!;$I$;29|)
                (|check-subtype| (>= G102085 0)
                  '(|NonNegativeInteger|) G102085))
                (QREFELT $ 32))
                |ILIST;split!;$I$;29|)
              (LETT |q| (QCDR |p|) |ILIST;split!;$I$;29|)
              (QRPLACD |p| NIL) (EXIT |q|)))))))))

(DEFUN |ILIST;mergeSort| (|f| |p| |n| $)
  (PROG (G102089 |l| |q|)
    (RETURN
      (SEQ (COND
        ((EQL |n| 2)
          (COND
            ((SPADCALL
              (SPADCALL (SPADCALL |p| (QREFELT $ 18))
                (QREFELT $ 13))
              (SPADCALL |p| (QREFELT $ 13)) |f|)
              (LETT |p| (SPADCALL |p| (QREFELT $ 28))
                |ILIST;mergeSort|))))))
        (EXIT (COND
          ((< |n| 3) |p|)
          ('T
            (SEQ (LETT |l|
              (PROG1 (LETT G102089 (QUOTIENT2 |n| 2)
                |ILIST;mergeSort|)
                (|check-subtype| (>= G102089 0)
                  '(|NonNegativeInteger|) G102089))
                |ILIST;mergeSort|)
              (LETT |q| (SPADCALL |p| |l| (QREFELT $ 57))
                |ILIST;mergeSort|)
              (LETT |p| (|ILIST;mergeSort| |f| |p| |l| $)
                |ILIST;mergeSort|)
              (LETT |q|
                (|ILIST;mergeSort| |f| |q| (- |n| |l|)
                  $)
                |ILIST;mergeSort|)
            )
          )
        )
      )
    )
  )

```

```

                                (EXIT (SPADCALL |f| |p| |q| (QREFELT $ 56)))))))))

(DEFUN |IndexedList| (&REST G102103 &AUX G102101)
  (DSETQ G102101 G102103)
  (PROG ()
    (RETURN
      (PROG (G102102)
        (RETURN
          (COND
            ((LETT G102102
              (|lassocShiftWithFunction| (|devalueList| G102101)
              (HGET |$ConstructorCache| '|IndexedList|)
              '|domainEqualList|)
              |IndexedList|)
            (|CDRwithIncrement| G102102))
          ('T
            (UNWIND-PROTECT
              (PROG1 (APPLY (|function| |IndexedList|) G102101)
                (LETT G102102 T |IndexedList|))
              (COND
                ((NOT G102102)
                  (HREM |$ConstructorCache| '|IndexedList|))))))))))

(DEFUN |IndexedList;| (|#1| |#2|)
  (PROG (DV$1 DV$2 |dv$| $ G102100 |pv$|)
    (RETURN
      (PROGN
        (LETT DV$1 (|devalue| |#1|) |IndexedList|)
        (LETT DV$2 (|devalue| |#2|) |IndexedList|)
        (LETT |dv$| (LIST '|IndexedList| DV$1 DV$2) |IndexedList|)
        (LETT $ (make-array 71) |IndexedList|)
        (QSETREFV $ 0 |dv$|)
        (QSETREFV $ 3
          (LETT |pv$|
            (|buildPredVector| 0 0
              (LIST (|HasCategory| |#1| '|SetCategory|))
              (|HasCategory| |#1|
                '|ConvertibleTo| (|InputForm|)))
              (LETT G102100
                (|HasCategory| |#1| '|OrderedSet|)
                |IndexedList|)
              (OR G102100
                (|HasCategory| |#1| '|SetCategory|))
              (|HasCategory| (|Integer|) '|OrderedSet|))
              (AND (|HasCategory| |#1|
                (LIST '|Evalable|
                  (|devalue| |#1|)))
                (|HasCategory| |#1| '|SetCategory|)))
              (OR (AND (|HasCategory| |#1|
                (LIST '|Evalable|

```

```

(|devaluate| |#1|)))
G102100)
(AND (|HasCategory| |#1|
      (LIST '|Evalable|
            (|devaluate| |#1|)))
      (|HasCategory| |#1|
        '(|SetCategory|))))))
(|IndexedList|))
(|haddProp| |$ConstructorCache| '|IndexedList| (LIST DV$1 DV$2)
  (CONS 1 $))
(|stuffDomainSlots| $)
(QSETREFV $ 6 |#1|)
(QSETREFV $ 7 |#2|)
(COND
  ((|testBitVector| |pv$| 1)
    (PROGN
      (QSETREFV $ 45
        (CONS (|dispatchFunction| |ILIST;coerce;$0f;21|) $))
      (QSETREFV $ 47
        (CONS (|dispatchFunction| |ILIST;=;2$B;22|) $))
      (QSETREFV $ 50
        (CONS (|dispatchFunction| |ILIST;latex;$S;23|) $))
      (QSETREFV $ 51
        (CONS (|dispatchFunction| |ILIST;member?;$S$B;24|) $))))))
(COND
  ((|testBitVector| |pv$| 1)
    (QSETREFV $ 53
      (CONS (|dispatchFunction|
              |ILIST;removeDuplicates!;2$;26|)
            $))))
$))))

(setf (get
  (QUOTE |IndexedList|)
  (QUOTE |infovec|))
  (LIST
    (QUOTE #(
      NIL NIL NIL NIL NIL NIL (|local| |#1|) (|local| |#2|)
      (|NonNegativeInteger|) |ILIST;#;$Nni;1| |ILIST;concat;$2$;2| (|Boolean|)
      |ILIST;eq?;2$B;3| |ILIST;first;$S;4| (QUOTE "first") |ILIST;elt;$first$;5|
      |ILIST;empty;$;6| |ILIST;empty?;$B;7| |ILIST;rest;2$;8| (QUOTE "rest")
      |ILIST;elt;$rest$;9| |ILIST;setfirst!;$2S;10| |ILIST;setelt;$first2$;11|
      |ILIST;setrest!;$3$;12| |ILIST;setelt;$rest2$;13| (|List| 6)
      |ILIST;construct;$L;14| |ILIST;parts;$L;15| |ILIST;reverse!;2$;16|
      |ILIST;reverse;2$;17| (|Integer|) |ILIST;minIndex;$I;18|
      |ILIST;rest;$Nni;$19| (0 . |cyclic?!|) |ILIST;copy;2$;20|
      (5 . |cycleEntry|) (|OutputForm|) (10 . |coerce|) (|List| |$|)
      (15 . |bracket|) (|List| 36) (20 . |list|) (25 . |commaSeparate|)
      (30 . |overbar|) (35 . |concat!|) (41 . |coerce|) (46 . |=|) (52 . |=|)
      (|String|) (58 . |latex|) (63 . |latex|) (68 . |member?|)
    ))
  )

```

```

|ILIST;concat!;3$;25| (74 . |removeDuplicates!|) (|Mapping| 11 6 6)
|ILIST;sort!;M2$;27| |ILIST;merge!;M3$;28| |ILIST;split!;$I$;29|
(|Mapping| 6 6 6) (|Equation| 6) (|List| 59) (|Mapping| 11 6) (|Void|)
(|UniversalSegment| 30) (QUOTE "last") (QUOTE "value") (|Mapping| 6 6)
(|InputForm|) (|SingleInteger|) (|List| 30) (|Union| 6 (QUOTE "failed"))))
(QUOTE #(
|~=| 79 |value| 85 |third| 90 |tail| 95 |swap!| 100 |split!| 107
|sorted?| 113 |sort!| 124 |sort| 135 |size?| 146 |setvalue!| 152
|setrest!| 158 |setlast!| 164 |setfirst!| 170 |setelt| 176
|setchildren!| 218 |select!| 224 |select| 230 |second| 236 |sample|
241 |reverse!| 245 |reverse| 250 |rest| 255 |removeDuplicates!|
266 |removeDuplicates| 271 |remove!| 276 |remove| 288 |reduce|
300 |qsetelt!| 321 |qelt| 328 |possiblyInfinite?| 334 |position|
339 |parts| 358 |nodes| 363 |node?| 368 |new| 374 |more?| 380
|minIndex| 386 |min| 391 |merge!| 397 |merge| 410 |members| 423
|member?| 428 |maxIndex| 434 |max| 439 |map!| 445 |map| 451 |list|
464 |less?| 469 |leaves| 475 |leaf?| 480 |latex| 485 |last| 490
|insert!| 501 |insert| 515 |indices| 529 |index?| 534 |hash| 540
|first| 545 |find| 556 |fill!| 562 |explicitlyFinite?| 568 |every?|
573 |eval| 579 |eq?| 605 |entry?| 611 |entries| 617 |empty?| 622
|empty| 627 |elt| 631 |distance| 674 |delete!| 680 |delete| 692
|cyclic?| 704 |cycleTail| 709 |cycleSplit!| 714 |cycleLength| 719
|cycleEntry| 724 |count| 729 |copyInto!| 741 |copy| 748 |convert|
753 |construct| 758 |concat!| 763 |concat| 775 |coerce| 798
|children| 803 |child?| 808 |any?| 814 |>=| 820 |>| 826 |=| 832
|<=| 838 |<| 844 |#| 850))
(QUOTE ((|shallowlyMutable| . 0) (|finiteAggregate| . 0)))
(CONS
(|makeByteWordVec2| 7 (QUOTE (0 0 0 0 0 0 0 0 0 3 0 0 7 4 0 0 7 1 2 4)))
(CONS
(QUOTE #(|ListAggregate&| |StreamAggregate&| |ExtensibleLinearAggregate&|
|FiniteLinearAggregate&| |UnaryRecursiveAggregate&| |LinearAggregate&| | |
|RecursiveAggregate&| |IndexedAggregate&| |Collection&|
|HomogeneousAggregate&| |OrderedSet&| |Aggregate&| |EltableAggregate&|
|Evalable&| |SetCategory&| NIL NIL |InnerEvalable&| NIL NIL
|BasicType&|)))
(CONS
(QUOTE #(
(|ListAggregate| 6) (|StreamAggregate| 6)
(|ExtensibleLinearAggregate| 6) (|FiniteLinearAggregate| 6)
(|UnaryRecursiveAggregate| 6) (|LinearAggregate| 6)
(|RecursiveAggregate| 6) (|IndexedAggregate| 30 6)
(|Collection| 6) (|HomogeneousAggregate| 6) (|OrderedSet|)
(|Aggregate|) (|EltableAggregate| 30 6) (|Evalable| 6) (|SetCategory|)
(|Type|) (|Eltable| 30 6) (|InnerEvalable| 6 6) (|CoercibleTo| 36)
(|ConvertibleTo| 67) (|BasicType|)))
(|makeByteWordVec2| 70
(QUOTE (1 0 11 0 33 1 0 0 0 35 1 6 36 0 37 1 36 0 38 39 1 40 0 36
41 1 36 0 38 42 1 36 0 0 43 2 40 0 0 36 44 1 0 36 0 45 2 6 11 0 0
46 2 0 11 0 0 47 1 6 48 0 49 1 0 48 0 50 2 0 11 6 0 51 1 0 0 0 53

```



```

2 1 11 0 0 1 1 0 6 0 1 1 0 6 0 1 1 0 0 0 1 3 0 62 0 30 30 1 2 0 0
0 30 57 1 3 11 0 1 2 0 11 54 0 1 1 3 0 0 1 2 0 0 54 0 55 1 3 0 0 1
2 0 0 54 0 1 2 0 11 0 8 1 2 0 6 0 6 1 2 0 0 0 0 23 2 0 6 0 6 1 2 0
6 0 6 21 3 0 6 0 30 6 1 3 0 6 0 63 6 1 3 0 6 0 64 6 1 3 0 0 0 19 0
24 3 0 6 0 14 6 22 3 0 6 0 65 6 1 2 0 0 0 38 1 2 0 0 61 0 1 2 0 0
61 0 1 1 0 6 0 1 0 0 0 1 1 0 0 0 28 1 0 0 0 29 2 0 0 0 8 32 1 0 0
0 18 1 1 0 0 53 1 1 0 0 1 2 1 0 6 0 1 2 0 0 61 0 1 2 1 0 6 0 1 2 0
0 61 0 1 4 1 6 58 0 6 6 1 2 0 6 58 0 1 3 0 6 58 0 6 1 3 0 6 0 30 6
1 2 0 6 0 30 1 1 0 11 0 1 2 1 30 6 0 1 3 1 30 6 0 30 1 2 0 30 61 0
1 1 0 25 0 27 1 0 38 0 1 2 1 11 0 0 1 2 0 0 8 6 1 2 0 11 0 8 1 1 5
30 0 31 2 3 0 0 0 1 2 3 0 0 0 1 3 0 0 54 0 0 56 2 3 0 0 0 1 3 0 0
54 0 0 1 1 0 25 0 1 2 1 11 6 0 51 1 5 30 0 1 2 3 0 0 0 1 2 0 0 66
0 1 3 0 0 58 0 0 1 2 0 0 66 0 1 1 0 0 6 1 2 0 11 0 8 1 1 0 25 0 1
1 0 11 0 1 1 1 48 0 50 2 0 0 0 8 1 1 0 6 0 1 3 0 0 6 0 30 1 3 0 0
0 0 30 1 3 0 0 0 0 30 1 3 0 0 6 0 30 1 1 0 69 0 1 2 0 11 30 0 1 1
1 68 0 1 2 0 0 0 8 1 1 0 6 0 13 2 0 70 61 0 1 2 0 0 0 6 1 1 0 11
0 1 2 0 11 61 0 1 3 6 0 0 6 6 1 3 6 0 0 25 25 1 2 6 0 0 59 1 2 6
0 0 60 1 2 0 11 0 0 12 2 1 11 6 0 1 1 0 25 0 1 1 0 11 0 17 0 0 0
16 2 0 6 0 30 1 3 0 6 0 30 6 1 2 0 0 0 63 1 2 0 6 0 64 1 2 0 0 0
19 20 2 0 6 0 14 15 2 0 6 0 65 1 2 0 30 0 0 1 2 0 0 0 63 1 2 0 0 0
30 1 2 0 0 0 63 1 2 0 0 0 30 1 1 0 11 0 33 1 0 0 0 1 1 0 0 0 1 1 0
8 0 1 1 0 0 0 35 2 1 8 6 0 1 2 0 8 61 0 1 3 0 0 0 0 30 1 1 0 0 0
34 1 2 67 0 1 1 0 0 25 26 2 0 0 0 0 52 2 0 0 0 6 1 1 0 0 38 1 2 0
0 0 6 1 2 0 0 6 0 10 2 0 0 0 0 1 1 1 36 0 45 1 0 38 0 1 2 1 11 0
0 1 2 0 11 61 0 1 2 3 11 0 0 1 2 3 11 0 0 1 2 1 11 0 0 47 2 3 11
0 0 1 2 3 11 0 0 1 1 0 8 0 9))))))
(QUOTE |lookupComplete|)))

```

INT.lsp BOOTSTRAP

INT depends on **OINTDOM** which depends on **ORDRING** which depends on **INT**. We need to break this cycle to build the algebra. So we keep a cached copy of the translated **INT** category which we can write into the **MID** directory. We compile the lisp code and copy the **INT.o** file to the **OUT** directory. This is eventually forcibly replaced by a recompiled version.

Note that this code is not included in the generated catdef.spad file.

— INT.lsp BOOTSTRAP —

```

(|/VERSIONCHECK| 2)

(DEFUN |INT;writeOMInt| (|dev| |x| |$|)
  (SEQ
    (COND

```

```

((|<| |x| 0)
 (SEQ
  (SPADCALL |dev| (QREFELT |$| 8))
  (SPADCALL |dev| "arith1" "unary_minus" (QREFELT |$| 10))
  (SPADCALL |dev| (|-| |x|) (QREFELT |$| 12))
  (EXIT (SPADCALL |dev| (QREFELT |$| 13))))))
((QUOTE T) (SPADCALL |dev| |x| (QREFELT |$| 12))))))

(DEFUN |INT;OMwrite;$S;2| (|x| |$|)
 (PROG (|sp| |dev| |s|)
 (RETURN
 (SEQ
  (LETT |s| "" |INT;OMwrite;$S;2|)
  (LETT |sp| (|OM-STRINGTOSTRINGPTR| |s|) |INT;OMwrite;$S;2|)
  (LETT |dev|
   (SPADCALL |sp| (SPADCALL (QREFELT |$| 15)) (QREFELT |$| 16))
   |INT;OMwrite;$S;2|)
  (SPADCALL |dev| (QREFELT |$| 17))
  (|INT;writeOMInt| |dev| |x| |$|)
  (SPADCALL |dev| (QREFELT |$| 18))
  (SPADCALL |dev| (QREFELT |$| 19))
  (LETT |s| (|OM-STRINGPTRTOSTRING| |sp|) |INT;OMwrite;$S;2|)
  (EXIT |s|))))))

(DEFUN |INT;OMwrite;$BS;3| (|x| |wholeObj| |$|)
 (PROG (|sp| |dev| |s|)
 (RETURN
 (SEQ
  (LETT |s| "" |INT;OMwrite;$BS;3|)
  (LETT |sp| (|OM-STRINGTOSTRINGPTR| |s|) |INT;OMwrite;$BS;3|)
  (LETT |dev|
   (SPADCALL |sp| (SPADCALL (QREFELT |$| 15)) (QREFELT |$| 16))
   |INT;OMwrite;$BS;3|)
  (COND (|wholeObj| (SPADCALL |dev| (QREFELT |$| 17))))
  (|INT;writeOMInt| |dev| |x| |$|)
  (COND (|wholeObj| (SPADCALL |dev| (QREFELT |$| 18))))
  (SPADCALL |dev| (QREFELT |$| 19))
  (LETT |s| (|OM-STRINGPTRTOSTRING| |sp|) |INT;OMwrite;$BS;3|)
  (EXIT |s|))))))

(DEFUN |INT;OMwrite;Omd$V;4| (|dev| |x| |$|)
 (SEQ
  (SPADCALL |dev| (QREFELT |$| 17))
  (|INT;writeOMInt| |dev| |x| |$|)
  (EXIT (SPADCALL |dev| (QREFELT |$| 18))))))

(DEFUN |INT;OMwrite;Omd$BV;5| (|dev| |x| |wholeObj| |$|)
 (SEQ
  (COND (|wholeObj| (SPADCALL |dev| (QREFELT |$| 17))))
  (|INT;writeOMInt| |dev| |x| |$|)

```

```

(EXIT (COND (|wholeObj| (SPADCALL |dev| (QREFELT |$| 18))))))

(PUT (QUOTE |INT;zero?;$B;6|) (QUOTE |SPADreplace|) (QUOTE ZEROP))

(DEFUN |INT;zero?;$B;6| (|x| |$|) (ZEROP |x|))

(PUT (QUOTE |INT;Zero;$;7|) (QUOTE |SPADreplace|) (QUOTE (XLAM NIL 0)))

(DEFUN |INT;Zero;$;7| (|x|) 0)

(PUT (QUOTE |INT;One;$;8|) (QUOTE |SPADreplace|) (QUOTE (XLAM NIL 1)))

(DEFUN |INT;One;$;8| (|x|) 1)

(PUT (QUOTE |INT;base;$;9|) (QUOTE |SPADreplace|) (QUOTE (XLAM NIL 2)))

(DEFUN |INT;base;$;9| (|x|) 2)

(PUT (QUOTE |INT;copy;2$;10|) (QUOTE |SPADreplace|) (QUOTE (XLAM (|x|) |x|)))

(DEFUN |INT;copy;2$;10| (|x| |$|) |x|)

(PUT
  (QUOTE |INT;inc;2$;11|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|x|) (|+| |x| 1))))

(DEFUN |INT;inc;2$;11| (|x| |$|) (|+| |x| 1))

(PUT
  (QUOTE |INT;dec;2$;12|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|x|) (|-| |x| 1))))

(DEFUN |INT;dec;2$;12| (|x| |$|) (|-| |x| 1))

(PUT (QUOTE |INT;hash;2$;13|) (QUOTE |SPADreplace|) (QUOTE SXHASH))

(DEFUN |INT;hash;2$;13| (|x| |$|) (SXHASH |x|))

(PUT (QUOTE |INT;negative?;$B;14|) (QUOTE |SPADreplace|) (QUOTE MINUSP))

(DEFUN |INT;negative?;$B;14| (|x| |$|) (MINUSP |x|))

(DEFUN |INT;coerce;$Of;15| (|x| |$|) (SPADCALL |x| (QREFELT |$| 35)))

(PUT
  (QUOTE |INT;coerce;2$;16|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|m|) |m|)))

```

```

(DEFUN |INT;coerce;2$;16| (|m| |$|) |m|)

(PUT
  (QUOTE |INT;convert;2$;17|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|x|) |x|)))

(DEFUN |INT;convert;2$;17| (|x| |$|) |x|)

(PUT
  (QUOTE |INT;length;2$;18|)
  (QUOTE |SPADreplace|)
  (QUOTE |INTEGER-LENGTH|))

(DEFUN |INT;length;2$;18| (|a| |$|) (|INTEGER-LENGTH| |a|))

(DEFUN |INT;addmod;4$;19| (|a| |b| |p| $)
  (PROG (|c| G86338)
    (RETURN
      (SEQ (EXIT (SEQ (SEQ (LETT |c| (+ |a| |b|) |INT;addmod;4$;19|)
        (EXIT (COND
          ((NULL (< |c| |p|))
            (PROGN
              (LETT G86338 (- |c| |p|)
                |INT;addmod;4$;19|)
              (GO G86338))))))
          (EXIT |c|)))
        G86338 (EXIT G86338))))))

(DEFUN |INT;submod;4$;20| (|a| |b| |p| |$|)
  (PROG (|c|)
    (RETURN
      (SEQ
        (LETT |c| (|-| |a| |b|) |INT;submod;4$;20|)
        (EXIT (COND ((|<| |c| 0) (|+| |c| |p|)) ((QUOTE T) |c|)))))))

(DEFUN |INT;mulmod;4$;21| (|a| |b| |p| |$|) (REMAINDER2 (|*| |a| |b|) |p|))

(DEFUN |INT;convert;$F;22| (|x| |$|) (SPADCALL |x| (QREFELT |$| 44)))

(PUT
  (QUOTE |INT;convert;$Df;23|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|x|) (FLOAT |x| |MOST-POSITIVE-LONG-FLOAT|))))

(DEFUN |INT;convert;$Df;23| (|x| |$|) (FLOAT |x| |MOST-POSITIVE-LONG-FLOAT|))

(DEFUN |INT;convert;$If;24| (|x| |$|) (SPADCALL |x| (QREFELT |$| 49)))

```

```

(PUT (QUOTE |INT;convert;$S;25|) (QUOTE |SPADreplace|) (QUOTE STRINGIMAGE))

(DEFUN |INT;convert;$S;25| (|x| |$|) (STRINGIMAGE |x|))

(DEFUN |INT;latex;$S;26| (|x| |$|)
  (PROG (|s|)
    (RETURN
      (SEQ
        (LETT |s| (STRINGIMAGE |x|) |INT;latex;$S;26|)
        (COND ((|<| -1 |x|) (COND ((|<| |x| 10) (EXIT |s|))))))
      (EXIT (STRCONC "{" (STRCONC |s| "}")")))))

(DEFUN |INT;positiveRemainder;3$;27| (|a| |b| |$|)
  (PROG (|r|)
    (RETURN
      (COND
        ((MINUSP (LETT |r| (REMAINDER2 |a| |b|) |INT;positiveRemainder;3$;27|))
          (COND
            ((MINUSP |b|) (| - | |r| |b|))
            ((QUOTE T) (|+| |r| |b|))))
        ((QUOTE T) |r|))))))

(PUT
  (QUOTE |INT;reducedSystem;2M;28|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|m| |m|))))

(DEFUN |INT;reducedSystem;2M;28| (|m| |$|) |m|)

(DEFUN |INT;reducedSystem;MVR;29| (|m| |v| |$|) (CONS |m| (QUOTE |vec|)))

(PUT (QUOTE |INT;abs;2$;30|) (QUOTE |SPADreplace|) (QUOTE ABS))

(DEFUN |INT;abs;2$;30| (|x| |$|) (ABS |x|))

(PUT (QUOTE |INT;random;$;31|) (QUOTE |SPADreplace|) (QUOTE |random|))

(DEFUN |INT;random;$;31| (|$|) (|random|))

(PUT (QUOTE |INT;random;2$;32|) (QUOTE |SPADreplace|) (QUOTE RANDOM))

(DEFUN |INT;random;2$;32| (|x| |$|) (RANDOM |x|))

(PUT (QUOTE |INT;=;2$B;33|) (QUOTE |SPADreplace|) (QUOTE EQL))

(DEFUN |INT;=;2$B;33| (|x| |y| |$|) (EQL |x| |y|))

(PUT (QUOTE |INT;<;2$B;34|) (QUOTE |SPADreplace|) (QUOTE |<|))

(DEFUN |INT;<;2$B;34| (|x| |y| |$|) (|<| |x| |y|))

```

```

(PUT (QUOTE |INT;-;2$;35|) (QUOTE |SPADreplace|) (QUOTE |-|))
(DEFUN |INT;-;2$;35| (|x| |$|) (|-| |x|))
(PUT (QUOTE |INT;+;3$;36|) (QUOTE |SPADreplace|) (QUOTE |+|))
(DEFUN |INT;+;3$;36| (|x| |y| |$|) (|+| |x| |y|))
(PUT (QUOTE |INT;-;3$;37|) (QUOTE |SPADreplace|) (QUOTE |-|))
(DEFUN |INT;-;3$;37| (|x| |y| |$|) (|-| |x| |y|))
(PUT (QUOTE |INT;*;3$;38|) (QUOTE |SPADreplace|) (QUOTE |*|))
(DEFUN |INT;*;3$;38| (|x| |y| |$|) (|*| |x| |y|))
(PUT (QUOTE |INT;*;3$;39|) (QUOTE |SPADreplace|) (QUOTE |*|))
(DEFUN |INT;*;3$;39| (|m| |y| |$|) (|*| |m| |y|))
(PUT (QUOTE |INT;**;$Nni$;40|) (QUOTE |SPADreplace|) (QUOTE EXPT))
(DEFUN |INT;**;$Nni$;40| (|x| |n| |$|) (EXPT |x| |n|))
(PUT (QUOTE |INT;odd?;$B;41|) (QUOTE |SPADreplace|) (QUOTE ODDP))
(DEFUN |INT;odd?;$B;41| (|x| |$|) (ODDP |x|))
(PUT (QUOTE |INT;max;3$;42|) (QUOTE |SPADreplace|) (QUOTE MAX))
(DEFUN |INT;max;3$;42| (|x| |y| |$|) (MAX |x| |y|))
(PUT (QUOTE |INT;min;3$;43|) (QUOTE |SPADreplace|) (QUOTE MIN))
(DEFUN |INT;min;3$;43| (|x| |y| |$|) (MIN |x| |y|))
(PUT (QUOTE |INT;divide;2$R;44|) (QUOTE |SPADreplace|) (QUOTE DIVIDE2))
(DEFUN |INT;divide;2$R;44| (|x| |y| |$|) (DIVIDE2 |x| |y|))
(PUT (QUOTE |INT;quo;3$;45|) (QUOTE |SPADreplace|) (QUOTE QUOTIENT2))
(DEFUN |INT;quo;3$;45| (|x| |y| |$|) (QUOTIENT2 |x| |y|))
(PUT (QUOTE |INT;rem;3$;46|) (QUOTE |SPADreplace|) (QUOTE REMAINDER2))
(DEFUN |INT;rem;3$;46| (|x| |y| |$|) (REMAINDER2 |x| |y|))
(PUT (QUOTE |INT;shift;3$;47|) (QUOTE |SPADreplace|) (QUOTE ASH))

```

```

(DEFUN |INT;shift;3$;47| (|x| |y| |$|) (ASH |x| |y|))

(DEFUN |INT;exquo;2$U;48| (|x| |y| |$|)
  (COND
    ((OR (ZEROP |y|) (NULL (ZEROP (REMAINDER2 |x| |y|)))) (CONS 1 "failed"))
    ((QUOTE T) (CONS 0 (QUOTIENT2 |x| |y|)))))

(DEFUN |INT;recip;$U;49| (|x| |$|)
  (COND
    ((OR (EQL |x| 1) (EQL |x| -1)) (CONS 0 |x|))
    ((QUOTE T) (CONS 1 "failed"))))

(PUT (QUOTE |INT;gcd;3$;50|) (QUOTE |SPADreplace|) (QUOTE GCD))

(DEFUN |INT;gcd;3$;50| (|x| |y| |$|) (GCD |x| |y|))

(DEFUN |INT;unitNormal;$R;51| (|x| |$|)
  (COND
    ((|<| |x| 0) (VECTOR -1 (|-| |x|) -1))
    ((QUOTE T) (VECTOR 1 |x| 1))))

(PUT (QUOTE |INT;unitCanonical;2$;52|) (QUOTE |SPADreplace|) (QUOTE ABS))

(DEFUN |INT;unitCanonical;2$;52| (|x| |$|) (ABS |x|))

(DEFUN |INT;solveLinearPolynomialEquation| (|lp| |p| |$|)
  (SPADCALL |lp| |p| (QREFELT |$| 91)))

(DEFUN |INT;squareFreePolynomial| (|p| |$|) (SPADCALL |p| (QREFELT |$| 95)))

(DEFUN |INT;factorPolynomial| (|p| $)
  (PROG (|pp| G86409)
    (RETURN
      (SEQ (LETT |pp| (SPADCALL |p| (QREFELT $ 96))
        |INT;factorPolynomial|)
        (EXIT (COND
          ((EQL (SPADCALL |pp| (QREFELT $ 97))
            (SPADCALL |p| (QREFELT $ 97)))
            (SPADCALL |p| (QREFELT $ 99)))
          ('T
            (SPADCALL (SPADCALL |pp| (QREFELT $ 99))
              (SPADCALL (CONS #'|INT;factorPolynomial!0| $)
                (SPADCALL
                  (PROG2 (LETT G86409
                    (SPADCALL
                      (SPADCALL |p| (QREFELT $ 97))
                      (SPADCALL |pp| (QREFELT $ 97))
                      (QREFELT $ 81))
                    |INT;factorPolynomial|)

```

```

                                (QCDR G86409)
                                (|check-union| (QEQCAR G86409 0) $
                                G86409))
                                (QREFELT $ 102))
                                (QREFELT $ 106))
                                (QREFELT $ 108)))))))))

(DEFUN |INT;factorPolynomial!0| (|#1| |$|) (SPADCALL |#1| (QREFELT |$| 100)))

(DEFUN |INT;factorSquareFreePolynomial| (|p| |$|)
  (SPADCALL |p| (QREFELT |$| 109)))

(DEFUN |INT;gcdPolynomial;3Sup;57| (|p| |q| |$|)
  (COND
    ((SPADCALL |p| (QREFELT |$| 110)) (SPADCALL |q| (QREFELT |$| 111)))
    ((SPADCALL |q| (QREFELT |$| 110)) (SPADCALL |p| (QREFELT |$| 111)))
    ((QUOTE T) (SPADCALL (LIST |p| |q|) (QREFELT |$| 114)))))

(DEFUN |Integer| ()
  (PROG ()
    (RETURN
      (PROG (G86434)
        (RETURN
          (COND
            ((LETT G86434 (HGET |$ConstructorCache| '|Integer|)
              |Integer|)
              (|CDRwithIncrement| (CDAR G86434)))
            ('T
              (UNWIND-PROTECT
                (PROG1 (CDDAR (HPUT |$ConstructorCache| '|Integer|
                  (LIST
                    (CONS NIL (CONS 1 (|Integer|;|))))))
                  (LETT G86434 T |Integer|))
                (COND
                  ((NOT G86434) (HREM |$ConstructorCache| '|Integer|)))))))))))

(DEFUN |Integer;| ()
  (PROG (|dv$| $ |pv$|)
    (RETURN
      (PROGN
        (LETT |dv$| '(|Integer|) |Integer|)
        (LETT $ (make-array 130) |Integer|)
        (QSETREFV $ 0 |dv$|)
        (QSETREFV $ 3
          (LETT |pv$| (|buildPredVector| 0 0 NIL) |Integer|))
        (|haddProp| |$ConstructorCache| '|Integer| NIL (CONS 1 $))
        (|stuffDomainSlots| $)
        (QSETREFV $ 69
          (QSETREFV $ 68 (CONS (|dispatchFunction| |INT;*;3$;39|) $)))
        $))))

```



```

(setf (get
  (QUOTE |Integer|)
  (QUOTE |infovec|))
  (LIST
    (QUOTE
      #(NIL NIL NIL NIL NIL NIL (|Void|) (|OpenMathDevice|) (0 . |OMputApp|)
        (|String|) (5 . |OMputSymbol|) (|Integer|) (12 . |OMputInteger|)
        (18 . |OMputEndApp|) (|OpenMathEncoding|) (23 . |OMencodingXML|)
        (27 . |OMopenString|) (33 . |OMputObject|) (38 . |OMputEndObject|)
        (43 . |OMclose|) |INT;OMwrite;$S;2| (|Boolean|) |INT;OMwrite;$BS;3|
        |INT;OMwrite;Omd$V;4| |INT;OMwrite;Omd$BV;5| |INT;zero?;$B;6|
        (CONS IDENTITY (FUNCALL (|dispatchFunction| |INT;Zero;$;7|) |$|))
        (CONS IDENTITY (FUNCALL (|dispatchFunction| |INT;One;$;8|) |$|))
        |INT;base;$;9| |INT;copy;2$;10| |INT;inc;2$;11| |INT;dec;2$;12|
        |INT;hash;2$;13| |INT;negative?;$B;14| (|OutputForm|)
        (48 . |outputForm|) |INT;coerce;$Of;15| |INT;coerce;2$;16|
        |INT;convert;2$;17| |INT;length;2$;18| |INT;addmod;4$;19|
        |INT;submod;4$;20| |INT;mulmod;4$;21| (|Float|) (53 . |coerce|)
        |INT;convert;$F;22| (|DoubleFloat|) |INT;convert;$Df;23| (|InputForm|)
        (58 . |convert|) |INT;convert;$If;24| |INT;convert;$S;25|
        |INT;latex;$S;26| |INT;positiveRemainder;3$;27| (|Matrix| 11)
        (|Matrix| |$|) |INT;reducedSystem;2M;28|
        (|Record| (|:| |mat| 54) (|:| |vec| (|Vector| 11)))
        (|Vector| |$|) |INT;reducedSystem;MVR;29| |INT;abs;2$;30|
        |INT;random;$;31| |INT;random;2$;32| |INT;=;2$B;33|
        |INT;<;2$B;34| |INT;-;2$;35| |INT;+;3$;36| |INT;-;3$;37| NIL NIL
        (|NonNegativeInteger|) |INT;**;$Nni$;40| |INT;odd?;$B;41|
        |INT;max;3$;42| |INT;min;3$;43|
        (|Record| (|:| |quotient| |$|) (|:| |remainder| |$|))
        |INT;divide;2$R;44| |INT;quo;3$;45| |INT;rem;3$;46| |INT;shift;3$;47|
        (|Union| |$| (QUOTE "failed")) |INT;exquo;2$U;48| |INT;recip;$U;49|
        |INT;gcd;3$;50|
        (|Record| (|:| |unit| |$|) (|:| |canonical| |$|) (|:| |associate| |$|))
        |INT;unitNormal;$R;51| |INT;unitCanonical;2$;52|
        (|Union| 88 (QUOTE "failed")) (|List| 89)
        (|SparseUnivariatePolynomial| 11)
        (|IntegerSolveLinearPolynomialEquation|)
        (63 . |solveLinearPolynomialEquation|) (|Factored| 93)
        (|SparseUnivariatePolynomial| |$$|)
        (|UnivariatePolynomialSquareFree| |$$| 93) (69 . |squareFree|)
        (74 . |primitivePart|) (79 . |leadingCoefficient|)
        (|GaloisGroupFactorizer| 93) (84 . |factor|) (89 . |coerce|)
        (|Factored| |$|) (94 . |factor|) (|Mapping| 93 |$$|)
        (|Factored| |$$|) (|FactoredFunctions2| |$$| 93) (99 . |map|)
        (|FactoredFunctionUtilities| 93) (105 . |mergeFactors|)
        (111 . |factorSquareFree|) (116 . |zero?|) (121 . |unitCanonical|)
        (|List| 93) (|HeuGcd| 93) (126 . |gcd|)
        (|SparseUnivariatePolynomial| |$|) |INT;gcdPolynomial;3Sup;57|
        (|Union| 118 (QUOTE "failed")) (|Fraction| 11)

```

```

(|PatternMatchResult| 11 |$|) (|Pattern| 11)
(|Union| 11 (QUOTE "failed")) (|Union| 123 (QUOTE "failed"))
(|List| |$|)
(|Record| (|:| |coef| 123) (|:| |generator| |$|))
(|Record| (|:| |coef1| |$|) (|:| |coef2| |$|))
(|Union| 125 (QUOTE "failed"))
(|Record| (|:| |coef1| |$|) (|:| |coef2| |$|) (|:| |generator| |$|))
(|PositiveInteger|) (|SingleInteger|))
(QUOTE #(|~|=| 131 |zero?| 137 |unitNormal| 142 |unitCanonical| 147
|unit?| 152 |symmetricRemainder| 157 |subtractIfCan| 163 |submod| 169
|squareFreePart| 176 |squareFree| 181 |sizeLess?| 186 |sign| 192
|shift| 197 |sample| 203 |retractIfCan| 207 |retract| 212 |rem| 217
|reducedSystem| 223 |recip| 234 |rationalIfCan| 239 |rational?| 244
|rational| 249 |random| 254 |quo| 263 |principalIdeal| 269
|prime?| 274 |powmod| 279 |positiveRemainder| 286 |positive?| 292
|permutation| 297 |patternMatch| 303 |one?| 310 |odd?| 315
|nextItem| 320 |negative?| 325 |multiEuclidean| 330 |mulmod| 336
|min| 343 |max| 349 |mask| 355 |length| 360 |lcm| 365 |latex| 376
|invmod| 381 |init| 387 |inc| 391 |hash| 396 |gcdPolynomial| 406
|gcd| 412 |factorial| 423 |factor| 428 |extendedEuclidean| 433
|exquo| 446 |expressIdealMember| 452 |even?| 458
|euclideanSize| 463 |divide| 468 |differentiate| 474 |dec| 485
|copy| 490 |convert| 495 |coerce| 525 |characteristic| 545
|bit?| 549 |binomial| 555 |base| 561 |associates?| 565
|addmod| 571 |abs| 578 |^| 583 |Zero| 595 |One| 599
|OMwrite| 603 D 627 |>=| 638 |>| 644 |=| 650 |<=| 656 |<| 662
|-| 668 |+| 679 |**| 685 |*| 697))
(QUOTE (
(|infinite| . 0) (|noetherian| . 0) (|canonicalsClosed| . 0)
(|canonical| . 0) (|canonicalUnitNormal| . 0)
(|multiplicativeValuation| . 0) (|noZeroDivisors| . 0)
((|commutative| "*") . 0) (|rightUnitary| . 0) (|leftUnitary| . 0)
(|unitsKnown| . 0)))
(CONS
(|makeByteWordVec2| 1
(QUOTE (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)))
(CONS
(QUOTE #(
|IntegerNumberSystem&| |EuclideanDomain&| |UniqueFactorizationDomain&|
NIL NIL |GcdDomain&| |IntegralDomain&| |Algebra&| NIL NIL
|DifferentialRing&| |OrderedRing&| NIL NIL |Module&| NIL NIL
|Ring&| NIL NIL NIL NIL NIL |AbelianGroup&| NIL NIL
|AbelianMonoid&| |Monoid&| NIL NIL |OrderedSet&|
|AbelianSemiGroup&| |SemiGroup&| NIL |SetCategory&|
NIL NIL NIL NIL NIL NIL |RetractableTo&| NIL |BasicType&| NIL))
(CONS
(QUOTE #(
(|IntegerNumberSystem|) (|EuclideanDomain|)
(|UniqueFactorizationDomain|) (|PrincipalIdealDomain|)

```

```

(|OrderedIntegralDomain|) (|GcdDomain|) (|IntegralDomain|)
(|Algebra| |$$|) (|CharacteristicZero|) (|LinearlyExplicitRingOver| 11)
(|DifferentialRing|) (|OrderedRing|) (|CommutativeRing|) (|EntireRing|)
(|Module| |$$|) (|OrderedAbelianGroup|) (|BiModule| |$$| |$$|)
(|Ring|) (|OrderedCancellationAbelianMonoid|) (|LeftModule| |$$|)
(|Rng|) (|RightModule| |$$|) (|OrderedAbelianMonoid|) (|AbelianGroup|)
(|OrderedAbelianSemiGroup|) (|CancellationAbelianMonoid|)
(|AbelianMonoid|) (|Monoid|) (|StepThrough|) (|PatternMatchable| 11)
(|OrderedSet|) (|AbelianSemiGroup|) (|SemiGroup|) (|RealConstant|)
(|SetCategory|) (|OpenMath|) (|ConvertibleTo| 9)
(|ConvertibleTo| 43) (|ConvertibleTo| 46)
(|CombinatorialFunctionCategory|) (|ConvertibleTo| 120)
(|ConvertibleTo| 48) (|RetractableTo| 11) (|ConvertibleTo| 11)
(|BasicType|) (|CoercibleTo| 34)))
(|makeByteWordVec2| 129 (QUOTE (1 7 6 0 8 3 7 6 0 9 9 10 2 7 6 0 11
12 1 7 6 0 13 0 14 0 15 2 7 0 9 14 16 1 7 6 0 17 1 7 6 0 18 1 7 6 0
19 1 34 0 11 35 1 43 0 11 44 1 48 0 11 49 2 90 87 88 89 91 1 94 92
93 95 1 93 0 0 96 1 93 2 0 97 1 98 92 93 99 1 93 0 2 100 1 0 101 0
102 2 105 92 103 104 106 2 107 92 92 92 108 1 98 92 93 109 1 93 21
0 110 1 93 0 0 111 1 113 93 112 114 2 0 21 0 0 1 1 0 21 0 25 1 0 84
0 85 1 0 0 0 86 1 0 21 0 1 2 0 0 0 0 1 2 0 80 0 0 1 3 0 0 0 0 41
1 0 0 0 1 1 0 101 0 1 2 0 21 0 0 1 1 0 11 0 1 2 0 0 0 0 79 0 0 0 1
1 0 121 0 1 1 0 11 0 1 2 0 0 0 0 78 2 0 57 55 58 59 1 0 54 55 56 1
0 80 0 82 1 0 117 0 1 1 0 21 0 1 1 0 118 0 1 1 0 0 0 62 0 0 0 61 2
0 0 0 0 77 1 0 124 123 1 1 0 21 0 1 3 0 0 0 0 0 1 2 0 0 0 0 53 1 0
21 0 1 2 0 0 0 0 1 3 0 119 0 120 119 1 1 0 21 0 1 1 0 21 0 72 1 0
80 0 1 1 0 21 0 33 2 0 122 123 0 1 3 0 0 0 0 0 42 2 0 0 0 0 74 2 0
0 0 0 73 1 0 0 0 1 1 0 0 0 39 1 0 0 123 1 2 0 0 0 0 1 1 0 9 0 52 2
0 0 0 0 1 0 0 0 1 1 0 0 0 30 1 0 0 0 32 1 0 129 0 1 2 0 115 115 115
116 2 0 0 0 0 83 1 0 0 123 1 1 0 0 0 1 1 0 101 0 102 3 0 126 0 0 0
1 2 0 127 0 0 1 2 0 80 0 0 81 2 0 122 123 0 1 1 0 21 0 1 1 0 70 0
1 2 0 75 0 0 76 1 0 0 0 1 2 0 0 0 70 1 1 0 0 0 31 1 0 0 0 29 1 0 9
0 51 1 0 46 0 47 1 0 43 0 45 1 0 48 0 50 1 0 120 0 1 1 0 11 0 38 1
0 0 11 37 1 0 0 11 37 1 0 0 0 1 1 0 34 0 36 0 0 70 1 2 0 21 0 0 1
2 0 0 0 0 1 0 0 0 28 2 0 21 0 0 1 3 0 0 0 0 40 1 0 0 0 60 2 0 0
0 70 1 2 0 0 0 128 1 0 0 0 26 0 0 0 27 3 0 6 7 0 21 24 2 0 9 0 21
22 2 0 6 7 0 23 1 0 9 0 20 1 0 0 0 1 2 0 0 0 70 1 2 0 21 0 0 1 2
0 21 0 0 1 2 0 21 0 0 63 2 0 21 0 0 1 2 0 21 0 0 64 2 0 0 0 0 67
1 0 0 0 65 2 0 0 0 0 66 2 0 0 0 70 71 2 0 0 0 128 1 2 0 0 0 0 68
2 0 0 11 0 69 2 0 0 70 0 1 2 0 0 128 0 1))))))
(QUOTE |lookupComplete|)))

(setf (get (QUOTE |Integer|) (QUOTE NILADIC)) T)

```

ISTRING.lsp BOOTSTRAP

ISTRING depends on a chain of files. We need to break this cycle to build the algebra. So we keep a cached copy of the translated **ISTRING** category which we can write into the **MID** directory. We compile the lisp code and copy the **ISTRING.o** file to the **OUT** directory. This is eventually forcibly replaced by a recompiled version.

Note that this code is not included in the generated catdef.spad file.

— ISTRING.lsp BOOTSTRAP —

```
(|/VERSIONCHECK| 2)

(PUT '|ISTRING;new;NniC$;1| '|SPADreplace| 'MAKE-FULL-CVEC)

(DEFUN |ISTRING;new;NniC$;1| (|n| |c| |$|) (|MAKE-FULL-CVEC| |n| |c|))

(PUT '|ISTRING;empty;$;2| '|SPADreplace|
  '(XLAM NIL (MAKE-FULL-CVEC 0)))

(DEFUN |ISTRING;empty;$;2| (|s|) (|MAKE-FULL-CVEC| 0))

(DEFUN |ISTRING;empty?;$B;3| (|s| |$|) (EQL (QCSIZE |s|) 0))

(PUT '|ISTRING;#;$Nni;4| '|SPADreplace| 'QCSIZE)

(DEFUN |ISTRING;#;$Nni;4| (|s| |$|) (QCSIZE |s|))

(PUT '|ISTRING;=;2$B;5| '|SPADreplace| 'EQUAL)

(DEFUN |ISTRING;=;2$B;5| (|s| |t| |$|) (EQUAL |s| |t|))

(PUT '|ISTRING;<;2$B;6| '|SPADreplace|
  '(XLAM (|s| |t|) (CGREATERP |t| |s|)))

(DEFUN |ISTRING;<;2$B;6| (|s| |t| |$|) (CGREATERP |t| |s|))

(PUT '|ISTRING;concat;3$;7| '|SPADreplace| 'STRCONC)

(DEFUN |ISTRING;concat;3$;7| (|s| |t| |$|) (STRCONC |s| |t|))

(PUT '|ISTRING;copy;2$;8| '|SPADreplace| 'COPY-SEQ)

(DEFUN |ISTRING;copy;2$;8| (|s| |$|) (|COPY-SEQ| |s|))

(DEFUN |ISTRING;insert;2$I$;9| (|s| |t| |i| |$|)
  (SPADCALL
    (SPADCALL
      (SPADCALL |s|
```

```

        (SPADCALL (QREFELT $ 6) (- |i| 1) (QREFELT $ 20))
        (QREFELT $ 21))
    |t| (QREFELT $ 16))
    (SPADCALL |s| (SPADCALL |i| (QREFELT $ 22)) (QREFELT $ 21))
    (QREFELT $ 16)))

(DEFUN |ISTRING;coerce;$Of;10| (|s| |$|) (SPADCALL |s| (QREFELT |$| 26)))

(DEFUN |ISTRING;minIndex;$I;11| (|s| |$|) (QREFELT |$| 6))

(DEFUN |ISTRING;upperCase!;2$;12| (|s| $)
  (SPADCALL (ELT $ 31) |s| (QREFELT $ 33)))

(DEFUN |ISTRING;lowerCase!;2$;13| (|s| $)
  (SPADCALL (ELT $ 36) |s| (QREFELT $ 33)))

(DEFUN |ISTRING;latex;$S;14| (|s| $)
  (STRCONC "\mbox{" (STRCONC |s| "'')"))

(DEFUN |ISTRING;replace;$Us2$;15| (|s| |sg| |t| $)
  (PROG (|l| |m| |n| |h| G91425 |r| G91433 G91432 |i| G91431
    |k|)
    (RETURN
      (SEQ (LETT |l| (- (SPADCALL |sg| (QREFELT $ 39)) (QREFELT $ 6))
        |ISTRING;replace;$Us2$;15|)
        (LETT |m| (SPADCALL |s| (QREFELT $ 13))
          |ISTRING;replace;$Us2$;15|)
        (LETT |n| (SPADCALL |t| (QREFELT $ 13))
          |ISTRING;replace;$Us2$;15|)
        (LETT |h|
          (COND
            ((SPADCALL |sg| (QREFELT $ 40))
              (- (SPADCALL |sg| (QREFELT $ 41)) (QREFELT $ 6)))
            ('T (- (SPADCALL |s| (QREFELT $ 42)) (QREFELT $ 6))))
          |ISTRING;replace;$Us2$;15|)
        (COND
          ((OR (OR (< |l| 0) (NULL (< |h| |m|))) (< |h| (- |l| 1)))
            (EXIT (|error| "index out of range"))))
        (LETT |r|
          (SPADCALL
            (PROG1 (LETT G91425
              (+ (- |m| (+ (- |h| |l|) 1)) |n|)
              |ISTRING;replace;$Us2$;15|)
              (|check-subtype| (>= G91425 0)
                '(|NonNegativeInteger|) G91425))
              (SPADCALL (QREFELT $ 43)) (QREFELT $ 9))
            |ISTRING;replace;$Us2$;15|)
        (SEQ (LETT |i| 0 |ISTRING;replace;$Us2$;15|)
          (LETT G91433 (- |l| 1) |ISTRING;replace;$Us2$;15|)
          (LETT |k| 0 |ISTRING;replace;$Us2$;15|) G190

```

```

(COND ((QSGREATERP |i| G91433) (GO G191)))
(SEQ (EXIT (QESSET |r| |k| (QENUM |s| |i|))))
(LETT |k|
  (PROG1 (QSADD1 |k|)
    (LETT |i| (QSADD1 |i|)
      |ISTRING;replace;$Us2$;15|)
    |ISTRING;replace;$Us2$;15|)
  (GO G190) G191 (EXIT NIL))
(SEQ (LETT |i| 0 |ISTRING;replace;$Us2$;15|)
  (LETT G91432 (- |n| 1) |ISTRING;replace;$Us2$;15|)
  (LETT |k| |k| |ISTRING;replace;$Us2$;15|) G190
  (COND ((QSGREATERP |i| G91432) (GO G191)))
  (SEQ (EXIT (QESSET |r| |k| (QENUM |t| |i|))))
  (LETT |k|
    (PROG1 (+ |k| 1)
      (LETT |i| (QSADD1 |i|)
        |ISTRING;replace;$Us2$;15|)
        |ISTRING;replace;$Us2$;15|)
      (GO G190) G191 (EXIT NIL))
  (SEQ (LETT |i| (+ |h| 1) |ISTRING;replace;$Us2$;15|)
    (LETT G91431 (- |m| 1) |ISTRING;replace;$Us2$;15|)
    (LETT |k| |k| |ISTRING;replace;$Us2$;15|) G190
    (COND ((> |i| G91431) (GO G191)))
    (SEQ (EXIT (QESSET |r| |k| (QENUM |s| |i|))))
    (LETT |k|
      (PROG1 (+ |k| 1)
        (LETT |i| (+ |i| 1) |ISTRING;replace;$Us2$;15|)
        |ISTRING;replace;$Us2$;15|)
        (GO G190) G191 (EXIT NIL))
    (EXIT |r|))))))

(DEFUN |ISTRING;setelt;$I2C;16| (|s| |i| |c| $)
  (SEQ (COND
    ((OR (< |i| (QREFELT $ 6))
      (< (SPADCALL |s| (QREFELT $ 42)) |i|))
      (|error| "index out of range"))
    ('T (SEQ (QESSET |s| (- |i| (QREFELT $ 6)) |c|) (EXIT |c|)))))

(DEFUN |ISTRING;substring?;2$IB;17| (|part| |whole| |startpos| $)
  (PROG (|np| |nw| |iw| |ip| G91443 G91442 G91438)
    (RETURN
      (SEQ (EXIT (SEQ (LETT |np| (QCSIZE |part|)
        |ISTRING;substring?;2$IB;17|)
        (LETT |nw| (QCSIZE |whole|)
          |ISTRING;substring?;2$IB;17|)
          (LETT |startpos| (- |startpos| (QREFELT $ 6))
            |ISTRING;substring?;2$IB;17|)
            (EXIT (COND
              ((< |startpos| 0)
                (|error| "index out of bounds"))
              (T (EXIT |startpos|))))
          (EXIT |startpos|))))
        (EXIT |startpos|))))

```

```

((< (- |nw| |startpos|) |np|) 'NIL)
('T
  (SEQ (SEQ
    (EXIT
      (SEQ
        (LETT |iw| |startpos|
          |ISTRING;substring?;2$IB;17|)
        (LETT |ip| 0
          |ISTRING;substring?;2$IB;17|)
        (LETT G91443 (- |np| 1)
          |ISTRING;substring?;2$IB;17|)
        G190
        (COND
          ((QSGREATERP |ip| G91443)
            (GO G191)))
        (SEQ
          (EXIT
            (COND
              ((NULL
                (EQL (QENUM |part| |ip|)
                  (QENUM |whole| |iw|)))
                (PROGN
                  (LETT G91438
                    (PROGN
                      (LETT G91442 'NIL
                        |ISTRING;substring?;2$IB;17|)
                      (GO G91442))
                      |ISTRING;substring?;2$IB;17|)
                      (GO G91438))))))
              (LETT |ip|
                (PROG1 (QSADD1 |ip|)
                  (LETT |iw| (+ |iw| 1)
                    |ISTRING;substring?;2$IB;17|)
                    |ISTRING;substring?;2$IB;17|)
                  (GO G190) G191 (EXIT NIL)))
                G91438 (EXIT G91438))
              (EXIT 'T))))))
          G91442 (EXIT G91442))))))

(DEFUN |ISTRING;position;2$2I;18| (|s| |t| |startpos| $)
  (PROG (|r|)
    (RETURN
      (SEQ (LETT |startpos| (- |startpos| (QREFELT $ 6))
        |ISTRING;position;2$2I;18|)
        (EXIT (COND
          ((< |startpos| 0) (|error| "index out of bounds"))
          ((NULL (< |startpos| (QCSIZE |t|)))
            (- (QREFELT $ 6) 1))
          ('T
            (SEQ (LETT |r| (STRPOS |s| |t| |startpos| NIL)

```

```

                                |ISTRING;position;2$I;18|)
(EXIT (COND
      ((EQ |r| NIL) (- (QREFELT $ 6) 1))
      ('T (+ |r| (QREFELT $ 6)))))))))

(DEFUN |ISTRING;position;C$I;19| (|c| |t| |startpos| $)
  (PROG (|r| G91454 G91453)
    (RETURN
      (SEQ (EXIT (SEQ (LETT |startpos| (- |startpos| (QREFELT $ 6))
                        |ISTRING;position;C$I;19|)
                    (EXIT (COND
                          ((< |startpos| 0)
                           (|error| "index out of bounds"))
                          ((NULL (< |startpos| (QCSIZE |t|)))
                           (- (QREFELT $ 6) 1))
                          ('T
                           (SEQ (SEQ
                                (LETT |r| |startpos|
                                    |ISTRING;position;C$I;19|)
                                (LETT G91454
                                    (QSDIFFERENCE (QCSIZE |t|) 1)
                                    |ISTRING;position;C$I;19|)
                                G190
                                (COND
                                  ((> |r| G91454) (GO G191)))
                                (SEQ
                                  (EXIT
                                    (COND
                                      ((EQL (QENUM |t| |r|) |c|)
                                       (PROGN
                                         (LETT G91453
                                             (+ |r| (QREFELT $ 6))
                                             |ISTRING;position;C$I;19|)
                                         (GO G91453))))))
                                  (LETT |r| (+ |r| 1)
                                    |ISTRING;position;C$I;19|)
                                  (GO G190) G191 (EXIT NIL))
                                  (EXIT (- (QREFELT $ 6) 1))))))
                                G91453 (EXIT G91453))))))

(DEFUN |ISTRING;position;Cc$I;20| (|cc| |t| |startpos| $)
  (PROG (|r| G91461 G91460)
    (RETURN
      (SEQ (EXIT (SEQ (LETT |startpos| (- |startpos| (QREFELT $ 6))
                        |ISTRING;position;Cc$I;20|)
                    (EXIT (COND
                          ((< |startpos| 0)
                           (|error| "index out of bounds"))
                          ((NULL (< |startpos| (QCSIZE |t|)))
                           (- (QREFELT $ 6) 1))

```



```

('T
  (SEQ (SEQ
    (LETT |r| |startpos|
      |ISTRING;position;Cc$2I;20|)
    (LETT G91461
      (QSDIFFERENCE (QCSIZE |t|) 1)
      |ISTRING;position;Cc$2I;20|)
    G190
    (COND
      ((> |r| G91461) (GO G191)))
    (SEQ
      (EXIT
        (COND
          ((SPADCALL (QENUM |t| |r|)
            |cc| (QREFELT $ 49))
            (PROGN
              (LETT G91460
                (+ |r| (QREFELT $ 6))
                |ISTRING;position;Cc$2I;20|)
                (GO G91460))))))
      (LETT |r| (+ |r| 1)
        |ISTRING;position;Cc$2I;20|)
      (GO G190) G191 (EXIT NIL))
    (EXIT (- (QREFELT $ 6) 1))))))
  G91460 (EXIT G91460))))))

(DEFUN |ISTRING;suffix?;2$B;21| (|s| |t| $)
  (PROG (|n| |m|)
    (RETURN
      (SEQ (LETT |n| (SPADCALL |t| (QREFELT $ 42))
        |ISTRING;suffix?;2$B;21|)
        (LETT |m| (SPADCALL |s| (QREFELT $ 42))
          |ISTRING;suffix?;2$B;21|)
        (EXIT (COND
          ((< |n| |m|) 'NIL)
          ('T
            (SPADCALL |s| |t| (- (+ (QREFELT $ 6) |n|) |m|)
              (QREFELT $ 46))))))))))

(DEFUN |ISTRING;split;$CL;22| (|s| |c| $)
  (PROG (|n| |j| |i| |l|)
    (RETURN
      (SEQ (LETT |n| (SPADCALL |s| (QREFELT $ 42))
        |ISTRING;split;$CL;22|)
        (SEQ (LETT |i| (QREFELT $ 6) |ISTRING;split;$CL;22|) G190
          (COND
            ((OR (> |i| |n|)
              (NULL (SPADCALL
                (SPADCALL |s| |i| (QREFELT $ 52)) |c|
                (QREFELT $ 53))))))

```

```

      (GO G191)))
    (SEQ (EXIT 0))
    (LETT |i| (+ |i| 1) |ISTRING;split;$CL;22|) (GO G190)
    G191 (EXIT NIL))
  (LETT |l| (SPADCALL (QREFELT $ 55)) |ISTRING;split;$CL;22|)
  (SEQ G190
    (COND
      ((NULL (COND
        ((< |n| |i|) 'NIL)
        ('T
          (SEQ (LETT |j|
            (SPADCALL |c| |s| |i|
              (QREFELT $ 48))
              |ISTRING;split;$CL;22|)
            (EXIT (COND
              ((< |j| (QREFELT $ 6)) 'NIL)
              ('T 'T)))))))
        (GO G191)))
      (SEQ (LETT |l|
        (SPADCALL
          (SPADCALL |s|
            (SPADCALL |i| (- |j| 1)
              (QREFELT $ 20))
              (QREFELT $ 21))
            |l| (QREFELT $ 56))
            |ISTRING;split;$CL;22|)
          (EXIT (SEQ (LETT |i| |j| |ISTRING;split;$CL;22|)
            G190
            (COND
              ((OR (> |i| |n|)
                (NULL
                  (SPADCALL
                    (SPADCALL |s| |i| (QREFELT $ 52))
                    |c| (QREFELT $ 53))))
                (GO G191)))
              (SEQ (EXIT 0))
              (LETT |i| (+ |i| 1)
                |ISTRING;split;$CL;22|)
              (GO G190) G191 (EXIT NIL))))
          NIL (GO G190) G191 (EXIT NIL))
        (COND
          ((NULL (< |n| |i|))
            (LETT |l|
              (SPADCALL
                (SPADCALL |s| (SPADCALL |i| |n| (QREFELT $ 20))
                  (QREFELT $ 21))
                  |l| (QREFELT $ 56))
                  |ISTRING;split;$CL;22|)))
            (EXIT (SPADCALL |l| (QREFELT $ 57)))))))

```

```

(DEFUN |ISTRING;split;$CcL;23| (|s| |cc| $)
  (PROG (|n| |j| |i| |l|)
    (RETURN
      (SEQ (LETT |n| (SPADCALL |s| (QREFELT $ 42))
        |ISTRING;split;$CcL;23|)
        (SEQ (LETT |i| (QREFELT $ 6) |ISTRING;split;$CcL;23|) G190
          (COND
            ((OR (> |i| |n|)
              (NULL (SPADCALL
                (SPADCALL |s| |i| (QREFELT $ 52)) |cc|
                (QREFELT $ 49))))
              (GO G191)))
          (SEQ (EXIT 0))
          (LETT |i| (+ |i| 1) |ISTRING;split;$CcL;23|) (GO G190)
          G191 (EXIT NIL))
        (LETT |l| (SPADCALL (QREFELT $ 55)) |ISTRING;split;$CcL;23|)
        (SEQ G190
          (COND
            ((NULL (COND
              ((< |n| |i|) 'NIL)
              ('T
                (SEQ (LETT |j|
                  (SPADCALL |cc| |s| |i|
                    (QREFELT $ 50))
                    |ISTRING;split;$CcL;23|)
                  (EXIT (COND
                    ((< |j| (QREFELT $ 6)) 'NIL)
                    ('T 'T)))))))
              (GO G191)))
            (SEQ (LETT |l|
              (SPADCALL
                (SPADCALL |s|
                  (SPADCALL |i| (- |j| 1)
                    (QREFELT $ 20))
                    (QREFELT $ 21))
                  |l| (QREFELT $ 56))
                |ISTRING;split;$CcL;23|)
              (EXIT (SEQ (LETT |i| |j| |ISTRING;split;$CcL;23|)
                G190
                (COND
                  ((OR (> |i| |n|)
                    (NULL
                      (SPADCALL
                        (SPADCALL |s| |i| (QREFELT $ 52))
                        |cc| (QREFELT $ 49))))
                    (GO G191)))
                  (SEQ (EXIT 0))
                  (LETT |i| (+ |i| 1)
                    |ISTRING;split;$CcL;23|)
                    (GO G190) G191 (EXIT NIL))))))
            ))
          ))
        ))
    ))
  )

```

```

        NIL (GO G190) G191 (EXIT NIL))
(COND
  ((NULL (< |n| |i|))
    (LETT |l|
      (SPADCALL
        (SPADCALL |s| (SPADCALL |i| |n| (QREFELT $ 20))
          (QREFELT $ 21))
        |l| (QREFELT $ 56))
      |ISTRING;split;$CcL;23|)))
  (EXIT (SPADCALL |l| (QREFELT $ 57))))))

(DEFUN |ISTRING;leftTrim;$C$;24| (|s| |c| $)
  (PROG (|n| |i|)
    (RETURN
      (SEQ (LETT |n| (SPADCALL |s| (QREFELT $ 42))
        |ISTRING;leftTrim;$C$;24|)
        (SEQ (LETT |i| (QREFELT $ 6) |ISTRING;leftTrim;$C$;24|) G190
          (COND
            ((OR (> |i| |n|)
              (NULL (SPADCALL
                (SPADCALL |s| |i| (QREFELT $ 52)) |c|
                (QREFELT $ 53))))
              (GO G191)))
            (SEQ (EXIT 0))
            (LETT |i| (+ |i| 1) |ISTRING;leftTrim;$C$;24|)
            (GO G190) G191 (EXIT NIL))
          (EXIT (SPADCALL |s| (SPADCALL |i| |n| (QREFELT $ 20))
            (QREFELT $ 21)))))))

(DEFUN |ISTRING;leftTrim;$Cc$;25| (|s| |cc| $)
  (PROG (|n| |i|)
    (RETURN
      (SEQ (LETT |n| (SPADCALL |s| (QREFELT $ 42))
        |ISTRING;leftTrim;$Cc$;25|)
        (SEQ (LETT |i| (QREFELT $ 6) |ISTRING;leftTrim;$Cc$;25|)
          G190
          (COND
            ((OR (> |i| |n|)
              (NULL (SPADCALL
                (SPADCALL |s| |i| (QREFELT $ 52)) |cc|
                (QREFELT $ 49))))
              (GO G191)))
            (SEQ (EXIT 0))
            (LETT |i| (+ |i| 1) |ISTRING;leftTrim;$Cc$;25|)
            (GO G190) G191 (EXIT NIL))
          (EXIT (SPADCALL |s| (SPADCALL |i| |n| (QREFELT $ 20))
            (QREFELT $ 21)))))))

(DEFUN |ISTRING;rightTrim;$C$;26| (|s| |c| $)
  (PROG (|j| G91487)

```

```

(RETURN
  (SEQ (SEQ (LETT |j| (SPADCALL |s| (QREFELT $ 42))
    |ISTRING;rightTrim;$C$;26|)
    (LETT G91487 (QREFELT $ 6)
    |ISTRING;rightTrim;$C$;26|)
    G190
    (COND
      ((OR (< |j| G91487)
        (NULL (SPADCALL
          (SPADCALL |s| |j| (QREFELT $ 52)) |c|
          (QREFELT $ 53))))
        (GO G191)))
    (SEQ (EXIT 0))
    (LETT |j| (+ |j| -1) |ISTRING;rightTrim;$C$;26|)
    (GO G190) G191 (EXIT NIL))
  (EXIT (SPADCALL |s|
    (SPADCALL (SPADCALL |s| (QREFELT $ 28)) |j|
    (QREFELT $ 20))
    (QREFELT $ 21))))))

(DEFUN |ISTRING;rightTrim;$Cc$;27| (|s| |cc| $)
  (PROG (|j| G91491)
    (RETURN
      (SEQ (SEQ (LETT |j| (SPADCALL |s| (QREFELT $ 42))
        |ISTRING;rightTrim;$Cc$;27|)
        (LETT G91491 (QREFELT $ 6)
        |ISTRING;rightTrim;$Cc$;27|)
        G190
        (COND
          ((OR (< |j| G91491)
            (NULL (SPADCALL
              (SPADCALL |s| |j| (QREFELT $ 52)) |cc|
              (QREFELT $ 49))))
            (GO G191)))
          (SEQ (EXIT 0))
          (LETT |j| (+ |j| -1) |ISTRING;rightTrim;$Cc$;27|)
          (GO G190) G191 (EXIT NIL))
        (EXIT (SPADCALL |s|
          (SPADCALL (SPADCALL |s| (QREFELT $ 28)) |j|
          (QREFELT $ 20))
          (QREFELT $ 21))))))

(DEFUN |ISTRING;concat;L$;28| (|l| $)
  (PROG (G91500 G91494 G91492 G91493 |t| |s| G91499 |i|)
    (RETURN
      (SEQ (LETT |t|
        (SPADCALL
          (PROGN
            (LETT G91493 NIL |ISTRING;concat;L$;28|)

```

```

(SEQ (LETT |s| NIL |ISTRING;concat;L$;28|)
      (LETT G91500 |l| |ISTRING;concat;L$;28|)
      G190
      (COND
        ((OR (ATOM G91500)
              (PROGN
                (LETT |s| (CAR G91500)
                      |ISTRING;concat;L$;28|)
                NIL))
         (GO G191)))
      (SEQ (EXIT (PROGN
                  (LETT G91494
                        (SPADCALL |s| (QREFELT $ 13))
                        |ISTRING;concat;L$;28|)
                  (COND
                    (G91493
                     (LETT G91492
                           (+ G91492 G91494)
                           |ISTRING;concat;L$;28|))
                    ('T
                     (PROGN
                      (LETT G91492 G91494
                            |ISTRING;concat;L$;28|)
                      (LETT G91493 'T
                            |ISTRING;concat;L$;28|)))))))
            (LETT G91500 (CDR G91500)
              |ISTRING;concat;L$;28|)
            (GO G190) G191 (EXIT NIL))
      (COND (G91493 G91492) ('T 0)))
      (SPADCALL (QREFELT $ 43)) (QREFELT $ 9))
      |ISTRING;concat;L$;28|)
(LETT |i| (QREFELT $ 6) |ISTRING;concat;L$;28|)
(SEQ (LETT |s| NIL |ISTRING;concat;L$;28|)
      (LETT G91499 |l| |ISTRING;concat;L$;28|) G190
      (COND
        ((OR (ATOM G91499)
              (PROGN
                (LETT |s| (CAR G91499)
                      |ISTRING;concat;L$;28|)
                NIL))
         (GO G191)))
      (SEQ (SPADCALL |t| |s| |i| (QREFELT $ 65))
            (EXIT (LETT |i|
                      (+ |i| (SPADCALL |s| (QREFELT $ 13)))
                      |ISTRING;concat;L$;28|)))
            (LETT G91499 (CDR G91499) |ISTRING;concat;L$;28|)
            (GO G190) G191 (EXIT NIL))
      (EXIT |t|))))))

(DEFUN |ISTRING;copyInto!;2$I$;29| (|y| |x| |s| $)

```

```

(PROG (|m| |n|)
  (RETURN
    (SEQ (LETT |m| (SPADCALL |x| (QREFELT $ 13))
          |ISTRING;copyInto!;2$I$;29|)
      (LETT |n| (SPADCALL |y| (QREFELT $ 13))
            |ISTRING;copyInto!;2$I$;29|)
      (LETT |s| (- |s| (QREFELT $ 6)) |ISTRING;copyInto!;2$I$;29|)
      (COND
        ((OR (< |s| 0) (< |n| (+ |s| |m|))))
        (EXIT (|error| "index out of range"))))
      (RPLACSTR |y| |s| |m| |x| 0 |m|) (EXIT |y|))))))

(DEFUN |ISTRING;elt;$IC;30| (|s| |i| $)
  (COND
    ((OR (< |i| (QREFELT $ 6)) (< (SPADCALL |s| (QREFELT $ 42)) |i|))
      (|error| "index out of range"))
    ('T (QENUM |s| (- |i| (QREFELT $ 6))))))

(DEFUN |ISTRING;elt;$Us$;31| (|s| |sg| $)
  (PROG (|l| |h|)
    (RETURN
      (SEQ (LETT |l| (- (SPADCALL |sg| (QREFELT $ 39)) (QREFELT $ 6))
            |ISTRING;elt;$Us$;31|)
          (LETT |h|
            (COND
              ((SPADCALL |sg| (QREFELT $ 40))
               (- (SPADCALL |sg| (QREFELT $ 41)) (QREFELT $ 6)))
              ('T (- (SPADCALL |s| (QREFELT $ 42)) (QREFELT $ 6))))
            |ISTRING;elt;$Us$;31|)
          (COND
            ((OR (< |l| 0)
              (NULL (< |h| (SPADCALL |s| (QREFELT $ 13)))))
              (EXIT (|error| "index out of bound"))))
            (EXIT (SUBSTRING |s| |l| (MAX 0 (+ (- |h| |l|) 1))))))))))

(DEFUN |ISTRING;hash;$I;32| (|s| $)
  (PROG (|n|)
    (RETURN
      (SEQ (LETT |n| (QCSIZE |s|) |ISTRING;hash;$I;32|)
          (EXIT (COND
            ((ZEROP |n|) 0)
            ((EQL |n| 1)
              (SPADCALL
                (SPADCALL |s| (QREFELT $ 6) (QREFELT $ 52))
                (QREFELT $ 67)))
            ('T
              (* (* (SPADCALL
                    (SPADCALL |s| (QREFELT $ 6)

```

```

(QREFELT $ 52))
(QREFELT $ 67))
(SPADCALL
  (SPADCALL |s| (- (+ (QREFELT $ 6) |n|) 1)
    (QREFELT $ 52))
    (QREFELT $ 67)))
(SPADCALL
  (SPADCALL |s|
    (+ (QREFELT $ 6) (QUOTIENT2 |n| 2))
    (QREFELT $ 52))
    (QREFELT $ 67)))))))))
(PUT ' |ISTRING;match;2$CNni;33| ' |SPADreplace| ' |stringMatch|)

(DEFUN |ISTRING;match;2$CNni;33| (|pattern| |target| |wildcard| $)
  (|stringMatch| |pattern| |target| |wildcard|))

(DEFUN |ISTRING;match?;2$CB;34| (|pattern| |target| |dontcare| $)
  (PROG (|n| |m| G91514 G91516 |s| G91518 G91526 |i| |p|
    G91519 |q|)
    (RETURN
      (SEQ (EXIT (SEQ (LETT |n| (SPADCALL |pattern| (QREFELT $ 42))
        |ISTRING;match?;2$CB;34|)
        (LETT |p|
          (PROG1 (LETT G91514
            (SPADCALL |dontcare| |pattern|
              (LETT |m|
                (SPADCALL |pattern|
                  (QREFELT $ 28))
                  |ISTRING;match?;2$CB;34|)
                  (QREFELT $ 48))
                  |ISTRING;match?;2$CB;34|)
                (|check-subtype| (>= G91514 0)
                  '(|NonNegativeInteger|) G91514))
                |ISTRING;match?;2$CB;34|)
          (EXIT (COND
            ((EQL |p| (- |m| 1))
              (SPADCALL |pattern| |target|
                (QREFELT $ 14)))
            ('T
              (SEQ (COND
                ((NULL (EQL |p| |m|))
                  (COND
                    (NULL
                      (SPADCALL
                        (SPADCALL |pattern|
                          (SPADCALL |m| (- |p| 1)
                            (QREFELT $ 20))
                            (QREFELT $ 21))
                          |target| (QREFELT $ 70)))

```



```

        (EXIT 'NIL))))))
(LETT |i| |p|
 |ISTRING;match?;2$CB;34|)
(LETT |q|
 (PROG1
  (LETT G91516
   (SPADCALL |dontcare| |pattern|
    (+ |p| 1) (QREFELT $ 48))
   |ISTRING;match?;2$CB;34|)
   (|check-subtype| (>= G91516 0)
    '(|NonNegativeInteger|)
    G91516))
 |ISTRING;match?;2$CB;34|)
 (SEQ G190
  (COND
   ((NULL
    (COND
     ((EQL |q| (- |m| 1)) 'NIL)
     ('T 'T))))
   (GO G191)))
 (SEQ
  (LETT |s|
   (SPADCALL |pattern|
    (SPADCALL (+ |p| 1) (- |q| 1)
     (QREFELT $ 20))
    (QREFELT $ 21))
   |ISTRING;match?;2$CB;34|)
  (LETT |i|
   (PROG1
    (LETT G91518
     (SPADCALL |s| |target| |i|
      (QREFELT $ 47))
     |ISTRING;match?;2$CB;34|)
     (|check-subtype|
      (>= G91518 0)
      '(|NonNegativeInteger|)
      G91518))
    |ISTRING;match?;2$CB;34|)
   (EXIT
    (COND
     ((EQL |i| (- |m| 1))
      (PROGN
       (LETT G91526 'NIL
        |ISTRING;match?;2$CB;34|)
       (GO G91526))))
    ('T
     (SEQ
      (LETT |i|
       (+ |i|
        (SPADCALL |s|

```

```

(QREFELT $ 13)))
|ISTRING;match?;2$CB;34|)
(LETT |p| |q|
|ISTRING;match?;2$CB;34|)
(EXIT
(LETT |q|
(PROG1
(LETT G91519
(SPADCALL |dontcare|
|pattern| (+ |q| 1)
(QREFELT $ 48))
|ISTRING;match?;2$CB;34|)
(|check-subtype|
(>= G91519 0)
'(|NonNegativeInteger|
G91519))
|ISTRING;match?;2$CB;34|))))))
NIL (GO G190) G191 (EXIT NIL))
(COND
((NULL (EQL |p| |n|))
(COND
((NULL
(SPADCALL
(SPADCALL |pattern|
(SPADCALL (+ |p| 1) |n|
(QREFELT $ 20))
(QREFELT $ 21))
|target| (QREFELT $ 51)))
(EXIT 'NIL))))))
(EXIT 'T))))))
G91526 (EXIT G91526))))))

(DEFUN |IndexedString| (G91535)
(PROG ()
(RETURN
(PROG (G91536)
(RETURN
(COND
((LETT G91536
(|lassocShiftWithFunction|
(LIST (|devaluate| G91535))
(HGET |$ConstructorCache| ' |IndexedString|)
'|domainEqualList|)
|IndexedString|)
(|CDRwithIncrement| G91536))
('T
(UNWIND-PROTECT
(PROG1 (|IndexedString| G91535)
(LETT G91536 T |IndexedString|))
(COND

```

```

((NOT G91536)
 (HREM |$ConstructorCache| '|IndexedString|'))))))))

(DEFUN |IndexedString;| (#1|)
  (PROG (DV$1 |dv$| $ G91534 G91533 |pv$|)
    (RETURN
      (PROGN
        (LETT DV$1 (|devaluate| |#1|) |IndexedString|)
        (LETT |dv$| (LIST '|IndexedString| DV$1) |IndexedString|)
        (LETT $ (make-array 83) |IndexedString|)
        (QSETREFV $ 0 |dv$|)
        (QSETREFV $ 3
          (LETT |pv$|
            (|buildPredVector| 0 0
              (LIST (|HasCategory| (|Character|)
                '|SetCategory|))
                (|HasCategory| (|Character|)
                '|ConvertibleTo| (|InputForm|)))
              (LETT G91534
                (|HasCategory| (|Character|)
                  '|OrderedSet|))
                |IndexedString|)
              (OR G91534
                (|HasCategory| (|Character|)
                  '|SetCategory|))
                (|HasCategory| (|Integer|) '|OrderedSet|))
              (LETT G91533
                (AND (|HasCategory| (|Character|)
                  '|Evalable| (|Character|)))
                  (|HasCategory| (|Character|)
                  '|SetCategory|))
                |IndexedString|)
              (OR (AND (|HasCategory| (|Character|)
                '|Evalable| (|Character|)))
                G91534)
                G91533)))
                |IndexedString|))
          (|haddProp| |$ConstructorCache| '|IndexedString| (LIST DV$1)
            (CONS 1 $))
          (|stuffDomainSlots| $)
          (QSETREFV $ 6 |#1|)
          $))))

(setf (get '|IndexedString| '|infovec|)
  (LIST '#(NIL NIL NIL NIL NIL NIL (|local| |#1|)
    (|NonNegativeInteger|) (|Character|) |ISTRING;new;NniC$;1|
    |ISTRING;empty;$;2| (|Boolean|) |ISTRING;empty?;$B;3|
    |ISTRING;#;$Nni;4| |ISTRING;=;2$B;5| |ISTRING;<;2$B;6|
    |ISTRING;concat;3$;7| |ISTRING;copy;2$;8| (|Integer|)
    (|UniversalSegment| 18) (0 . SEGMENT)

```

```

|ISTRING;elt;$Us$;31| (6 . SEGMENT)
|ISTRING;insert;2$I$;9| (|String|) (|OutputForm|)
(11 . |outputForm|) |ISTRING;coerce;$Of;10|
|ISTRING;minIndex;$I;11| (|CharacterClass|)
(16 . |upperCase|) (20 . |upperCase|) (|Mapping| 8 8)
(25 . |map!|) |ISTRING;upperCase!;2$;12|
(31 . |lowerCase|) (35 . |lowerCase|)
|ISTRING;lowerCase!;2$;13| |ISTRING;latex;$S;14|
(40 . |lo|) (45 . |hasHi|) (50 . |hi|) (55 . |maxIndex|)
(60 . |space|) |ISTRING;replace;$Us2$;15|
|ISTRING;setelt;$I2C;16| |ISTRING;substring?;2$IB;17|
|ISTRING;position;2$2I;18| |ISTRING;position;C$2I;19|
(64 . |member?|) |ISTRING;position;Cc$2I;20|
|ISTRING;suffix?;2$B;21| |ISTRING;elt;$IC;30| (70 . =)
(|List| $$) (76 . |empty|) (80 . |concat|)
(86 . |reverse!|) (|List| $) |ISTRING;split;$CL;22|
|ISTRING;split;$CcL;23| |ISTRING;leftTrim;$C$;24| | |
|ISTRING;leftTrim;$Cc$;25| |ISTRING;rightTrim;$C$;26|
|ISTRING;rightTrim;$Cc$;27| |ISTRING;copyInto!;2$I$;29|
|ISTRING;concat;L$;28| (91 . |ord|) |ISTRING;hash;$I;32|
|ISTRING;match;2$CnI;33| (96 . |prefix?|)
|ISTRING;match?;2$CB;34| (|List| 8) (|List| 74)
(|Equation| 8) (|Mapping| 8 8 8) (|InputForm|)
(|SingleInteger|) (|Mapping| 11 8) (|Mapping| 11 8 8)
(|Void|) (|Union| 8 "failed") (|List| 18))
'#(~= 102 |upperCase!| 108 |upperCase| 113 |trim| 118 |swap!|
130 |suffix?| 137 |substring?| 143 |split| 150 |sorted?|
162 |sort!| 173 |sort| 184 |size?| 195 |setelt| 201
|select| 215 |sample| 221 |rightTrim| 225 |reverse!| 237
|reverse| 242 |replace| 247 |removeDuplicates| 254
|remove| 259 |reduce| 271 |qsetelt!| 292 |qelt| 299
|prefix?| 305 |position| 311 |parts| 344 |new| 349 |more?|
355 |minIndex| 361 |min| 366 |merge| 372 |members| 385
|member?| 390 |maxIndex| 396 |max| 401 |match?| 407
|match| 414 |map!| 421 |map| 427 |lowerCase!| 440
|lowerCase| 445 |less?| 450 |leftTrim| 456 |latex| 468
|insert| 473 |indices| 487 |index?| 492 |hash| 498 |first|
508 |find| 513 |fill!| 519 |every?| 525 |eval| 531 |eq?|
557 |entry?| 563 |entries| 569 |empty?| 574 |empty| 579
|elt| 583 |delete| 608 |count| 620 |copyInto!| 632 |copy|
639 |convert| 644 |construct| 649 |concat| 654 |coerce|
677 |any?| 687 >= 693 > 699 = 705 <= 711 < 717 |#| 723)
'((|shallowlyMutable| . 0) (|finiteAggregate| . 0))
(CONS (|makeByteWordVec2| 7
      '(0 0 0 0 0 0 3 0 0 7 4 0 0 7 1 2 4))
  (CONS '#(|StringAggregate&|
           |OneDimensionalArrayAggregate&|
           |FiniteLinearAggregate&| |LinearAggregate&|
           |IndexedAggregate&| |Collection&|
           |HomogeneousAggregate&| |OrderedSet&|

```

```

|Aggregate&| |EltableAggregate&| |Evaluable&|
|SetCategory&| NIL NIL |InnerEvaluable&| NIL
NIL |BasicType&|)
(CONS '#(|StringAggregate|)
      (|OneDimensionalArrayAggregate| 8)
      (|FiniteLinearAggregate| 8)
      (|LinearAggregate| 8)
      (|IndexedAggregate| 18 8)
      (|Collection| 8)
      (|HomogeneousAggregate| 8)
      (|OrderedSet|) (|Aggregate|)
      (|EltableAggregate| 18 8) (|Evaluable| 8)
      (|SetCategory|) (|Type|)
      (|Eltable| 18 8) (|InnerEvaluable| 8 8)
      (|CoercibleTo| 25) (|ConvertibleTo| 76)
      (|BasicType|))
(|makeByteWordVec2| 82
  '(2 19 0 18 18 20 1 19 0 18 22 1 25 0
    24 26 0 29 0 30 1 8 0 0 31 2 0 0 32 0
    33 0 29 0 35 1 8 0 0 36 1 19 18 0 39
    1 19 11 0 40 1 19 18 0 41 1 0 18 0 42
    0 8 0 43 2 29 11 8 0 49 2 8 11 0 0 53
    0 54 0 55 2 54 0 2 0 56 1 54 0 0 57 1
    8 18 0 67 2 0 11 0 0 70 2 1 11 0 0 1
    1 0 0 0 34 1 0 0 0 1 2 0 0 0 8 1 2 0
    0 0 29 1 3 0 80 0 18 18 1 2 0 11 0 0
    51 3 0 11 0 0 18 46 2 0 58 0 29 60 2
    0 58 0 8 59 1 3 11 0 1 2 0 11 79 0 1
    1 3 0 0 1 2 0 0 79 0 1 1 3 0 0 1 2 0
    0 79 0 1 2 0 11 0 7 1 3 0 8 0 19 8 1
    3 0 8 0 18 8 45 2 0 0 78 0 1 0 0 0 1
    2 0 0 0 8 63 2 0 0 0 29 64 1 0 0 0 1
    1 0 0 0 1 3 0 0 0 19 0 44 1 1 0 0 1 2
    1 0 8 0 1 2 0 0 78 0 1 4 1 8 75 0 8 8
    1 3 0 8 75 0 8 1 2 0 8 75 0 1 3 0 8 0
    18 8 1 2 0 8 0 18 1 2 0 11 0 0 70 3 1
    18 8 0 18 48 2 1 18 8 0 1 3 0 18 29 0
    18 50 3 0 18 0 0 18 47 2 0 18 78 0 1
    1 0 72 0 1 2 0 0 7 8 9 2 0 11 0 7 1 1
    5 18 0 28 2 3 0 0 0 1 2 3 0 0 0 1 3 0
    0 79 0 0 1 1 0 72 0 1 2 1 11 8 0 1 1
    5 18 0 42 2 3 0 0 0 1 3 0 11 0 0 8 71
    3 0 7 0 0 8 69 2 0 0 32 0 33 3 0 0 75
    0 0 1 2 0 0 32 0 1 1 0 0 0 37 1 0 0 0
    1 2 0 11 0 7 1 2 0 0 0 8 61 2 0 0 0
    29 62 1 1 24 0 38 3 0 0 8 0 18 1 3 0
    0 0 0 18 23 1 0 82 0 1 2 0 11 18 0 1
    1 1 77 0 1 1 0 18 0 68 1 5 8 0 1 2 0
    81 78 0 1 2 0 0 0 8 1 2 0 11 78 0 1 3
    6 0 0 72 72 1 3 6 0 0 8 8 1 2 6 0 0

```

```

73 1 2 6 0 0 74 1 2 0 11 0 0 1 2 1 11
8 0 1 1 0 72 0 1 1 0 11 0 12 0 0 0 10
2 0 0 0 0 1 2 0 0 0 19 21 2 0 8 0 18
52 3 0 8 0 18 8 1 2 0 0 0 18 1 2 0 0
0 19 1 2 1 7 8 0 1 2 0 7 78 0 1 3 0 0
0 0 18 65 1 0 0 0 17 1 2 76 0 1 1 0 0
72 1 1 0 0 58 66 2 0 0 0 0 16 2 0 0 0
8 1 2 0 0 8 0 1 1 1 25 0 27 1 0 0 8 1
2 0 11 78 0 1 2 3 11 0 0 1 2 3 11 0 0
1 2 1 11 0 0 14 2 3 11 0 0 1 2 3 11 0
0 15 1 0 7 0 13))))))

' |lookupComplete|))

```

LIST.lsp BOOTSTRAP

LIST depends on a chain of files. We need to break this cycle to build the algebra. So we keep a cached copy of the translated **LIST** category which we can write into the **MID** directory. We compile the lisp code and copy the **LIST.o** file to the **OUT** directory. This is eventually forcibly replaced by a recompiled version.

Note that this code is not included in the generated catdef.spad file.

— LIST.lsp BOOTSTRAP —

```

(|/VERSIONCHECK| 2)

(PUT (QUOTE |LIST;nil;$;1|) (QUOTE |SPADreplace|) (QUOTE (XLAM NIL NIL)))

(DEFUN |LIST;nil;$;1| (|$|) NIL)

(PUT (QUOTE |LIST>null;$B;2|) (QUOTE |SPADreplace|) (QUOTE NULL))

(DEFUN |LIST>null;$B;2| (|l| |$|) (NULL |l|))

(PUT (QUOTE |LIST;cons;S2$;3|) (QUOTE |SPADreplace|) (QUOTE CONS))

(DEFUN |LIST;cons;S2$;3| (|s| |l| |$|) (CONS |s| |l|))

(PUT (QUOTE |LIST;append;3$;4|) (QUOTE |SPADreplace|) (QUOTE APPEND))

(DEFUN |LIST;append;3$;4| (|l| |t| |$|) (APPEND |l| |t|))

(DEFUN |LIST;writeOMList| (|dev| |x| |$|)
  (SEQ
    (SPADCALL |dev| (QREFELT |$| 14))

```

```

(SPADCALL |dev| "list1" "list" (QREFELT |$| 16))
(SEQ
  G190
  (COND
    ((NULL (COND ((NULL |x|) (QUOTE NIL)) ((QUOTE T) (QUOTE T)))) (GO G191)))
  (SEQ
    (SPADCALL |dev| (|SPADfirst| |x|) (QUOTE NIL) (QREFELT |$| 17))
    (EXIT (LETT |x| (CDR |x|) |LIST;writeOMList|)))
  NIL
  (GO G190)
  G191
  (EXIT NIL))
(EXIT (SPADCALL |dev| (QREFELT |$| 18)))))

(DEFUN |LIST;OMwrite;$S;6| (|x| |$|)
  (PROG (|sp| |dev| |s|)
    (RETURN
      (SEQ
        (LETT |s| "" |LIST;OMwrite;$S;6|)
        (LETT |sp| (|OM-STRINGTOSTRINGPTR| |s|) |LIST;OMwrite;$S;6|)
        (LETT |dev|
          (SPADCALL |sp| (SPADCALL (QREFELT |$| 20)) (QREFELT |$| 21))
          |LIST;OMwrite;$S;6|)
        (SPADCALL |dev| (QREFELT |$| 22))
        (|LIST;writeOMList| |dev| |x| |$|)
        (SPADCALL |dev| (QREFELT |$| 23))
        (SPADCALL |dev| (QREFELT |$| 24))
        (LETT |s| (|OM-STRINGPTRTOSTRING| |sp|) |LIST;OMwrite;$S;6|)
        (EXIT |s|))))))

(DEFUN |LIST;OMwrite;$BS;7| (|x| |wholeObj| |$|)
  (PROG (|sp| |dev| |s|)
    (RETURN
      (SEQ
        (LETT |s| "" |LIST;OMwrite;$BS;7|)
        (LETT |sp| (|OM-STRINGTOSTRINGPTR| |s|) |LIST;OMwrite;$BS;7|)
        (LETT |dev|
          (SPADCALL |sp| (SPADCALL (QREFELT |$| 20)) (QREFELT |$| 21))
          |LIST;OMwrite;$BS;7|)
        (COND (|wholeObj| (SPADCALL |dev| (QREFELT |$| 22))))
        (|LIST;writeOMList| |dev| |x| |$|)
        (COND (|wholeObj| (SPADCALL |dev| (QREFELT |$| 23))))
        (SPADCALL |dev| (QREFELT |$| 24))
        (LETT |s| (|OM-STRINGPTRTOSTRING| |sp|) |LIST;OMwrite;$BS;7|)
        (EXIT |s|))))))

(DEFUN |LIST;OMwrite;Omd$V;8| (|dev| |x| |$|)
  (SEQ
    (SPADCALL |dev| (QREFELT |$| 22))
    (|LIST;writeOMList| |dev| |x| |$|)

```

```

(EXIT (SPADCALL |dev| (QREFELT |$| 23))))))

(DEFUN |LIST;OMwrite;Omd$BV;9| (|dev| |x| |wholeObj| |$|)
  (SEQ
    (COND (|wholeObj| (SPADCALL |dev| (QREFELT |$| 22))))
    (|LIST;writeOMList| |dev| |x| |$|)
    (EXIT (COND (|wholeObj| (SPADCALL |dev| (QREFELT |$| 23)))))))

(DEFUN |LIST;setUnion;3$;10| (|l1| |l2| |$|)
  (SPADCALL (SPADCALL |l1| |l2| (QREFELT |$| 29)) (QREFELT |$| 30)))

(DEFUN |LIST;setIntersection;3$;11| (|l1| |l2| |$|)
  (PROG (|u|)
    (RETURN
      (SEQ
        (LETT |u| NIL |LIST;setIntersection;3$;11|)
        (LETT |l1| (SPADCALL |l1| (QREFELT |$| 30)) |LIST;setIntersection;3$;11|)
        (SEQ
          G190
          (COND
            ((NULL (COND ((NULL |l1|) (QUOTE NIL)) ((QUOTE T) (QUOTE T))))
            (GO G191)))
          (SEQ
            (COND
              ((SPADCALL (|SPADfirst| |l1|) |l2| (QREFELT |$| 32))
                (LETT |u| (CONS (|SPADfirst| |l1|) |u|) |LIST;setIntersection;3$;11|)))
              (EXIT (LETT |l1| (CDR |l1|) |LIST;setIntersection;3$;11|)))
            NIL
            (GO G190)
            G191
            (EXIT NIL))
            (EXIT |u|))))))

(DEFUN |LIST;setDifference;3$;12| (|l1| |l2| |$|)
  (PROG (|l11| |lu|)
    (RETURN
      (SEQ
        (LETT |l1| (SPADCALL |l1| (QREFELT |$| 30)) |LIST;setDifference;3$;12|)
        (LETT |lu| NIL |LIST;setDifference;3$;12|)
        (SEQ
          G190
          (COND
            ((NULL (COND ((NULL |l1|) (QUOTE NIL)) ((QUOTE T) (QUOTE T))))
            (GO G191)))
          (SEQ
            (LETT |l11|
              (SPADCALL |l1| 1 (QREFELT |$| 35))
              |LIST;setDifference;3$;12|)
            (COND
              ((NULL (SPADCALL |l11| |l2| (QREFELT |$| 32))))

```



```

        (LETT |lu| (CONS |l11| |lu|) |LIST;setDifference;3$;12|)))
    (EXIT (LETT |l1| (CDR |l1|) |LIST;setDifference;3$;12|)))
    NIL
    (GO G190)
    G191
    (EXIT NIL))
    (EXIT |lu|))))))

(DEFUN |LIST;convert;$If;13| (|x| $)
  (PROG (G102544 |a| G102545)
    (RETURN
      (SEQ (SPADCALL
        (CONS (SPADCALL (SPADCALL "construct" (QREFELT $ 38))
          (QREFELT $ 40))
        (PROGN
          (LETT G102544 NIL |LIST;convert;$If;13|)
          (SEQ (LETT |a| NIL |LIST;convert;$If;13|)
            (LETT G102545 |x| |LIST;convert;$If;13|)
            G190
            (COND
              ((OR (ATOM G102545)
                (PROGN
                  (LETT |a| (CAR G102545)
                    |LIST;convert;$If;13|)
                    NIL))
                (GO G191))))
            (SEQ (EXIT (LETT G102544
              (CONS
                (SPADCALL |a| (QREFELT $ 41))
                G102544)
              |LIST;convert;$If;13|))))
            (LETT G102545 (CDR G102545)
              |LIST;convert;$If;13|)
            (GO G190) G191 (EXIT (NREVERSEO G102544))))))
          (QREFELT $ 43))))))

(DEFUN |List| (G102555)
  (PROG ()
    (RETURN
      (PROG (G102556)
        (RETURN
          (COND
            ((LETT G102556
              (|lassocShiftWithFunction|
                (LIST (|devaluate| G102555))
                (HGET |$ConstructorCache| ' |List|)
                '|domainEqualList|)
              |List|)
              (|CDRwithIncrement| G102556))
            ('T

```

```

      (UNWIND-PROTECT
        (PROG1 (|List;| G102555) (LETT G102556 T |List|))
        (COND
          ((NOT G102556) (HREM |$ConstructorCache| '|List|)))))))))

(DEFUN |List;| (|#1|)
  (PROG (DV$1 |dv$| $ G102554 |pv$|)
    (RETURN
      (PROGN
        (LETT DV$1 (|devaluate| |#1|) |List|)
        (LETT |dv$| (LIST '|List| DV$1) |List|)
        (LETT $ (make-array 62) |List|)
        (QSETREFV $ 0 |dv$|)
        (QSETREFV $ 3
          (LETT |pv$|
            (|buildPredVector| 0 0
              (LIST (|HasCategory| |#1| '|SetCategory|))
                (|HasCategory| |#1|
                  '|ConvertibleTo| (|InputForm|)))
              (LETT G102554
                (|HasCategory| |#1| '|OrderedSet|))
                |List|)
              (OR G102554
                (|HasCategory| |#1| '|SetCategory|))
                (|HasCategory| |#1| '|OpenMath|))
                (|HasCategory| (|Integer|) '|OrderedSet|))
                (AND (|HasCategory| |#1|
                  (LIST '|Evalable|
                    (|devaluate| |#1|)))
                    (|HasCategory| |#1| '|SetCategory|)))
                (OR (AND (|HasCategory| |#1|
                  (LIST '|Evalable|
                    (|devaluate| |#1|)))
                    G102554)
                  (AND (|HasCategory| |#1|
                    (LIST '|Evalable|
                      (|devaluate| |#1|)))
                      (|HasCategory| |#1|
                        '|SetCategory|))))))
                |List|))
              (|haddProp| |$ConstructorCache| '|List| (LIST DV$1) (CONS 1 $))
              (|stuffDomainSlots| $)
              (QSETREFV $ 6 |#1|)
              (COND
                ((|testBitVector| |pv$| 5)
                  (PROGN
                    (QSETREFV $ 25
                      (CONS (|dispatchFunction| |LIST;OMwrite;$S;6|) $))
                    (QSETREFV $ 26
                      (CONS (|dispatchFunction| |LIST;OMwrite;$BS;7|) $))

```

```

(QSETREFV $ 27
  (CONS (|dispatchFunction| |LIST;OMwrite;Omd$V;8|) $))
(QSETREFV $ 28
  (CONS (|dispatchFunction| |LIST;OMwrite;Omd$BV;9|) $))))))
(COND
  ((|testBitVector| |pv$| 1)
  (PROGN
    (QSETREFV $ 31
      (CONS (|dispatchFunction| |LIST;setUnion;3$;10|) $))
    (QSETREFV $ 33
      (CONS (|dispatchFunction|
              |LIST;setIntersection;3$;11|)
            $))
    (QSETREFV $ 36
      (CONS (|dispatchFunction| |LIST;setDifference;3$;12|)
            $))))))
(COND
  ((|testBitVector| |pv$| 2)
  (QSETREFV $ 44
    (CONS (|dispatchFunction| |LIST;convert;$If;13|) $))))
$))))

(setf (get
  (QUOTE |List|)
  (QUOTE |infovec|))
  (LIST
    (QUOTE #(
      NIL NIL NIL NIL NIL (|IndexedList| 6 (NRTEVAL 1)) (|local| |#1|)
      |LIST;nil;$;1| (|Boolean|) |LIST>null;$B;2| |LIST;cons;$2$;3|
      |LIST;append;3$;4| (|Void|) (|OpenMathDevice|) (0 . |OMputApp|)
      (|String|) (5 . |OMputSymbol|) (12 . |OMwrite|) (19 . |OMputEndApp|)
      (|OpenMathEncoding|) (24 . |OMencodingXML|) (28 . |OMopenString|)
      (34 . |OMputObject|) (39 . |OMputEndObject|) (44 . |OMclose|)
      (49 . |OMwrite|) (54 . |OMwrite|) (60 . |OMwrite|) (66 . |OMwrite|)
      (73 . |concat|) (79 . |removeDuplicates|) (84 . |setUnion|)
      (90 . |member?|) (96 . |setIntersection|) (|Integer|) (102 . |elt|)
      (108 . |setDifference|) (|Symbol|) (114 . |coerce|) (|InputForm|)
      (119 . |convert|) (124 . |convert|) (|List| |$|) (129 . |convert|)
      (134 . |convert|) (|Mapping| 6 6 6) (|NonNegativeInteger|)
      (|List| 6) (|List| 49) (|Equation| 6) (|Mapping| 8 6)
      (|Mapping| 8 6 6) (|UniversalSegment| 34) (QUOTE "last")
      (QUOTE "rest") (QUOTE "first") (QUOTE "value") (|Mapping| 6 6)
      (|SingleInteger|) (|OutputForm|) (|List| 34) (|Union| 6 (QUOTE "failed"))))
    (QUOTE #(|setUnion| 139 |setIntersection| 145 |setDifference| 151
      |removeDuplicates| 157 |null| 162 |nil| 167 |member?| 171 |elt| 177
      |convert| 183 |cons| 188 |concat| 194 |append| 200 |OMwrite| 206))
    (QUOTE ((|shallowlyMutable| . 0) (|finiteAggregate| . 0)))
    (CONS
      (|makeByteWordVec2| 8 (QUOTE (0 0 0 0 0 0 0 0 3 0 0 8 4 0 0 8 1 2 4 5)))
      (CONS (QUOTE #(

```

```

|ListAggregate&| |StreamAggregate&| |ExtensibleLinearAggregate&| | |
|FiniteLinearAggregate&| |UnaryRecursiveAggregate&| |LinearAggregate&|
|RecursiveAggregate&| |IndexedAggregate&| |Collection&|
|HomogeneousAggregate&| |OrderedSet&| |Aggregate&| |EltableAggregate&|
|Evalable&| |SetCategory&| NIL NIL |InnerEvalable&| NIL NIL
|BasicType&| NIL))
(CONS
  (QUOTE #((|ListAggregate| 6) (|StreamAggregate| 6)
    (|ExtensibleLinearAggregate| 6) (|FiniteLinearAggregate| 6)
    (|UnaryRecursiveAggregate| 6) (|LinearAggregate| 6)
    (|RecursiveAggregate| 6) (|IndexedAggregate| 34 6) (|Collection| 6)
    (|HomogeneousAggregate| 6) (|OrderedSet|) (|Aggregate|)
    (|EltableAggregate| 34 6) (|Evalable| 6) (|SetCategory|)
    (|Type|) (|Eltable| 34 6) (|InnerEvalable| 6 6) (|CoercibleTo| 59)
    (|ConvertibleTo| 39) (|BasicType|) (|OpenMath|)))
  (|makeByteWordVec2| 44
    (QUOTE (
      1 13 12 0 14 3 13 12 0 15 15 16 3 6 12 13 0 8 17 1 13 12 0 18 0
      19 0 20 2 13 0 15 19 21 1 13 12 0 22 1 13 12 0 23 1 13 12 0 24 1 0 15
      0 25 2 0 15 0 8 26 2 0 12 13 0 27 3 0 12 13 0 8 28 2 0 0 0 0 29 1 0 0
      0 30 2 0 0 0 0 31 2 0 8 6 0 32 2 0 0 0 0 33 2 0 6 0 34 35 2 0 0 0 0 36
      1 37 0 15 38 1 39 0 37 40 1 6 39 0 41 1 39 0 42 43 1 0 39 0 44 2 1 0 0
      0 31 2 1 0 0 0 33 2 1 0 0 0 36 1 1 0 0 30 1 0 8 0 9 0 0 0 7 2 1 8 6 0
      32 2 0 6 0 34 35 1 2 39 0 44 2 0 0 6 0 10 2 0 0 0 0 29 2 0 0 0 0 11 3
      5 12 13 0 8 28 2 5 12 13 0 27 1 5 15 0 25 2 5 15 0 8 26))))))
  (QUOTE |lookupIncomplete|)))

```

NNI.lsp BOOTSTRAP

NNI depends on itself. We need to break this cycle to build the algebra. So we keep a cached copy of the translated **NNI** category which we can write into the **MID** directory. We compile the lisp code and copy the **NNI.o** file to the **OUT** directory. This is eventually forcibly replaced by a recompiled version.

Note that this code is not included in the generated catdef.spad file.

— NNI.lsp BOOTSTRAP —

```

(|/VERSIONCHECK| 2)

(SETQ |$CategoryFrame|
  (|put| '|NonNegativeInteger| '|SuperDomain| '|Integer|)
  (|put| '|Integer| '|SubDomain|
    (CONS '(|NonNegativeInteger| COND ((<|#1| 0) 'NIL)
      ('T 'T))
    (DELASC '|NonNegativeInteger|

```

```

(|get| '(|Integer|) '|SubDomain|
|CategoryFrame|))
|CategoryFrame|)))

(PUT '|NNI;sup;3$;1| '|SPADreplace| 'MAX)

(DEFUN |NNI;sup;3$;1| (|x| |y| |$|) (MAX |x| |y|))

(PUT '|NNI;shift;$I$;2| '|SPADreplace| 'ASH)

(DEFUN |NNI;shift;$I$;2| (|x| |n| |$|) (ASH |x| |n|))

(DEFUN |NNI;subtractIfCan;2$U;3| (|x| |y| |$|)
  (PROG (|c|)
    (RETURN
      (SEQ
        (LETT |c| (|-| |x| |y|) |NNI;subtractIfCan;2$U;3|)
        (EXIT
          (COND
            ((|<| |c| 0) (CONS 1 "failed"))
            ((QUOTE T) (CONS 0 |c|))))))))))

(DEFUN |NonNegativeInteger| ()
  (PROG ()
    (RETURN
      (PROG (G96708)
        (RETURN
          (COND
            ((LETT G96708
              (HGET |$ConstructorCache| '|NonNegativeInteger|
|NonNegativeInteger|)
(|CDRwithIncrement| (CDAR G96708)))
('T
  (UNWIND-PROTECT
    (PROG1 (CDDAR (HPUT |$ConstructorCache|
'|NonNegativeInteger|
  (LIST
    (CONS NIL
      (CONS 1 (|NonNegativeInteger;|))))))
    (LETT G96708 T |NonNegativeInteger|))
    (COND
      ((NOT G96708)
        (HREM |$ConstructorCache| '|NonNegativeInteger|))))))))))

(DEFUN |NonNegativeInteger;| ()
  (PROG (|dv$| $ |pv$|)
    (RETURN
      (PROGN
        (LETT |dv$| '(|NonNegativeInteger|) |NonNegativeInteger|)
        (LETT $ (make-array 17) |NonNegativeInteger|)

```

```

(QSETREFV $ 0 |dv$|)
(QSETREFV $ 3
  (LETT |pv$| (|buildPredVector| 0 0 NIL)
    |NonNegativeInteger|))
(|haddProp| |$ConstructorCache| '|NonNegativeInteger| NIL
  (CONS 1 $))
(|stuffDomainSlots| $)
$)))

(setf (get
  (QUOTE |NonNegativeInteger|)
  (QUOTE |infovec|))
  (LIST
    (QUOTE
      #(NIL NIL NIL NIL NIL
        (|Integer|)
        |NNI;sup;3$;1|
        |NNI;shift;$I$;2|
        (|Union| |$| (QUOTE "failed"))
        |NNI;subtractIfCan;2$U;3|
        (|Record| (|:| |quotient| |$|) (|:| |remainder| |$|))
        (|PositiveInteger|)
        (|Boolean|)
        (|NonNegativeInteger|)
        (|SingleInteger|)
        (|String|)
        (|OutputForm|)))
    (QUOTE
      #(|~|=| 0 |zero?| 6 |sup| 11 |subtractIfCan| 17 |shift| 23 |sample| 29
        |rem| 33 |recip| 39 |random| 44 |quo| 49 |one?| 55 |min| 60 |max| 66
        |latex| 72 |hash| 77 |gcd| 82 |exquo| 88 |divide| 94 |coerce| 100
        |^| 105 |Zero| 117 |One| 121 |>=| 125 |>| 131 |=| 137 |<=| 143
        |<| 149 |+| 155 |**| 161 |*| 173))
    (QUOTE (((|commutative| "*" . 0)))
    (CONS
      (|makeByteWordVec2| 1 (QUOTE (0 0 0 0 0 0 0 0 0 0 0 0)))
      (CONS
        (QUOTE
          #(NIL NIL NIL NIL NIL
            |Monoid&|
            |AbelianMonoid&|
            |OrderedSet&|
            |SemiGroup&|
            |AbelianSemiGroup&|
            |SetCategory&|
            |BasicType&|
            NIL))
        (CONS
          (QUOTE
            #((|OrderedAbelianMonoidSup|)

```

```

(|OrderedCancellationAbelianMonoid|)
(|OrderedAbelianMonoid|)
(|OrderedAbelianSemiGroup|)
(|CancellationAbelianMonoid|)
(|Monoid|)
(|AbelianMonoid|)
(|OrderedSet|)
(|SemiGroup|)
(|AbelianSemiGroup|)
(|SetCategory|)
(|BasicType|)
(|CoercibleTo| 16)))
(|makeByteWordVec2| 16
(QUOTE
  (2 0 12 0 0 1 1 0 12 0 1 2 0 0 0 0 6 2 0 8 0 0 9 2 0 0 0 5 7 0 0
   0 1 2 0 0 0 0 1 1 0 8 0 1 1 0 0 0 1 2 0 0 0 0 1 1 0 12 0 1 2 0
   0 0 0 1 2 0 0 0 0 1 1 0 15 0 1 1 0 14 0 1 2 0 0 0 0 1 2 0 8 0 0
   1 2 0 10 0 0 1 1 0 16 0 1 2 0 0 0 11 1 2 0 0 0 13 1 0 0 0 1 0 0
   0 1 2 0 12 0 0 1 2 0 12 0 0 1 2 0 12 0 0 1 2 0 12 0 0 1 2 0 12
   0 0 1 2 0 0 0 0 1 2 0 0 0 11 1 2 0 0 0 13 1 2 0 0 0 0 1 2 0 0
   11 0 1 2 0 0 13 0 1))))))
(QUOTE |lookupComplete|)))

(setf (get (QUOTE |NonNegativeInteger|) (QUOTE NILADIC)) T)

```

OUTFORM.lsp BOOTSTRAP

OUTFORM depends on itself. We need to break this cycle to build the algebra. So we keep a cached copy of the translated **OUTFORM** category which we can write into the **MID** directory. We compile the lisp code and copy the **OUTFORM.o** file to the **OUT** directory. This is eventually forcibly replaced by a recompiled version.

Note that this code is not included in the generated catdef.spad file.

— OUTFORM.lsp BOOTSTRAP —

```

(|/VERSIONCHECK| 2)

(PUT (QUOTE |OUTFORM;print;$V;1|) (QUOTE |SPADreplace|) (QUOTE |mathprint|))

(DEFUN |OUTFORM;print;$V;1| (|x| |$|) (|mathprint| |x|))

(DEFUN |OUTFORM;message;$S;2| (|s| |$|)
  (COND
    ((SPADCALL |s| (QREFELT |$| 11)) (SPADCALL (QREFELT |$| 12)))
  ))

```

```

((QUOTE T) |s|)))

(DEFUN |OUTFORM;messagePrint;SV;3| (|s| |$|)
  (SPADCALL (SPADCALL |s| (QREFELT |$| 13)) (QREFELT |$| 8)))

(PUT (QUOTE |OUTFORM;=;2$B;4|) (QUOTE |SPADreplace|) (QUOTE EQUAL))

(DEFUN |OUTFORM;=;2$B;4| (|a| |b| |$|) (EQUAL |a| |b|))

(DEFUN |OUTFORM;=;3$;5| (|a| |b| |$|)
  (LIST (|OUTFORM;sform| "=" |$|) |a| |b|))

(PUT
  (QUOTE |OUTFORM;coerce;2$;6|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|a|) |a|)))

(DEFUN |OUTFORM;coerce;2$;6| (|a| |$|) |a|)

(PUT
  (QUOTE |OUTFORM;outputForm;I$;7|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|n|) |n|)))

(DEFUN |OUTFORM;outputForm;I$;7| (|n| |$|) |n|)

(PUT
  (QUOTE |OUTFORM;outputForm;S$;8|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|e|) |e|)))

(DEFUN |OUTFORM;outputForm;S$;8| (|e| |$|) |e|)

(PUT
  (QUOTE |OUTFORM;outputForm;Df$;9|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|f|) |f|)))

(DEFUN |OUTFORM;outputForm;Df$;9| (|f| |$|) |f|)

(PUT (QUOTE |OUTFORM;sform|) (QUOTE |SPADreplace|) (QUOTE (XLAM (|s|) |s|)))

(DEFUN |OUTFORM;sform| (|s| |$|) |s|)

(PUT (QUOTE |OUTFORM;eform|) (QUOTE |SPADreplace|) (QUOTE (XLAM (|e|) |e|)))

(DEFUN |OUTFORM;eform| (|e| |$|) |e|)

(PUT (QUOTE |OUTFORM;ifform|) (QUOTE |SPADreplace|) (QUOTE (XLAM (|n|) |n|)))

```



```

(DEFUN |OUTFORM;iform| (|n| |$|) |n|)

(DEFUN |OUTFORM;outputForm;$;13| (|s| |$|)
  (|OUTFORM;sform|
    (SPADCALL
      (SPADCALL (QREFELT |$| 26))
      (SPADCALL |s| (SPADCALL (QREFELT |$| 26)) (QREFELT |$| 27))
      (QREFELT |$| 28))
    |$|))

(PUT
  (QUOTE |OUTFORM;width;$I;14|)
  (QUOTE |SPADreplace|)
  (QUOTE |outformWidth|))

(DEFUN |OUTFORM;width;$I;14| (|a| |$|) (|outformWidth| |a|))

(PUT (QUOTE |OUTFORM;height;$I;15|) (QUOTE |SPADreplace|) (QUOTE |height|))

(DEFUN |OUTFORM;height;$I;15| (|a| |$|) (|height| |a|))

(PUT
  (QUOTE |OUTFORM;subHeight;$I;16|)
  (QUOTE |SPADreplace|)
  (QUOTE |subspan|))

(DEFUN |OUTFORM;subHeight;$I;16| (|a| |$|) (|subspan| |a|))

(PUT
  (QUOTE |OUTFORM;superHeight;$I;17|)
  (QUOTE |SPADreplace|)
  (QUOTE |superspan|))

(DEFUN |OUTFORM;superHeight;$I;17| (|a| |$|) (|superspan| |a|))

(PUT
  (QUOTE |OUTFORM;height;I;18|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM NIL 20)))

(DEFUN |OUTFORM;height;I;18| (|$|) 20)

(PUT
  (QUOTE |OUTFORM;width;I;19|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM NIL 66)))

(DEFUN |OUTFORM;width;I;19| (|$|) 66)

(DEFUN |OUTFORM;center;$I$;20| (|a| |w| |$|)

```

```

(SPADCALL
  (SPADCALL
    (QUOTIENT2 (|-| |w| (SPADCALL |a| (QREFELT |$| 30))) 2)
    (QREFELT |$| 36))
  |a|
  (QREFELT |$| 37)))

(DEFUN |OUTFORM;left;$I$;21| (|a| |w| |$|)
  (SPADCALL
    |a|
    (SPADCALL (|-| |w| (SPADCALL |a| (QREFELT |$| 30))) (QREFELT |$| 36))
    (QREFELT |$| 37)))

(DEFUN |OUTFORM;right;$I$;22| (|a| |w| |$|)
  (SPADCALL
    (SPADCALL (|-| |w| (SPADCALL |a| (QREFELT |$| 30))) (QREFELT |$| 36))
    |a|
    (QREFELT |$| 37)))

(DEFUN |OUTFORM;center;2$;23| (|a| |$|)
  (SPADCALL |a| (SPADCALL (QREFELT |$| 35)) (QREFELT |$| 38)))

(DEFUN |OUTFORM;left;2$;24| (|a| |$|)
  (SPADCALL |a| (SPADCALL (QREFELT |$| 35)) (QREFELT |$| 39)))

(DEFUN |OUTFORM;right;2$;25| (|a| |$|)
  (SPADCALL |a| (SPADCALL (QREFELT |$| 35)) (QREFELT |$| 40)))

(DEFUN |OUTFORM;vspace;I$;26| (|n| |$|)
  (COND
    ((EQL |n| 0) (SPADCALL (QREFELT |$| 12)))
    ((QUOTE T)
      (SPADCALL
        (|OUTFORM;sform| " " |$|)
        (SPADCALL (|-| |n| 1) (QREFELT |$| 44))
        (QREFELT |$| 45)))))

(DEFUN |OUTFORM;hspace;I$;27| (|n| |$|)
  (COND
    ((EQL |n| 0) (SPADCALL (QREFELT |$| 12)))
    ((QUOTE T) (|OUTFORM;sform| (|fillerSpaces| |n|) |$|))))

(DEFUN |OUTFORM;rspace;2I$;28| (|n| |m| |$|)
  (COND
    ((OR (EQL |n| 0) (EQL |m| 0)) (SPADCALL (QREFELT |$| 12)))
    ((QUOTE T)
      (SPADCALL
        (SPADCALL |n| (QREFELT |$| 36))
        (SPADCALL |n| (|-| |m| 1) (QREFELT |$| 46))
        (QREFELT |$| 45)))))

```

```

(DEFUN |OUTFORM;matrix;L$;29| (|l| $)
  (PROG (G82748 |l| G82749 |lv|)
    (RETURN
      (SEQ (LETT |lv|
        (PROGN
          (LETT G82748 NIL |OUTFORM;matrix;L$;29|)
          (SEQ (LETT |l| NIL |OUTFORM;matrix;L$;29|)
            (LETT G82749 |l| |OUTFORM;matrix;L$;29|) G190
            (COND
              ((OR (ATOM G82749)
                (PROGN
                  (LETT |l| (CAR G82749)
                    |OUTFORM;matrix;L$;29|)
                    NIL)))
                (GO G191)))
            (SEQ (EXIT (LETT G82748
              (CONS (LIST2VEC |l|) G82748)
              |OUTFORM;matrix;L$;29|)))
              (LETT G82749 (CDR G82749)
                |OUTFORM;matrix;L$;29|)
              (GO G190) G191 (EXIT (NREVERSEO G82748))))
            |OUTFORM;matrix;L$;29|)
          (EXIT (CONS (|OUTFORM;eform| 'MATRIX $) (LIST2VEC |lv|)))))))

(DEFUN |OUTFORM;pil;L$;30| (|l| |$|)
  (CONS (|OUTFORM;eform| (QUOTE SC) |$|) |l|))

(DEFUN |OUTFORM;commaSeparate;L$;31| (|l| |$|)
  (CONS (|OUTFORM;eform| (QUOTE AGGLST) |$|) |l|))

(DEFUN |OUTFORM;semicolonSeparate;L$;32| (|l| |$|)
  (CONS (|OUTFORM;eform| (QUOTE AGGSET) |$|) |l|))

(DEFUN |OUTFORM;blankSeparate;L$;33| (|l| $)
  (PROG (|c| |u| G82757 |l|)
    (RETURN
      (SEQ (LETT |c| (|OUTFORM;eform| 'CONCATB $)
        |OUTFORM;blankSeparate;L$;33|)
        (LETT |l| NIL |OUTFORM;blankSeparate;L$;33|)
        (SEQ (LETT |u| NIL |OUTFORM;blankSeparate;L$;33|)
          (LETT G82757 (SPADCALL |l| (QREFELT $ 53))
            |OUTFORM;blankSeparate;L$;33|)
          G190
          (COND
            ((OR (ATOM G82757)
              (PROGN
                (LETT |u| (CAR G82757)
                  |OUTFORM;blankSeparate;L$;33|)
                  NIL)))
              (GO G191)))
          (SEQ (EXIT (LETT G82757
            (CONS (LIST2VEC |l|) G82757)
            |OUTFORM;blankSeparate;L$;33|)))
            (LETT G82757 (CDR G82757)
              |OUTFORM;blankSeparate;L$;33|)
            (GO G190) G191 (EXIT (NREVERSEO G82757))))
            |OUTFORM;blankSeparate;L$;33|)
          (EXIT (CONS (|OUTFORM;eform| 'CONCATB $) (LIST2VEC |lv|)))))))

```

```

      (GO G191)))
    (SEQ (EXIT (COND
      ((EQCAR |u| |c|)
        (LETT |l1|
          (SPADCALL (CDR |u|) |l1|
            (QREFELT $ 54))
          |OUTFORM;blankSeparate;L$;33|))
      ('T
        (LETT |l1| (CONS |u| |l1|)
          |OUTFORM;blankSeparate;L$;33|))))))
    (LETT G82757 (CDR G82757)
      |OUTFORM;blankSeparate;L$;33|)
    (GO G190) G191 (EXIT NIL))
  (EXIT (CONS |c| |l1|))))))

(DEFUN |OUTFORM;brace;2$;34| (|a| |$|)
  (LIST (|OUTFORM;iform| (QUOTE BRACE) |$|) |a|))

(DEFUN |OUTFORM;brace;L$;35| (|l| |$|)
  (SPADCALL (SPADCALL |l| (QREFELT |$| 51)) (QREFELT |$| 56)))

(DEFUN |OUTFORM;bracket;2$;36| (|a| |$|)
  (LIST (|OUTFORM;iform| (QUOTE BRACKET) |$|) |a|))

(DEFUN |OUTFORM;bracket;L$;37| (|l| |$|)
  (SPADCALL (SPADCALL |l| (QREFELT |$| 51)) (QREFELT |$| 58)))

(DEFUN |OUTFORM;paren;2$;38| (|a| |$|)
  (LIST (|OUTFORM;iform| (QUOTE PAREN) |$|) |a|))

(DEFUN |OUTFORM;paren;L$;39| (|l| |$|)
  (SPADCALL (SPADCALL |l| (QREFELT |$| 51)) (QREFELT |$| 60)))

(DEFUN |OUTFORM;sub;3$;40| (|a| |b| |$|)
  (LIST (|OUTFORM;iform| (QUOTE SUB) |$|) |a| |b|))

(DEFUN |OUTFORM;super;3$;41| (|a| |b| |$|)
  (LIST
    (|OUTFORM;iform| (QUOTE SUPERSUB) |$|)
    |a|
    (|OUTFORM;sform| " " |$|) |b|))

(DEFUN |OUTFORM;presub;3$;42| (|a| |b| |$|)
  (LIST
    (|OUTFORM;iform| (QUOTE SUPERSUB) |$|)
    |a|
    (|OUTFORM;sform| " " |$|)
    (|OUTFORM;sform| " " |$|)
    (|OUTFORM;sform| " " |$|)
    |b|))

```

```

(DEFUN |OUTFORM;presuper;3$;43| (|a| |b| |$|)
  (LIST
    (|OUTFORM;eform| (QUOTE SUPERSUB) |$|)
    |a|
    (|OUTFORM;sform| " " |$|)
    (|OUTFORM;sform| " " |$|)
    |b|))

(DEFUN |OUTFORM;scripts;$L$;44| (|a| |l| |$|)
  (COND
    ((SPADCALL |l| (QREFELT |$| 66)) |a|)
    ((SPADCALL (SPADCALL |l| (QREFELT |$| 67)) (QREFELT |$| 66))
      (SPADCALL |a| (SPADCALL |l| (QREFELT |$| 68)) (QREFELT |$| 62)))
    ((QUOTE T) (CONS (|OUTFORM;eform| (QUOTE SUPERSUB) |$|) (CONS |a| |l|))))))

(DEFUN |OUTFORM;supersub;$L$;45| (|a| |l| |$|)
  (SEQ
    (COND
      ((ODDP (SPADCALL |l| (QREFELT |$| 71)))
        (LETT |l|
          (SPADCALL |l| (LIST (SPADCALL (QREFELT |$| 12))) (QREFELT |$| 73))
          |OUTFORM;supersub;$L$;45|)))
      (EXIT (CONS (|OUTFORM;eform| (QUOTE ALTSUPERSUB) |$|) (CONS |a| |l|))))))

(DEFUN |OUTFORM;hconcat;3$;46| (|a| |b| |$|)
  (LIST (|OUTFORM;eform| (QUOTE CONCAT) |$|) |a| |b|))

(DEFUN |OUTFORM;hconcat;$L$;47| (|l| |$|)
  (CONS (|OUTFORM;eform| (QUOTE CONCAT) |$|) |l|))

(DEFUN |OUTFORM;vconcat;3$;48| (|a| |b| |$|)
  (LIST (|OUTFORM;eform| (QUOTE VCONCAT) |$|) |a| |b|))

(DEFUN |OUTFORM;vconcat;$L$;49| (|l| |$|)
  (CONS (|OUTFORM;eform| (QUOTE VCONCAT) |$|) |l|))

(DEFUN |OUTFORM;~=;3$;50| (|a| |b| |$|)
  (LIST (|OUTFORM;sform| "~=" |$|) |a| |b|))

(DEFUN |OUTFORM;<;3$;51| (|a| |b| |$|)
  (LIST (|OUTFORM;sform| "<" |$|) |a| |b|))

(DEFUN |OUTFORM;>;3$;52| (|a| |b| |$|)
  (LIST (|OUTFORM;sform| ">" |$|) |a| |b|))

(DEFUN |OUTFORM;<=;3$;53| (|a| |b| |$|)
  (LIST (|OUTFORM;sform| "<=" |$|) |a| |b|))

(DEFUN |OUTFORM;>=;3$;54| (|a| |b| |$|)

```

```

(LIST (|OUTFORM;sform| ">=" |$|) |a| |b|))

(DEFUN |OUTFORM;+;3$;55| (|a| |b| |$|)
  (LIST (|OUTFORM;sform| "+" |$|) |a| |b|))

(DEFUN |OUTFORM;-;3$;56| (|a| |b| |$|)
  (LIST (|OUTFORM;sform| "-" |$|) |a| |b|))

(DEFUN |OUTFORM;-;2$;57| (|a| |$|)
  (LIST (|OUTFORM;sform| "-" |$|) |a|))

(DEFUN |OUTFORM;*;3$;58| (|a| |b| |$|)
  (LIST (|OUTFORM;sform| "*" |$|) |a| |b|))

(DEFUN |OUTFORM;/;3$;59| (|a| |b| |$|)
  (LIST (|OUTFORM;sform| "/" |$|) |a| |b|))

(DEFUN |OUTFORM;**;3$;60| (|a| |b| |$|)
  (LIST (|OUTFORM;sform| "**" |$|) |a| |b|))

(DEFUN |OUTFORM;div;3$;61| (|a| |b| |$|)
  (LIST (|OUTFORM;sform| "div" |$|) |a| |b|))

(DEFUN |OUTFORM;rem;3$;62| (|a| |b| |$|)
  (LIST (|OUTFORM;sform| "rem" |$|) |a| |b|))

(DEFUN |OUTFORM;quo;3$;63| (|a| |b| |$|)
  (LIST (|OUTFORM;sform| "quo" |$|) |a| |b|))

(DEFUN |OUTFORM;exquo;3$;64| (|a| |b| |$|)
  (LIST (|OUTFORM;sform| "exquo" |$|) |a| |b|))

(DEFUN |OUTFORM;and;3$;65| (|a| |b| |$|)
  (LIST (|OUTFORM;sform| "and" |$|) |a| |b|))

(DEFUN |OUTFORM;or;3$;66| (|a| |b| |$|)
  (LIST (|OUTFORM;sform| "or" |$|) |a| |b|))

(DEFUN |OUTFORM;not;2$;67| (|a| |$|)
  (LIST (|OUTFORM;sform| "not" |$|) |a|))

(DEFUN |OUTFORM;SEGMENT;3$;68| (|a| |b| |$|)
  (LIST (|OUTFORM;eform| (QUOTE SEGMENT) |$|) |a| |b|))

(DEFUN |OUTFORM;SEGMENT;2$;69| (|a| |$|)
  (LIST (|OUTFORM;eform| (QUOTE SEGMENT) |$|) |a|))

(DEFUN |OUTFORM;binomial;3$;70| (|a| |b| |$|)
  (LIST (|OUTFORM;eform| (QUOTE BINOMIAL) |$|) |a| |b|))

```

```

(DEFUN |OUTFORM;empty;$;71| (|$|)
  (LIST (|OUTFORM;eform| (QUOTE NOTHING) |$|)))

(DEFUN |OUTFORM;infix?;$B;72| (|a| $)
  (PROG (G82802 |e|)
    (RETURN
      (SEQ (EXIT (SEQ (LETT |e|
        (COND
          ((IDENTP |a|) |a|)
          ((STRINGP |a|) (INTERN |a|))
          ('T
            (PROGN
              (LETT G82802 'NIL
                |OUTFORM;infix?;$B;72|)
              (GO G82802))))
          |OUTFORM;infix?;$B;72|)
        (EXIT (COND ((GET |e| 'INFXOP) 'T) ('T 'NIL))))))
      G82802 (EXIT G82802))))))

(PUT (QUOTE |OUTFORM;elt;$L$;73|) (QUOTE |SPADreplace|) (QUOTE CONS))

(DEFUN |OUTFORM;elt;$L$;73| (|a| |l| |$|) (CONS |a| |l|))

(DEFUN |OUTFORM;prefix;$L$;74| (|a| |l| |$|)
  (COND
    ((NULL (SPADCALL |a| (QREFELT |$| 98))) (CONS |a| |l|))
    ((QUOTE T)
      (SPADCALL |a|
        (SPADCALL (SPADCALL |l| (QREFELT |$| 51)) (QREFELT |$| 60))
        (QREFELT |$| 37)))))

(DEFUN |OUTFORM;infix;$L$;75| (|a| |l| |$|)
  (COND
    ((SPADCALL |l| (QREFELT |$| 66)) (SPADCALL (QREFELT |$| 12)))
    ((SPADCALL (SPADCALL |l| (QREFELT |$| 67)) (QREFELT |$| 66))
      (SPADCALL |l| (QREFELT |$| 68)))
    ((SPADCALL |a| (QREFELT |$| 98)) (CONS |a| |l|))
    ((QUOTE T)
      (SPADCALL
        (LIST
          (SPADCALL |l| (QREFELT |$| 68))
          |a|
          (SPADCALL |a| (SPADCALL |l| (QREFELT |$| 101)) (QREFELT |$| 102)))
        (QREFELT |$| 75)))))

(DEFUN |OUTFORM;infix;4$;76| (|a| |b| |c| |$|)
  (COND
    ((SPADCALL |a| (QREFELT |$| 98)) (LIST |a| |b| |c|))
    ((QUOTE T) (SPADCALL (LIST |b| |a| |c|) (QREFELT |$| 75)))))

```

```

(DEFUN |OUTFORM;postfix;3$;77| (|a| |b| |$|)
  (SPADCALL |b| |a| (QREFELT |$| 37)))

(DEFUN |OUTFORM;string;2$;78| (|a| |$|)
  (LIST (|OUTFORM;iform| (QUOTE STRING) |$|) |a|))

(DEFUN |OUTFORM;quote;2$;79| (|a| |$|)
  (LIST (|OUTFORM;iform| (QUOTE QUOTE) |$|) |a|))

(DEFUN |OUTFORM;overbar;2$;80| (|a| |$|)
  (LIST (|OUTFORM;iform| (QUOTE OVERBAR) |$|) |a|))

(DEFUN |OUTFORM;dot;2$;81| (|a| |$|)
  (SPADCALL |a| (|OUTFORM;sform| "." |$|) (QREFELT |$| 63)))

(DEFUN |OUTFORM;prime;2$;82| (|a| |$|)
  (SPADCALL |a| (|OUTFORM;sform| "," |$|) (QREFELT |$| 63)))

(DEFUN |OUTFORM;dot;$Nni$;83| (|a| |nn| |$|)
  (PROG (|s|)
    (RETURN
      (SEQ
        (LETT |s|
          (|MAKE-FULL-CVEC| |nn| (SPADCALL "." (QREFELT |$| 110)))
          |OUTFORM;dot;$Nni$;83|)
        (EXIT (SPADCALL |a| (|OUTFORM;sform| |s| |$|) (QREFELT |$| 63)))))))

(DEFUN |OUTFORM;prime;$Nni$;84| (|a| |nn| |$|)
  (PROG (|s|)
    (RETURN
      (SEQ
        (LETT |s|
          (|MAKE-FULL-CVEC| |nn| (SPADCALL "," (QREFELT |$| 110)))
          |OUTFORM;prime;$Nni$;84|)
        (EXIT (SPADCALL |a| (|OUTFORM;sform| |s| |$|) (QREFELT |$| 63)))))))

(DEFUN |OUTFORM;overlabel;3$;85| (|a| |b| |$|)
  (LIST (|OUTFORM;iform| (QUOTE OVERLABEL) |$|) |a| |b|))

(DEFUN |OUTFORM;box;2$;86| (|a| |$|)
  (LIST (|OUTFORM;iform| (QUOTE BOX) |$|) |a|))

(DEFUN |OUTFORM;zag;3$;87| (|a| |b| |$|)
  (LIST (|OUTFORM;iform| (QUOTE ZAG) |$|) |a| |b|))

(DEFUN |OUTFORM;root;2$;88| (|a| |$|)
  (LIST (|OUTFORM;iform| (QUOTE ROOT) |$|) |a|))

(DEFUN |OUTFORM;root;3$;89| (|a| |b| |$|)
  (LIST (|OUTFORM;iform| (QUOTE ROOT) |$|) |a| |b|))

```



```

(DEFUN |OUTFORM;over;3$;90| (|a| |b| |$|)
  (LIST (|OUTFORM;eform| (QUOTE OVER) |$|) |a| |b|))

(DEFUN |OUTFORM;slash;3$;91| (|a| |b| |$|)
  (LIST (|OUTFORM;eform| (QUOTE SLASH) |$|) |a| |b|))

(DEFUN |OUTFORM;assign;3$;92| (|a| |b| |$|)
  (LIST (|OUTFORM;eform| (QUOTE LET) |$|) |a| |b|))

(DEFUN |OUTFORM;label;3$;93| (|a| |b| |$|)
  (LIST (|OUTFORM;eform| (QUOTE EQUATNUM) |$|) |a| |b|))

(DEFUN |OUTFORM;rarrow;3$;94| (|a| |b| |$|)
  (LIST (|OUTFORM;eform| (QUOTE TAG) |$|) |a| |b|))

(DEFUN |OUTFORM;differentiate;$Nni$;95| (|a| |nn| $)
  (PROG (G82832 |r| |s|)
    (RETURN
      (SEQ (COND
        ((ZEROP |nn|) |a|)
        ((< |nn| 4) (SPADCALL |a| |nn| (QREFELT $ 112)))
        ('T
          (SEQ (LETT |r|
            (SPADCALL
              (PROG1 (LETT G82832 |nn|
                |OUTFORM;differentiate;$Nni$;95|)
                (|check-subtype| (> G82832 0)
                  '(|PositiveInteger|) G82832))
              (QREFELT $ 125))
              |OUTFORM;differentiate;$Nni$;95|)
            (LETT |s| (SPADCALL |r| (QREFELT $ 126))
              |OUTFORM;differentiate;$Nni$;95|)
            (EXIT (SPADCALL |a|
              (SPADCALL (|OUTFORM;sform| |s| $)
                (QREFELT $ 60))
                (QREFELT $ 63))))))))))

(DEFUN |OUTFORM;sum;2$;96| (|a| |$|)
  (LIST (|OUTFORM;eform| (QUOTE SIGMA) |$|) (SPADCALL (QREFELT |$| 12)) |a|))

(DEFUN |OUTFORM;sum;3$;97| (|a| |b| |$|)
  (LIST (|OUTFORM;eform| (QUOTE SIGMA) |$|) |b| |a|))

(DEFUN |OUTFORM;sum;4$;98| (|a| |b| |c| |$|)
  (LIST (|OUTFORM;eform| (QUOTE SIGMA2) |$|) |b| |c| |a|))

(DEFUN |OUTFORM;prod;2$;99| (|a| |$|)
  (LIST (|OUTFORM;eform| (QUOTE PI) |$|) (SPADCALL (QREFELT |$| 12)) |a|))

```

```

(DEFUN |OUTFORM;prod;3$;100| (|a| |b| |$|)
  (LIST (|OUTFORM;eform| (QUOTE PI) |$|) |b| |a|))

(DEFUN |OUTFORM;prod;4$;101| (|a| |b| |c| |$|)
  (LIST (|OUTFORM;eform| (QUOTE PI2) |$|) |b| |c| |a|))

(DEFUN |OUTFORM;int;2$;102| (|a| |$|)
  (LIST
    (|OUTFORM;eform| (QUOTE INTSIGN) |$|)
    (SPADCALL (QREFELT |$| 12))
    (SPADCALL (QREFELT |$| 12))
    |a|))

(DEFUN |OUTFORM;int;3$;103| (|a| |b| |$|)
  (LIST
    (|OUTFORM;eform| (QUOTE INTSIGN) |$|)
    |b|
    (SPADCALL (QREFELT |$| 12))
    |a|))

(DEFUN |OUTFORM;int;4$;104| (|a| |b| |c| |$|)
  (LIST (|OUTFORM;eform| (QUOTE INTSIGN) |$|) |b| |c| |a|))

(DEFUN |OutputForm| ()
  (PROG ()
    (RETURN
      (PROG (G82846)
        (RETURN
          (COND
            ((LETT G82846 (HGET |$ConstructorCache| '|OutputForm|)
              |OutputForm|)
              (|CDRwithIncrement| (CDAR G82846)))
            ('T
              (UNWIND-PROTECT
                (PROG1 (CDDAR (HPUT |$ConstructorCache| '|OutputForm|
                  (LIST
                    (CONS NIL (CONS 1 (|OutputForm;|))))))
                  (LETT G82846 T |OutputForm|))
                (COND
                  ((NOT G82846)
                    (HREM |$ConstructorCache| '|OutputForm|))))))))))

(DEFUN |OutputForm;| ()
  (PROG (|dv$| $ |pv$|)
    (RETURN
      (PROGN
        (LETT |dv$| '(|OutputForm|) |OutputForm|)
        (LETT $ (make-array 138) |OutputForm|)
        (QSETREFV $ 0 |dv$|)
        (QSETREFV $ 3

```

```

        (LETT |pv$| (|buildPredVector| 0 0 NIL) |OutputForm|))
        (|haddProp| |$ConstructorCache| ' |OutputForm| NIL (CONS 1 $))
        (|stuffDomainSlots| $)
        (QSETREFV $ 6 (|List| $))
        $))))

(setf (get
      (QUOTE |OutputForm|)
      (QUOTE |infovec|))
      (LIST
        (QUOTE #(
          NIL NIL NIL NIL NIL NIL (QUOTE |Rep|) (|Void|) |OUTFORM;print;$V;1|
          (|Boolean|) (|String|) (0 . |empty?|) |OUTFORM;empty;$;71|
          |OUTFORM;message;$;2| |OUTFORM;messagePrint;SV;3|
          |OUTFORM;=;2$B;4| |OUTFORM;=;3$;5| (|OutputForm|)
          |OUTFORM;coerce;2$;6| (|Integer|) |OUTFORM;outputForm;I$;7|
          (|Symbol|) |OUTFORM;outputForm;S$;8| (|DoubleFloat|)
          |OUTFORM;outputForm;Df$;9| (|Character|) (5 . |quote|)
          (9 . |concat|) (15 . |concat|) |OUTFORM;outputForm;S$;13|
          |OUTFORM;width;$I;14| |OUTFORM;height;$I;15| | |
          |OUTFORM;subHeight;$I;16| |OUTFORM;superHeight;$I;17|
          |OUTFORM;height;I;18| |OUTFORM;width;I;19| |OUTFORM;hspace;I$;27|
          |OUTFORM;hconcat;3$;46| |OUTFORM;center;$I$;20|
          |OUTFORM;left;$I$;21| |OUTFORM;right;$I$;22| |OUTFORM;center;2$;23|
          |OUTFORM;left;2$;24| |OUTFORM;right;2$;25| |OUTFORM;vspace;I$;26|
          |OUTFORM;vconcat;3$;48| |OUTFORM;rspace;2I$;28| (|List| 49)
          |OUTFORM;matrix;L$;29| (|List| |$|) |OUTFORM;pile;L$;30|
          |OUTFORM;commaSeparate;L$;31| |OUTFORM;semicolonSeparate;L$;32|
          (21 . |reverse|) (26 . |append|) |OUTFORM;blankSeparate;L$;33|
          |OUTFORM;brace;2$;34| |OUTFORM;brace;L$;35| |OUTFORM;bracket;2$;36|
          |OUTFORM;bracket;L$;37| |OUTFORM;paren;2$;38| |OUTFORM;paren;L$;39|
          |OUTFORM;sub;3$;40| |OUTFORM;super;3$;41| |OUTFORM;presub;3$;42|
          |OUTFORM;presuper;3$;43| (32 . |null|) (37 . |rest|) (42 . |first|)
          |OUTFORM;scripts;$L$;44| (|NonNegativeInteger|) (47 . |#|)
          (|List| |$$|) (52 . |append|) |OUTFORM;supersub;$L$;45|
          |OUTFORM;hconcat;L$;47| |OUTFORM;vconcat;L$;49| |OUTFORM;^=;3$;50|
          |OUTFORM;<;3$;51| |OUTFORM;>;3$;52| |OUTFORM;<=;3$;53|
          |OUTFORM;>=;3$;54| |OUTFORM;+;3$;55| |OUTFORM;-;3$;56|
          |OUTFORM;-;2$;57| |OUTFORM;*;3$;58| |OUTFORM;/;3$;59|
          |OUTFORM;**;3$;60| |OUTFORM;div;3$;61| |OUTFORM;rem;3$;62|
          |OUTFORM;quo;3$;63| |OUTFORM;exquo;3$;64| |OUTFORM;and;3$;65|
          |OUTFORM;or;3$;66| |OUTFORM;not;2$;67| |OUTFORM;SEGMENT;3$;68|
          |OUTFORM;SEGMENT;2$;69| |OUTFORM;binomial;3$;70|
          |OUTFORM;infix?;$B;72| |OUTFORM;elt;$L$;73| |OUTFORM;prefix;$L$;74|
          (58 . |rest|) |OUTFORM;infix;$L$;75| |OUTFORM;infix;4$;76|
          |OUTFORM;postfix;3$;77| |OUTFORM;string;2$;78| |OUTFORM;quote;2$;79|
          |OUTFORM;overbar;2$;80| |OUTFORM;dot;2$;81| |OUTFORM;prime;2$;82|
          (63 . |char|) |OUTFORM;dot;$Nni$;83| |OUTFORM;prime;$Nni$;84|
          |OUTFORM;overlabel;3$;85| |OUTFORM;box;2$;86| |OUTFORM;zag;3$;87|
          |OUTFORM;root;2$;88| |OUTFORM;root;3$;89| |OUTFORM;over;3$;90|

```

```

|OUTFORM;slash;3$;91| |OUTFORM;assign;3$;92|
|OUTFORM;label;3$;93| |OUTFORM;arrow;3$;94| (|PositiveInteger|)
(|NumberFormats|) (68 . |FormatRoman|) (73 . |lowerCase|)
|OUTFORM;differentiate;$Nni$;95| |OUTFORM;sum;2$;96| | |
|OUTFORM;sum;3$;97| |OUTFORM;sum;4$;98| |OUTFORM;prod;2$;99|
|OUTFORM;prod;3$;100| |OUTFORM;prod;4$;101| |OUTFORM;int;2$;102|
|OUTFORM;int;3$;103| |OUTFORM;int;4$;104| (|SingleInteger|))
(QUOTE #(|^=| 78 |zag| 84 |width| 90 |vspace| 99 |vconcat| 104
|supersub| 115 |superHeight| 121 |super| 126 |sum| 132 |subHeight|
150 |sub| 155 |string| 161 |slash| 166 |semicolonSeparate| 172
|scripts| 177 |rspace| 183 |root| 189 |right| 200 |rem| 211
|rarrow| 217 |quote| 223 |quo| 228 |prod| 234 |print| 252
|prime| 257 |presuper| 268 |presub| 274 |prefix| 280 |postfix|
286 |pile| 292 |paren| 297 |overlabel| 307 |overbar| 313 |over|
318 |outputForm| 324 |or| 344 |not| 350 |messagePrint| 355
|message| 360 |matrix| 365 |left| 370 |latex| 381 |label| 386
|int| 392 |infix?| 410 |infix| 415 |hspace| 428 |height| 433
|hconcat| 442 |hash| 453 |exquo| 458 |empty| 464 |elt| 468 |dot|
474 |div| 485 |differentiate| 491 |commaSeparate| 497 |coerce|
502 |center| 507 |bracket| 518 |brace| 528 |box| 538 |blankSeparate|
543 |binomial| 548 |assign| 554 |and| 560 |^=| 566 SEGMENT 572
|>=| 583 |>| 589 |=| 595 |<=| 607 |<| 613 |/>| 619 |-| 625 |+|
636 |**| 642 |*| 648))
(QUOTE NIL)
(CONS
(|makeByteWordVec2| 1 (QUOTE (0 0 0)))
(CONS
(QUOTE #(|SetCategory&| |BasicType&| NIL))
(CONS
(QUOTE #((|SetCategory|) (|BasicType|) (|CoercibleTo| 17)))
(|makeByteWordVec2| 137 (QUOTE (1 10 9 0 11 0 25 0 26 2 10 0 0 25
27 2 10 0 25 0 28 1 6 0 0 53 2 6 0 0 0 54 1 6 9 0 66 1 6 0 0 67 1
6 2 0 68 1 6 70 0 71 2 72 0 0 0 73 1 72 0 0 101 1 25 0 10 110 1 124
10 123 125 1 10 0 0 126 2 0 9 0 0 1 2 0 0 0 0 115 0 0 19 35 1 0 19
0 30 1 0 0 19 44 1 0 0 49 76 2 0 0 0 0 45 2 0 0 0 49 74 1 0 19 0
33 2 0 0 0 63 2 0 0 0 0 129 3 0 0 0 0 130 1 0 0 0 128 1 0 19
0 32 2 0 0 0 0 62 1 0 0 0 105 2 0 0 0 0 119 1 0 0 49 52 2 0 0 0
49 69 2 0 0 19 19 46 1 0 0 0 116 2 0 0 0 0 117 1 0 0 0 43 2 0 0
0 19 40 2 0 0 0 0 89 2 0 0 0 0 122 1 0 0 0 106 2 0 0 0 0 90 3 0
0 0 0 0 133 1 0 0 0 131 2 0 0 0 0 132 1 0 7 0 8 2 0 0 0 70 112 1
0 0 0 109 2 0 0 0 0 65 2 0 0 0 0 64 2 0 0 0 49 100 2 0 0 0 0 104
1 0 0 49 50 1 0 0 49 61 1 0 0 0 60 2 0 0 0 0 113 1 0 0 0 107 2 0
0 0 0 118 1 0 0 10 29 1 0 0 23 24 1 0 0 21 22 1 0 0 19 20 2 0 0
0 0 93 1 0 0 0 94 1 0 7 10 14 1 0 0 10 13 1 0 0 47 48 1 0 0 0 42
2 0 0 0 19 39 1 0 10 0 1 2 0 0 0 0 121 3 0 0 0 0 0 136 2 0 0 0 0
135 1 0 0 0 134 1 0 9 0 98 2 0 0 0 49 102 3 0 0 0 0 0 103 1 0 0
19 36 0 0 19 34 1 0 19 0 31 1 0 0 49 75 2 0 0 0 0 37 1 0 137 0 1
2 0 0 0 0 91 0 0 0 12 2 0 0 0 49 99 2 0 0 0 70 111 1 0 0 0 108 2
0 0 0 0 88 2 0 0 0 70 127 1 0 0 49 51 1 0 17 0 18 1 0 0 0 41 2 0
0 0 19 38 1 0 0 0 58 1 0 0 49 59 1 0 0 49 57 1 0 0 0 56 1 0 0 0

```



```

        (LETT G96739 T |PositiveInteger|)
      (COND
        ((NOT G96739)
          (HREM |$ConstructorCache| '|PositiveInteger|)))))))))

(DEFUN |PositiveInteger;| ()
  (PROG (|dv$| $ |pv$|)
    (RETURN
      (PROGN
        (LETT |dv$| '|PositiveInteger| |PositiveInteger|)
        (LETT $ (make-array 12) |PositiveInteger|)
        (QSETREFV $ 0 |dv$|)
        (QSETREFV $ 3
          (LETT |pv$| (|buildPredVector| 0 0 NIL) |PositiveInteger|)
          (|haddProp| |$ConstructorCache| '|PositiveInteger| NIL
            (CONS 1 $))
          (|stuffDomainSlots| $)
          $))))

(setf (get
  (QUOTE |PositiveInteger|)
  (QUOTE |infovec|))
  (LIST
    (QUOTE
      #(NIL NIL NIL NIL NIL
        (|NonNegativeInteger|)
        (|PositiveInteger|)
        (|Boolean|)
        (|Union| |$| (QUOTE "failed"))
        (|SingleInteger|)
        (|String|)
        (|OutputForm|)))
    (QUOTE #(|~|=| 0 |sample| 6 |recip| 10 |one?| 15 |min| 20 |max| 26
      |latex| 32 |hash| 37 |gcd| 42 |coerce| 48 |^| 53 |One| 65
      |>=| 69 |>| 75 |=| 81 |<=| 87 |<| 93 |+| 99 |**| 105 |*| 117))
    (QUOTE (((|commutative| "*" ) . 0)))
    (CONS
      (|makeByteWordVec2| 1 (QUOTE (0 0 0 0 0 0)))
      (CONS
        (QUOTE #(|Monoid&| |AbelianSemiGroup&| |SemiGroup&| |OrderedSet&|
          |SetCategory&| |BasicType&| NIL))
        (CONS
          (QUOTE #(
            (|Monoid|)
            (|AbelianSemiGroup|)
            (|SemiGroup|)
            (|OrderedSet|)
            (|SetCategory|)
            (|BasicType|)
            (|CoercibleTo| 11)))
          )
        )
      )
    )
  )

```

```

(|makeByteWordVec2| 11
 (QUOTE (2 0 7 0 0 1 0 0 0 1 1 0 8 0 1 1 0 7 0 1 2 0 0 0 0 1 2 0 0 0
         0 1 1 0 10 0 1 1 0 9 0 1 2 0 0 0 0 1 1 0 11 0 1 2 0 0 0 6 1
         2 0 0 0 5 1 0 0 0 1 2 0 7 0 0 1 2 0 7 0 0 1 2 0 7 0 0 1 2 0
         7 0 0 1 2 0 7 0 0 1 2 0 0 0 0 1 2 0 0 0 6 1 2 0 0 0 5 1 2 0
         0 0 0 1 2 0 0 6 0 1))))))
(QUOTE |lookupComplete|)))

(setf (get (QUOTE |PositiveInteger|) (QUOTE NILADIC)) T)

```

PRIMARR.lsp BOOTSTRAP

PRIMARR depends on itself. We need to break this cycle to build the algebra. So we keep a cached copy of the translated **PRIMARR** category which we can write into the **MID** directory. We compile the lisp code and copy the **PRIMARR.o** file to the **OUT** directory. This is eventually forcibly replaced by a recompiled version.

Note that this code is not included in the generated catdef.spad file.

— PRIMARR.lsp BOOTSTRAP —

```

(|/VERSIONCHECK| 2)

(put (QUOTE |PRIMARR;#;$Nni;1|) (QUOTE |SPADreplace|) (QUOTE QVSIZE))

(defun |PRIMARR;#;$Nni;1| (|x| |$|) (QVSIZE |x|))

(put (QUOTE |PRIMARR;minIndex;$I;2|)
 (QUOTE |SPADreplace|) (QUOTE (XLAM (|x|) 0)))

(defun |PRIMARR;minIndex;$I;2| (|x| |$|) 0)

(put (QUOTE |PRIMARR;empty;$;3|)
 (QUOTE |SPADreplace|) (QUOTE (XLAM NIL (make-array 0))))

(defun |PRIMARR;empty;$;3| (|x| |$|) (make-array 0))

(defun |PRIMARR;new;NniS$;4| (|n| |x| |$|)
 (SPADCALL (make-array |n|) |x| (QREFELT |$| 12)))

(put (QUOTE |PRIMARR;qelt;$IS;5|) (QUOTE |SPADreplace|) (QUOTE ELT))

(defun |PRIMARR;qelt;$IS;5| (|x| |i| |$|) (ELT |x| |i|))

(put (QUOTE |PRIMARR;elt;$IS;6|) (QUOTE |SPADreplace|) (QUOTE ELT))

```

```

(DEFUN |PRIMARR;elt;$IS;6| (|x| |i| |$|) (ELT |x| |i|))

(PUT (QUOTE |PRIMARR;qsetelt!;$I2S;7|) (QUOTE |SPADreplace|) (QUOTE SETELT))

(DEFUN |PRIMARR;qsetelt!;$I2S;7| (|x| |i| |s| |$|) (SETELT |x| |i| |s|))

(PUT (QUOTE |PRIMARR;setelt;$I2S;8|) (QUOTE |SPADreplace|) (QUOTE SETELT))

(DEFUN |PRIMARR;setelt;$I2S;8| (|x| |i| |s| |$|) (SETELT |x| |i| |s|))

(DEFUN |PRIMARR;fill!;$S$;9| (|x| |s| $)
  (PROG (|i| G82338)
    (RETURN
      (SEQ (SEQ (LETT |i| 0 |PRIMARR;fill!;$S$;9|)
        (LETT G82338 (QVMAXINDEX |x|) |PRIMARR;fill!;$S$;9|)
        G190 (COND ((QSGREATERP |i| G82338) (GO G191)))
        (SEQ (EXIT (SETELT |x| |i| |s|)))
        (LETT |i| (QSADD1 |i|) |PRIMARR;fill!;$S$;9|) (GO G190)
        G191 (EXIT NIL))
        (EXIT |x|))))))

(DEFUN |PrimitiveArray| (G82348)
  (PROG ()
    (RETURN
      (PROG (G82349)
        (RETURN
          (COND
            ((LETT G82349
              (|lassocShiftWithFunction|
                (LIST (|devaluate| G82348))
                (HGET |$ConstructorCache| '|PrimitiveArray|)
                '|domainEqualList|)
              |PrimitiveArray|)
              (|CDRwithIncrement| G82349))
            ('T
              (UNWIND-PROTECT
                (PROG1 (|PrimitiveArray| G82348)
                  (LETT G82349 T |PrimitiveArray|))
                (COND
                  ((NOT G82349)
                    (HREM |$ConstructorCache| '|PrimitiveArray|))))))))))

(DEFUN |PrimitiveArray;| (|#1|)
  (PROG (DV$1 |dv$| $ G82347 |pv$|)
    (RETURN
      (PROGN
        (LETT DV$1 (|devaluate| |#1|) |PrimitiveArray|)
        (LETT |dv$| (LIST '|PrimitiveArray| DV$1) |PrimitiveArray|)
        (LETT $ (make-array 35) |PrimitiveArray|)

```



```

(QSETREFV $ 0 |dv$|)
(QSETREFV $ 3
  (LETT |pv$|
    (|buildPredVector| 0 0
      (LIST (|HasCategory| |#1| '(|SetCategory|))
        (|HasCategory| |#1|
          '(|ConvertibleTo| (|InputForm|)))
        (LETT G82347
          (|HasCategory| |#1| '(|OrderedSet|))
          |PrimitiveArray|)
        (OR G82347
          (|HasCategory| |#1| '(|SetCategory|))
          (|HasCategory| (|Integer|) '(|OrderedSet|))
          (AND (|HasCategory| |#1|
            (LIST '|Evalable|
              (|devaluate| |#1|)))
            (|HasCategory| |#1| '(|SetCategory|)))
          (OR (AND (|HasCategory| |#1|
            (LIST '|Evalable|
              (|devaluate| |#1|)))
            G82347)
            (AND (|HasCategory| |#1|
              (LIST '|Evalable|
                (|devaluate| |#1|)))
              (|HasCategory| |#1|
                '(|SetCategory|))))))
          |PrimitiveArray|))
      (|haddProp| |$ConstructorCache| '|PrimitiveArray| (LIST DV$1)
        (CONS 1 $))
      (|stuffDomainSlots| $)
      (QSETREFV $ 6 |#1|)
      $))))

(setf (get (QUOTE |PrimitiveArray|) (QUOTE |infovec|))
  (LIST
    (QUOTE
      #(NIL NIL NIL NIL NIL NIL (|local| |#1|) (|NonNegativeInteger|)
        |PRIMARR;#;$Nni;1| (|Integer|) |PRIMARR;minIndex;$I;2|
        |PRIMARR;empty;$;3| |PRIMARR;fill!;$S$;9| |PRIMARR;new;Nni$;4|
        |PRIMARR;qelt;$IS;5| |PRIMARR;elt;$IS;6| |PRIMARR;qsetelt!;$I2S;7|
        |PRIMARR;setelt;$I2S;8| (|Mapping| 6 6 6) (|Boolean|) (|List| 6)
        (|Equation| 6) (|List| 21) (|Mapping| 19 6) (|Mapping| 19 6 6)
        (|UniversalSegment| 9) (|Void|) (|Mapping| 6 6) (|InputForm|)
        (|OutputForm|) (|String|) (|SingleInteger|) (|List| |$|)
        (|Union| 6 (QUOTE "failed")) (|List| 9)))
    (QUOTE
      #(|~|=| 0 |swap!| 6 |sorted?| 13 |sort!| 24 |sort| 35 |size?| 46 |setelt|
        52 |select| 66 |sample| 72 |reverse!| 76 |reverse| 81 |removeDuplicates|
        86 |remove| 91 |reduce| 103 |qsetelt!| 124 |qelt| 131 |position| 137
        |parts| 156 |new| 161 |more?| 167 |minIndex| 173 |min| 178 |merge| 184

```

```

|members| 197 |member?| 202 |maxIndex| 208 |max| 213 |map!| 219 |map|
225 |less?| 238 |latex| 244 |insert| 249 |indices| 263 |index?| 268
|hash| 274 |first| 279 |find| 284 |fill!| 290 |every?| 296 |eval| 302
|eq?| 328 |entry?| 334 |entries| 340 |empty?| 345 |empty| 350 |elt| 354
|delete| 373 |count| 385 |copyInto!| 397 |copy| 404 |convert| 409
|construct| 414 |concat| 419 |coerce| 442 |any?| 447 |>=| 453 |>| 459
|=| 465 |<=| 471 |<| 477 |#| 483))
(QUOTE ((|shallowlyMutable| . 0) (|finiteAggregate| . 0)))
(CONS
  (|makeByteWordVec2| 7 (QUOTE (0 0 0 0 0 0 3 0 0 7 4 0 0 7 1 2 4)))
  (CONS
    (QUOTE #(|OneDimensionalArrayAggregate&| |FiniteLinearAggregate&|
      |LinearAggregate&| |IndexedAggregate&| |Collection&|
      |HomogeneousAggregate&| |OrderedSet&| |Aggregate&| |EltableAggregate&|
      |Evalable&| |SetCategory&| NIL NIL |InnerEvalable&| NIL NIL |BasicType&|))
    (CONS
      (QUOTE
        #((|OneDimensionalArrayAggregate| 6) (|FiniteLinearAggregate| 6)
          (|LinearAggregate| 6) (|IndexedAggregate| 9 6) (|Collection| 6)
          (|HomogeneousAggregate| 6) (|OrderedSet|) (|Aggregate|)
          (|EltableAggregate| 9 6) (|Evalable| 6) (|SetCategory|) (|Type|)
          (|Eltable| 9 6) (|InnerEvalable| 6 6) (|CoercibleTo| 29)
          (|ConvertibleTo| 28) (|BasicType|)))
        (|makeByteWordVec2| 34
          (QUOTE
            (2 1 19 0 0 1 3 0 26 0 9 9 1 1 3 19 0 1 2 0 19 24 0 1 1 3 0 0 1 2 0 0
              24 0 1 1 3 0 0 1 2 0 0 24 0 1 2 0 19 0 7 1 3 0 6 0 25 6 1 3 0 6 0 9
              6 17 2 0 0 23 0 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 0 0 1 2 1 0 6 0 1
              2 0 0 23 0 1 4 1 6 18 0 6 6 1 3 0 6 18 0 6 1 2 0 6 18 0 1 3 0 6 0 9
              6 16 2 0 6 0 9 14 2 1 9 6 0 1 3 1 9 6 0 9 1 2 0 9 23 0 1 1 0 20 0 1
              2 0 0 7 6 13 2 0 19 0 7 1 1 5 9 0 10 2 3 0 0 0 1 2 3 0 0 0 1 3 0 0
              24 0 0 1 1 0 20 0 1 2 1 19 6 0 1 1 5 9 0 1 2 3 0 0 0 1 2 0 0 27 0 1
              3 0 0 18 0 0 1 2 0 0 27 0 1 2 0 19 0 7 1 1 1 30 0 1 3 0 0 0 0 9 1 3
              0 0 6 0 9 1 1 0 34 0 1 2 0 19 9 0 1 1 1 31 0 1 1 5 6 0 1 2 0 33 23
              0 1 2 0 0 0 6 12 2 0 19 23 0 1 3 6 0 0 20 20 1 2 6 0 0 21 1 3 6 0 0
              6 6 1 2 6 0 0 22 1 2 0 19 0 0 1 2 1 19 6 0 1 1 0 20 0 1 1 0 19 0 1
              0 0 0 11 2 0 0 0 25 1 2 0 6 0 9 15 3 0 6 0 9 6 1 2 0 0 0 9 1 2 0 0
              0 25 1 2 1 7 6 0 1 2 0 7 23 0 1 3 0 0 0 0 9 1 1 0 0 0 1 1 2 28 0 1
              1 0 0 20 1 1 0 0 32 1 2 0 0 6 0 1 2 0 0 0 0 1 2 0 0 0 6 1 1 1 29 0
              1 2 0 19 23 0 1 2 3 19 0 0 1 2 3 19 0 0 1 2 1 19 0 0 1 2 3 19 0 0 1
              2 3 19 0 0 1 1 0 7 0 8))))))
(QUOTE |lookupComplete|)))

```

REF.lsp BOOTSTRAP

REF depends on a chain of files. We need to break this cycle to build the algebra. So

we keep a cached copy of the translated **REF** category which we can write into the **MID** directory. We compile the lisp code and copy the **REF.o** file to the **OUT** directory. This is eventually forcibly replaced by a recompiled version.

Note that this code is not included in the generated catdef.spad file.

— REF.lsp BOOTSTRAP —

```
(|/VERSIONCHECK| 2)

(PUT (QUOTE |REF;=;2$B;1|) (QUOTE |SPADreplace|) (QUOTE EQ))

(DEFUN |REF;=;2$B;1| (|p| |q| |$|) (EQ |p| |q|))

(PUT (QUOTE |REF;ref;$S;2|) (QUOTE |SPADreplace|) (QUOTE LIST))

(DEFUN |REF;ref;$S;2| (|v| |$|) (LIST |v|))

(PUT (QUOTE |REF;elt;$S;3|) (QUOTE |SPADreplace|) (QUOTE QCAR))

(DEFUN |REF;elt;$S;3| (|p| |$|) (QCAR |p|))

(DEFUN |REF;setelt;$S;4| (|p| |v| |$|) (PROGN (RPLACA |p| |v|) (QCAR |p|)))

(PUT (QUOTE |REF;deref;$S;5|) (QUOTE |SPADreplace|) (QUOTE QCAR))

(DEFUN |REF;deref;$S;5| (|p| |$|) (QCAR |p|))

(DEFUN |REF;setref;$S;6| (|p| |v| |$|) (PROGN (RPLACA |p| |v|) (QCAR |p|)))

(DEFUN |REF;coerce;$Of;7| (|p| |$|)
  (SPADCALL
    (SPADCALL "ref" (QREFELT |$| 17))
    (LIST (SPADCALL (QCAR |p|) (QREFELT |$| 18)))
    (QREFELT |$| 20)))

(DEFUN |Reference| (G82336)
  (PROG ()
    (RETURN
      (PROG (G82337)
        (RETURN
          (COND
            ((LETT G82337
              (|lassocShiftWithFunction|
                (LIST (|devaluate| G82336))
                (HGET |$ConstructorCache| '|Reference|)
                '|domainEqualList|)
              |Reference|)
            (|CDRwithIncrement| G82337)))
    ('T
```

```

      (UNWIND-PROTECT
        (PROG1 (|Reference;| G82336)
          (LETT G82337 T |Reference|))
        (COND
          ((NOT G82337) (HREM |$ConstructorCache| '|Reference|)))))))))

(DEFUN |Reference;| (|#1|)
  (PROG (DV$1 |dv$| $ |pv$|)
    (RETURN
      (PROGN
        (LETT DV$1 (|devalue| |#1|) |Reference|)
        (LETT |dv$| (LIST '|Reference| DV$1) |Reference|)
        (LETT $ (make-array 23) |Reference|)
        (QSETREFV $ 0 |dv$|)
        (QSETREFV $ 3
          (LETT |pv$|
            (|buildPredVector| 0 0
              (LIST (|HasCategory| |#1| '|SetCategory|))))
            |Reference|))
        (|haddProp| |$ConstructorCache| '|Reference| (LIST DV$1)
          (CONS 1 $))
        (|stuffDomainSlots| $)
        (QSETREFV $ 6 |#1|)
        (QSETREFV $ 7 (|Record| (|:| |value| |#1|)))
        (COND
          ((|testBitVector| |pv$| 1)
            (QSETREFV $ 21
              (CONS (|dispatchFunction| |REF;coerce;$Of;7|) $))))
          $))))

(setf (get
  (QUOTE |Reference|)
  (QUOTE |infovec|))
  (LIST
    (QUOTE
      #(NIL NIL NIL NIL NIL NIL (|local| |#1|) (QUOTE |Rep|) (|Boolean|)
        |REF;|=;2$B;1| |REF;ref;$S;2| |REF;elt;$S;3| |REF;setelt;$2S;4|
        |REF;deref;$S;5| |REF;setref;$2S;6| (|String|) (|OutputForm|)
        (0 . |message|) (5 . |coerce|) (|List| |$|) (10 . |prefix|)
        (16 . |coerce|) (|SingleInteger|)))
    (QUOTE #(|~|=| 21 |setref| 27 |setelt| 33 |ref| 39 |latex| 44 |hash| 49
      |elt| 54 |deref| 59 |coerce| 64 |=| 69))
    (QUOTE NIL)
    (CONS
      (|makeByteWordVec2| 1 (QUOTE (1 0 1 1)))
      (CONS
        (QUOTE #(|SetCategory&| NIL |BasicType&| NIL))
        (CONS
          (QUOTE #((|SetCategory|) (|Type|) (|BasicType|) (|CoercibleTo| 16)))
          (|makeByteWordVec2| 22

```

```

(QUOTE (1 16 0 15 17 1 6 16 0 18 2 16 0 0 19 20 1 0 16 0 21 2 1 8 0
        0 1 2 0 6 0 6 14 2 0 6 0 6 12 1 0 0 6 10 1 1 15 0 1 1 1 22
        0 1 1 0 6 0 11 1 0 6 0 13 1 1 16 0 21 2 0 8 0 0 9))))))
(QUOTE |lookupComplete|)))

```

SINT.lsp BOOTSTRAP

— SINT.lsp BOOTSTRAP —

```

(/VERSIONCHECK 2)

(DEFUN |SINT;writeOMSingleInt| (|dev| |x| $)
  (SEQ
    (COND
      ((QSLESSP |x| 0)
        (SEQ
          (SPADCALL |dev| (QREFELT $ 9))
          (SPADCALL |dev| "arith1" "unaryminus" (QREFELT $ 11))
          (SPADCALL |dev| (QSMINUS |x|) (QREFELT $ 13))
          (EXIT (SPADCALL |dev| (QREFELT $ 14))))))
      ((QUOTE T) (SPADCALL |dev| |x| (QREFELT $ 13))))))

(DEFUN |SINT;OMwrite;$S;2| (|x| $)
  (PROG (|sp| |dev| |s|)
    (RETURN
      (SEQ
        (LETT |s| "" |SINT;OMwrite;$S;2|)
        (LETT |sp| (OM-STRINGTOSTRINGPTR |s|) |SINT;OMwrite;$S;2|)
        (LETT |dev|
          (SPADCALL |sp| (SPADCALL (QREFELT $ 16)) (QREFELT $ 17))
          |SINT;OMwrite;$S;2|)
        (SPADCALL |dev| (QREFELT $ 18))
        (|SINT;writeOMSingleInt| |dev| |x| $)
        (SPADCALL |dev| (QREFELT $ 19))
        (SPADCALL |dev| (QREFELT $ 20))
        (LETT |s| (OM-STRINGPTRTOSTRING |sp|) |SINT;OMwrite;$S;2|)
        (EXIT |s|))))))

(DEFUN |SINT;OMwrite;$BS;3| (|x| |wholeObj| $)
  (PROG (|sp| |dev| |s|)
    (RETURN
      (SEQ
        (LETT |s| "" |SINT;OMwrite;$BS;3|)
        (LETT |sp| (OM-STRINGTOSTRINGPTR |s|) |SINT;OMwrite;$BS;3|)

```

```

(LETT |dev|
  (SPADCALL |sp| (SPADCALL (QREFELT $ 16)) (QREFELT $ 17))
  |SINT;OMwrite;$BS;3|)
(COND (|wholeObj| (SPADCALL |dev| (QREFELT $ 18))))
(|SINT;writeOMSingleInt| |dev| |x| $)
(COND (|wholeObj| (SPADCALL |dev| (QREFELT $ 19))))
(SPADCALL |dev| (QREFELT $ 20))
(LETT |s| (OM-STRINGPTRTOSTRING |sp|) |SINT;OMwrite;$BS;3|)
(EXIT |s|))))))

(DEFUN |SINT;OMwrite;Omd$V;4| (|dev| |x| $)
  (SEQ
    (SPADCALL |dev| (QREFELT $ 18))
    (|SINT;writeOMSingleInt| |dev| |x| $)
    (EXIT (SPADCALL |dev| (QREFELT $ 19)))))

(DEFUN |SINT;OMwrite;Omd$BV;5| (|dev| |x| |wholeObj| $)
  (SEQ
    (COND (|wholeObj| (SPADCALL |dev| (QREFELT $ 18))))
    (|SINT;writeOMSingleInt| |dev| |x| $)
    (EXIT (COND (|wholeObj| (SPADCALL |dev| (QREFELT $ 19)))))))

(PUT
  (QUOTE |SINT;reducedSystem;MM;6|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|m|) |m|)))

(DEFUN |SINT;reducedSystem;MM;6| (|m| $) |m|)

(DEFUN |SINT;coerce;$Of;7| (|x| $)
  (SPADCALL |x| (QREFELT $ 30)))

(PUT
  (QUOTE |SINT;convert;$I;8|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM (|x|) |x|)))

(DEFUN |SINT;convert;$I;8| (|x| $) |x|)

(DEFUN |SINT;*;I2$;9| (|i| |y| $)
  (QSTIMES (SPADCALL |i| (QREFELT $ 33)) |y|))

(PUT
  (QUOTE |SINT;Zero;$;10|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM NIL 0)))

(DEFUN |SINT;Zero;$;10| ($) 0)

(PUT

```

```

(QUOTE |SINT;One;$;11|)
(QUOTE |SPADreplace|)
(QUOTE (XLAM NIL 1)))

(DEFUN |SINT;One;$;11| ($) 1)

(PUT
  (QUOTE |SINT;base;$;12|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM NIL 2)))

(DEFUN |SINT;base;$;12| ($) 2)

(PUT
  (QUOTE |SINT;max;$;13|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM NIL MOST-POSITIVE-FIXNUM)))

(DEFUN |SINT;max;$;13| ($) MOST-POSITIVE-FIXNUM)

(PUT
  (QUOTE |SINT;min;$;14|)
  (QUOTE |SPADreplace|)
  (QUOTE (XLAM NIL MOST-NEGATIVE-FIXNUM)))

(DEFUN |SINT;min;$;14| ($) MOST-NEGATIVE-FIXNUM)

(PUT
  (QUOTE |SINT;=;2$B;15|)
  (QUOTE |SPADreplace|)
  (QUOTE EQL))

(DEFUN |SINT;=;2$B;15| (|x| |y| $)
  (EQL |x| |y|))

(PUT
  (QUOTE |SINT;~;2$;16|)
  (QUOTE |SPADreplace|)
  (QUOTE LOGNOT))

(DEFUN |SINT;~;2$;16| (|x| $)
  (LOGNOT |x|))

(PUT
  (QUOTE |SINT;not;2$;17|)
  (QUOTE |SPADreplace|)
  (QUOTE LOGNOT))

(DEFUN |SINT;not;2$;17| (|x| $)
  (LOGNOT |x|))

```

```
(PUT
  (QUOTE |SINT;/\;3$;18|)
  (QUOTE |SPADreplace|)
  (QUOTE LOGAND))

(DEFUN |SINT;/\;3$;18| (|x| |y| $)
  (LOGAND |x| |y|))

(PUT
  (QUOTE |SINT;\/;3$;19|)
  (QUOTE |SPADreplace|)
  (QUOTE LOGIOR))

(DEFUN |SINT;\/;3$;19| (|x| |y| $)
  (LOGIOR |x| |y|))

(PUT
  (QUOTE |SINT;Not;2$;20|)
  (QUOTE |SPADreplace|)
  (QUOTE LOGNOT))

(DEFUN |SINT;Not;2$;20| (|x| $)
  (LOGNOT |x|))

(PUT
  (QUOTE |SINT;And;3$;21|)
  (QUOTE |SPADreplace|)
  (QUOTE LOGAND))

(DEFUN |SINT;And;3$;21| (|x| |y| $)
  (LOGAND |x| |y|))

(PUT
  (QUOTE |SINT;Or;3$;22|)
  (QUOTE |SPADreplace|)
  (QUOTE LOGIOR))

(DEFUN |SINT;Or;3$;22| (|x| |y| $)
  (LOGIOR |x| |y|))

(PUT
  (QUOTE |SINT;xor;3$;23|)
  (QUOTE |SPADreplace|)
  (QUOTE LOGXOR))

(DEFUN |SINT;xor;3$;23| (|x| |y| $)
  (LOGXOR |x| |y|))

(PUT
```



```

(QUOTE |SINT;<;2$B;24|)
(QUOTE |SPADreplace|)
(QUOTE QSLESSP))

(DEFUN |SINT;<;2$B;24| (|x| |y| $)
  (QSLESSP |x| |y|))

(PUT
  (QUOTE |SINT;inc;2$;25|)
  (QUOTE |SPADreplace|)
  (QUOTE QSADD1))

(DEFUN |SINT;inc;2$;25| (|x| $)
  (QSADD1 |x|))

(PUT
  (QUOTE |SINT;dec;2$;26|)
  (QUOTE |SPADreplace|)
  (QUOTE QSSUB1))

(DEFUN |SINT;dec;2$;26| (|x| $)
  (QSSUB1 |x|))

(PUT
  (QUOTE |SINT;-;2$;27|)
  (QUOTE |SPADreplace|)
  (QUOTE QSMINUS))

(DEFUN |SINT;-;2$;27| (|x| $)
  (QSMINUS |x|))

(PUT
  (QUOTE |SINT;+;3$;28|)
  (QUOTE |SPADreplace|)
  (QUOTE QSPLUS))

(DEFUN |SINT;+;3$;28| (|x| |y| $)
  (QSPLUS |x| |y|))

(PUT
  (QUOTE |SINT;-;3$;29|)
  (QUOTE |SPADreplace|)
  (QUOTE QSDIFFERENCE))

(DEFUN |SINT;-;3$;29| (|x| |y| $)
  (QSDIFFERENCE |x| |y|))

(PUT
  (QUOTE |SINT;*;3$;30|)
  (QUOTE |SPADreplace|)

```

```

(QUOTE QSTIMES))

(DEFUN |SINT;*;3$;30| (|x| |y| $)
  (QSTIMES |x| |y|))

(DEFUN |SINT;**;$Nni$;31| (|x| |n| $)
  (SPADCALL (EXPT |x| |n|) (QREFELT $ 33)))

(PUT
  (QUOTE |SINT;quo;3$;32|)
  (QUOTE |SPADreplace|)
  (QUOTE QSQUOTIENT))

(DEFUN |SINT;quo;3$;32| (|x| |y| $)
  (QSQUOTIENT |x| |y|))

(PUT
  (QUOTE |SINT;rem;3$;33|)
  (QUOTE |SPADreplace|)
  (QUOTE QSREMAINDER))

(DEFUN |SINT;rem;3$;33| (|x| |y| $)
  (QSREMAINDER |x| |y|))

(DEFUN |SINT;divide;2$R;34| (|x| |y| $)
  (CONS (QSQUOTIENT |x| |y|) (QSREMAINDER |x| |y|)))

(PUT (QUOTE |SINT;gcd;3$;35|)
  (QUOTE |SPADreplace|) (QUOTE GCD))

(DEFUN |SINT;gcd;3$;35| (|x| |y| $)
  (GCD |x| |y|))

(PUT
  (QUOTE |SINT;abs;2$;36|)
  (QUOTE |SPADreplace|)
  (QUOTE QSABSVAL))

(DEFUN |SINT;abs;2$;36| (|x| $)
  (QSABSVAL |x|))

(PUT
  (QUOTE |SINT;odd?;$B;37|)
  (QUOTE |SPADreplace|)
  (QUOTE QSODDP))

(DEFUN |SINT;odd?;$B;37| (|x| $)
  (QSODDP |x|))

(PUT

```

```

(QUOTE |SINT;zero?;$B;38|)
(QUOTE |SPADreplace|)
(QUOTE QSZEROP))

(DEFUN |SINT;zero?;$B;38| (|x| $)
  (QSZEROP |x|))

(PUT
  (QUOTE |SINT;max;3$;39|)
  (QUOTE |SPADreplace|)
  (QUOTE QSMAX))

(DEFUN |SINT;max;3$;39| (|x| |y| $)
  (QSMAX |x| |y|))

(PUT
  (QUOTE |SINT;min;3$;40|)
  (QUOTE |SPADreplace|)
  (QUOTE QSMIN))

(DEFUN |SINT;min;3$;40| (|x| |y| $)
  (QSMIN |x| |y|))

(PUT
  (QUOTE |SINT;hash;2$;41|)
  (QUOTE |SPADreplace|)
  (QUOTE SXHASH))

(DEFUN |SINT;hash;2$;41| (|x| $)
  (SXHASH |x|))

(PUT
  (QUOTE |SINT;length;2$;42|)
  (QUOTE |SPADreplace|)
  (QUOTE INTEGER-LENGTH))

(DEFUN |SINT;length;2$;42| (|x| $)
  (INTEGER-LENGTH |x|))

(PUT
  (QUOTE |SINT;shift;3$;43|)
  (QUOTE |SPADreplace|)
  (QUOTE QSLEFTSHIFT))

(DEFUN |SINT;shift;3$;43| (|x| |n| $)
  (QSLEFTSHIFT |x| |n|))

(PUT
  (QUOTE |SINT;mulmod;4$;44|)
  (QUOTE |SPADreplace|)

```

```

(QUOTE QSMULTMOD))

(DEFUN |SINT;mulmod;4$;44| (|a| |b| |p| $)
  (QSMULTMOD |a| |b| |p|))

(PUT
  (QUOTE |SINT;addmod;4$;45|)
  (QUOTE |SPADreplace|)
  (QUOTE QSADDMOD))

(DEFUN |SINT;addmod;4$;45| (|a| |b| |p| $)
  (QSADDMOD |a| |b| |p|))

(PUT
  (QUOTE |SINT;submod;4$;46|)
  (QUOTE |SPADreplace|)
  (QUOTE QSDIFMOD))

(DEFUN |SINT;submod;4$;46| (|a| |b| |p| $)
  (QSDIFMOD |a| |b| |p|))

(PUT
  (QUOTE |SINT;negative?;$B;47|)
  (QUOTE |SPADreplace|)
  (QUOTE QSMINUSP))

(DEFUN |SINT;negative?;$B;47| (|x| $)
  (QSMINUSP |x|))

(PUT
  (QUOTE |SINT;reducedSystem;MVR;48|)
  (QUOTE |SPADreplace|)
  (QUOTE CONS))

(DEFUN |SINT;reducedSystem;MVR;48| (|m| |v| $)
  (CONS |m| |v|))

(DEFUN |SINT;positiveRemainder;3$;49| (|x| |n| $)
  (PROG (|r|)
    (RETURN
      (SEQ
        (LETT |r| (QSREMAINDER |x| |n|) |SINT;positiveRemainder;3$;49|)
        (EXIT
          (COND
            ((QSMINUSP |r|)
              (COND
                ((QSMINUSP |n|) (QSDIFFERENCE |x| |n|))
                ((QUOTE T) (QSPLUS |r| |n|))))
            ((QUOTE T) |r|)))))))

```

```

(DEFUN |SINT;coerce;I$;50| (|x| $)
  (SEQ
    (COND
      ((NULL (< MOST-POSITIVE-FIXNUM |x|))
        (COND ((NULL (< |x| MOST-NEGATIVE-FIXNUM)) (EXIT |x|))))
      (EXIT (|error| "integer too large to represent in a machine word"))))

(DEFUN |SINT;random;$;51| ($)
  (SEQ
    (SETELT $ 6 (REMAINDER (TIMES 314159269 (QREFELT $ 6)) 2147483647))
    (EXIT (REMAINDER (QREFELT $ 6) 67108864)))

(PUT
  (QUOTE |SINT;random;2$;52|)
  (QUOTE |SPADreplace|)
  (QUOTE RANDOM))

(DEFUN |SINT;random;2$;52| (|n| $)
  (RANDOM |n|))

(DEFUN |SINT;unitNormal;$R;53| (|x| $)
  (COND
    ((QSLESSP |x| 0) (VECTOR -1 (QSMINUS |x|) -1))
    ((QUOTE T) (VECTOR 1 |x| 1))))

(DEFUN |SingleInteger| ()
  (PROG ()
    (RETURN
      (PROG (G1358)
        (RETURN
          (COND
            ((LETT G1358 (HGET |$ConstructorCache| '|SingleInteger|)
              |SingleInteger|)
              (|CDRwithIncrement| (CDAR G1358)))
            ('T
              (UNWIND-PROTECT
                (PROG1 (CDDAR (HPUT |$ConstructorCache| '|SingleInteger|
                  (LIST
                    (CONS NIL
                      (CONS 1 (|SingleInteger;|))))))
                  (LETT G1358 T |SingleInteger|))
                (COND
                  ((NOT G1358)
                    (HREM |$ConstructorCache| '|SingleInteger|))))))))))

(DEFUN |SingleInteger;| ()
  (PROG (|dv$| $ |pv$|)
    (RETURN
      (PROGN
        (LETT |dv$| '(|SingleInteger|) |SingleInteger|)

```

```

(LETT $ (make-array 103) |SingleInteger|)
(QSETREFV $ 0 |dv$|)
(QSETREFV $ 3
  (LETT |pv$| (|buildPredVector| 0 0 NIL) |SingleInteger|))
(|haddProp| |$ConstructorCache| '|SingleInteger| NIL
  (CONS 1 $))
(|stuffDomainSlots| $)
(QSETREFV $ 6 1)
$)))

(setf (get '|SingleInteger| '|infovec|)
  (LIST '#(NIL NIL NIL NIL NIL NIL '|seed| (|Void|)
    (|OpenMathDevice|) (0 . |OMputApp|) (|String|)
    (5 . |OMputSymbol|) (|Integer|) (12 . |OMputInteger|)
    (18 . |OMputEndApp|) (|OpenMathEncoding|)
    (23 . |OMencodingXML|) (27 . |OMopenString|)
    (33 . |OMputObject|) (38 . |OMputEndObject|)
    (43 . |OMclose|) |SINT;OMwrite;$S;2| (|Boolean|)
    |SINT;OMwrite;$BS;3| |SINT;OMwrite;Omd$V;4|
    |SINT;OMwrite;Omd$BV;5| (|Matrix| 12) (|Matrix| $)
    |SINT;reducedSystem;MM;6| (|OutputForm|) (48 . |coerce|)
    |SINT;coerce;$Of;7| |SINT;convert;$I;8| (53 . |coerce|)
    |SINT;*;I2$;9|
    (CONS IDENTITY
      (FUNCALL (|dispatchFunction| |SINT;Zero;$;10|) $))
    (CONS IDENTITY
      (FUNCALL (|dispatchFunction| |SINT;One;$;11|) $))
    |SINT;base;$;12| |SINT;max;$;13| |SINT;min;$;14|
    |SINT;=;2$B;15| |SINT;~;2$;16| |SINT;not;2$;17|
    |SINT;/\;3$;18| |SINT;\;/;3$;19| |SINT;Not;2$;20|
    |SINT;And;3$;21| |SINT;Or;3$;22| |SINT;xor;3$;23|
    |SINT;<;2$B;24| |SINT;inc;2$;25| |SINT;dec;2$;26|
    |SINT;-;2$;27| |SINT;+;3$;28| |SINT;-;3$;29|
    |SINT;*;3$;30| (|NonNegativeInteger|) |SINT;**;$Nni$;31|
    |SINT;quo;3$;32| |SINT;rem;3$;33|
    (|Record| (|:| |quotient| $) (|:| |remainder| $))
    |SINT;divide;2$R;34| |SINT;gcd;3$;35| |SINT;abs;2$;36|
    |SINT;odd?;$B;37| |SINT;zero?;$B;38| |SINT;max;3$;39|
    |SINT;min;3$;40| |SINT;hash;2$;41| |SINT;length;2$;42|
    |SINT;shift;3$;43| |SINT;mulmod;4$;44| |SINT;addmod;4$;45|
    |SINT;submod;4$;46| |SINT;negative?;$B;47|
    (|Record| (|:| |mat| 26) (|:| |vec| (|Vector| 12)))
    (|Vector| $) |SINT;reducedSystem;MVR;48|
    |SINT;positiveRemainder;3$;49| |SINT;coerce;I$;50|
    |SINT;random;$;51| |SINT;random;2$;52|
    (|Record| (|:| |unit| $) (|:| |canonical| $)
      (|:| |associate| $))
    |SINT;unitNormal;$R;53| (|Union| 85 '"failed")
    (|Fraction| 12) (|Union| $ '"failed") (|Float|)
    (|DoubleFloat|) (|Pattern| 12) (|PatternMatchResult| 12 $)
  ))

```

```

(|InputForm|) (|Union| 12 "failed")
(|Record| (|:| |coef| 94) (|:| |generator| $)) (|List| $)
(|Union| 94 "failed")
(|Record| (|:| |coef1| $) (|:| |coef2| $)
  (|:| |generator| $))
(|Record| (|:| |coef1| $) (|:| |coef2| $))
(|Union| 97 "failed") (|Factored| $)
(|SparseUnivariatePolynomial| $) (|PositiveInteger|)
(|SingleInteger|))
'#(= 58 ~ 64 |zero?| 69 |xor| 74 |unitNormal| 80
|unitCanonical| 85 |unit?| 90 |symmetricRemainder| 95
|subtractIfCan| 101 |submod| 107 |squareFreePart| 114
|squareFree| 119 |sizeLess?| 124 |sign| 130 |shift| 135
|sample| 141 |retractIfCan| 145 |retract| 150 |rem| 155
|reducedSystem| 161 |recip| 172 |rationalIfCan| 177
|rational?| 182 |rational| 187 |random| 192 |quo| 201
|principalIdeal| 207 |prime?| 212 |powmod| 217
|positiveRemainder| 224 |positive?| 230 |permutation| 235
|patternMatch| 241 |one?| 248 |odd?| 253 |not| 258
|nextItem| 263 |negative?| 268 |multiEuclidean| 273
|mulmod| 279 |min| 286 |max| 296 |mask| 306 |length| 311
|lcm| 316 |latex| 327 |invmod| 332 |init| 338 |inc| 342
|hash| 347 |gcdPolynomial| 357 |gcd| 363 |factorial| 374
|factor| 379 |extendedEuclidean| 384 |exquo| 397
|expressIdealMember| 403 |even?| 409 |euclideanSize| 414
|divide| 419 |differentiate| 425 |dec| 436 |copy| 441
|convert| 446 |coerce| 471 |characteristic| 491 |bit?| 495
|binomial| 501 |base| 507 |associates?| 511 |addmod| 517
|abs| 524 ^ 529 |\\|/| 541 |Zero| 547 |Or| 551 |One| 557
|OMwrite| 561 |Not| 585 D 590 |And| 601 >= 607 > 613 = 619
<= 625 < 631 |/\| 637 - 643 + 654 ** 660 * 672)
'((|noetherian| . 0) (|canonicalsClosed| . 0)
(|canonical| . 0) (|canonicalUnitNormal| . 0)
(|multiplicativeValuation| . 0) (|noZeroDivisors| . 0)
((|commutative| "*") . 0) (|rightUnitary| . 0)
(|leftUnitary| . 0) (|unitsKnown| . 0))
(CONS (|makeByteWordVec2| 1
  '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0))
  (CONS '#(|IntegerNumberSystem&| |EuclideanDomain&|
    |UniqueFactorizationDomain&| NIL NIL
    |GcdDomain&| |IntegralDomain&| |Algebra&|
    |Module&| NIL |Module&| NIL NIL |Module&| NIL
    |DifferentialRing&| |OrderedRing&| NIL
    |Module&| NIL |Module&| NIL NIL NIL NIL NIL
    NIL |Ring&| NIL NIL NIL NIL NIL NIL NIL NIL
    NIL NIL NIL NIL NIL |AbelianGroup&| NIL NIL
    |AbelianMonoid&| |Monoid&| NIL NIL NIL NIL

```

```

(|OrderedSet&| |AbelianSemiGroup&| |SemiGroup&|
|Logic&| NIL |SetCategory&| NIL NIL NIL NIL
|RetractableTo&| NIL NIL NIL |RetractableTo&|
NIL NIL NIL NIL NIL NIL |RetractableTo&| NIL
|BasicType&| NIL)
(CONS '#((|IntegerNumberSystem|)
(|EuclideanDomain|)
(|UniqueFactorizationDomain|)
(|PrincipalIdealDomain|)
(|OrderedIntegralDomain|) (|GcdDomain|)
(|IntegralDomain|) (|Algebra| $$)
(|Module| 12)
(|LinearlyExplicitRingOver| 12)
(|Module| G1062)
(|LinearlyExplicitRingOver| G1062)
(|CharacteristicZero|)
(|Module| G106217)
(|LinearlyExplicitRingOver| G106217)
(|DifferentialRing|) (|OrderedRing|)
(|CommutativeRing|) (|Module| |t#1|)
(|EntireRing|) (|Module| $$)
(|BiModule| 12 12)
(|BiModule| G1062 G1062)
(|BiModule| G106217 G106217)
(|OrderedAbelianGroup|)
(|BiModule| |t#1| |t#1|)
(|BiModule| $$ $$) (|Ring|)
(|RightModule| 12) (|LeftModule| 12)
(|RightModule| G1062)
(|LeftModule| G1062)
(|RightModule| G106217)
(|LeftModule| G106217)
(|OrderedCancellationAbelianMonoid|)
(|RightModule| |t#1|)
(|LeftModule| |t#1|) (|LeftModule| $$)
(|Rng|) (|RightModule| $$)
(|OrderedAbelianMonoid|)
(|AbelianGroup|)
(|OrderedAbelianSemiGroup|)
(|CancellationAbelianMonoid|)
(|AbelianMonoid|) (|Monoid|)
(|PatternMatchable| 12)
(|PatternMatchable| G1065)
(|StepThrough|)
(|PatternMatchable| G106220)
(|OrderedSet|) (|AbelianSemiGroup|)
(|SemiGroup|) (|Logic|) (|RealConstant|)
(|SetCategory|) (|OpenMath|)
(|CoercibleTo| G82356)
(|ConvertibleTo| 89)

```



```

(|ConvertibleTo| 91)
(|RetractableTo| 12)
(|ConvertibleTo| 12)
(|ConvertibleTo| G1064)
(|ConvertibleTo| G1063)
(|RetractableTo| G1061)
(|ConvertibleTo| G1060)
(|ConvertibleTo| 87)
(|ConvertibleTo| 88)
(|CombinatorialFunctionCategory|)
(|ConvertibleTo| G106219)
(|ConvertibleTo| G106218)
(|RetractableTo| G106216)
(|ConvertibleTo| G106215) (|BasicType|)
(|CoercibleTo| 29))
(|makeByteWordVec2| 102
  '(1 8 7 0 9 3 8 7 0 10 10 11 2 8 7 0 12
    13 1 8 7 0 14 0 15 0 16 2 8 0 10 15
    17 1 8 7 0 18 1 8 7 0 19 1 8 7 0 20 1
    12 29 0 30 1 0 0 12 33 2 0 22 0 0 1 1
    0 0 0 41 1 0 22 0 65 2 0 0 0 0 48 1 0
    82 0 83 1 0 0 0 1 1 0 22 0 1 2 0 0 0
    0 1 2 0 86 0 0 1 3 0 0 0 0 0 73 1 0 0
    0 1 1 0 99 0 1 2 0 22 0 0 1 1 0 12 0
    1 2 0 0 0 0 70 0 0 0 1 1 0 92 0 1 1 0
    12 0 1 2 0 0 0 0 59 1 0 26 27 28 2 0
    75 27 76 77 1 0 86 0 1 1 0 84 0 1 1 0
    22 0 1 1 0 85 0 1 1 0 0 0 81 0 0 0 80
    2 0 0 0 0 58 1 0 93 94 1 1 0 22 0 1 3
    0 0 0 0 0 1 2 0 0 0 0 78 1 0 22 0 1 2
    0 0 0 0 1 3 0 90 0 89 90 1 1 0 22 0 1
    1 0 22 0 64 1 0 0 0 42 1 0 86 0 1 1 0
    22 0 74 2 0 95 94 0 1 3 0 0 0 0 0 71
    0 0 0 39 2 0 0 0 0 67 0 0 0 38 2 0 0
    0 0 66 1 0 0 0 1 1 0 0 0 69 1 0 0 94
    1 2 0 0 0 0 1 1 0 10 0 1 2 0 0 0 0 1
    0 0 0 1 1 0 0 0 50 1 0 0 0 68 1 0 102
    0 1 2 0 100 100 100 1 1 0 0 94 1 2 0
    0 0 0 62 1 0 0 0 1 1 0 99 0 1 2 0 96
    0 0 1 3 0 98 0 0 0 1 2 0 86 0 0 1 2 0
    95 94 0 1 1 0 22 0 1 1 0 56 0 1 2 0
    60 0 0 61 1 0 0 0 1 2 0 0 0 56 1 1 0
    0 0 51 1 0 0 0 1 1 0 87 0 1 1 0 88 0
    1 1 0 89 0 1 1 0 91 0 1 1 0 12 0 32 1
    0 0 12 79 1 0 0 0 1 1 0 0 12 79 1 0
    29 0 31 0 0 56 1 2 0 22 0 0 1 2 0 0 0
    0 1 0 0 0 37 2 0 22 0 0 1 3 0 0 0 0 0
    72 1 0 0 0 63 2 0 0 0 56 1 2 0 0 0
    101 1 2 0 0 0 0 44 0 0 0 35 2 0 0 0 0
    47 0 0 0 36 3 0 7 8 0 22 25 2 0 10 0

```

```

22 23 2 0 7 8 0 24 1 0 10 0 21 1 0 0
0 45 1 0 0 0 1 2 0 0 0 56 1 2 0 0 0
46 2 0 22 0 0 1 2 0 22 0 0 1 2 0 22 0
0 40 2 0 22 0 0 1 2 0 22 0 0 49 2 0 0
0 0 43 1 0 0 0 52 2 0 0 0 0 54 2 0 0
0 0 53 2 0 0 0 56 57 2 0 0 0 101 1 2
0 0 0 0 55 2 0 0 12 0 34 2 0 0 56 0 1
2 0 0 101 0 1))))))

'|lookupComplete|))

(setf (get (QUOTE |SingleInteger|) (QUOTE NILADIC)) T)

```

SYMBOL.lsp BOOTSTRAP

SYMBOL depends on a chain of files. We need to break this cycle to build the algebra. So we keep a cached copy of the translated **SYMBOL** category which we can write into the **MID** directory. We compile the lisp code and copy the **SYMBOL.o** file to the **OUT** directory. This is eventually forcibly replaced by a recompiled version.

Note that this code is not included in the generated catdef.spad file.

— SYMBOL.lsp BOOTSTRAP —

```

(|/VERSIONCHECK| 2)

(DEFUN |SYMBOL;writeOMSym| (|dev| |x| $)
  (COND
    ((SPADCALL |x| (QREFELT $ 21))
      (|error| "Cannot convert a scripted symbol to OpenMath"))
    ('T (SPADCALL |dev| |x| (QREFELT $ 25)))))

(DEFUN |SYMBOL;OMwrite;$S;2| (|x| $)
  (PROG (|sp| |dev| |s|)
    (RETURN
      (SEQ (LETT |s| "" |SYMBOL;OMwrite;$S;2|)
        (LETT |sp| (OM-STRINGTOSTRINGPTR |s|) |SYMBOL;OMwrite;$S;2|)
        (LETT |dev|
          (SPADCALL |sp| (SPADCALL (QREFELT $ 27))
            (QREFELT $ 29))
          |SYMBOL;OMwrite;$S;2|)
        (SPADCALL |dev| (QREFELT $ 30))
        (|SYMBOL;writeOMSym| |dev| |x| $)
        (SPADCALL |dev| (QREFELT $ 31))
        (SPADCALL |dev| (QREFELT $ 32))
        (LETT |s| (OM-STRINGPTRTOSTRING |sp|) |SYMBOL;OMwrite;$S;2|)

```

```

(EXIT |s|))))))

(DEFUN |SYMBOL;OMwrite;$BS;3| (|x| |wholeObj| $)
  (PROG (|sp| |dev| |s|)
    (RETURN
      (SEQ (LETT |s| "" |SYMBOL;OMwrite;$BS;3|)
        (LETT |sp| (OM-STRINGTOSTRINGPTR |s|)
          |SYMBOL;OMwrite;$BS;3|)
        (LETT |dev|
          (SPADCALL |sp| (SPADCALL (QREFELT $ 27))
            (QREFELT $ 29))
          |SYMBOL;OMwrite;$BS;3|)
        (COND (|wholeObj| (SPADCALL |dev| (QREFELT $ 30))))
        (|SYMBOL;writeOMSym| |dev| |x| $)
        (COND (|wholeObj| (SPADCALL |dev| (QREFELT $ 31))))
        (SPADCALL |dev| (QREFELT $ 32))
        (LETT |s| (OM-STRINGPTRTOSTRING |sp|)
          |SYMBOL;OMwrite;$BS;3|)
        (EXIT |s|))))))

(DEFUN |SYMBOL;OMwrite;Omd$V;4| (|dev| |x| $)
  (SEQ (SPADCALL |dev| (QREFELT $ 30))
    (|SYMBOL;writeOMSym| |dev| |x| $)
    (EXIT (SPADCALL |dev| (QREFELT $ 31)))))

(DEFUN |SYMBOL;OMwrite;Omd$BV;5| (|dev| |x| |wholeObj| $)
  (SEQ (COND (|wholeObj| (SPADCALL |dev| (QREFELT $ 30))))
    (|SYMBOL;writeOMSym| |dev| |x| $)
    (EXIT (COND (|wholeObj| (SPADCALL |dev| (QREFELT $ 31)))))))

(DEFUN |SYMBOL;convert;$If;6| (|s| |$|) (SPADCALL |s| (QREFELT |$| 44)))

(PUT '|SYMBOL;convert;2$;7| '|SPADreplace| '(XLAM (|s|) |s|))

(DEFUN |SYMBOL;convert;2$;7| (|s| |$|) |s|)

(DEFUN |SYMBOL;coerce;$S;8| (|s| |$|) (VALUES (INTERN |s|)))

(PUT '|SYMBOL;=;2$B;9| '|SPADreplace| 'EQUAL)

(DEFUN |SYMBOL;=;2$B;9| (|x| |y| |$|) (EQUAL |x| |y|))

(PUT '|SYMBOL;<;2$B;10| '|SPADreplace|
  '(XLAM (|x| |y|) (GGREATERP |y| |x|)))

(DEFUN |SYMBOL;<;2$B;10| (|x| |y| |$|) (GGREATERP |y| |x|))

(DEFUN |SYMBOL;coerce;$Of;11| (|x| |$|) (SPADCALL |x| (QREFELT |$| 51)))

(DEFUN |SYMBOL;subscript;$L$;12| (|sy| |lx| $)

```

```

(SPADCALL |sy| (LIST |lx| NIL NIL NIL NIL) (QREFELT $ 54)))

(DEFUN |SYMBOL;elt;$L$;13| (|sy| |lx| $)
  (SPADCALL |sy| |lx| (QREFELT $ 56)))

(DEFUN |SYMBOL;superscript;$L$;14| (|sy| |lx| $)
  (SPADCALL |sy| (LIST NIL |lx| NIL NIL NIL) (QREFELT $ 54)))

(DEFUN |SYMBOL;argscript;$L$;15| (|sy| |lx| $)
  (SPADCALL |sy| (LIST NIL NIL NIL NIL |lx|) (QREFELT $ 54)))

(DEFUN |SYMBOL;patternMatch;$P2Pmr;16| (|x| |p| |l| $)
  (SPADCALL |x| |p| |l| (QREFELT $ 63)))

(DEFUN |SYMBOL;patternMatch;$P2Pmr;17| (|x| |p| |l| $)
  (SPADCALL |x| |p| |l| (QREFELT $ 69)))

(DEFUN |SYMBOL;convert;$P;18| (|x| |$|) (SPADCALL |x| (QREFELT |$| 72)))

(DEFUN |SYMBOL;convert;$P;19| (|x| |$|) (SPADCALL |x| (QREFELT |$| 74)))

(DEFUN |SYMBOL;syprefix| (|sc| $)
  (PROG (|ns| G108218 |n| G108219)
    (RETURN
      (SEQ (LETT |ns|
        (LIST (LENGTH (QVELT |sc| 3)) (LENGTH (QVELT |sc| 2))
          (LENGTH (QVELT |sc| 1)) (LENGTH (QVELT |sc| 0)))
        |SYMBOL;syprefix|)
      (SEQ G190
        (COND
          ((NULL (COND
            (((< (LENGTH |ns|) 2) 'NIL)
            ('T (ZEROP (|SPADfirst| |ns|)))))
          (GO G191)))
        (SEQ (EXIT (LETT |ns| (CDR |ns|) |SYMBOL;syprefix|)))
        NIL (GO G190) G191 (EXIT NIL))
      (EXIT (SPADCALL
        (CONS (STRCONC (QREFELT $ 37)
          (|SYMBOL;istring|
            (LENGTH (QVELT |sc| 4)) $))
        (PROGN
          (LETT G108218 NIL |SYMBOL;syprefix|)
          (SEQ (LETT |n| NIL |SYMBOL;syprefix|)
            (LETT G108219 (NREVERSE |ns|)
              |SYMBOL;syprefix|)
            G190
            (COND
              ((OR (ATOM G108219)
                (PROGN
                  (LETT |n| (CAR G108219)

```

```

        |SYMBOL;symprefix|)
      NIL))
    (GO G191)))
  (SEQ (EXIT
    (LETT G108218
      (CONS (|SYMBOL;istring| |n| $)
        G108218)
      |SYMBOL;symprefix|)))
    (LETT G108219 (CDR G108219)
      |SYMBOL;symprefix|)
    (GO G190) G191
    (EXIT (NREVERSEO G108218))))))
  (QREFELT $ 77))))))

(DEFUN |SYMBOL;syscripts| (|sc| $)
  (PROG (|all|)
    (RETURN
      (SEQ (LETT |all| (QVELT |sc| 3) |SYMBOL;syscripts|)
        (LETT |all| (SPADCALL (QVELT |sc| 2) |all| (QREFELT $ 78))
          |SYMBOL;syscripts|)
        (LETT |all| (SPADCALL (QVELT |sc| 1) |all| (QREFELT $ 78))
          |SYMBOL;syscripts|)
        (LETT |all| (SPADCALL (QVELT |sc| 0) |all| (QREFELT $ 78))
          |SYMBOL;syscripts|)
        (EXIT (SPADCALL |all| (QVELT |sc| 4) (QREFELT $ 78))))))

(DEFUN |SYMBOL;script;$L$;22| (|sy| |ls| $)
  (PROG (|sc|)
    (RETURN
      (SEQ (LETT |sc| (VECTOR NIL NIL NIL NIL NIL)
        |SYMBOL;script;$L$;22|)
        (COND
          ((NULL (NULL |ls|))
            (SEQ (QSETVELT |sc| 0 (|SPADfirst| |ls|))
              (EXIT (LETT |ls| (CDR |ls|) |SYMBOL;script;$L$;22|))))))
        (COND
          ((NULL (NULL |ls|))
            (SEQ (QSETVELT |sc| 1 (|SPADfirst| |ls|))
              (EXIT (LETT |ls| (CDR |ls|) |SYMBOL;script;$L$;22|))))))
        (COND
          ((NULL (NULL |ls|))
            (SEQ (QSETVELT |sc| 2 (|SPADfirst| |ls|))
              (EXIT (LETT |ls| (CDR |ls|) |SYMBOL;script;$L$;22|))))))
        (COND
          ((NULL (NULL |ls|))
            (SEQ (QSETVELT |sc| 3 (|SPADfirst| |ls|))
              (EXIT (LETT |ls| (CDR |ls|) |SYMBOL;script;$L$;22|))))))
        (COND
          ((NULL (NULL |ls|))
            (SEQ (QSETVELT |sc| 4 (|SPADfirst| |ls|))

```

```

        (EXIT (LETT |ls| (CDR |ls|) |SYMBOL;script;$L$;22|))))
    (EXIT (SPADCALL |sy| |sc| (QREFELT $ 80))))))

(DEFUN |SYMBOL;script;$R$;23| (|sy| |sc| $)
  (COND
    ((SPADCALL |sy| (QREFELT $ 21))
      (|error| "Cannot add scripts to a scripted symbol"))
    ('T
      (CONS (SPADCALL
        (SPADCALL
          (STRCONC (|SYMBOL;syprefix| |sc| $)
            (SPADCALL (SPADCALL |sy| (QREFELT $ 81))
              (QREFELT $ 82)))
          (QREFELT $ 47))
            (QREFELT $ 52))
        (|SYMBOL;syscripts| |sc| $))))))

(DEFUN |SYMBOL;string;$S;24| (|e| $)
  (COND
    ((NULL (SPADCALL |e| (QREFELT $ 21))) (PNAME |e|))
    ('T (|error| "Cannot form string from non-atomic symbols."))))

(DEFUN |SYMBOL;latex;$S;25| (|e| $)
  (PROG (|ss| |lo| |sc| |s|)
    (RETURN
      (SEQ (LETT |s| (PNAME (SPADCALL |e| (QREFELT $ 81)))
        |SYMBOL;latex;$S;25|)
        (COND
          ((< 1 (QCSIZE |s|))
            (COND
              ((NULL (SPADCALL (SPADCALL |s| 1 (QREFELT $ 83))
                (SPADCALL "\\\" (QREFELT $ 40))
                (QREFELT $ 84)))
                (LETT |s| (STRCONC "\\mbox{\\it \" (STRCONC |s| \"}")
                  |SYMBOL;latex;$S;25|))))))
            (COND ((NULL (SPADCALL |e| (QREFELT $ 21))) (EXIT |s|)))
            (LETT |ss| (SPADCALL |e| (QREFELT $ 85))
              |SYMBOL;latex;$S;25|)
            (LETT |lo| (QVELT |ss| 0) |SYMBOL;latex;$S;25|)
            (COND
              ((NULL (NULL |lo|))
                (SEQ (LETT |sc| \"_{\" |SYMBOL;latex;$S;25|)
                  (SEQ G190
                    (COND
                      ((NULL (COND ((NULL |lo|) 'NIL) ('T 'T)))
                        (GO G191)))
                    (SEQ (LETT |sc|
                      (STRCONC |sc|
                        (SPADCALL (|SPADfirst| |lo|)
                          (QREFELT $ 86)))

```

```

|SYMBOL;latex;$S;25|)
(LETT |lo| (CDR |lo|)
|SYMBOL;latex;$S;25|)
(EXIT (COND
((NULL (NULL |lo|))
(LETT |sc| (STRCONC |sc| ", ")
|SYMBOL;latex;$S;25|))))))
NIL (GO G190) G191 (EXIT NIL))
(LETT |sc| (STRCONC |sc| "}") |SYMBOL;latex;$S;25|)
(EXIT (LETT |s| (STRCONC |s| |sc|)
|SYMBOL;latex;$S;25|))))))
(LETT |lo| (QVELT |ss| 1) |SYMBOL;latex;$S;25|)
(COND
((NULL (NULL |lo|))
(SEQ (LETT |sc| "^{" |SYMBOL;latex;$S;25|)
(SEQ G190
(COND
((NULL (COND ((NULL |lo|) 'NIL) ('T 'T)))
(GO G191)))
(SEQ (LETT |sc|
(STRCONC |sc|
(SPADCALL (|SPADfirst| |lo|)
(QREFELT $ 86)))
|SYMBOL;latex;$S;25|)
(LETT |lo| (CDR |lo|)
|SYMBOL;latex;$S;25|)
(EXIT (COND
((NULL (NULL |lo|))
(LETT |sc| (STRCONC |sc| ", ")
|SYMBOL;latex;$S;25|))))))
NIL (GO G190) G191 (EXIT NIL))
(LETT |sc| (STRCONC |sc| "}") |SYMBOL;latex;$S;25|)
(EXIT (LETT |s| (STRCONC |s| |sc|)
|SYMBOL;latex;$S;25|))))))
(LETT |lo| (QVELT |ss| 2) |SYMBOL;latex;$S;25|)
(COND
((NULL (NULL |lo|))
(SEQ (LETT |sc| "{}^{" |SYMBOL;latex;$S;25|)
(SEQ G190
(COND
((NULL (COND ((NULL |lo|) 'NIL) ('T 'T)))
(GO G191)))
(SEQ (LETT |sc|
(STRCONC |sc|
(SPADCALL (|SPADfirst| |lo|)
(QREFELT $ 86)))
|SYMBOL;latex;$S;25|)
(LETT |lo| (CDR |lo|)
|SYMBOL;latex;$S;25|)
(EXIT (COND

```

```

((NULL (NULL |lo|))
 (LETT |sc| (STRCONC |sc| ", ")
 |SYMBOL;latex;$S;25|))))
NIL (GO G190) G191 (EXIT NIL))
(LETT |sc| (STRCONC |sc| "}") |SYMBOL;latex;$S;25|)
(EXIT (LETT |s| (STRCONC |sc| |s|)
 |SYMBOL;latex;$S;25|))))
(LETT |lo| (QVELT |ss| 3) |SYMBOL;latex;$S;25|)
(COND
 ((NULL (NULL |lo|))
 (SEQ (LETT |sc| "{}_{" |SYMBOL;latex;$S;25|)
 (SEQ G190
 (COND
 ((NULL (COND ((NULL |lo|) 'NIL) ('T 'T)))
 (GO G191)))
 (SEQ (LETT |sc|
 (STRCONC |sc|
 (SPADCALL (|SPADfirst| |lo|)
 (QREFELT $ 86)))
 |SYMBOL;latex;$S;25|)
 (LETT |lo| (CDR |lo|)
 |SYMBOL;latex;$S;25|)
 (EXIT (COND
 ((NULL (NULL |lo|))
 (LETT |sc| (STRCONC |sc| ", ")
 |SYMBOL;latex;$S;25|))))))
NIL (GO G190) G191 (EXIT NIL))
(LETT |sc| (STRCONC |sc| "}") |SYMBOL;latex;$S;25|)
(EXIT (LETT |s| (STRCONC |sc| |s|)
 |SYMBOL;latex;$S;25|))))
(LETT |lo| (QVELT |ss| 4) |SYMBOL;latex;$S;25|)
(COND
 ((NULL (NULL |lo|))
 (SEQ (LETT |sc| "\\left( {" |SYMBOL;latex;$S;25|)
 (SEQ G190
 (COND
 ((NULL (COND ((NULL |lo|) 'NIL) ('T 'T)))
 (GO G191)))
 (SEQ (LETT |sc|
 (STRCONC |sc|
 (SPADCALL (|SPADfirst| |lo|)
 (QREFELT $ 86)))
 |SYMBOL;latex;$S;25|)
 (LETT |lo| (CDR |lo|)
 |SYMBOL;latex;$S;25|)
 (EXIT (COND
 ((NULL (NULL |lo|))
 (LETT |sc| (STRCONC |sc| ", ")
 |SYMBOL;latex;$S;25|))))))
NIL (GO G190) G191 (EXIT NIL))

```



```

        (LETT |sc| (STRCONC |sc| "}" \\right)")
        |SYMBOL;latex;$S;25|)
    (EXIT (LETT |s| (STRCONC |s| |sc|)
            |SYMBOL;latex;$S;25|))))))
    (EXIT |s|))))))

(DEFUN |SYMBOL;anyRadix| (|n| |s| $)
  (PROG (|qr| |ns| G108274)
    (RETURN
      (SEQ (EXIT (SEQ (LETT |ns| "" |SYMBOL;anyRadix|)
                      (EXIT (SEQ G190 NIL
                                (SEQ (LETT |qr|
                                      (DIVIDE2 |n| (QCSIZE |s|))
                                      |SYMBOL;anyRadix|)
                                      (LETT |n| (QCAR |qr|)
                                      |SYMBOL;anyRadix|)
                                      (LETT |ns|
                                      (SPADCALL
                                        (SPADCALL |s|
                                          (+ (QCDR |qr|)
                                            (SPADCALL |s| (QREFELT $ 88)))
                                          (QREFELT $ 83))
                                        |ns| (QREFELT $ 89))
                                      |SYMBOL;anyRadix|)
                                      (EXIT
                                        (COND
                                          ((ZEROP |n|)
                                            (PROGN
                                              (LETT G108274 |ns|
                                              |SYMBOL;anyRadix|)
                                              (GO G108274))))))
                                      NIL (GO G190) G191 (EXIT NIL))))))
                      G108274 (EXIT G108274))))))

(DEFUN |SYMBOL;new;$;27| ($)
  (PROG (|sym|)
    (RETURN
      (SEQ (LETT |sym|
                (|SYMBOL;anyRadix|
                  (SPADCALL (QREFELT $ 9) (QREFELT $ 90))
                  (QREFELT $ 18) $)
                |SYMBOL;new;$;27|)
            (SPADCALL (QREFELT $ 9)
              (+ (SPADCALL (QREFELT $ 9) (QREFELT $ 90)) 1)
              (QREFELT $ 91))
            (EXIT (SPADCALL (STRCONC "%" |sym|) (QREFELT $ 47))))))

(DEFUN |SYMBOL;new;2$;28| (|x| $)
  (PROG (|u| |n| |xx|)
    (RETURN

```

```

(SEQ (LETT |n|
      (SEQ (LETT |u|
            (SPADCALL |x| (QREFELT $ 12)
                        (QREFELT $ 94))
            |SYMBOL;new;2$;28|)
            (EXIT (COND
                  ((QEQCAR |u| 1) 0)
                  ('T (+ (QCDR |u|) 1))))))
      |SYMBOL;new;2$;28|)
      (SPADCALL (QREFELT $ 12) |x| |n| (QREFELT $ 95))
      (LETT |xx|
            (COND
              ((NULL (SPADCALL |x| (QREFELT $ 21)))
               (SPADCALL |x| (QREFELT $ 82)))
              ('T
               (SPADCALL (SPADCALL |x| (QREFELT $ 81))
                         (QREFELT $ 82))))
            |SYMBOL;new;2$;28|)
      (LETT |xx| (STRCONC "%" |xx|) |SYMBOL;new;2$;28|)
      (LETT |xx|
            (COND
              ((NULL (< (SPADCALL
                        (SPADCALL |xx|
                          (SPADCALL |xx| (QREFELT $ 96))
                          (QREFELT $ 83))
                          (QREFELT $ 17) (QREFELT $ 97))
                        (SPADCALL (QREFELT $ 17) (QREFELT $ 88))))
               (STRCONC |xx|
                        (|SYMBOL;anyRadix| |n| (QREFELT $ 19) $)))
              ('T
               (STRCONC |xx|
                        (|SYMBOL;anyRadix| |n| (QREFELT $ 17) $))))
            |SYMBOL;new;2$;28|)
      (COND
        ((NULL (SPADCALL |x| (QREFELT $ 21)))
         (EXIT (SPADCALL |xx| (QREFELT $ 47))))
        (EXIT (SPADCALL (SPADCALL |xx| (QREFELT $ 47))
                        (SPADCALL |x| (QREFELT $ 85)) (QREFELT $ 80))))))

(DEFUN |SYMBOL;resetNew;V;29| ($)
  (PROG (|k| G108297)
    (RETURN
      (SEQ (SPADCALL (QREFELT $ 9) 0 (QREFELT $ 91))
            (SEQ (LETT |k| NIL |SYMBOL;resetNew;V;29|)
                  (LETT G108297 (SPADCALL (QREFELT $ 12) (QREFELT $ 100))
                        |SYMBOL;resetNew;V;29|)
                  G190
                  (COND
                    ((OR (ATOM G108297)
                        (PROGN

```

```

        (LETT |k| (CAR G108297)
          |SYMBOL;resetNew;V;29|)
      NIL))
    (GO G191)))
  (SEQ (EXIT (SPADCALL |k| (QREFELT $ 12)
    (QREFELT $ 101))))
  (LETT G108297 (CDR G108297) |SYMBOL;resetNew;V;29|)
  (GO G190) G191 (EXIT NIL))
  (EXIT (SPADCALL (QREFELT $ 102))))))

(DEFUN |SYMBOL;scripted?;$B;30| (|sy| $)
  (COND ((ATOM |sy|) 'NIL) ('T 'T)))

(DEFUN |SYMBOL;name;2$;31| (|sy| $)
  (PROG (|str| |i| G108304 G108303 G108301)
    (RETURN
      (SEQ (EXIT (COND
        ((NULL (SPADCALL |sy| (QREFELT $ 21))) |sy|)
        ('T
          (SEQ (LETT |str|
            (SPADCALL
              (SPADCALL
                (SPADCALL |sy| (QREFELT $ 104))
                (QREFELT $ 105))
                (QREFELT $ 82))
                |SYMBOL;name;2$;31|)
            (SEQ (EXIT (SEQ
              (LETT |i| (+ (QREFELT $ 38) 1)
                |SYMBOL;name;2$;31|)
              (LETT G108304 (QCSIZE |str|)
                |SYMBOL;name;2$;31|)
              G190
              (COND
                ((> |i| G108304) (GO G191)))
              (SEQ
                (EXIT
                  (COND
                    ((NULL
                      (SPADCALL
                        (SPADCALL |str| |i|
                          (QREFELT $ 83))
                          (QREFELT $ 106)))
                    (PROGN
                      (LETT G108301
                        (PROGN
                          (LETT G108303
                            (SPADCALL
                              (SPADCALL |str|
                                (SPADCALL |i|
                                  (QCSIZE |str|)

```

```

                                (QREFELT $ 108))
                                (QREFELT $ 109))
                                (QREFELT $ 47))
                                |SYMBOL;name;2$;31|)
                                (GO G108303))
                                |SYMBOL;name;2$;31|)
                                (GO G108301))))))
                                (LETT |i| (+ |i| 1)
                                |SYMBOL;name;2$;31|)
                                (GO G190) G191 (EXIT NIL)))
                                G108301 (EXIT G108301))
                                (EXIT (|error| "Improper scripted symbol")))))))
G108303 (EXIT G108303))))))

(DEFUN |SYMBOL;scripts;$R;32| (|sy| $)
  (PROG (|lscripts| |str| |nstr| |j| G108307 |nscripts| |m| |n| G108316
    |i| G108317 |a| G108318 |allscripts|)
    (RETURN
      (SEQ (COND
        ((NULL (SPADCALL |sy| (QREFELT $ 21)))
          (VECTOR NIL NIL NIL NIL NIL))
        ('T
          (SEQ (LETT |nscripts| (LIST 0 0 0 0 0)
            |SYMBOL;scripts;$R;32|)
            (LETT |lscripts| (LIST NIL NIL NIL NIL NIL)
              |SYMBOL;scripts;$R;32|)
            (LETT |str|
              (SPADCALL
                (SPADCALL (SPADCALL |sy| (QREFELT $ 104))
                  (QREFELT $ 105))
                  (QREFELT $ 82))
                |SYMBOL;scripts;$R;32|)
            (LETT |nstr| (QCSIZE |str|) |SYMBOL;scripts;$R;32|)
            (LETT |m| (SPADCALL |nscripts| (QREFELT $ 111))
              |SYMBOL;scripts;$R;32|)
            (SEQ (LETT |j| (+ (QREFELT $ 38) 1)
              |SYMBOL;scripts;$R;32|)
              (LETT |i| |m| |SYMBOL;scripts;$R;32|) G190
              (COND
                ((OR (> |j| |nstr|)
                  (NULL (SPADCALL
                    (SPADCALL |str| |j|
                      (QREFELT $ 83))
                      (QREFELT $ 106))))))
                (GO G191)))
            (SEQ (EXIT (SPADCALL |nscripts| |i|
              (PROG1
                (LETT G108307
                  (-
                    (SPADCALL

```

```

        (SPADCALL |str| |j|
         (QREFELT $ 83))
        (QREFELT $ 41))
        (QREFELT $ 42))
        |SYMBOL;scripts;$R;32|)
        (|check-subtype| (>= G108307 0)
         '(|NonNegativeInteger|) G108307))
        (QREFELT $ 113))))
(LETT |i|
  (PROG1 (+ |i| 1)
    (LETT |j| (+ |j| 1)
      |SYMBOL;scripts;$R;32|))
    |SYMBOL;scripts;$R;32|)
  (GO G190) G191 (EXIT NIL))
(LETT |nscripts|
  (SPADCALL (CDR |nscripts|)
    (|SPADfirst| |nscripts|) (QREFELT $ 114))
    |SYMBOL;scripts;$R;32|)
(LETT |allscripts|
  (SPADCALL (SPADCALL |sy| (QREFELT $ 104))
    (QREFELT $ 115))
    |SYMBOL;scripts;$R;32|)
(LETT |m| (SPADCALL |lscripts| (QREFELT $ 116))
  |SYMBOL;scripts;$R;32|)
(SEQ (LETT |n| NIL |SYMBOL;scripts;$R;32|)
  (LETT G108316 |nscripts|
    |SYMBOL;scripts;$R;32|)
  (LETT |i| |m| |SYMBOL;scripts;$R;32|) G190
  (COND
    ((OR (ATOM G108316)
      (PROGN
        (LETT |n| (CAR G108316)
          |SYMBOL;scripts;$R;32|)
        NIL))
      (GO G191)))
  (SEQ (EXIT (COND
    ((<
      (SPADCALL |allscripts|
        (QREFELT $ 117))
      |n|)
      (|error|
        "Improper script count in symbol"))
    ('T
      (SEQ
        (SPADCALL |lscripts| |i|
          (PROGN
            (LETT G108317 NIL
              |SYMBOL;scripts;$R;32|)
            (SEQ
              (LETT |a| NIL

```

```

|SYMBOL;scripts;$R;32|)
(LETT G108318
  (SPADCALL |allscripts| |n|
    (QREFELT $ 118))
  |SYMBOL;scripts;$R;32|)
G190
(COND
  ((OR (ATOM G108318)
    (PROGN
      (LETT |a|
        (CAR G108318)
        |SYMBOL;scripts;$R;32|)
        NIL))
      (GO G191)))
  (SEQ
    (EXIT
      (LETT G108317
        (CONS
          (SPADCALL |a|
            (QREFELT $ 52))
            G108317)
          |SYMBOL;scripts;$R;32|)))
    (LETT G108318 (CDR G108318)
      |SYMBOL;scripts;$R;32|)
    (GO G190) G191
    (EXIT (NREVERSEO G108317))))
(QREFELT $ 119))
(EXIT
  (LETT |allscripts|
    (SPADCALL |allscripts| |n|
      (QREFELT $ 120))
    |SYMBOL;scripts;$R;32|))))))
(LETT |i|
  (PROG1 (+ |i| 1)
    (LETT G108316 (CDR G108316)
      |SYMBOL;scripts;$R;32|)
      |SYMBOL;scripts;$R;32|)
    (GO G190) G191 (EXIT NIL))
  (EXIT (VECTOR (SPADCALL |lscripts| |m|
    (QREFELT $ 121))
    (SPADCALL |lscripts| (+ |m| 1)
      (QREFELT $ 121))
    (SPADCALL |lscripts| (+ |m| 2)
      (QREFELT $ 121))
    (SPADCALL |lscripts| (+ |m| 3)
      (QREFELT $ 121))
    (SPADCALL |lscripts| (+ |m| 4)
      (QREFELT $ 121))))))))))

(DEFUN |SYMBOL;istring| (|n| $)

```

```

(COND
  ((< 9 |n|) (|error| "Can have at most 9 scripts of each kind"))
  ('T (ELT (QREFELT $ 16) (+ |n| 0))))))

(DEFUN |SYMBOL;list;$L;34| (|sy| $)
  (COND
    ((NULL (SPADCALL |sy| (QREFELT $ 21)))
      (|error| "Cannot convert a symbol to a list if it is not subscripted"))
    ('T |sy|)))

(DEFUN |SYMBOL;sample;$;35| (|$|) (SPADCALL "aSymbol" (QREFELT |$| 47)))

(DEFUN |Symbol| ()
  (PROG ()
    (RETURN
      (PROG (G108325)
        (RETURN
          (COND
            ((LETT G108325 (HGET |$ConstructorCache| '|Symbol|)
              |Symbol|)
              (|CDRwithIncrement| (CDAR G108325)))
            ('T
              (UNWIND-PROTECT
                (PROG1 (CDDAR (HPUT |$ConstructorCache| '|Symbol|
                  (LIST
                    (CONS NIL (CONS 1 (|Symbol;|))))))
                  (LETT G108325 T |Symbol|)))
                (COND
                  ((NOT G108325) (HREM |$ConstructorCache| '|Symbol|))))))))))

(DEFUN |Symbol;| ()
  (PROG (|dv$| $ |pv$|)
    (RETURN
      (PROGN
        (LETT |dv$| '(|Symbol|) |Symbol|)
        (LETT $ (make-array 124) |Symbol|)
        (QSETREFV $ 0 |dv$|)
        (QSETREFV $ 3
          (LETT |pv$| (|buildPredVector| 0 0 NIL) |Symbol|))
        (|haddProp| |$ConstructorCache| '|Symbol| NIL (CONS 1 $))
        (|stuffDomainSlots| $)
        (QSETREFV $ 9 (SPADCALL 0 (QREFELT $ 8)))
        (QSETREFV $ 12 (SPADCALL (QREFELT $ 11)))
        (QSETREFV $ 16
          (SPADCALL (LIST "0" "1" "2" "3" "4" "5" "6" "7" "8" "9")
            (QREFELT $ 15)))
        (QSETREFV $ 17 "0123456789")
        (QSETREFV $ 18 "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
        (QSETREFV $ 19 "abcdefghijklmnopqrstuvwxyz")
        (QSETREFV $ 37 "*"))
      )
    )
  )

```

```

(QSETREFV $ 38 (QCSIZE (QREFELT $ 37)))
(QSETREFV $ 42
  (SPADCALL (SPADCALL "0" (QREFELT $ 40)) (QREFELT $ 41)))
$)))

(setf (get 'Symbol 'infovec)
  (LIST '#(NIL NIL NIL NIL NIL NIL (Integer) (Reference) 6)
    (0 . |ref|) 'count| (AssociationList) $$ 6)
    (5 . |empty|) 'xcount| (List) 28) (PrimitiveArray) 28)
    (9 . |construct|) 'listings| 'nums| 'ALPHAS 'alphas|
    (Boolean) |SYMBOL;scripted?;$B;30| (Void) (Symbol)
    (OpenMathDevice) (14 . |OMputVariable|)
    (OpenMathEncoding) (20 . |OMencodingXML|) (String)
    (24 . |OMopenString|) (30 . |OMputObject|)
    (35 . |OMputEndObject|) (40 . |OMclose|)
    |SYMBOL;OMwrite;$S;2| |SYMBOL;OMwrite;$BS;3|
    |SYMBOL;OMwrite;Omd$V;4| |SYMBOL;OMwrite;Omd$BV;5| 'lhd|
    'lhd| (Character) (45 . |char|) (50 . |ord|) 'ord0|
    (InputForm) (55 . |convert|) |SYMBOL;convert;$If;6|
    |SYMBOL;convert;2$;7| |SYMBOL;coerce;$S;8|
    |SYMBOL;=;2$B;9| |SYMBOL;<;2$B;10| (OutputForm)
    (60 . |outputForm|) |SYMBOL;coerce;$Of;11| (List) 55)
    |SYMBOL;script;$L$;22| (List) 50)
    |SYMBOL;subscript;$L$;12| |SYMBOL;elt;$L$;13|
    |SYMBOL;superscript;$L$;14| |SYMBOL;argscript;$L$;15|
    (PatternMatchResult) 6 23) (Pattern) 6)
    (PatternMatchSymbol) 6) (65 . |patternMatch|)
    (PatternMatchResult) 6 $) |SYMBOL;patternMatch;$P2Pmr;16|
    (PatternMatchResult) (Float) 23) (Pattern) (Float))
    (PatternMatchSymbol) (Float)) (72 . |patternMatch|)
    (PatternMatchResult) (Float) $)
    |SYMBOL;patternMatch;$P2Pmr;17| (79 . |coerce|)
    |SYMBOL;convert;$P;18| (84 . |coerce|)
    |SYMBOL;convert;$P;19| (List) $) (89 . |concat|)
    (94 . |concat|)
    (Record) (|:| |sub| 55) (|:| |sup| 55) (|:| |presup| 55)
      (|:| |presub| 55) (|:| |larg| 55))
    |SYMBOL;script;$R$;23| |SYMBOL;name;2$;31|
    |SYMBOL;string;$S;24| (100 . |elt|) (106 . =)
    |SYMBOL;scripts;$R;32| (112 . |latex|)
    |SYMBOL;latex;$S;25| (117 . |minIndex|) (122 . |concat|)
    (128 . |elt|) (133 . |setelt|) |SYMBOL;new;$;27|
    (Union) 6 "failed") (139 . |search|) (145 . |setelt|)
    (152 . |maxIndex|) (157 . |position|) |SYMBOL;new;2$;28|
    (List) $$) (163 . |keys|) (168 . |remove!|)
    (174 . |void|) |SYMBOL;resetNew;V;29| |SYMBOL;list;$L;34|
    (178 . |first|) (183 . |digit?|) (UniversalSegment) 6)
    (188 . SEGMENT) (194 . |elt|) (List) 112)
    (200 . |minIndex|) (NonNegativeInteger) (205 . |setelt|)
    (212 . |concat|) (218 . |rest|) (223 . |minIndex|)

```



```

(228 . |#|) (233 . |first|) (239 . |setelt|)
(246 . |rest|) (252 . |elt|)
(CONS IDENTITY
  (FUNCALL (|dispatchFunction| |SYMBOL;sample;$;35|)
    $))
(|SingleInteger|))
'#(~= 258 |superscript| 264 |subscript| 270 |string| 276
|scripts| 281 |scripted?| 286 |script| 291 |sample| 303
|resetNew| 307 |patternMatch| 311 |new| 325 |name| 334
|min| 339 |max| 345 |list| 351 |latex| 356 |hash| 361
|elt| 366 |convert| 372 |coerce| 392 |largscript| 402
|OMwrite| 408 >= 432 > 438 = 444 <= 450 < 456)
'NIL
(CONS (|makeByteWordVec2| 1 '(0 0 0 0 0 0 0 0 0 0))
  (CONS '#(|OrderedSet&| NIL NIL |SetCategory&|
    |BasicType&| NIL NIL NIL NIL NIL NIL)
    (CONS '#(|OrderedSet|)
      (|PatternMatchable| (|Float|))
      (|PatternMatchable| 6) (|SetCategory|)
      (|BasicType|) (|ConvertibleTo| 67)
      (|ConvertibleTo| 61)
      (|ConvertibleTo| 23) (|OpenMath|)
      (|ConvertibleTo| 43) (|CoercibleTo| 50))
      (|makeByteWordVec2| 123
        '(1 7 0 6 8 0 10 0 11 1 14 0 13 15 2 24
          22 0 23 25 0 26 0 27 2 24 0 28 26 29
          1 24 22 0 30 1 24 22 0 31 1 24 22 0
          32 1 39 0 28 40 1 39 6 0 41 1 43 0 23
          44 1 50 0 23 51 3 62 60 23 61 60 63 3
          68 66 23 67 66 69 1 67 0 23 72 1 61 0
          23 74 1 28 0 76 77 2 55 0 0 0 78 2 28
          39 0 6 83 2 39 20 0 0 84 1 50 28 0 86
          1 28 6 0 88 2 28 0 39 0 89 1 7 6 0 90
          2 7 6 0 6 91 2 10 93 2 0 94 3 10 6 0
          2 6 95 1 28 6 0 96 2 28 6 39 0 97 1
          10 99 0 100 2 10 93 2 0 101 0 22 0
          102 1 99 2 0 105 1 39 20 0 106 2 107
          0 6 6 108 2 28 0 0 107 109 1 110 6 0
          111 3 110 112 0 6 112 113 2 110 0 0
          112 114 1 99 0 0 115 1 53 6 0 116 1
          99 112 0 117 2 99 0 0 112 118 3 53 55
          0 6 55 119 2 99 0 0 112 120 2 53 55 0
          6 121 2 0 20 0 0 1 2 0 0 0 55 58 2 0
          0 0 55 56 1 0 28 0 82 1 0 79 0 85 1 0
          20 0 21 2 0 0 0 53 54 2 0 0 0 79 80 0
          0 0 122 0 0 22 103 3 0 64 0 61 64 65
          3 0 70 0 67 70 71 1 0 0 0 98 0 0 0 92
          1 0 0 0 81 2 0 0 0 0 1 2 0 0 0 0 1 1
          0 76 0 104 1 0 28 0 87 1 0 123 0 1 2
          0 0 0 55 57 1 0 61 0 75 1 0 67 0 73 1

```

```

                                0 23 0 46 1 0 43 0 45 1 0 0 28 47 1 0
                                50 0 52 2 0 0 0 55 59 3 0 22 24 0 20
                                36 2 0 28 0 20 34 2 0 22 24 0 35 1 0
                                28 0 33 2 0 20 0 0 1 2 0 20 0 0 1 2 0
                                20 0 0 48 2 0 20 0 0 1 2 0 20 0 0 49))))))
      '|lookupComplete|))

(setf (get '|Symbol| 'NILADIC) T)

```

VECTOR.lsp BOOTSTRAP

VECTOR depends on itself. We need to break this cycle to build the algebra. So we keep a cached copy of the translated **VECTOR** category which we can write into the **MID** directory. We compile the lisp code and copy the **VECTOR.o** file to the **OUT** directory. This is eventually forcibly replaced by a recompiled version.

Note that this code is not included in the generated catdef.spad file.

— VECTOR.lsp BOOTSTRAP —

```

(|/VERSIONCHECK| 2)

(DEFUN |VECTOR;vector;L$;1| (|1| |$|) (SPADCALL |1| (QREFELT |$| 8)))

(DEFUN |VECTOR;convert;$If;2| (|x| |$|)
  (SPADCALL
    (LIST
      (SPADCALL (SPADCALL "vector" (QREFELT |$| 12)) (QREFELT |$| 14))
      (SPADCALL (SPADCALL |x| (QREFELT |$| 15)) (QREFELT |$| 16)))
    (QREFELT |$| 18)))

(DEFUN |Vector| (G84134)
  (PROG ()
    (RETURN
      (PROG (G84135)
        (RETURN
          (COND
            ((LETT G84135
              (|assocShiftWithFunction|
                (LIST (|devaluate| G84134))
                (HGET |$ConstructorCache| '|Vector|)
                '|domainEqualList|)
              |Vector|)
            (|CDRwithIncrement| G84135)))
        ('T

```

```

      (UNWIND-PROTECT
        (PROG1 (|Vector;| G84134) (LETT G84135 T |Vector|))
        (COND
          ((NOT G84135) (HREM |$ConstructorCache| '|Vector|)))))))))

(DEFUN |Vector;| (|#1|)
  (PROG (DV$1 |dv$| $ G84133 |pv$|)
    (RETURN
      (PROGN
        (LETT DV$1 (|devaluate| |#1|) |Vector|)
        (LETT |dv$| (LIST '|Vector| DV$1) |Vector|)
        (LETT $ (make-array 36) |Vector|)
        (QSETREFV $ 0 |dv$|)
        (QSETREFV $ 3
          (LETT |pv$|
            (|buildPredVector| 0 0
              (LIST (|HasCategory| |#1| '|SetCategory|))
                (|HasCategory| |#1|
                  '|ConvertibleTo| (|InputForm|)))
              (LETT G84133
                (|HasCategory| |#1| '|OrderedSet|))
                |Vector|)
              (OR G84133
                (|HasCategory| |#1| '|SetCategory|))
                (|HasCategory| (|Integer|) '|OrderedSet|))
                (|HasCategory| |#1| '|AbelianSemiGroup|))
                (|HasCategory| |#1| '|AbelianMonoid|))
                (|HasCategory| |#1| '|AbelianGroup|))
                (|HasCategory| |#1| '|Monoid|))
                (|HasCategory| |#1| '|Ring|))
                (AND (|HasCategory| |#1|
                  '|RadicalCategory|))
                  (|HasCategory| |#1| '|Ring|)))
              (AND (|HasCategory| |#1|
                (LIST '|Evalable|
                  (|devaluate| |#1|)))
                  (|HasCategory| |#1| '|SetCategory|)))
              (OR (AND (|HasCategory| |#1|
                (LIST '|Evalable|
                  (|devaluate| |#1|)))
                  G84133)
                (AND (|HasCategory| |#1|
                  (LIST '|Evalable|
                    (|devaluate| |#1|)))
                    (|HasCategory| |#1|
                      '|SetCategory|))))))
                |Vector|))
          (|haddProp| |$ConstructorCache| '|Vector| (LIST DV$1)
            (CONS 1 $))
          (|stuffDomainSlots| $)

```

```

(QSETREFV $ 6 |#1|)
(COND
  ((|testBitVector| |pv$| 2)
    (QSETREFV $ 19
      (CONS (|dispatchFunction| |VECTOR;convert;$If;2|) $))))
  $))))

(setf (get
  (QUOTE |Vector|)
  (QUOTE |infovec|))
  (LIST
    (QUOTE #(NIL NIL NIL NIL NIL (|IndexedVector| 6 (NRTEVAL 1)) (|local| |#1|)
      (|List| 6) (0 . |construct|) |VECTOR;vector;L$;1| (|String|) (|Symbol|)
      (5 . |coerce|) (|InputForm|) (10 . |convert|) (15 . |parts|)
      (20 . |convert|) (|List| |$|) (25 . |convert|) (30 . |convert|)
      (|Mapping| 6 6 6) (|Boolean|) (|NonNegativeInteger|) (|List| 24)
      (|Equation| 6) (|Integer|) (|Mapping| 21 6) (|Mapping| 21 6 6)
      (|UniversalSegment| 25) (|Void|) (|Mapping| 6 6) (|Matrix| 6)
      (|OutputForm|) (|SingleInteger|) (|Union| 6 (QUOTE "failed"))
      (|List| 25)))
    (QUOTE #(|vector| 35 |parts| 40 |convert| 45 |construct| 50))
    (QUOTE ((|shallowlyMutable| . 0) (|finiteAggregate| . 0)))
    (CONS
      (|makeByteWordVec2| 13 (QUOTE (0 0 0 0 0 0 3 0 0 13 4 0 0 13 1 2 4)))
      (CONS
        (QUOTE #(|VectorCategory&| |OneDimensionalArrayAggregate&|
          |FiniteLinearAggregate&| |LinearAggregate&| |IndexedAggregate&|
          |Collection&| |HomogeneousAggregate&| |OrderedSet&| |Aggregate&|
          |EltableAggregate&| |Evalable&| |SetCategory&| NIL NIL
          |InnerEvalable&| NIL NIL |BasicType&|))
        (CONS
          (QUOTE #((|VectorCategory| 6) (|OneDimensionalArrayAggregate| 6)
            (|FiniteLinearAggregate| 6) (|LinearAggregate| 6)
            (|IndexedAggregate| 25 6) (|Collection| 6)
            (|HomogeneousAggregate| 6) (|OrderedSet|) (|Aggregate|)
            (|EltableAggregate| 25 6) (|Evalable| 6) (|SetCategory|)
            (|Type|) (|Eltable| 25 6) (|InnerEvalable| 6 6)
            (|CoercibleTo| 32) (|ConvertibleTo| 13) (|BasicType|)))
          (|makeByteWordVec2| 19
            (QUOTE (1 0 0 7 8 1 11 0 10 12 1 13 0 11 14 1 0 7 0 15 1 7 13 0 16 1 13
              0 17 18 1 0 13 0 19 1 0 0 7 9 1 0 7 0 15 1 2 13 0 19 1 0 0 7 8))))))
    (QUOTE |lookupIncomplete|)))

```

Chapter 29

Chunk collections

— algebra —

```
\getchunk{domain AFFPL AffinePlane}  
\getchunk{domain AFFPLPS AffinePlaneOverPseudoAlgebraicClosureOfFiniteField}  
\getchunk{domain AFFSP AffineSpace}  
\getchunk{domain ALGSC AlgebraGivenByStructuralConstants}  
\getchunk{domain ALGFF AlgebraicFunctionField}  
\getchunk{domain AN AlgebraicNumber}  
\getchunk{domain ANON AnonymousFunction}  
\getchunk{domain ANTISYM AntiSymm}  
\getchunk{domain ANY Any}  
\getchunk{domain ASTACK ArrayStack}  
\getchunk{domain ASP1 Asp1}  
\getchunk{domain ASP10 Asp10}  
\getchunk{domain ASP12 Asp12}  
\getchunk{domain ASP19 Asp19}  
\getchunk{domain ASP20 Asp20}  
\getchunk{domain ASP24 Asp24}  
\getchunk{domain ASP27 Asp27}  
\getchunk{domain ASP28 Asp28}  
\getchunk{domain ASP29 Asp29}  
\getchunk{domain ASP30 Asp30}  
\getchunk{domain ASP31 Asp31}  
\getchunk{domain ASP33 Asp33}  
\getchunk{domain ASP34 Asp34}  
\getchunk{domain ASP35 Asp35}  
\getchunk{domain ASP4 Asp4}  
\getchunk{domain ASP41 Asp41}  
\getchunk{domain ASP42 Asp42}  
\getchunk{domain ASP49 Asp49}  
\getchunk{domain ASP50 Asp50}
```

```

\getchunk{domain ASP55 Asp55}
\getchunk{domain ASP6 Asp6}
\getchunk{domain ASP7 Asp7}
\getchunk{domain ASP73 Asp73}
\getchunk{domain ASP74 Asp74}
\getchunk{domain ASP77 Asp77}
\getchunk{domain ASP78 Asp78}
\getchunk{domain ASP8 Asp8}
\getchunk{domain ASP80 Asp80}
\getchunk{domain ASP9 Asp9}
\getchunk{domain JORDAN AssociatedJordanAlgebra}
\getchunk{domain LIE AssociatedLieAlgebra}
\getchunk{domain ALIST AssociationList}
\getchunk{domain ATTRIBUT AttributeButtons}
\getchunk{domain AUTOMOR Automorphism}

\getchunk{domain BBTREE BalancedBinaryTree}
\getchunk{domain BPADIC BalancedPAdicInteger}
\getchunk{domain BPADICRT BalancedPAdicRational}
\getchunk{domain BFUNCT BasicFunctions}
\getchunk{domain BOP BasicOperator}
\getchunk{domain BINARY BinaryExpansion}
\getchunk{domain BINFILE BinaryFile}
\getchunk{domain BSTREE BinarySearchTree}
\getchunk{domain BTOURN BinaryTournament}
\getchunk{domain BTREE BinaryTree}
\getchunk{domain BITS Bits}
\getchunk{domain BLHN BlowUpWithHamburgerNoether}
\getchunk{domain BLQT BlowUpWithQuadTrans}
\getchunk{domain BOOLEAN Boolean}

\getchunk{domain CARD CardinalNumber}
\getchunk{domain CARTEN CartesianTensor}
\getchunk{domain CHAR Character}
\getchunk{domain CCLASS CharacterClass}
\getchunk{domain CLIF CliffordAlgebra}
\getchunk{domain COLOR Color}
\getchunk{domain COMM Commutator}
\getchunk{domain COMPLEX Complex}
\getchunk{domain CDFMAT ComplexDoubleFloatMatrix}
\getchunk{domain CDFVEC ComplexDoubleFloatVector}
\getchunk{domain CONTFRAC ContinuedFraction}

\getchunk{domain DHMATRIX DenavitHartenbergMatrix}
\getchunk{domain DBASE Database}
\getchunk{domain DLIST DataList}
\getchunk{domain DECIMAL DecimalExpansion}
\getchunk{domain DEQUEUE Dequeue}
\getchunk{domain DERHAM DeRhamComplex}
\getchunk{domain DSMP DifferentialSparseMultivariatePolynomial}

```

```

\getchunk{domain DIRPROD DirectProduct}
\getchunk{domain DPMM DirectProductMatrixModule}
\getchunk{domain DPMO DirectProductModule}
\getchunk{domain DIRRING DirichletRing}
\getchunk{domain DMP DistributedMultivariatePolynomial}
\getchunk{domain DIV Divisor}
\getchunk{domain DFLOAT DoubleFloat}
\getchunk{domain DFMAT DoubleFloatMatrix}
\getchunk{domain DFVEC DoubleFloatVector}
\getchunk{domain DROPT DrawOption}
\getchunk{domain D01AJFA d01ajfAnnaType}
\getchunk{domain D01AKFA d01akfAnnaType}
\getchunk{domain D01ALFA d01alfAnnaType}
\getchunk{domain D01AMFA d01amfAnnaType}
\getchunk{domain D01ANFA d01anfAnnaType}
\getchunk{domain D01APFA d01apfAnnaType}
\getchunk{domain D01AQFA d01aqfAnnaType}
\getchunk{domain D01ASFA d01asfAnnaType}
\getchunk{domain D01FCFA d01fcfAnnaType}
\getchunk{domain D01GBFA d01gbfAnnaType}
\getchunk{domain D01TRNS d01TransformFunctionType}
\getchunk{domain D02BBFA d02bbfAnnaType}
\getchunk{domain D02BHFA d02bhfAnnaType}
\getchunk{domain D02CJFA d02cjfAnnaType}
\getchunk{domain D02EJFA d02ejfAnnaType}
\getchunk{domain D03EEFA d03eefAnnaType}
\getchunk{domain D03FAFA d03fafAnnaType}

\getchunk{domain EFULS ElementaryFunctionsUnivariateLaurentSeries}
\getchunk{domain EFUPXS ElementaryFunctionsUnivariatePuisseuxSeries}
\getchunk{domain EQTBL EqTable}
\getchunk{domain EQ Equation}
\getchunk{domain EXPEXPAN ExponentialExpansion}
\getchunk{domain EXPUPXS ExponentialOfUnivariatePuisseuxSeries}
\getchunk{domain EMR EuclideanModularRing}
\getchunk{domain EXIT Exit}
\getchunk{domain EXPR Expression}
\getchunk{domain EAB ExtAlgBasis}
\getchunk{domain E04DGFA e04dgmAnnaType}
\getchunk{domain E04FDFA e04fdfAnnaType}
\getchunk{domain E04GCFA e04gcfAnnaType}
\getchunk{domain E04JFAFA e04jafAnnaType}
\getchunk{domain E04MBFA e04mbfAnnaType}
\getchunk{domain E04NAFA e04nafAnnaType}
\getchunk{domain E04UCFA e04ucfAnnaType}

\getchunk{domain FR Factored}
\getchunk{domain FILE File}
\getchunk{domain FNAME FileName}
\getchunk{domain FARRAY FlexibleArray}

```



```

\getchunk{domain FDIV FiniteDivisor}
\getchunk{domain FF FiniteField}
\getchunk{domain FFCG FiniteFieldCyclicGroup}
\getchunk{domain FFCGX FiniteFieldCyclicGroupExtension}
\getchunk{domain FFCGP FiniteFieldCyclicGroupExtensionByPolynomial}
\getchunk{domain FFX FiniteFieldExtension}
\getchunk{domain FFP FiniteFieldExtensionByPolynomial}
\getchunk{domain FFNB FiniteFieldNormalBasis}
\getchunk{domain FFNBX FiniteFieldNormalBasisExtension}
\getchunk{domain FFNBP FiniteFieldNormalBasisExtensionByPolynomial}
\getchunk{domain FLOAT Float}
\getchunk{domain FC FortranCode}
\getchunk{domain FEXPR FortranExpression}
\getchunk{domain FORTRAN FortranProgram}
\getchunk{domain FST FortranScalarType}
\getchunk{domain FTEM FortranTemplate}
\getchunk{domain FT FortranType}
\getchunk{domain FCOMP FourierComponent}
\getchunk{domain FSERIES FourierSeries}
\getchunk{domain FRAC Fraction}
\getchunk{domain FRIDEAL FractionalIdeal}
\getchunk{domain FRMOD FramedModule}
\getchunk{domain FAGROUP FreeAbelianGroup}
\getchunk{domain FAMONOID FreeAbelianMonoid}
\getchunk{domain FGROUPO FreeGroup}
\getchunk{domain FM FreeModule}
\getchunk{domain FM1 FreeModule1}
\getchunk{domain FMONOID FreeMonoid}
\getchunk{domain FNLA FreeNilpotentLie}
\getchunk{domain FPARFRAC FullPartialFractionExpansion}
\getchunk{domain FUNCTION FunctionCalled}

\getchunk{domain GDMP GeneralDistributedMultivariatePolynomial}
\getchunk{domain GMODPOL GeneralModulePolynomial}
\getchunk{domain GCNAALG GenericNonAssociativeAlgebra}
\getchunk{domain GPOLSET GeneralPolynomialSet}
\getchunk{domain GSTBL GeneralSparseTable}
\getchunk{domain GTSET GeneralTriangularSet}
\getchunk{domain GSERIES GeneralUnivariatePowerSeries}
\getchunk{domain GRIMAGE GraphImage}
\getchunk{domain GOPT GuessOption}
\getchunk{domain GOPTO GuessOptionFunctionsO}

\getchunk{domain HASHTBL HashTable}
\getchunk{domain HEAP Heap}
\getchunk{domain HEXADEC HexadecimalExpansion}
\getchunk{domain HTMLFORM HTMLFormat}
\getchunk{domain HDP HomogeneousDirectProduct}
\getchunk{domain HDMP HomogeneousDistributedMultivariatePolynomial}
\getchunk{domain HELLDIV HyperellipticFiniteDivisor}

```

```

\getchunk{domain ICP InfClsPt}
\getchunk{domain ICARD IndexCard}
\getchunk{domain IBITS IndexedBits}
\getchunk{domain IDPAG IndexedDirectProductAbelianGroup}
\getchunk{domain IDPAM IndexedDirectProductAbelianMonoid}
\getchunk{domain IDPO IndexedDirectProductObject}
\getchunk{domain IDPOAM IndexedDirectProductOrderedAbelianMonoid}
\getchunk{domain IDPOAMS IndexedDirectProductOrderedAbelianMonoidSup}
\getchunk{domain INDE IndexedExponents}
\getchunk{domain IFARRAY IndexedFlexibleArray}
\getchunk{domain ILIST IndexedList}
\getchunk{domain IMATRIX IndexedMatrix}
\getchunk{domain IARRAY1 IndexedOneDimensionalArray}
\getchunk{domain ISTRING IndexedString}
\getchunk{domain IARRAY2 IndexedTwoDimensionalArray}
\getchunk{domain IVECTOR IndexedVector}
\getchunk{domain ITUPLE InfiniteTuple}
\getchunk{domain INFCLSPT InfinitelyClosePoint}
\getchunk{domain INFCLSPS InfinitelyClosePointOverPseudoAlgebraicClosureOfFiniteField}
\getchunk{domain IAN InnerAlgebraicNumber}
\getchunk{domain IFF InnerFiniteField}
\getchunk{domain IFAMON InnerFreeAbelianMonoid}
\getchunk{domain IIARRAY2 InnerIndexedTwoDimensionalArray}
\getchunk{domain IPADIC InnerPAdicInteger}
\getchunk{domain IPF InnerPrimeField}
\getchunk{domain ISUPS InnerSparseUnivariatePowerSeries}
\getchunk{domain INTABL InnerTable}
\getchunk{domain ITAYLOR InnerTaylorSeries}
\getchunk{domain INFORM InputForm}
\getchunk{domain INT Integer}
\getchunk{domain ZMOD IntegerMod}
\getchunk{domain INTFTBL IntegrationFunctionsTable}
\getchunk{domain IR IntegrationResult}
\getchunk{domain INTRVL Interval}

\getchunk{domain KERNEL Kernel}
\getchunk{domain KAFILE KeyedAccessFile}

\getchunk{domain LAUPOL LaurentPolynomial}
\getchunk{domain LIB Library}
\getchunk{domain LEXP LieExponentials}
\getchunk{domain LPOLY LiePolynomial}
\getchunk{domain LSQM LieSquareMatrix}
\getchunk{domain LOD0 LinearOrdinaryDifferentialOperator}
\getchunk{domain LOD01 LinearOrdinaryDifferentialOperator1}
\getchunk{domain LOD02 LinearOrdinaryDifferentialOperator2}
\getchunk{domain LIST List}
\getchunk{domain LMOPS ListMonoidOps}
\getchunk{domain LMDICT ListMultiDictionary}

```

```

\getchunk{domain LA LocalAlgebra}
\getchunk{domain LO Localize}
\getchunk{domain LWORD LyndonWord}

\getchunk{domain MCMPLX MachineComplex}
\getchunk{domain MFLOAT MachineFloat}
\getchunk{domain MINT MachineInteger}
\getchunk{domain MAGMA Magma}
\getchunk{domain MKCHSET MakeCachableSet}
\getchunk{domain MMLFORM MathMLFormat}
\getchunk{domain MATRIX Matrix}
\getchunk{domain MODMON ModMonic}
\getchunk{domain MODMONOM ModuleMonomial}
\getchunk{domain MODFIELD ModularField}
\getchunk{domain MODRING ModularRing}
\getchunk{domain MODOP ModuleOperator}
\getchunk{domain MOEBIUS MoebiusTransform}
\getchunk{domain MRING MonoidRing}
\getchunk{domain MSET Multiset}
\getchunk{domain MPOLY MultivariatePolynomial}
\getchunk{domain MYEXPR MyExpression}
\getchunk{domain MYUP MyUnivariatePolynomial}

\getchunk{domain NSDPS NeitherSparseOrDensePowerSeries}
\getchunk{domain NSMP NewSparseMultivariatePolynomial}
\getchunk{domain NSUP NewSparseUnivariatePolynomial}
\getchunk{domain NONE None}
\getchunk{domain NNI NonNegativeInteger}
\getchunk{domain NOTTING NottinghamGroup}
\getchunk{domain NIPROB NumericalIntegrationProblem}
\getchunk{domain ODEPROB NumericalODEProblem}
\getchunk{domain OPTPROB NumericalOptimizationProblem}
\getchunk{domain PDEPROB NumericalPDEProblem}

\getchunk{domain OCT Octonion}
\getchunk{domain ODEIFTBL ODEIntensityFunctionsTable}
\getchunk{domain ARRAY1 OneDimensionalArray}
\getchunk{domain ONECOMP OnePointCompletion}
\getchunk{domain OMCONN OpenMathConnection}
\getchunk{domain OMDEV OpenMathDevice}
\getchunk{domain OMENC OpenMathEncoding}
\getchunk{domain OMERR OpenMathError}
\getchunk{domain OMERRK OpenMathErrorKind}
\getchunk{domain OP Operator}
\getchunk{domain OML0 OppositeMonogenicLinearOperator}
\getchunk{domain ORDCOMP OrderedCompletion}
\getchunk{domain ODP OrderedDirectProduct}
\getchunk{domain OFMONOID OrderedFreeMonoid}
\getchunk{domain OVAR OrderedVariableList}
\getchunk{domain ODPOL OrderlyDifferentialPolynomial}

```

```

\getchunk{domain ODVAR OrderlyDifferentialVariable}
\getchunk{domain ODR OrdinaryDifferentialRing}
\getchunk{domain OWP OrdinaryWeightedPolynomials}
\getchunk{domain OSI OrdSetInts}
\getchunk{domain OUTFORM OutputForm}

\getchunk{domain PADIC PAdicInteger}
\getchunk{domain PADICRC PAdicRationalConstructor}
\getchunk{domain PADICRAT PAdicRational}
\getchunk{domain PALETTE Palette}
\getchunk{domain PARPCURV ParametricPlaneCurve}
\getchunk{domain PARSCURV ParametricSpaceCurve}
\getchunk{domain PARSURF ParametricSurface}
\getchunk{domain PFR PartialFraction}
\getchunk{domain PLACES Places}
\getchunk{domain PLACESPS PlacesOverPseudoAlgebraicClosureOfFiniteField}
\getchunk{domain PARTITION Partition}
\getchunk{domain PATTERN Pattern}
\getchunk{domain PATLRES PatternMatchListResult}
\getchunk{domain PATRES PatternMatchResult}
\getchunk{domain PENDTREE PendantTree}
\getchunk{domain PERM Permutation}
\getchunk{domain PERMGRP PermutationGroup}
\getchunk{domain HACKPI Pi}
\getchunk{domain ACPLLOT PlaneAlgebraicCurvePlot}
\getchunk{domain PLCS Plcs}
\getchunk{domain PLOT Plot}
\getchunk{domain PLOT3D Plot3D}
\getchunk{domain PBWL B PoincareBirkhoffWittLyndonBasis}
\getchunk{domain POINT Point}
\getchunk{domain POLY Polynomial}
\getchunk{domain IDEAL PolynomialIdeals}
\getchunk{domain PR PolynomialRing}
\getchunk{domain PI PositiveInteger}
\getchunk{domain PF PrimeField}
\getchunk{domain PRIMARR PrimitiveArray}
\getchunk{domain PRODUCT Product}
\getchunk{domain PROJPL ProjectivePlane}
\getchunk{domain PROJSP ProjectiveSpace}
\getchunk{domain PACEXT PseudoAlgebraicClosureOfAlgExtOfRationalNumber}
\getchunk{domain PACOFF PseudoAlgebraicClosureOfFiniteField}
\getchunk{domain PACRAT PseudoAlgebraicClosureOfRationalNumber}

\getchunk{domain QFORM QuadraticForm}
\getchunk{domain QALGSET QuasiAlgebraicSet}
\getchunk{domain QUAT Quaternion}
\getchunk{domain QEQUAT QueryEquation}
\getchunk{domain QUEUE Queue}

\getchunk{domain RADFF RadicalFunctionField}

```

```

\getchunk{domain RADIX RadixExpansion}
\getchunk{domain RECLOS RealClosure}
\getchunk{domain RMATRIX RectangularMatrix}
\getchunk{domain REF Reference}
\getchunk{domain RGCHAIN RegularChain}
\getchunk{domain REGSET RegularTriangularSet}
\getchunk{domain RESRING ResidueRing}
\getchunk{domain RESULT Result}
\getchunk{domain RULE RewriteRule}
\getchunk{domain ROIRC RightOpenIntervalRootCharacterization}
\getchunk{domain ROMAN RomanNumeral}
\getchunk{domain ROUTINE RoutinesTable}
\getchunk{domain RULECOLD RuleCalled}
\getchunk{domain RULESET Ruleset}

\getchunk{domain FORMULA ScriptFormulaFormat}
\getchunk{domain SEG Segment}
\getchunk{domain SEGBIND SegmentBinding}
\getchunk{domain SET Set}
\getchunk{domain SEX SExpression}
\getchunk{domain SEXOF SExpressionOf}
\getchunk{domain SAE SimpleAlgebraicExtension}
\getchunk{domain SFORT SimpleFortranProgram}
\getchunk{domain SINT SingleInteger}
\getchunk{domain SAOS SingletonAsOrderedSet}
\getchunk{domain SDPOL SequentialDifferentialPolynomial}
\getchunk{domain SDVAR SequentialDifferentialVariable}
\getchunk{domain SETMN SetOfMIntegersInOneToN}
\getchunk{domain SMP SparseMultivariatePolynomial}
\getchunk{domain SMTS SparseMultivariateTaylorSeries}
\getchunk{domain STBL SparseTable}
\getchunk{domain SULS SparseUnivariateLaurentSeries}
\getchunk{domain SUP SparseUnivariatePolynomial}
\getchunk{domain SUPEXP SparseUnivariatePolynomialExpressions}
\getchunk{domain SUPXS SparseUnivariatePuisseuxSeries}
\getchunk{domain ORESUP SparseUnivariateSkewPolynomial}
\getchunk{domain SUTS SparseUnivariateTaylorSeries}
\getchunk{domain SHDP SplitHomogeneousDirectProduct}
\getchunk{domain SPLNODE SplittingNode}
\getchunk{domain SPLTREE SplittingTree}
\getchunk{domain SREGSET SquareFreeRegularTriangularSet}
\getchunk{domain SQMATRIX SquareMatrix}
\getchunk{domain STACK Stack}
\getchunk{domain STREAM Stream}
\getchunk{domain STRING String}
\getchunk{domain STRTBL StringTable}
\getchunk{domain SUBSPACE SubSpace}
\getchunk{domain COMPPROP SubSpaceComponentProperty}
\getchunk{domain SUCH SuchThat}
\getchunk{domain SWITCH Switch}

```

```

\getchunk{domain SYMBOL Symbol}
\getchunk{domain SYMTAB SymbolTable}
\getchunk{domain SYMPOLY SymmetricPolynomial}

\getchunk{domain TABLE Table}
\getchunk{domain TABLEAU Tableau}
\getchunk{domain TS TaylorSeries}
\getchunk{domain TEX TexFormat}
\getchunk{domain TEXTFILE TextFile}
\getchunk{domain SYMS TheSymbolTable}
\getchunk{domain M3D ThreeDimensionalMatrix}
\getchunk{domain VIEW3D ThreeDimensionalViewport}
\getchunk{domain SPACE3 ThreeSpace}
\getchunk{domain TREE Tree}
\getchunk{domain TUBE TubePlot}
\getchunk{domain TUPLE Tuple}
\getchunk{domain ARRAY2 TwoDimensionalArray}
\getchunk{domain VIEW2D TwoDimensionalViewport}

\getchunk{domain UFPS UnivariateFormalPowerSeries}
\getchunk{domain ULS UnivariateLaurentSeries}
\getchunk{domain ULSCONS UnivariateLaurentSeriesConstructor}
\getchunk{domain UP UnivariatePolynomial}
\getchunk{domain UPXS UnivariatePuisseuxSeries}
\getchunk{domain UPXSCONS UnivariatePuisseuxSeriesConstructor}
\getchunk{domain UPXSSING UnivariatePuisseuxSeriesWithExponentialSingularity}
\getchunk{domain OREUP UnivariateSkewPolynomial}
\getchunk{domain UTS UnivariateTaylorSeries}
\getchunk{domain UTSZ UnivariateTaylorSeriesCZero}
\getchunk{domain UNISEG UniversalSegment}

\getchunk{domain U8MAT U8Matrix}
\getchunk{domain U16MAT U16Matrix}
\getchunk{domain U32MAT U32Matrix}
\getchunk{domain U8VEC U8Vector}
\getchunk{domain U16VEC U16Vector}
\getchunk{domain U32VEC U32Vector}
\getchunk{domain VARIABLE Variable}
\getchunk{domain VECTOR Vector}
\getchunk{domain VOID Void}

\getchunk{domain WP WeightedPolynomials}
\getchunk{domain WUTSET WuWenTsunTriangularSet}

\getchunk{domain XDPOLY XDistributedPolynomial}
\getchunk{domain XPBWPOLY XPBWPolynomial}
\getchunk{domain XPOLY XPolynomial}
\getchunk{domain XPR XPolynomialRing}
\getchunk{domain XRPOLY XRecursivePolynomial}

```

Chapter 30

Index

Index

- *
 - NOTTING, [1744](#)
- **
 - NOTTING, [1744](#)
- ?
 - ALGFF, [28](#)
 - ALGSC, [15](#)
 - AN, [35](#)
 - ANTISYM, [40](#)
 - BINARY, [292](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - CARTEN, [356](#)
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)
 - CLIF, [402](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DERHAM, [530](#)
 - DFLOAT, [587](#)
 - DFMAT, [599](#)
 - DFVEC, [605](#)
 - DHMATRIX, [491](#)
 - DIRPROD, [547](#)
 - DIRRING, [564](#)
 - DIV, [575](#)
 - DMP, [572](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DSMP, [542](#)
 - EMR, [702](#)
 - EQ, [691](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FAGROUP, [1004](#)
 - FDIV, [813](#)
 - FEXPR, [947](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FLOAT, [908](#)
 - FM, [1013](#)
 - FM1, [1017](#)
 - FNLA, [1027](#)
 - FR, [786](#)
 - FRAC, [986](#)
 - FSERIES, [978](#)
 - GCNAALG, [1065](#)
 - GDMP, [1052](#)
 - GMODPOL, [1059](#)
 - GSERIES, [1090](#)
 - HACKPI, [1978](#)
 - HDMP, [1180](#)
 - HDP, [1173](#)
 - HELLFDIV, [1183](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - IDPAG, [1202](#)
 - IFF, [1284](#)
 - IMATRIX, [1239](#)
 - INT, [1362](#)
 - INTRVL, [1384](#)
 - IPADIC, [1294](#)
 - IPF, [1303](#)
 - IR, [1375](#)
 - ISUPS, [1311](#)

ITAYLOR, 1338
IVECTOR, 1260
JORDAN, 223
LA, 1520
LAUPOL, 1422
LIE, 229
LO, 1523
LODO, 1469
LODO1, 1479
LODO2, 1491
LPOLY, 1447
LSQM, 1456
MATRIX, 1623
MCMPLX, 1543
MFLOAT, 1548
MINT, 1557
MODFIELD, 1638
MODMON, 1632
MODOP, 1647, 1804
MODRING, 1641
MPOLY, 1682
MRING, 1659
MYEXPR, 1688
MYUP, 1695
NSDPS, 1703
NSMP, 1714
NSUP, 1729
OCT, 1765
ODP, 1816
ODPOL, 1852
ODR, 1858
OMLO, 1806
ONECOMP, 1777
ORDCOMP, 1810
ORESUP, 2492
OREUP, 2872
OUTFORM, 1868
OWP, 1861
PACOFF, 2136
PACRAT, 2146
PADIC, 1881
PADICRAT, 1886
PADICRC, 1891
PF, 2105
PFR, 1914
PLACES, 2019
PLACESPS, 2021
POINT, 2060
POLY, 2078
PR, 2092
PRODUCT, 2113
QFORM, 2154
QUAT, 2167
RADFF, 2194
RADIX, 2206
RECLOS, 2237
RESRING, 2296
RMATRIX, 2246
ROMAN, 2326
SAE, 2400
SD, 2572
SDPOL, 2386
SHDP, 2509
SINT, 2412
SMP, 2423
SMTS, 2441
SQMATRIX, 2548
SULS, 2457
SUP, 2467
SUPEXPR, 2481
SUPXS, 2487
SUTS, 2497
SYMPOLY, 2654
TS, 2671
U16MAT, 2906
U32MAT, 2911
U8MAT, 2902
UFPS, 2789
ULS, 2796
ULSCONS, 2804
UP, 2828
UPXS, 2834
UPXSCONS, 2842
UPXSSING, 2852
UTS, 2877
UTSZ, 2887
VECTOR, 2942
WP, 2949
XDPOLY, 2967
XPBWPLYL, 2987
XPOLY, 2998
XPR, 3007

- XRPOLY, [3013](#)
 - ZMOD, [1368](#)
- /
 - NOTTING, [1744](#)
- =
 - NOTTING, [1744](#)
- ?/ΓE30F?
 - BITS, [315](#)
 - BOOLEAN, [322](#)
- ?<?
 - ALIST, [235](#)
 - AN, [35](#)
 - ARRAY1, [1774](#)
 - BINARY, [292](#)
 - BITS, [315](#)
 - BOOLEAN, [322](#)
 - BOP, [274](#)
 - BPADICRT, [263](#)
 - BSD, [286](#)
 - CARD, [332](#)
 - CCLASS, [382](#)
 - CDFVEC, [432](#)
 - CHAR, [373](#)
 - COMPLEX, [419](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - DFVEC, [605](#)
 - DIRPROD, [547](#)
 - DLIST, [460](#)
 - DMP, [572](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DSMP, [542](#)
 - EAB, [743](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FAGROUP, [1004](#)
 - FARRAY, [886](#)
 - FCOMP, [975](#)
 - FEXPR, [947](#)
 - FLOAT, [908](#)
 - FMONOID, [1021](#)
 - FRAC, [986](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - HDP, [1173](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - IARRAY1, [1244](#)
 - IBITS, [1199](#)
 - ICARD, [1193](#)
 - IDPOAM, [1212](#)
 - IDPOAMS, [1215](#)
 - IFARRAY, [1222](#)
 - ILIST, [1231](#)
 - INDE, [1218](#)
 - INT, [1362](#)
 - INTRVL, [1384](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - KERNEL, [1404](#)
 - LA, [1520](#)
 - LIST, [1504](#)
 - LO, [1523](#)
 - LWORD, [1532](#)
 - MAGMA, [1565](#)
 - MCMPLEX, [1543](#)
 - MFLOAT, [1548](#)
 - MINT, [1557](#)
 - MKCHSET, [1570](#)
 - MODMON, [1632](#)
 - MODMONOM, [1644](#)
 - MPOLY, [1682](#)
 - MSET, [1670](#)
 - MYEXPR, [1688](#)
 - MYUP, [1695](#)
 - NNI, [1739](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - OCT, [1765](#)
 - ODP, [1816](#)
 - ODPOL, [1852](#)
 - ODVAR, [1855](#)
 - OFMONOID, [1829](#)
 - ONECOMP, [1777](#)
 - ORDCOMP, [1810](#)
 - OSI, [1864](#)
 - OUTFORM, [1868](#)
 - OVAR, [1836](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)

- PBWL_B, 2054
- PERM, 1949
- PERMGRP, 1960
- PI, 2100
- POINT, 2060
- POLY, 2078
- PRIMARR, 2109
- PRODUCT, 2113
- PRITITION, 1923
- QUAT, 2167
- RADIX, 2206
- RECLOS, 2237
- ROMAN, 2326
- SAOS, 2418
- SDPOL, 2386
- SDVAR, 2389
- SET, 2372
- SHDP, 2509
- SINT, 2412
- SMP, 2423
- STRING, 2607
- SULS, 2457
- SUP, 2467
- SUPEXPR, 2481
- SYMBOL, 2640
- U16VEC, 2924
- U32VEC, 2931
- U8VEC, 2917
- ULS, 2796
- ULSCONS, 2804
- UP, 2828
- VECTOR, 2942
- ?<=?
- ALIST, 235
- AN, 35
- ARRAY1, 1774
- BINARY, 292
- BITS, 315
- BOOLEAN, 322
- BOP, 274
- BPADICRT, 263
- BSD, 286
- CARD, 332
- CDFVEC, 432
- CHAR, 373
- COMPLEX, 419
- DECIMAL, 465
- DFLOAT, 587
- DFVEC, 605
- DIRPROD, 547
- DIV, 575
- DLIST, 460
- DMP, 572
- DPMM, 553
- DPMO, 557
- DSMP, 542
- EAB, 743
- EXPEXPAN, 712
- EXPR, 724
- EXPUPXS, 740
- FAGROUP, 1004
- FARRAY, 886
- FCOMP, 975
- FEXPR, 947
- FLOAT, 908
- FMONOID, 1021
- FRAC, 986
- GDMP, 1052
- HDMP, 1180
- HDP, 1173
- HEXADEC, 1143
- IAN, 1276
- IARRAY1, 1244
- IBITS, 1199
- ICARD, 1193
- IDPOAM, 1212
- IDPOAMS, 1215
- IFARRAY, 1222
- ILIST, 1231
- INDE, 1218
- INT, 1362
- INTRVL, 1384
- ISTRING, 1249
- IVECTOR, 1260
- KERNEL, 1404
- LA, 1520
- LIST, 1504
- LO, 1523
- LWORD, 1532
- MAGMA, 1565
- MCMP_{PLX}, 1543
- MFLOAT, 1548

- MINT, [1557](#)
- MKCHSET, [1570](#)
- MODMON, [1632](#)
- MODMONOM, [1644](#)
- MPOLY, [1682](#)
- MYEXPR, [1688](#)
- MYUP, [1695](#)
- NNI, [1739](#)
- NSMP, [1714](#)
- NSUP, [1729](#)
- OCT, [1765](#)
- ODP, [1816](#)
- ODPOL, [1852](#)
- ODVAR, [1855](#)
- OFMONOID, [1829](#)
- ONECOMP, [1777](#)
- ORDCOMP, [1810](#)
- OSI, [1864](#)
- OUTFORM, [1868](#)
- OVAR, [1836](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PBWLB, [2054](#)
- PERM, [1949](#)
- PERMGRP, [1960](#)
- PI, [2100](#)
- POINT, [2060](#)
- POLY, [2078](#)
- PRIMARR, [2109](#)
- PRODUCT, [2113](#)
- PRITION, [1923](#)
- QUAT, [2167](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- ROMAN, [2326](#)
- SAOS, [2418](#)
- SDPOL, [2386](#)
- SDVAR, [2389](#)
- SHDP, [2509](#)
- SINT, [2412](#)
- SMP, [2423](#)
- STRING, [2607](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUEXPR, [2481](#)
- SYMBOL, [2640](#)
- U16VEC, [2924](#)
- U32VEC, [2931](#)
- U8VEC, [2917](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- VECTOR, [2942](#)
- ?=?
- BSD, [286](#)
- ?>?
- ALIST, [235](#)
- AN, [35](#)
- ARRAY1, [1774](#)
- BINARY, [292](#)
- BITS, [315](#)
- BOOLEAN, [322](#)
- BOP, [274](#)
- BPADICRT, [263](#)
- BSD, [286](#)
- CARD, [332](#)
- CDFVEC, [432](#)
- CHAR, [373](#)
- COMPLEX, [419](#)
- DECIMAL, [465](#)
- DFLOAT, [587](#)
- DFVEC, [605](#)
- DIRPROD, [547](#)
- DLIST, [460](#)
- DMP, [572](#)
- DPMM, [553](#)
- DPMO, [557](#)
- DSMP, [542](#)
- EAB, [743](#)
- EXPEXPAN, [712](#)
- EXPR, [724](#)
- EXPUPXS, [740](#)
- FAGROUP, [1004](#)
- FARRAY, [886](#)
- FCOMP, [975](#)
- FEXPR, [947](#)
- FLOAT, [908](#)
- FMONOID, [1021](#)
- FRAC, [986](#)
- GDMP, [1052](#)
- HDMP, [1180](#)
- HDP, [1173](#)

- HEXADEC, [1143](#)
- IAN, [1276](#)
- IARRAY1, [1244](#)
- IBITS, [1199](#)
- ICARD, [1193](#)
- IDPOAM, [1212](#)
- IDPOAMS, [1215](#)
- IFARRAY, [1222](#)
- ILIST, [1231](#)
- INDE, [1218](#)
- INT, [1362](#)
- INTRVL, [1384](#)
- ISTRING, [1249](#)
- IVECTOR, [1260](#)
- KERNEL, [1404](#)
- LA, [1520](#)
- LIST, [1504](#)
- LO, [1523](#)
- LWORD, [1532](#)
- MAGMA, [1565](#)
- MCMPLX, [1543](#)
- MFLOAT, [1548](#)
- MINT, [1557](#)
- MKCHSET, [1570](#)
- MODMON, [1632](#)
- MODMONOM, [1644](#)
- MPOLY, [1682](#)
- MYEXPR, [1688](#)
- MYUP, [1695](#)
- NNI, [1739](#)
- NSMP, [1714](#)
- NSUP, [1729](#)
- OCT, [1765](#)
- ODP, [1816](#)
- ODPOL, [1852](#)
- ODVAR, [1855](#)
- OFMONOID, [1829](#)
- ONECOMP, [1777](#)
- ORDCOMP, [1810](#)
- OSI, [1864](#)
- OUTFORM, [1868](#)
- OVAR, [1836](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PBWLb, [2054](#)
- PERM, [1949](#)
- PI, [2100](#)
- POINT, [2060](#)
- POLY, [2078](#)
- PRIMARR, [2109](#)
- PRODUCT, [2113](#)
- PRITION, [1923](#)
- QUAT, [2167](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- ROMAN, [2326](#)
- SAOS, [2418](#)
- SDPOL, [2386](#)
- SDVAR, [2389](#)
- SHDP, [2509](#)
- SINT, [2412](#)
- SMP, [2423](#)
- STRING, [2607](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SYMBOL, [2640](#)
- U16VEC, [2924](#)
- U32VEC, [2931](#)
- U8VEC, [2917](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- VECTOR, [2942](#)
- ?>=?
- ALIST, [235](#)
- AN, [35](#)
- ARRAY1, [1774](#)
- BINARY, [292](#)
- BITS, [315](#)
- BOOLEAN, [322](#)
- BOP, [274](#)
- BPADICRT, [263](#)
- BSD, [286](#)
- CARD, [332](#)
- CDFVEC, [432](#)
- CHAR, [373](#)
- COMPLEX, [419](#)
- DECIMAL, [465](#)
- DFLOAT, [587](#)
- DFVEC, [605](#)
- DIRPROD, [547](#)

- DLIST, 460
DMP, 572
DPMM, 553
DPMO, 557
DSMP, 542
EAB, 743
EXPEXPAN, 712
EXPR, 724
EXPUPXS, 740
FAGROUP, 1004
FARRAY, 886
FCOMP, 975
FEXPR, 947
FLOAT, 908
FMONOID, 1021
FRAC, 986
GDMP, 1052
HDMP, 1180
HDP, 1173
HEXADEC, 1143
IAN, 1276
IARRAY1, 1244
IBITS, 1199
ICARD, 1193
IDPOAM, 1212
IDPOAMS, 1215
IFARRAY, 1222
ILIST, 1231
INDE, 1218
INT, 1362
INTRVL, 1384
ISTRING, 1249
IVECTOR, 1260
KERNEL, 1404
LA, 1520
LIST, 1504
LO, 1523
LWORD, 1532
MAGMA, 1565
MCMPLEX, 1543
MFLOAT, 1548
MINT, 1557
MKCHSET, 1570
MODMON, 1632
MODMONOM, 1644
MPOLY, 1682
MYEXPR, 1688
MYUP, 1695
NNI, 1739
NSMP, 1714
NSUP, 1729
OCT, 1765
ODP, 1816
ODPOL, 1852
ODVAR, 1855
OFMONOID, 1829
ONECOMP, 1777
ORDCOMP, 1810
OSI, 1864
OUTFORM, 1868
OVAR, 1836
PADICRAT, 1886
PADICRC, 1891
PBWLB, 2054
PERM, 1949
PI, 2100
POINT, 2060
POLY, 2078
PRIMARR, 2109
PRODUCT, 2113
PRITITION, 1923
QUAT, 2167
RADIX, 2206
RECLOS, 2237
ROMAN, 2326
SAOS, 2418
SDPOL, 2386
SDVAR, 2389
SHDP, 2509
SINT, 2412
SMP, 2423
STRING, 2607
SULS, 2457
SUP, 2467
SUPEXP, 2481
SYMBOL, 2640
U16VEC, 2924
U32VEC, 2931
U8VEC, 2917
ULS, 2796
ULSCONS, 2804
UP, 2828

- VECTOR, [2942](#)
- ? $\Gamma E30F$ /?
 - IBITS, [1199](#)
 - SINT, [2412](#)
- ? $\Gamma E30F$ /?
 - BITS, [315](#)
 - BOOLEAN, [322](#)
- ? =?
 - DIRRING, [564](#)
- ?**?
 - ALGFF, [28](#)
 - ALGSC, [15](#)
 - AN, [35](#)
 - ANTISYM, [40](#)
 - AUTOMOR, [245](#)
 - BINARY, [292](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - CARD, [332](#)
 - CDFMAT, [426](#)
 - CLIF, [402](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DERHAM, [530](#)
 - DFLOAT, [587](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - DIRPROD, [547](#)
 - DIRRING, [564](#)
 - DMP, [572](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DSMP, [542](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EMR, [702](#)
 - EQ, [691](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FEXPR, [947](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FGROUP, [1010](#)
 - FLOAT, [908](#)
 - FMONOID, [1021](#)
 - FNLA, [1027](#)
 - FR, [786](#)
 - FRAC, [986](#)
 - FRIDEAL, [995](#)
 - FRMOD, [1000](#)
 - Fseries, [978](#)
 - GCNAALG, [1065](#)
 - GDMP, [1052](#)
 - Gseries, [1090](#)
 - HACKPI, [1978](#)
 - HDMP, [1180](#)
 - HDP, [1173](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - IDEAL, [2081](#)
 - IFF, [1284](#)
 - IMATRIX, [1239](#)
 - INFORM, [1343](#)
 - INT, [1362](#)
 - INTRVL, [1384](#)
 - IPADIC, [1294](#)
 - IPF, [1303](#)
 - ISUPS, [1311](#)
 - ITAYLOR, [1338](#)
 - JORDAN, [223](#)
 - LA, [1520](#)
 - LAUPOL, [1422](#)
 - LEXP, [1435](#)
 - LIE, [229](#)
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - MCMPLEX, [1543](#)
 - MFLOAT, [1548](#)
 - MINT, [1557](#)
 - MODFIELD, [1638](#)

- MODMON, 1632
 MODOP, 1647, 1804
 MODRING, 1641
 MOEBIUS, 1654
 MPOLY, 1682
 MRING, 1659
 MYEXPR, 1688
 MYUP, 1695
 NNI, 1739
 NSDPS, 1703
 NSMP, 1714
 NSUP, 1729
 OCT, 1765
 ODP, 1816
 ODPOL, 1852
 ODR, 1858
 OFMONOID, 1829
 OMLO, 1806
 ONECOMP, 1777
 ORDCOMP, 1810
 ORESUP, 2492
 OREUP, 2872
 OUTFORM, 1868
 OWP, 1861
 PACOFF, 2136
 PACRAT, 2146
 PADIC, 1881
 PADICRAT, 1886
 PADICRC, 1891
 PATTERN, 1928
 PERM, 1949
 PF, 2105
 PFR, 1914
 PI, 2100
 POLY, 2078
 PR, 2092
 PRODUCT, 2113
 QUAT, 2167
 RADFF, 2194
 RADIX, 2206
 RECLOS, 2237
 RESRING, 2296
 ROMAN, 2326
 SAE, 2400
 SD, 2572
 SDPOL, 2386
 SHDP, 2509
 SINT, 2412
 SMP, 2423
 SMTS, 2441
 SQMATRIX, 2548
 SULS, 2457
 SUP, 2467
 SUPEXPR, 2481
 SUPXS, 2487
 SUTS, 2497
 SYMPOLY, 2654
 TS, 2671
 U16MAT, 2906
 U32MAT, 2911
 U8MAT, 2902
 UFPS, 2789
 ULS, 2796
 ULSCONS, 2804
 UP, 2828
 UPXS, 2834
 UPXSCONS, 2842
 UPXSSING, 2852
 UTS, 2877
 UTSZ, 2887
 WP, 2949
 XDPOLY, 2967
 XPBWPOLYL, 2987
 XPOLY, 2998
 XPR, 3007
 XRPOLY, 3013
 ZMOD, 1368
 ??
 ALGFF, 28
 ALGSC, 15
 AN, 35
 ANTISYM, 40
 AUTOMOR, 245
 BINARY, 292
 BPADIC, 258
 BPADICRT, 263
 CARD, 332
 CARTEN, 356
 CDFMAT, 426
 CDFVEC, 432
 CLIF, 402
 COLOR, 407

- COMPLEX, [419](#)
CONTFRAC, [445](#)
DECIMAL, [465](#)
DERHAM, [530](#)
DFLOAT, [587](#)
DFMAT, [599](#)
DFVEC, [605](#)
DHMATRIX, [491](#)
DIRPROD, [547](#)
DIRRING, [564](#)
DIV, [575](#)
DMP, [572](#)
DPMM, [553](#)
DPMO, [557](#)
DSMP, [542](#)
EMR, [702](#)
EQ, [691](#)
EXPEXPAN, [712](#)
EXPR, [724](#)
EXPUPXS, [740](#)
FAGROUP, [1004](#)
FAMONOID, [1007](#)
FDIV, [813](#)
FEXPR, [947](#)
FF, [820](#)
FFCG, [825](#)
FFCGP, [835](#)
FFCGX, [830](#)
FFNB, [860](#)
FFNBP, [872](#)
FFNBX, [866](#)
FFP, [851](#)
FFX, [846](#)
FGROUP, [1010](#)
FLOAT, [908](#)
FM, [1013](#)
FM1, [1017](#)
FMONOID, [1021](#)
FNLA, [1027](#)
FR, [786](#)
FRAC, [986](#)
FRIDEAL, [995](#)
FRMOD, [1000](#)
FSERIES, [978](#)
GCNAALG, [1065](#)
GDMP, [1052](#)
GMODPOL, [1059](#)
GSERIES, [1090](#)
HACKPI, [1978](#)
HDMP, [1180](#)
HDP, [1173](#)
HELLFDIV, [1183](#)
HEXADEC, [1143](#)
IAN, [1276](#)
IDEAL, [2081](#)
IDPAG, [1202](#)
IDPAM, [1206](#)
IDPOAM, [1212](#)
IDPOAMS, [1215](#)
IFAMON, [1287](#)
IFF, [1284](#)
IMATRIX, [1239](#)
INDE, [1218](#)
INFORM, [1343](#)
INT, [1362](#)
INTRVL, [1384](#)
IPADIC, [1294](#)
IPF, [1303](#)
IR, [1375](#)
ISUPS, [1311](#)
ITAYLOR, [1338](#)
IVECTOR, [1260](#)
JORDAN, [223](#)
LA, [1520](#)
LAUPOL, [1422](#)
LEXP, [1435](#)
LIE, [229](#)
LO, [1523](#)
LODO, [1469](#)
LODO1, [1479](#)
LODO2, [1491](#)
LPOLY, [1447](#)
LSQM, [1456](#)
MAGMA, [1565](#)
MATRIX, [1623](#)
MCMPLEX, [1543](#)
MFLOAT, [1548](#)
MINT, [1557](#)
MODFIELD, [1638](#)
MODMON, [1632](#)
MODOP, [1647](#), [1804](#)
MODRING, [1641](#)

- MOEBIUS, 1654
 MPOLY, 1682
 MRING, 1659
 MYEXPR, 1688
 MYUP, 1695
 NNI, 1739
 NSDPS, 1703
 NSMP, 1714
 NSUP, 1729
 OCT, 1765
 ODP, 1816
 ODPOL, 1852
 ODR, 1858
 OFMONOID, 1829
 OMLO, 1806
 ONECOMP, 1777
 ORDCOMP, 1810
 ORESUP, 2492
 OREUP, 2872
 OUTFORM, 1868
 OWP, 1861
 PACOFF, 2136
 PACRAT, 2146
 PADIC, 1881
 PADICRAT, 1886
 PADICRC, 1891
 PATTERN, 1928
 PERM, 1949
 PF, 2105
 PFR, 1914
 PI, 2100
 PLACES, 2019
 PLACESPS, 2021
 POINT, 2060
 POLY, 2078
 PR, 2092
 PRODUCT, 2113
 PRITITION, 1923
 QFORM, 2154
 QUAT, 2167
 RADFF, 2194
 RADIX, 2206
 RECLOS, 2237
 RESRING, 2296
 RMATRIX, 2246
 ROMAN, 2326
 SAE, 2400
 SD, 2572
 SDPOL, 2386
 SHDP, 2509
 SINT, 2412
 SMP, 2423
 SMTS, 2441
 SQMATRIX, 2548
 SULS, 2457
 SUP, 2467
 SUPEXPR, 2481
 SUPXS, 2487
 SUTS, 2497
 SYMPOLY, 2654
 TS, 2671
 U16MAT, 2906
 U32MAT, 2911
 U8MAT, 2902
 UFPS, 2789
 ULS, 2796
 ULSCONS, 2804
 UP, 2828
 UPXS, 2834
 UPXSCONS, 2842
 UPXSSING, 2852
 UTS, 2877
 UTSZ, 2887
 VECTOR, 2942
 WP, 2949
 XDPOLY, 2967
 XPBWPOLYL, 2987
 XPOLY, 2998
 XPR, 3007
 XRPOLY, 3013
 ZMOD, 1368
 ?+?
 ALGFF, 28
 ALGSC, 15
 AN, 35
 ANTISYM, 40
 BINARY, 292
 BPADIC, 258
 BPADICRT, 263
 CARD, 332
 CARTEN, 356
 CDFMAT, 426

- CDFVEC, 432
CLIF, 402
COLOR, 407
COMPLEX, 419
CONTFRAC, 445
DBASE, 454
DECIMAL, 465
DERHAM, 530
DFLOAT, 587
DFMAT, 599
DFVEC, 605
DHMATRIX, 491
DIRPROD, 547
DIRRING, 564
DIV, 575
DMP, 572
DPMO, 557
DSMP, 542
EMR, 702
EQ, 691
EXPEXPAN, 712
EXPR, 724
EXPUPXS, 740
FAGROUP, 1004
FAMONOID, 1007
FDIV, 813
FEXPR, 947
FF, 820
FFCG, 825
FFCGP, 835
FFCGX, 830
FFNB, 860
FFNBP, 872
FFNBX, 866
FFP, 851
FFX, 846
FLOAT, 908
FM, 1013
FM1, 1017
FNLA, 1027
FPARFRAC, 1040
FR, 786
FRAC, 986
FSERIES, 978
GCNAALG, 1065
GDMP, 1052
GMODPOL, 1059
GSERIES, 1090
HACKPI, 1978
HDMP, 1180
HDP, 1173
HELLFDIV, 1183
HEXADEC, 1143
IAN, 1276
IDEAL, 2081
IDPAG, 1202
IDPAM, 1206
IDPOAM, 1212
IDPOAMS, 1215
IFAMON, 1287
IFF, 1284
IMATRIX, 1239
INDE, 1218
INFORM, 1343
INT, 1362
INTRVL, 1384
IPADIC, 1294
IPF, 1303
IR, 1375
ISUPS, 1311
ITAYLOR, 1338
IVECTOR, 1260
JORDAN, 223
LA, 1520
LAUPOL, 1422
LIE, 229
LO, 1523
LODO, 1469
LODO1, 1479
LODO2, 1491
LPOLY, 1447
LSQM, 1456
MATRIX, 1623
MCMPLX, 1543
MFLOAT, 1548
MINT, 1557
MODFIELD, 1638
MODMON, 1632
MODOP, 1647, 1804
MODRING, 1641
MPOLY, 1682
MRING, 1659

- MYEXPR, 1688
 MYUP, 1695
 NNI, 1739
 NSDPS, 1703
 NSMP, 1714
 NSUP, 1729
 OCT, 1765
 ODP, 1816
 ODPOL, 1852
 ODR, 1858
 OMLO, 1806
 ONECOMP, 1777
 ORDCOMP, 1810
 ORESUP, 2492
 OREUP, 2872
 OUTFORM, 1868
 OWP, 1861
 PACOFF, 2136
 PACRAT, 2146
 PADIC, 1881
 PADICRAT, 1886
 PADICRC, 1891
 PATTERN, 1928
 PF, 2105
 PFR, 1914
 PI, 2100
 PLACES, 2019
 PLACESPS, 2021
 POINT, 2060
 POLY, 2078
 PR, 2092
 PRODUCT, 2113
 PRITITION, 1923
 QFORM, 2154
 QUAT, 2167
 RADFF, 2194
 RADIX, 2206
 RECLOS, 2237
 RESRING, 2296
 RMATRIX, 2246
 ROMAN, 2326
 SAE, 2400
 SD, 2572
 SDPOL, 2386
 SHDP, 2509
 SINT, 2412
 SMP, 2423
 SMTS, 2441
 SQMATRIX, 2548
 SULS, 2457
 SUP, 2467
 SUPEXPR, 2481
 SUPXS, 2487
 SUTS, 2497
 SYMPOLY, 2654
 TS, 2671
 U16MAT, 2906
 U32MAT, 2911
 U8MAT, 2902
 UFPS, 2789
 ULS, 2796
 ULSCONS, 2804
 UP, 2828
 UPXS, 2834
 UPXSCONS, 2842
 UPXSING, 2852
 UTS, 2877
 UTSZ, 2887
 VECTOR, 2942
 WP, 2949
 XDPOLY, 2967
 XPBWPLYL, 2987
 XPOLY, 2998
 XPR, 3007
 XRPOLY, 3013
 ZMOD, 1368
 ?-?
 ALGFF, 28
 ALGSC, 15
 AN, 35
 ANTISYM, 40
 BINARY, 292
 BPADIC, 258
 BPADICRT, 263
 CARD, 332
 CARTEN, 356
 CDFMAT, 426
 CDFVEC, 432
 CLIF, 402
 COMPLEX, 419
 CONTFRAC, 445
 DBASE, 454

- DECIMAL, [465](#)
DERHAM, [530](#)
DFLOAT, [587](#)
DFMAT, [599](#)
DFVEC, [605](#)
DHMATRIX, [491](#)
DIRPROD, [547](#)
DIRRING, [564](#)
DIV, [575](#)
DMP, [572](#)
DPMM, [553](#)
DPMO, [557](#)
DSMP, [542](#)
EMR, [702](#)
EQ, [691](#)
EXPEXPAN, [712](#)
EXPR, [724](#)
EXPUPXS, [740](#)
FAGROUP, [1004](#)
FDIV, [813](#)
FEXPR, [947](#)
FF, [820](#)
FFCG, [825](#)
FFCGP, [835](#)
FFCGX, [830](#)
FFNB, [860](#)
FFNBP, [872](#)
FFNBX, [866](#)
FFP, [851](#)
FFX, [846](#)
FLOAT, [908](#)
FM, [1013](#)
FM1, [1017](#)
FNLA, [1027](#)
FR, [786](#)
FRAC, [986](#)
FSERIES, [978](#)
GCNAALG, [1065](#)
GDMP, [1052](#)
GMODPOL, [1059](#)
GSERIES, [1090](#)
HACKPI, [1978](#)
HDMP, [1180](#)
HDP, [1173](#)
HELLFDIV, [1183](#)
HEXADEC, [1143](#)
IAN, [1276](#)
IDPAG, [1202](#)
IFF, [1284](#)
IMATRIX, [1239](#)
INT, [1362](#)
INTRVL, [1384](#)
IPADIC, [1294](#)
IPF, [1303](#)
IR, [1375](#)
ISUPS, [1311](#)
ITAYLOR, [1338](#)
IVECTOR, [1260](#)
JORDAN, [223](#)
LA, [1520](#)
LAUPOL, [1422](#)
LIE, [229](#)
LO, [1523](#)
LODO, [1469](#)
LODO1, [1479](#)
LODO2, [1491](#)
LPOLY, [1447](#)
LSQM, [1456](#)
MATRIX, [1623](#)
MCMPLEX, [1543](#)
MFLOAT, [1548](#)
MINT, [1557](#)
MODFIELD, [1638](#)
MODMON, [1632](#)
MODOP, [1647](#), [1804](#)
MODRING, [1641](#)
MPOLY, [1682](#)
MRING, [1659](#)
MYEXPR, [1688](#)
MYUP, [1695](#)
NSDPS, [1703](#)
NSMP, [1714](#)
NSUP, [1729](#)
OCT, [1765](#)
ODP, [1816](#)
ODPOL, [1852](#)
ODR, [1858](#)
OMLO, [1806](#)
ONECOMP, [1777](#)
ORDCOMP, [1810](#)
ORESUP, [2492](#)
OREUP, [2872](#)

- OUTFORM, 1868
- OWP, 1861
- PACOFF, 2136
- PACRAT, 2146
- PADIC, 1881
- PADICRAT, 1886
- PADICRC, 1891
- PF, 2105
- PFR, 1914
- PLACES, 2019
- PLACESPS, 2021
- POINT, 2060
- POLY, 2078
- PR, 2092
- PRODUCT, 2113
- QFORM, 2154
- QUAT, 2167
- RADFF, 2194
- RADIX, 2206
- RECLOS, 2237
- RESRING, 2296
- RMATRIX, 2246
- ROMAN, 2326
- SAE, 2400
- SD, 2572
- SDPOL, 2386
- SHDP, 2509
- SINT, 2412
- SMP, 2423
- SMTS, 2441
- SQMATRIX, 2548
- SULS, 2457
- SUP, 2467
- SUPEXPR, 2481
- SUPXS, 2487
- SUTS, 2497
- SYMPOLY, 2654
- TS, 2671
- U16MAT, 2906
- U32MAT, 2911
- U8MAT, 2902
- UFPS, 2789
- ULS, 2796
- ULSCONS, 2804
- UP, 2828
- UPXS, 2834
- UPXSCONS, 2842
- UPXSSING, 2852
- UTS, 2877
- UTSZ, 2887
- VECTOR, 2942
- WP, 2949
- XDPOLY, 2967
- XPBWPOLYL, 2987
- XPOLY, 2998
- XPR, 3007
- XPOLY, 3013
- ZMOD, 1368
- ?..?
- OUTFORM, 1868
- SEG, 2359
- UNISEG, 2896
- ???
- AFFPLPS, 7
- AFFSP, 9
- ALGSC, 15
- ALIST, 235
- ARRAY1, 1774
- AUTOMOR, 245
- BINARY, 292
- BITS, 315
- BPADICRT, 263
- CARTEN, 356
- CDFVEC, 432
- COMPLEX, 419
- DBASE, 454
- DECIMAL, 465
- DFVEC, 605
- DIRPROD, 547
- DIRRING, 564
- DLIST, 460
- DPMM, 553
- DPMO, 557
- EMR, 702
- EQTBL, 699
- EXPEXPAN, 712
- EXPUPXS, 740
- FARRAY, 886
- FR, 786
- FRAC, 986
- GCNAALG, 1065
- GSERIES, 1090

- GSTBL, 1079
- HASHTBL, 1120
- HDP, 1173
- HEXADEC, 1143
- IARRAY1, 1244
- IBITS, 1199
- ICARD, 1193
- IFARRAY, 1222
- ILIST, 1231
- INFORM, 1343
- INTABL, 1336
- ISTRING, 1249
- ISUPS, 1311
- IVECTOR, 1260
- JORDAN, 223
- KAFILE, 1414
- LIB, 1429
- LIE, 229
- LIST, 1504
- LODO, 1469
- LODO1, 1479
- LODO2, 1491
- LSQM, 1456
- MCMPLEX, 1543
- MODMON, 1632
- MODOP, 1647, 1804
- MYUP, 1695
- NSDPS, 1703
- NSUP, 1729
- OCT, 1765
- ODP, 1816
- OUTFORM, 1868
- PADICRAT, 1886
- PADICRC, 1891
- PERM, 1949
- PERMGRP, 1960
- PLACES, 2019
- PLACESPS, 2021
- POINT, 2060
- PRIMARR, 2109
- PROJPL, 2117
- PROJPLPS, 2119
- PROJSP, 2122
- QFORM, 2154
- QUAT, 2167
- RADIX, 2206
- RESULT, 2301
- ROUTINE, 2331
- RULE, 2305
- RULESET, 2342
- SEX, 2392
- SEXOF, 2394
- SHDP, 2509
- STBL, 2450
- STREAM, 2582
- STRING, 2607
- STRTBL, 2610
- SULS, 2457
- SUP, 2467
- SUPEXPR, 2481
- SUPXS, 2487
- SUTS, 2497
- SYMBOL, 2640
- TABLE, 2664
- U16VEC, 2924
- U32VEC, 2931
- U8VEC, 2917
- UFPS, 2789
- ULS, 2796
- ULSCONS, 2804
- UP, 2828
- UPXS, 2834
- UPXSCONS, 2842
- UTS, 2877
- UTSZ, 2887
- VECTOR, 2942
- ?.count
 - DLIST, 460
- ?.first
 - ALIST, 235
 - DLIST, 460
 - ILIST, 1231
 - LIST, 1504
 - NSDPS, 1703
 - STREAM, 2582
- ?.last
 - ALIST, 235
 - DLIST, 460
 - ILIST, 1231
 - LIST, 1504
 - NSDPS, 1703
 - STREAM, 2582

- ?.left
 - BBTREE, 253
 - BSTREE, 302
 - BTOURN, 306
 - BTREE, 310
 - PENDTREE, 1945
- ?.rest
 - ALIST, 235
 - DLIST, 460
 - ILIST, 1231
 - LIST, 1504
 - NSDPS, 1703
 - STREAM, 2582
- ?.right
 - BBTREE, 253
 - BSTREE, 302
 - BTOURN, 306
 - BTREE, 310
 - PENDTREE, 1945
- ?.sort
 - DLIST, 460
- ?.unique
 - DLIST, 460
- ?.value
 - ALIST, 235
 - BBTREE, 253
 - BSTREE, 302
 - BTOURN, 306
 - BTREE, 310
 - DLIST, 460
 - DSTREE, 535
 - ILIST, 1231
 - LIST, 1504
 - NSDPS, 1703
 - PENDTREE, 1945
 - SPLTREE, 2518
 - STREAM, 2582
 - TREE, 2741
- ?/ΓE30F?
 - IBITS, 1199
 - SINT, 2412
- ?/?
 - ALGFF, 28
 - AN, 35
 - AUTOMOR, 245
 - BINARY, 292
 - BPADICRT, 263
 - CDFMAT, 426
 - CLIF, 402
 - COMPLEX, 419
 - CONTFRAC, 445
 - DECIMAL, 465
 - DFLOAT, 587
 - DFMAT, 599
 - DHMATRIX, 491
 - DIRPROD, 547
 - DMP, 572
 - DPMM, 553
 - DPMO, 557
 - DSMP, 542
 - EQ, 691
 - EXPEXPAN, 712
 - EXPR, 724
 - EXPUPXS, 740
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866
 - FFP, 851
 - FFX, 846
 - FGROUP, 1010
 - FLOAT, 908
 - FRAC, 986
 - FRIDEAL, 995
 - GSERIES, 1090
 - HACKPI, 1978
 - HDMP, 1180
 - HDP, 1173
 - HEXADEC, 1143
 - IAN, 1276
 - IFF, 1284
 - IMATRIX, 1239
 - INFORM, 1343
 - IPF, 1303
 - ISUPS, 1311
 - LA, 1520
 - LEXP, 1435
 - LO, 1523
 - LPOLY, 1447

- LSQM, [1456](#)
- MATRIX, [1623](#)
- MCMPLEX, [1543](#)
- MFLOAT, [1548](#)
- MODFIELD, [1638](#)
- MODMON, [1632](#)
- MOEBIUS, [1654](#)
- MPOLY, [1682](#)
- MYEXPR, [1688](#)
- MYUP, [1695](#)
- NSDPS, [1703](#)
- NSMP, [1714](#)
- NSUP, [1729](#)
- ODP, [1816](#)
- ODPOL, [1852](#)
- ODR, [1858](#)
- OUTFORM, [1868](#)
- OWP, [1861](#)
- PACOFF, [2136](#)
- PACRAT, [2146](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PATTERN, [1928](#)
- PERM, [1949](#)
- PF, [2105](#)
- PFR, [1914](#)
- POLY, [2078](#)
- PR, [2092](#)
- PRODUCT, [2113](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- RMATRIX, [2246](#)
- SAE, [2400](#)
- SD, [2572](#)
- SDPOL, [2386](#)
- SHDP, [2509](#)
- SMP, [2423](#)
- SQMATRIX, [2548](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXP, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- SYMPOLY, [2654](#)
- TS, [2671](#)
- U16MAT, [2906](#)
- U32MAT, [2911](#)
- U8MAT, [2902](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSING, [2852](#)
- UTSZ, [2887](#)
- WP, [2949](#)
- ?=?
 - AFFPLPS, [7](#)
 - AFFSP, [9](#)
 - ALGFF, [28](#)
 - ALGSC, [15](#)
 - ALIST, [235](#)
 - AN, [35](#)
 - ANON, [38](#)
 - ANTISYM, [40](#)
 - ANY, [50](#)
 - ARRAY1, [1774](#)
 - ARRAY2, [2763](#)
 - ASTACK, [65](#)
 - ATTRBUT, [239](#)
 - AUTOMOR, [245](#)
 - BBTREE, [253](#)
 - BFUNCT, [265](#)
 - BINARY, [292](#)
 - BINFILE, [295](#)
 - BITS, [315](#)
 - BLHN, [317](#)
 - BLQT, [319](#)
 - BOOLEAN, [322](#)
 - BOP, [274](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - BSTREE, [302](#)
 - BTourn, [306](#)
 - BTREE, [310](#)
 - CARD, [332](#)
 - CARTEN, [356](#)
 - CCLASS, [382](#)
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)

- CHAR, 373
CLIF, 402
COLOR, 407
COMM, 411
COMPLEX, 419
COMPPROP, 2624
CONTFRAC, 445
D01AJFA, 614
D01AKFA, 617
D01ALFA, 620
D01AMFA, 623
D01APFA, 629, 633
D01ASFA, 636
D01FCFA, 639
D01GBFA, 643
D01TRNS, 646
D02BBFA, 651
D02BHFA, 654
D02CJFA, 658
D02EJFA, 661
D03EEFA, 665
D03FAFA, 668
D10ANFA, 626
DBASE, 454
DECIMAL, 465
DEQUEUE, 512
DERHAM, 530
DFLOAT, 587
DFMAT, 599
DFVEC, 605
DHMATRIX, 491
DIRPROD, 547
DIRRING, 564
DIV, 575
DLIST, 460
DMP, 572
DPMM, 553
DPMO, 557
DROPT, 608
DSMP, 542
DSTREE, 535
E04DGFA, 747
E04FDFA, 750
E04GCFA, 754
E04JAFA, 758
E04MBFA, 762
E04NAFA, 765
E04UCFA, 769
EAB, 743
EMR, 702
EQ, 691
EQTBL, 699
EXIT, 707
EXPEXPAN, 712
EXPR, 724
EXPUPXS, 740
FAGROUP, 1004
FAMONOID, 1007
FARRAY, 886
FC, 931
FCOMP, 975
FDIV, 813
FEXPR, 947
FF, 820
FFCG, 825
FFCGP, 835
FFCGX, 830
FFNB, 860
FFNBP, 872
FFNBX, 866
FFP, 851
FFX, 846
FGROUP, 1010
FILE, 802
FLOAT, 908
FM, 1013
FM1, 1017
FMONOID, 1021
FNAME, 810
FNLA, 1027
FORMULA, 2346
FPARFRAC, 1040
FR, 786
FRAC, 986
FRIDEAL, 995
FRMOD, 1000
FSERIES, 978
FST, 962
FT, 971
FTEM, 967
FUNCTION, 1045
GCNAALG, 1065

GDMP, 1052
GMODPOL, 1059
GOPT, 1104
GOPT0, 1110
GPOLSET, 1074
GRIMAGE, 1094
GSERIES, 1090
GSTBL, 1079
GTSET, 1083
HACKPI, 1978
HASHTBL, 1120
HDMP, 1180
HDP, 1173
HEAP, 1134
HELLFDIV, 1183
HEXADEC, 1143
HTMLFORM, 1152
IAN, 1276
IARRAY1, 1244
IARRAY2, 1256
IBITS, 1199
IC, 1191
ICARD, 1193
IDEAL, 2081
IDPAG, 1202
IDPAM, 1206
IDPO, 1209
IDPOAM, 1212
IDPOAMS, 1215
IFAMON, 1287
IFARRAY, 1222
IFF, 1284
IIARRAY2, 1290
ILIST, 1231
IMATRIX, 1239
INDE, 1218
INFCLSPS, 1271
INFCLSPT, 1265
INFORM, 1343
INT, 1362
INTABL, 1336
INTRVL, 1384
IPADIC, 1294
IPF, 1303
IR, 1375
ISTRING, 1249
ISUPS, 1311
ITAYLOR, 1338
IVECTOR, 1260
JORDAN, 223
KAFILE, 1414
KERNEL, 1404
LA, 1520
LAUPOL, 1422
LEXP, 1435
LIB, 1429
LIE, 229
LIST, 1504
LMDICT, 1515
LMOPS, 1509
LO, 1523
LODO, 1469
LODO1, 1479
LODO2, 1491
LPOLY, 1447
LSQM, 1456
LWORD, 1532
M3D, 2703
MAGMA, 1565
MATRIX, 1623
MCMPLX, 1543
MFLOAT, 1548
MINT, 1557
MKCHSET, 1570
MMLFORM, 1603
MODFIELD, 1638
MODMON, 1632
MODMONOM, 1644
MODOP, 1647, 1804
MODRING, 1641
MOEBIUS, 1654
MPOLY, 1682
MRING, 1659
MSET, 1670
MYEXPR, 1688
MYUP, 1695
NIPROB, 1746
NNI, 1739
NONE, 1737
NSDPS, 1703
NSMP, 1714
NSUP, 1729

- OCT, [1765](#)
ODEPROB, [1750](#)
ODP, [1816](#)
ODPOL, [1852](#)
ODR, [1858](#)
ODVAR, [1855](#)
OFMONOID, [1829](#)
OMENC, [1789](#)
OMERR, [1792](#)
OMERRK, [1794](#)
OMLO, [1806](#)
ONECOMP, [1777](#)
OPTPROB, [1753](#)
ORDCOMP, [1810](#)
ORESUP, [2492](#)
OREUP, [2872](#)
OSI, [1864](#)
OUTFORM, [1868](#)
OVAR, [1836](#)
OWP, [1861](#)
PACOFF, [2136](#)
PACRAT, [2146](#)
PADIC, [1881](#)
PADICRAT, [1886](#)
PADICRC, [1891](#)
PALETTE, [1896](#)
PATLRES, [1937](#)
PATRES, [1940](#)
PATTERN, [1928](#)
PBWLB, [2054](#)
PDEPROB, [1756](#)
PENDTREE, [1945](#)
PERM, [1949](#)
PERMGRP, [1960](#)
PF, [2105](#)
PFR, [1914](#)
PI, [2100](#)
PLACES, [2019](#)
PLACESPS, [2021](#)
POINT, [2060](#)
POLY, [2078](#)
PR, [2092](#)
PRIMARR, [2109](#)
PRODUCT, [2113](#)
PROJPL, [2117](#)
PROJPLPS, [2119](#)
PROJSP, [2122](#)
PRITITION, [1923](#)
QALGSET, [2157](#)
QFORM, [2154](#)
QUAT, [2167](#)
QUEUE, [2184](#)
RADFF, [2194](#)
RADIX, [2206](#)
RECLOS, [2237](#)
REF, [2249](#)
REGSET, [2286](#)
RESRING, [2296](#)
RESULT, [2301](#)
RGCHAIN, [2255](#)
RMATRIX, [2246](#)
ROIRC, [2310](#)
ROMAN, [2326](#)
ROUTINE, [2331](#)
RULE, [2305](#)
RULECOLD, [2340](#)
RULESET, [2342](#)
SAE, [2400](#)
SAOS, [2418](#)
SD, [2572](#)
SDPOL, [2386](#)
SDVAR, [2389](#)
SEG, [2359](#)
SEGBIND, [2364](#)
SET, [2372](#)
SETMN, [2377](#)
SEX, [2392](#)
SEXOF, [2394](#)
SHDP, [2509](#)
SINT, [2412](#)
SMP, [2423](#)
SMTS, [2441](#)
SPACE3, [2731](#)
SPLNODE, [2512](#)
SPLTREE, [2518](#)
SQMATRIX, [2548](#)
SREGSET, [2535](#)
STACK, [2563](#)
STBL, [2450](#)
STREAM, [2582](#)
STRING, [2607](#)
STRTBL, [2610](#)

- SUBSPACE, 2614
- SUCH, 2627
- SULS, 2457
- SUP, 2467
- SUPEXPR, 2481
- SUPXS, 2487
- SUTS, 2497
- SYMBOL, 2640
- SYMPOLY, 2654
- TABLE, 2664
- TEX, 2677
- TEXTFILE, 2692
- TREE, 2741
- TS, 2671
- TUPLE, 2752
- U16MAT, 2906
- U16VEC, 2924
- U32MAT, 2911
- U32VEC, 2931
- U8MAT, 2902
- U8VEC, 2917
- UFPS, 2789
- ULS, 2796
- ULSCONS, 2804
- UNISEG, 2896
- UP, 2828
- UPXS, 2834
- UPXSCONS, 2842
- UPXSSING, 2852
- UTS, 2877
- UTSZ, 2887
- VARIABLE, 2936
- VECTOR, 2942
- VIEW2d, 2770
- VIEW3D, 2710
- WP, 2949
- WUTSET, 2959
- XDPOLY, 2967
- XPBWPLYL, 2987
- XPOLY, 2998
- XPR, 3007
- XPOLY, 3013
- ZMOD, 1368
- ?SEGMENT
 - OUTFORM, 1868
 - UNISEG, 2896
- ?^=?
 - OUTFORM, 1868
- ?^?
 - ALGFF, 28
 - AN, 35
 - ANTISYM, 40
 - AUTOMOR, 245
 - BINARY, 292
 - BPADIC, 258
 - BPADICRT, 263
 - CARD, 332
 - CLIF, 402
 - COMPLEX, 419
 - CONTFRAC, 445
 - DECIMAL, 465
 - DERHAM, 530
 - DFLOAT, 587
 - DIRPROD, 547
 - DIRRING, 564
 - DMP, 572
 - DPMM, 553
 - DPMO, 557
 - DSMP, 542
 - EMR, 702
 - EQ, 691
 - EXPEXPAN, 712
 - EXPR, 724
 - EXPUPXS, 740
 - FEXPR, 947
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866
 - FFP, 851
 - FFX, 846
 - FGROUP, 1010
 - FLOAT, 908
 - FMONOID, 1021
 - FR, 786
 - FRAC, 986
 - FRIDEAL, 995
 - FRMOD, 1000
 - FSERIES, 978

- GDMP, 1052
GSERIES, 1090
HACKPI, 1978
HDMP, 1180
HDP, 1173
HEXADEC, 1143
IAN, 1276
IFF, 1284
INT, 1362
INTRVL, 1384
IPADIC, 1294
IPF, 1303
ISUPS, 1311
ITAYLOR, 1338
LA, 1520
LAUPOL, 1422
LEXP, 1435
LODO, 1469
LODO1, 1479
LODO2, 1491
LSQM, 1456
MCMPLX, 1543
MFLOAT, 1548
MINT, 1557
MODFIELD, 1638
MODMON, 1632
MODOP, 1647, 1804
MODRING, 1641
MOEBIUS, 1654
MPOLY, 1682
MRING, 1659
MYEXPR, 1688
MYUP, 1695
NNI, 1739
NSDPS, 1703
NSMP, 1714
NSUP, 1729
OCT, 1765
ODP, 1816
ODPOL, 1852
ODR, 1858
OFMONOID, 1829
OMLO, 1806
ONECOMP, 1777
ORDCOMP, 1810
ORESUP, 2492
OREUP, 2872
OWP, 1861
PACOFF, 2136
PACRAT, 2146
PADIC, 1881
PADICRAT, 1886
PADICRC, 1891
PERM, 1949
PF, 2105
PFR, 1914
PI, 2100
POLY, 2078
PR, 2092
PRODUCT, 2113
QUAT, 2167
RADFF, 2194
RADIX, 2206
RECLOS, 2237
RESRING, 2296
ROMAN, 2326
SAE, 2400
SD, 2572
SDPOL, 2386
SHDP, 2509
SINT, 2412
SMP, 2423
SMTS, 2441
SQMATRIX, 2548
SULS, 2457
SUP, 2467
SUPEXP, 2481
SUPXS, 2487
SUTS, 2497
SYMPOLY, 2654
TS, 2671
UFPS, 2789
ULS, 2796
ULSCONS, 2804
UP, 2828
UPXS, 2834
UPXSCONS, 2842
UPXSING, 2852
UTS, 2877
UTSZ, 2887
WP, 2949
XDPOLY, 2967

- XPBWPLYL, [2987](#)
- XPOLY, [2998](#)
- XPR, [3007](#)
- XPOLY, [3013](#)
- ZMOD, [1368](#)
- ? =?
 - FC, [931](#)
 - HTMLFORM, [1152](#)
 - OMERR, [1792](#)
 - SUBSPACE, [2614](#)
- ?~ =?
 - AFFPLPS, [7](#)
 - AFFSP, [9](#)
 - ALGFF, [28](#)
 - ALGSC, [15](#)
 - ALIST, [235](#)
 - AN, [35](#)
 - ANON, [38](#)
 - ANTISYM, [40](#)
 - ANY, [50](#)
 - ARRAY1, [1774](#)
 - ARRAY2, [2763](#)
 - ASTACK, [65](#)
 - ATTRBUT, [239](#)
 - AUTOMOR, [245](#)
 - BBTREE, [253](#)
 - BFUNCT, [265](#)
 - BINARY, [292](#)
 - BINFILE, [295](#)
 - BITS, [315](#)
 - BLHN, [317](#)
 - BLQT, [319](#)
 - BOOLEAN, [322](#)
 - BOP, [274](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - BSD, [286](#)
 - BSTREE, [302](#)
 - BTOURN, [306](#)
 - BTREE, [310](#)
 - CARD, [332](#)
 - CARTEN, [356](#)
 - CCLASS, [382](#)
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)
 - CHAR, [373](#)
 - CLIF, [402](#)
 - COLOR, [407](#)
 - COMM, [411](#)
 - COMPLEX, [419](#)
 - COMPPROP, [2624](#)
 - CONTFRAC, [445](#)
 - D01AJFA, [614](#)
 - D01AKFA, [617](#)
 - D01ALFA, [620](#)
 - D01AMFA, [623](#)
 - D01APFA, [629](#), [633](#)
 - D01ASFA, [636](#)
 - D01FCFA, [639](#)
 - D01GBFA, [643](#)
 - D01TRNS, [646](#)
 - D02BBFA, [651](#)
 - D02BHFA, [654](#)
 - D02CJFA, [658](#)
 - D02EJFA, [661](#)
 - D03EEFA, [665](#)
 - D03FAFA, [668](#)
 - D10ANFA, [626](#)
 - DBASE, [454](#)
 - DECIMAL, [465](#)
 - DEQUEUE, [512](#)
 - DERHAM, [530](#)
 - DFLOAT, [587](#)
 - DFMAT, [599](#)
 - DFVEC, [605](#)
 - DHMATRIX, [491](#)
 - DIRPROD, [547](#)
 - DIV, [575](#)
 - DLIST, [460](#)
 - DMP, [572](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DROPT, [608](#)
 - DSMP, [542](#)
 - DSTREE, [535](#)
 - E04DGFA, [747](#)
 - E04FDFA, [750](#)
 - E04GCFA, [754](#)
 - E04JAFa, [758](#)
 - E04MBFA, [762](#)
 - E04NAFA, [765](#)
 - E04UCFA, [769](#)

- EAB, 743
EMR, 702
EQ, 691
EQTBL, 699
EXIT, 707
EXPEXPAN, 712
EXPR, 724
EXPUPXS, 740
FAGROUP, 1004
FAMONOID, 1007
FARRAY, 886
FCOMP, 975
FDIV, 813
FEXPR, 947
FF, 820
FFCG, 825
FFCGP, 835
FFCGX, 830
FFNB, 860
FFNBP, 872
FFNBX, 866
FFP, 851
FFX, 846
FGROUP, 1010
FILE, 802
FLOAT, 908
FM, 1013
FM1, 1017
FMONOID, 1021
FNAME, 810
FNLA, 1027
FORMULA, 2346
FPARFRAC, 1040
FR, 786
FRAC, 986
FRIDEAL, 995
FRMOD, 1000
FSERIES, 978
FT, 971
FTEM, 967
FUNCTION, 1045
GCNAALG, 1065
GDMP, 1052
GMODPOL, 1059
GOPT, 1104
GOPT0, 1110
GPOLSET, 1074
GRIMAGE, 1094
GSERIES, 1090
GSTBL, 1079
GTSET, 1083
HACKPI, 1978
HASHTBL, 1120
HDMP, 1180
HDP, 1173
HEAP, 1134
HELLFDIV, 1183
HEXADEC, 1143
IAN, 1276
IARRAY1, 1244
IARRAY2, 1256
IBITS, 1199
IC, 1191
ICARD, 1193
IDEAL, 2081
IDPAG, 1202
IDPAM, 1206
IDPO, 1209
IDPOAM, 1212
IDPOAMS, 1215
IFAMON, 1287
IFARRAY, 1222
IFF, 1284
IARRAY2, 1290
ILIST, 1231
IMATRIX, 1239
INDE, 1218
INFCLSPS, 1271
INFCLSPT, 1265
INFORM, 1343
INT, 1362
INTABL, 1336
INTRVL, 1384
IPADIC, 1294
IPF, 1303
IR, 1375
ISTRING, 1249
ISUPS, 1311
ITAYLOR, 1338
IVECTOR, 1260
JORDAN, 223
KAFILE, 1414

KERNEL, 1404
LA, 1520
LAUPOL, 1422
LEXP, 1435
LIB, 1429
LIE, 229
LIST, 1504
LMDICT, 1515
LMOPS, 1509
LO, 1523
LODO, 1469
LODO1, 1479
LODO2, 1491
LPOLY, 1447
LSQM, 1456
LWORD, 1532
M3D, 2703
MAGMA, 1565
MATRIX, 1623
MCMPLX, 1543
MFLOAT, 1548
MINT, 1557
MKCHSET, 1570
MMLFORM, 1603
MODFIELD, 1638
MODMON, 1632
MODMONOM, 1644
MODOP, 1647, 1804
MODRING, 1641
MOEBIUS, 1654
MPOLY, 1682
MRING, 1659
MSET, 1670
MYEXPR, 1688
MYUP, 1695
NIPROB, 1746
NNI, 1739
NONE, 1737
NSDPS, 1703
NSMP, 1714
NSUP, 1729
OCT, 1765
ODEPROB, 1750
ODP, 1816
ODPOL, 1852
ODR, 1858
ODVAR, 1855
OFMONOID, 1829
OMENC, 1789
OMERRK, 1794
OMLO, 1806
ONECOMP, 1777
OPTPROB, 1753
ORDCOMP, 1810
ORESUP, 2492
OREUP, 2872
OSI, 1864
OUTFORM, 1868
OVAR, 1836
OWP, 1861
PACOFF, 2136
PACRAT, 2146
PADIC, 1881
PADICRAT, 1886
PADICRC, 1891
PALETTE, 1896
PATLRES, 1937
PATRES, 1940
PATTERN, 1928
PBWLB, 2054
PDEPROB, 1756
PENDTREE, 1945
PERM, 1949
PERMGRP, 1960
PF, 2105
PFR, 1914
PI, 2100
PLACES, 2019
PLACESPS, 2021
POINT, 2060
POLY, 2078
PR, 2092
PRIMARR, 2109
PRODUCT, 2113
PROJPL, 2117
PROJPLPS, 2119
PROJSP, 2122
PRTITION, 1923
QALGSET, 2157
QFORM, 2154
QUAT, 2167
QUEUE, 2184

- RADFF, [2194](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- REF, [2249](#)
- REGSET, [2286](#)
- RESRING, [2296](#)
- RESULT, [2301](#)
- RGCHAIN, [2255](#)
- RMATRIX, [2246](#)
- ROIRC, [2310](#)
- ROMAN, [2326](#)
- ROUTINE, [2331](#)
- RULE, [2305](#)
- RULECOLD, [2340](#)
- RULESET, [2342](#)
- SAE, [2400](#)
- SAOS, [2418](#)
- SD, [2572](#)
- SDPOL, [2386](#)
- SDVAR, [2389](#)
- SEG, [2359](#)
- SEGBIND, [2364](#)
- SET, [2372](#)
- SETMN, [2377](#)
- SEX, [2392](#)
- SEXOF, [2394](#)
- SHDP, [2509](#)
- SINT, [2412](#)
- SMP, [2423](#)
- SMTS, [2441](#)
- SPACE3, [2731](#)
- SPLNODE, [2512](#)
- SPLTREE, [2518](#)
- SQMATRIX, [2548](#)
- SREGSET, [2535](#)
- STACK, [2563](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- SUCH, [2627](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXP, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- SYMBOL, [2640](#)
- SYMPOLY, [2654](#)
- TABLE, [2664](#)
- TEX, [2677](#)
- TEXTFILE, [2692](#)
- TREE, [2741](#)
- TS, [2671](#)
- TUPLE, [2752](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UNISEG, [2896](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- VARIABLE, [2936](#)
- VECTOR, [2942](#)
- VIEW2d, [2770](#)
- VIEW3D, [2710](#)
- WP, [2949](#)
- WUTSET, [2959](#)
- XDPOLY, [2967](#)
- XPBWPOLYL, [2987](#)
- XPOLY, [2998](#)
- XPR, [3007](#)
- XPOLY, [3013](#)
- ZMOD, [1368](#)
- ?and?
 - BITS, [315](#)
 - BOOLEAN, [322](#)
 - IBITS, [1199](#)
 - OUTFORM, [1868](#)
- ?div?
 - OFMONOID, [1829](#)
 - OUTFORM, [1868](#)
- ?or?
 - BITS, [315](#)

- BOOLEAN, 322
- IBITS, 1199
- OUTFORM, 1868
- ?quo?
 - ALGFF, 28
 - AN, 35
 - BINARY, 292
 - BPADIC, 258
 - BPADICRT, 263
 - COMPLEX, 419
 - CONTFRAC, 445
 - DECIMAL, 465
 - DFLOAT, 587
 - EMR, 702
 - EXPEXPAN, 712
 - EXPR, 724
 - EXPUPXS, 740
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866
 - FFP, 851
 - FFX, 846
 - FLOAT, 908
 - FRAC, 986
 - GSERIES, 1090
 - HACKPI, 1978
 - HEXADEC, 1143
 - IAN, 1276
 - IFF, 1284
 - INT, 1362
 - IPADIC, 1294
 - IPF, 1303
 - LAUPOL, 1422
 - MCMPLEX, 1543
 - MFLOAT, 1548
 - MINT, 1557
 - MODFIELD, 1638
 - MODMON, 1632
 - MYEXPR, 1688
 - MYUP, 1695
 - NNI, 1739
 - NSDPS, 1703
 - NSUP, 1729
 - ODR, 1858
 - OUTFORM, 1868
 - PACOFF, 2136
 - PACRAT, 2146
 - PADIC, 1881
 - PADICRAT, 1886
 - PADICRC, 1891
 - PF, 2105
 - PFR, 1914
 - RADFF, 2194
 - RADIX, 2206
 - RECLOS, 2237
 - ROMAN, 2326
 - SAE, 2400
 - SINT, 2412
 - SULS, 2457
 - SUP, 2467
 - SUPEXP, 2481
 - SUPXS, 2487
 - ULS, 2796
 - ULSCONS, 2804
 - UP, 2828
 - UPXS, 2834
 - UPXSCONS, 2842
- ?rem?
 - ALGFF, 28
 - AN, 35
 - BINARY, 292
 - BPADIC, 258
 - BPADICRT, 263
 - COMPLEX, 419
 - CONTFRAC, 445
 - DECIMAL, 465
 - DFLOAT, 587
 - EMR, 702
 - EXPEXPAN, 712
 - EXPR, 724
 - EXPUPXS, 740
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866

- FFP, [851](#)
- FFX, [846](#)
- FLOAT, [908](#)
- FRAC, [986](#)
- GSERIES, [1090](#)
- HACKPI, [1978](#)
- HEXADEC, [1143](#)
- IAN, [1276](#)
- IFF, [1284](#)
- INT, [1362](#)
- IPADIC, [1294](#)
- IPF, [1303](#)
- LAUPOL, [1422](#)
- MCMPLX, [1543](#)
- MFLOAT, [1548](#)
- MINT, [1557](#)
- MODFIELD, [1638](#)
- MODMON, [1632](#)
- MYEXPR, [1688](#)
- MYUP, [1695](#)
- NNI, [1739](#)
- NSDPS, [1703](#)
- NSUP, [1729](#)
- ODR, [1858](#)
- OUTFORM, [1868](#)
- PACOFF, [2136](#)
- PACRAT, [2146](#)
- PADIC, [1881](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PF, [2105](#)
- PFR, [1914](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- ROMAN, [2326](#)
- SAE, [2400](#)
- SINT, [2412](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- #?
- CDFMAT, [426](#)
- CDFVEC, [432](#)
- DEQUEUE, [512](#)
- DFMAT, [599](#)
- DFVEC, [605](#)
- STREAM, [2582](#)
- #?
- ALIST, [235](#)
- ARRAY1, [1774](#)
- ARRAY2, [2763](#)
- ASTACK, [65](#)
- BBTREE, [253](#)
- BITS, [315](#)
- BSTREE, [302](#)
- BTOURN, [306](#)
- BTREE, [310](#)
- CCLASS, [382](#)
- DHMATRIX, [491](#)
- DIRPROD, [547](#)
- DLIST, [460](#)
- DPMM, [553](#)
- DPMO, [557](#)
- DSTREE, [535](#)
- EQTBL, [699](#)
- FARRAY, [886](#)
- GPOLSET, [1074](#)
- GSTBL, [1079](#)
- GTSET, [1083](#)
- HASHTBL, [1120](#)
- HDP, [1173](#)
- HEAP, [1134](#)
- IARRAY1, [1244](#)
- IARRAY2, [1256](#)
- IBITS, [1199](#)
- IFARRAY, [1222](#)
- IIARRAY2, [1290](#)
- ILIST, [1231](#)
- IMATRIX, [1239](#)
- INFORM, [1343](#)
- INTABL, [1336](#)
- ISTRING, [1249](#)
- IVECTOR, [1260](#)
- KAFILE, [1414](#)
- LIB, [1429](#)

- LIST, [1504](#)
- LMDICT, [1515](#)
- LSQM, [1456](#)
- M3D, [2703](#)
- MATRIX, [1623](#)
- MSET, [1670](#)
- NSDPS, [1703](#)
- ODP, [1816](#)
- PENDTREE, [1945](#)
- POINT, [2060](#)
- PRIMARR, [2109](#)
- QUEUE, [2184](#)
- REGSET, [2286](#)
- RESULT, [2301](#)
- RGCHAIN, [2255](#)
- RMATRIX, [2246](#)
- ROUTINE, [2331](#)
- SET, [2372](#)
- SEX, [2392](#)
- SEXOF, [2394](#)
- SHDP, [2509](#)
- SPLTREE, [2518](#)
- SQMATRIX, [2548](#)
- SREGSET, [2535](#)
- STACK, [2563](#)
- STBL, [2450](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- TREE, [2741](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- WUTSET, [2959](#)
- XPR, [3007](#)
- ^
- NOTTING, [1744](#)
- ^=
- NOTTING, [1744](#)
- ^?
- BITS, [315](#)
- BOOLEAN, [322](#)
- IBITS, [1199](#)
- ?
- SINT, [2412](#)
- ~?
- BITS, [315](#)
- BOOLEAN, [322](#)
- IBITS, [1199](#)
- 0
- ALGFF, [28](#)
- ALGSC, [15](#)
- AN, [35](#)
- ANTISYM, [40](#)
- BINARY, [292](#)
- BPADIC, [258](#)
- BPADICRT, [263](#)
- CARD, [332](#)
- CARTEN, [356](#)
- CLIF, [402](#)
- COMPLEX, [419](#)
- CONTFRAC, [445](#)
- DECIMAL, [465](#)
- DERHAM, [530](#)
- DFLOAT, [587](#)
- DIRPROD, [547](#)
- DIRRING, [564](#)
- DIV, [575](#)
- DMP, [572](#)
- DPMM, [553](#)
- DPMO, [557](#)
- DSMP, [542](#)
- EMR, [702](#)
- EQ, [691](#)
- EXPEXPAN, [712](#)
- EXPR, [724](#)
- EXPUPXS, [740](#)
- FAGROUP, [1004](#)
- FAMONOID, [1007](#)
- FDIV, [813](#)
- FEXPR, [947](#)
- FF, [820](#)
- FFCG, [825](#)
- FFCGP, [835](#)
- FFCGX, [830](#)
- FFNB, [860](#)
- FFNBP, [872](#)

- FFNBX, 866
FFP, 851
FFX, 846
FLOAT, 908
FM, 1013
FM1, 1017
FNLA, 1027
FR, 786
FRAC, 986
FSERIES, 978
GCNAALG, 1065
GDMP, 1052
GMODPOL, 1059
GSERIES, 1090
HACKPI, 1978
HDMP, 1180
HDP, 1173
HELLFDIV, 1183
HEXADEC, 1143
IAN, 1276
IDPAG, 1202
IDPAM, 1206
IDPOAM, 1212
IDPOAMS, 1215
IFAMON, 1287
IFF, 1284
INDE, 1218
INFORM, 1343
INT, 1362
INTRVL, 1384
IPADIC, 1294
IPF, 1303
IR, 1375
ISUPS, 1311
ITAYLOR, 1338
JORDAN, 223
LA, 1520
LAUPOL, 1422
LIE, 229
LO, 1523
LODO, 1469
LODO1, 1479
LODO2, 1491
LPOLY, 1447
LSQM, 1456
MCMPLX, 1543
MFLOAT, 1548
MINT, 1557
MODFIELD, 1638
MODMON, 1632
MODOP, 1647, 1804
MODRING, 1641
MPOLY, 1682
MRING, 1659
MYEXPR, 1688
MYUP, 1695
NNI, 1739
NSDPS, 1703
NSMP, 1714
NSUP, 1729
OCT, 1765
ODP, 1816
ODPOL, 1852
ODR, 1858
OMLO, 1806
ONECOMP, 1777
ORDCOMP, 1810
ORESUP, 2492
OREUP, 2872
OWP, 1861
PACOFF, 2136
PACRAT, 2146
PADIC, 1881
PADICRAT, 1886
PADICRC, 1891
PATTERN, 1928
PF, 2105
PFR, 1914
POLY, 2078
PR, 2092
PRODUCT, 2113
PRITITION, 1923
QFORM, 2154
QUAT, 2167
RADFF, 2194
RADIX, 2206
RECLOS, 2237
RESRING, 2296
RMATRIX, 2246
ROMAN, 2326
SAE, 2400
SD, 2572

- SDPOL, [2386](#)
 - SHDP, [2509](#)
 - SINT, [2412](#)
 - SMP, [2423](#)
 - SMTS, [2441](#)
 - SQMATRIX, [2548](#)
 - SULS, [2457](#)
 - SUP, [2467](#)
 - SUPEXP, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - SYMPOLY, [2654](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UP, [2828](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UPXSING, [2852](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
 - WP, [2949](#)
 - XDPOLY, [2967](#)
 - XPBWPLYL, [2987](#)
 - XPOLY, [2998](#)
 - XPR, [3007](#)
 - XPOLY, [3013](#)
 - ZMOD, [1368](#)
- 1
- ALGFF, [28](#)
 - AN, [35](#)
 - ANTISYM, [40](#)
 - AUTOMOR, [245](#)
 - BINARY, [292](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - CARD, [332](#)
 - CARTEN, [356](#)
 - CLIF, [402](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DERHAM, [530](#)
 - DFLOAT, [587](#)
 - DIRPROD, [547](#)
 - DIRRING, [564](#)
 - DMP, [572](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DSMP, [542](#)
 - EMR, [702](#)
 - EQ, [691](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FEXPR, [947](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FGROUP, [1010](#)
 - FLOAT, [908](#)
 - FMONOID, [1021](#)
 - FR, [786](#)
 - FRAC, [986](#)
 - FRIDEAL, [995](#)
 - FRMOD, [1000](#)
 - FSERIES, [978](#)
 - GDMP, [1052](#)
 - GSERIES, [1090](#)
 - HACKPI, [1978](#)
 - HDMP, [1180](#)
 - HDP, [1173](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - IFF, [1284](#)
 - INFORM, [1343](#)
 - INT, [1362](#)
 - INTRVL, [1384](#)
 - IPADIC, [1294](#)
 - IPF, [1303](#)
 - ISUPS, [1311](#)
 - ITAYLOR, [1338](#)
 - LA, [1520](#)
 - LAUPOL, [1422](#)
 - LEXP, [1435](#)

- LODO, 1469
LODO1, 1479
LODO2, 1491
LSQM, 1456
MCMPLX, 1543
MFLOAT, 1548
MINT, 1557
MODFIELD, 1638
MODMON, 1632
MODOP, 1647, 1804
MODRING, 1641
MOEBIUS, 1654
MPOLY, 1682
MRING, 1659
MYEXPR, 1688
MYUP, 1695
NNI, 1739
NOTTING, 1744
NSDPS, 1703
NSMP, 1714
NSUP, 1729
OCT, 1765
ODP, 1816
ODPOL, 1852
ODR, 1858
OFMONOID, 1829
OMLO, 1806
ONECOMP, 1777
ORDCOMP, 1810
ORESUP, 2492
OREUP, 2872
OWP, 1861
PACOFF, 2136
PACRAT, 2146
PADIC, 1881
PADICRAT, 1886
PADICRC, 1891
PATTERN, 1928
PBWLB, 2054
PERM, 1949
PF, 2105
PFR, 1914
PI, 2100
POLY, 2078
PR, 2092
PRODUCT, 2113
QUAT, 2167
RADFF, 2194
RADIX, 2206
RECLOS, 2237
RESRING, 2296
ROMAN, 2326
SAE, 2400
SDPOL, 2386
SHDP, 2509
SINT, 2412
SMP, 2423
SMTS, 2441
SQMATRIX, 2548
SULS, 2457
SUP, 2467
SUPEXP, 2481
SUPXS, 2487
SUTS, 2497
SYMPOLY, 2654
TS, 2671
UFPS, 2789
ULS, 2796
ULSCONS, 2804
UP, 2828
UPXS, 2834
UPXSCONS, 2842
UPXSING, 2852
UTS, 2877
UTSZ, 2887
WP, 2949
XDPOLY, 2967
XPBWPLYL, 2987
XPOLY, 2998
XPR, 3007
XPOLY, 3013
ZMOD, 1368
- any?
IFARRAY, 1222
- abs
BINARY, 292
BPADICRT, 263
COMPLEX, 419
DECIMAL, 465
DFLOAT, 587

- DIRPROD, [547](#)
- DPMM, [553](#)
- DPMO, [557](#)
- EXPEXPAN, [712](#)
- EXPR, [724](#)
- FEXPR, [947](#)
- FLOAT, [908](#)
- FRAC, [986](#)
- HDP, [1173](#)
- HEXADEC, [1143](#)
- INT, [1362](#)
- LA, [1520](#)
- MCMPLEX, [1543](#)
- MFLOAT, [1548](#)
- MINT, [1557](#)
- OCT, [1765](#)
- ODP, [1816](#)
- ONECOMP, [1777](#)
- ORDCOMP, [1810](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- QUAT, [2167](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- ROMAN, [2326](#)
- SHDP, [2509](#)
- SINT, [2412](#)
- SULS, [2457](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- absolutelyIrreducible?
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- acos
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FEXPR, [947](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLEX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- acosh
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLEX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- acoshIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- acosIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- acot
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)

- EFUPXS, [681](#)
- EXPR, [724](#)
- EXPUPXS, [740](#)
- FLOAT, [908](#)
- GSERIES, [1090](#)
- INTRVL, [1384](#)
- MCMPLX, [1543](#)
- SMTS, [2441](#)
- SULS, [2457](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- TS, [2671](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- acoth
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- acothIfCan
 - EFULS, [672](#)
- EFUPXS, [681](#)
- acotIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- ACPLOT, [1992](#)
 - coerce, [1992](#)
 - listBranches, [1992](#)
 - makeSketch, [1992](#)
 - refine, [1992](#)
 - xRange, [1992](#)
 - yRange, [1992](#)
- acsc
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- acsch
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLX, [1543](#)

SMTS, [2441](#)
 SULS, [2457](#)
 SUPEXP, [2481](#)
 SUPXS, [2487](#)
 SUTS, [2497](#)
 TS, [2671](#)
 UFPS, [2789](#)
 ULS, [2796](#)
 ULSCONS, [2804](#)
 UPXS, [2834](#)
 UPXSCONS, [2842](#)
 UTS, [2877](#)
 UTSZ, [2887](#)
 acschIfCan
 EFULS, [672](#)
 EFUPXS, [681](#)
 acscIfCan
 EFULS, [672](#)
 EFUPXS, [681](#)
 actualExtensionV
 IC, [1191](#)
 INFCLSPS, [1271](#)
 INFCLSPT, [1265](#)
 adaptive
 DROPT, [608](#)
 adaptive3D?
 PLOT3D, [2042](#)
 adaptive?
 PLOT, [2029](#)
 addBadValue
 PATTERN, [1928](#)
 additive?
 DIRRING, [564](#)
 addMatch
 PATRES, [1940](#)
 addMatchRestricted
 PATRES, [1940](#)
 addmod
 INT, [1362](#)
 MINT, [1557](#)
 ROMAN, [2326](#)
 SINT, [2412](#)
 addPoint
 SUBSPACE, [2614](#)
 addPoint2
 SUBSPACE, [2614](#)

addPointLast
 SUBSPACE, [2614](#)
 adjoint
 LODO, [1469](#)
 LODO1, [1479](#)
 LODO2, [1491](#)
 MODOP, [1647](#), [1804](#)
 AffinePlane, [4](#)
 AffinePlaneOverPseudoAlgebraicClosureOfFiniteField,
 [7](#)
 affinePoint
 AFFPLPS, [7](#)
 AFFSP, [9](#)
 AffineSpace, [9](#)
 AFFPL, [4](#)
 AFFPLPS, [7](#)
 ?..?, [7](#)
 ?=?, [7](#)
 ?~=?, [7](#)
 affinePoint, [7](#)
 coerce, [7](#)
 conjugate, [7](#)
 definingField, [7](#)
 degree, [7](#)
 hash, [7](#)
 latex, [7](#)
 list, [7](#)
 orbit, [7](#)
 origin, [7](#)
 pointValue, [7](#)
 rational?, [7](#)
 removeConjugate, [7](#)
 setelt, [7](#)
 AFFSP, [9](#)
 ?..?, [9](#)
 ?=?, [9](#)
 ?~=?, [9](#)
 affinePoint, [9](#)
 coerce, [9](#)
 conjugate, [9](#)
 definingField, [9](#)
 degree, [9](#)
 hash, [9](#)
 latex, [9](#)
 list, [9](#)
 orbit, [9](#)

- origin, 9
 - pointValue, 9
 - rational?, 9
 - removeConjugate, 9
 - setelt, 9
- airyAi
 - DFLOAT, 587
 - EXPR, 724
- airyBi
 - DFLOAT, 587
 - EXPR, 724
- Aleph
 - CARD, 332
- AlgebraGivenByStructuralConstants, 14
- algebraic?
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866
 - FFP, 851
 - FFX, 846
 - GTSET, 1083
 - IFF, 1284
 - IPF, 1303
 - PACOFF, 2136
 - PACRAT, 2146
 - PF, 2105
 - REGSET, 2286
 - RGCHAIN, 2255
 - SREGSET, 2535
 - WUTSET, 2959
- algebraicCoefficients?
 - REGSET, 2286
 - RGCHAIN, 2255
 - SREGSET, 2535
- AlgebraicFunctionField, 27
- AlgebraicNumber, 35
- algebraicOf
 - RECLOS, 2237
- algebraicVariables
 - GTSET, 1083
 - REGSET, 2286
 - RGCHAIN, 2255
- SREGSET, 2535
- WUTSET, 2959
- ALGFF, 27
 - , 28
 - ?**?, 28
 - ?*?, 28
 - ?+?, 28
 - ?-?, 28
 - ?/?, 28
 - ?=?, 28
 - ?^?, 28
 - ?~=?, 28
 - ?quo?, 28
 - ?rem?, 28
 - 0, 28
 - 1, 28
 - absolutelyIrreducible?, 28
 - algSplitSimple, 28
 - associates?, 28
 - basis, 28
 - branchPoint?, 28
 - branchPointAtInfinity?, 28
 - characteristic, 28
 - characteristicPolynomial, 28
 - charthRoot, 28
 - coerce, 28
 - complementaryBasis, 28
 - conditionP, 28
 - convert, 28
 - coordinates, 28
 - createPrimitiveElement, 28
 - D, 28
 - definingPolynomial, 28
 - derivationCoordinates, 28
 - differentiate, 28
 - discreteLog, 28
 - discriminant, 28
 - divide, 28
 - elliptic, 28
 - elt, 28
 - euclideanSize, 28
 - expressIdealMember, 28
 - exquo, 28
 - extendedEuclidean, 28
 - factor, 28
 - factorsOfCyclicGroupSize, 28

- gcd, 28
- gcdPolynomial, 28
- generator, 28
- genus, 28
- hash, 28
- hyperelliptic, 28
- index, 28
- init, 28
- integral?, 28
- integralAtInfinity?, 28
- integralBasis, 28
- integralBasisAtInfinity, 28
- integralCoordinates, 28
- integralDerivationMatrix, 28
- integralMatrix, 28
- integralMatrixAtInfinity, 28
- integralRepresents, 28
- inv, 28
- inverseIntegralMatrix, 28
- inverseIntegralMatrixAtInfinity, 28
- knownInfBasis, 28
- latex, 28
- lcm, 28
- lift, 28
- lookup, 28
- minimalPolynomial, 28
- multiEuclidean, 28
- nextItem, 28
- nonSingularModel, 28
- norm, 28
- normalizeAtInfinity, 28
- numberOfComponents, 28
- one?, 28
- order, 28
- prime?, 28
- primeFrobenius, 28
- primitive?, 28
- primitiveElement, 28
- primitivePart, 28
- principalIdeal, 28
- ramified?, 28
- ramifiedAtInfinity?, 28
- random, 28
- rank, 28
- rationalPoint?, 28
- rationalPoints, 28
- recip, 28
- reduce, 28
- reduceBasisAtInfinity, 28
- reducedSystem, 28
- regularRepresentation, 28
- representationType, 28
- represents, 28
- retract, 28
- retractIfCan, 28
- sample, 28
- singular?, 28
- singularAtInfinity?, 28
- size, 28
- sizeLess?, 28
- squareFree, 28
- squareFreePart, 28
- subtractIfCan, 28
- tableForDiscreteLogarithm, 28
- trace, 28
- traceMatrix, 28
- unit?, 28
- unitCanonical, 28
- unitNormal, 28
- yCoordinates, 28
- zero?, 28
- ALGSC, 14
- , 15
- ?**, 15
- ?*, 15
- ?+, 15
- ?-, 15
- ?., 15
- ?=, 15
- ?~=, 15
- 0, 15
- alternative?, 15
- antiAssociative?, 15
- antiCommutative?, 15
- antiCommutator, 15
- apply, 15
- associative?, 15
- associator, 15
- associatorDependence, 15
- basis, 15
- coerce, 15
- commutative?, 15

- commutator, 15
- conditionsForIdempotents, 15
- convert, 15
- coordinates, 15
- flexible?, 15
- hash, 15
- jacobiIdentity?, 15
- jordanAdmissible?, 15
- jordanAlgebra?, 15
- latex, 15
- leftAlternative?, 15
- leftCharacteristicPolynomial, 15
- leftDiscriminant, 15
- leftMinimalPolynomial, 15
- leftNorm, 15
- leftPower, 15
- leftRankPolynomial, 15
- leftRecip, 15
- leftRegularRepresentation, 15
- leftTrace, 15
- leftTraceMatrix, 15
- leftUnit, 15
- leftUnits, 15
- lieAdmissible?, 15
- lieAlgebra?, 15
- noncommutativeJordanAlgebra?, 15
- plenaryPower, 15
- powerAssociative?, 15
- rank, 15
- recip, 15
- represents, 15
- rightAlternative?, 15
- rightCharacteristicPolynomial, 15
- rightDiscriminant, 15
- rightMinimalPolynomial, 15
- rightNorm, 15
- rightPower, 15
- rightRankPolynomial, 15
- rightRecip, 15
- rightRegularRepresentation, 15
- rightTrace, 15
- rightTraceMatrix, 15
- rightUnit, 15
- rightUnits, 15
- sample, 15
- someBasis, 15
- structuralConstants, 15
- subtractIfCan, 15
- unit, 15
- zero?, 15
- algSplitSimple
 - ALGFF, 28
 - RADFF, 2194
- ALIST, 235
 - ?<?, 235
 - ?<=?, 235
 - ?>?, 235
 - ?>=?, 235
 - ?., 235
 - ?.first, 235
 - ?.last, 235
 - ?.rest, 235
 - ?.value, 235
 - ?=?, 235
 - ?~=?, 235
 - #?, 235
 - any?, 235
 - assoc, 235
 - bag, 235
 - child?, 235
 - children, 235
 - coerce, 235
 - concat, 235
 - construct, 235
 - convert, 235
 - copy, 235
 - copyInto, 235
 - count, 235
 - cycleEntry, 235
 - cycleLength, 235
 - cycleSplit, 235
 - cycleTail, 235
 - cyclic?, 235
 - delete, 235
 - dictionary, 235
 - distance, 235
 - elt, 235
 - empty, 235
 - empty?, 235
 - entries, 235
 - entry?, 235
 - eq?, 235

- eval, [235](#)
- every?, [235](#)
- explicitlyFinite?, [235](#)
- extract, [235](#)
- fill, [235](#)
- find, [235](#)
- first, [235](#)
- hash, [235](#)
- index?, [235](#)
- indices, [235](#)
- insert, [235](#)
- inspect, [235](#)
- key?, [235](#)
- keys, [235](#)
- last, [235](#)
- latex, [235](#)
- leaf?, [235](#)
- leaves, [235](#)
- less?, [235](#)
- list, [235](#)
- map, [235](#)
- max, [235](#)
- maxIndex, [235](#)
- member?, [235](#)
- members, [235](#)
- merge, [235](#)
- min, [235](#)
- minIndex, [235](#)
- more?, [235](#)
- new, [235](#)
- node?, [235](#)
- nodes, [235](#)
- parts, [235](#)
- position, [235](#)
- possiblyInfinite?, [235](#)
- qelt, [235](#)
- qsetelt, [235](#)
- reduce, [235](#)
- remove, [235](#)
- removeDuplicates, [235](#)
- rest, [235](#)
- reverse, [235](#)
- sample, [235](#)
- search, [235](#)
- second, [235](#)
- select, [235](#)
- setchildren, [235](#)
- setelt, [235](#)
- setfirst, [235](#)
- setlast, [235](#)
- setrest, [235](#)
- setvalue, [235](#)
- size?, [235](#)
- sort, [235](#)
- sorted?, [235](#)
- split, [235](#)
- swap, [235](#)
- table, [235](#)
- tail, [235](#)
- third, [235](#)
- value, [235](#)
- allDegrees
 - GOPT, [1104](#)
 - GOPT0, [1110](#)
- allRootsOf
 - RECLOS, [2237](#)
 - ROIRC, [2310](#)
- alphabetic
 - CCLASS, [382](#)
- alphabetic?
 - CHAR, [373](#)
- alphanumeric
 - CCLASS, [382](#)
- alphanumeric?
 - CHAR, [373](#)
- alternative?
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- AN, [35](#)
 - , [35](#)
 - ?<?, [35](#)
 - ?<=?, [35](#)
 - ?>?, [35](#)
 - ?>=?, [35](#)
 - ?**?, [35](#)
 - ?*?, [35](#)
 - ?+?, [35](#)
 - ?-?, [35](#)
 - ?/?, [35](#)

- [?=?, 35](#)
- [?^?, 35](#)
- [?~=?, 35](#)
- [?quo?, 35](#)
- [?rem?, 35](#)
- [0, 35](#)
- [1, 35](#)
- [associates?, 35](#)
- [belong?, 35](#)
- [box, 35](#)
- [characteristic, 35](#)
- [coerce, 35](#)
- [convert, 35](#)
- [D, 35](#)
- [definingPolynomial, 35](#)
- [denom, 35](#)
- [differentiate, 35](#)
- [distribute, 35](#)
- [divide, 35](#)
- [elt, 35](#)
- [euclideanSize, 35](#)
- [eval, 35](#)
- [even?, 35](#)
- [expressIdealMember, 35](#)
- [exquo, 35](#)
- [extendedEuclidean, 35](#)
- [factor, 35](#)
- [freeOf?, 35](#)
- [gcd, 35](#)
- [gcdPolynomial, 35](#)
- [hash, 35](#)
- [height, 35](#)
- [inv, 35](#)
- [is?, 35](#)
- [kernel, 35](#)
- [kernels, 35](#)
- [latex, 35](#)
- [lcm, 35](#)
- [mainKernel, 35](#)
- [map, 35](#)
- [max, 35](#)
- [min, 35](#)
- [minPoly, 35](#)
- [multiEuclidean, 35](#)
- [norm, 35](#)
- [nthRoot, 35](#)
- [numer, 35](#)
- [odd?, 35](#)
- [one?, 35](#)
- [operator, 35](#)
- [operators, 35](#)
- [paren, 35](#)
- [prime?, 35](#)
- [principalIdeal, 35](#)
- [recip, 35](#)
- [reduce, 35](#)
- [reducedSystem, 35](#)
- [retract, 35](#)
- [retractIfCan, 35](#)
- [rootOf, 35](#)
- [rootsOf, 35](#)
- [sample, 35](#)
- [sizeLess?, 35](#)
- [sqrt, 35](#)
- [squareFree, 35](#)
- [squareFreePart, 35](#)
- [subst, 35](#)
- [subtractIfCan, 35](#)
- [tower, 35](#)
- [unit?, 35](#)
- [unitCanonical, 35](#)
- [unitNormal, 35](#)
- [zero?, 35](#)
- [zeroOf, 35](#)
- [zerosOf, 35](#)
- An
 - [MODMON, 1632](#)
- AND
 - [SWITCH, 2629](#)
- And
 - [IBITS, 1199](#)
 - [SINT, 2412](#)
- ANON, 38
 - [?=?, 38](#)
 - [?~=?, 38](#)
 - [coerce, 38](#)
 - [hash, 38](#)
 - [latex, 38](#)
- AnonymousFunction, 38
- antiAssociative?
 - [ALGSC, 15](#)
 - [GCNAALG, 1065](#)

- JORDAN, [223](#)
- LIE, [229](#)
- LSQM, [1456](#)
- antiCommutative?
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
- antiCommutator
 - ALGSC, [15](#)
 - FNLA, [1027](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- ANTISYM, [40](#)
- ?, [40](#)
- ?**?, [40](#)
- ?*?, [40](#)
- ?+?, [40](#)
- ?-?, [40](#)
- ?=?, [40](#)
- ?^?, [40](#)
- ?~=?, [40](#)
- 0, [40](#)
- 1, [40](#)
- characteristic, [40](#)
- coefficient, [40](#)
- coerce, [40](#)
- degree, [40](#)
- exp, [40](#)
- generator, [40](#)
- hash, [40](#)
- homogeneous?, [40](#)
- latex, [40](#)
- leadingBasisTerm, [40](#)
- leadingCoefficient, [40](#)
- map, [40](#)
- one?, [40](#)
- recip, [40](#)
- reductum, [40](#)
- retract, [40](#)
- retractable?, [40](#)
- retractIfCan, [40](#)
- sample, [40](#)
- subtractIfCan, [40](#)
- zero?, [40](#)
- AntiSymm, [40](#)
- antisymmetric?
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - RMATRIX, [2246](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- ANY, [50](#)
- ?=?, [50](#)
- ?~=?, [50](#)
- any, [50](#)
- coerce, [50](#)
- dom, [50](#)
- domainOf, [50](#)
- hash, [50](#)
- latex, [50](#)
- obj, [50](#)
- objectOf, [50](#)
- showTypeInOutput, [50](#)
- Any, [50](#)
- any
 - ANY, [50](#)
- any?
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - ARRAY2, [2763](#)
 - ASTACK, [65](#)
 - BBTREE, [253](#)
 - BITS, [315](#)
 - BSTREE, [302](#)
 - BTourn, [306](#)
 - BTREE, [310](#)
 - CCLASS, [382](#)
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)
 - DEQUEUE, [512](#)
 - DFMAT, [599](#)
 - DFVEC, [605](#)
 - DHMATRIX, [491](#)

- DIRPROD, [547](#)
- DLIST, [460](#)
- DPMM, [553](#)
- DPMO, [557](#)
- DSTREE, [535](#)
- EQTBL, [699](#)
- FARRAY, [886](#)
- GPOLSET, [1074](#)
- GSTBL, [1079](#)
- GTSET, [1083](#)
- HASHTBL, [1120](#)
- HDP, [1173](#)
- HEAP, [1134](#)
- IARRAY1, [1244](#)
- IARRAY2, [1256](#)
- IBITS, [1199](#)
- IIARRAY2, [1290](#)
- ILIST, [1231](#)
- IMATRIX, [1239](#)
- INTABL, [1336](#)
- ISTRING, [1249](#)
- IVECTOR, [1260](#)
- KAFILE, [1414](#)
- LIB, [1429](#)
- LIST, [1504](#)
- LMDICT, [1515](#)
- LSQM, [1456](#)
- M3D, [2703](#)
- MATRIX, [1623](#)
- MSET, [1670](#)
- NSDPS, [1703](#)
- ODP, [1816](#)
- PENDTREE, [1945](#)
- POINT, [2060](#)
- PRIMARR, [2109](#)
- QUEUE, [2184](#)
- REGSET, [2286](#)
- RESULT, [2301](#)
- RGCHAIN, [2255](#)
- RMATRIX, [2246](#)
- ROUTINE, [2331](#)
- SET, [2372](#)
- SHDP, [2509](#)
- SPLTREE, [2518](#)
- SQMATRIX, [2548](#)
- SREGSET, [2535](#)
- STACK, [2563](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- TREE, [2741](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- WUTSET, [2959](#)
- append
 - LIST, [1504](#)
- appendPoint
 - GRIMAGE, [1094](#)
- apply
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - LSQM, [1456](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
- applyQuote
 - EXPR, [724](#)
 - MYEXPR, [1688](#)
- approximants
 - CONTFRAC, [445](#)
- approximate
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - EXPUPXS, [740](#)
 - GSERIES, [1090](#)
 - IPADIC, [1294](#)
 - ISUPS, [1311](#)
 - NSDPS, [1703](#)
 - PADIC, [1881](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)

- RECLOS, [2237](#)
- ROIRC, [2310](#)
- SULS, [2457](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- argscript
 - SYMBOL, [2640](#)
- argument
 - COMPLEX, [419](#)
 - FCOMP, [975](#)
 - KERNEL, [1404](#)
 - MCMPLX, [1543](#)
- argumentListOf
 - SYMS, [2696](#)
- arity
 - BOP, [274](#)
- ARRAY1, [1774](#)
 - ?<?, [1774](#)
 - ?<=?, [1774](#)
 - ?>?, [1774](#)
 - ?>=?, [1774](#)
 - ?.?, [1774](#)
 - ?=?, [1774](#)
 - ?~=?, [1774](#)
 - #?, [1774](#)
 - any?, [1774](#)
 - coerce, [1774](#)
 - concat, [1774](#)
 - construct, [1774](#)
 - convert, [1774](#)
 - copy, [1774](#)
 - copyInto, [1774](#)
 - count, [1774](#)
 - delete, [1774](#)
 - elt, [1774](#)
 - empty, [1774](#)
 - empty?, [1774](#)
 - entries, [1774](#)
 - entry?, [1774](#)
 - eq?, [1774](#)
 - eval, [1774](#)
 - every?, [1774](#)
 - fill, [1774](#)
 - find, [1774](#)
 - first, [1774](#)
 - hash, [1774](#)
 - index?, [1774](#)
 - indices, [1774](#)
 - insert, [1774](#)
 - latex, [1774](#)
 - less?, [1774](#)
 - map, [1774](#)
 - max, [1774](#)
 - maxIndex, [1774](#)
 - member?, [1774](#)
 - members, [1774](#)
 - merge, [1774](#)
 - min, [1774](#)
 - minIndex, [1774](#)
 - more?, [1774](#)
 - new, [1774](#)
 - oneDimensionalArray, [1774](#)
 - parts, [1774](#)
 - position, [1774](#)
 - qelt, [1774](#)
 - qsetelt, [1774](#)
 - reduce, [1774](#)
 - remove, [1774](#)
 - removeDuplicates, [1774](#)
 - reverse, [1774](#)
 - sample, [1774](#)
 - select, [1774](#)
 - setelt, [1774](#)
 - size?, [1774](#)
 - sort, [1774](#)
 - sorted?, [1774](#)
 - swap, [1774](#)
- ARRAY2, [2763](#)
 - ?=?, [2763](#)
 - ?~=?, [2763](#)
 - #?, [2763](#)
 - any?, [2763](#)
 - coerce, [2763](#)
 - column, [2763](#)
 - copy, [2763](#)

- count, [2763](#)
- elt, [2763](#)
- empty, [2763](#)
- empty?, [2763](#)
- eq?, [2763](#)
- eval, [2763](#)
- every?, [2763](#)
- fill, [2763](#)
- hash, [2763](#)
- latex, [2763](#)
- less?, [2763](#)
- map, [2763](#)
- maxColIndex, [2763](#)
- maxRowIndex, [2763](#)
- member?, [2763](#)
- members, [2763](#)
- minColIndex, [2763](#)
- minRowIndex, [2763](#)
- more?, [2763](#)
- ncols, [2763](#)
- new, [2763](#)
- nrows, [2763](#)
- parts, [2763](#)
- qelt, [2763](#)
- qsetelt, [2763](#)
- row, [2763](#)
- sample, [2763](#)
- setColumn, [2763](#)
- setelt, [2763](#)
- setRow, [2763](#)
- size?, [2763](#)
- ArrayStack, [65](#)
- arrayStack
 - ASTACK, [65](#)
- asec
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLEX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- asech
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLEX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- asechIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- asecIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- asin
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)

- EFUPXS, [681](#)
- EXPR, [724](#)
- EXPUPXS, [740](#)
- FEXPR, [947](#)
- FLOAT, [908](#)
- GSERIES, [1090](#)
- INTRVL, [1384](#)
- MCMPLEX, [1543](#)
- SMTS, [2441](#)
- SULS, [2457](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- TS, [2671](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- asinh
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLEX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- asinIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- ASP1, [71](#)
 - coerce, [71](#)
 - outputAsFortran, [71](#)
 - retract, [71](#)
 - retractIfCan, [71](#)
- Asp1, [71](#)
- ASP10, [75](#)
 - coerce, [75](#)
 - outputAsFortran, [75](#)
 - retract, [75](#)
 - retractIfCan, [75](#)
- Asp10, [75](#)
- ASP12, [80](#)
 - coerce, [80](#)
 - outputAsFortran, [80](#)
- Asp12, [80](#)
- ASP19, [85](#)
 - coerce, [85](#)
 - outputAsFortran, [85](#)
 - retract, [85](#)
 - retractIfCan, [85](#)
- Asp19, [85](#)
- ASP20, [92](#)
 - coerce, [93](#), [98](#)
 - outputAsFortran, [93](#), [98](#)
 - retract, [93](#), [98](#)
 - retractIfCan, [93](#), [98](#)
- Asp20, [92](#)
- ASP24, [98](#)
- Asp24, [98](#)
- ASP27, [102](#)
 - coerce, [102](#)
 - outputAsFortran, [102](#)
- Asp27, [102](#)
- ASP28, [108](#)
 - coerce, [108](#)
 - outputAsFortran, [108](#)
- Asp28, [108](#)
- ASP29, [114](#)
 - coerce, [114](#)
 - outputAsFortran, [114](#)

Asp29, 114
 ASP30, 118
 coerce, 118
 outputAsFortran, 118
 Asp30, 118
 ASP31, 122
 coerce, 122
 outputAsFortran, 122
 retract, 122
 retractIfCan, 122
 Asp31, 122
 ASP33, 127
 coerce, 127
 outputAsFortran, 127
 Asp33, 127
 ASP34, 130
 coerce, 130
 outputAsFortran, 130
 Asp34, 130
 ASP35, 134
 coerce, 134
 outputAsFortran, 134
 retract, 134
 retractIfCan, 134
 Asp35, 134
 ASP4, 140
 coerce, 140
 outputAsFortran, 140
 retract, 140
 retractIfCan, 140
 Asp4, 140
 ASP41, 144
 coerce, 144
 outputAsFortran, 144
 retract, 144
 retractIfCan, 144
 Asp41, 144
 ASP42, 151
 coerce, 151
 outputAsFortran, 151
 retract, 151
 retractIfCan, 151
 Asp42, 151
 ASP49, 158
 coerce, 158
 outputAsFortran, 158

 retract, 158
 retractIfCan, 158
 Asp49, 158
 ASP50, 164
 coerce, 164
 outputAsFortran, 164
 retract, 164
 retractIfCan, 164
 Asp50, 164
 ASP55, 169
 coerce, 170
 outputAsFortran, 170
 retract, 170
 retractIfCan, 170
 Asp55, 169
 ASP6, 176
 coerce, 176
 outputAsFortran, 176
 retract, 176
 retractIfCan, 176
 Asp6, 176
 ASP7, 181
 coerce, 181
 outputAsFortran, 181
 retract, 181
 retractIfCan, 181
 Asp7, 181
 ASP73, 185
 coerce, 186
 outputAsFortran, 186
 retract, 186
 retractIfCan, 186
 Asp73, 185
 ASP74, 191
 coerce, 191
 outputAsFortran, 191
 retract, 191
 retractIfCan, 191
 Asp74, 191
 ASP77, 196
 coerce, 197
 outputAsFortran, 197
 retract, 197
 retractIfCan, 197
 Asp77, 196
 ASP78, 201

- coerce, [201](#)
 - outputAsFortran, [201](#)
 - retract, [201](#)
 - retractIfCan, [201](#)
- Asp78, [201](#)
- ASP8, [206](#)
 - coerce, [206](#)
 - outputAsFortran, [206](#)
- Asp8, [206](#)
- ASP80, [211](#)
 - coerce, [211](#)
 - outputAsFortran, [211](#)
 - retract, [211](#)
 - retractIfCan, [211](#)
- Asp80, [211](#)
- ASP9, [216](#)
 - coerce, [216](#)
 - outputAsFortran, [216](#)
 - retract, [216](#)
 - retractIfCan, [216](#)
- Asp9, [216](#)
- assert
 - BOP, [274](#)
- assign
 - FC, [931](#)
 - OUTFORM, [1868](#)
- assoc
 - ALIST, [235](#)
- AssociatedJordanAlgebra, [222](#)
- AssociatedLieAlgebra, [228](#)
- associates?
 - ALGFF, [28](#)
 - AN, [35](#)
 - BINARY, [292](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - DIRRING, [564](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EMR, [702](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FLOAT, [908](#)
 - FR, [786](#)
 - FRAC, [986](#)
 - GDMP, [1052](#)
 - GSERIES, [1090](#)
 - HACKPI, [1978](#)
 - HDMP, [1180](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - IFF, [1284](#)
 - INT, [1362](#)
 - INTRVL, [1384](#)
 - IPADIC, [1294](#)
 - IPF, [1303](#)
 - ISUPS, [1311](#)
 - ITAYLOR, [1338](#)
 - LAUPOL, [1422](#)
 - MCMPLX, [1543](#)
 - MFLOAT, [1548](#)
 - MINT, [1557](#)
 - MODFIELD, [1638](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYEXPR, [1688](#)
 - MYUP, [1695](#)
 - NSDPS, [1703](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - ODR, [1858](#)
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
 - PADIC, [1881](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - PF, [2105](#)

- PFR, [1914](#)
- POLY, [2078](#)
- PR, [2092](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- ROMAN, [2326](#)
- SAE, [2400](#)
- SDPOL, [2386](#)
- SINT, [2412](#)
- SMP, [2423](#)
- SMTS, [2441](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- SYMPOLY, [2654](#)
- TS, [2671](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- AssociationList, [235](#)
- associative?
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- associator
 - ALGSC, [15](#)
 - FNLA, [1027](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- associatorDependence
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
- LIE, [229](#)
- LSQM, [1456](#)
- ASTACK, [65](#)
 - ?=?, [65](#)
 - ?~=?, [65](#)
 - #?, [65](#)
 - any?, [65](#)
 - arrayStack, [65](#)
 - bag, [65](#)
 - coerce, [65](#)
 - copy, [65](#)
 - count, [65](#)
 - depth, [65](#)
 - empty, [65](#)
 - empty?, [65](#)
 - eq?, [65](#)
 - eval, [65](#)
 - every?, [65](#)
 - extract, [65](#)
 - hash, [65](#)
 - insert, [65](#)
 - inspect, [65](#)
 - latex, [65](#)
 - less?, [65](#)
 - map, [65](#)
 - member?, [65](#)
 - members, [65](#)
 - more?, [65](#)
 - parts, [65](#)
 - pop, [65](#)
 - push, [65](#)
 - sample, [65](#)
 - size?, [65](#)
 - top, [65](#)
- atan
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FEXPR, [947](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLX, [1543](#)

- SMTS, [2441](#)
- SULS, [2457](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- TS, [2671](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- atanh
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLEX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- atanhIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- atanIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- atom?
 - INFORM, [1343](#)
 - SEX, [2392](#)
 - SEXOF, [2394](#)
- atoms
 - PATLRES, [1937](#)
- ATTRIBUT, [239](#)
 - ?=?, [239](#)
 - ?~=?, [239](#)
 - coerce, [239](#)
 - decrease, [239](#)
 - getButtonValue, [239](#)
 - hash, [239](#)
 - increase, [239](#)
 - latex, [239](#)
 - resetAttributeButtons, [239](#)
 - setAttributeButtonStep, [239](#)
 - setButtonValue, [239](#)
- AttributeButtons, [239](#)
- augment
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
- AUTOMOR, [245](#)
 - ?**?, [245](#)
 - ?*?, [245](#)
 - ?..?, [245](#)
 - ?/?, [245](#)
 - ?=?, [245](#)
 - ?^?, [245](#)
 - ?~=?, [245](#)
 - 1, [245](#)
 - coerce, [245](#)
 - commutator, [245](#)
 - conjugate, [245](#)
 - hash, [245](#)
 - inv, [245](#)
 - latex, [245](#)
 - morphism, [245](#)
 - one?, [245](#)
 - recip, [245](#)
 - sample, [245](#)
- Automorphism, [245](#)
- autoReduced?
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)

- axes
 - VIEW2d, 2770
 - VIEW3D, 2710
- back
 - DEQUEUE, 512
 - QUEUE, 2184
- backOldPos
 - IDEAL, 2081
- bag
 - ALIST, 235
 - ASTACK, 65
 - CCLASS, 382
 - DEQUEUE, 512
 - EQTBL, 699
 - GSTBL, 1079
 - HASHTBL, 1120
 - HEAP, 1134
 - INTABL, 1336
 - KAFIL, 1414
 - LIB, 1429
 - LMDICT, 1515
 - MSET, 1670
 - QUEUE, 2184
 - RESULT, 2301
 - ROUTINE, 2331
 - SET, 2372
 - STACK, 2563
 - STBL, 2450
 - STRTBL, 2610
 - TABLE, 2664
- BalancedBinaryTree, 252
- balancedBinaryTree
 - BBTREE, 253
- BalancedPAdicInteger, 258
- BalancedPAdicRational, 262
- base
 - DFLOAT, 587
 - FLOAT, 908
 - INT, 1362
 - MFLOAT, 1548
 - MINT, 1557
 - PERMGRP, 1960
 - ROMAN, 2326
 - SINT, 2412
- BasicFunctions, 265
- BasicOperator, 273
- basicSet
 - GTSET, 1083
 - REGSET, 2286
 - RGCHAIN, 2255
 - SREGSET, 2535
 - WUTSET, 2959
- BasicStochasticDifferential, 285
- basis
 - ALGFF, 28
 - ALGSC, 15
 - COMPLEX, 419
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866
 - FFP, 851
 - FFX, 846
 - FRIDEAL, 995
 - FRMOD, 1000
 - GCNAALG, 1065
 - IFF, 1284
 - IPF, 1303
 - JORDAN, 223
 - LIE, 229
 - LSQM, 1456
 - MCMPLEX, 1543
 - PF, 2105
 - RADFF, 2194
 - SAE, 2400
- BBTREE, 252
 - ?left, 253
 - ?right, 253
 - ?value, 253
 - ?=?, 253
 - ?~=?, 253
 - #?, 253
 - any?, 253
 - balancedBinaryTree, 253
 - child?, 253
 - children, 253
 - coerce, 253
 - copy, 253

- count, 253
- cyclic?, 253
- distance, 253
- empty, 253
- empty?, 253
- eq?, 253
- eval, 253
- every?, 253
- hash, 253
- latex, 253
- leaf?, 253
- leaves, 253
- left, 253
- less?, 253
- map, 253
- mapDown, 253
- mapUp, 253
- member?, 253
- members, 253
- more?, 253
- node, 253
- node?, 253
- nodes, 253
- parts, 253
- right, 253
- sample, 253
- setchildren, 253
- setelt, 253
- setleaves, 253
- setleft, 253
- setright, 253
- setvalue, 253
- size?, 253
- value, 253
- belong?
 - AN, 35
 - EXPR, 724
 - FEXPR, 947
 - IAN, 1276
 - MYEXPR, 1688
- besselI
 - DFLOAT, 587
 - EXPR, 724
- besselJ
 - DFLOAT, 587
 - EXPR, 724
- besselK
 - DFLOAT, 587
 - EXPR, 724
- besselY
 - DFLOAT, 587
 - EXPR, 724
- Beta
 - DFLOAT, 587
 - EXPR, 724
- bfEntry
 - BFUNCT, 265
- bfKeys
 - BFUNCT, 265
- BFUNCT, 265
 - ?=?, 265
 - ?~=?, 265
 - bfEntry, 265
 - bfKeys, 265
 - coerce, 265
 - hash, 265
 - latex, 265
- BINARY, 292
 - , 292
 - ?<?, 292
 - ?<=?, 292
 - ?>?, 292
 - ?>=?, 292
 - ?**?, 292
 - ?*?, 292
 - ?+?, 292
 - ?-?, 292
 - ?., 292
 - ?/?, 292
 - ?=, 292
 - ?^?, 292
 - ?~=?, 292
 - ?quo?, 292
 - ?rem?, 292
 - 0, 292
 - 1, 292
 - abs, 292
 - associates?, 292
 - binary, 292
 - ceiling, 292
 - characteristic, 292
 - charthRoot, 292

- coerce, 292
- conditionP, 292
- convert, 292
- D, 292
- denom, 292
- denominator, 292
- differentiate, 292
- divide, 292
- euclideanSize, 292
- eval, 292
- expressIdealMember, 292
- exquo, 292
- extendedEuclidean, 292
- factor, 292
- factorPolynomial, 292
- factorSquareFreePolynomial, 292
- floor, 292
- fractionPart, 292
- gcd, 292
- gcdPolynomial, 292
- hash, 292
- init, 292
- inv, 292
- latex, 292
- lcm, 292
- map, 292
- max, 292
- min, 292
- multiEuclidean, 292
- negative?, 292
- nextItem, 292
- numer, 292
- numerator, 292
- one?, 292
- patternMatch, 292
- positive?, 292
- prime?, 292
- principalIdeal, 292
- random, 292
- recip, 292
- reducedSystem, 292
- retract, 292
- retractIfCan, 292
- sample, 292
- sign, 292
- sizeLess?, 292
- solveLinearPolynomialEquation, 292
- squareFree, 292
- squareFreePart, 292
- squareFreePolynomial, 292
- subtractIfCan, 292
- unit?, 292
- unitCanonical, 292
- unitNormal, 292
- wholePart, 292
- zero?, 292
- binary
 - BINARY, 292
 - INFORM, 1343
- BinaryExpansion, 292
- BinaryFile, 295
- BinarySearchTree, 302
- binarySearchTree
 - BSTREE, 302
- BinaryTournament, 306
- binaryTournament
 - BTOURN, 306
- BinaryTree, 310
- binaryTree
 - BTREE, 310
- BINFILE, 295
 - ?=?, 295
 - ?~=?, 295
 - close, 295
 - coerce, 295
 - hash, 295
 - iomode, 295
 - latex, 295
 - name, 295
 - open, 295
 - position, 295
 - read, 295
 - readIfCan, 295
 - reopen, 295
 - write, 295
- binomial
 - EXPR, 724
 - INT, 1362
 - MINT, 1557
 - MYEXPR, 1688
 - OUTFORM, 1868
 - ROMAN, 2326

- SINT, [2412](#)
- binomThmExpt
 - DMP, [572](#)
 - DSMP, [542](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - POLY, [2078](#)
 - PR, [2092](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - SYMPOLY, [2654](#)
 - UP, [2828](#)
 - UPXSSING, [2852](#)
- birth
 - SUBSPACE, [2614](#)
- bit?
 - INT, [1362](#)
 - MINT, [1557](#)
 - ROMAN, [2326](#)
 - SINT, [2412](#)
- BITS, [314](#)
 - ?/ΓE30F?, [315](#)
 - ?<?, [315](#)
 - ?<=?, [315](#)
 - ?>?, [315](#)
 - ?>=?, [315](#)
 - ?ΓE30F/?, [315](#)
 - ?., [315](#)
 - ?=?, [315](#)
 - ?~=?, [315](#)
 - ?and?, [315](#)
 - ?or?, [315](#)
 - #?, [315](#)
 - ^?, [315](#)
 - ~?, [315](#)
 - any?, [315](#)
 - bits, [315](#)
 - coerce, [315](#)
 - concat, [315](#)
 - construct, [315](#)
 - convert, [315](#)
 - copy, [315](#)
 - copyInto, [315](#)
 - count, [315](#)
 - delete, [315](#)
 - elt, [315](#)
 - empty, [315](#)
 - empty?, [315](#)
 - entries, [315](#)
 - entry?, [315](#)
 - eq?, [315](#)
 - eval, [315](#)
 - every?, [315](#)
 - fill, [315](#)
 - find, [315](#)
 - first, [315](#)
 - hash, [315](#)
 - index?, [315](#)
 - indices, [315](#)
 - insert, [315](#)
 - latex, [315](#)
 - less?, [315](#)
 - map, [315](#)
 - max, [315](#)
 - maxIndex, [315](#)
 - member?, [315](#)
 - members, [315](#)
 - merge, [315](#)
 - min, [315](#)
 - minIndex, [315](#)
 - more?, [315](#)
 - nand, [315](#)
 - new, [315](#)
 - nor, [315](#)
 - not?, [315](#)
 - parts, [315](#)
 - position, [315](#)
 - qelt, [315](#)
 - qsetelt, [315](#)
 - reduce, [315](#)
 - remove, [315](#)
 - removeDuplicates, [315](#)
 - reverse, [315](#)
 - sample, [315](#)

- setelt, [315](#)
 - size?, [315](#)
 - sort, [315](#)
 - sorted?, [315](#)
 - swap, [315](#)
 - xor, [315](#)
- Bits, [314](#)
- bits
 - BITS, [315](#)
 - DFLOAT, [587](#)
 - FLOAT, [908](#)
 - MFLOAT, [1548](#)
- blankSeparate
 - OUTFORM, [1868](#)
- BLHN, [317](#)
 - ?=?, [317](#)
 - ?~=?, [317](#)
 - chartCoord, [317](#)
 - coerce, [317](#)
 - createHN, [317](#)
 - excepCoord, [317](#)
 - hash, [317](#)
 - infClsPt?, [317](#)
 - latex, [317](#)
 - quotValuation, [317](#)
 - ramifMult, [317](#)
 - transCoord, [317](#)
 - type, [317](#)
- block
 - FC, [931](#)
- BlowUpWithHamburgerNoether, [317](#)
- BlowUpWithQuadTrans, [319](#)
- BLQT, [319](#)
 - ?=?, [319](#)
 - ?~=?, [319](#)
 - chartCoord, [319](#)
 - coerce, [319](#)
 - createHN, [319](#)
 - excepCoord, [319](#)
 - hash, [319](#)
 - infClsPt?, [319](#)
 - latex, [319](#)
 - quotValuation, [319](#)
 - ramifMult, [319](#)
 - transCoord, [319](#)
 - type, [319](#)
- blue
 - COLOR, [407](#)
- BOOLEAN, [322](#)
 - ?/ΓE30F?, [322](#)
 - ?<?, [322](#)
 - ?<=?, [322](#)
 - ?>?, [322](#)
 - ?>=?, [322](#)
 - ?ΓE30F/?, [322](#)
 - ?=?, [322](#)
 - ?~=?, [322](#)
 - ?and?, [322](#)
 - ?or?, [322](#)
 - ^?, [322](#)
 - ~?, [322](#)
 - coerce, [322](#)
 - convert, [322](#)
 - false, [322](#)
 - hash, [322](#)
 - implies, [322](#)
 - index, [322](#)
 - latex, [322](#)
 - lookup, [322](#)
 - max, [322](#)
 - min, [322](#)
 - nand, [322](#)
 - nor, [322](#)
 - not?, [322](#)
 - random, [322](#)
 - size, [322](#)
 - test, [322](#)
 - true, [322](#)
 - xor, [322](#)
- Boolean, [322](#)
- BOP, [273](#)
 - ?<?, [274](#)
 - ?<=?, [274](#)
 - ?>?, [274](#)
 - ?>=?, [274](#)
 - ?=?, [274](#)
 - ?~=?, [274](#)
 - arity, [274](#)
 - assert, [274](#)
 - coerce, [274](#)
 - comparison, [274](#)
 - copy, [274](#)

- deleteProperty, 274
- display, 274
- equality, 274
- has?, 274
- hash, 274
- input, 274
- is?, 274
- latex, 274
- max, 274
- min, 274
- name, 274
- nary?, 274
- nullary?, 274
- operator, 274
- properties, 274
- property, 274
- setProperties, 274
- setProperty, 274
- unary?, 274
- weight, 274
- box
 - AN, 35
 - EXPR, 724
 - FEXPR, 947
 - IAN, 1276
 - MYEXPR, 1688
 - OUTFORM, 1868
- BPADIC, 258
 - , 258
 - ?**?, 258
 - ?*?, 258
 - ?+?, 258
 - ?-?, 258
 - ?=?, 258
 - ?^?, 258
 - ?~=?, 258
 - ?quo?, 258
 - ?rem?, 258
 - 0, 258
 - 1, 258
 - approximate, 258
 - associates?, 258
 - characteristic, 258
 - coerce, 258
 - complete, 258
 - digits, 258
 - divide, 258
 - euclideanSize, 258
 - expressIdealMember, 258
 - exquo, 258
 - extend, 258
 - extendedEuclidean, 258
 - gcd, 258
 - gcdPolynomial, 258
 - hash, 258
 - latex, 258
 - lcm, 258
 - moduloP, 258
 - modulus, 258
 - multiEuclidean, 258
 - one?, 258
 - order, 258
 - principalIdeal, 258
 - quotientByP, 258
 - recip, 258
 - root, 258
 - sample, 258
 - sizeLess?, 258
 - sqrt, 258
 - subtractIfCan, 258
 - unit?, 258
 - unitCanonical, 258
 - unitNormal, 258
 - zero?, 258
- BPADICRT, 262
 - , 263
 - ?<?, 263
 - ?<=?, 263
 - ?>?, 263
 - ?>=?, 263
 - ?**?, 263
 - ?*?, 263
 - ?+?, 263
 - ?-?, 263
 - ?., 263
 - ?/?, 263
 - ?=?, 263
 - ?^?, 263
 - ?~=?, 263
 - ?quo?, 263
 - ?rem?, 263
 - 0, 263

- 1, 263
- abs, 263
- approximate, 263
- associates?, 263
- ceiling, 263
- characteristic, 263
- charthRoot, 263
- coerce, 263
- conditionP, 263
- continuedFraction, 263
- convert, 263
- D, 263
- denom, 263
- denominator, 263
- differentiate, 263
- divide, 263
- euclideanSize, 263
- eval, 263
- expressIdealMember, 263
- exquo, 263
- extendedEuclidean, 263
- factor, 263
- factorPolynomial, 263
- factorSquareFreePolynomial, 263
- floor, 263
- fractionPart, 263
- gcd, 263
- gcdPolynomial, 263
- hash, 263
- init, 263
- inv, 263
- latex, 263
- lcm, 263
- map, 263
- max, 263
- min, 263
- multiEuclidean, 263
- negative?, 263
- nextItem, 263
- numer, 263
- numerator, 263
- one?, 263
- patternMatch, 263
- positive?, 263
- prime?, 263
- principalIdeal, 263
- random, 263
- recip, 263
- reducedSystem, 263
- removeZeroes, 263
- retract, 263
- retractIfCan, 263
- sample, 263
- sign, 263
- sizeLess?, 263
- solveLinearPolynomialEquation, 263
- squareFree, 263
- squareFreePart, 263
- squareFreePolynomial, 263
- subtractIfCan, 263
- unit?, 263
- unitCanonical, 263
- unitNormal, 263
- wholePart, 263
- zero?, 263
- brace
 - CCLASS, 382
 - MSET, 1670
 - OUTFORM, 1868
 - SET, 2372
- bracket
 - OUTFORM, 1868
- branchPoint?
 - ALGFF, 28
 - RADFF, 2194
- branchPointAtInfinity?
 - ALGFF, 28
 - RADFF, 2194
- bright
 - PALETTE, 1896
- BSD, 285
 - ?<?, 286
 - ?<=?, 286
 - ?=?, 286
 - ?>?, 286
 - ?>=?, 286
 - ?~=?, 286
 - coerce, 286
 - convert, 286
 - convertIfCan, 286
 - copyBSD, 286
 - copyIto, 286

- d, 286
- getSmgl, 286
- hash, 286
- introduce, 286
- latex, 286
- max, 286
- min, 286
- BSTREE, 302
 - ?.left, 302
 - ?.right, 302
 - ?.value, 302
 - ?=?, 302
 - ?~=?, 302
 - #?, 302
 - any?, 302
 - binarySearchTree, 302
 - child?, 302
 - children, 302
 - coerce, 302
 - copy, 302
 - count, 302
 - cyclic?, 302
 - distance, 302
 - empty, 302
 - empty?, 302
 - eq?, 302
 - eval, 302
 - every?, 302
 - hash, 302
 - insert, 302
 - insertRoot, 302
 - latex, 302
 - leaf?, 302
 - leaves, 302
 - left, 302
 - less?, 302
 - map, 302
 - member?, 302
 - members, 302
 - more?, 302
 - node, 302
 - node?, 302
 - nodes, 302
 - parts, 302
 - right, 302
 - sample, 302
 - setchildren, 302
 - setelt, 302
 - setleft, 302
 - setright, 302
 - setvalue, 302
 - size?, 302
 - split, 302
 - value, 302
- BTOURN, 306
 - ?.left, 306
 - ?.right, 306
 - ?.value, 306
 - ?=?, 306
 - ?~=?, 306
 - #?, 306
 - any?, 306
 - binaryTournament, 306
 - child?, 306
 - children, 306
 - coerce, 306
 - copy, 306
 - count, 306
 - cyclic?, 306
 - distance, 306
 - empty, 306
 - empty?, 306
 - eq?, 306
 - eval, 306
 - every?, 306
 - hash, 306
 - insert, 306
 - latex, 306
 - leaf?, 306
 - leaves, 306
 - left, 306
 - less?, 306
 - map, 306
 - member?, 306
 - members, 306
 - more?, 306
 - node, 306
 - node?, 306
 - nodes, 306
 - parts, 306
 - right, 306
 - sample, 306

- setchildren, 306
- setelt, 306
- setleft, 306
- setright, 306
- setvalue, 306
- size?, 306
- value, 306
- BTREE, 310
 - ?left, 310
 - ?right, 310
 - ?value, 310
 - ?=?, 310
 - ?~=?, 310
 - #?, 310
 - any?, 310
 - binaryTree, 310
 - child?, 310
 - children, 310
 - coerce, 310
 - copy, 310
 - count, 310
 - cyclic?, 310
 - distance, 310
 - empty, 310
 - empty?, 310
 - eq?, 310
 - eval, 310
 - every?, 310
 - hash, 310
 - latex, 310
 - leaf?, 310
 - leaves, 310
 - left, 310
 - less?, 310
 - map, 310
 - member?, 310
 - members, 310
 - more?, 310
 - node, 310
 - node?, 310
 - nodes, 310
 - parts, 310
 - right, 310
 - sample, 310
 - setchildren, 310
 - setelt, 310
 - setleft, 310
 - setright, 310
 - setvalue, 310
 - size?, 310
 - value, 310
- build
 - GMODPOL, 1059
- BY
 - SEG, 2359
 - UNISEG, 2896
- cAcos
 - ISUPS, 1311
- cAcosh
 - ISUPS, 1311
- cAcot
 - ISUPS, 1311
- cAcoth
 - ISUPS, 1311
- cAcsc
 - ISUPS, 1311
- cAcsch
 - ISUPS, 1311
- call
 - FC, 931
- car
 - INFORM, 1343
 - SEX, 2392
 - SEXOF, 2394
- CARD, 332
 - ?<?, 332
 - ?<=?, 332
 - ?>?, 332
 - ?>=?, 332
 - ?**?, 332
 - ?*?, 332
 - ?+?, 332
 - ?-?, 332
 - ?=?, 332
 - ?^?, 332
 - ?~=?, 332
 - 0, 332
 - 1, 332
 - Aleph, 332
 - coerce, 332
 - countable?, 332

- finite?, 332
- generalizedContinuumHypothesisAssumedcAsec 332
- generalizedContinuumHypothesisAssumedcAsech 332
- hash, 332
- latex, 332
- max, 332
- min, 332
- one?, 332
- recip, 332
- retract, 332
- retractIfCan, 332
- sample, 332
- zero?, 332
- cardinality
 - CCLASS, 382
 - SET, 2372
- CardinalNumber, 332
- CARTEN, 356
 - , 356
 - ?*, 356
 - ?+, 356
 - ?-, 356
 - ?.?, 356
 - ?=, 356
 - ?~=, 356
 - 0, 356
 - 1, 356
 - coerce, 356
 - contract, 356
 - degree, 356
 - elt, 356
 - hash, 356
 - kroneckerDelta, 356
 - latex, 356
 - leviCivitaSymbol, 356
 - product, 356
 - rank, 356
 - ravel, 356
 - reindex, 356
 - retract, 356
 - retractIfCan, 356
 - sample, 356
 - transpose, 356
 - unravel, 356
 - CartesianTensor, 356
 - ISUPS, 1311
 - ISUPS, 1311
 - cAsin
 - ISUPS, 1311
 - cAsinh
 - ISUPS, 1311
 - cAtan
 - ISUPS, 1311
 - cAtanh
 - ISUPS, 1311
 - CCLASS, 381
 - ?<?, 382
 - ?=, 382
 - ?~=, 382
 - #?, 382
 - alphabetic, 382
 - alphanumeric, 382
 - any?, 382
 - bag, 382
 - brace, 382
 - cardinality, 382
 - charClass, 382
 - coerce, 382
 - complement, 382
 - construct, 382
 - convert, 382
 - copy, 382
 - count, 382
 - dictionary, 382
 - difference, 382
 - digit, 382
 - empty, 382
 - empty?, 382
 - eq?, 382
 - eval, 382
 - every?, 382
 - extract, 382
 - find, 382
 - hash, 382
 - hexDigit, 382
 - index, 382
 - insert, 382
 - inspect, 382

- intersect, 382
- latex, 382
- less?, 382
- lookup, 382
- lowerCase, 382
- map, 382
- max, 382
- member?, 382
- members, 382
- min, 382
- more?, 382
- parts, 382
- random, 382
- reduce, 382
- remove, 382
- removeDuplicates, 382
- sample, 382
- select, 382
- set, 382
- size, 382
- size?, 382
- subset?, 382
- symmetricDifference, 382
- union, 382
- universe, 382
- upperCase, 382
- cCos
 - ISUPS, 1311
- cCosh
 - ISUPS, 1311
- cCot
 - ISUPS, 1311
- cCoth
 - ISUPS, 1311
- cCsc
 - ISUPS, 1311
- cCsch
 - ISUPS, 1311
- CDFMAT, 426
 - , 426
 - ?**?, 426
 - ?*?, 426
 - ?+?, 426
 - ?-?, 426
 - ?/?, 426
 - ?=?, 426
 - ?~=?, 426
 - #?, 426
 - antisymmetric?, 426
 - any?, 426
 - coerce, 426
 - column, 426
 - columnSpace, 426
 - copy, 426
 - count, 426
 - determinant, 426
 - diagonal?, 426
 - diagonalMatrix, 426
 - elt, 426
 - empty, 426
 - empty?, 426
 - eq?, 426
 - eval, 426
 - every?, 426
 - exquo, 426
 - fill, 426
 - hash, 426
 - horizConcat, 426
 - inverse, 426
 - latex, 426
 - less?, 426
 - listOfLists, 426
 - map, 426
 - matrix, 426
 - maxColIndex, 426
 - maxRowIndex, 426
 - member?, 426
 - members, 426
 - minColIndex, 426
 - minordet, 426
 - minRowIndex, 426
 - more?, 426
 - ncols, 426
 - new, 426
 - nrows, 426
 - nullity, 426
 - nullSpace, 426
 - parts, 426
 - pfaffian, 426
 - qelt, 426
 - qnew, 426
 - qsetelt, 426

- rank, [426](#)
- row, [426](#)
- rowEchelon, [426](#)
- sample, [426](#)
- scalarMatrix, [426](#)
- setColumn, [426](#)
- setelt, [426](#)
- setRow, [426](#)
- setsubMatrix, [426](#)
- size?, [426](#)
- square?, [426](#)
- squareTop, [426](#)
- subMatrix, [426](#)
- swapColumns, [426](#)
- swapRows, [426](#)
- symmetric?, [426](#)
- transpose, [426](#)
- vertConcat, [426](#)
- zero, [426](#)
- CDFVEC, [432](#)
- ?, [432](#)
- ?<?, [432](#)
- ?<=?, [432](#)
- ?>?, [432](#)
- ?>=?, [432](#)
- ?*?, [432](#)
- ?+?, [432](#)
- ?-?, [432](#)
- ?., [432](#)
- ?=?, [432](#)
- ?~=?, [432](#)
- #?, [432](#)
- any?, [432](#)
- coerce, [432](#)
- concat, [432](#)
- construct, [432](#)
- convert, [432](#)
- copy, [432](#)
- copyInto, [432](#)
- count, [432](#)
- cross, [432](#)
- delete, [432](#)
- dot, [432](#)
- elt, [432](#)
- empty, [432](#)
- empty?, [432](#)
- entries, [432](#)
- entry?, [432](#)
- eq?, [432](#)
- eval, [432](#)
- every?, [432](#)
- fill, [432](#)
- find, [432](#)
- first, [432](#)
- hash, [432](#)
- index?, [432](#)
- indices, [432](#)
- insert, [432](#)
- latex, [432](#)
- length, [432](#)
- less?, [432](#)
- magnitude, [432](#)
- map, [432](#)
- max, [432](#)
- maxIndex, [432](#)
- member?, [432](#)
- members, [432](#)
- merge, [432](#)
- min, [432](#)
- minIndex, [432](#)
- more?, [432](#)
- new, [432](#)
- outerProduct, [432](#)
- parts, [432](#)
- position, [432](#)
- qelt, [432](#)
- qnew, [432](#)
- qsetelt, [432](#)
- reduce, [432](#)
- remove, [432](#)
- removeDuplicates, [432](#)
- reverse, [432](#)
- sample, [432](#)
- select, [432](#)
- setelt, [432](#)
- size?, [432](#)
- sort, [432](#)
- sorted?, [432](#)
- swap, [432](#)
- vector, [432](#)
- zero, [432](#)
- cdr

- INFORM, 1343
- SEX, 2392
- SEXOF, 2394
- ceiling
 - BINARY, 292
 - BPADICRT, 263
 - DECIMAL, 465
 - DFLOAT, 587
 - EXPEXPAN, 712
 - FLOAT, 908
 - FRAC, 986
 - HEXADEC, 1143
 - MFLOAT, 1548
 - PADICRAT, 1886
 - PADICRC, 1891
 - RADIX, 2206
 - SULS, 2457
 - ULS, 2796
 - ULSCONS, 2804
- center
 - EXPUPXS, 740
 - GSERIES, 1090
 - ISUPS, 1311
 - NSDPS, 1703
 - OUTFORM, 1868
 - SULS, 2457
 - SUPXS, 2487
 - SUTS, 2497
 - UFPS, 2789
 - ULS, 2796
 - ULSCONS, 2804
 - UPXS, 2834
 - UPXSCONS, 2842
 - UTS, 2877
 - UTSZ, 2887
- cExp
 - ISUPS, 1311
- changeBase
 - MFLOAT, 1548
- changeMeasure
 - ROUTINE, 2331
- changeThreshold
 - ROUTINE, 2331
- changeWeightLevel
 - OWP, 1861
 - WP, 2949
- CHAR, 373
 - ?<?, 373
 - ?<=?, 373
 - ?>?, 373
 - ?>=?, 373
 - ?=?, 373
 - ?~=?, 373
 - alphabetic?, 373
 - alphanumeric?, 373
 - char, 373
 - coerce, 373
 - digit?, 373
 - escape, 373
 - hash, 373
 - hexDigit?, 373
 - index, 373
 - latex, 373
 - lookup, 373
 - lowerCase, 373
 - lowerCase?, 373
 - max, 373
 - min, 373
 - ord, 373
 - quote, 373
 - random, 373
 - size, 373
 - space, 373
 - upperCase, 373
 - upperCase?, 373
- char
 - CHAR, 373
- Character, 373
- character?
 - FST, 962
- CharacterClass, 381
- characteristic
 - ALGFF, 28
 - AN, 35
 - ANTISYM, 40
 - BINARY, 292
 - BPADIC, 258
 - BPADICRT, 263
 - CLIF, 402
 - COMPLEX, 419
 - CONTFRAC, 445
 - DECIMAL, 465

- DERHAM, 530
DFLOAT, 587
DIRPROD, 547
DIRRING, 564
DMP, 572
DPMM, 553
DPMO, 557
DSMP, 542
EMR, 702
EQ, 691
EXPEXPAN, 712
EXPR, 724
EXPUPXS, 740
FEXPR, 947
FF, 820
FFCG, 825
FFCGP, 835
FFCGX, 830
FFNB, 860
FFNBP, 872
FFNBX, 866
FFP, 851
FFX, 846
FLOAT, 908
FR, 786
FRAC, 986
FSERIES, 978
GDMP, 1052
GSERIES, 1090
HACKPI, 1978
HDMP, 1180
HDP, 1173
HEXADEC, 1143
IAN, 1276
IFF, 1284
INT, 1362
INTRVL, 1384
IPADIC, 1294
IPF, 1303
ISUPS, 1311
ITAYLOR, 1338
LA, 1520
LAUPOL, 1422
LODO, 1469
LODO1, 1479
LODO2, 1491
LSQM, 1456
MCMPLX, 1543
MFLOAT, 1548
MINT, 1557
MODFIELD, 1638
MODMON, 1632
MODOP, 1647, 1804
MODRING, 1641
MPOLY, 1682
MRING, 1659
MYEXPR, 1688
MYUP, 1695
NSDPS, 1703
NSMP, 1714
NSUP, 1729
OCT, 1765
ODP, 1816
ODPOL, 1852
ODR, 1858
OMLO, 1806
ONECOMP, 1777
ORDCOMP, 1810
ORESUP, 2492
OREUP, 2872
OWP, 1861
PACOFF, 2136
PACRAT, 2146
PADIC, 1881
PADICRAT, 1886
PADICRC, 1891
PF, 2105
PFR, 1914
POLY, 2078
PR, 2092
QUAT, 2167
RADFF, 2194
RADIX, 2206
RECLOS, 2237
RESRING, 2296
ROMAN, 2326
SAE, 2400
SDPOL, 2386
SHDP, 2509
SINT, 2412
SMP, 2423
SMTS, 2441

- SQMATRIX, [2548](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- SYMPOLY, [2654](#)
- TS, [2671](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- WP, [2949](#)
- XDPOLY, [2967](#)
- XPBWPLY, [2987](#)
- XPOLY, [2998](#)
- XPR, [3007](#)
- XPOLY, [3013](#)
- ZMOD, [1368](#)
- characteristicPolynomial
 - ALGFF, [28](#)
 - COMPLEX, [419](#)
 - MCMPLEX, [1543](#)
 - RADFF, [2194](#)
 - SAE, [2400](#)
- characteristicSerie
 - WUTSET, [2959](#)
- characteristicSet
 - WUTSET, [2959](#)
- charClass
 - CCLASS, [382](#)
- chartCoord
 - BLHN, [317](#)
 - BLQT, [319](#)
- charthRoot
 - ALGFF, [28](#)
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - DECIMAL, [465](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FRAC, [986](#)
 - GDMP, [1052](#)
 - GSERIES, [1090](#)
 - HDMP, [1180](#)
 - HEXADEC, [1143](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - ISUPS, [1311](#)
 - LAUPOL, [1422](#)
 - MCMPLEX, [1543](#)
 - MODMON, [1632](#)
 - MODOP, [1647](#), [1804](#)
 - MPOLY, [1682](#)
 - MRING, [1659](#)
 - MYEXPR, [1688](#)
 - MYUP, [1695](#)
 - NSDPS, [1703](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - OCT, [1765](#)
 - ODPOL, [1852](#)
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - PF, [2105](#)
 - POLY, [2078](#)
 - PR, [2092](#)
 - QUAT, [2167](#)
 - RADFF, [2194](#)
 - RADIX, [2206](#)
 - SAE, [2400](#)
 - SDPOL, [2386](#)

- SMP, [2423](#)
- SMTS, [2441](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- SYMPOLY, [2654](#)
- TS, [2671](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- chartV
 - IC, [1191](#)
 - INFCLSPS, [1271](#)
 - INFCLSPT, [1265](#)
- check
 - GOPT, [1104](#)
 - GOPT0, [1110](#)
 - SPACE3, [2731](#)
- checkExtraValues
 - GOPT, [1104](#)
- checkOptions
 - GOPT0, [1110](#)
- child
 - SUBSPACE, [2614](#)
- child?
 - ALIST, [235](#)
 - BBTREE, [253](#)
 - BSTREE, [302](#)
 - BTOURN, [306](#)
 - BTREE, [310](#)
 - DLIST, [460](#)
 - DSTREE, [535](#)
 - ILIST, [1231](#)
 - LIST, [1504](#)
 - NSDPS, [1703](#)
 - PENDTREE, [1945](#)
 - SPLTREE, [2518](#)
 - STREAM, [2582](#)
 - TREE, [2741](#)
- children
 - ALIST, [235](#)
 - BBTREE, [253](#)
 - BSTREE, [302](#)
 - BTOURN, [306](#)
 - BTREE, [310](#)
 - DLIST, [460](#)
 - DSTREE, [535](#)
 - ILIST, [1231](#)
 - LIST, [1504](#)
 - NSDPS, [1703](#)
 - PENDTREE, [1945](#)
 - SPLTREE, [2518](#)
 - STREAM, [2582](#)
 - SUBSPACE, [2614](#)
 - TREE, [2741](#)
- Ci
 - EXPR, [724](#)
- clearTheFTable
 - INTFTBL, [1371](#)
- clearTheIFTable
 - ODEIFTBL, [1768](#)
- clearTheSymbolTable
 - SYMS, [2696](#)
- CLIF, [401](#)
 - , [402](#)
 - ?**, [402](#)
 - ?*, [402](#)
 - ?+, [402](#)
 - ?-, [402](#)
 - ?/?, [402](#)
 - ?=, [402](#)
 - ?^?, [402](#)
 - ?~=, [402](#)
 - 0, [402](#)
 - 1, [402](#)
 - characteristic, [402](#)
 - coefficient, [402](#)
 - coerce, [402](#)
 - dimension, [402](#)
 - e, [402](#)
 - hash, [402](#)
 - latex, [402](#)
 - monomial, [402](#)
 - one?, [402](#)

- recip, [402](#)
 - sample, [402](#)
 - subtractIfCan, [402](#)
 - zero?, [402](#)
- CliffordAlgebra, [401](#)
- clip
 - DROPT, [608](#)
- clipSurface
 - VIEW3D, [2710](#)
- cLog
 - ISUPS, [1311](#)
- close
 - COMPPROP, [2624](#)
 - VIEW2d, [2770](#)
 - VIEW3D, [2710](#)
- closeComponent
 - SUBSPACE, [2614](#)
- closed?
 - COMPPROP, [2624](#)
 - TUBE, [2749](#)
- closedCurve
 - SPACE3, [2731](#)
- closedCurve?
 - SPACE3, [2731](#)
- code
 - FC, [931](#)
- coef
 - LPOLY, [1447](#)
 - XDPOLY, [2967](#)
 - XPBWPOLYL, [2987](#)
 - XPOLY, [2998](#)
 - XPR, [3007](#)
 - XPOLY, [3013](#)
- coefficient
 - ANTISYM, [40](#)
 - CLIF, [402](#)
 - DERHAM, [530](#)
 - DIV, [575](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPUPXS, [740](#)
 - FAGROUP, [1004](#)
 - FAMONOID, [1007](#)
 - FM1, [1017](#)
 - GDMP, [1052](#)
 - GSERIES, [1090](#)
 - HDMP, [1180](#)
 - IFAMON, [1287](#)
 - ISUPS, [1311](#)
 - LAUPOL, [1422](#)
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - LPOLY, [1447](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MRING, [1659](#)
 - MYUP, [1695](#)
 - NSDPS, [1703](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - OMLO, [1806](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
 - POLY, [2078](#)
 - PR, [2092](#)
 - SD, [2572](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - SYMPOLY, [2654](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UP, [2828](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UPXSING, [2852](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
 - XDPOLY, [2967](#)
 - XPBWPOLYL, [2987](#)
 - XPR, [3007](#)
- coefficients
 - DMP, [572](#)

- DSMP, [542](#)
- FM1, [1017](#)
- GDMP, [1052](#)
- HDMP, [1180](#)
- ITAYLOR, [1338](#)
- LODO, [1469](#)
- LODO1, [1479](#)
- LODO2, [1491](#)
- LPOLY, [1447](#)
- MODMON, [1632](#)
- MPOLY, [1682](#)
- MRING, [1659](#)
- MYUP, [1695](#)
- NSMP, [1714](#)
- NSUP, [1729](#)
- ODPOL, [1852](#)
- ORESUP, [2492](#)
- OREUP, [2872](#)
- POLY, [2078](#)
- PR, [2092](#)
- SDPOL, [2386](#)
- SMP, [2423](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUTS, [2497](#)
- SYMPOLY, [2654](#)
- UFPS, [2789](#)
- UP, [2828](#)
- UPXSSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- XDPOLY, [2967](#)
- XPBWPOLYL, [2987](#)
- XPR, [3007](#)
- coefOfFirstNonZeroTerm
 - NSDPS, [1703](#)
- coerce
 - ACPLOT, [1992](#)
 - AFFPLPS, [7](#)
 - AFFSP, [9](#)
 - ALGFF, [28](#)
 - ALGSC, [15](#)
 - ALIST, [235](#)
 - AN, [35](#)
 - ANON, [38](#)
 - ANTISYM, [40](#)
 - ANY, [50](#)
 - ARRAY1, [1774](#)
 - ARRAY2, [2763](#)
 - ASP1, [71](#)
 - ASP10, [75](#)
 - ASP12, [80](#)
 - ASP19, [85](#)
 - ASP20, [93](#), [98](#)
 - ASP27, [102](#)
 - ASP28, [108](#)
 - ASP29, [114](#)
 - ASP30, [118](#)
 - ASP31, [122](#)
 - ASP33, [127](#)
 - ASP34, [130](#)
 - ASP35, [134](#)
 - ASP4, [140](#)
 - ASP41, [144](#)
 - ASP42, [151](#)
 - ASP49, [158](#)
 - ASP50, [164](#)
 - ASP55, [170](#)
 - ASP6, [176](#)
 - ASP7, [181](#)
 - ASP73, [186](#)
 - ASP74, [191](#)
 - ASP77, [197](#)
 - ASP78, [201](#)
 - ASP8, [206](#)
 - ASP80, [211](#)
 - ASP9, [216](#)
 - ASTACK, [65](#)
 - ATTRBUT, [239](#)
 - AUTOMOR, [245](#)
 - BBTREE, [253](#)
 - BFUNCT, [265](#)
 - BINARY, [292](#)
 - BINFILE, [295](#)
 - BITS, [315](#)
 - BLHN, [317](#)
 - BLQT, [319](#)
 - BOOLEAN, [322](#)
 - BOP, [274](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - BSD, [286](#)

- BSTREE, 302
BTourn, 306
BTREE, 310
CARD, 332
CARTEN, 356
CCLASS, 382
CDFMAT, 426
CDFVEC, 432
CHAR, 373
CLIF, 402
COLOR, 407
COMM, 411
COMPLEX, 419
COMPPROP, 2624
CONTFRAC, 445
D01AJFA, 614
D01AKFA, 617
D01ALFA, 620
D01AMFA, 623
D01APFA, 629
D01AQFA, 633
D01ASFA, 636
D01FCFA, 639
D01GBFA, 643
D01TRNS, 646
D02BBFA, 651
D02BHFA, 654
D02CJFA, 658
D02EJFA, 661
D03EEFA, 665
D03FAFA, 668
D10ANFA, 626
DBASE, 454
DECIMAL, 465
DEQUEUE, 512
DERHAM, 530
DFLOAT, 587
DFMAT, 599
DFVEC, 605
DHMATRIX, 491
DIRPROD, 547
DIRRING, 564
DIV, 575
DLIST, 460
DMP, 572
DPMM, 553
DPMO, 557
DROPT, 608
DSMP, 542
DSTREE, 535
E04DGFA, 747
E04FDFA, 750
E04GCFA, 754
E04JAFa, 758
E04MBFA, 762
E04NAFA, 765
E04UCFA, 769
EAB, 743
EMR, 702
EQ, 691
EQTBL, 699
EXIT, 707
EXPEXPAN, 712
EXPR, 724
EXPUPXS, 740
FAGROUP, 1004
FAMONOID, 1007
FARRAY, 886
FC, 931
FCOMP, 975
FDIV, 813
FEXPR, 947
FF, 820
FFCG, 825
FFCGP, 835
FFCGX, 830
FFNB, 860
FFNBP, 872
FFNBX, 866
FFP, 851
FFX, 846
FGROUP, 1010
FILE, 802
FLOAT, 908
FM, 1013
FM1, 1017
FMONOID, 1021
FNAME, 810
FNLA, 1027
FORMULA, 2346
FORTRAN, 956
FPARFRAC, 1040

- FR, [786](#)
- FRAC, [986](#)
- FRIDEAL, [995](#)
- FRMOD, [1000](#)
- FSERIES, [978](#)
- FST, [962](#)
- FT, [971](#)
- FTEM, [967](#)
- FUNCTION, [1045](#)
- GCNAALG, [1065](#)
- GDMP, [1052](#)
- GMODPOL, [1059](#)
- GOPT, [1104](#)
- GOPT0, [1110](#)
- GPOLSET, [1074](#)
- GRIMAGE, [1094](#)
- GSERIES, [1090](#)
- GSTBL, [1079](#)
- GTSET, [1083](#)
- HACKPI, [1978](#)
- HASHTBL, [1120](#)
- HDMP, [1180](#)
- HDP, [1173](#)
- HEAP, [1134](#)
- HELLFDIV, [1183](#)
- HEXADEC, [1143](#)
- HTMLFORM, [1152](#)
- IAN, [1276](#)
- IARRAY1, [1244](#)
- IARRAY2, [1256](#)
- IBITS, [1199](#)
- IC, [1191](#)
- ICARD, [1193](#)
- IDEAL, [2081](#)
- IDPAG, [1202](#)
- IDPAM, [1206](#)
- IDPO, [1209](#)
- IDPOAM, [1212](#)
- IDPOAMS, [1215](#)
- IFAMON, [1287](#)
- IFARRAY, [1222](#)
- IFF, [1284](#)
- IIARRAY2, [1290](#)
- ILIST, [1231](#)
- IMATRIX, [1239](#)
- INDE, [1218](#)
- INFCLSPS, [1271](#)
- INFCLSPT, [1265](#)
- INFORM, [1343](#)
- INT, [1362](#)
- INTABL, [1336](#)
- INTRVL, [1384](#)
- IPADIC, [1294](#)
- IPF, [1303](#)
- IR, [1375](#)
- ISTRING, [1249](#)
- ISUPS, [1311](#)
- ITAYLOR, [1338](#)
- ITUPLE, [1262](#)
- IVECTOR, [1260](#)
- JORDAN, [223](#)
- KAFILE, [1414](#)
- KERNEL, [1404](#)
- LA, [1520](#)
- LAUPOL, [1422](#)
- LEXP, [1435](#)
- LIB, [1429](#)
- LIE, [229](#)
- LIST, [1504](#)
- LMDICT, [1515](#)
- LMOPS, [1509](#)
- LO, [1523](#)
- LODO, [1469](#)
- LODO1, [1479](#)
- LODO2, [1491](#)
- LPOLY, [1447](#)
- LSQM, [1456](#)
- LWORD, [1532](#)
- M3D, [2703](#)
- MAGMA, [1565](#)
- MATRIX, [1623](#)
- MCMPLEX, [1543](#)
- MFLOAT, [1548](#)
- MINT, [1557](#)
- MKCHSET, [1570](#)
- MMLFORM, [1603](#)
- MODFIELD, [1638](#)
- MODMON, [1632](#)
- MODMONOM, [1644](#)
- MODOP, [1647](#), [1804](#)
- MODRING, [1641](#)
- MOEBIUS, [1654](#)

- MPOLY, 1682
MRING, 1659
MSET, 1670
MYEXPR, 1688
MYUP, 1695
NIPROB, 1746
NNI, 1739
NONE, 1737
NOTTING, 1744
NSDPS, 1703
NSMP, 1714
NSUP, 1729
OCT, 1765
ODEPROB, 1750
ODP, 1816
ODPOL, 1852
ODR, 1858
ODVAR, 1855
OFMONOID, 1829
OMENC, 1789
OMERR, 1792
OMERRK, 1794
OMLO, 1806
ONECOMP, 1777
OPTPROB, 1753
ORDCOMP, 1810
ORESUP, 2492
OREUP, 2872
OSI, 1864
OUTFORM, 1868
OVAR, 1836
OWP, 1861
PACOFF, 2136
PACRAT, 2146
PADIC, 1881
PADICRAT, 1886
PADICRC, 1891
PALETTE, 1896
PATLRES, 1937
PATRES, 1940
PATTERN, 1928
PBWLB, 2054
PDEPROB, 1756
PENDTREE, 1945
PERM, 1949
PERMGRP, 1960
PF, 2105
PFR, 1914
PI, 2100
PLACES, 2019
PLACESPS, 2021
PLOT, 2029
PLOT3D, 2042
POINT, 2060
POLY, 2078
PR, 2092
PRIMARR, 2109
PRODUCT, 2113
PROJPL, 2117
PROJPLPS, 2119
PROJSP, 2122
PRITION, 1923
QALGSET, 2157
QEQUAT, 2169
QFORM, 2154
QUAT, 2167
QUEUE, 2184
RADFF, 2194
RADIX, 2206
RECLOS, 2237
REF, 2249
REGSET, 2286
RESRING, 2296
RESULT, 2301
RGCHAIN, 2255
RMATRIX, 2246
ROIRC, 2310
ROMAN, 2326
ROUTINE, 2331
RULE, 2305
RULECOLD, 2340
RULESET, 2342
SAE, 2400
SAOS, 2418
SD, 2572
SDPOL, 2386
SDVAR, 2389
SEG, 2359
SEGBIND, 2364
SET, 2372
SETMN, 2377
SEX, 2392

- SEXOF, [2394](#)
- SFORT, [2405](#)
- SHDP, [2509](#)
- SINT, [2412](#)
- SMP, [2423](#)
- SMTS, [2441](#)
- SPACE3, [2731](#)
- SPLNODE, [2512](#)
- SPLTREE, [2518](#)
- SQMATRIX, [2548](#)
- SREGSET, [2535](#)
- STACK, [2563](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- SUBSPACE, [2614](#)
- SUCH, [2627](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- SWITCH, [2629](#)
- SYMBOL, [2640](#)
- SYMPOLY, [2654](#)
- SYMS, [2696](#)
- SYMTAB, [2648](#)
- TABLE, [2664](#)
- TABLEAU, [2666](#)
- TEX, [2677](#)
- TEXTFILE, [2692](#)
- TREE, [2741](#)
- TS, [2671](#)
- TUPLE, [2752](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UNISEG, [2896](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- VARIABLE, [2936](#)
- VECTOR, [2942](#)
- VIEW2d, [2770](#)
- VIEW3D, [2710](#)
- VOID, [2945](#)
- WP, [2949](#)
- WUTSET, [2959](#)
- XDPOLY, [2967](#)
- XPBWPLYL, [2987](#)
- XPOLY, [2998](#)
- XPR, [3007](#)
- XPOLY, [3013](#)
- ZMOD, [1368](#)
- coerceImages
 - PERM, [1949](#)
- coerceL
 - HTMLFORM, [1152](#)
 - MMLFORM, [1603](#)
- coerceListOfPairs
 - PERM, [1949](#)
- coercePreimagesImages
 - PERM, [1949](#)
- coerceS
 - HTMLFORM, [1152](#)
 - MMLFORM, [1603](#)
- coHeight
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- collect
 - DIV, [575](#)
 - GPOLSET, [1074](#)
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- collectQuasiMonic
 - GTSET, [1083](#)

- REGSET, [2286](#)
- RGCHAIN, [2255](#)
- SREGSET, [2535](#)
- WUTSET, [2959](#)
- collectUnder
 - GPOLSET, [1074](#)
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- collectUpper
 - GPOLSET, [1074](#)
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- COLOR, [407](#)
 - ?*?, [407](#)
 - ?+?, [407](#)
 - ?=?, [407](#)
 - ?~=?, [407](#)
 - blue, [407](#)
 - coerce, [407](#)
 - color, [407](#)
 - green, [407](#)
 - hash, [407](#)
 - hue, [407](#)
 - latex, [407](#)
 - numberOfHues, [407](#)
 - red, [407](#)
 - yellow, [407](#)
- Color, [407](#)
- color
 - COLOR, [407](#)
- colorDef
 - VIEW3D, [2710](#)
- colorFunction
 - DROPT, [608](#)
- column
 - ARRAY2, [2763](#)
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IARRAY2, [1256](#)
 - IIARRAY2, [1290](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - RMATRIX, [2246](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- columnSpace
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- COMM, [411](#)
 - ?=?, [411](#)
 - ?~=?, [411](#)
 - coerce, [411](#)
 - hash, [411](#)
 - latex, [411](#)
 - mkcomm, [411](#)
- commaSeparate
 - OUTFORM, [1868](#)
- comment
 - FC, [931](#)
- common
 - FC, [931](#)
- commutative?
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- commutativeEquality
 - LMOPS, [1509](#)
- Commutator, [411](#)
- commutator
 - ALGSC, [15](#)
 - AUTOMOR, [245](#)
 - EQ, [691](#)
 - EXPR, [724](#)
 - FGROUP, [1010](#)
 - FNLA, [1027](#)
 - FRIDEAL, [995](#)
 - GCNAALG, [1065](#)

- JORDAN, 223
- LEXP, 1435
- LIE, 229
- LSQM, 1456
- MOEBIUS, 1654
- MYEXPR, 1688
- NOTTING, 1744
- PERM, 1949
- PRODUCT, 2113
- compactFraction
 - PFR, 1914
- comparison
 - BOP, 274
- compile
 - INFORM, 1343
- complement
 - CCLASS, 382
 - SET, 2372
- complementaryBasis
 - ALGFF, 28
 - RADFF, 2194
- complete
 - BPADIC, 258
 - CONTRAC, 445
 - EXPUPXS, 740
 - GSERIES, 1090
 - IPADIC, 1294
 - ISUPS, 1311
 - NSDPS, 1703
 - PADIC, 1881
 - SMTS, 2441
 - STREAM, 2582
 - SULS, 2457
 - SUPXS, 2487
 - SUTS, 2497
 - TS, 2671
 - UFPS, 2789
 - ULS, 2796
 - ULSCONS, 2804
 - UPXS, 2834
 - UPXSCONS, 2842
 - UTS, 2877
 - UTSZ, 2887
- COMPLEX, 418
 - , 419
 - ?<?, 419
 - ?<=?, 419
 - ?>?, 419
 - ?>=?, 419
 - ?**?, 419
 - ?*?, 419
 - ?+?, 419
 - ?-?, 419
 - ?., 419
 - ?/? , 419
 - ?=?, 419
 - ?^?, 419
 - ?~=?, 419
 - ?quo?, 419
 - ?rem?, 419
 - 0, 419
 - 1, 419
 - abs, 419
 - acos, 419
 - acosh, 419
 - acot, 419
 - acoth, 419
 - acsc, 419
 - acsch, 419
 - argument, 419
 - asec, 419
 - asech, 419
 - asin, 419
 - asinh, 419
 - associates?, 419
 - atan, 419
 - atanh, 419
 - basis, 419
 - characteristic, 419
 - characteristicPolynomial, 419
 - charthRoot, 419
 - coerce, 419
 - complex, 419
 - conditionP, 419
 - conjugate, 419
 - convert, 419
 - coordinates, 419
 - cos, 419
 - cosh, 419
 - cot, 419
 - coth, 419
 - createPrimitiveElement, 419

- csc, [419](#)
- csch, [419](#)
- D, [419](#)
- definingPolynomial, [419](#)
- derivationCoordinates, [419](#)
- differentiate, [419](#)
- discreteLog, [419](#)
- discriminant, [419](#)
- divide, [419](#)
- euclideanSize, [419](#)
- eval, [419](#)
- exp, [419](#)
- expressIdealMember, [419](#)
- exquo, [419](#)
- extendedEuclidean, [419](#)
- factor, [419](#)
- factorPolynomial, [419](#)
- factorsOfCyclicGroupSize, [419](#)
- factorSquareFreePolynomial, [419](#)
- gcd, [419](#)
- gcdPolynomial, [419](#)
- generator, [419](#)
- hash, [419](#)
- imag, [419](#)
- imaginary, [419](#)
- index, [419](#)
- init, [419](#)
- inv, [419](#)
- latex, [419](#)
- lcm, [419](#)
- lift, [419](#)
- log, [419](#)
- lookup, [419](#)
- map, [419](#)
- max, [419](#)
- min, [419](#)
- minimalPolynomial, [419](#)
- multiEuclidean, [419](#)
- nextItem, [419](#)
- norm, [419](#)
- nthRoot, [419](#)
- OMwrite, [419](#)
- one?, [419](#)
- order, [419](#)
- patternMatch, [419](#)
- pi, [419](#)
- polarCoordinates, [419](#)
- prime?, [419](#)
- primeFrobenius, [419](#)
- primitive?, [419](#)
- primitiveElement, [419](#)
- principalIdeal, [419](#)
- random, [419](#)
- rank, [419](#)
- rational, [419](#)
- rational?, [419](#)
- rationalIfCan, [419](#)
- real, [419](#)
- recip, [419](#)
- reduce, [419](#)
- reducedSystem, [419](#)
- regularRepresentation, [419](#)
- representationType, [419](#)
- represents, [419](#)
- retract, [419](#)
- retractIfCan, [419](#)
- sample, [419](#)
- sec, [419](#)
- sech, [419](#)
- sin, [419](#)
- sinh, [419](#)
- size, [419](#)
- sizeLess?, [419](#)
- solveLinearPolynomialEquation, [419](#)
- sqrt, [419](#)
- squareFree, [419](#)
- squareFreePart, [419](#)
- squareFreePolynomial, [419](#)
- subtractIfCan, [419](#)
- tableForDiscreteLogarithm, [419](#)
- tan, [419](#)
- tanh, [419](#)
- trace, [419](#)
- traceMatrix, [419](#)
- unit?, [419](#)
- unitCanonical, [419](#)
- unitNormal, [419](#)
- zero?, [419](#)
- Complex, [418](#)
- complex
 - COMPLEX, [419](#)
 - MCMPLEX, [1543](#)

- complex?
 - FST, [962](#)
- ComplexDoubleFloatMatrix, [426](#)
- ComplexDoubleFloatVector, [432](#)
- component
 - GRIMAGE, [1094](#)
- components
 - SPACE3, [2731](#)
- composite
 - MODMON, [1632](#)
 - MYUP, [1695](#)
 - NSUP, [1729](#)
 - SPACE3, [2731](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - UP, [2828](#)
- composites
 - SPACE3, [2731](#)
- COMPPROP, [2624](#)
 - ?=?, [2624](#)
 - ?~=?, [2624](#)
 - close, [2624](#)
 - closed?, [2624](#)
 - coerce, [2624](#)
 - copy, [2624](#)
 - hash, [2624](#)
 - latex, [2624](#)
 - new, [2624](#)
 - solid, [2624](#)
 - solid?, [2624](#)
- computePowers
 - MODMON, [1632](#)
- concat
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - BITS, [315](#)
 - CDFVEC, [432](#)
 - DFVEC, [605](#)
 - DIV, [575](#)
 - DLIST, [460](#)
 - FARRAY, [886](#)
 - IARRAY1, [1244](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - ILIST, [1231](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - LIST, [1504](#)
 - NSDPS, [1703](#)
 - POINT, [2060](#)
 - PRIMARR, [2109](#)
 - ROUTINE, [2331](#)
 - STREAM, [2582](#)
 - STRING, [2607](#)
 - U16VEC, [2924](#)
 - U32VEC, [2931](#)
 - U8VEC, [2917](#)
 - VECTOR, [2942](#)
- cond
 - FC, [931](#)
- condition
 - SPLNODE, [2512](#)
- conditionP
 - ALGFF, [28](#)
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - DECIMAL, [465](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPEXPAN, [712](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FRAC, [986](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - HEXADEC, [1143](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - MCMPLEX, [1543](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)

- ODPOL, 1852
- PACOFF, 2136
- PADICRAT, 1886
- PADICRC, 1891
- PF, 2105
- POLY, 2078
- RADFF, 2194
- RADIX, 2206
- SAE, 2400
- SDPOL, 2386
- SMP, 2423
- SULS, 2457
- SUP, 2467
- SUPEXPR, 2481
- ULS, 2796
- ULSCONS, 2804
- UP, 2828
- conditions
 - SPLTREE, 2518
- conditionsForIdempotents
 - ALGSC, 15
 - GCNAALG, 1065
 - JORDAN, 223
 - LIE, 229
 - LSQM, 1456
- conjug
 - MODOP, 1647, 1804
- conjugate
 - AFFPLPS, 7
 - AFFSP, 9
 - AUTOMOR, 245
 - COMPLEX, 419
 - EQ, 691
 - EXPR, 724
 - FGROUP, 1010
 - FRIDEAL, 995
 - LEXP, 1435
 - MCMPLEX, 1543
 - MOEBIUS, 1654
 - MYEXPR, 1688
 - NOTTING, 1744
 - OCT, 1765
 - PACOFF, 2136
 - PACRAT, 2146
 - PERM, 1949
 - PRODUCT, 2113
 - PROJPL, 2117
 - PROJPLPS, 2119
 - PROJSP, 2122
 - PRITION, 1923
 - QUAT, 2167
- connect
 - VIEW2d, 2770
- cons
 - LIST, 1504
 - STREAM, 2582
- constant
 - XDPOLY, 2967
 - XPBWPLYL, 2987
 - XPOLY, 2998
 - XPR, 3007
 - XPOLY, 3013
- constant?
 - PATTERN, 1928
 - XDPOLY, 2967
 - XPBWPLYL, 2987
 - XPOLY, 2998
 - XPR, 3007
 - XPOLY, 3013
- construct
 - ALIST, 235
 - ARRAY1, 1774
 - BITS, 315
 - CCLASS, 382
 - CDFVEC, 432
 - DFVEC, 605
 - DLIST, 460
 - EQTBL, 699
 - FARRAY, 886
 - FPARFRAC, 1040
 - FT, 971
 - GPOLSET, 1074
 - GSTBL, 1079
 - GTSET, 1083
 - HASHTBL, 1120
 - IARRAY1, 1244
 - IBITS, 1199
 - IFARRAY, 1222
 - ILIST, 1231
 - INTABL, 1336
 - ISTRING, 1249
 - ITUPLE, 1262

- IVECTOR, [1260](#)
- KAFILE, [1414](#)
- LIB, [1429](#)
- LIST, [1504](#)
- LMDICT, [1515](#)
- LPOLY, [1447](#)
- M3D, [2703](#)
- MODMONOM, [1644](#)
- MSET, [1670](#)
- NSDPS, [1703](#)
- PATRES, [1940](#)
- POINT, [2060](#)
- PRIMARR, [2109](#)
- REGSET, [2286](#)
- RESULT, [2301](#)
- RGCHAIN, [2255](#)
- ROUTINE, [2331](#)
- SET, [2372](#)
- SPLNODE, [2512](#)
- SPLTREE, [2518](#)
- SREGSET, [2535](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- SUCH, [2627](#)
- TABLE, [2664](#)
- U16VEC, [2924](#)
- U32VEC, [2931](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- WUTSET, [2959](#)
- contains?
 - INTRVL, [1384](#)
- content
 - DSMP, [542](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
 - POLY, [2078](#)
 - PR, [2092](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - SYMPOLY, [2654](#)
 - UP, [2828](#)
 - UPXSING, [2852](#)
- CONTFRAC, [445](#)
 - ?, [445](#)
 - ?**?, [445](#)
 - ?*?, [445](#)
 - ?+?, [445](#)
 - ?-?, [445](#)
 - ?/?, [445](#)
 - ?=?, [445](#)
 - ?^?, [445](#)
 - ?~=?, [445](#)
 - ?quo?, [445](#)
 - ?rem?, [445](#)
 - 0, [445](#)
 - 1, [445](#)
 - approximants, [445](#)
 - associates?, [445](#)
 - characteristic, [445](#)
 - coerce, [445](#)
 - complete, [445](#)
 - continuedFraction, [445](#)
 - convergents, [445](#)
 - denominators, [445](#)
 - divide, [445](#)
 - euclideanSize, [445](#)
 - expressIdealMember, [445](#)
 - exquo, [445](#)
 - extend, [445](#)
 - extendedEuclidean, [445](#)
 - factor, [445](#)
 - gcd, [445](#)
 - gcdPolynomial, [445](#)
 - hash, [445](#)
 - inv, [445](#)
 - latex, [445](#)

- lcm, [445](#)
- multiEuclidean, [445](#)
- numerators, [445](#)
- one?, [445](#)
- partialDenominators, [445](#)
- partialNumerators, [445](#)
- partialQuotients, [445](#)
- prime?, [445](#)
- principalIdeal, [445](#)
- recip, [445](#)
- reducedContinuedFraction, [445](#)
- reducedForm, [445](#)
- sample, [445](#)
- sizeLess?, [445](#)
- squareFree, [445](#)
- squareFreePart, [445](#)
- subtractIfCan, [445](#)
- unit?, [445](#)
- unitCanonical, [445](#)
- unitNormal, [445](#)
- wholePart, [445](#)
- zero?, [445](#)
- continue
 - FC, [931](#)
- ContinuedFraction, [445](#)
- continuedFraction
 - BPADICRT, [263](#)
 - CONTFRAC, [445](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
- contract
 - CARTEN, [356](#)
- controlPanel
 - VIEW2d, [2770](#)
 - VIEW3D, [2710](#)
- convergents
 - CONTFRAC, [445](#)
- convert
 - ALGFF, [28](#)
 - ALGSC, [15](#)
 - ALIST, [235](#)
 - AN, [35](#)
 - ARRAY1, [1774](#)
 - BINARY, [292](#)
 - BITS, [315](#)
 - BOOLEAN, [322](#)
 - BPADICRT, [263](#)
 - BSD, [286](#)
 - CCLASS, [382](#)
 - CDFVEC, [432](#)
 - COMPLEX, [419](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - DFVEC, [605](#)
 - DLIST, [460](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EQTBL, [699](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - FARRAY, [886](#)
 - FLOAT, [908](#)
 - FORMULA, [2346](#)
 - FPARFRAC, [1040](#)
 - FR, [786](#)
 - FRAC, [986](#)
 - GCNAALG, [1065](#)
 - GPOLSET, [1074](#)
 - GSTBL, [1079](#)
 - GTSET, [1083](#)
 - HACKPI, [1978](#)
 - HASHTBL, [1120](#)
 - HDMP, [1180](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - IARRAY1, [1244](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - ILIST, [1231](#)
 - INFORM, [1343](#)
 - INT, [1362](#)
 - INTABL, [1336](#)
 - IPF, [1303](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - JORDAN, [223](#)
 - KAFILE, [1414](#)
 - KERNEL, [1404](#)
 - LAUPOL, [1422](#)
 - LIB, [1429](#)
 - LIE, [229](#)
 - LIST, [1504](#)

- LMDICT, [1515](#)
- LSQM, [1456](#)
- MATRIX, [1623](#)
- MCMPLEX, [1543](#)
- MFLOAT, [1548](#)
- MINT, [1557](#)
- MODMON, [1632](#)
- MPOLY, [1682](#)
- MSET, [1670](#)
- MYEXPR, [1688](#)
- MYUP, [1695](#)
- NSDPS, [1703](#)
- NSUP, [1729](#)
- OCT, [1765](#)
- OVAR, [1836](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PATTERN, [1928](#)
- PF, [2105](#)
- POINT, [2060](#)
- POLY, [2078](#)
- PRIMARR, [2109](#)
- PRITION, [1923](#)
- QUAT, [2167](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- REGSET, [2286](#)
- RESULT, [2301](#)
- RGCHAIN, [2255](#)
- RMATRIX, [2246](#)
- ROMAN, [2326](#)
- ROUTINE, [2331](#)
- SAE, [2400](#)
- SAOS, [2418](#)
- SDPOL, [2386](#)
- SEG, [2359](#)
- SET, [2372](#)
- SEX, [2392](#)
- SEXOF, [2394](#)
- SINT, [2412](#)
- SMP, [2423](#)
- SQMATRIX, [2548](#)
- SREGSET, [2535](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SYMBOL, [2640](#)
- TABLE, [2664](#)
- TEX, [2677](#)
- U16VEC, [2924](#)
- U32VEC, [2931](#)
- U8VEC, [2917](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UNISEG, [2896](#)
- UP, [2828](#)
- VECTOR, [2942](#)
- WUTSET, [2959](#)
- ZMOD, [1368](#)
- convertIfCan
 - BSD, [286](#)
- coord
 - DROPT, [608](#)
- coordinate
 - PARPCURV, [1899](#)
 - PARSCURV, [1901](#)
 - PARSURF, [1904](#)
- coordinates
 - ALGFF, [28](#)
 - ALGSC, [15](#)
 - COMPLEX, [419](#)
 - DROPT, [608](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - GCNAALG, [1065](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
 - MCMPLEX, [1543](#)

- PF, [2105](#)
- RADFF, [2194](#)
- SAE, [2400](#)
- copy
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - ARRAY2, [2763](#)
 - ASTACK, [65](#)
 - BBTREE, [253](#)
 - BITS, [315](#)
 - BOP, [274](#)
 - BSTREE, [302](#)
 - BTOURN, [306](#)
 - BTREE, [310](#)
 - CCLASS, [382](#)
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)
 - COMPPROP, [2624](#)
 - DEQUEUE, [512](#)
 - DFMAT, [599](#)
 - DFVEC, [605](#)
 - DHMATRIX, [491](#)
 - DIRPROD, [547](#)
 - DLIST, [460](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DSTREE, [535](#)
 - EQTBL, [699](#)
 - FARRAY, [886](#)
 - GPOLSET, [1074](#)
 - GSTBL, [1079](#)
 - GTSET, [1083](#)
 - HASHTBL, [1120](#)
 - HDP, [1173](#)
 - HEAP, [1134](#)
 - IARRAY1, [1244](#)
 - IARRAY2, [1256](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - IIARRAY2, [1290](#)
 - ILIST, [1231](#)
 - IMATRIX, [1239](#)
 - INT, [1362](#)
 - INTABL, [1336](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - KAFILE, [1414](#)
 - LIB, [1429](#)
 - LIST, [1504](#)
 - LMDICT, [1515](#)
 - LSQM, [1456](#)
 - M3D, [2703](#)
 - MATRIX, [1623](#)
 - MINT, [1557](#)
 - MSET, [1670](#)
 - NSDPS, [1703](#)
 - ODP, [1816](#)
 - PATTERN, [1928](#)
 - PENDTREE, [1945](#)
 - POINT, [2060](#)
 - PRIMARR, [2109](#)
 - QUEUE, [2184](#)
 - REGSET, [2286](#)
 - RESULT, [2301](#)
 - RGCHAIN, [2255](#)
 - RMATRIX, [2246](#)
 - ROMAN, [2326](#)
 - ROUTINE, [2331](#)
 - SET, [2372](#)
 - SHDP, [2509](#)
 - SINT, [2412](#)
 - SPACE3, [2731](#)
 - SPLNODE, [2512](#)
 - SPLTREE, [2518](#)
 - SQMATRIX, [2548](#)
 - SREGSET, [2535](#)
 - STACK, [2563](#)
 - STBL, [2450](#)
 - STREAM, [2582](#)
 - STRING, [2607](#)
 - STRTBL, [2610](#)
 - TABLE, [2664](#)
 - TREE, [2741](#)
 - U16MAT, [2906](#)
 - U16VEC, [2924](#)
 - U32MAT, [2911](#)
 - U32VEC, [2931](#)
 - U8MAT, [2902](#)
 - U8VEC, [2917](#)
 - VECTOR, [2942](#)
 - WUTSET, [2959](#)
- copyBSD

- BSD, [286](#)
- copyDrift
 - SD, [2572](#)
- copyIto
 - BSD, [286](#)
- copyQuadVar
 - SD, [2572](#)
- cos
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FCOMP, [975](#)
 - FEXPR, [947](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- cosh
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- coshIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- cosIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- cot
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- coth
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)

- EFUPXS, [681](#)
- EXPR, [724](#)
- EXPUPXS, [740](#)
- FLOAT, [908](#)
- GSERIES, [1090](#)
- INTRVL, [1384](#)
- MCMPLEX, [1543](#)
- SMTS, [2441](#)
- SULS, [2457](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- TS, [2671](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- cothIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- cotIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- count
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - ARRAY2, [2763](#)
 - ASTACK, [65](#)
 - BBTREE, [253](#)
 - BITS, [315](#)
 - BSTREE, [302](#)
 - BTOURN, [306](#)
 - BTREE, [310](#)
 - CCLASS, [382](#)
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)
 - DEQUEUE, [512](#)
 - DFMAT, [599](#)
 - DFVEC, [605](#)
 - DHMATRIX, [491](#)
 - DIRPROD, [547](#)
 - DLIST, [460](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DSTREE, [535](#)
 - EQTBL, [699](#)
 - FARRAY, [886](#)
 - GPOLSET, [1074](#)
 - GSTBL, [1079](#)
 - GTSET, [1083](#)
 - HASHTBL, [1120](#)
 - HDP, [1173](#)
 - HEAP, [1134](#)
 - IARRAY1, [1244](#)
 - IARRAY2, [1256](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - IIARRAY2, [1290](#)
 - ILIST, [1231](#)
 - IMATRIX, [1239](#)
 - INTABL, [1336](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - KAFILE, [1414](#)
 - LIB, [1429](#)
 - LIST, [1504](#)
 - LMDICT, [1515](#)
 - LSQM, [1456](#)
 - M3D, [2703](#)
 - MATRIX, [1623](#)
 - MSET, [1670](#)
 - NSDPS, [1703](#)
 - ODP, [1816](#)
 - PENDTREE, [1945](#)
 - POINT, [2060](#)
 - PRIMARR, [2109](#)
 - QUEUE, [2184](#)
 - REGSET, [2286](#)
 - RESULT, [2301](#)
 - RGCHAIN, [2255](#)
 - RMATRIX, [2246](#)
 - ROUTINE, [2331](#)
 - SET, [2372](#)
 - SHDP, [2509](#)
 - SPLTREE, [2518](#)
 - SQMATRIX, [2548](#)
 - SREGSET, [2535](#)
 - STACK, [2563](#)
 - STBL, [2450](#)

- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- TREE, [2741](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- WUTSET, [2959](#)
- countable?
 - CARD, [332](#)
- cPower
 - ISUPS, [1311](#)
- cRationalPower
 - ISUPS, [1311](#)
- create
 - IC, [1191](#)
 - INFCLSPS, [1271](#)
 - INFCLSPT, [1265](#)
 - PLACES, [2019](#)
 - PLACESPS, [2021](#)
 - SAOS, [2418](#)
- create3Space
 - SPACE3, [2731](#)
- createHN
 - BLHN, [317](#)
 - BLQT, [319](#)
- createNormalElement
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - PF, [2105](#)
- createPrimitiveElement
 - ALGFF, [28](#)
- COMPLEX, [419](#)
- FF, [820](#)
- FFCG, [825](#)
- FFCGP, [835](#)
- FFCGX, [830](#)
- FFNB, [860](#)
- FFNBP, [872](#)
- FFNBX, [866](#)
- FFP, [851](#)
- FFX, [846](#)
- IFF, [1284](#)
- IPF, [1303](#)
- MCMLPX, [1543](#)
- PACOFF, [2136](#)
- PF, [2105](#)
- RADFF, [2194](#)
- SAE, [2400](#)
- cross
 - CDFVEC, [432](#)
 - DFVEC, [605](#)
 - IVECTOR, [1260](#)
 - POINT, [2060](#)
 - VECTOR, [2942](#)
- csc
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMLPX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)

- UTSZ, [2887](#)
- csch
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- cschIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- cscIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- cSec
 - ISUPS, [1311](#)
- cSech
 - ISUPS, [1311](#)
- cSin
 - ISUPS, [1311](#)
- cSinh
 - ISUPS, [1311](#)
- csubst
 - SMTS, [2441](#)
- cTan
 - ISUPS, [1311](#)
- cTanh
 - ISUPS, [1311](#)
- currentSubProgram
 - SYMS, [2696](#)
- curve
 - PARPCURV, [1899](#)
 - PARSCURV, [1901](#)
 - SPACE3, [2731](#)
- curve?
 - SPACE3, [2731](#)
- curveColor
 - DROPT, [608](#)
- curveV
 - IC, [1191](#)
 - INFCLSPS, [1271](#)
 - INFCLSPT, [1265](#)
- cycle
 - PERM, [1949](#)
- cycleEntry
 - ALIST, [235](#)
 - DLIST, [460](#)
 - ILIST, [1231](#)
 - LIST, [1504](#)
 - NSDPS, [1703](#)
 - STREAM, [2582](#)
- cycleLength
 - ALIST, [235](#)
 - DLIST, [460](#)
 - ILIST, [1231](#)
 - LIST, [1504](#)
 - NSDPS, [1703](#)
 - STREAM, [2582](#)
- cyclePartition
 - PERM, [1949](#)
- cycleRagits
 - RADIX, [2206](#)
- cycles
 - PERM, [1949](#)
- cycleTail
 - ALIST, [235](#)
 - DLIST, [460](#)
 - ILIST, [1231](#)
 - LIST, [1504](#)
 - NSDPS, [1703](#)
 - STREAM, [2582](#)
- cyclic?
 - ALIST, [235](#)
 - BBTREE, [253](#)
 - BSTREE, [302](#)

- BTOURN, 306
- BTREE, 310
- DLIST, 460
- DSTREE, 535
- ILIST, 1231
- LIST, 1504
- NSDPS, 1703
- PENDTREE, 1945
- SPLTREE, 2518
- STREAM, 2582
- TREE, 2741
- cyclicCopy
 - TREE, 2741
- cyclicEntries
 - TREE, 2741
- cyclicEqual?
 - TREE, 2741
- cyclicParents
 - TREE, 2741
- D
 - ALGFF, 28
 - AN, 35
 - BINARY, 292
 - BPADICRT, 263
 - COMPLEX, 419
 - DECIMAL, 465
 - DFLOAT, 587
 - DIRPROD, 547
 - DMP, 572
 - DPMM, 553
 - DPMO, 557
 - DSMP, 542
 - EQ, 691
 - EXPEXPAN, 712
 - EXPR, 724
 - EXPUPXS, 740
 - FEXPR, 947
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866
 - FFP, 851
 - FFX, 846
 - FLOAT, 908
 - FPARFRAC, 1040
 - FR, 786
 - FRAC, 986
 - GDMP, 1052
 - GSERIES, 1090
 - HDMP, 1180
 - HDP, 1173
 - HEXADEC, 1143
 - IAN, 1276
 - IFF, 1284
 - INT, 1362
 - IPF, 1303
 - ISUPS, 1311
 - LAUPOL, 1422
 - LODO, 1469
 - LODO1, 1479
 - LODO2, 1491
 - LSQM, 1456
 - MCMPLX, 1543
 - MINT, 1557
 - MODMON, 1632
 - MPOLY, 1682
 - MYEXPR, 1688
 - MYUP, 1695
 - NSDPS, 1703
 - NSMP, 1714
 - NSUP, 1729
 - ODP, 1816
 - ODPOL, 1852
 - ODR, 1858
 - OMLO, 1806
 - PACOFF, 2136
 - PADICRAT, 1886
 - PADICRC, 1891
 - PF, 2105
 - POLY, 2078
 - QUAT, 2167
 - RADFF, 2194
 - RADIX, 2206
 - ROMAN, 2326
 - SAE, 2400
 - SDPOL, 2386
 - SHDP, 2509
 - SINT, 2412

- SMP, [2423](#)
- SMTS, [2441](#)
- SQMATRIX, [2548](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXP, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- TS, [2671](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- d
 - BSD, [286](#)
 - D01AJFA, [614](#)
 - ?=?, [614](#)
 - ?~=?, [614](#)
 - coerce, [614](#)
 - hash, [614](#)
 - latex, [614](#)
 - measure, [614](#)
 - numericalIntegration, [614](#)
 - d01ajfAnnaType, [614](#)
 - D01AKFA, [617](#)
 - ?=?, [617](#)
 - ?~=?, [617](#)
 - coerce, [617](#)
 - hash, [617](#)
 - latex, [617](#)
 - measure, [617](#)
 - numericalIntegration, [617](#)
 - d01akfAnnaType, [617](#)
 - D01ALFA, [620](#)
 - ?=?, [620](#)
 - ?~=?, [620](#)
 - coerce, [620](#)
 - hash, [620](#)
 - latex, [620](#)
 - measure, [620](#)
 - numericalIntegration, [620](#)
 - d01alfAnnaType, [620](#)
 - D01AMFA, [623](#)
 - ?=?, [623](#)
 - ?~=?, [623](#)
 - coerce, [623](#)
 - hash, [623](#)
 - latex, [623](#)
 - measure, [623](#)
 - numericalIntegration, [623](#)
 - d01amfAnnaType, [623](#)
 - D01ANFA, [626](#)
 - d01anfAnnaType, [626](#)
 - D01APFA, [629](#)
 - ?=?, [629](#), [633](#)
 - ?~=?, [629](#), [633](#)
 - coerce, [629](#)
 - hash, [629](#), [633](#)
 - latex, [629](#), [633](#)
 - measure, [629](#), [633](#)
 - numericalIntegration, [629](#), [633](#)
 - d01apfAnnaType, [629](#)
 - D01AQFA, [633](#)
 - coerce, [633](#)
 - d01aqfAnnaType, [633](#)
 - D01ASFA, [636](#)
 - ?=?, [636](#)
 - ?~=?, [636](#)
 - coerce, [636](#)
 - hash, [636](#)
 - latex, [636](#)
 - measure, [636](#)
 - numericalIntegration, [636](#)
 - d01asfAnnaType, [636](#)
 - D01FCFA, [639](#)
 - ?=?, [639](#)
 - ?~=?, [639](#)
 - coerce, [639](#)
 - hash, [639](#)
 - latex, [639](#)
 - measure, [639](#)
 - numericalIntegration, [639](#)
 - d01fcfAnnaType, [639](#)
 - D01GBFA, [643](#)
 - ?=?, [643](#)
 - ?~=?, [643](#)
 - coerce, [643](#)
 - hash, [643](#)

- latex, [643](#)
 - measure, [643](#)
 - numericalIntegration, [643](#)
- d01gbfAnnaType, [643](#)
- d01TransformFunctionType, [646](#)
- D01TRNS, [646](#)
 - ?=?, [646](#)
 - ?~=?, [646](#)
 - coerce, [646](#)
 - hash, [646](#)
 - latex, [646](#)
 - measure, [646](#)
 - numericalIntegration, [646](#)
- D02BBFA, [651](#)
 - ?=?, [651](#)
 - ?~=?, [651](#)
 - coerce, [651](#)
 - hash, [651](#)
 - latex, [651](#)
 - measure, [651](#)
 - ODESolve, [651](#)
- d02bbfAnnaType, [651](#)
- D02BHFA, [654](#)
 - ?=?, [654](#)
 - ?~=?, [654](#)
 - coerce, [654](#)
 - hash, [654](#)
 - latex, [654](#)
 - measure, [654](#)
 - ODESolve, [654](#)
- d02bhfAnnaType, [654](#)
- D02CJFA, [658](#)
 - ?=?, [658](#)
 - ?~=?, [658](#)
 - coerce, [658](#)
 - hash, [658](#)
 - latex, [658](#)
 - measure, [658](#)
 - ODESolve, [658](#)
- d02cjfAnnaType, [658](#)
- D02EJFA, [661](#)
 - ?=?, [661](#)
 - ?~=?, [661](#)
 - coerce, [661](#)
 - hash, [661](#)
 - latex, [661](#)
 - measure, [661](#)
 - ODESolve, [661](#)
- d02ejfAnnaType, [661](#)
- D03EEFA, [665](#)
 - ?=?, [665](#)
 - ?~=?, [665](#)
 - coerce, [665](#)
 - hash, [665](#)
 - latex, [665](#)
 - measure, [665](#)
 - PDESolve, [665](#)
- d03eefAnnaType, [665](#)
- D03FAFA, [668](#)
 - ?=?, [668](#)
 - ?~=?, [668](#)
 - coerce, [668](#)
 - hash, [668](#)
 - latex, [668](#)
 - measure, [668](#)
 - PDESolve, [668](#)
- d03fafAnnaType, [668](#)
- D10ANFA
 - ?=?, [626](#)
 - ?~=?, [626](#)
 - coerce, [626](#)
 - hash, [626](#)
 - latex, [626](#)
 - measure, [626](#)
 - numericalIntegration, [626](#)
- dark
 - PALETTE, [1896](#)
- Database, [454](#)
- DataList, [459](#)
- datalist
 - DLIST, [460](#)
- DBASE, [454](#)
 - ?+?, [454](#)
 - ?-?, [454](#)
 - ?., [454](#)
 - ?=?, [454](#)
 - ?~=?, [454](#)
 - coerce, [454](#)
 - display, [454](#)
 - fullDisplay, [454](#)
 - hash, [454](#)
 - latex, [454](#)

- debug
 - GOPT, [1104](#)
 - GOPT0, [1110](#)
 - PLOT, [2029](#)
- debug3D
 - PLOT3D, [2042](#)
- dec
 - INT, [1362](#)
 - MINT, [1557](#)
 - ROMAN, [2326](#)
 - SINT, [2412](#)
- DECIMAL, [465](#)
- ?, [465](#)
- ?<?, [465](#)
- ?<=?, [465](#)
- ?>?, [465](#)
- ?>=?, [465](#)
- ?**?, [465](#)
- ?*?, [465](#)
- ?+?, [465](#)
- ?-?, [465](#)
- ?., [465](#)
- ?/?, [465](#)
- ?=?, [465](#)
- ?^?, [465](#)
- ?~=?, [465](#)
- ?quo?, [465](#)
- ?rem?, [465](#)
- 0, [465](#)
- 1, [465](#)
- abs, [465](#)
- associates?, [465](#)
- ceiling, [465](#)
- characteristic, [465](#)
- charthRoot, [465](#)
- coerce, [465](#)
- conditionP, [465](#)
- convert, [465](#)
- D, [465](#)
- decimal, [465](#)
- denom, [465](#)
- denominator, [465](#)
- differentiate, [465](#)
- divide, [465](#)
- euclideanSize, [465](#)
- eval, [465](#)
- expressIdealMember, [465](#)
- exquo, [465](#)
- extendedEuclidean, [465](#)
- factor, [465](#)
- factorPolynomial, [465](#)
- factorSquareFreePolynomial, [465](#)
- floor, [465](#)
- fractionPart, [465](#)
- gcd, [465](#)
- gcdPolynomial, [465](#)
- hash, [465](#)
- init, [465](#)
- inv, [465](#)
- latex, [465](#)
- lcm, [465](#)
- map, [465](#)
- max, [465](#)
- min, [465](#)
- multiEuclidean, [465](#)
- negative?, [465](#)
- nextItem, [465](#)
- numer, [465](#)
- numerator, [465](#)
- one?, [465](#)
- patternMatch, [465](#)
- positive?, [465](#)
- prime?, [465](#)
- principalIdeal, [465](#)
- random, [465](#)
- recip, [465](#)
- reducedSystem, [465](#)
- retract, [465](#)
- retractIfCan, [465](#)
- sample, [465](#)
- sign, [465](#)
- sizeLess?, [465](#)
- solveLinearPolynomialEquation, [465](#)
- squareFree, [465](#)
- squareFreePart, [465](#)
- squareFreePolynomial, [465](#)
- subtractIfCan, [465](#)
- unit?, [465](#)
- unitCanonical, [465](#)
- unitNormal, [465](#)
- wholePart, [465](#)
- zero?, [465](#)

- decimal
 - DECIMAL, [465](#)
- DecimalExpansion, [465](#)
- declare
 - INFORM, [1343](#)
- decompose
 - FDIV, [813](#)
 - HELLFDIV, [1183](#)
- decrease
 - ATTRBUT, [239](#)
- decreasePrecision
 - DFLOAT, [587](#)
 - FLOAT, [908](#)
 - MFLOAT, [1548](#)
- deepCopy
 - SUBSPACE, [2614](#)
- deepestInitial
 - NSMP, [1714](#)
- deepestTail
 - NSMP, [1714](#)
- deepExpand
 - FNLA, [1027](#)
- defineProperty
 - SUBSPACE, [2614](#)
- definingEquations
 - QALGSET, [2157](#)
- definingField
 - AFFPLPS, [7](#)
 - AFFSP, [9](#)
 - PROJPL, [2117](#)
 - PROJPLPS, [2119](#)
 - PROJSP, [2122](#)
- definingInequation
 - QALGSET, [2157](#)
- definingPolynomial
 - ALGFF, [28](#)
 - AN, [35](#)
 - COMPLEX, [419](#)
 - EXPR, [724](#)
 - FEXPR, [947](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - GDMP, [1052](#)
 - GSERIES, [1090](#)
 - GTSET, [1083](#)
 - HDMP, [1180](#)
 - IC, [1191](#)
 - IFF, [1284](#)
 - INFCLSPS, [1271](#)
 - INFCLSPT, [1265](#)
 - IPF, [1303](#)
 - ISUPS, [1311](#)
 - LAUPOL, [1422](#)
 - LODO, [1469](#)
- degree
 - AFFPLPS, [7](#)
 - AFFSP, [9](#)
 - ANTISYM, [40](#)
 - CARTEN, [356](#)
 - DERHAM, [530](#)
 - DIV, [575](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EAB, [743](#)
 - EXPUPXS, [740](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - GDMP, [1052](#)
 - GSERIES, [1090](#)
 - GTSET, [1083](#)
 - HDMP, [1180](#)
 - IC, [1191](#)
 - IFF, [1284](#)
 - INFCLSPS, [1271](#)
 - INFCLSPT, [1265](#)
 - IPF, [1303](#)
 - ISUPS, [1311](#)
 - LAUPOL, [1422](#)
 - LODO, [1469](#)

- LODO1, [1479](#)
- LODO2, [1491](#)
- LPOLY, [1447](#)
- MODMON, [1632](#)
- MPOLY, [1682](#)
- MYUP, [1695](#)
- NSDPS, [1703](#)
- NSMP, [1714](#)
- NSUP, [1729](#)
- ODPOL, [1852](#)
- OMLO, [1806](#)
- ORESUP, [2492](#)
- OREUP, [2872](#)
- PACOFF, [2136](#)
- PACRAT, [2146](#)
- PERM, [1949](#)
- PERMGRP, [1960](#)
- PF, [2105](#)
- PLACES, [2019](#)
- PLACESPS, [2021](#)
- POLY, [2078](#)
- PR, [2092](#)
- PROJPL, [2117](#)
- PROJPLPS, [2119](#)
- PROJSP, [2122](#)
- REGSET, [2286](#)
- RGCHAIN, [2255](#)
- SDPOL, [2386](#)
- SMP, [2423](#)
- SMTS, [2441](#)
- SREGSET, [2535](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- SYMPOLY, [2654](#)
- TS, [2671](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- WUTSET, [2959](#)
- XDPOLY, [2967](#)
- XPBWPOLYL, [2987](#)
- XPOLY, [2998](#)
- XRPOLY, [3013](#)
- delay
 - NSDPS, [1703](#)
 - STREAM, [2582](#)
- delete
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - BITS, [315](#)
 - CDFVEC, [432](#)
 - DFVEC, [605](#)
 - DLIST, [460](#)
 - FARRAY, [886](#)
 - IARRAY1, [1244](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - ILIST, [1231](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - LIST, [1504](#)
 - NSDPS, [1703](#)
 - POINT, [2060](#)
 - PRIMARR, [2109](#)
 - STREAM, [2582](#)
 - STRING, [2607](#)
 - U16VEC, [2924](#)
 - U32VEC, [2931](#)
 - U8VEC, [2917](#)
 - VECTOR, [2942](#)
- delta
 - SETMN, [2377](#)
- DenavitHartenbergMatrix, [491](#)
- denom
 - AN, [35](#)
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - DECIMAL, [465](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - FRAC, [986](#)
 - FRIDEAL, [995](#)
 - HEXADEC, [1143](#)

- IAN, [1276](#)
- LA, [1520](#)
- LO, [1523](#)
- MYEXPR, [1688](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- RADIX, [2206](#)
- SULS, [2457](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- denominator
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - DECIMAL, [465](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - FRAC, [986](#)
 - HEXADEC, [1143](#)
 - MYEXPR, [1688](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - RADIX, [2206](#)
 - SULS, [2457](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
- denominators
 - CONTFRAC, [445](#)
- depth
 - ASTACK, [65](#)
 - DEQUEUE, [512](#)
 - PATTERN, [1928](#)
 - STACK, [2563](#)
- DEQUEUE, [512](#)
- ?=?, [512](#)
- ?~=?, [512](#)
- #?, [512](#)
- any?, [512](#)
- back, [512](#)
- bag, [512](#)
- bottom, [512](#)
- coerce, [512](#)
- copy, [512](#)
- count, [512](#)
- depth, [512](#)
- dequeue, [512](#)
- empty, [512](#)
- empty?, [512](#)
- enqueue, [512](#)
- eq?, [512](#)
- eval, [512](#)
- every?, [512](#)
- extract, [512](#)
- extractBottom, [512](#)
- extractTop, [512](#)
- front, [512](#)
- hash, [512](#)
- height, [512](#)
- insert, [512](#)
- insertBottom, [512](#)
- insertTop, [512](#)
- inspect, [512](#)
- latex, [512](#)
- length, [512](#)
- less?, [512](#)
- map, [512](#)
- member?, [512](#)
- members, [512](#)
- more?, [512](#)
- parts, [512](#)
- pop, [512](#)
- push, [512](#)
- reverse, [512](#)
- rotate, [512](#)
- sample, [512](#)
- size?, [512](#)
- top, [512](#)
- Dequeue, [512](#)
- dequeue
 - DEQUEUE, [512](#)
- deref
 - REF, [2249](#)
- DERHAM, [530](#)
- , [530](#)
- ?**, [530](#)
- ?*?, [530](#)
- ?+?, [530](#)
- ?-?, [530](#)
- ?=?, [530](#)
- ?^?, [530](#)
- ?~=?, [530](#)
- 0, [530](#)
- 1, [530](#)

- characteristic, [530](#)
- coefficient, [530](#)
- coerce, [530](#)
- degree, [530](#)
- exteriorDifferential, [530](#)
- generator, [530](#)
- hash, [530](#)
- homogeneous?, [530](#)
- latex, [530](#)
- leadingBasisTerm, [530](#)
- leadingCoefficient, [530](#)
- map, [530](#)
- one?, [530](#)
- recip, [530](#)
- reductum, [530](#)
- retract, [530](#)
- retractable?, [530](#)
- retractIfCan, [530](#)
- sample, [530](#)
- subtractIfCan, [530](#)
- totalDifferential, [530](#)
- zero?, [530](#)
- DeRhamComplex, [530](#)
- derivationCoordinates
 - ALGFF, [28](#)
 - COMPLEX, [419](#)
 - MCMPLEX, [1543](#)
 - RADFF, [2194](#)
 - SAE, [2400](#)
- DesingTree, [534](#)
- destruct
 - INFORM, [1343](#)
 - PATRES, [1940](#)
 - SEX, [2392](#)
 - SEXOF, [2394](#)
- determinant
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- DFLOAT, [586](#)
- ?, [587](#)
- ?<?, [587](#)
- ?<=?, [587](#)
- ?>?, [587](#)
- ?>=?, [587](#)
- ?**?, [587](#)
- ?*?, [587](#)
- ?+?, [587](#)
- ?-?, [587](#)
- ?/?, [587](#)
- ?=?, [587](#)
- ?^?, [587](#)
- ?~=?, [587](#)
- ?quo?, [587](#)
- ?rem?, [587](#)
- 0, [587](#)
- 1, [587](#)
- abs, [587](#)
- acos, [587](#)
- acosh, [587](#)
- acot, [587](#)
- acoth, [587](#)
- acsc, [587](#)
- acsch, [587](#)
- airyAi, [587](#)
- airyBi, [587](#)
- asec, [587](#)
- asech, [587](#)
- asin, [587](#)
- asinh, [587](#)
- associates?, [587](#)
- atan, [587](#)
- atanh, [587](#)
- base, [587](#)
- besselI, [587](#)
- besselJ, [587](#)
- besselK, [587](#)
- besselY, [587](#)
- Beta, [587](#)
- bits, [587](#)
- ceiling, [587](#)
- characteristic, [587](#)
- coerce, [587](#)
- convert, [587](#)
- cos, [587](#)

- cosh, 587
- cot, 587
- coth, 587
- csc, 587
- csch, 587
- D, 587
- decreasePrecision, 587
- differentiate, 587
- digamma, 587
- digits, 587
- divide, 587
- doubleFloatFormat, 587
- euclideanSize, 587
- exp, 587
- exp1, 587
- exponent, 587
- expressIdealMember, 587
- exquo, 587
- extendedEuclidean, 587
- factor, 587
- float, 587
- floor, 587
- fractionPart, 587
- Gamma, 587
- gcd, 587
- gcdPolynomial, 587
- hash, 587
- increasePrecision, 587
- integerDecode, 587
- inv, 587
- latex, 587
- lcm, 587
- log, 587
- log10, 587
- log2, 587
- machineFraction, 587
- mantissa, 587
- max, 587
- min, 587
- multiEuclidean, 587
- negative?, 587
- norm, 587
- nthRoot, 587
- OMwrite, 587
- one?, 587
- order, 587
- patternMatch, 587
- pi, 587
- polygamma, 587
- positive?, 587
- precision, 587
- prime?, 587
- principalIdeal, 587
- rationalApproximation, 587
- recip, 587
- retract, 587
- retractIfCan, 587
- round, 587
- sample, 587
- sec, 587
- sech, 587
- sign, 587
- sin, 587
- sinh, 587
- sizeLess?, 587
- sqrt, 587
- squareFree, 587
- squareFreePart, 587
- subtractIfCan, 587
- tan, 587
- tanh, 587
- truncate, 587
- unit?, 587
- unitCanonical, 587
- unitNormal, 587
- wholePart, 587
- zero?, 587
- DFMAT, 599
- ?, 599
- ?**?, 599
- ?*?, 599
- ?+?, 599
- ?-?, 599
- ?/?, 599
- ?=?, 599
- ?~=?, 599
- #?, 599
- antisymmetric?, 599
- any?, 599
- coerce, 599
- column, 599
- columnSpace, 599

- copy, 599
- count, 599
- determinant, 599
- diagonal?, 599
- diagonalMatrix, 599
- elt, 599
- empty, 599
- empty?, 599
- eq?, 599
- eval, 599
- every?, 599
- exquo, 599
- fill, 599
- hash, 599
- horizConcat, 599
- inverse, 599
- latex, 599
- less?, 599
- listOfLists, 599
- map, 599
- matrix, 599
- maxColIndex, 599
- maxRowIndex, 599
- member?, 599
- members, 599
- minColIndex, 599
- minordet, 599
- minRowIndex, 599
- more?, 599
- ncols, 599
- new, 599
- nrows, 599
- nullity, 599
- nullSpace, 599
- parts, 599
- pfaffian, 599
- qelt, 599
- qnew, 599
- qsetelt, 599
- rank, 599
- row, 599
- rowEchelon, 599
- sample, 599
- scalarMatrix, 599
- setColumn, 599
- setelt, 599
- setRow, 599
- setsubMatrix, 599
- size?, 599
- square?, 599
- squareTop, 599
- subMatrix, 599
- swapColumns, 599
- swapRows, 599
- symmetric?, 599
- transpose, 599
- vertConcat, 599
- zero, 599
- DFVEC, 605
 - .?, 605
 - ?<?, 605
 - ?<=?, 605
 - ?>?, 605
 - ?>=?, 605
 - ?*?, 605
 - ?+?, 605
 - ?-?, 605
 - ?..?, 605
 - ?=?, 605
 - ?~=?, 605
 - #?, 605
 - any?, 605
 - coerce, 605
 - concat, 605
 - construct, 605
 - convert, 605
 - copy, 605
 - copyInto, 605
 - count, 605
 - cross, 605
 - delete, 605
 - dot, 605
 - elt, 605
 - empty, 605
 - empty?, 605
 - entries, 605
 - entry?, 605
 - eq?, 605
 - eval, 605
 - every?, 605
 - fill, 605
 - find, 605

- first, 605
- hash, 605
- index?, 605
- indices, 605
- insert, 605
- latex, 605
- length, 605
- less?, 605
- magnitude, 605
- map, 605
- max, 605
- maxIndex, 605
- member?, 605
- members, 605
- merge, 605
- min, 605
- minIndex, 605
- more?, 605
- new, 605
- outerProduct, 605
- parts, 605
- position, 605
- qelt, 605
- qnew, 605
- qsetelt, 605
- reduce, 605
- remove, 605
- removeDuplicates, 605
- reverse, 605
- sample, 605
- select, 605
- setelt, 605
- size?, 605
- sort, 605
- sorted?, 605
- swap, 605
- zero, 605
- DHMATRIX, 491
 - ?, 491
 - ***?, 491
 - *?*, 491
 - ?+?, 491
 - ?-?, 491
 - ?/? , 491
 - ?=?, 491
 - ?~=?, 491
 - #?, 491
 - antisymmetric?, 491
 - any?, 491
 - coerce, 491
 - column, 491
 - copy, 491
 - count, 491
 - determinant, 491
 - diagonal?, 491
 - diagonalMatrix, 491
 - elt, 491
 - empty, 491
 - empty?, 491
 - eq?, 491
 - eval, 491
 - every?, 491
 - exquo, 491
 - fill, 491
 - hash, 491
 - horizConcat, 491
 - identity, 491
 - inverse, 491
 - latex, 491
 - less?, 491
 - listOfLists, 491
 - map, 491
 - matrix, 491
 - maxColIndex, 491
 - maxRowIndex, 491
 - member?, 491
 - members, 491
 - minColIndex, 491
 - minordet, 491
 - minRowIndex, 491
 - more?, 491
 - ncols, 491
 - new, 491
 - nrows, 491
 - nullity, 491
 - nullSpace, 491
 - parts, 491
 - qelt, 491
 - qsetelt, 491
 - rank, 491
 - rotatex, 491
 - rotatey, 491

- rotatez, [491](#)
- row, [491](#)
- rowEchelon, [491](#)
- sample, [491](#)
- scalarMatrix, [491](#)
- scale, [491](#)
- setColumn, [491](#)
- setelt, [491](#)
- setRow, [491](#)
- setsubMatrix, [491](#)
- size?, [491](#)
- square?, [491](#)
- squareTop, [491](#)
- subMatrix, [491](#)
- swapColumns, [491](#)
- swapRows, [491](#)
- symmetric?, [491](#)
- translate, [491](#)
- transpose, [491](#)
- vertConcat, [491](#)
- zero, [491](#)
- diagonal
 - LSQM, [1456](#)
 - SQMATRIX, [2548](#)
- diagonal?
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - RMATRIX, [2246](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- diagonalMatrix
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
- U8MAT, [2902](#)
- diagonalProduct
 - LSQM, [1456](#)
 - SQMATRIX, [2548](#)
- diagonals
 - VIEW3D, [2710](#)
- dictionary
 - ALIST, [235](#)
 - CCLASS, [382](#)
 - EQTBL, [699](#)
 - GSTBL, [1079](#)
 - HASHTBL, [1120](#)
 - INTABL, [1336](#)
 - KAFILE, [1414](#)
 - LIB, [1429](#)
 - LMDICT, [1515](#)
 - MSET, [1670](#)
 - RESULT, [2301](#)
 - ROUTINE, [2331](#)
 - SET, [2372](#)
 - STBL, [2450](#)
 - STRTBL, [2610](#)
 - TABLE, [2664](#)
- difference
 - CCLASS, [382](#)
 - MSET, [1670](#)
 - SET, [2372](#)
- DifferentialSparseMultivariatePolynomial, [541](#)
- differentialVariables
 - DSMP, [542](#)
 - ODPOL, [1852](#)
 - SDPOL, [2386](#)
- differentiate
 - ALGFF, [28](#)
 - AN, [35](#)
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - DIRPROD, [547](#)
 - DMP, [572](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DSMP, [542](#)
 - EQ, [691](#)

- EXPEXPAN, 712
- EXPR, 724
- EXPUPXS, 740
- FEXPR, 947
- FF, 820
- FFCG, 825
- FFCGP, 835
- FFCGX, 830
- FFNB, 860
- FFNBP, 872
- FFNBX, 866
- FFP, 851
- FFX, 846
- FLOAT, 908
- FPARFRAC, 1040
- FR, 786
- FRAC, 986
- GDMP, 1052
- GSERIES, 1090
- HDMP, 1180
- HDP, 1173
- HEXADEC, 1143
- IAN, 1276
- IFF, 1284
- INT, 1362
- IPF, 1303
- IR, 1375
- ISUPS, 1311
- LAUPOL, 1422
- LSQM, 1456
- MCMPLEX, 1543
- MINT, 1557
- MODMON, 1632
- MPOLY, 1682
- MYEXPR, 1688
- MYUP, 1695
- NSDPS, 1703
- NSMP, 1714
- NSUP, 1729
- ODP, 1816
- ODPOL, 1852
- ODR, 1858
- ODVAR, 1855
- OMLO, 1806
- OUTFORM, 1868
- PACOFF, 2136
- PADICRAT, 1886
- PADICRC, 1891
- PF, 2105
- POLY, 2078
- QUAT, 2167
- RADFF, 2194
- RADIX, 2206
- ROMAN, 2326
- SAE, 2400
- SDPOL, 2386
- SDVAR, 2389
- SHDP, 2509
- SINT, 2412
- SMP, 2423
- SMTS, 2441
- SQMATRIX, 2548
- SULS, 2457
- SUP, 2467
- SUPEXP, 2481
- SUPXS, 2487
- SUTS, 2497
- TS, 2671
- UFPS, 2789
- ULS, 2796
- ULSCONS, 2804
- UP, 2828
- UPXS, 2834
- UPXSCONS, 2842
- UTS, 2877
- UTSZ, 2887
- digamma
 - DFLOAT, 587
 - EXPR, 724
- digit
 - CCLASS, 382
- digit?
 - CHAR, 373
- digits
 - BPADIC, 258
 - DFLOAT, 587
 - FLOAT, 908
 - IPADIC, 1294
 - MFLOAT, 1548
 - PADIC, 1881
- dilog
 - EXPR, 724

- dim
 - PALETTE, 1896
- dimension
 - CLIF, 402
 - DIRPROD, 547
 - DPMM, 553
 - DPMO, 557
 - EQ, 691
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866
 - FFP, 851
 - FFX, 846
 - FNLA, 1027
 - HDP, 1173
 - IDEAL, 2081
 - IFF, 1284
 - IPF, 1303
 - ODP, 1816
 - PACOFF, 2136
 - PACRAT, 2146
 - PF, 2105
 - POINT, 2060
 - RMATRIX, 2246
 - SHDP, 2509
- dimensions
 - VIEW2d, 2770
 - VIEW3D, 2710
- dimensionsOf
 - FT, 971
- directory
 - FNAME, 810
- DirectProduct, 547
- directProduct
 - DIRPROD, 547
 - DPMM, 553
 - DPMO, 557
 - HDP, 1173
 - ODP, 1816
 - SHDP, 2509
- DirectProductMatrixModule, 552
- DirectProductModule, 557
- directSum
 - LODO, 1469
 - LODO1, 1479
 - LODO2, 1491
- DirichletRing, 563
- DIRPROD, 547
 - , 547
 - ?<?, 547
 - ?<=?, 547
 - ?>?, 547
 - ?>=?, 547
 - ?**?, 547
 - ?*?, 547
 - ?+?, 547
 - ?-?, 547
 - ?.?, 547
 - ?/? , 547
 - ?=?, 547
 - ?^?, 547
 - ?~=?, 547
 - #?, 547
 - 0, 547
 - 1, 547
 - abs, 547
 - any?, 547
 - characteristic, 547
 - coerce, 547
 - copy, 547
 - count, 547
 - D, 547
 - differentiate, 547
 - dimension, 547
 - directProduct, 547
 - dot, 547
 - elt, 547
 - empty, 547
 - empty?, 547
 - entries, 547
 - entry?, 547
 - eq?, 547
 - eval, 547
 - every?, 547
 - fill, 547
 - first, 547
 - hash, 547
 - index, 547

- index?, [547](#)
- indices, [547](#)
- latex, [547](#)
- less?, [547](#)
- lookup, [547](#)
- map, [547](#)
- max, [547](#)
- maxIndex, [547](#)
- member?, [547](#)
- members, [547](#)
- min, [547](#)
- minIndex, [547](#)
- more?, [547](#)
- negative?, [547](#)
- one?, [547](#)
- parts, [547](#)
- positive?, [547](#)
- qelt, [547](#)
- qsetelt, [547](#)
- random, [547](#)
- recip, [547](#)
- reducedSystem, [547](#)
- retract, [547](#)
- retractIfCan, [547](#)
- sample, [547](#)
- setelt, [547](#)
- sign, [547](#)
- size, [547](#)
- size?, [547](#)
- subtractIfCan, [547](#)
- sup, [547](#)
- swap, [547](#)
- unitVector, [547](#)
- zero?, [547](#)
- DIRRING, [563](#)
- ?, [564](#)
- ? =?, [564](#)
- ?**?, [564](#)
- ?*?, [564](#)
- ?+?, [564](#)
- ?-?, [564](#)
- ?., [564](#)
- ?=, [564](#)
- ?^?, [564](#)
- 0, [564](#)
- 1, [564](#)
- additive?, [564](#)
- associates?, [564](#)
- characteristic, [564](#)
- coerce, [564](#)
- exquo, [564](#)
- hash, [564](#)
- latex, [564](#)
- multiplicative?, [564](#)
- one?, [564](#)
- recip, [564](#)
- sample, [564](#)
- subtractIfCan, [564](#)
- unit?, [564](#)
- unitCanonical, [564](#)
- unitNormal, [564](#)
- zero?, [564](#)
- zeta, [564](#)
- discreteLog
 - ALGFF, [28](#)
 - COMPLEX, [419](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - MCMPLEX, [1543](#)
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
 - PF, [2105](#)
 - RADFF, [2194](#)
 - SAE, [2400](#)
- discriminant
 - ALGFF, [28](#)
 - COMPLEX, [419](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - MCMPLEX, [1543](#)
 - MODMON, [1632](#)

- MPOLY, 1682
- MYUP, 1695
- NSMP, 1714
- NSUP, 1729
- ODPOL, 1852
- POLY, 2078
- RADFF, 2194
- SAE, 2400
- SDPOL, 2386
- SMP, 2423
- SUP, 2467
- SUEXPR, 2481
- UP, 2828
- display
 - BOP, 274
 - DBASE, 454
 - FORMULA, 2346
 - HTMLFORM, 1152
 - ICARD, 1193
 - MMLFORM, 1603
 - TEX, 2677
- displayAsGF
 - GOPT0, 1110
- displayKind
 - GOPT, 1104
- distance
 - ALIST, 235
 - BBTREE, 253
 - BSTREE, 302
 - BTOURN, 306
 - BTREE, 310
 - DLIST, 460
 - DSTREE, 535
 - ILIST, 1231
 - LIST, 1504
 - NSDPS, 1703
 - PENDTREE, 1945
 - SPLTREE, 2518
 - STREAM, 2582
 - TREE, 2741
- distinguishedRootsOf
 - PACOFF, 2136
 - PACRAT, 2146
- distribute
 - AN, 35
 - EXPR, 724
 - FEXPR, 947
 - IAN, 1276
 - MYEXPR, 1688
- DistributedMultivariatePolynomial, 571
- DIV, 575
 - .?, 575
 - ?<=?, 575
 - ?*?, 575
 - ?+?, 575
 - ?-?, 575
 - ?=?, 575
 - ?~=?, 575
 - 0, 575
 - coefficient, 575
 - coerce, 575
 - collect, 575
 - concat, 575
 - degree, 575
 - divOfPole, 575
 - divOfZero, 575
 - effective?, 575
 - hash, 575
 - head, 575
 - highCommonTerms, 575
 - incr, 575
 - latex, 575
 - mapCoef, 575
 - mapGen, 575
 - nthCoef, 575
 - nthFactor, 575
 - reductum, 575
 - retract, 575
 - retractIfCan, 575
 - sample, 575
 - size, 575
 - split, 575
 - subtractIfCan, 575
 - supp, 575
 - suppOfPole, 575
 - suppOfZero, 575
 - terms, 575
 - zero?, 575
- divide
 - ALGFF, 28
 - AN, 35
 - BINARY, 292

- BPADIC, [258](#)
- BPADICRT, [263](#)
- COMPLEX, [419](#)
- CONTFRAC, [445](#)
- DECIMAL, [465](#)
- DFLOAT, [587](#)
- EMR, [702](#)
- EXPEXPAN, [712](#)
- EXPR, [724](#)
- EXPUPXS, [740](#)
- FF, [820](#)
- FFCG, [825](#)
- FFCGP, [835](#)
- FFCGX, [830](#)
- FFNB, [860](#)
- FFNBP, [872](#)
- FFNBX, [866](#)
- FFP, [851](#)
- FFX, [846](#)
- FLOAT, [908](#)
- FMONOID, [1021](#)
- FRAC, [986](#)
- GSERIES, [1090](#)
- HACKPI, [1978](#)
- HEXADEC, [1143](#)
- IAN, [1276](#)
- IFF, [1284](#)
- INT, [1362](#)
- IPADIC, [1294](#)
- IPF, [1303](#)
- LAUPOL, [1422](#)
- MCMPLEX, [1543](#)
- MFLOAT, [1548](#)
- MINT, [1557](#)
- MODFIELD, [1638](#)
- MODMON, [1632](#)
- MYEXPR, [1688](#)
- MYUP, [1695](#)
- NNI, [1739](#)
- NSDPS, [1703](#)
- NSUP, [1729](#)
- ODR, [1858](#)
- PACOFF, [2136](#)
- PACRAT, [2146](#)
- PADIC, [1881](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PF, [2105](#)
- PFR, [1914](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- ROMAN, [2326](#)
- SAE, [2400](#)
- SINT, [2412](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXP, [2481](#)
- SUPXS, [2487](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- divideExponents
 - MODMON, [1632](#)
 - MYUP, [1695](#)
 - NSUP, [1729](#)
 - SUP, [2467](#)
 - SUPEXP, [2481](#)
 - UP, [2828](#)
- Divisor, [575](#)
- divisor
 - FDIV, [813](#)
 - HELLFDIV, [1183](#)
- divOfPole
 - DIV, [575](#)
- divOfZero
 - DIV, [575](#)
- DLIST, [459](#)
 - ?<?, [460](#)
 - ?<=?, [460](#)
 - ?>?, [460](#)
 - ?>=?, [460](#)
 - ?.?, [460](#)
 - ?.count, [460](#)
 - ?.first, [460](#)
 - ?.last, [460](#)
 - ?.rest, [460](#)
 - ?.sort, [460](#)
 - ?.unique, [460](#)
 - ?.value, [460](#)

- [=?](#), 460
- [?~=?](#), 460
- [#?](#), 460
- [any?](#), 460
- [child?](#), 460
- [children](#), 460
- [coerce](#), 460
- [concat](#), 460
- [construct](#), 460
- [convert](#), 460
- [copy](#), 460
- [copyInto](#), 460
- [count](#), 460
- [cycleEntry](#), 460
- [cycleLength](#), 460
- [cycleSplit](#), 460
- [cycleTail](#), 460
- [cyclic?](#), 460
- [datalist](#), 460
- [delete](#), 460
- [distance](#), 460
- [elt](#), 460
- [empty](#), 460
- [empty?](#), 460
- [entries](#), 460
- [entry?](#), 460
- [eq?](#), 460
- [eval](#), 460
- [every?](#), 460
- [explicitlyFinite?](#), 460
- [fill](#), 460
- [find](#), 460
- [first](#), 460
- [hash](#), 460
- [index?](#), 460
- [indices](#), 460
- [insert](#), 460
- [last](#), 460
- [latex](#), 460
- [leaf?](#), 460
- [leaves](#), 460
- [less?](#), 460
- [list](#), 460
- [map](#), 460
- [max](#), 460
- [maxIndex](#), 460
- [member?](#), 460
- [members](#), 460
- [merge](#), 460
- [min](#), 460
- [minIndex](#), 460
- [more?](#), 460
- [new](#), 460
- [node?](#), 460
- [nodes](#), 460
- [parts](#), 460
- [position](#), 460
- [possiblyInfinite?](#), 460
- [qelt](#), 460
- [qsetelt](#), 460
- [reduce](#), 460
- [remove](#), 460
- [removeDuplicates](#), 460
- [rest](#), 460
- [reverse](#), 460
- [sample](#), 460
- [second](#), 460
- [select](#), 460
- [setchildren](#), 460
- [setelt](#), 460
- [setfirst](#), 460
- [setlast](#), 460
- [setrest](#), 460
- [setvalue](#), 460
- [size?](#), 460
- [sort](#), 460
- [sorted?](#), 460
- [split](#), 460
- [swap](#), 460
- [tail](#), 460
- [third](#), 460
- [value](#), 460
- [DMP](#), 571
 - [-?](#), 572
 - [?<?](#), 572
 - [?<=?](#), 572
 - [?>?](#), 572
 - [?>=?](#), 572
 - [?**?](#), 572
 - [?*?](#), 572
 - [?+?](#), 572
 - [?-?](#), 572

- [?/?](#), 572
- [?=?](#), 572
- [?^?](#), 572
- [?~=?](#), 572
- [0](#), 572
- [1](#), 572
- [associates?](#), 572
- [binomThmExpt](#), 572
- [characteristic](#), 572
- [charthRoot](#), 572
- [coefficient](#), 572
- [coefficients](#), 572
- [coerce](#), 572
- [conditionP](#), 572
- [convert](#), 572
- [D](#), 572
- [degree](#), 572
- [differentiate](#), 572
- [discriminant](#), 572
- [eval](#), 572
- [exquo](#), 572
- [factor](#), 572
- [factorPolynomial](#), 572
- [factorSquareFreePolynomial](#), 572
- [gcd](#), 572
- [gcdPolynomial](#), 572
- [ground](#), 572
- [ground?](#), 572
- [hash](#), 572
- [isExpt](#), 572
- [isPlus](#), 572
- [isTimes](#), 572
- [latex](#), 572
- [lcm](#), 572
- [leadingCoefficient](#), 572
- [leadingMonomial](#), 572
- [mainVariable](#), 572
- [map](#), 572
- [mapExponents](#), 572
- [max](#), 572
- [min](#), 572
- [minimumDegree](#), 572
- [monicDivide](#), 572
- [monomial](#), 572
- [monomial?](#), 572
- [monomials](#), 572
- [multivariate](#), 572
- [numberOfMonomials](#), 572
- [one?](#), 572
- [patternMatch](#), 572
- [pomopo](#), 572
- [prime?](#), 572
- [primitiveMonomials](#), 572
- [primitivePart](#), 572
- [recip](#), 572
- [reducedSystem](#), 572
- [reductum](#), 572
- [reorder](#), 572
- [resultant](#), 572
- [retract](#), 572
- [retractIfCan](#), 572
- [sample](#), 572
- [solveLinearPolynomialEquation](#), 572
- [squareFree](#), 572
- [squareFreePart](#), 572
- [squareFreePolynomial](#), 572
- [subtractIfCan](#), 572
- [totalDegree](#), 572
- [unit?](#), 572
- [unitCanonical](#), 572
- [unitNormal](#), 572
- [univariate](#), 572
- [variables](#), 572
- [zero?](#), 572
- [dom](#)
 - [ANY](#), 50
- [domainOf](#)
 - [ANY](#), 50
- [dominantTerm](#)
 - [UPXSSING](#), 2852
- [dot](#)
 - [CDFVEC](#), 432
 - [DFVEC](#), 605
 - [DIRPROD](#), 547
 - [DPMM](#), 553
 - [DPMO](#), 557
 - [HDP](#), 1173
 - [IVECTOR](#), 1260
 - [ODP](#), 1816
 - [OUTFORM](#), 1868
 - [POINT](#), 2060
 - [SHDP](#), 2509

- VECTOR, 2942
- double?
 - FST, 962
- doubleComplex?
 - FST, 962
- DoubleFloat, 586
- doubleFloatFormat
 - DFLOAT, 587
- DoubleFloatMatrix, 599
- DoubleFloatVector, 605
- DPMM, 552
 - , 553
 - ?<?, 553
 - ?<=?, 553
 - ?>?, 553
 - ?>=?, 553
 - ?**?, 553
 - ?*?, 553
 - ?-?, 553
 - ?., 553
 - ?/?, 553
 - ?=?, 553
 - ?^?, 553
 - ?~=?, 553
 - #?, 553
 - 0, 553
 - 1, 553
 - abs, 553
 - any?, 553
 - characteristic, 553
 - coerce, 553
 - copy, 553
 - count, 553
 - D, 553
 - differentiate, 553
 - dimension, 553
 - directProduct, 553
 - dot, 553
 - elt, 553
 - empty, 553
 - empty?, 553
 - entries, 553
 - entry?, 553
 - eq?, 553
 - eval, 553
 - every?, 553
 - fill, 553
 - first, 553
 - hash, 553
 - index, 553
 - index?, 553
 - indices, 553
 - latex, 553
 - less?, 553
 - lookup, 553
 - map, 553
 - max, 553
 - maxIndex, 553
 - member?, 553
 - members, 553
 - min, 553
 - minIndex, 553
 - more?, 553
 - negative?, 553
 - one?, 553
 - parts, 553
 - positive?, 553
 - qelt, 553
 - qsetelt, 553
 - random, 553
 - recip, 553
 - reducedSystem, 553
 - retract, 553
 - retractIfCan, 553
 - sample, 553
 - setelt, 553
 - sign, 553
 - size, 553
 - size?, 553
 - subtractIfCan, 553
 - sup, 553
 - swap, 553
 - unitVector, 553
 - zero?, 553
- DPMO, 557
 - , 557
 - ?<?, 557
 - ?<=?, 557
 - ?>?, 557
 - ?>=?, 557
 - ?**?, 557
 - ?*?, 557

- [?+?](#), 557
- [?-?](#), 557
- [?..?](#), 557
- [?/?](#), 557
- [?=?](#), 557
- [?^?](#), 557
- [?~=?](#), 557
- [#?](#), 557
- [0](#), 557
- [1](#), 557
- [abs](#), 557
- [any?](#), 557
- [characteristic](#), 557
- [coerce](#), 557
- [copy](#), 557
- [count](#), 557
- [D](#), 557
- [differentiate](#), 557
- [dimension](#), 557
- [directProduct](#), 557
- [dot](#), 557
- [elt](#), 557
- [empty](#), 557
- [empty?](#), 557
- [entries](#), 557
- [entry?](#), 557
- [eq?](#), 557
- [eval](#), 557
- [every?](#), 557
- [fill](#), 557
- [first](#), 557
- [hash](#), 557
- [index](#), 557
- [index?](#), 557
- [indices](#), 557
- [latex](#), 557
- [less?](#), 557
- [lookup](#), 557
- [map](#), 557
- [max](#), 557
- [maxIndex](#), 557
- [member?](#), 557
- [members](#), 557
- [min](#), 557
- [minIndex](#), 557
- [more?](#), 557
- [negative?](#), 557
- [one?](#), 557
- [parts](#), 557
- [positive?](#), 557
- [qelt](#), 557
- [qsetelt](#), 557
- [random](#), 557
- [recip](#), 557
- [reducedSystem](#), 557
- [retract](#), 557
- [retractIfCan](#), 557
- [sample](#), 557
- [setelt](#), 557
- [sign](#), 557
- [size](#), 557
- [size?](#), 557
- [subtractIfCan](#), 557
- [sup](#), 557
- [swap](#), 557
- [unitVector](#), 557
- [zero?](#), 557
- [DrawOption](#), 608
- [drawStyle](#)
 - [VIEW3D](#), 2710
- [drift](#)
 - [SD](#), 2572
- [DROPT](#), 608
 - [?=?](#), 608
 - [?~=?](#), 608
 - [adaptive](#), 608
 - [clip](#), 608
 - [coerce](#), 608
 - [colorFunction](#), 608
 - [coord](#), 608
 - [coordinates](#), 608
 - [curveColor](#), 608
 - [hash](#), 608
 - [latex](#), 608
 - [option](#), 608
 - [option?](#), 608
 - [pointColor](#), 608
 - [range](#), 608
 - [ranges](#), 608
 - [space](#), 608
 - [style](#), 608
 - [title](#), 608

- toScale, 608
- tubePoints, 608
- tubeRadius, 608
- unit, 608
- var1Steps, 608
- var2Steps, 608
- viewpoint, 608
- DSMP, 541
- ?, 542
- ?<?, 542
- ?<=?, 542
- ?>?, 542
- ?>=?, 542
- ?**?, 542
- ?*?, 542
- ?+?, 542
- ?-?, 542
- ?/?, 542
- ?=?, 542
- ?^?, 542
- ?~=?, 542
- 0, 542
- 1, 542
- associates?, 542
- binomThmExpt, 542
- characteristic, 542
- charthRoot, 542
- coefficient, 542
- coefficients, 542
- coerce, 542
- conditionP, 542
- content, 542
- convert, 542
- D, 542
- degree, 542
- differentialVariables, 542
- differentiate, 542
- discriminant, 542
- eval, 542
- exquo, 542
- factor, 542
- factorPolynomial, 542
- factorSquareFreePolynomial, 542
- gcd, 542
- gcdPolynomial, 542
- ground, 542
- ground?, 542
- hash, 542
- initial, 542
- isExpt, 542
- isobaric?, 542
- isPlus, 542
- isTimes, 542
- latex, 542
- lcm, 542
- leader, 542
- leadingCoefficient, 542
- leadingMonomial, 542
- makeVariable, 542
- map, 542
- mapExponents, 542
- max, 542
- min, 542
- minimumDegree, 542
- monicDivide, 542
- monomial, 542
- monomial?, 542
- monomials, 542
- multivariate, 542
- numberOfMonomials, 542
- one?, 542
- order, 542
- patternMatch, 542
- pomopo, 542
- prime?, 542
- primitiveMonomials, 542
- primitivePart, 542
- recip, 542
- reducedSystem, 542
- reductum, 542
- resultant, 542
- retract, 542
- retractIfCan, 542
- sample, 542
- separant, 542
- solveLinearPolynomialEquation, 542
- squareFree, 542
- squareFreePart, 542
- squareFreePolynomial, 542
- subtractIfCan, 542
- totalDegree, 542
- unit?, 542

- unitCanonical, 542
- unitNormal, 542
- univariate, 542
- variables, 542
- weight, 542
- weights, 542
- zero?, 542
- DSTREE, 534
 - ?.value, 535
 - ?=?, 535
 - ?~=?, 535
 - #?, 535
 - any?, 535
 - child?, 535
 - children, 535
 - coerce, 535
 - copy, 535
 - count, 535
 - cyclic?, 535
 - distance, 535
 - empty, 535
 - empty?, 535
 - encode, 535
 - eq?, 535
 - eval, 535
 - every?, 535
 - fullOut, 535
 - fullOutput, 535
 - hash, 535
 - latex, 535
 - leaf?, 535
 - leaves, 535
 - less?, 535
 - map, 535
 - member?, 535
 - members, 535
 - more?, 535
 - node?, 535
 - nodes, 535
 - parts, 535
 - sample, 535
 - setchildren, 535
 - setelt, 535
 - setvalue, 535
 - size?, 535
 - tree, 535
 - value, 535
- duplicates
 - LMDICT, 1515
 - MSET, 1670
- duplicates?
 - LMDICT, 1515
- e
 - CLIF, 402
- E04DGFA, 747
 - ?=?, 747
 - ?~=?, 747
 - coerce, 747
 - hash, 747
 - latex, 747
 - measure, 747
 - numericalOptimization, 747
- e04dgfAnnaType, 747
- E04FDFA, 750
 - ?=?, 750
 - ?~=?, 750
 - coerce, 750
 - hash, 750
 - latex, 750
 - measure, 750
 - numericalOptimization, 750
- e04fdfAnnaType, 750
- E04GCFA, 754
 - ?=?, 754
 - ?~=?, 754
 - coerce, 754
 - hash, 754
 - latex, 754
 - measure, 754
 - numericalOptimization, 754
- e04gcfAnnaType, 754
- E04JAFA, 758
 - ?=?, 758
 - ?~=?, 758
 - coerce, 758
 - hash, 758
 - latex, 758
 - measure, 758
 - numericalOptimization, 758
- e04jafAnnaType, 758
- E04MBFA, 762

- $?=?$, 762
- $?^?=?$, 762
- coerce, 762
- hash, 762
- latex, 762
- measure, 762
- numericalOptimization, 762
- e04mbfAnnaType, 762
- E04NAFA, 765
 - $?=?$, 765
 - $?^?=?$, 765
 - coerce, 765
 - hash, 765
 - latex, 765
 - measure, 765
 - numericalOptimization, 765
- e04nafAnnaType, 765
- E04UCFA, 769
 - $?=?$, 769
 - $?^?=?$, 769
 - coerce, 769
 - hash, 769
 - latex, 769
 - measure, 769
 - numericalOptimization, 769
- e04ucfAnnaType, 769
- EAB, 743
 - $?<?$, 743
 - $?<=?$, 743
 - $?>?$, 743
 - $?>=?$, 743
 - $?=?$, 743
 - $?^?=?$, 743
 - coerce, 743
 - degree, 743
 - exponents, 743
 - hash, 743
 - latex, 743
 - max, 743
 - min, 743
 - Nul, 743
- effective?
 - DIV, 575
- EFULS, 672
 - $?^{**}?$, 672
 - acos, 672
 - acosh, 672
 - acoshIfCan, 672
 - acosIfCan, 672
 - acot, 672
 - acoth, 672
 - acothIfCan, 672
 - acotIfCan, 672
 - acsc, 672
 - acsch, 672
 - acschIfCan, 672
 - acscIfCan, 672
 - asec, 672
 - asech, 672
 - asechIfCan, 672
 - asecIfCan, 672
 - asin, 672
 - asinh, 672
 - asinhIfCan, 672
 - asinIfCan, 672
 - atan, 672
 - atanh, 672
 - atanhIfCan, 672
 - atanIfCan, 672
 - cos, 672
 - cosh, 672
 - coshIfCan, 672
 - cosIfCan, 672
 - cot, 672
 - coth, 672
 - cothIfCan, 672
 - cotIfCan, 672
 - csc, 672
 - csch, 672
 - cschIfCan, 672
 - cscIfCan, 672
 - exp, 672
 - expIfCan, 672
 - log, 672
 - logIfCan, 672
 - nthRootIfCan, 672
 - sec, 672
 - sech, 672
 - sechIfCan, 672
 - secIfCan, 672
 - sin, 672
 - sinh, 672

- sinhIfCan, [672](#)
- sinIfCan, [672](#)
- tan, [672](#)
- tanh, [672](#)
- tanhIfCan, [672](#)
- tanIfCan, [672](#)
- EFUPXS, [681](#)
- ***?, [681](#)
- acos, [681](#)
- acosh, [681](#)
- acoshIfCan, [681](#)
- acosIfCan, [681](#)
- acot, [681](#)
- acoth, [681](#)
- acothIfCan, [681](#)
- acotIfCan, [681](#)
- acsc, [681](#)
- acsch, [681](#)
- acschIfCan, [681](#)
- acscIfCan, [681](#)
- asec, [681](#)
- asech, [681](#)
- asechIfCan, [681](#)
- asecIfCan, [681](#)
- asin, [681](#)
- asinh, [681](#)
- asinhIfCan, [681](#)
- asinIfCan, [681](#)
- atan, [681](#)
- atanh, [681](#)
- atanhIfCan, [681](#)
- atanIfCan, [681](#)
- cos, [681](#)
- cosh, [681](#)
- coshIfCan, [681](#)
- cosIfCan, [681](#)
- cot, [681](#)
- coth, [681](#)
- cothIfCan, [681](#)
- cotIfCan, [681](#)
- csc, [681](#)
- csch, [681](#)
- cschIfCan, [681](#)
- cscIfCan, [681](#)
- exp, [681](#)
- expIfCan, [681](#)
- log, [681](#)
- logIfCan, [681](#)
- nthRootIfCan, [681](#)
- sec, [681](#)
- sech, [681](#)
- sechIfCan, [681](#)
- secIfCan, [681](#)
- sin, [681](#)
- sinh, [681](#)
- sinhIfCan, [681](#)
- sinIfCan, [681](#)
- tan, [681](#)
- tanh, [681](#)
- tanhIfCan, [681](#)
- tanIfCan, [681](#)
- Ei
 - EXPR, [724](#)
- elem?
 - IR, [1375](#)
- element?
 - IDEAL, [2081](#)
- ElementaryFunctionsUnivariateLaurentSeries, [672](#)
- ElementaryFunctionsUnivariatePuisseuxSeries, [681](#)
- elements
 - SETMN, [2377](#)
- elliptic
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- elt
 - ALGFF, [28](#)
 - ALIST, [235](#)
 - AN, [35](#)
 - ARRAY1, [1774](#)
 - ARRAY2, [2763](#)
 - BITS, [315](#)
 - CARTEN, [356](#)
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)
 - DFMAT, [599](#)
 - DFVEC, [605](#)
 - DHMATRIX, [491](#)
 - DIRPROD, [547](#)
 - DLIST, [460](#)
 - DPMM, [553](#)

- DPMO, 557
- EQTBL, 699
- EXPR, 724
- FARRAY, 886
- FEXPR, 947
- GSTBL, 1079
- HASHTBL, 1120
- HDP, 1173
- IAN, 1276
- IARRAY1, 1244
- IARRAY2, 1256
- IBITS, 1199
- IFARRAY, 1222
- IIARRAY2, 1290
- ILIST, 1231
- IMATRIX, 1239
- INTABL, 1336
- ISTRING, 1249
- IVECTOR, 1260
- KAFILE, 1414
- LIB, 1429
- LIST, 1504
- LSQM, 1456
- M3D, 2703
- MATRIX, 1623
- MODMON, 1632
- MYEXPR, 1688
- MYUP, 1695
- NSDPS, 1703
- NSUP, 1729
- ODP, 1816
- PATTERN, 1928
- POINT, 2060
- PRIMARR, 2109
- RADFF, 2194
- REF, 2249
- RESULT, 2301
- RMATRIX, 2246
- ROUTINE, 2331
- RULE, 2305
- RULESET, 2342
- SHDP, 2509
- SQMATRIX, 2548
- STBL, 2450
- STREAM, 2582
- STRING, 2607
- STRTBL, 2610
- SUP, 2467
- SUPEXP, 2481
- TABLE, 2664
- U16MAT, 2906
- U16VEC, 2924
- U32MAT, 2911
- U32VEC, 2931
- U8MAT, 2902
- U8VEC, 2917
- UP, 2828
- VECTOR, 2942
- empty
 - ALIST, 235
 - ARRAY1, 1774
 - ARRAY2, 2763
 - ASTACK, 65
 - BBTREE, 253
 - BITS, 315
 - BSTREE, 302
 - BTOURN, 306
 - BTREE, 310
 - CCLASS, 382
 - CDFMAT, 426
 - CDFVEC, 432
 - DEQUEUE, 512
 - DFMAT, 599
 - DFVEC, 605
 - DHMATRIX, 491
 - DIRPROD, 547
 - DLIST, 460
 - DPMM, 553
 - DPMO, 557
 - DSTREE, 535
 - EQTBL, 699
 - FARRAY, 886
 - GPOLSET, 1074
 - GSTBL, 1079
 - GTSET, 1083
 - HASHTBL, 1120
 - HDP, 1173
 - HEAP, 1134
 - IARRAY1, 1244
 - IARRAY2, 1256
 - IBITS, 1199
 - IFARRAY, 1222

IIARRAY2, 1290
ILIST, 1231
IMATRIX, 1239
INTABL, 1336
ISTRING, 1249
IVECTOR, 1260
KAFILE, 1414
LIB, 1429
LIST, 1504
LMDICT, 1515
LSQM, 1456
M3D, 2703
MATRIX, 1623
MSET, 1670
NSDPS, 1703
ODP, 1816
OUTFORM, 1868
PENDTREE, 1945
POINT, 2060
PRIMARR, 2109
QALGSET, 2157
QUEUE, 2184
REGSET, 2286
RESULT, 2301
RGCHAIN, 2255
RMATRIX, 2246
ROUTINE, 2331
SET, 2372
SHDP, 2509
SPLNODE, 2512
SPLTREE, 2518
SQMATRIX, 2548
SREGSET, 2535
STACK, 2563
STBL, 2450
STREAM, 2582
STRING, 2607
STRTBL, 2610
SYMS, 2696
SYMTAB, 2648
TABLE, 2664
TREE, 2741
U16MAT, 2906
U16VEC, 2924
U32MAT, 2911
U32VEC, 2931
U8MAT, 2902
U8VEC, 2917
VECTOR, 2942
WUTSET, 2959
empty?
ALIST, 235
ARRAY1, 1774
ARRAY2, 2763
ASTACK, 65
BBTREE, 253
BITS, 315
BSTREE, 302
BTourn, 306
BTREE, 310
CCLASS, 382
CDFMAT, 426
CDFVEC, 432
DEQUEUE, 512
DFMAT, 599
DFVEC, 605
DHMATRIX, 491
DIRPROD, 547
DLIST, 460
DPMM, 553
DPMO, 557
DSTREE, 535
EQTBL, 699
FARRAY, 886
GPOLSET, 1074
GSTBL, 1079
GTSET, 1083
HASHTBL, 1120
HDP, 1173
HEAP, 1134
IARRAY1, 1244
IARRAY2, 1256
IBITS, 1199
IFARRAY, 1222
IIARRAY2, 1290
ILIST, 1231
IMATRIX, 1239
INTABL, 1336
ISTRING, 1249
IVECTOR, 1260
KAFILE, 1414
LIB, 1429

- LIST, [1504](#)
- LMDICT, [1515](#)
- LSQM, [1456](#)
- M3D, [2703](#)
- MATRIX, [1623](#)
- MSET, [1670](#)
- NSDPS, [1703](#)
- ODP, [1816](#)
- PENDTREE, [1945](#)
- POINT, [2060](#)
- PRIMARR, [2109](#)
- QALGSET, [2157](#)
- QUEUE, [2184](#)
- REGSET, [2286](#)
- RESULT, [2301](#)
- RGCHAIN, [2255](#)
- RMATRIX, [2246](#)
- ROUTINE, [2331](#)
- SET, [2372](#)
- SHDP, [2509](#)
- SPLNODE, [2512](#)
- SPLTREE, [2518](#)
- SQMATRIX, [2548](#)
- SREGSET, [2535](#)
- STACK, [2563](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- TREE, [2741](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- WUTSET, [2959](#)
- EMR, [701](#)
- ?, [702](#)
- ?**, [702](#)
- ?*, [702](#)
- ?+?, [702](#)
- ?-?, [702](#)
- ?.?, [702](#)
- ?=?, [702](#)
- ?^?, [702](#)
- ?~=?, [702](#)
- ?quo?, [702](#)
- ?rem?, [702](#)
- 0, [702](#)
- 1, [702](#)
- associates?, [702](#)
- characteristic, [702](#)
- coerce, [702](#)
- divide, [702](#)
- euclideanSize, [702](#)
- expressIdealMember, [702](#)
- exQuo, [702](#)
- exquo, [702](#)
- extendedEuclidean, [702](#)
- gcd, [702](#)
- gcdPolynomial, [702](#)
- hash, [702](#)
- inv, [702](#)
- latex, [702](#)
- lcm, [702](#)
- modulus, [702](#)
- multiEuclidean, [702](#)
- one?, [702](#)
- principalIdeal, [702](#)
- recip, [702](#)
- reduce, [702](#)
- sample, [702](#)
- sizeLess?, [702](#)
- subtractIfCan, [702](#)
- unit?, [702](#)
- unitCanonical, [702](#)
- unitNormal, [702](#)
- zero?, [702](#)
- encode
 - DSTREE, [535](#)
- endOfFile?
 - TEXTFILE, [2692](#)
- endSubProgram
 - SYMS, [2696](#)
- enterPointData
 - SPACE3, [2731](#)
- entries
 - ALIST, [235](#)
 - ARRAY1, [1774](#)

- BITS, [315](#)
- CDFVEC, [432](#)
- DFVEC, [605](#)
- DIRPROD, [547](#)
- DLIST, [460](#)
- DPMM, [553](#)
- DPMO, [557](#)
- EQTBL, [699](#)
- FARRAY, [886](#)
- GSTBL, [1079](#)
- HASHTBL, [1120](#)
- HDP, [1173](#)
- IARRAY1, [1244](#)
- IBITS, [1199](#)
- IFARRAY, [1222](#)
- ILIST, [1231](#)
- INTABL, [1336](#)
- INTFTBL, [1371](#)
- ISTRING, [1249](#)
- IVECTOR, [1260](#)
- KAFILE, [1414](#)
- LIB, [1429](#)
- LIST, [1504](#)
- NSDPS, [1703](#)
- ODP, [1816](#)
- POINT, [2060](#)
- PRIMARR, [2109](#)
- RESULT, [2301](#)
- ROUTINE, [2331](#)
- SHDP, [2509](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- U16VEC, [2924](#)
- U32VEC, [2931](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- entry
 - INTFTBL, [1371](#)
- entry?
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - BITS, [315](#)
 - CDFVEC, [432](#)
 - DFVEC, [605](#)
 - DIRPROD, [547](#)
 - DLIST, [460](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - EQTBL, [699](#)
 - FARRAY, [886](#)
 - GSTBL, [1079](#)
 - HASHTBL, [1120](#)
 - HDP, [1173](#)
 - IARRAY1, [1244](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - ILIST, [1231](#)
 - INTABL, [1336](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - KAFILE, [1414](#)
 - LIB, [1429](#)
 - LIST, [1504](#)
 - NSDPS, [1703](#)
 - ODP, [1816](#)
 - POINT, [2060](#)
 - PRIMARR, [2109](#)
 - RESULT, [2301](#)
 - ROUTINE, [2331](#)
 - SHDP, [2509](#)
 - STBL, [2450](#)
 - STREAM, [2582](#)
 - STRING, [2607](#)
 - STRTBL, [2610](#)
 - TABLE, [2664](#)
 - U16VEC, [2924](#)
 - U32VEC, [2931](#)
 - U8VEC, [2917](#)
 - VECTOR, [2942](#)
- enumerate
 - SETMN, [2377](#)
- epilogue
 - FORMULA, [2346](#)
 - TEX, [2677](#)
- EQ, [691](#)
 - , [691](#)
 - ?**?, [691](#)
 - ?*?, [691](#)
 - ?+?, [691](#)

- ?-?, [691](#)
- ?/?, [691](#)
- ?=?, [691](#)
- ?^?, [691](#)
- ?~=?, [691](#)
- 0, [691](#)
- 1, [691](#)
- characteristic, [691](#)
- coerce, [691](#)
- commutator, [691](#)
- conjugate, [691](#)
- D, [691](#)
- differentiate, [691](#)
- dimension, [691](#)
- equation, [691](#)
- eval, [691](#)
- factorAndSplit, [691](#)
- hash, [691](#)
- inv, [691](#)
- latex, [691](#)
- leftOne, [691](#)
- leftZero, [691](#)
- lhs, [691](#)
- map, [691](#)
- one?, [691](#)
- recip, [691](#)
- rhs, [691](#)
- rightOne, [691](#)
- rightZero, [691](#)
- sample, [691](#)
- subst, [691](#)
- subtractIfCan, [691](#)
- swap, [691](#)
- SWITCH, [2629](#)
- zero?, [691](#)
- eq
 - INFORM, [1343](#)
 - SEX, [2392](#)
 - SEXOF, [2394](#)
- eq?
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - ARRAY2, [2763](#)
 - ASTACK, [65](#)
 - BBTREE, [253](#)
 - BITS, [315](#)
 - BSTREE, [302](#)
 - BTOURN, [306](#)
 - BTREE, [310](#)
 - CCLASS, [382](#)
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)
 - DEQUEUE, [512](#)
 - DFMAT, [599](#)
 - DFVEC, [605](#)
 - DHMATRIX, [491](#)
 - DIRPROD, [547](#)
 - DLIST, [460](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DSTREE, [535](#)
 - EQTBL, [699](#)
 - FARRAY, [886](#)
 - GPOLSET, [1074](#)
 - GSTBL, [1079](#)
 - GTSET, [1083](#)
 - HASHTBL, [1120](#)
 - HDP, [1173](#)
 - HEAP, [1134](#)
 - IARRAY1, [1244](#)
 - IARRAY2, [1256](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - IARRAY2, [1290](#)
 - ILIST, [1231](#)
 - IMATRIX, [1239](#)
 - INTABL, [1336](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - KAFILE, [1414](#)
 - LIB, [1429](#)
 - LIST, [1504](#)
 - LMDICT, [1515](#)
 - LSQM, [1456](#)
 - M3D, [2703](#)
 - MATRIX, [1623](#)
 - MSET, [1670](#)
 - NSDPS, [1703](#)
 - ODP, [1816](#)
 - PENDTREE, [1945](#)
 - POINT, [2060](#)
 - PRIMARR, [2109](#)

- QUEUE, [2184](#)
- REGSET, [2286](#)
- RESULT, [2301](#)
- RGCHAIN, [2255](#)
- RMATRIX, [2246](#)
- ROUTINE, [2331](#)
- SET, [2372](#)
- SHDP, [2509](#)
- SPLTREE, [2518](#)
- SQMATRIX, [2548](#)
- SREGSET, [2535](#)
- STACK, [2563](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- TREE, [2741](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- WUTSET, [2959](#)
- EqTable, [698](#)
- EQTBL, [698](#)
 - ?., [699](#)
 - ?=, [699](#)
 - ?~=, [699](#)
 - #?, [699](#)
 - any?, [699](#)
 - bag, [699](#)
 - coerce, [699](#)
 - construct, [699](#)
 - convert, [699](#)
 - copy, [699](#)
 - count, [699](#)
 - dictionary, [699](#)
 - elt, [699](#)
 - empty, [699](#)
 - empty?, [699](#)
 - entries, [699](#)
 - entry?, [699](#)
 - eq?, [699](#)
 - eval, [699](#)
 - every?, [699](#)
 - extract, [699](#)
 - fill, [699](#)
 - find, [699](#)
 - first, [699](#)
 - hash, [699](#)
 - index?, [699](#)
 - indices, [699](#)
 - insert, [699](#)
 - inspect, [699](#)
 - key?, [699](#)
 - keys, [699](#)
 - latex, [699](#)
 - less?, [699](#)
 - map, [699](#)
 - maxIndex, [699](#)
 - member?, [699](#)
 - members, [699](#)
 - minIndex, [699](#)
 - more?, [699](#)
 - parts, [699](#)
 - qelt, [699](#)
 - qsetelt, [699](#)
 - reduce, [699](#)
 - remove, [699](#)
 - removeDuplicates, [699](#)
 - sample, [699](#)
 - search, [699](#)
 - select, [699](#)
 - setelt, [699](#)
 - size?, [699](#)
 - swap, [699](#)
 - table, [699](#)
- equality
 - BOP, [274](#)
- Equation, [691](#)
- equation
 - EQ, [691](#)
 - QEQUAT, [2169](#)
 - SD, [2572](#)
 - SEGBIND, [2364](#)
- erf
 - EXPR, [724](#)
- errorInfo
 - OMERR, [1792](#)

- errorKind
 - OMERR, [1792](#)
- escape
 - CHAR, [373](#)
- EuclideanModularRing, [701](#)
- euclideanSize
 - ALGFF, [28](#)
 - AN, [35](#)
 - BINARY, [292](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - EMR, [702](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FLOAT, [908](#)
 - FRAC, [986](#)
 - GSERIES, [1090](#)
 - HACKPI, [1978](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - IFF, [1284](#)
 - INT, [1362](#)
 - IPADIC, [1294](#)
 - IPF, [1303](#)
 - LAUPOL, [1422](#)
 - MCMPLEX, [1543](#)
 - MFLOAT, [1548](#)
 - MINT, [1557](#)
 - MODFIELD, [1638](#)
 - MODMON, [1632](#)
 - MYEXPR, [1688](#)
 - MYUP, [1695](#)
 - NSDPS, [1703](#)
 - NSUP, [1729](#)
 - ODR, [1858](#)
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
 - PADIC, [1881](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - PF, [2105](#)
 - PFR, [1914](#)
 - RADFF, [2194](#)
 - RADIX, [2206](#)
 - RECLOS, [2237](#)
 - ROMAN, [2326](#)
 - SAE, [2400](#)
 - SINT, [2412](#)
 - SULS, [2457](#)
 - SUP, [2467](#)
 - SUPEXP, [2481](#)
 - SUPXS, [2487](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UP, [2828](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
- eval
 - ALIST, [235](#)
 - AN, [35](#)
 - ARRAY1, [1774](#)
 - ARRAY2, [2763](#)
 - ASTACK, [65](#)
 - BBTREE, [253](#)
 - BINARY, [292](#)
 - BITS, [315](#)
 - BPADICRT, [263](#)
 - BSTREE, [302](#)
 - BTOUN, [306](#)
 - BTREE, [310](#)
 - CCLASS, [382](#)
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)
 - COMPLEX, [419](#)
 - DECIMAL, [465](#)
 - DEQUEUE, [512](#)
 - DFMAT, [599](#)
 - DFVEC, [605](#)

DHMATRIX, 491
DIRPROD, 547
DLIST, 460
DMP, 572
DPMM, 553
DPMO, 557
DSMP, 542
DSTREE, 535
EQ, 691
EQTBL, 699
EXPEXPAN, 712
EXPR, 724
EXPUPXS, 740
FARRAY, 886
FEXPR, 947
FR, 786
FRAC, 986
GDMP, 1052
GPOLSET, 1074
GSERIES, 1090
GSTBL, 1079
GTSET, 1083
HASHTBL, 1120
HDMP, 1180
HDP, 1173
HEAP, 1134
HEXADEC, 1143
IAN, 1276
IARRAY1, 1244
IARRAY2, 1256
IBITS, 1199
IFARRAY, 1222
IIARRAY2, 1290
ILIST, 1231
IMATRIX, 1239
INTABL, 1336
ISTRING, 1249
ISUPS, 1311
IVECTOR, 1260
KAFILE, 1414
LIB, 1429
LIST, 1504
LMDICT, 1515
LPOLY, 1447
LSQM, 1456
M3D, 2703
MATRIX, 1623
MCMPLX, 1543
MODMON, 1632
MOEBIUS, 1654
MPOLY, 1682
MSET, 1670
MYEXPR, 1688
MYUP, 1695
NSDPS, 1703
NSMP, 1714
NSUP, 1729
OCT, 1765
ODP, 1816
ODPOL, 1852
PADICRAT, 1886
PADICRC, 1891
PENDTREE, 1945
PERM, 1949
POINT, 2060
POLY, 2078
PRIMARR, 2109
QUAT, 2167
QUEUE, 2184
RADIX, 2206
REGSET, 2286
RESULT, 2301
RGCHAIN, 2255
RMATRIX, 2246
ROUTINE, 2331
SDPOL, 2386
SET, 2372
SHDP, 2509
SMP, 2423
SMTS, 2441
SPLTREE, 2518
SQMATRIX, 2548
SREGSET, 2535
STACK, 2563
STBL, 2450
STREAM, 2582
STRING, 2607
STRTBL, 2610
SULS, 2457
SUP, 2467
SUPEXP, 2481
SUPXS, 2487

- SUTS, [2497](#)
- TABLE, [2664](#)
- TREE, [2741](#)
- TS, [2671](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- VECTOR, [2942](#)
- WUTSET, [2959](#)
- evaluate
 - MODOP, [1647](#), [1804](#)
- evaluateInverse
 - MODOP, [1647](#), [1804](#)
- even?
 - AN, [35](#)
 - EXPR, [724](#)
 - FEXPR, [947](#)
 - IAN, [1276](#)
 - INT, [1362](#)
 - MINT, [1557](#)
 - MYEXPR, [1688](#)
 - PERM, [1949](#)
 - ROMAN, [2326](#)
 - SINT, [2412](#)
- evenlambert
 - UFPS, [2789](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- every?
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - ARRAY2, [2763](#)
 - ASTACK, [65](#)
 - BBTREE, [253](#)
 - BITS, [315](#)
 - BSTREE, [302](#)
 - BTOURN, [306](#)
 - BTREE, [310](#)
 - CCLASS, [382](#)
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)
 - DEQUEUE, [512](#)
 - DFMAT, [599](#)
 - DFVEC, [605](#)
 - DHMATRIX, [491](#)
 - DIRPROD, [547](#)
 - DLIST, [460](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DSTREE, [535](#)
 - EQTBL, [699](#)
 - FARRAY, [886](#)
 - GPOLSET, [1074](#)
 - GSTBL, [1079](#)
 - GTSET, [1083](#)
 - HASHTBL, [1120](#)
 - HDP, [1173](#)
 - HEAP, [1134](#)
 - IARRAY1, [1244](#)
 - IARRAY2, [1256](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - IIARRAY2, [1290](#)
 - ILIST, [1231](#)
 - IMATRIX, [1239](#)
 - INTABL, [1336](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - KAFILE, [1414](#)
 - LIB, [1429](#)
 - LIST, [1504](#)
 - LMDICT, [1515](#)
 - LSQM, [1456](#)
 - M3D, [2703](#)
 - MATRIX, [1623](#)
 - MSET, [1670](#)
 - NSDPS, [1703](#)
 - ODP, [1816](#)
 - PENDTREE, [1945](#)
 - POINT, [2060](#)
 - PRIMARR, [2109](#)

- QUEUE, [2184](#)
- REGSET, [2286](#)
- RESULT, [2301](#)
- RGCHAIN, [2255](#)
- RMATRIX, [2246](#)
- ROUTINE, [2331](#)
- SET, [2372](#)
- SHDP, [2509](#)
- SPLTREE, [2518](#)
- SQMATRIX, [2548](#)
- SREGSET, [2535](#)
- STACK, [2563](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- TREE, [2741](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- WUTSET, [2959](#)
- exactQuotient
 - NSMP, [1714](#)
- excepCoord
 - BLHN, [317](#)
 - BLQT, [319](#)
- excpDivV
 - IC, [1191](#)
 - INFCLSPS, [1271](#)
 - INFCLSPT, [1265](#)
- exists?
 - FNAME, [810](#)
- EXIT, [707](#)
 - ?=?, [707](#)
 - ?~=?, [707](#)
 - coerce, [707](#)
 - hash, [707](#)
 - latex, [707](#)
- Exit, [707](#)
- exp
 - ANTISYM, [40](#)
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FEXPR, [947](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - LEXP, [1435](#)
 - MCMPLX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXP, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
 - XPBWPOLYL, [2987](#)
- exp1
 - DFLOAT, [587](#)
 - FLOAT, [908](#)
- expand
 - FR, [786](#)
 - SEG, [2359](#)
 - UNISEG, [2896](#)
 - XPOLY, [2998](#)
 - XPOLY, [3013](#)
- EXPEXPAN, [711](#)
 - .?, [712](#)
 - ?<?, [712](#)
 - ?<=?, [712](#)
 - ?>?, [712](#)
 - ?>=?, [712](#)
 - ?**?, [712](#)
 - ?*?, [712](#)
 - ?+?, [712](#)
 - ?-?, [712](#)
 - ?..?, [712](#)

- [?/?](#), 712
- [?=?](#), 712
- [?^?](#), 712
- [?~=?](#), 712
- [?quo?](#), 712
- [?rem?](#), 712
- [0](#), 712
- [1](#), 712
- [abs](#), 712
- [associates?](#), 712
- [ceiling](#), 712
- [characteristic](#), 712
- [charthRoot](#), 712
- [coerce](#), 712
- [conditionP](#), 712
- [convert](#), 712
- [D](#), 712
- [denom](#), 712
- [denominator](#), 712
- [differentiate](#), 712
- [divide](#), 712
- [euclideanSize](#), 712
- [eval](#), 712
- [expressIdealMember](#), 712
- [exquo](#), 712
- [extendedEuclidean](#), 712
- [factor](#), 712
- [factorPolynomial](#), 712
- [factorSquareFreePolynomial](#), 712
- [floor](#), 712
- [fractionPart](#), 712
- [gcd](#), 712
- [gcdPolynomial](#), 712
- [hash](#), 712
- [init](#), 712
- [inv](#), 712
- [latex](#), 712
- [lcm](#), 712
- [limitPlus](#), 712
- [map](#), 712
- [max](#), 712
- [min](#), 712
- [multiEuclidean](#), 712
- [negative?](#), 712
- [nextItem](#), 712
- [numer](#), 712
- [numerator](#), 712
- [one?](#), 712
- [patternMatch](#), 712
- [positive?](#), 712
- [prime?](#), 712
- [principalIdeal](#), 712
- [random](#), 712
- [recip](#), 712
- [reducedSystem](#), 712
- [retract](#), 712
- [retractIfCan](#), 712
- [sample](#), 712
- [sign](#), 712
- [sizeLess?](#), 712
- [solveLinearPolynomialEquation](#), 712
- [squareFree](#), 712
- [squareFreePart](#), 712
- [squareFreePolynomial](#), 712
- [subtractIfCan](#), 712
- [unit?](#), 712
- [unitCanonical](#), 712
- [unitNormal](#), 712
- [wholePart](#), 712
- [zero?](#), 712
- [expIfCan](#)
 - [EFULS](#), 672
 - [EFUPXS](#), 681
- [explicitEntries?](#)
 - [NSDPS](#), 1703
 - [STREAM](#), 2582
- [explicitlyEmpty?](#)
 - [NSDPS](#), 1703
 - [STREAM](#), 2582
- [explicitlyFinite?](#)
 - [ALIST](#), 235
 - [DLIST](#), 460
 - [ILIST](#), 1231
 - [LIST](#), 1504
 - [NSDPS](#), 1703
 - [STREAM](#), 2582
- [exponent](#)
 - [DFLOAT](#), 587
 - [EXPUPXS](#), 740
 - [FLOAT](#), 908
 - [FR](#), 786
 - [MFLOAT](#), 1548

- MODMONOM, [1644](#)
- exponential
 - EXPUPXS, [740](#)
- ExponentialExpansion, [711](#)
- ExponentialOfUnivariatePuisseuxSeries, [739](#)
- exponentialOrder
 - EXPUPXS, [740](#)
- exponents
 - EAB, [743](#)
- EXPR, [723](#)
 - , [724](#)
 - ?<?, [724](#)
 - ?<=?, [724](#)
 - ?>?, [724](#)
 - ?>=?, [724](#)
 - ?**?, [724](#)
 - ?*?, [724](#)
 - ?+?, [724](#)
 - ?-?, [724](#)
 - ?/?, [724](#)
 - ?=?, [724](#)
 - ?^?, [724](#)
 - ?~=?, [724](#)
 - ?quo?, [724](#)
 - ?rem?, [724](#)
 - 0, [724](#)
 - 1, [724](#)
 - abs, [724](#)
 - acos, [724](#)
 - acosh, [724](#)
 - acot, [724](#)
 - acoth, [724](#)
 - acsc, [724](#)
 - acsch, [724](#)
 - airyAi, [724](#)
 - airyBi, [724](#)
 - applyQuote, [724](#)
 - asec, [724](#)
 - asech, [724](#)
 - asin, [724](#)
 - asinh, [724](#)
 - associates?, [724](#)
 - atan, [724](#)
 - atanh, [724](#)
 - belong?, [724](#)
 - besselI, [724](#)
 - besselJ, [724](#)
 - besselK, [724](#)
 - besselY, [724](#)
 - Beta, [724](#)
 - binomial, [724](#)
 - box, [724](#)
 - characteristic, [724](#)
 - charthRoot, [724](#)
 - Ci, [724](#)
 - coerce, [724](#)
 - commutator, [724](#)
 - conjugate, [724](#)
 - convert, [724](#)
 - cos, [724](#)
 - cosh, [724](#)
 - cot, [724](#)
 - coth, [724](#)
 - csc, [724](#)
 - csch, [724](#)
 - D, [724](#)
 - definingPolynomial, [724](#)
 - denom, [724](#)
 - denominator, [724](#)
 - differentiate, [724](#)
 - digamma, [724](#)
 - dilog, [724](#)
 - distribute, [724](#)
 - divide, [724](#)
 - Ei, [724](#)
 - elt, [724](#)
 - erf, [724](#)
 - euclideanSize, [724](#)
 - eval, [724](#)
 - even?, [724](#)
 - exp, [724](#)
 - expressIdealMember, [724](#)
 - exquo, [724](#)
 - extendedEuclidean, [724](#)
 - factor, [724](#)
 - factorial, [724](#)
 - factorials, [724](#)
 - factorPolynomial, [724](#)
 - freeOf?, [724](#)
 - Gamma, [724](#)
 - gcd, [724](#)
 - gcdPolynomial, [724](#)

- ground, 724
- ground?, 724
- hash, 724
- height, 724
- integral, 724
- inv, 724
- is?, 724
- isExpt, 724
- isMult, 724
- isPlus, 724
- isPower, 724
- isTimes, 724
- kernel, 724
- kernels, 724
- latex, 724
- lcm, 724
- li, 724
- log, 724
- mainKernel, 724
- map, 724
- max, 724
- min, 724
- minPoly, 724
- multiEuclidean, 724
- nthRoot, 724
- number?, 724
- numer, 724
- numerator, 724
- odd?, 724
- one?, 724
- operator, 724
- operators, 724
- paren, 724
- patternMatch, 724
- permutation, 724
- pi, 724
- polygamma, 724
- prime?, 724
- principalIdeal, 724
- product, 724
- recip, 724
- reduce, 724
- reducedSystem, 724
- retract, 724
- retractIfCan, 724
- rootOf, 724
- rootsOf, 724
- sample, 724
- sec, 724
- sech, 724
- Si, 724
- simplifyPower, 724
- sin, 724
- sinh, 724
- sizeLess?, 724
- sqr, 724
- squareFree, 724
- squareFreePart, 724
- squareFreePolynomial, 724
- subst, 724
- subtractIfCan, 724
- summation, 724
- tan, 724
- tanh, 724
- tower, 724
- unit?, 724
- unitCanonical, 724
- unitNormal, 724
- univariate, 724
- variables, 724
- zero?, 724
- zeroOf, 724
- zerosOf, 724
- expr
 - INFORM, 1343
 - SEX, 2392
 - SEXOF, 2394
- expressIdealMember
 - ALGFF, 28
 - AN, 35
 - BINARY, 292
 - BPADIC, 258
 - BPADICRT, 263
 - COMPLEX, 419
 - CONTFRAC, 445
 - DECIMAL, 465
 - DFLOAT, 587
 - EMR, 702
 - EXPEXPAN, 712
 - EXPR, 724
 - EXPUPXS, 740
 - FF, 820

- FFCG, 825
- FFCGP, 835
- FFCGX, 830
- FFNB, 860
- FFNBP, 872
- FFNBX, 866
- FFP, 851
- FFX, 846
- FLOAT, 908
- FRAC, 986
- GSERIES, 1090
- HACKPI, 1978
- HEXADEC, 1143
- IAN, 1276
- IFF, 1284
- INT, 1362
- IPADIC, 1294
- IPF, 1303
- LAUPOL, 1422
- MCMPLX, 1543
- MFLOAT, 1548
- MINT, 1557
- MODFIELD, 1638
- MODMON, 1632
- MYEXPR, 1688
- MYUP, 1695
- NSDPS, 1703
- NSUP, 1729
- ODR, 1858
- PACOFF, 2136
- PACRAT, 2146
- PADIC, 1881
- PADICRAT, 1886
- PADICRC, 1891
- PF, 2105
- PFR, 1914
- RADFF, 2194
- RADIX, 2206
- RECLOS, 2237
- ROMAN, 2326
- SAE, 2400
- SINT, 2412
- SULS, 2457
- SUP, 2467
- SUPEXPR, 2481
- SUPXS, 2487
- ULS, 2796
- ULSCONS, 2804
- UP, 2828
- UPXS, 2834
- UPXSCONS, 2842
- Expression, 723
- expres
 - HTMLFORM, 1152
 - MMLFORM, 1603
- EXPUPXS, 739
 - , 740
 - ?<?, 740
 - ?<=?, 740
 - ?>?, 740
 - ?>=?, 740
 - ?**?, 740
 - ?*?, 740
 - ?+?, 740
 - ?-?, 740
 - ?., 740
 - ?/?, 740
 - ?=?, 740
 - ?^?, 740
 - ?~=?, 740
 - ?quo?, 740
 - ?rem?, 740
 - 0, 740
 - 1, 740
 - acos, 740
 - acosh, 740
 - acot, 740
 - acoth, 740
 - acsc, 740
 - acsch, 740
 - approximate, 740
 - asec, 740
 - asech, 740
 - asin, 740
 - asinh, 740
 - associates?, 740
 - atan, 740
 - atanh, 740
 - center, 740
 - characteristic, 740
 - charthRoot, 740
 - coefficient, 740

- coerce, 740
- complete, 740
- cos, 740
- cosh, 740
- cot, 740
- coth, 740
- csc, 740
- csch, 740
- D, 740
- degree, 740
- differentiate, 740
- divide, 740
- euclideanSize, 740
- eval, 740
- exp, 740
- exponent, 740
- exponential, 740
- exponentialOrder, 740
- expressIdealMember, 740
- exquo, 740
- extend, 740
- extendedEuclidean, 740
- factor, 740
- gcd, 740
- gcdPolynomial, 740
- hash, 740
- integrate, 740
- inv, 740
- latex, 740
- lcm, 740
- leadingCoefficient, 740
- leadingMonomial, 740
- log, 740
- map, 740
- max, 740
- min, 740
- monomial, 740
- monomial?, 740
- multiEuclidean, 740
- multiplyExponents, 740
- nthRoot, 740
- one?, 740
- order, 740
- pi, 740
- pole?, 740
- prime?, 740
- principalIdeal, 740
- recip, 740
- reductum, 740
- sample, 740
- sec, 740
- sech, 740
- series, 740
- sin, 740
- sinh, 740
- sizeLess?, 740
- sqrt, 740
- squareFree, 740
- squareFreePart, 740
- subtractIfCan, 740
- tan, 740
- tanh, 740
- terms, 740
- truncate, 740
- unit?, 740
- unitCanonical, 740
- unitNormal, 740
- variable, 740
- variables, 740
- zero?, 740
- exQuo
 - EMR, 702
 - MODFIELD, 1638
 - MODRING, 1641
- exquo
 - ALGFF, 28
 - AN, 35
 - BINARY, 292
 - BPADIC, 258
 - BPADICRT, 263
 - CDFMAT, 426
 - COMPLEX, 419
 - CONTFRAC, 445
 - DECIMAL, 465
 - DFLOAT, 587
 - DFMAT, 599
 - DHMATRIX, 491
 - DIRRING, 564
 - DMP, 572
 - DSMP, 542
 - EMR, 702
 - EXPEXPAN, 712

EXPR, [724](#)
 EXPUPXS, [740](#)
 FF, [820](#)
 FFCG, [825](#)
 FFCGP, [835](#)
 FFCGX, [830](#)
 FFNB, [860](#)
 FFNB, [872](#)
 FFNBX, [866](#)
 FFP, [851](#)
 FFX, [846](#)
 FLOAT, [908](#)
 FR, [786](#)
 FRAC, [986](#)
 GDMP, [1052](#)
 GSERIES, [1090](#)
 HACKPI, [1978](#)
 HDMP, [1180](#)
 HEXADEC, [1143](#)
 IAN, [1276](#)
 IFF, [1284](#)
 IMATRIX, [1239](#)
 INT, [1362](#)
 INTRVL, [1384](#)
 IPADIC, [1294](#)
 IPF, [1303](#)
 ISUPS, [1311](#)
 ITAYLOR, [1338](#)
 LAUPOL, [1422](#)
 LODO, [1469](#)
 LODO1, [1479](#)
 LODO2, [1491](#)
 LSQM, [1456](#)
 MATRIX, [1623](#)
 MCMPLEX, [1543](#)
 MFLOAT, [1548](#)
 MINT, [1557](#)
 MODFIELD, [1638](#)
 MODMON, [1632](#)
 MPOLY, [1682](#)
 MYEXPR, [1688](#)
 MYUP, [1695](#)
 NNI, [1739](#)
 NSDPS, [1703](#)
 NSMP, [1714](#)
 NSUP, [1729](#)
 ODPOL, [1852](#)
 ODR, [1858](#)
 ORESUP, [2492](#)
 OREUP, [2872](#)
 OUTFORM, [1868](#)
 PACOFF, [2136](#)
 PACRAT, [2146](#)
 PADIC, [1881](#)
 PADICRAT, [1886](#)
 PADICRC, [1891](#)
 PF, [2105](#)
 PFR, [1914](#)
 POLY, [2078](#)
 PR, [2092](#)
 RADFF, [2194](#)
 RADIX, [2206](#)
 RECLOS, [2237](#)
 RMATRIX, [2246](#)
 ROMAN, [2326](#)
 SAE, [2400](#)
 SDPOL, [2386](#)
 SINT, [2412](#)
 SMP, [2423](#)
 SMTS, [2441](#)
 SQMATRIX, [2548](#)
 SULS, [2457](#)
 SUP, [2467](#)
 SUPEXP, [2481](#)
 SUPXS, [2487](#)
 SUTS, [2497](#)
 SYMPOLY, [2654](#)
 TS, [2671](#)
 U16MAT, [2906](#)
 U32MAT, [2911](#)
 U8MAT, [2902](#)
 UFPS, [2789](#)
 ULS, [2796](#)
 ULSCONS, [2804](#)
 UP, [2828](#)
 UPXS, [2834](#)
 UPXSCONS, [2842](#)
 UPXSING, [2852](#)
 UTS, [2877](#)
 UTSZ, [2887](#)
 ExtAlgBasis, [743](#)
 extDegree

- PACOFF, [2136](#)
- PACRAT, [2146](#)
- extend
 - BPADIC, [258](#)
 - CONTFRAC, [445](#)
 - EXPUPXS, [740](#)
 - GSERIES, [1090](#)
 - GTSET, [1083](#)
 - IPADIC, [1294](#)
 - ISUPS, [1311](#)
 - NSDPS, [1703](#)
 - PADIC, [1881](#)
 - POINT, [2060](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SMTS, [2441](#)
 - SREGSET, [2535](#)
 - STREAM, [2582](#)
 - SULS, [2457](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
 - WUTSET, [2959](#)
- extendedEuclidean
 - ALGFF, [28](#)
 - AN, [35](#)
 - BINARY, [292](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - EMR, [702](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FLOAT, [908](#)
 - FRAC, [986](#)
 - GSERIES, [1090](#)
 - HACKPI, [1978](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - IFF, [1284](#)
 - INT, [1362](#)
 - IPADIC, [1294](#)
 - IPF, [1303](#)
 - LAUPOL, [1422](#)
 - MCMPLEX, [1543](#)
 - MFLOAT, [1548](#)
 - MINT, [1557](#)
 - MODFIELD, [1638](#)
 - MODMON, [1632](#)
 - MYEXPR, [1688](#)
 - MYUP, [1695](#)
 - NSDPS, [1703](#)
 - NSUP, [1729](#)
 - ODR, [1858](#)
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
 - PADIC, [1881](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - PF, [2105](#)
 - PFR, [1914](#)
 - RADFF, [2194](#)
 - RADIX, [2206](#)
 - RECLOS, [2237](#)
 - ROMAN, [2326](#)
 - SAE, [2400](#)
 - SINT, [2412](#)
 - SULS, [2457](#)
 - SUP, [2467](#)
 - SUPEXP, [2481](#)
 - SUPXS, [2487](#)
 - ULS, [2796](#)

- ULSCONS, [2804](#)
 - UP, [2828](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
- extendedResultant
 - NSUP, [1729](#)
- extendedSubResultantGcd
 - NSMP, [1714](#)
 - NSUP, [1729](#)
- extendIfCan
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- extension
 - FNAME, [810](#)
- extensionDegree
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
 - PF, [2105](#)
- exteriorDifferential
 - DERHAM, [530](#)
- external?
 - FT, [971](#)
- externalList
 - SYMTAB, [2648](#)
- extractClosed
 - SUBSPACE, [2614](#)
- extractIndex
 - SUBSPACE, [2614](#)
- extractPoint
 - SUBSPACE, [2614](#)
- extractProperty
 - SUBSPACE, [2614](#)
- extractSplittingLeaf
 - SPLTREE, [2518](#)
- eyeDistance
 - VIEW3D, [2710](#)
- factor
 - ALGFF, [28](#)
 - AN, [35](#)
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FLOAT, [908](#)
 - FR, [786](#)
 - FRAC, [986](#)
 - GDMP, [1052](#)
 - GSERIES, [1090](#)
 - HACKPI, [1978](#)
 - HDMP, [1180](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - IFF, [1284](#)
 - INT, [1362](#)
 - IPF, [1303](#)
 - LWORD, [1532](#)
 - MCMPLEX, [1543](#)
 - MFLOAT, [1548](#)
 - MINT, [1557](#)
 - MODFIELD, [1638](#)
 - MODMON, [1632](#)

- MPOLY, [1682](#)
- MYEXPR, [1688](#)
- MYUP, [1695](#)
- NSDPS, [1703](#)
- NSMP, [1714](#)
- NSUP, [1729](#)
- ODPOL, [1852](#)
- ODR, [1858](#)
- PACOFF, [2136](#)
- PACRAT, [2146](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PF, [2105](#)
- PFR, [1914](#)
- POLY, [2078](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- ROMAN, [2326](#)
- SAE, [2400](#)
- SDPOL, [2386](#)
- SINT, [2412](#)
- SMP, [2423](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- factorAndSplit
 - EQ, [691](#)
- Factored, [786](#)
- factorial
 - EXPR, [724](#)
 - INT, [1362](#)
 - MINT, [1557](#)
 - MYEXPR, [1688](#)
 - ROMAN, [2326](#)
 - SINT, [2412](#)
- factorials
 - EXPR, [724](#)
 - MYEXPR, [1688](#)
- factorList
 - FR, [786](#)
- factorPolynomial
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - DECIMAL, [465](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - FRAC, [986](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - HEXADEC, [1143](#)
 - MCMPLEX, [1543](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - POLY, [2078](#)
 - RADIX, [2206](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SULS, [2457](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UP, [2828](#)
- factors
 - FGROUP, [1010](#)
 - FMONOID, [1021](#)
 - FR, [786](#)
 - OFMONOID, [1829](#)
- factorsOfCyclicGroupSize
 - ALGFF, [28](#)
 - COMPLEX, [419](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)

- FFNBP, [872](#)
- FFNBX, [866](#)
- FFP, [851](#)
- FFX, [846](#)
- IFF, [1284](#)
- IPF, [1303](#)
- MCMPLX, [1543](#)
- PACOFF, [2136](#)
- PF, [2105](#)
- RADFF, [2194](#)
- SAE, [2400](#)
- factorSquareFreePolynomial
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - DECIMAL, [465](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPEXPAN, [712](#)
 - FRAC, [986](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - HEXADEC, [1143](#)
 - MCMPLX, [1543](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - POLY, [2078](#)
 - RADIX, [2206](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SULS, [2457](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UP, [2828](#)
- FAGROUP, [1004](#)
 - , [1004](#)
 - ?<?, [1004](#)
 - ?<=?, [1004](#)
 - ?>?, [1004](#)
 - ?>=?, [1004](#)
 - ?*?, [1004](#)
 - ?+?, [1004](#)
 - ?-?, [1004](#)
 - ?=?, [1004](#)
 - ?~=?, [1004](#)
 - 0, [1004](#)
 - coefficient, [1004](#)
 - coerce, [1004](#)
 - hash, [1004](#)
 - highCommonTerms, [1004](#)
 - latex, [1004](#)
 - mapCoef, [1004](#)
 - mapGen, [1004](#)
 - max, [1004](#)
 - min, [1004](#)
 - nthCoef, [1004](#)
 - nthFactor, [1004](#)
 - retract, [1004](#)
 - retractIfCan, [1004](#)
 - sample, [1004](#)
 - size, [1004](#)
 - subtractIfCan, [1004](#)
 - terms, [1004](#)
 - zero?, [1004](#)
- failed
 - PATLRES, [1937](#)
 - PATRES, [1940](#)
- failed?
 - PATLRES, [1937](#)
 - PATRES, [1940](#)
- false
 - BOOLEAN, [322](#)
- FAMONOID, [1007](#)
 - ?*?, [1007](#)
 - ?+?, [1007](#)
 - ?=?, [1007](#)
 - ?~=?, [1007](#)
 - 0, [1007](#)
 - coefficient, [1007](#)
 - coerce, [1007](#)
 - hash, [1007](#)
 - highCommonTerms, [1007](#)
 - latex, [1007](#)
 - mapCoef, [1007](#)

- mapGen, 1007
- nthCoef, 1007
- nthFactor, 1007
- retract, 1007
- retractIfCan, 1007
- sample, 1007
- size, 1007
- subtractIfCan, 1007
- terms, 1007
- zero?, 1007
- FARRAY, 885
 - ?<?, 886
 - ?<=?, 886
 - ?>?, 886
 - ?>=?, 886
 - ?..?, 886
 - ?=?, 886
 - ?~=?, 886
 - #?, 886
 - any?, 886
 - coerce, 886
 - concat, 886
 - construct, 886
 - convert, 886
 - copy, 886
 - copyInto, 886
 - count, 886
 - delete, 886
 - elt, 886
 - empty, 886
 - empty?, 886
 - entries, 886
 - entry?, 886
 - eq?, 886
 - eval, 886
 - every?, 886
 - fill, 886
 - find, 886
 - first, 886
 - flexibleArray, 886
 - hash, 886
 - index?, 886
 - indices, 886
 - insert, 886
 - latex, 886
 - less?, 886
 - map, 886
 - max, 886
 - maxIndex, 886
 - member?, 886
 - members, 886
 - merge, 886
 - min, 886
 - minIndex, 886
 - more?, 886
 - new, 886
 - parts, 886
 - physicalLength, 886
 - position, 886
 - qelt, 886
 - qsetelt, 886
 - reduce, 886
 - remove, 886
 - removeDuplicates, 886
 - reverse, 886
 - sample, 886
 - select, 886
 - setelt, 886
 - shrinkable, 886
 - size?, 886
 - sort, 886
 - sorted?, 886
 - swap, 886
- FC, 931
 - ?=?, 931
 - ? =?, 931
 - assign, 931
 - block, 931
 - call, 931
 - code, 931
 - coerce, 931
 - comment, 931
 - common, 931
 - cond, 931
 - continue, 931
 - forLoop, 931
 - getCode, 931
 - goto, 931
 - hash, 931
 - latex, 931
 - operation, 931
 - printCode, 931

- printStatement, 931
- repeatUntilLoop, 931
- returns, 931
- save, 931
- setLabelValue, 931
- stop, 931
- whileLoop, 931
- FCOMP, 975
 - ?<?, 975
 - ?<=?, 975
 - ?>?, 975
 - ?>=?, 975
 - ?=?, 975
 - ?~=?, 975
 - argument, 975
 - coerce, 975
 - cos, 975
 - hash, 975
 - latex, 975
 - max, 975
 - min, 975
 - sin, 975
 - sin?, 975
- FDIV, 813
 - , 813
 - ?*?, 813
 - ?+?, 813
 - ?-?, 813
 - ?=?, 813
 - ?~=?, 813
 - 0, 813
 - coerce, 813
 - decompose, 813
 - divisor, 813
 - finiteBasis, 813
 - generator, 813
 - hash, 813
 - ideal, 813
 - latex, 813
 - lSpaceBasis, 813
 - principal?, 813
 - reduce, 813
 - sample, 813
 - subtractIfCan, 813
 - zero?, 813
- FEXPR, 946
 - , 947
 - ?<?, 947
 - ?<=?, 947
 - ?>?, 947
 - ?>=?, 947
 - ?**?, 947
 - ?*?, 947
 - ?+?, 947
 - ?-?, 947
 - ?=?, 947
 - ?^?, 947
 - ?~=?, 947
 - 0, 947
 - 1, 947
 - abs, 947
 - acos, 947
 - asin, 947
 - atan, 947
 - belong?, 947
 - box, 947
 - characteristic, 947
 - coerce, 947
 - cos, 947
 - cosh, 947
 - D, 947
 - definingPolynomial, 947
 - differentiate, 947
 - distribute, 947
 - elt, 947
 - eval, 947
 - even?, 947
 - exp, 947
 - freeOf?, 947
 - hash, 947
 - height, 947
 - is?, 947
 - kernel, 947
 - kernels, 947
 - latex, 947
 - log, 947
 - log10, 947
 - mainKernel, 947
 - map, 947
 - max, 947
 - min, 947
 - minPoly, 947

- odd?, 947
- one?, 947
- operator, 947
- operators, 947
- paren, 947
- pi, 947
- recip, 947
- retract, 947
- retractIfCan, 947
- sample, 947
- sin, 947
- sinh, 947
- sqrt, 947
- subst, 947
- subtractIfCan, 947
- tan, 947
- tanh, 947
- tower, 947
- useNagFunctions, 947
- variables, 947
- zero?, 947
- FF, 819
 - ?, 820
 - ?**?, 820
 - ?*?, 820
 - ?+?, 820
 - ?-?, 820
 - ?/?, 820
 - ?=?, 820
 - ?^?, 820
 - ?~=?, 820
 - ?quo?, 820
 - ?rem?, 820
 - 0, 820
 - 1, 820
 - algebraic?, 820
 - associates?, 820
 - basis, 820
 - characteristic, 820
 - charthRoot, 820
 - coerce, 820
 - conditionP, 820
 - coordinates, 820
 - createNormalElement, 820
 - createPrimitiveElement, 820
 - D, 820
 - definingPolynomial, 820
 - degree, 820
 - differentiate, 820
 - dimension, 820
 - discreteLog, 820
 - divide, 820
 - euclideanSize, 820
 - expressIdealMember, 820
 - exquo, 820
 - extendedEuclidean, 820
 - extensionDegree, 820
 - factor, 820
 - factorsOfCyclicGroupSize, 820
 - Frobenius, 820
 - gcd, 820
 - gcdPolynomial, 820
 - generator, 820
 - hash, 820
 - index, 820
 - inGroundField?, 820
 - init, 820
 - inv, 820
 - latex, 820
 - lcm, 820
 - linearAssociatedExp, 820
 - linearAssociatedLog, 820
 - linearAssociatedOrder, 820
 - lookup, 820
 - minimalPolynomial, 820
 - multiEuclidean, 820
 - nextItem, 820
 - norm, 820
 - normal?, 820
 - normalElement, 820
 - one?, 820
 - order, 820
 - prime?, 820
 - primeFrobenius, 820
 - primitive?, 820
 - primitiveElement, 820
 - principalIdeal, 820
 - random, 820
 - recip, 820
 - representationType, 820
 - represents, 820
 - retract, 820

- retractIfCan, 820
- sample, 820
- size, 820
- sizeLess?, 820
- squareFree, 820
- squareFreePart, 820
- subtractIfCan, 820
- tableForDiscreteLogarithm, 820
- trace, 820
- transcendenceDegree, 820
- transcendent?, 820
- unit?, 820
- unitCanonical, 820
- unitNormal, 820
- zero?, 820
- FFCG, 824
- ?, 825
- ?**?, 825
- ?*?, 825
- ?+?, 825
- ?-?, 825
- ?/?, 825
- ?=?, 825
- ?^?, 825
- ?~=?, 825
- ?quo?, 825
- ?rem?, 825
- 0, 825
- 1, 825
- algebraic?, 825
- associates?, 825
- basis, 825
- characteristic, 825
- charthRoot, 825
- coerce, 825
- conditionP, 825
- coordinates, 825
- createNormalElement, 825
- createPrimitiveElement, 825
- D, 825
- definingPolynomial, 825
- degree, 825
- differentiate, 825
- dimension, 825
- discreteLog, 825
- divide, 825
- euclideanSize, 825
- expressIdealMember, 825
- exquo, 825
- extendedEuclidean, 825
- extensionDegree, 825
- factor, 825
- factorsOfCyclicGroupSize, 825
- Frobenius, 825
- gcd, 825
- gcdPolynomial, 825
- generator, 825
- getZechTable, 825
- hash, 825
- index, 825
- inGroundField?, 825
- init, 825
- inv, 825
- latex, 825
- lcm, 825
- linearAssociatedExp, 825
- linearAssociatedLog, 825
- linearAssociatedOrder, 825
- lookup, 825
- minimalPolynomial, 825
- multiEuclidean, 825
- nextItem, 825
- norm, 825
- normal?, 825
- normalElement, 825
- one?, 825
- order, 825
- prime?, 825
- primeFrobenius, 825
- primitive?, 825
- primitiveElement, 825
- principalIdeal, 825
- random, 825
- recip, 825
- representationType, 825
- represents, 825
- retract, 825
- retractIfCan, 825
- sample, 825
- size, 825
- sizeLess?, 825
- squareFree, 825

- squareFreePart, 825
- subtractIfCan, 825
- tableForDiscreteLogarithm, 825
- trace, 825
- transcendenceDegree, 825
- transcendent?, 825
- unit?, 825
- unitCanonical, 825
- unitNormal, 825
- zero?, 825
- FFCGP, 834
 - ?, 835
 - ?**?, 835
 - ?*?, 835
 - ?+?, 835
 - ?-?, 835
 - ?/? , 835
 - ?=?, 835
 - ?^?, 835
 - ?~=?, 835
 - ?quo?, 835
 - ?rem?, 835
 - 0, 835
 - 1, 835
 - algebraic?, 835
 - associates?, 835
 - basis, 835
 - characteristic, 835
 - charthRoot, 835
 - coerce, 835
 - conditionP, 835
 - coordinates, 835
 - createNormalElement, 835
 - createPrimitiveElement, 835
 - D, 835
 - definingPolynomial, 835
 - degree, 835
 - differentiate, 835
 - dimension, 835
 - discreteLog, 835
 - divide, 835
 - euclideanSize, 835
 - expressIdealMember, 835
 - exquo, 835
 - extendedEuclidean, 835
 - extensionDegree, 835
 - factor, 835
 - factorsOfCyclicGroupSize, 835
 - Frobenius, 835
 - gcd, 835
 - gcdPolynomial, 835
 - generator, 835
 - getZechTable, 835
 - hash, 835
 - index, 835
 - inGroundField?, 835
 - init, 835
 - inv, 835
 - latex, 835
 - lcm, 835
 - linearAssociatedExp, 835
 - linearAssociatedLog, 835
 - linearAssociatedOrder, 835
 - lookup, 835
 - minimalPolynomial, 835
 - multiEuclidean, 835
 - nextItem, 835
 - norm, 835
 - normal?, 835
 - normalElement, 835
 - one?, 835
 - order, 835
 - prime?, 835
 - primeFrobenius, 835
 - primitive?, 835
 - primitiveElement, 835
 - principalIdeal, 835
 - random, 835
 - recip, 835
 - representationType, 835
 - represents, 835
 - retract, 835
 - retractIfCan, 835
 - sample, 835
 - size, 835
 - sizeLess?, 835
 - squareFree, 835
 - squareFreePart, 835
 - subtractIfCan, 835
 - tableForDiscreteLogarithm, 835
 - trace, 835
 - transcendenceDegree, 835

- transcendent?, 835
- unit?, 835
- unitCanonical, 835
- unitNormal, 835
- zero?, 835
- FFCGX, 829
- ?, 830
- ?**?, 830
- ?*?, 830
- ?+?, 830
- ?-?, 830
- ?/?, 830
- ?=?, 830
- ?^?, 830
- ?~=?, 830
- ?quo?, 830
- ?rem?, 830
- 0, 830
- 1, 830
- algebraic?, 830
- associates?, 830
- basis, 830
- characteristic, 830
- charthRoot, 830
- coerce, 830
- conditionP, 830
- coordinates, 830
- createNormalElement, 830
- createPrimitiveElement, 830
- D, 830
- definingPolynomial, 830
- degree, 830
- differentiate, 830
- dimension, 830
- discreteLog, 830
- divide, 830
- euclideanSize, 830
- expressIdealMember, 830
- exquo, 830
- extendedEuclidean, 830
- extensionDegree, 830
- factor, 830
- factorsOfCyclicGroupSize, 830
- Frobenius, 830
- gcd, 830
- gcdPolynomial, 830
- generator, 830
- getZechTable, 830
- hash, 830
- index, 830
- inGroundField?, 830
- init, 830
- inv, 830
- latex, 830
- lcm, 830
- linearAssociatedExp, 830
- linearAssociatedLog, 830
- linearAssociatedOrder, 830
- lookup, 830
- minimalPolynomial, 830
- multiEuclidean, 830
- nextItem, 830
- norm, 830
- normal?, 830
- normalElement, 830
- one?, 830
- order, 830
- prime?, 830
- primeFrobenius, 830
- primitive?, 830
- primitiveElement, 830
- principalIdeal, 830
- random, 830
- recip, 830
- representationType, 830
- represents, 830
- retract, 830
- retractIfCan, 830
- sample, 830
- size, 830
- sizeLess?, 830
- squareFree, 830
- squareFreePart, 830
- subtractIfCan, 830
- tableForDiscreteLogarithm, 830
- trace, 830
- transcendenceDegree, 830
- transcendent?, 830
- unit?, 830
- unitCanonical, 830
- unitNormal, 830
- zero?, 830

- FFNB, 859
 - ?, 860
 - ***?, 860
 - *?*, 860
 - ?+?, 860
 - ?-?, 860
 - ?/? , 860
 - ?=?, 860
 - ?^?, 860
 - ?~=?, 860
 - ?quo?, 860
 - ?rem?, 860
 - 0, 860
 - 1, 860
 - algebraic?, 860
 - associates?, 860
 - basis, 860
 - characteristic, 860
 - charthRoot, 860
 - coerce, 860
 - conditionP, 860
 - coordinates, 860
 - createNormalElement, 860
 - createPrimitiveElement, 860
 - D, 860
 - definingPolynomial, 860
 - degree, 860
 - differentiate, 860
 - dimension, 860
 - discreteLog, 860
 - divide, 860
 - euclideanSize, 860
 - expressIdealMember, 860
 - exquo, 860
 - extendedEuclidean, 860
 - extensionDegree, 860
 - factor, 860
 - factorsOfCyclicGroupSize, 860
 - Frobenius, 860
 - gcd, 860
 - gcdPolynomial, 860
 - generator, 860
 - getMultiplicationMatrix, 860
 - getMultiplicationTable, 860
 - hash, 860
 - index, 860
 - inGroundField?, 860
 - init, 860
 - inv, 860
 - latex, 860
 - lcm, 860
 - linearAssociatedExp, 860
 - linearAssociatedLog, 860
 - linearAssociatedOrder, 860
 - lookup, 860
 - minimalPolynomial, 860
 - multiEuclidean, 860
 - nextItem, 860
 - norm, 860
 - normal?, 860
 - normalElement, 860
 - one?, 860
 - order, 860
 - prime?, 860
 - primeFrobenius, 860
 - primitive?, 860
 - primitiveElement, 860
 - principalIdeal, 860
 - random, 860
 - recip, 860
 - representationType, 860
 - represents, 860
 - retract, 860
 - retractIfCan, 860
 - sample, 860
 - size, 860
 - sizeLess?, 860
 - sizeMultiplication, 860
 - squareFree, 860
 - squareFreePart, 860
 - subtractIfCan, 860
 - tableForDiscreteLogarithm, 860
 - trace, 860
 - transcendenceDegree, 860
 - transcendent?, 860
 - unit?, 860
 - unitCanonical, 860
 - unitNormal, 860
 - zero?, 860
- FFNBP, 871
 - ?, 872
 - ***?, 872

- [?*?, 872](#)
- [?+?, 872](#)
- [?-?, 872](#)
- [?/? , 872](#)
- [?=?, 872](#)
- [?^?, 872](#)
- [?~=?, 872](#)
- [?quo?, 872](#)
- [?rem?, 872](#)
- [0, 872](#)
- [1, 872](#)
- [algebraic?, 872](#)
- [associates?, 872](#)
- [basis, 872](#)
- [characteristic, 872](#)
- [charthRoot, 872](#)
- [coerce, 872](#)
- [conditionP, 872](#)
- [coordinates, 872](#)
- [createNormalElement, 872](#)
- [createPrimitiveElement, 872](#)
- [D, 872](#)
- [definingPolynomial, 872](#)
- [degree, 872](#)
- [differentiate, 872](#)
- [dimension, 872](#)
- [discreteLog, 872](#)
- [divide, 872](#)
- [euclideanSize, 872](#)
- [expressIdealMember, 872](#)
- [exquo, 872](#)
- [extendedEuclidean, 872](#)
- [extensionDegree, 872](#)
- [factor, 872](#)
- [factorsOfCyclicGroupSize, 872](#)
- [Frobenius, 872](#)
- [gcd, 872](#)
- [gcdPolynomial, 872](#)
- [generator, 872](#)
- [getMultiplicationMatrix, 872](#)
- [getMultiplicationTable, 872](#)
- [hash, 872](#)
- [index, 872](#)
- [inGroundField?, 872](#)
- [init, 872](#)
- [inv, 872](#)
- [latex, 872](#)
- [lcm, 872](#)
- [linearAssociatedExp, 872](#)
- [linearAssociatedLog, 872](#)
- [linearAssociatedOrder, 872](#)
- [lookup, 872](#)
- [minimalPolynomial, 872](#)
- [multiEuclidean, 872](#)
- [nextItem, 872](#)
- [norm, 872](#)
- [normal?, 872](#)
- [normalElement, 872](#)
- [one?, 872](#)
- [order, 872](#)
- [prime?, 872](#)
- [primeFrobenius, 872](#)
- [primitive?, 872](#)
- [primitiveElement, 872](#)
- [principalIdeal, 872](#)
- [random, 872](#)
- [recip, 872](#)
- [representationType, 872](#)
- [represents, 872](#)
- [retract, 872](#)
- [retractIfCan, 872](#)
- [sample, 872](#)
- [size, 872](#)
- [sizeLess?, 872](#)
- [sizeMultiplication, 872](#)
- [squareFree, 872](#)
- [squareFreePart, 872](#)
- [subtractIfCan, 872](#)
- [tableForDiscreteLogarithm, 872](#)
- [trace, 872](#)
- [transcendenceDegree, 872](#)
- [transcendent?, 872](#)
- [unit?, 872](#)
- [unitCanonical, 872](#)
- [unitNormal, 872](#)
- [zero?, 872](#)
- [FFNBX, 865](#)
- [-?, 866](#)
- [***?, 866](#)
- [*?*, 866](#)
- [?+?, 866](#)
- [?-?, 866](#)

- [?/?](#), 866
- [?=?](#), 866
- [?^?](#), 866
- [?~=?](#), 866
- [?quo?](#), 866
- [?rem?](#), 866
- [0](#), 866
- [1](#), 866
- [algebraic?](#), 866
- [associates?](#), 866
- [basis](#), 866
- [characteristic](#), 866
- [charthRoot](#), 866
- [coerce](#), 866
- [conditionP](#), 866
- [coordinates](#), 866
- [createNormalElement](#), 866
- [createPrimitiveElement](#), 866
- [D](#), 866
- [definingPolynomial](#), 866
- [degree](#), 866
- [differentiate](#), 866
- [dimension](#), 866
- [discreteLog](#), 866
- [divide](#), 866
- [euclideanSize](#), 866
- [expressIdealMember](#), 866
- [exquo](#), 866
- [extendedEuclidean](#), 866
- [extensionDegree](#), 866
- [factor](#), 866
- [factorsOfCyclicGroupSize](#), 866
- [Frobenius](#), 866
- [gcd](#), 866
- [gcdPolynomial](#), 866
- [generator](#), 866
- [getMultiplicationMatrix](#), 866
- [getMultiplicationTable](#), 866
- [hash](#), 866
- [index](#), 866
- [inGroundField?](#), 866
- [init](#), 866
- [inv](#), 866
- [latex](#), 866
- [lcm](#), 866
- [linearAssociatedExp](#), 866
- [linearAssociatedLog](#), 866
- [linearAssociatedOrder](#), 866
- [lookup](#), 866
- [minimalPolynomial](#), 866
- [multiEuclidean](#), 866
- [nextItem](#), 866
- [norm](#), 866
- [normal?](#), 866
- [normalElement](#), 866
- [one?](#), 866
- [order](#), 866
- [prime?](#), 866
- [primeFrobenius](#), 866
- [primitive?](#), 866
- [primitiveElement](#), 866
- [principalIdeal](#), 866
- [random](#), 866
- [recip](#), 866
- [representationType](#), 866
- [represents](#), 866
- [retract](#), 866
- [retractIfCan](#), 866
- [sample](#), 866
- [size](#), 866
- [sizeLess?](#), 866
- [sizeMultiplication](#), 866
- [squareFree](#), 866
- [squareFreePart](#), 866
- [subtractIfCan](#), 866
- [tableForDiscreteLogarithm](#), 866
- [trace](#), 866
- [transcendenceDegree](#), 866
- [transcendent?](#), 866
- [unit?](#), 866
- [unitCanonical](#), 866
- [unitNormal](#), 866
- [zero?](#), 866
- [FFP](#), 850
 - [-?](#), 851
 - [?**?](#), 851
 - [?*?](#), 851
 - [?+?](#), 851
 - [?-?](#), 851
 - [?/?](#), 851
 - [?=?](#), 851
 - [?^?](#), 851

- [?~=?](#), 851
- [?quo?](#), 851
- [?rem?](#), 851
- [0](#), 851
- [1](#), 851
- [algebraic?](#), 851
- [associates?](#), 851
- [basis](#), 851
- [characteristic](#), 851
- [charthRoot](#), 851
- [coerce](#), 851
- [conditionP](#), 851
- [coordinates](#), 851
- [createNormalElement](#), 851
- [createPrimitiveElement](#), 851
- [D](#), 851
- [definingPolynomial](#), 851
- [degree](#), 851
- [differentiate](#), 851
- [dimension](#), 851
- [discreteLog](#), 851
- [divide](#), 851
- [euclideanSize](#), 851
- [expressIdealMember](#), 851
- [exquo](#), 851
- [extendedEuclidean](#), 851
- [extensionDegree](#), 851
- [factor](#), 851
- [factorsOfCyclicGroupSize](#), 851
- [Frobenius](#), 851
- [gcd](#), 851
- [gcdPolynomial](#), 851
- [generator](#), 851
- [hash](#), 851
- [index](#), 851
- [inGroundField?](#), 851
- [init](#), 851
- [inv](#), 851
- [latex](#), 851
- [lcm](#), 851
- [linearAssociatedExp](#), 851
- [linearAssociatedLog](#), 851
- [linearAssociatedOrder](#), 851
- [lookup](#), 851
- [minimalPolynomial](#), 851
- [multiEuclidean](#), 851
- [nextItem](#), 851
- [norm](#), 851
- [normal?](#), 851
- [normalElement](#), 851
- [one?](#), 851
- [order](#), 851
- [prime?](#), 851
- [primeFrobenius](#), 851
- [primitive?](#), 851
- [primitiveElement](#), 851
- [principalIdeal](#), 851
- [random](#), 851
- [recip](#), 851
- [representationType](#), 851
- [represents](#), 851
- [retract](#), 851
- [retractIfCan](#), 851
- [sample](#), 851
- [size](#), 851
- [sizeLess?](#), 851
- [squareFree](#), 851
- [squareFreePart](#), 851
- [subtractIfCan](#), 851
- [tableForDiscreteLogarithm](#), 851
- [trace](#), 851
- [transcendenceDegree](#), 851
- [transcendent?](#), 851
- [unit?](#), 851
- [unitCanonical](#), 851
- [unitNormal](#), 851
- [zero?](#), 851
- [FFX](#), 845
- [-?](#), 846
- [?***?](#), 846
- [?*?](#), 846
- [?+?](#), 846
- [?-?](#), 846
- [?/?](#), 846
- [?=?](#), 846
- [?^?](#), 846
- [?~=?](#), 846
- [?quo?](#), 846
- [?rem?](#), 846
- [0](#), 846
- [1](#), 846
- [algebraic?](#), 846

- associates?, 846
- basis, 846
- characteristic, 846
- charthRoot, 846
- coerce, 846
- conditionP, 846
- coordinates, 846
- createNormalElement, 846
- createPrimitiveElement, 846
- D, 846
- definingPolynomial, 846
- degree, 846
- differentiate, 846
- dimension, 846
- discreteLog, 846
- divide, 846
- euclideanSize, 846
- expressIdealMember, 846
- exquo, 846
- extendedEuclidean, 846
- extensionDegree, 846
- factor, 846
- factorsOfCyclicGroupSize, 846
- Frobenius, 846
- gcd, 846
- gcdPolynomial, 846
- generator, 846
- hash, 846
- index, 846
- inGroundField?, 846
- init, 846
- inv, 846
- latex, 846
- lcm, 846
- linearAssociatedExp, 846
- linearAssociatedLog, 846
- linearAssociatedOrder, 846
- lookup, 846
- minimalPolynomial, 846
- multiEuclidean, 846
- nextItem, 846
- norm, 846
- normal?, 846
- normalElement, 846
- one?, 846
- order, 846
- prime?, 846
- primeFrobenius, 846
- primitive?, 846
- primitiveElement, 846
- principalIdeal, 846
- random, 846
- recip, 846
- representationType, 846
- represents, 846
- retract, 846
- retractIfCan, 846
- sample, 846
- size, 846
- sizeLess?, 846
- squareFree, 846
- squareFreePart, 846
- subtractIfCan, 846
- tableForDiscreteLogarithm, 846
- trace, 846
- transcendenceDegree, 846
- transcendent?, 846
- unit?, 846
- unitCanonical, 846
- unitNormal, 846
- zero?, 846
- FGROUP, 1010
 - ?**?, 1010
 - ?*?, 1010
 - ?/?, 1010
 - ?=?, 1010
 - ?^?, 1010
 - ?~=?, 1010
 - 1, 1010
 - coerce, 1010
 - commutator, 1010
 - conjugate, 1010
 - factors, 1010
 - hash, 1010
 - inv, 1010
 - latex, 1010
 - mapExpon, 1010
 - mapGen, 1010
 - nthExpon, 1010
 - nthFactor, 1010
 - one?, 1010
 - recip, 1010

- retract, [1010](#)
 - retractIfCan, [1010](#)
 - sample, [1010](#)
 - size, [1010](#)
- figureUnits
 - GRIMAGE, [1094](#)
- FILE, [802](#)
 - ?=?, [802](#)
 - ?~=?, [802](#)
 - close, [802](#)
 - coerce, [802](#)
 - flush, [802](#)
 - hash, [802](#)
 - iomode, [802](#)
 - latex, [802](#)
 - name, [802](#)
 - open, [802](#)
 - read, [802](#)
 - readIfCan, [802](#)
 - reopen, [802](#)
 - write, [802](#)
- File, [802](#)
- FileName, [810](#)
- filename
 - FNAME, [810](#)
- filterUntil
 - ITUPLE, [1262](#)
 - STREAM, [2582](#)
- filterUpTo
 - NSDPS, [1703](#)
- filterWhile
 - ITUPLE, [1262](#)
 - STREAM, [2582](#)
- find
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - BITS, [315](#)
 - CCLASS, [382](#)
 - CDFVEC, [432](#)
 - DFVEC, [605](#)
 - DLIST, [460](#)
 - EQTBL, [699](#)
 - FARRAY, [886](#)
 - GPOLSET, [1074](#)
 - GSTBL, [1079](#)
 - GTSET, [1083](#)
 - HASHTBL, [1120](#)
 - IARRAY1, [1244](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - ILIST, [1231](#)
 - INTABL, [1336](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - KAFILE, [1414](#)
 - LIB, [1429](#)
 - LIST, [1504](#)
 - LMDICT, [1515](#)
 - MSET, [1670](#)
 - NSDPS, [1703](#)
 - POINT, [2060](#)
 - PRIMARR, [2109](#)
 - REGSET, [2286](#)
 - RESULT, [2301](#)
 - RGCHAIN, [2255](#)
 - ROUTINE, [2331](#)
 - SET, [2372](#)
 - SREGSET, [2535](#)
 - STBL, [2450](#)
 - STREAM, [2582](#)
 - STRING, [2607](#)
 - STRTBL, [2610](#)
 - TABLE, [2664](#)
 - U16VEC, [2924](#)
 - U32VEC, [2931](#)
 - U8VEC, [2917](#)
 - VECTOR, [2942](#)
 - WUTSET, [2959](#)
- findCoef
 - NSDPS, [1703](#)
- findCycle
 - STREAM, [2582](#)
- findTerm
 - NSDPS, [1703](#)
- finite?
 - CARD, [332](#)
 - ONECOMP, [1777](#)
 - ORDCOMP, [1810](#)
- finiteBasis
 - FDIV, [813](#)
- FiniteDivisor, [813](#)
- FiniteField, [819](#)

- FiniteFieldCyclicGroup, [824](#)
- FiniteFieldCyclicGroupExtension, [829](#)
- FiniteFieldCyclicGroupExtensionByPolynomial, [834](#)
- FiniteFieldExtension, [845](#)
- FiniteFieldExtensionByPolynomial, [850](#)
- FiniteFieldNormalBasis, [859](#)
- FiniteFieldNormalBasisExtension, [865](#)
- FiniteFieldNormalBasisExtensionByPolynomial, [871](#)
- fintegrate
 - SMTS, [2441](#)
 - TS, [2671](#)
- first
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - BITS, [315](#)
 - CDFVEC, [432](#)
 - DFVEC, [605](#)
 - DIRPROD, [547](#)
 - DLIST, [460](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - EQTBL, [699](#)
 - FARRAY, [886](#)
 - GSTBL, [1079](#)
 - GTSET, [1083](#)
 - HASHTBL, [1120](#)
 - HDP, [1173](#)
 - IARRAY1, [1244](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - ILIST, [1231](#)
 - INTABL, [1336](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - KAFILE, [1414](#)
 - LIB, [1429](#)
 - LIST, [1504](#)
 - MAGMA, [1565](#)
 - NSDPS, [1703](#)
 - ODP, [1816](#)
 - OFMONOID, [1829](#)
 - PBWLb, [2054](#)
 - POINT, [2060](#)
 - PRIMARR, [2109](#)
 - REGSET, [2286](#)
 - RESULT, [2301](#)
 - RGCHAIN, [2255](#)
 - ROUTINE, [2331](#)
 - SHDP, [2509](#)
 - SREGSET, [2535](#)
 - STBL, [2450](#)
 - STREAM, [2582](#)
 - STRING, [2607](#)
 - STRTBL, [2610](#)
 - TABLE, [2664](#)
 - U16VEC, [2924](#)
 - U32VEC, [2931](#)
 - U8VEC, [2917](#)
 - VECTOR, [2942](#)
 - WUTSET, [2959](#)
- firstDenom
 - PFR, [1914](#)
- firstNumer
 - PFR, [1914](#)
- fixedPoints
 - PERM, [1949](#)
- flagFactor
 - FR, [786](#)
- flatten
 - INFORM, [1343](#)
- flexible?
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- FlexibleArray, [885](#)
- flexibleArray
 - FARRAY, [886](#)
 - IFARRAY, [1222](#)
- FLOAT, [908](#)
 - ?, [908](#)
 - ?<?, [908](#)
 - ?<=?, [908](#)
 - ?>?, [908](#)
 - ?>=?, [908](#)
 - ?**?, [908](#)
 - ?*?, [908](#)
 - ?+?, [908](#)
 - ?-?, [908](#)

- [?/?](#), 908
- [?=?](#), 908
- [?^?](#), 908
- [?~=?](#), 908
- [?quo?](#), 908
- [?rem?](#), 908
- 0, 908
- 1, 908
- abs, 908
- acos, 908
- acosh, 908
- acot, 908
- acoth, 908
- acsc, 908
- acsch, 908
- asec, 908
- asech, 908
- asin, 908
- asinh, 908
- associates?, 908
- atan, 908
- atanh, 908
- base, 908
- bits, 908
- ceiling, 908
- characteristic, 908
- coerce, 908
- convert, 908
- cos, 908
- cosh, 908
- cot, 908
- coth, 908
- csc, 908
- csch, 908
- D, 908
- decreasePrecision, 908
- differentiate, 908
- digits, 908
- divide, 908
- euclideanSize, 908
- exp, 908
- exp1, 908
- exponent, 908
- expressIdealMember, 908
- exquo, 908
- extendedEuclidean, 908
- factor, 908
- float, 908
- floor, 908
- fractionPart, 908
- gcd, 908
- gcdPolynomial, 908
- hash, 908
- increasePrecision, 908
- inv, 908
- latex, 908
- lcm, 908
- log, 908
- log10, 908
- log2, 908
- mantissa, 908
- max, 908
- min, 908
- multiEuclidean, 908
- negative?, 908
- norm, 908
- normalize, 908
- nthRoot, 908
- OMwrite, 908
- one?, 908
- order, 908
- outputFixed, 908
- outputFloating, 908
- outputGeneral, 908
- outputSpacing, 908
- patternMatch, 908
- pi, 908
- positive?, 908
- precision, 908
- prime?, 908
- principalIdeal, 908
- rationalApproximation, 908
- recip, 908
- relerror, 908
- retract, 908
- retractIfCan, 908
- round, 908
- sample, 908
- sec, 908
- sech, 908
- shift, 908
- sign, 908

- sin, 908
- sinh, 908
- sizeLess?, 908
- sqrt, 908
- squareFree, 908
- squareFreePart, 908
- subtractIfCan, 908
- tan, 908
- tanh, 908
- truncate, 908
- unit?, 908
- unitCanonical, 908
- unitNormal, 908
- wholePart, 908
- zero?, 908
- Float, 908
- float
 - DFLOAT, 587
 - FLOAT, 908
 - INFORM, 1343
 - MFLOAT, 1548
 - SEX, 2392
 - SEXOF, 2394
- float?
 - INFORM, 1343
 - SEX, 2392
 - SEXOF, 2394
- floor
 - BINARY, 292
 - BPADICRT, 263
 - DECIMAL, 465
 - DFLOAT, 587
 - EXPEXPAN, 712
 - FLOAT, 908
 - FRAC, 986
 - HEXADEC, 1143
 - MFLOAT, 1548
 - PADICRAT, 1886
 - PADICRC, 1891
 - RADIX, 2206
 - SULS, 2457
 - ULS, 2796
 - ULSCONS, 2804
- flush
 - FILE, 802
- FM, 1013
- ?, 1013
- ?*?, 1013
- ?+?, 1013
- ?-?, 1013
- ?=?, 1013
- ?~=?, 1013
- 0, 1013
- coerce, 1013
- hash, 1013
- latex, 1013
- leadingCoefficient, 1013
- leadingSupport, 1013
- map, 1013
- monomial, 1013
- reductum, 1013
- sample, 1013
- subtractIfCan, 1013
- zero?, 1013
- FM1, 1016
- ?, 1017
- ?*?, 1017
- ?+?, 1017
- ?-?, 1017
- ?=?, 1017
- ?~=?, 1017
- 0, 1017
- coefficient, 1017
- coefficients, 1017
- coerce, 1017
- hash, 1017
- latex, 1017
- leadingCoefficient, 1017
- leadingMonomial, 1017
- leadingTerm, 1017
- listOfTerms, 1017
- map, 1017
- monom, 1017
- monomial?, 1017
- monomials, 1017
- numberOfMonomials, 1017
- reductum, 1017
- retract, 1017
- retractIfCan, 1017
- sample, 1017
- subtractIfCan, 1017
- zero?, 1017

- fmeeg
 - MYUP, [1695](#)
 - NSUP, [1729](#)
 - PR, [2092](#)
 - SUP, [2467](#)
 - SYMPOLY, [2654](#)
 - UP, [2828](#)
- FMONOID, [1021](#)
 - ?<?, [1021](#)
 - ?<=?, [1021](#)
 - ?>?, [1021](#)
 - ?>=?, [1021](#)
 - ?**?, [1021](#)
 - ?*?, [1021](#)
 - ?=?, [1021](#)
 - ?^?, [1021](#)
 - ?~=?, [1021](#)
 - 1, [1021](#)
 - coerce, [1021](#)
 - divide, [1021](#)
 - factors, [1021](#)
 - hash, [1021](#)
 - hclf, [1021](#)
 - hcrf, [1021](#)
 - latex, [1021](#)
 - lquo, [1021](#)
 - mapExpon, [1021](#)
 - mapGen, [1021](#)
 - max, [1021](#)
 - min, [1021](#)
 - nthExpon, [1021](#)
 - nthFactor, [1021](#)
 - one?, [1021](#)
 - overlap, [1021](#)
 - recip, [1021](#)
 - retract, [1021](#)
 - retractIfCan, [1021](#)
 - rquo, [1021](#)
 - sample, [1021](#)
 - size, [1021](#)
- FNAME, [810](#)
 - ?=?, [810](#)
 - ?~=?, [810](#)
 - coerce, [810](#)
 - directory, [810](#)
 - exists?, [810](#)
 - extension, [810](#)
 - filename, [810](#)
 - hash, [810](#)
 - latex, [810](#)
 - name, [810](#)
 - new, [810](#)
 - readable?, [810](#)
 - writable?, [810](#)
- FNLA, [1027](#)
 - ~?, [1027](#)
 - ?**?, [1027](#)
 - ?*?, [1027](#)
 - ?+?, [1027](#)
 - ?-?, [1027](#)
 - ?=?, [1027](#)
 - ?~=?, [1027](#)
 - 0, [1027](#)
 - antiCommutator, [1027](#)
 - associator, [1027](#)
 - coerce, [1027](#)
 - commutator, [1027](#)
 - deepExpand, [1027](#)
 - dimension, [1027](#)
 - generator, [1027](#)
 - hash, [1027](#)
 - latex, [1027](#)
 - leftPower, [1027](#)
 - plenaryPower, [1027](#)
 - rightPower, [1027](#)
 - sample, [1027](#)
 - shallowExpand, [1027](#)
 - subtractIfCan, [1027](#)
 - zero?, [1027](#)
- forLoop
 - FC, [931](#)
- FORMULA, [2346](#)
 - ?=?, [2346](#)
 - ?~=?, [2346](#)
 - coerce, [2346](#)
 - convert, [2346](#)
 - display, [2346](#)
 - epilogue, [2346](#)
 - formula, [2346](#)
 - hash, [2346](#)
 - latex, [2346](#)
 - new, [2346](#)

- prologue, [2346](#)
 - setEpilogue, [2346](#)
 - setFormula, [2346](#)
 - setPrologue, [2346](#)
 - formula
 - FORMULA, [2346](#)
 - FORTRAN, [955](#)
 - coerce, [956](#)
 - outputAsFortran, [956](#)
 - fortran
 - SFORT, [2405](#)
 - fortranCarriageReturn
 - FTEM, [967](#)
 - fortranCharacter
 - FT, [971](#)
 - FortranCode, [931](#)
 - fortranComplex
 - FT, [971](#)
 - fortranDouble
 - FT, [971](#)
 - fortranDoubleComplex
 - FT, [971](#)
 - FortranExpression, [946](#)
 - fortranInteger
 - FT, [971](#)
 - fortranLiteral
 - FTEM, [967](#)
 - fortranLiteralLine
 - FTEM, [967](#)
 - fortranLogical
 - FT, [971](#)
 - FortranProgram, [955](#)
 - fortranReal
 - FT, [971](#)
 - FortranScalarType, [962](#)
 - FortranTemplate, [967](#)
 - FortranType, [971](#)
 - fortranTypeOf
 - SYMTAB, [2648](#)
 - foundPlaces
 - PLACES, [2019](#)
 - PLACESPS, [2021](#)
 - FourierComponent, [975](#)
 - FourierSeries, [978](#)
 - FPARFRAC, [1040](#)
 - ?+?, [1040](#)
 - ?=?, [1040](#)
 - ?~=?, [1040](#)
 - coerce, [1040](#)
 - construct, [1040](#)
 - convert, [1040](#)
 - D, [1040](#)
 - differentiate, [1040](#)
 - fracPart, [1040](#)
 - fullPartialFraction, [1040](#)
 - hash, [1040](#)
 - latex, [1040](#)
 - polyPart, [1040](#)
- FR, [786](#)
- ?, [786](#)
 - ?**?, [786](#)
 - ?*?, [786](#)
 - ?+?, [786](#)
 - ?-?, [786](#)
 - ??.?, [786](#)
 - ?=?, [786](#)
 - ?^?, [786](#)
 - ?~=?, [786](#)
 - 0, [786](#)
 - 1, [786](#)
 - associates?, [786](#)
 - characteristic, [786](#)
 - coerce, [786](#)
 - convert, [786](#)
 - D, [786](#)
 - differentiate, [786](#)
 - eval, [786](#)
 - expand, [786](#)
 - exponent, [786](#)
 - exquo, [786](#)
 - factor, [786](#)
 - factorList, [786](#)
 - factors, [786](#)
 - flagFactor, [786](#)
 - gcd, [786](#)
 - gcdPolynomial, [786](#)
 - hash, [786](#)
 - irreducibleFactor, [786](#)
 - latex, [786](#)
 - lcm, [786](#)
 - makeFR, [786](#)
 - map, [786](#)

- nilFactor, 786
- nthExponent, 786
- nthFactor, 786
- nthFlag, 786
- numberOfFactors, 786
- one?, 786
- prime?, 786
- primeFactor, 786
- rational, 786
- rational?, 786
- rationalIfCan, 786
- recip, 786
- retract, 786
- retractIfCan, 786
- sample, 786
- sqrFactor, 786
- squareFree, 786
- squareFreePart, 786
- subtractIfCan, 786
- unit, 786
- unit?, 786
- unitCanonical, 786
- unitNormal, 786
- unitNormalize, 786
- zero?, 786
- FRAC, 985
- , 986
- ?<?, 986
- ?<=?, 986
- ?>?, 986
- ?>=?, 986
- ?**?, 986
- ?*?, 986
- ?+?, 986
- ?-?, 986
- ?., 986
- ?/?, 986
- ?=?, 986
- ?^?, 986
- ?~=?, 986
- ?quo?, 986
- ?rem?, 986
- 0, 986
- 1, 986
- abs, 986
- associates?, 986
- ceiling, 986
- characteristic, 986
- charthRoot, 986
- coerce, 986
- conditionP, 986
- convert, 986
- D, 986
- denom, 986
- denominator, 986
- differentiate, 986
- divide, 986
- euclideanSize, 986
- eval, 986
- expressIdealMember, 986
- exquo, 986
- extendedEuclidean, 986
- factor, 986
- factorPolynomial, 986
- factorSquareFreePolynomial, 986
- floor, 986
- fractionPart, 986
- gcd, 986
- gcdPolynomial, 986
- hash, 986
- init, 986
- inv, 986
- latex, 986
- lcm, 986
- map, 986
- max, 986
- min, 986
- multiEuclidean, 986
- negative?, 986
- nextItem, 986
- numer, 986
- numerator, 986
- OMwrite, 986
- one?, 986
- patternMatch, 986
- positive?, 986
- prime?, 986
- principalIdeal, 986
- random, 986
- recip, 986
- reducedSystem, 986
- retract, 986

- retractIfCan, 986
 - sample, 986
 - sign, 986
 - sizeLess?, 986
 - solveLinearPolynomialEquation, 986
 - squareFree, 986
 - squareFreePart, 986
 - squareFreePolynomial, 986
 - subtractIfCan, 986
 - unit?, 986
 - unitCanonical, 986
 - unitNormal, 986
 - wholePart, 986
 - zero?, 986
- fracPart
 - FPARFRAC, 1040
- Fraction, 985
- FractionalIdeal, 994
- fractionPart
 - BINARY, 292
 - BPADICRT, 263
 - DECIMAL, 465
 - DFLOAT, 587
 - EXPEXPAN, 712
 - FLOAT, 908
 - FRAC, 986
 - HEXADEC, 1143
 - MFLOAT, 1548
 - PADICRAT, 1886
 - PADICRC, 1891
 - RADIX, 2206
 - SULS, 2457
 - ULS, 2796
 - ULSCONS, 2804
- fractRadix
 - RADIX, 2206
- fractRagits
 - RADIX, 2206
- FramedModule, 1000
- FreeAbelianGroup, 1004
- FreeAbelianMonoid, 1007
- FreeGroup, 1010
- FreeModule, 1013
- FreeModule1, 1016
- FreeMonoid, 1021
- FreeNilpotentLie, 1027
- freeOf?
 - AN, 35
 - EXPR, 724
 - FEXPR, 947
 - IAN, 1276
 - MYEXPR, 1688
 - SD, 2572
- FRIDEAL, 994
 - ?**?, 995
 - ?*?, 995
 - ?/? , 995
 - ?=?, 995
 - ?^?, 995
 - ?~=?, 995
 - 1, 995
 - basis, 995
 - coerce, 995
 - commutator, 995
 - conjugate, 995
 - denom, 995
 - hash, 995
 - ideal, 995
 - inv, 995
 - latex, 995
 - minimize, 995
 - norm, 995
 - numer, 995
 - one?, 995
 - randomLC, 995
 - recip, 995
 - sample, 995
- FRMOD, 1000
 - ?**?, 1000
 - ?*?, 1000
 - ?=?, 1000
 - ?^?, 1000
 - ?~=?, 1000
 - 1, 1000
 - basis, 1000
 - coerce, 1000
 - hash, 1000
 - latex, 1000
 - module, 1000
 - norm, 1000
 - one?, 1000
 - recip, 1000

- sample, 1000
- Frobenius
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866
 - FFP, 851
 - FFX, 846
 - IFF, 1284
 - IPF, 1303
 - PACOFF, 2136
 - PACRAT, 2146
 - PF, 2105
- frobenius
 - MODMON, 1632
- front
 - DEQUEUE, 512
 - QUEUE, 2184
- first
 - NSDPS, 1703
 - STREAM, 2582
- FSERIES, 978
 - , 978
 - ?**?, 978
 - ?*?, 978
 - ?+?, 978
 - ?-?, 978
 - ?=?, 978
 - ?^?, 978
 - ?~=?, 978
 - 0, 978
 - 1, 978
 - characteristic, 978
 - coerce, 978
 - hash, 978
 - latex, 978
 - makeCos, 978
 - makeSin, 978
 - one?, 978
 - recip, 978
 - sample, 978
 - subtractIfCan, 978
 - zero?, 978
- FST, 962
 - ?=?, 962
 - character?, 962
 - coerce, 962
 - complex?, 962
 - double?, 962
 - doubleComplex?, 962
 - integer?, 962
 - logical?, 962
 - real?, 962
- FT, 971
 - ?=?, 971
 - ?~=?, 971
 - coerce, 971
 - construct, 971
 - dimensionsOf, 971
 - external?, 971
 - fortranCharacter, 971
 - fortranComplex, 971
 - fortranDouble, 971
 - fortranDoubleComplex, 971
 - fortranInteger, 971
 - fortranLogical, 971
 - fortranReal, 971
 - hash, 971
 - latex, 971
 - scalarTypeOf, 971
- fTable
 - INTFTBL, 1371
- FTEM, 967
 - ?=?, 967
 - ?~=?, 967
 - close, 967
 - coerce, 967
 - fortranCarriageReturn, 967
 - fortranLiteral, 967
 - fortranLiteralLine, 967
 - hash, 967
 - iomode, 967
 - latex, 967
 - name, 967
 - open, 967
 - processTemplate, 967
 - read, 967
 - reopen, 967
 - write, 967

- fullDisplay
 - DBASE, [454](#)
 - ICARD, [1193](#)
- fullOut
 - DSTREE, [535](#)
 - IC, [1191](#)
 - INFCLSPS, [1271](#)
 - INFCLSPT, [1265](#)
- fullOutput
 - DSTREE, [535](#)
 - IC, [1191](#)
 - INFCLSPS, [1271](#)
 - INFCLSPT, [1265](#)
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
- fullPartialFraction
 - FPARFRAC, [1040](#)
- FullPartialFractionExpansion, [1040](#)
- FUNCTION, [1045](#)
 - ?=?, [1045](#)
 - ?~=?, [1045](#)
 - coerce, [1045](#)
 - hash, [1045](#)
 - latex, [1045](#)
 - name, [1045](#)
- function
 - INFORM, [1343](#)
- FunctionCalled, [1045](#)
- functionName
 - GOPT, [1104](#)
 - GOPT0, [1110](#)
- functionNames
 - GOPT, [1104](#)
- Gamma
 - DFLOAT, [587](#)
 - EXPR, [724](#)
- gcd
 - ALGFF, [28](#)
 - AN, [35](#)
 - BINARY, [292](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EMR, [702](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FLOAT, [908](#)
 - FR, [786](#)
 - FRAC, [986](#)
 - GDMP, [1052](#)
 - GSERIES, [1090](#)
 - HACKPI, [1978](#)
 - HDMP, [1180](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - IFF, [1284](#)
 - INT, [1362](#)
 - INTRVL, [1384](#)
 - IPADIC, [1294](#)
 - IPF, [1303](#)
 - LAUPOL, [1422](#)
 - MCMPLEX, [1543](#)
 - MFLOAT, [1548](#)
 - MINT, [1557](#)
 - MODFIELD, [1638](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYEXPR, [1688](#)
 - MYUP, [1695](#)
 - NNI, [1739](#)
 - NSDPS, [1703](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - ODR, [1858](#)
 - PACOFF, [2136](#)

- PACRAT, [2146](#)
- PADIC, [1881](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PF, [2105](#)
- PFR, [1914](#)
- PI, [2100](#)
- POLY, [2078](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- ROMAN, [2326](#)
- SAE, [2400](#)
- SDPOL, [2386](#)
- SINT, [2412](#)
- SMP, [2423](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- gcdPolynomial
 - ALGFF, [28](#)
 - AN, [35](#)
 - BINARY, [292](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EMR, [702](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FLOAT, [908](#)
 - FR, [786](#)
 - FRAC, [986](#)
 - GDMP, [1052](#)
 - GSERIES, [1090](#)
 - HACKPI, [1978](#)
 - HDMP, [1180](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - IFF, [1284](#)
 - INT, [1362](#)
 - INTRVL, [1384](#)
 - IPADIC, [1294](#)
 - IPF, [1303](#)
 - LAUPOL, [1422](#)
 - MCMPLEX, [1543](#)
 - MFLOAT, [1548](#)
 - MINT, [1557](#)
 - MODFIELD, [1638](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYEXPR, [1688](#)
 - MYUP, [1695](#)
 - NSDPS, [1703](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - ODR, [1858](#)
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
 - PADIC, [1881](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - PF, [2105](#)
 - PFR, [1914](#)
 - POLY, [2078](#)
 - RADFF, [2194](#)
 - RADIX, [2206](#)
 - RECLOS, [2237](#)
 - ROMAN, [2326](#)
 - SAE, [2400](#)
 - SDPOL, [2386](#)

- SINT, [2412](#)
- SMP, [2423](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXP, [2481](#)
- SUPXS, [2487](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- GCNAALG, [1064](#)
- ?, [1065](#)
- ?**?, [1065](#)
- ?*?, [1065](#)
- ?+?, [1065](#)
- ?-?, [1065](#)
- ?..?, [1065](#)
- ?=?, [1065](#)
- ?~=?, [1065](#)
- 0, [1065](#)
- alternative?, [1065](#)
- antiAssociative?, [1065](#)
- antiCommutative?, [1065](#)
- antiCommutator, [1065](#)
- apply, [1065](#)
- associative?, [1065](#)
- associator, [1065](#)
- associatorDependence, [1065](#)
- basis, [1065](#)
- coerce, [1065](#)
- commutative?, [1065](#)
- commutator, [1065](#)
- conditionsForIdempotents, [1065](#)
- convert, [1065](#)
- coordinates, [1065](#)
- flexible?, [1065](#)
- generic, [1065](#)
- genericLeftDiscriminant, [1065](#)
- genericLeftMinimalPolynomial, [1065](#)
- genericLeftNorm, [1065](#)
- genericLeftTrace, [1065](#)
- genericLeftTraceForm, [1065](#)
- genericRightDiscriminant, [1065](#)
- genericRightMinimalPolynomial, [1065](#)
- genericRightNorm, [1065](#)
- genericRightTrace, [1065](#)
- genericRightTraceForm, [1065](#)
- hash, [1065](#)
- jacobiIdentity?, [1065](#)
- jordanAdmissible?, [1065](#)
- jordanAlgebra?, [1065](#)
- latex, [1065](#)
- leftAlternative?, [1065](#)
- leftCharacteristicPolynomial, [1065](#)
- leftDiscriminant, [1065](#)
- leftMinimalPolynomial, [1065](#)
- leftNorm, [1065](#)
- leftPower, [1065](#)
- leftRankPolynomial, [1065](#)
- leftRecip, [1065](#)
- leftRegularRepresentation, [1065](#)
- leftTrace, [1065](#)
- leftTraceMatrix, [1065](#)
- leftUnit, [1065](#)
- leftUnits, [1065](#)
- lieAdmissible?, [1065](#)
- lieAlgebra?, [1065](#)
- noncommutativeJordanAlgebra?, [1065](#)
- plenaryPower, [1065](#)
- powerAssociative?, [1065](#)
- rank, [1065](#)
- recip, [1065](#)
- represents, [1065](#)
- rightAlternative?, [1065](#)
- rightCharacteristicPolynomial, [1065](#)
- rightDiscriminant, [1065](#)
- rightMinimalPolynomial, [1065](#)
- rightNorm, [1065](#)
- rightPower, [1065](#)
- rightRankPolynomial, [1065](#)
- rightRecip, [1065](#)
- rightRegularRepresentation, [1065](#)
- rightTrace, [1065](#)
- rightTraceMatrix, [1065](#)
- rightUnit, [1065](#)
- rightUnits, [1065](#)
- sample, [1065](#)
- someBasis, [1065](#)
- structuralConstants, [1065](#)
- subtractIfCan, [1065](#)
- unit, [1065](#)

- zero?, 1065
- GDMP, 1052
- ?, 1052
- ?<?, 1052
- ?<=?, 1052
- ?>?, 1052
- ?>=?, 1052
- ?**?, 1052
- ?*?, 1052
- ?+?, 1052
- ?-?, 1052
- ?=?, 1052
- ?^?, 1052
- ?~=?, 1052
- 0, 1052
- 1, 1052
- associates?, 1052
- binomThmExpt, 1052
- characteristic, 1052
- charthRoot, 1052
- coefficient, 1052
- coefficients, 1052
- coerce, 1052
- conditionP, 1052
- content, 1052
- D, 1052
- degree, 1052
- differentiate, 1052
- discriminant, 1052
- eval, 1052
- exquo, 1052
- factor, 1052
- factorPolynomial, 1052
- factorSquareFreePolynomial, 1052
- gcd, 1052
- gcdPolynomial, 1052
- ground, 1052
- ground?, 1052
- hash, 1052
- isExpt, 1052
- isPlus, 1052
- isTimes, 1052
- latex, 1052
- lcm, 1052
- leadingCoefficient, 1052
- leadingMonomial, 1052
- mainVariable, 1052
- map, 1052
- mapExponents, 1052
- max, 1052
- min, 1052
- minimumDegree, 1052
- monicDivide, 1052
- monomial, 1052
- monomial?, 1052
- monomials, 1052
- multivariate, 1052
- numberOfMonomials, 1052
- one?, 1052
- patternMatch, 1052
- pomopo, 1052
- prime?, 1052
- primitiveMonomials, 1052
- primitivePart, 1052
- recip, 1052
- reducedSystem, 1052
- reductum, 1052
- reorder, 1052
- resultant, 1052
- retract, 1052
- retractIfCan, 1052
- sample, 1052
- solveLinearPolynomialEquation, 1052
- squareFree, 1052
- squareFreePart, 1052
- squareFreePolynomial, 1052
- subtractIfCan, 1052
- totalDegree, 1052
- unit?, 1052
- unitCanonical, 1052
- unitNormal, 1052
- univariate, 1052
- variables, 1052
- zero?, 1052
- GE
- SWITCH, 2629
- GeneralDistributedMultivariatePolynomial, 1052
- generalizedContinuumHypothesisAssumed
- CARD, 332
- generalizedContinuumHypothesisAssumed?
- CARD, 332
- generalLambert

- UFPS, [2789](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- GeneralModulePolynomial, [1059](#)
- GeneralPolynomialSet, [1074](#)
- generalPosition
 - IDEAL, [2081](#)
- GeneralSparseTable, [1078](#)
- GeneralTriangularSet, [1082](#)
- GeneralUnivariatePowerSeries, [1089](#)
- generate
 - ITUPLE, [1262](#)
 - STREAM, [2582](#)
- generator
 - ALGFF, [28](#)
 - ANTISYM, [40](#)
 - COMPLEX, [419](#)
 - DERHAM, [530](#)
 - FDIV, [813](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FNLA, [1027](#)
 - HELLFDIV, [1183](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - MCMPLEX, [1543](#)
 - PF, [2105](#)
 - RADFF, [2194](#)
 - SAE, [2400](#)
- generators
 - IDEAL, [2081](#)
 - PERMGRP, [1960](#)
- generic
 - GCNAALG, [1065](#)
- generic?
 - PATTERN, [1928](#)
- genericLeftDiscriminant
 - GCNAALG, [1065](#)
- genericLeftMinimalPolynomial
 - GCNAALG, [1065](#)
- genericLeftNorm
 - GCNAALG, [1065](#)
- genericLeftTrace
 - GCNAALG, [1065](#)
- genericLeftTraceForm
 - GCNAALG, [1065](#)
- GenericNonAssociativeAlgebra, [1064](#)
- genericRightDiscriminant
 - GCNAALG, [1065](#)
- genericRightMinimalPolynomial
 - GCNAALG, [1065](#)
- genericRightNorm
 - GCNAALG, [1065](#)
- genericRightTrace
 - GCNAALG, [1065](#)
- genericRightTraceForm
 - GCNAALG, [1065](#)
- genus
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- getBadValues
 - PATTERN, [1928](#)
- getButtonValue
 - ATTRBUT, [239](#)
- getCode
 - FC, [931](#)
- getCurve
 - TUBE, [2749](#)
- getExplanations
 - ROUTINE, [2331](#)
- getGraph
 - VIEW2d, [2770](#)
- getMatch
 - PATRES, [1940](#)
- getMeasure
 - ROUTINE, [2331](#)
- getMultiplicationMatrix
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
- getMultiplicationTable
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
- getPickedPoints

- VIEW2d, 2770
- getRef
 - ISUPS, 1311
- getSmgl
 - BSD, 286
- getStream
 - ISUPS, 1311
- getZechTable
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
- GMODPOL, 1059
 - −?, 1059
 - ?*?, 1059
 - ?+?, 1059
 - ?−?, 1059
 - ?=?, 1059
 - ?~=?, 1059
 - 0, 1059
 - build, 1059
 - coerce, 1059
 - hash, 1059
 - latex, 1059
 - leadingCoefficient, 1059
 - leadingExponent, 1059
 - leadingIndex, 1059
 - leadingMonomial, 1059
 - monomial, 1059
 - multMonom, 1059
 - reductum, 1059
 - sample, 1059
 - subtractIfCan, 1059
 - unitVector, 1059
 - zero?, 1059
- GOPT, 1104
 - ?=?, 1104
 - ?~=?, 1104
 - allDegrees, 1104
 - check, 1104
 - checkExtraValues, 1104
 - coerce, 1104
 - debug, 1104
 - displayKind, 1104
 - functionName, 1104
 - functionNames, 1104
 - hash, 1104
 - homogeneous, 1104
 - indexName, 1104
 - latex, 1104
 - maxDegree, 1104
 - maxDerivative, 1104
 - maxLevel, 1104
 - maxMixedDegree, 1104
 - maxPower, 1104
 - maxShift, 1104
 - maxSubst, 1104
 - one, 1104
 - option, 1104
 - safety, 1104
 - Somos, 1104
 - variableName, 1104
- GOPT0, 1110
 - ?=?, 1110
 - ?~=?, 1110
 - allDegrees, 1110
 - check, 1110
 - checkOptions, 1110
 - coerce, 1110
 - debug, 1110
 - displayAsGF, 1110
 - functionName, 1110
 - hash, 1110
 - homogeneous, 1110
 - indexName, 1110
 - latex, 1110
 - maxDegree, 1110
 - maxDerivative, 1110
 - maxLevel, 1110
 - maxMixedDegree, 1110
 - maxPower, 1110
 - maxShift, 1110
 - maxSubst, 1110
 - MonteCarlo, 1110
 - one, 1110
 - safety, 1110
 - Somos, 1110
 - variableName, 1110
- goto
 - FC, 931
- GPOLSET, 1074
 - ?=?, 1074
 - ?~=?, 1074

- [#?](#), 1074
- [any?](#), 1074
- [coerce](#), 1074
- [collect](#), 1074
- [collectUnder](#), 1074
- [collectUpper](#), 1074
- [construct](#), 1074
- [convert](#), 1074
- [copy](#), 1074
- [count](#), 1074
- [empty](#), 1074
- [empty?](#), 1074
- [eq?](#), 1074
- [eval](#), 1074
- [every?](#), 1074
- [find](#), 1074
- [hash](#), 1074
- [headRemainder](#), 1074
- [latex](#), 1074
- [less?](#), 1074
- [mainVariable?](#), 1074
- [mainVariables](#), 1074
- [map](#), 1074
- [member?](#), 1074
- [members](#), 1074
- [more?](#), 1074
- [mvar](#), 1074
- [parts](#), 1074
- [reduce](#), 1074
- [remainder](#), 1074
- [remove](#), 1074
- [removeDuplicates](#), 1074
- [retract](#), 1074
- [retractIfCan](#), 1074
- [rewriteIdealWithHeadRemainder](#), 1074
- [rewriteIdealWithRemainder](#), 1074
- [roughBase?](#), 1074
- [roughEqualIdeals?](#), 1074
- [roughSubIdeal?](#), 1074
- [roughUnitIdeal?](#), 1074
- [sample](#), 1074
- [select](#), 1074
- [size?](#), 1074
- [sort](#), 1074
- [triangular?](#), 1074
- [trivialIdeal?](#), 1074
- [variables](#), 1074
- [GraphImage](#), 1094
- [graphImage](#)
 - [GRIMAGE](#), 1094
- [graphs](#)
 - [VIEW2d](#), 2770
- [graphState](#)
 - [VIEW2d](#), 2770
- [graphStates](#)
 - [VIEW2d](#), 2770
- [green](#)
 - [COLOR](#), 407
- [GRIMAGE](#), 1094
 - [=?](#), 1094
 - [?~=?](#), 1094
 - [appendPoint](#), 1094
 - [coerce](#), 1094
 - [component](#), 1094
 - [figureUnits](#), 1094
 - [graphImage](#), 1094
 - [hash](#), 1094
 - [key](#), 1094
 - [latex](#), 1094
 - [makeGraphImage](#), 1094
 - [point](#), 1094
 - [pointLists](#), 1094
 - [putColorInfo](#), 1094
 - [ranges](#), 1094
 - [units](#), 1094
- [groebner](#)
 - [IDEAL](#), 2081
- [groebner?](#)
 - [IDEAL](#), 2081
- [groebnerIdeal](#)
 - [IDEAL](#), 2081
- [ground](#)
 - [DMP](#), 572
 - [DSMP](#), 542
 - [EXPR](#), 724
 - [GDMP](#), 1052
 - [HDMP](#), 1180
 - [MODMON](#), 1632
 - [MPOLY](#), 1682
 - [MYEXPR](#), 1688
 - [MYUP](#), 1695
 - [NSMP](#), 1714

- NSUP, [1729](#)
- ODPOL, [1852](#)
- POLY, [2078](#)
- PR, [2092](#)
- SDPOL, [2386](#)
- SMP, [2423](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SYMPOLY, [2654](#)
- UP, [2828](#)
- UPXSSING, [2852](#)
- ground?
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPR, [724](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYEXPR, [1688](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
 - POLY, [2078](#)
 - PR, [2092](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - SYMPOLY, [2654](#)
 - UP, [2828](#)
 - UPXSSING, [2852](#)
- GSERIES, [1089](#)
- ?, [1090](#)
- ?**?, [1090](#)
- ?*?, [1090](#)
- ?+?, [1090](#)
- ?-?, [1090](#)
- ?.?, [1090](#)
- ?/? , [1090](#)
- ?=?, [1090](#)
- ?^?, [1090](#)
- ?~=?, [1090](#)
- ?quo?, [1090](#)
- ?rem?, [1090](#)
- 0, [1090](#)
- 1, [1090](#)
- acos, [1090](#)
- acosh, [1090](#)
- acot, [1090](#)
- acoth, [1090](#)
- acsc, [1090](#)
- acsch, [1090](#)
- approximate, [1090](#)
- asec, [1090](#)
- asech, [1090](#)
- asin, [1090](#)
- asinh, [1090](#)
- associates?, [1090](#)
- atan, [1090](#)
- atanh, [1090](#)
- center, [1090](#)
- characteristic, [1090](#)
- charthRoot, [1090](#)
- coefficient, [1090](#)
- coerce, [1090](#)
- complete, [1090](#)
- cos, [1090](#)
- cosh, [1090](#)
- cot, [1090](#)
- coth, [1090](#)
- csc, [1090](#)
- csch, [1090](#)
- D, [1090](#)
- degree, [1090](#)
- differentiate, [1090](#)
- divide, [1090](#)
- euclideanSize, [1090](#)
- eval, [1090](#)
- exp, [1090](#)
- expressIdealMember, [1090](#)
- exquo, [1090](#)
- extend, [1090](#)
- extendedEuclidean, [1090](#)
- factor, [1090](#)
- gcd, [1090](#)
- gcdPolynomial, [1090](#)
- hash, [1090](#)
- integrate, [1090](#)

- inv, 1090
- latex, 1090
- lcm, 1090
- leadingCoefficient, 1090
- leadingMonomial, 1090
- log, 1090
- map, 1090
- monomial, 1090
- monomial?, 1090
- multiEuclidean, 1090
- multiplyExponents, 1090
- nthRoot, 1090
- one?, 1090
- order, 1090
- pi, 1090
- pole?, 1090
- prime?, 1090
- principalIdeal, 1090
- recip, 1090
- reductum, 1090
- sample, 1090
- sec, 1090
- sech, 1090
- series, 1090
- sin, 1090
- sinh, 1090
- sizeLess?, 1090
- sqrt, 1090
- squareFree, 1090
- squareFreePart, 1090
- subtractIfCan, 1090
- tan, 1090
- tanh, 1090
- terms, 1090
- truncate, 1090
- unit?, 1090
- unitCanonical, 1090
- unitNormal, 1090
- variable, 1090
- variables, 1090
- zero?, 1090
- GSTBL, 1078
- ?.?, 1079
- ?=, 1079
- ?~=, 1079
- #?, 1079
- any?, 1079
- bag, 1079
- coerce, 1079
- construct, 1079
- convert, 1079
- copy, 1079
- count, 1079
- dictionary, 1079
- elt, 1079
- empty, 1079
- empty?, 1079
- entries, 1079
- entry?, 1079
- eq?, 1079
- eval, 1079
- every?, 1079
- extract, 1079
- fill, 1079
- find, 1079
- first, 1079
- hash, 1079
- index?, 1079
- indices, 1079
- insert, 1079
- inspect, 1079
- key?, 1079
- keys, 1079
- latex, 1079
- less?, 1079
- map, 1079
- maxIndex, 1079
- member?, 1079
- members, 1079
- minIndex, 1079
- more?, 1079
- parts, 1079
- qelt, 1079
- qsetelt, 1079
- reduce, 1079
- remove, 1079
- removeDuplicates, 1079
- sample, 1079
- search, 1079
- select, 1079
- setelt, 1079
- size?, 1079

- swap, 1079
- table, 1079
- GT
 - SWITCH, 2629
- GTSET, 1082
 - ?=?, 1083
 - ?~=?, 1083
 - #?, 1083
 - algebraic?, 1083
 - algebraicVariables, 1083
 - any?, 1083
 - autoReduced?, 1083
 - basicSet, 1083
 - coerce, 1083
 - coHeight, 1083
 - collect, 1083
 - collectQuasiMonic, 1083
 - collectUnder, 1083
 - collectUpper, 1083
 - construct, 1083
 - convert, 1083
 - copy, 1083
 - count, 1083
 - degree, 1083
 - empty, 1083
 - empty?, 1083
 - eq?, 1083
 - eval, 1083
 - every?, 1083
 - extend, 1083
 - extendIfCan, 1083
 - find, 1083
 - first, 1083
 - hash, 1083
 - headReduce, 1083
 - headReduced?, 1083
 - headRemainder, 1083
 - infRittWu?, 1083
 - initiallyReduce, 1083
 - initiallyReduced?, 1083
 - initials, 1083
 - last, 1083
 - latex, 1083
 - less?, 1083
 - mainVariable?, 1083
 - mainVariables, 1083
 - map, 1083
 - member?, 1083
 - members, 1083
 - more?, 1083
 - mvar, 1083
 - normalized?, 1083
 - parts, 1083
 - quasiComponent, 1083
 - reduce, 1083
 - reduceByQuasiMonic, 1083
 - reduced?, 1083
 - remainder, 1083
 - remove, 1083
 - removeDuplicates, 1083
 - removeZero, 1083
 - rest, 1083
 - retract, 1083
 - retractIfCan, 1083
 - rewriteIdealWithHeadRemainder, 1083
 - rewriteIdealWithRemainder, 1083
 - rewriteSetWithReduction, 1083
 - roughBase?, 1083
 - roughEqualIdeals?, 1083
 - roughSubIdeal?, 1083
 - roughUnitIdeal?, 1083
 - sample, 1083
 - select, 1083
 - size?, 1083
 - sort, 1083
 - stronglyReduce, 1083
 - stronglyReduced?, 1083
 - triangular?, 1083
 - trivialIdeal?, 1083
 - variables, 1083
 - zeroSetSplit, 1083
 - zeroSetSplitIntoTriangularSystems, 1083
- GuessOption, 1104
- GuessOptionFunctions0, 1110
- HACKPI, 1977
 - , 1978
 - ?**?, 1978
 - ?*?, 1978
 - ?+?, 1978
 - ?-?, 1978
 - ?/?, 1978

- [?=?, 1978](#)
- [?^?, 1978](#)
- [?~=?, 1978](#)
- [?quo?, 1978](#)
- [?rem?, 1978](#)
- [0, 1978](#)
- [1, 1978](#)
- [associates?, 1978](#)
- [characteristic, 1978](#)
- [coerce, 1978](#)
- [convert, 1978](#)
- [divide, 1978](#)
- [euclideanSize, 1978](#)
- [expressIdealMember, 1978](#)
- [exquo, 1978](#)
- [extendedEuclidean, 1978](#)
- [factor, 1978](#)
- [gcd, 1978](#)
- [gcdPolynomial, 1978](#)
- [hash, 1978](#)
- [inv, 1978](#)
- [latex, 1978](#)
- [lcm, 1978](#)
- [multiEuclidean, 1978](#)
- [one?, 1978](#)
- [pi, 1978](#)
- [prime?, 1978](#)
- [principalIdeal, 1978](#)
- [recip, 1978](#)
- [retract, 1978](#)
- [retractIfCan, 1978](#)
- [sample, 1978](#)
- [sizeLess?, 1978](#)
- [squareFree, 1978](#)
- [squareFreePart, 1978](#)
- [subtractIfCan, 1978](#)
- [unit?, 1978](#)
- [unitCanonical, 1978](#)
- [unitNormal, 1978](#)
- [zero?, 1978](#)
- [halfExtendedResultant1](#)
 - [NSUP, 1729](#)
- [halfExtendedResultant2](#)
 - [NSUP, 1729](#)
- [halfExtendedSubResultantGcd1](#)
 - [NSMP, 1714](#)
- [NSUP, 1729](#)
- [halfExtendedSubResultantGcd2](#)
 - [NSMP, 1714](#)
 - [NSUP, 1729](#)
- [has?](#)
 - [BOP, 274](#)
- [hash](#)
 - [AFFPLPS, 7](#)
 - [AFFSP, 9](#)
 - [ALGFF, 28](#)
 - [ALGSC, 15](#)
 - [ALIST, 235](#)
 - [AN, 35](#)
 - [ANON, 38](#)
 - [ANTISYM, 40](#)
 - [ANY, 50](#)
 - [ARRAY1, 1774](#)
 - [ARRAY2, 2763](#)
 - [ASTACK, 65](#)
 - [ATTRBUT, 239](#)
 - [AUTOMOR, 245](#)
 - [BBTREE, 253](#)
 - [BFUNCT, 265](#)
 - [BINARY, 292](#)
 - [BINFILE, 295](#)
 - [BITS, 315](#)
 - [BLHN, 317](#)
 - [BLQT, 319](#)
 - [BOOLEAN, 322](#)
 - [BOP, 274](#)
 - [BPADIC, 258](#)
 - [BPADICRT, 263](#)
 - [BSD, 286](#)
 - [BSTREE, 302](#)
 - [BTOURN, 306](#)
 - [BTREE, 310](#)
 - [CARD, 332](#)
 - [CARTEN, 356](#)
 - [CCLASS, 382](#)
 - [CDFMAT, 426](#)
 - [CDFVEC, 432](#)
 - [CHAR, 373](#)
 - [CLIF, 402](#)
 - [COLOR, 407](#)
 - [COMM, 411](#)
 - [COMPLEX, 419](#)

- COMPPROP, 2624
CONTFRAC, 445
D01AJFA, 614
D01AKFA, 617
D01ALFA, 620
D01AMFA, 623
D01APFA, 629, 633
D01ASFA, 636
D01FCFA, 639
D01GBFA, 643
D01TRNS, 646
D02BBFA, 651
D02BHFA, 654
D02CJFA, 658
D02EJFA, 661
D03EEFA, 665
D03FAFA, 668
D10ANFA, 626
DBASE, 454
DECIMAL, 465
DEQUEUE, 512
DERHAM, 530
DFLOAT, 587
DFMAT, 599
DFVEC, 605
DHMATRIX, 491
DIRPROD, 547
DIRRING, 564
DIV, 575
DLIST, 460
DMP, 572
DPMM, 553
DPMO, 557
DROPT, 608
DSMP, 542
DSTREE, 535
E04DGFA, 747
E04FDFA, 750
E04GCFA, 754
E04JAFA, 758
E04MBFA, 762
E04NAFA, 765
E04UCFA, 769
EAB, 743
EMR, 702
EQ, 691
EQTBL, 699
EXIT, 707
EXPEXPAN, 712
EXPR, 724
EXPUPXS, 740
FAGROUP, 1004
FAMONOID, 1007
FARRAY, 886
FC, 931
FCOMP, 975
FDIV, 813
FEXPR, 947
FF, 820
FFCG, 825
FFCGP, 835
FFCGX, 830
FFNB, 860
FFNBP, 872
FFNBX, 866
FFP, 851
FFX, 846
FGROUP, 1010
FILE, 802
FLOAT, 908
FM, 1013
FM1, 1017
FMONOID, 1021
FNAME, 810
FNLA, 1027
FORMULA, 2346
FPARFRAC, 1040
FR, 786
FRAC, 986
FRIDEAL, 995
FRMOD, 1000
FSERIES, 978
FT, 971
FTEM, 967
FUNCTION, 1045
GCNAALG, 1065
GDMP, 1052
GMODPOL, 1059
GOPT, 1104
GOPT0, 1110
GPOLSET, 1074
GRIMAGE, 1094

- GSERIES, 1090
GSTBL, 1079
GTSET, 1083
HACKPI, 1978
HASHTBL, 1120
HDMP, 1180
HDP, 1173
HEAP, 1134
HELLFDIV, 1183
HEXADEC, 1143
HTMLFORM, 1152
IAN, 1276
IARRAY1, 1244
IARRAY2, 1256
IBITS, 1199
IC, 1191
ICARD, 1193
IDEAL, 2081
IDPAG, 1202
IDPAM, 1206
IDPO, 1209
IDPOAM, 1212
IDPOAMS, 1215
IFAMON, 1287
IFARRAY, 1222
IFF, 1284
IIARRAY2, 1290
ILIST, 1231
IMATRIX, 1239
INDE, 1218
INFCLSPS, 1271
INFCLSPT, 1265
INFORM, 1343
INT, 1362
INTABL, 1336
INTRVL, 1384
IPADIC, 1294
IPF, 1303
IR, 1375
ISTRING, 1249
ISUPS, 1311
ITAYLOR, 1338
IVECTOR, 1260
JORDAN, 223
KAFILE, 1414
KERNEL, 1404
LA, 1520
LAUPOL, 1422
LEXP, 1435
LIB, 1429
LIE, 229
LIST, 1504
LMDICT, 1515
LMOPS, 1509
LO, 1523
LODO, 1469
LODO1, 1479
LODO2, 1491
LPOLY, 1447
LSQM, 1456
LWORD, 1532
M3D, 2703
MAGMA, 1565
MATRIX, 1623
MCMPLX, 1543
MFLOAT, 1548
MINT, 1557
MKCHSET, 1570
MMLFORM, 1603
MODFIELD, 1638
MODMON, 1632
MODMONOM, 1644
MODOP, 1647, 1804
MODRING, 1641
MOEBIUS, 1654
MPOLY, 1682
MRING, 1659
MSET, 1670
MYEXPR, 1688
MYUP, 1695
NIPROB, 1746
NNI, 1739
NONE, 1737
NOTTING, 1744
NSDPS, 1703
NSMP, 1714
NSUP, 1729
OCT, 1765
ODEPROB, 1750
ODP, 1816
ODPOL, 1852
ODR, 1858

ODVAR, 1855
OFMONOID, 1829
OMENC, 1789
OMERR, 1792
OMERRK, 1794
OMLO, 1806
ONECOMP, 1777
OPTPROB, 1753
ORDCOMP, 1810
ORESUP, 2492
OREUP, 2872
OSI, 1864
OUTFORM, 1868
OVAR, 1836
OWP, 1861
PACOFF, 2136
PACRAT, 2146
PADIC, 1881
PADICRAT, 1886
PADICRC, 1891
PALETTE, 1896
PATLRES, 1937
PATRES, 1940
PATTERN, 1928
PBWLB, 2054
PDEPROB, 1756
PENDTREE, 1945
PERM, 1949
PERMGRP, 1960
PF, 2105
PFR, 1914
PI, 2100
PLACES, 2019
PLACESPS, 2021
POINT, 2060
POLY, 2078
PR, 2092
PRIMARR, 2109
PRODUCT, 2113
PROJPL, 2117
PROJPLPS, 2119
PROJSP, 2122
PRITION, 1923
QALGSET, 2157
QFORM, 2154
QUAT, 2167
QUEUE, 2184
RADFF, 2194
RADIX, 2206
RECLOS, 2237
REF, 2249
REGSET, 2286
RESRING, 2296
RESULT, 2301
RGCHAIN, 2255
RMATRIX, 2246
ROIRC, 2310
ROMAN, 2326
ROUTINE, 2331
RULE, 2305
RULECOLD, 2340
RULESET, 2342
SAE, 2400
SAOS, 2418
SD, 2572
SDPOL, 2386
SDVAR, 2389
SEG, 2359
SEGBIND, 2364
SET, 2372
SETMN, 2377
SEX, 2392
SEXOF, 2394
SHDP, 2509
SINT, 2412
SMP, 2423
SMTS, 2441
SPACE3, 2731
SPLNODE, 2512
SPLTREE, 2518
SQMATRIX, 2548
SREGSET, 2535
STACK, 2563
STBL, 2450
STREAM, 2582
STRING, 2607
STRTBL, 2610
SUBSPACE, 2614
SUCH, 2627
SULS, 2457
SUP, 2467
SUPEXPR, 2481

- SUPXS, [2487](#)
- SUTS, [2497](#)
- SYMBOL, [2640](#)
- SYMPOLY, [2654](#)
- TABLE, [2664](#)
- TEX, [2677](#)
- TEXTFILE, [2692](#)
- TREE, [2741](#)
- TS, [2671](#)
- TUPLE, [2752](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UNISEG, [2896](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- VARIABLE, [2936](#)
- VECTOR, [2942](#)
- VIEW2d, [2770](#)
- VIEW3D, [2710](#)
- WP, [2949](#)
- WUTSET, [2959](#)
- XDPOLY, [2967](#)
- XPBWPOLYL, [2987](#)
- XPOLY, [2998](#)
- XPR, [3007](#)
- XPOLY, [3013](#)
- ZMOD, [1368](#)
- hasHi
 - UNISEG, [2896](#)
- HashTable, [1120](#)
- HASHTBL, [1120](#)
 - ?.?, [1120](#)
 - ?=?, [1120](#)
 - ?~=?, [1120](#)
 - #?, [1120](#)
- any?, [1120](#)
- bag, [1120](#)
- coerce, [1120](#)
- construct, [1120](#)
- convert, [1120](#)
- copy, [1120](#)
- count, [1120](#)
- dictionary, [1120](#)
- elt, [1120](#)
- empty, [1120](#)
- empty?, [1120](#)
- entries, [1120](#)
- entry?, [1120](#)
- eq?, [1120](#)
- eval, [1120](#)
- every?, [1120](#)
- extract, [1120](#)
- fill, [1120](#)
- find, [1120](#)
- first, [1120](#)
- hash, [1120](#)
- index?, [1120](#)
- indices, [1120](#)
- insert, [1120](#)
- inspect, [1120](#)
- key?, [1120](#)
- keys, [1120](#)
- latex, [1120](#)
- less?, [1120](#)
- map, [1120](#)
- maxIndex, [1120](#)
- member?, [1120](#)
- members, [1120](#)
- minIndex, [1120](#)
- more?, [1120](#)
- parts, [1120](#)
- qelt, [1120](#)
- qsetelt, [1120](#)
- reduce, [1120](#)
- remove, [1120](#)
- removeDuplicates, [1120](#)
- sample, [1120](#)
- search, [1120](#)
- select, [1120](#)
- setelt, [1120](#)
- size?, [1120](#)

- swap, [1120](#)
- table, [1120](#)
- hasPredicate?
 - PATTERN, [1928](#)
- hasTopPredicate?
 - PATTERN, [1928](#)
- hclf
 - FMONOID, [1021](#)
 - OFMONOID, [1829](#)
- hconcat
 - OUTFORM, [1868](#)
- hcrf
 - FMONOID, [1021](#)
 - OFMONOID, [1829](#)
- HDMP, [1179](#)
- , [1180](#)
- ?<?, [1180](#)
- ?<=?, [1180](#)
- ?>?, [1180](#)
- ?>=?, [1180](#)
- ?**?, [1180](#)
- ?*?, [1180](#)
- ?+?, [1180](#)
- ?-?, [1180](#)
- ?/?, [1180](#)
- ?=?, [1180](#)
- ?^?, [1180](#)
- ?~=?, [1180](#)
- 0, [1180](#)
- 1, [1180](#)
- associates?, [1180](#)
- binomThmExpt, [1180](#)
- characteristic, [1180](#)
- charthRoot, [1180](#)
- coefficient, [1180](#)
- coefficients, [1180](#)
- coerce, [1180](#)
- conditionP, [1180](#)
- content, [1180](#)
- convert, [1180](#)
- D, [1180](#)
- degree, [1180](#)
- differentiate, [1180](#)
- discriminant, [1180](#)
- eval, [1180](#)
- exquo, [1180](#)
- factor, [1180](#)
- factorPolynomial, [1180](#)
- factorSquareFreePolynomial, [1180](#)
- gcd, [1180](#)
- gcdPolynomial, [1180](#)
- ground, [1180](#)
- ground?, [1180](#)
- hash, [1180](#)
- isExpt, [1180](#)
- isPlus, [1180](#)
- isTimes, [1180](#)
- latex, [1180](#)
- lcm, [1180](#)
- leadingCoefficient, [1180](#)
- leadingMonomial, [1180](#)
- mainVariable, [1180](#)
- map, [1180](#)
- mapExponents, [1180](#)
- max, [1180](#)
- min, [1180](#)
- minimumDegree, [1180](#)
- monicDivide, [1180](#)
- monomial, [1180](#)
- monomial?, [1180](#)
- monomials, [1180](#)
- multivariate, [1180](#)
- numberOfMonomials, [1180](#)
- one?, [1180](#)
- patternMatch, [1180](#)
- pomopo, [1180](#)
- prime?, [1180](#)
- primitiveMonomials, [1180](#)
- primitivePart, [1180](#)
- recip, [1180](#)
- reducedSystem, [1180](#)
- reductum, [1180](#)
- reorder, [1180](#)
- resultant, [1180](#)
- retract, [1180](#)
- retractIfCan, [1180](#)
- sample, [1180](#)
- solveLinearPolynomialEquation, [1180](#)
- squareFree, [1180](#)
- squareFreePart, [1180](#)
- squareFreePolynomial, [1180](#)
- subtractIfCan, [1180](#)

- totalDegree, 1180
- unit?, 1180
- unitCanonical, 1180
- unitNormal, 1180
- univariate, 1180
- variables, 1180
- zero?, 1180
- HDP, 1172
- ?, 1173
- ?<?, 1173
- ?<=?, 1173
- ?>?, 1173
- ?>=?, 1173
- ?**?, 1173
- ?*?, 1173
- ?+?, 1173
- ?-?, 1173
- ?.?, 1173
- ?/? , 1173
- ?=?, 1173
- ?^?, 1173
- ?~=?, 1173
- #?, 1173
- 0, 1173
- 1, 1173
- abs, 1173
- any?, 1173
- characteristic, 1173
- coerce, 1173
- copy, 1173
- count, 1173
- D, 1173
- differentiate, 1173
- dimension, 1173
- directProduct, 1173
- dot, 1173
- elt, 1173
- empty, 1173
- empty?, 1173
- entries, 1173
- entry?, 1173
- eq?, 1173
- eval, 1173
- every?, 1173
- fill, 1173
- first, 1173
- hash, 1173
- index, 1173
- index?, 1173
- indices, 1173
- latex, 1173
- less?, 1173
- lookup, 1173
- map, 1173
- max, 1173
- maxIndex, 1173
- member?, 1173
- members, 1173
- min, 1173
- minIndex, 1173
- more?, 1173
- negative?, 1173
- one?, 1173
- parts, 1173
- positive?, 1173
- qelt, 1173
- qsetelt, 1173
- random, 1173
- recip, 1173
- reducedSystem, 1173
- retract, 1173
- retractIfCan, 1173
- sample, 1173
- setelt, 1173
- sign, 1173
- size, 1173
- size?, 1173
- subtractIfCan, 1173
- sup, 1173
- swap, 1173
- unitVector, 1173
- zero?, 1173
- head
 - DIV, 575
 - NSMP, 1714
- headReduce
 - GTSET, 1083
 - NSMP, 1714
 - REGSET, 2286
 - RGCHAIN, 2255
 - SREGSET, 2535
 - WUTSET, 2959

- headReduced?
 - GTSET, [1083](#)
 - NSMP, [1714](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- headRemainder
 - GPOLSET, [1074](#)
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- HEAP, [1134](#)
 - ?=?, [1134](#)
 - ?~=?, [1134](#)
 - #?, [1134](#)
 - any?, [1134](#)
 - bag, [1134](#)
 - coerce, [1134](#)
 - copy, [1134](#)
 - count, [1134](#)
 - empty, [1134](#)
 - empty?, [1134](#)
 - eq?, [1134](#)
 - eval, [1134](#)
 - every?, [1134](#)
 - extract, [1134](#)
 - hash, [1134](#)
 - heap, [1134](#)
 - insert, [1134](#)
 - inspect, [1134](#)
 - latex, [1134](#)
 - less?, [1134](#)
 - map, [1134](#)
 - max, [1134](#)
 - member?, [1134](#)
 - members, [1134](#)
 - merge, [1134](#)
 - more?, [1134](#)
 - parts, [1134](#)
 - sample, [1134](#)
 - size?, [1134](#)
- Heap, [1134](#)
- heap
 - HEAP, [1134](#)
- height
 - AN, [35](#)
 - DEQUEUE, [512](#)
 - EXPR, [724](#)
 - FEXPR, [947](#)
 - IAN, [1276](#)
 - KERNEL, [1404](#)
 - MYEXPR, [1688](#)
 - OUTFORM, [1868](#)
- HELLFDIV, [1183](#)
 - .?, [1183](#)
 - ?*?, [1183](#)
 - ?+?, [1183](#)
 - ?-?, [1183](#)
 - ?=?, [1183](#)
 - ?~=?, [1183](#)
 - 0, [1183](#)
 - coerce, [1183](#)
 - decompose, [1183](#)
 - divisor, [1183](#)
 - generator, [1183](#)
 - hash, [1183](#)
 - ideal, [1183](#)
 - latex, [1183](#)
 - principal?, [1183](#)
 - reduce, [1183](#)
 - sample, [1183](#)
 - subtractIfCan, [1183](#)
 - zero?, [1183](#)
- hex
 - HEXADEC, [1143](#)
- HEXADEC, [1142](#)
 - .?, [1143](#)
 - ?<?, [1143](#)
 - ?<=?, [1143](#)
 - ?>?, [1143](#)
 - ?>=?, [1143](#)
 - ?**?, [1143](#)
 - ?*?, [1143](#)
 - ?+?, [1143](#)
 - ?-?, [1143](#)
 - ?., [1143](#)
 - ?/? , [1143](#)
 - ?=?, [1143](#)
 - ?^?, [1143](#)

- [?~=?](#), 1143
- [?quo?](#), 1143
- [?rem?](#), 1143
- [0](#), 1143
- [1](#), 1143
- [abs](#), 1143
- [associates?](#), 1143
- [ceiling](#), 1143
- [characteristic](#), 1143
- [charthRoot](#), 1143
- [coerce](#), 1143
- [conditionP](#), 1143
- [convert](#), 1143
- [D](#), 1143
- [denom](#), 1143
- [denominator](#), 1143
- [differentiate](#), 1143
- [divide](#), 1143
- [euclideanSize](#), 1143
- [eval](#), 1143
- [expressIdealMember](#), 1143
- [exquo](#), 1143
- [extendedEuclidean](#), 1143
- [factor](#), 1143
- [factorPolynomial](#), 1143
- [factorSquareFreePolynomial](#), 1143
- [floor](#), 1143
- [fractionPart](#), 1143
- [gcd](#), 1143
- [gcdPolynomial](#), 1143
- [hash](#), 1143
- [hex](#), 1143
- [init](#), 1143
- [inv](#), 1143
- [latex](#), 1143
- [lcm](#), 1143
- [map](#), 1143
- [max](#), 1143
- [min](#), 1143
- [multiEuclidean](#), 1143
- [negative?](#), 1143
- [nextItem](#), 1143
- [numer](#), 1143
- [numerator](#), 1143
- [one?](#), 1143
- [patternMatch](#), 1143
- [positive?](#), 1143
- [prime?](#), 1143
- [principalIdeal](#), 1143
- [random](#), 1143
- [recip](#), 1143
- [reducedSystem](#), 1143
- [retract](#), 1143
- [retractIfCan](#), 1143
- [sample](#), 1143
- [sign](#), 1143
- [sizeLess?](#), 1143
- [solveLinearPolynomialEquation](#), 1143
- [squareFree](#), 1143
- [squareFreePart](#), 1143
- [squareFreePolynomial](#), 1143
- [subtractIfCan](#), 1143
- [unit?](#), 1143
- [unitCanonical](#), 1143
- [unitNormal](#), 1143
- [wholePart](#), 1143
- [zero?](#), 1143
- [HexadecimalExpansion](#), 1142
- [hexDigit](#)
 - [CCLASS](#), 382
- [hexDigit?](#)
 - [CHAR](#), 373
- [hi](#)
 - [SEG](#), 2359
 - [UNISEG](#), 2896
- [high](#)
 - [SEG](#), 2359
 - [UNISEG](#), 2896
- [highCommonTerms](#)
 - [DIV](#), 575
 - [FAGROUP](#), 1004
 - [FAMONOID](#), 1007
 - [IFAMON](#), 1287
- [hitherPlane](#)
 - [VIEW3D](#), 2710
- [homogeneous](#)
 - [GOPT](#), 1104
 - [GOPT0](#), 1110
- [homogeneous?](#)
 - [ANTISYM](#), 40
 - [DERHAM](#), 530
- [HomogeneousDirectProduct](#), 1172

- HomogeneousDistributedMultivariatePolynomial, [?/?](#), [1276](#)
[1179](#)
- homogenize [?=?](#), [1276](#)
[?^?](#), [1276](#)
[?~=?](#), [1276](#)
[?quo?](#), [1276](#)
[?rem?](#), [1276](#)
- horizConcat
[0](#), [1276](#)
[1](#), [1276](#)
[associates?](#), [1276](#)
[belong?](#), [1276](#)
[box](#), [1276](#)
[characteristic](#), [1276](#)
[coerce](#), [1276](#)
[convert](#), [1276](#)
[D](#), [1276](#)
[definingPolynomial](#), [1276](#)
[denom](#), [1276](#)
[differentiate](#), [1276](#)
[distribute](#), [1276](#)
[divide](#), [1276](#)
[elt](#), [1276](#)
[euclideanSize](#), [1276](#)
[eval](#), [1276](#)
[even?](#), [1276](#)
[expressIdealMember](#), [1276](#)
[exquo](#), [1276](#)
[extendedEuclidean](#), [1276](#)
[factor](#), [1276](#)
[freeOf?](#), [1276](#)
[gcd](#), [1276](#)
[gcdPolynomial](#), [1276](#)
[hash](#), [1276](#)
[height](#), [1276](#)
[inv](#), [1276](#)
[is?](#), [1276](#)
[kernel](#), [1276](#)
[kernels](#), [1276](#)
[latex](#), [1276](#)
[lcm](#), [1276](#)
[mainKernel](#), [1276](#)
[map](#), [1276](#)
[max](#), [1276](#)
[min](#), [1276](#)
[minPoly](#), [1276](#)
[multiEuclidean](#), [1276](#)
[norm](#), [1276](#)
- hspace
[OUTFORM](#), [1868](#)
- HTMLFORM, [1152](#)
[?=?](#), [1152](#)
[? =?](#), [1152](#)
[coerce](#), [1152](#)
[coerceL](#), [1152](#)
[coerceS](#), [1152](#)
[display](#), [1152](#)
[expres](#), [1152](#)
[hash](#), [1152](#)
[latex](#), [1152](#)
- HTMLFormat, [1152](#)
- hue
[COLOR](#), [407](#)
[PALETTE](#), [1896](#)
- hyperelliptic
[ALGFF](#), [28](#)
[RADFF](#), [2194](#)
- HyperellipticFiniteDivisor, [1183](#)
- IAN, [1276](#)
[-?](#), [1276](#)
[?<?](#), [1276](#)
[?<=?](#), [1276](#)
[?>?](#), [1276](#)
[?>=?](#), [1276](#)
[?**?](#), [1276](#)
[?*?](#), [1276](#)
[?+?](#), [1276](#)
[?-?](#), [1276](#)

- nthRoot, [1276](#)
- numer, [1276](#)
- odd?, [1276](#)
- one?, [1276](#)
- operator, [1276](#)
- operators, [1276](#)
- paren, [1276](#)
- prime?, [1276](#)
- principalIdeal, [1276](#)
- recip, [1276](#)
- reduce, [1276](#)
- reducedSystem, [1276](#)
- retract, [1276](#)
- retractIfCan, [1276](#)
- rootOf, [1276](#)
- rootsOf, [1276](#)
- sample, [1276](#)
- sizeLess?, [1276](#)
- sqrt, [1276](#)
- squareFree, [1276](#)
- squareFreePart, [1276](#)
- subst, [1276](#)
- subtractIfCan, [1276](#)
- tower, [1276](#)
- trueEqual, [1276](#)
- unit?, [1276](#)
- unitCanonical, [1276](#)
- unitNormal, [1276](#)
- zero?, [1276](#)
- zeroOf, [1276](#)
- zerosOf, [1276](#)
- IARRAY1, [1243](#)
 - ?<?, [1244](#)
 - ?<=?, [1244](#)
 - ?>?, [1244](#)
 - ?>=?, [1244](#)
 - ?..?, [1244](#)
 - ?=?, [1244](#)
 - ?~=?, [1244](#)
 - #?, [1244](#)
 - any?, [1244](#)
 - coerce, [1244](#)
 - concat, [1244](#)
 - construct, [1244](#)
 - convert, [1244](#)
 - copy, [1244](#)
 - copyInto, [1244](#)
 - count, [1244](#)
 - delete, [1244](#)
 - elt, [1244](#)
 - empty, [1244](#)
 - empty?, [1244](#)
 - entries, [1244](#)
 - entry?, [1244](#)
 - eq?, [1244](#)
 - eval, [1244](#)
 - every?, [1244](#)
 - fill, [1244](#)
 - find, [1244](#)
 - first, [1244](#)
 - hash, [1244](#)
 - index?, [1244](#)
 - indices, [1244](#)
 - insert, [1244](#)
 - latex, [1244](#)
 - less?, [1244](#)
 - map, [1244](#)
 - max, [1244](#)
 - maxIndex, [1244](#)
 - member?, [1244](#)
 - members, [1244](#)
 - merge, [1244](#)
 - min, [1244](#)
 - minIndex, [1244](#)
 - more?, [1244](#)
 - new, [1244](#)
 - parts, [1244](#)
 - position, [1244](#)
 - qelt, [1244](#)
 - qsetelt, [1244](#)
 - reduce, [1244](#)
 - remove, [1244](#)
 - removeDuplicates, [1244](#)
 - reverse, [1244](#)
 - sample, [1244](#)
 - select, [1244](#)
 - setelt, [1244](#)
 - size?, [1244](#)
 - sort, [1244](#)
 - sorted?, [1244](#)
 - swap, [1244](#)
- IARRAY2, [1256](#)

- [?=?, 1256](#)
- [?~=?, 1256](#)
- [#?, 1256](#)
- [any?, 1256](#)
- [coerce, 1256](#)
- [column, 1256](#)
- [copy, 1256](#)
- [count, 1256](#)
- [elt, 1256](#)
- [empty, 1256](#)
- [empty?, 1256](#)
- [eq?, 1256](#)
- [eval, 1256](#)
- [every?, 1256](#)
- [fill, 1256](#)
- [hash, 1256](#)
- [latex, 1256](#)
- [less?, 1256](#)
- [map, 1256](#)
- [maxColIndex, 1256](#)
- [maxRowIndex, 1256](#)
- [member?, 1256](#)
- [members, 1256](#)
- [minColIndex, 1256](#)
- [minRowIndex, 1256](#)
- [more?, 1256](#)
- [ncols, 1256](#)
- [new, 1256](#)
- [nrows, 1256](#)
- [parts, 1256](#)
- [qelt, 1256](#)
- [qsetelt, 1256](#)
- [row, 1256](#)
- [sample, 1256](#)
- [setColumn, 1256](#)
- [setelt, 1256](#)
- [setRow, 1256](#)
- [size?, 1256](#)
- [IBITS, 1199](#)
- [?<?, 1199](#)
- [?<=?, 1199](#)
- [?>?, 1199](#)
- [?>=?, 1199](#)
- [?GE30F/?, 1199](#)
- [?..?, 1199](#)
- [?/GE30F?, 1199](#)
- [?=?, 1199](#)
- [?~=?, 1199](#)
- [?and?, 1199](#)
- [?or?, 1199](#)
- [#?, 1199](#)
- [^?, 1199](#)
- [~?, 1199](#)
- [And, 1199](#)
- [any?, 1199](#)
- [coerce, 1199](#)
- [concat, 1199](#)
- [construct, 1199](#)
- [convert, 1199](#)
- [copy, 1199](#)
- [copyInto, 1199](#)
- [count, 1199](#)
- [delete, 1199](#)
- [elt, 1199](#)
- [empty, 1199](#)
- [empty?, 1199](#)
- [entries, 1199](#)
- [entry?, 1199](#)
- [eq?, 1199](#)
- [eval, 1199](#)
- [every?, 1199](#)
- [fill, 1199](#)
- [find, 1199](#)
- [first, 1199](#)
- [hash, 1199](#)
- [index?, 1199](#)
- [indices, 1199](#)
- [insert, 1199](#)
- [latex, 1199](#)
- [less?, 1199](#)
- [map, 1199](#)
- [max, 1199](#)
- [maxIndex, 1199](#)
- [member?, 1199](#)
- [members, 1199](#)
- [merge, 1199](#)
- [min, 1199](#)
- [minIndex, 1199](#)
- [more?, 1199](#)
- [nand, 1199](#)
- [new, 1199](#)
- [nor, 1199](#)

- Not, [1199](#)
- not?, [1199](#)
- Or, [1199](#)
- parts, [1199](#)
- position, [1199](#)
- qelt, [1199](#)
- qsetelt, [1199](#)
- reduce, [1199](#)
- removeDuplicates, [1199](#)
- reverse, [1199](#)
- sample, [1199](#)
- select, [1199](#)
- size?, [1199](#)
- sort, [1199](#)
- sorted?, [1199](#)
- swap, [1199](#)
- xor, [1199](#)
- IC
 - ?=?, [1191](#)
 - ?~=?, [1191](#)
 - actualExtensionV, [1191](#)
 - chartV, [1191](#)
 - coerce, [1191](#)
 - create, [1191](#)
 - curveV, [1191](#)
 - degree, [1191](#)
 - excpDivV, [1191](#)
 - fullOut, [1191](#)
 - fullOutput, [1191](#)
 - hash, [1191](#)
 - latex, [1191](#)
 - localParamV, [1191](#)
 - localPointV, [1191](#)
 - multV, [1191](#)
 - pointV, [1191](#)
 - setchart, [1191](#)
 - setcurve, [1191](#)
 - setexcpDiv, [1191](#)
 - setlocalParam, [1191](#)
 - setlocalPoint, [1191](#)
 - setmult, [1191](#)
 - setpoint, [1191](#)
 - setsubmult, [1191](#)
 - setsymbName, [1191](#)
 - subMultV, [1191](#)
 - symbNameV, [1191](#)
- ICARD, [1193](#)
 - ?<?, [1193](#)
 - ?<=?, [1193](#)
 - ?>?, [1193](#)
 - ?>=?, [1193](#)
 - ?., [1193](#)
 - ?=?, [1193](#)
 - ?~=?, [1193](#)
 - coerce, [1193](#)
 - display, [1193](#)
 - fullDisplay, [1193](#)
 - hash, [1193](#)
 - latex, [1193](#)
 - max, [1193](#)
 - min, [1193](#)
- iCompose
- ISUPS, [1311](#)
- ICP, [1190](#)
- IDEAL, [2081](#)
 - ?**?, [2081](#)
 - ?*?, [2081](#)
 - ?+?, [2081](#)
 - ?=?, [2081](#)
 - ?~=?, [2081](#)
 - backOldPos, [2081](#)
 - coerce, [2081](#)
 - dimension, [2081](#)
 - element?, [2081](#)
 - generalPosition, [2081](#)
 - generators, [2081](#)
 - groebner, [2081](#)
 - groebner?, [2081](#)
 - groebnerIdeal, [2081](#)
 - hash, [2081](#)
 - ideal, [2081](#)
 - in?, [2081](#)
 - inRadical?, [2081](#)
 - intersect, [2081](#)
 - latex, [2081](#)
 - leadingIdeal, [2081](#)
 - one?, [2081](#)
 - quotient, [2081](#)
 - relationsIdeal, [2081](#)
 - saturate, [2081](#)
 - zero?, [2081](#)
 - zeroDim?, [2081](#)

- ideal
 - FDIV, [813](#)
 - FRIDEAL, [995](#)
 - HELLFDIV, [1183](#)
 - IDEAL, [2081](#)
- idealSimplify
 - QALGSET, [2157](#)
- identification
 - LEXP, [1435](#)
- identity
 - DHMATRIX, [491](#)
- identityMatrix
 - M3D, [2703](#)
- IDPAG, [1202](#)
 - , [1202](#)
 - ?*?, [1202](#)
 - ?+?, [1202](#)
 - ?-?, [1202](#)
 - ?=?, [1202](#)
 - ?~=?, [1202](#)
 - 0, [1202](#)
 - coerce, [1202](#)
 - hash, [1202](#)
 - latex, [1202](#)
 - leadingCoefficient, [1202](#)
 - leadingSupport, [1202](#)
 - map, [1202](#)
 - monomial, [1202](#)
 - reductum, [1202](#)
 - sample, [1202](#)
 - subtractIfCan, [1202](#)
 - zero?, [1202](#)
- IDPAM, [1205](#)
 - ?*?, [1206](#)
 - ?+?, [1206](#)
 - ?=?, [1206](#)
 - ?~=?, [1206](#)
 - 0, [1206](#)
 - coerce, [1206](#)
 - hash, [1206](#)
 - latex, [1206](#)
 - leadingCoefficient, [1206](#)
 - leadingSupport, [1206](#)
 - map, [1206](#)
 - monomial, [1206](#)
 - reductum, [1206](#)
 - sample, [1206](#)
 - zero?, [1206](#)
- IDPO, [1209](#)
 - ?=?, [1209](#)
 - ?~=?, [1209](#)
 - coerce, [1209](#)
 - hash, [1209](#)
 - latex, [1209](#)
 - leadingCoefficient, [1209](#)
 - leadingSupport, [1209](#)
 - map, [1209](#)
 - monomial, [1209](#)
 - reductum, [1209](#)
- IDPOAM, [1212](#)
 - ?<?, [1212](#)
 - ?<=?, [1212](#)
 - ?>?, [1212](#)
 - ?>=?, [1212](#)
 - ?*?, [1212](#)
 - ?+?, [1212](#)
 - ?=?, [1212](#)
 - ?~=?, [1212](#)
 - 0, [1212](#)
 - coerce, [1212](#)
 - hash, [1212](#)
 - latex, [1212](#)
 - leadingCoefficient, [1212](#)
 - leadingSupport, [1212](#)
 - map, [1212](#)
 - max, [1212](#)
 - min, [1212](#)
 - monomial, [1212](#)
 - reductum, [1212](#)
 - sample, [1212](#)
 - zero?, [1212](#)
- IDPOAMS, [1214](#)
 - ?<?, [1215](#)
 - ?<=?, [1215](#)
 - ?>?, [1215](#)
 - ?>=?, [1215](#)
 - ?*?, [1215](#)
 - ?+?, [1215](#)
 - ?=?, [1215](#)
 - ?~=?, [1215](#)
 - 0, [1215](#)
 - coerce, [1215](#)

- hash, [1215](#)
- latex, [1215](#)
- leadingCoefficient, [1215](#)
- leadingSupport, [1215](#)
- map, [1215](#)
- max, [1215](#)
- min, [1215](#)
- monomial, [1215](#)
- reductum, [1215](#)
- sample, [1215](#)
- subtractIfCan, [1215](#)
- sup, [1215](#)
- zero?, [1215](#)
- iExquo
 - ISUPS, [1311](#)
- IFAMON, [1286](#)
 - ?*?, [1287](#)
 - ?+?, [1287](#)
 - ?=?, [1287](#)
 - ?~=?, [1287](#)
 - 0, [1287](#)
 - coefficient, [1287](#)
 - coerce, [1287](#)
 - hash, [1287](#)
 - highCommonTerms, [1287](#)
 - latex, [1287](#)
 - mapCoef, [1287](#)
 - mapGen, [1287](#)
 - nthCoef, [1287](#)
 - nthFactor, [1287](#)
 - retract, [1287](#)
 - retractIfCan, [1287](#)
 - sample, [1287](#)
 - size, [1287](#)
 - subtractIfCan, [1287](#)
 - terms, [1287](#)
 - zero?, [1287](#)
- IFARRAY, [1222](#)
 - ?<?, [1222](#)
 - ?<=?, [1222](#)
 - ?>?, [1222](#)
 - ?>=?, [1222](#)
 - ?.?, [1222](#)
 - ?=?, [1222](#)
 - ?~=?, [1222](#)
 - #?, [1222](#)
 - any?, [1222](#)
 - coerce, [1222](#)
 - concat, [1222](#)
 - construct, [1222](#)
 - convert, [1222](#)
 - copy, [1222](#)
 - copyInto, [1222](#)
 - count, [1222](#)
 - delete, [1222](#)
 - elt, [1222](#)
 - empty, [1222](#)
 - empty?, [1222](#)
 - entries, [1222](#)
 - entry?, [1222](#)
 - eq?, [1222](#)
 - eval, [1222](#)
 - every?, [1222](#)
 - fill, [1222](#)
 - find, [1222](#)
 - first, [1222](#)
 - flexibleArray, [1222](#)
 - hash, [1222](#)
 - index?, [1222](#)
 - indices, [1222](#)
 - insert, [1222](#)
 - latex, [1222](#)
 - less?, [1222](#)
 - map, [1222](#)
 - max, [1222](#)
 - maxIndex, [1222](#)
 - member?, [1222](#)
 - members, [1222](#)
 - merge, [1222](#)
 - min, [1222](#)
 - minIndex, [1222](#)
 - more?, [1222](#)
 - new, [1222](#)
 - parts, [1222](#)
 - physicalLength, [1222](#)
 - position, [1222](#)
 - qelt, [1222](#)
 - qsetelt, [1222](#)
 - reduce, [1222](#)
 - remove, [1222](#)
 - removeDuplicates, [1222](#)
 - reverse, [1222](#)

- sample, [1222](#)
- select, [1222](#)
- setelt, [1222](#)
- shrinkable, [1222](#)
- size?, [1222](#)
- sort, [1222](#)
- sorted?, [1222](#)
- swap, [1222](#)
- IFF, [1283](#)
 - ?, [1284](#)
 - ?**?, [1284](#)
 - ?*?, [1284](#)
 - ?+?, [1284](#)
 - ?-?, [1284](#)
 - ?/?, [1284](#)
 - ?=?, [1284](#)
 - ?^?, [1284](#)
 - ?~=?, [1284](#)
 - ?quo?, [1284](#)
 - ?rem?, [1284](#)
 - 0, [1284](#)
 - 1, [1284](#)
 - algebraic?, [1284](#)
 - associates?, [1284](#)
 - basis, [1284](#)
 - characteristic, [1284](#)
 - charthRoot, [1284](#)
 - coerce, [1284](#)
 - conditionP, [1284](#)
 - coordinates, [1284](#)
 - createNormalElement, [1284](#)
 - createPrimitiveElement, [1284](#)
 - D, [1284](#)
 - definingPolynomial, [1284](#)
 - degree, [1284](#)
 - differentiate, [1284](#)
 - dimension, [1284](#)
 - discreteLog, [1284](#)
 - divide, [1284](#)
 - euclideanSize, [1284](#)
 - expressIdealMember, [1284](#)
 - exquo, [1284](#)
 - extendedEuclidean, [1284](#)
 - extensionDegree, [1284](#)
 - factor, [1284](#)
 - factorsOfCyclicGroupSize, [1284](#)
 - Frobenius, [1284](#)
 - gcd, [1284](#)
 - gcdPolynomial, [1284](#)
 - generator, [1284](#)
 - hash, [1284](#)
 - index, [1284](#)
 - inGroundField?, [1284](#)
 - init, [1284](#)
 - inv, [1284](#)
 - latex, [1284](#)
 - lcm, [1284](#)
 - linearAssociatedExp, [1284](#)
 - linearAssociatedLog, [1284](#)
 - linearAssociatedOrder, [1284](#)
 - lookup, [1284](#)
 - minimalPolynomial, [1284](#)
 - multiEuclidean, [1284](#)
 - nextItem, [1284](#)
 - norm, [1284](#)
 - normal?, [1284](#)
 - normalElement, [1284](#)
 - one?, [1284](#)
 - order, [1284](#)
 - prime?, [1284](#)
 - primeFrobenius, [1284](#)
 - primitive?, [1284](#)
 - primitiveElement, [1284](#)
 - principalIdeal, [1284](#)
 - random, [1284](#)
 - recip, [1284](#)
 - representationType, [1284](#)
 - represents, [1284](#)
 - retract, [1284](#)
 - retractIfCan, [1284](#)
 - sample, [1284](#)
 - size, [1284](#)
 - sizeLess?, [1284](#)
 - squareFree, [1284](#)
 - squareFreePart, [1284](#)
 - subtractIfCan, [1284](#)
 - tableForDiscreteLogarithm, [1284](#)
 - trace, [1284](#)
 - transcendenceDegree, [1284](#)
 - transcendent?, [1284](#)
 - unit?, [1284](#)
 - unitCanonical, [1284](#)

- unitNormal, 1284
- zero?, 1284
- iFTable
 - ODEIFTBL, 1768
- IIARRAY2, 1290
 - ?=?, 1290
 - ?~=?, 1290
 - #?, 1290
 - any?, 1290
 - coerce, 1290
 - column, 1290
 - copy, 1290
 - count, 1290
 - elt, 1290
 - empty, 1290
 - empty?, 1290
 - eq?, 1290
 - eval, 1290
 - every?, 1290
 - fill, 1290
 - hash, 1290
 - latex, 1290
 - less?, 1290
 - map, 1290
 - maxColIndex, 1290
 - maxRowIndex, 1290
 - member?, 1290
 - members, 1290
 - minColIndex, 1290
 - minRowIndex, 1290
 - more?, 1290
 - ncols, 1290
 - new, 1290
 - nrows, 1290
 - parts, 1290
 - qelt, 1290
 - qsetelt, 1290
 - row, 1290
 - sample, 1290
 - setColumn, 1290
 - setelt, 1290
 - setRow, 1290
 - size?, 1290
- ILIST, 1231
 - ?<?, 1231
 - ?<=?, 1231
 - ?>?, 1231
 - ?>=?, 1231
 - ?.?, 1231
 - ?.first, 1231
 - ?.last, 1231
 - ?.rest, 1231
 - ?.value, 1231
 - ?=?, 1231
 - ?~=?, 1231
 - #?, 1231
 - any?, 1231
 - child?, 1231
 - children, 1231
 - coerce, 1231
 - concat, 1231
 - construct, 1231
 - convert, 1231
 - copy, 1231
 - copyInto, 1231
 - count, 1231
 - cycleEntry, 1231
 - cycleLength, 1231
 - cycleSplit, 1231
 - cycleTail, 1231
 - cyclic?, 1231
 - delete, 1231
 - distance, 1231
 - elt, 1231
 - empty, 1231
 - empty?, 1231
 - entries, 1231
 - entry?, 1231
 - eq?, 1231
 - eval, 1231
 - every?, 1231
 - explicitlyFinite?, 1231
 - fill, 1231
 - find, 1231
 - first, 1231
 - hash, 1231
 - index?, 1231
 - indices, 1231
 - insert, 1231
 - last, 1231
 - latex, 1231
 - leaf?, 1231

- leaves, [1231](#)
- less?, [1231](#)
- list, [1231](#)
- map, [1231](#)
- max, [1231](#)
- maxIndex, [1231](#)
- member?, [1231](#)
- members, [1231](#)
- merge, [1231](#)
- min, [1231](#)
- minIndex, [1231](#)
- more?, [1231](#)
- new, [1231](#)
- node?, [1231](#)
- nodes, [1231](#)
- parts, [1231](#)
- position, [1231](#)
- possiblyInfinite?, [1231](#)
- qelt, [1231](#)
- qsetelt, [1231](#)
- reduce, [1231](#)
- remove, [1231](#)
- removeDuplicates, [1231](#)
- rest, [1231](#)
- reverse, [1231](#)
- sample, [1231](#)
- second, [1231](#)
- select, [1231](#)
- setchildren, [1231](#)
- setelt, [1231](#)
- setfirst, [1231](#)
- setlast, [1231](#)
- setrest, [1231](#)
- setvalue, [1231](#)
- size?, [1231](#)
- sort, [1231](#)
- sorted?, [1231](#)
- split, [1231](#)
- swap, [1231](#)
- tail, [1231](#)
- third, [1231](#)
- value, [1231](#)
- imag
 - COMPLEX, [419](#)
 - MCMPLX, [1543](#)
- imagE
 - OCT, [1765](#)
- images
 - ACPLOT, [1992](#)
 - AFFPL, [4](#)
 - AFFPLPS, [7](#)
 - AFFSP, [9](#)
 - ALGFF, [27](#)
 - ALGSC, [14](#)
 - ALIST, [235](#)
 - AN, [35](#)
 - ANON, [38](#)
 - ANTISYM, [40](#)
 - ANY, [50](#)
 - ARRAY1, [1774](#)
 - ARRAY2, [2763](#)
 - ASP1, [71](#)
 - ASP10, [75](#)
 - ASP12, [80](#)
 - ASP19, [85](#)
 - ASP20, [92](#)
 - ASP24, [98](#)
 - ASP27, [102](#)
 - ASP28, [108](#)
 - ASP29, [114](#)
 - ASP30, [118](#)
 - ASP31, [122](#)
 - ASP33, [127](#)
 - ASP34, [130](#)
 - ASP35, [134](#)
 - ASP4, [140](#)
 - ASP41, [144](#)
 - ASP42, [151](#)
 - ASP49, [158](#)
 - ASP50, [164](#)
 - ASP55, [170](#)
 - ASP6, [176](#)
 - ASP7, [181](#)
 - ASP73, [185](#)
 - ASP74, [191](#)
 - ASP77, [196](#)
 - ASP78, [201](#)
 - ASP8, [206](#)
 - ASP80, [211](#)
 - ASP9, [216](#)
 - ASTACK, [65](#)
 - ATTRBUT, [239](#)

- AUTOMOR, 245
BBTREE, 252
BFUNCT, 265
BINARY, 292
BINFILE, 295
BITS, 314
BLHN, 317
BLQT, 319
BOOLEAN, 322
BOP, 273
BPADIC, 258
BPADICRT, 263
BSD, 285
BSTREE, 302
BTourn, 306
BTREE, 310
CARD, 332
CARTEN, 356
CCLASS, 381
CDFMAT, 426
CDFVEC, 432
CHAR, 373
CLIF, 401
COLOR, 407
COMM, 411
COMPLEX, 418
COMPPROP, 2624
CONTRFAC, 445
D01AJFA, 614
D01AKFA, 617
D01ALFA, 620
D01AMFA, 623
D01ANFA, 626
D01APFA, 629
D01AQFA, 633
D01ASFA, 636
D01FCFA, 639
D01GBFA, 643
D01TRNS, 646
D02BBFA, 651
D02BHFA, 654
D02CJFA, 658
D02EJFA, 661
D03EEFA, 665
D03FAFA, 668
DBASE, 454
DECIMAL, 465
DEQUEUE, 512
DERHAM, 530
DFLOAT, 586
DFMAT, 599
DFVEC, 605
DHMATRIX, 491
DIRPROD, 547
DIRRING, 563
DIV, 575
DLIST, 459
DMP, 571
DPMM, 552
DPMO, 557
DROPT, 608
DSMP, 541
DSTREE, 534
E04DGFA, 747
E04FDFA, 750
E04GCFA, 754
E04JAFa, 758
E04MBFA, 762
E04NAFA, 765
E04UCFA, 769
EAB, 743
EFULS, 672
EFUPXS, 681
EMR, 701
EQ, 691
EQTBL, 698
EXIT, 707
EXPEXPAN, 711
EXPR, 723
EXPUPXS, 739
FAGROUP, 1004
FAMONOID, 1007
FARRAY, 885
FC, 931
FCOMP, 975
FDIV, 813
FEXPR, 946
FF, 819
FFCG, 824
FFCGP, 834
FFCGX, 829
FFNB, 859

- FFNBP, [871](#)
- FFNBX, [865](#)
- FFP, [850](#)
- FFX, [845](#)
- FGROUP, [1010](#)
- FILE, [802](#)
- FLOAT, [908](#)
- FM, [1013](#)
- FM1, [1016](#)
- FMONOID, [1021](#)
- FNAME, [810](#)
- FNLA, [1027](#)
- FORMULA, [2346](#)
- FORTRAN, [955](#)
- FPARFRAC, [1040](#)
- FR, [786](#)
- FRAC, [985](#)
- FRIDEAL, [994](#)
- FRMOD, [1000](#)
- FSERIES, [978](#)
- FST, [962](#)
- FT, [971](#)
- FTEM, [967](#)
- FUNCTION, [1045](#)
- GCNAALG, [1064](#)
- GDMP, [1052](#)
- GMODPOL, [1059](#)
- GOPT, [1104](#)
- GOPT0, [1110](#)
- GPOLSET, [1074](#)
- GRIMAGE, [1094](#)
- GSERIES, [1089](#)
- GSTBL, [1078](#)
- GTSET, [1082](#)
- HACKPI, [1977](#)
- HASHTBL, [1120](#)
- HDMP, [1179](#)
- HDP, [1172](#)
- HEAP, [1134](#)
- HELLFDIV, [1183](#)
- HEXADEC, [1142](#)
- HTMLFORM, [1152](#)
- IAN, [1276](#)
- IARRAY1, [1243](#)
- IARRAY2, [1256](#)
- IBITS, [1199](#)
- ICARD, [1193](#)
- ICP, [1190](#)
- IDEAL, [2081](#)
- IDPAG, [1202](#)
- IDPAM, [1205](#)
- IDPO, [1209](#)
- IDPOAM, [1212](#)
- IDPOAMS, [1214](#)
- IFAMON, [1286](#)
- IFARRAY, [1222](#)
- IFF, [1283](#)
- IIARRAY2, [1290](#)
- ILIST, [1231](#)
- IMATRIX, [1238](#)
- INDE, [1217](#)
- INFCLSPS, [1271](#)
- INFCLSPT, [1265](#)
- INFORM, [1343](#)
- INT, [1361](#)
- INTABL, [1335](#)
- INTFTBL, [1371](#)
- INTRVL, [1384](#)
- IPADIC, [1294](#)
- IPF, [1303](#)
- IR, [1375](#)
- ISTRING, [1249](#)
- ISUPS, [1310](#)
- ITAYLOR, [1338](#)
- ITUPLE, [1262](#)
- IVECTOR, [1260](#)
- JORDAN, [222](#)
- KAFILE, [1413](#)
- KERNEL, [1404](#)
- LA, [1520](#)
- LAUPOL, [1421](#)
- LEXP, [1435](#)
- LIB, [1428](#)
- LIE, [228](#)
- LIST, [1504](#)
- LMDICT, [1515](#)
- LMOPS, [1509](#)
- LO, [1522](#)
- LODO, [1469](#)
- LODO1, [1479](#)
- LODO2, [1491](#)
- LPOLY, [1446](#)

- LSQM, 1455
LWORD, 1531
M3D, 2702
MAGMA, 1565
MATRIX, 1622
MCMPLX, 1542
MFLOAT, 1547
MINT, 1557
MKCHSET, 1570
MMLFORM, 1603
MODFIELD, 1638
MODMON, 1631
MODMONOM, 1644
MODOP, 1647
MODRING, 1641
MOEBIUS, 1654
MPOLY, 1681
MRING, 1658
MSET, 1670
MYEXPR, 1687
MYUP, 1694
NIPROB, 1746
NNI, 1739
NONE, 1737
NOTTING, 1744
NSDPS, 1701
NSMP, 1713
NSUP, 1728
OCT, 1765
ODEIFTBL, 1768
ODEPROB, 1750
ODP, 1816
ODPOL, 1851
ODR, 1858
ODVAR, 1855
OFMONOID, 1829
OMCONN, 1781
OMDEV, 1784
OMENC, 1789
OMERR, 1792
OMERRK, 1794
OMLO, 1806
ONECOMP, 1777
OP, 1804
OPTPROB, 1753
ORDCOMP, 1809
ORESUP, 2492
OREUP, 2872
OSI, 1864
OUTFORM, 1867
OVAR, 1836
OWP, 1861
PACEXT, 2126
PACOFF, 2135
PACRAT, 2146
PADIC, 1881
PADICRAT, 1885
PADICRC, 1890
PALETTE, 1896
PARPCURV, 1899
PARSCURV, 1901
PARSURF, 1904
PATLRES, 1937
PATRES, 1940
PATTERN, 1928
PBWLB, 2054
PDEPROB, 1756
PENDTREE, 1944
PERM, 1949
PERMGRP, 1959
PF, 2104
PFR, 1913
PI, 2100
PLACES, 2019
PLACESPS, 2021
PLCS, 2023
PLOT, 2029
PLOT3D, 2042
POINT, 2059
POLY, 2077
PR, 2092
PRIMARR, 2109
PRODUCT, 2112
PROJPL, 2117
PROJPLPS, 2119
PROJSP, 2121
PRTITION, 1923
QALGSET, 2157
QEQUAT, 2169
QFORM, 2154
QUAT, 2167
QUEUE, 2184

- RADFF, [2193](#)
- RADIX, [2205](#)
- RECLOS, [2236](#)
- REF, [2249](#)
- REGSET, [2285](#)
- RESRING, [2296](#)
- RESULT, [2300](#)
- RGCHAIN, [2254](#)
- RMATRIX, [2245](#)
- ROIRC, [2310](#)
- ROMAN, [2326](#)
- ROUTINE, [2330](#)
- RULE, [2305](#)
- RULECOLD, [2340](#)
- RULESET, [2342](#)
- SAE, [2399](#)
- SAOS, [2418](#)
- SD, [2572](#)
- SDPOL, [2385](#)
- SDVAR, [2389](#)
- SEG, [2359](#)
- SEGBIND, [2364](#)
- SET, [2372](#)
- SETMN, [2377](#)
- SEX, [2391](#)
- SEXOF, [2394](#)
- SFORT, [2405](#)
- SHDP, [2508](#)
- SINT, [2411](#)
- SMP, [2422](#)
- SMTS, [2440](#)
- SPACE3, [2731](#)
- SPLNODE, [2512](#)
- SPLTREE, [2518](#)
- SQMATRIX, [2547](#)
- SREGSET, [2534](#)
- STACK, [2563](#)
- STBL, [2450](#)
- STREAM, [2581](#)
- STRING, [2606](#)
- STRTBL, [2610](#)
- SUBSPACE, [2614](#)
- SUCH, [2627](#)
- SULS, [2456](#)
- SUP, [2466](#)
- SUPEXPR, [2480](#)
- SUPXS, [2487](#)
- SUTS, [2496](#)
- SWITCH, [2629](#)
- SYMBOL, [2640](#)
- SYMPOLY, [2654](#)
- SYMS, [2696](#)
- SYMTAB, [2648](#)
- TABLE, [2663](#)
- TABEAU, [2666](#)
- TEX, [2676](#)
- TEXTFILE, [2692](#)
- TREE, [2740](#)
- TS, [2670](#)
- TUBE, [2749](#)
- TUPLE, [2752](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2910](#)
- U32VEC, [2931](#)
- U8MAT, [2901](#)
- U8VEC, [2917](#)
- UFPS, [2788](#)
- ULS, [2795](#)
- ULSCONS, [2803](#)
- UNISEG, [2896](#)
- UP, [2827](#)
- UPXS, [2833](#)
- UPXSCONS, [2841](#)
- UPXSSING, [2852](#)
- UTS, [2876](#)
- UTSZ, [2886](#)
- VARIABLE, [2936](#)
- VECTOR, [2941](#)
- VIEW2D, [2769](#)
- VIEW3D, [2710](#)
- VOID, [2945](#)
- WP, [2949](#)
- WUTSET, [2958](#)
- XDPOLY, [2967](#)
- XPBWPLY, [2987](#)
- XPOLY, [2998](#)
- XPR, [3007](#)
- XPOLY, [3013](#)
- ZMOD, [1367](#)
- imagI
- OCT, [1765](#)

- QUAT, 2167
- imagi
 - OCT, 1765
- imaginary
 - COMPLEX, 419
 - MCMPLX, 1543
- imagJ
 - OCT, 1765
 - QUAT, 2167
- imagj
 - OCT, 1765
- imagK
 - OCT, 1765
 - QUAT, 2167
- imagk
 - OCT, 1765
- IMATRIX, 1238
 - , 1239
 - ?**?, 1239
 - ?*?, 1239
 - ?+?, 1239
 - ?-?, 1239
 - ?/?, 1239
 - ?=?, 1239
 - ?~=?, 1239
 - #?, 1239
 - antisymmetric?, 1239
 - any?, 1239
 - coerce, 1239
 - column, 1239
 - copy, 1239
 - count, 1239
 - determinant, 1239
 - diagonal?, 1239
 - diagonalMatrix, 1239
 - elt, 1239
 - empty, 1239
 - empty?, 1239
 - eq?, 1239
 - eval, 1239
 - every?, 1239
 - exquo, 1239
 - fill, 1239
 - hash, 1239
 - horizConcat, 1239
 - inverse, 1239
 - latex, 1239
 - less?, 1239
 - listOfLists, 1239
 - map, 1239
 - matrix, 1239
 - maxColIndex, 1239
 - maxRowIndex, 1239
 - member?, 1239
 - members, 1239
 - minColIndex, 1239
 - minordet, 1239
 - minRowIndex, 1239
 - more?, 1239
 - ncols, 1239
 - new, 1239
 - nrows, 1239
 - nullity, 1239
 - nullSpace, 1239
 - parts, 1239
 - qelt, 1239
 - qsetelt, 1239
 - rank, 1239
 - row, 1239
 - rowEchelon, 1239
 - sample, 1239
 - scalarMatrix, 1239
 - setColumn, 1239
 - setelt, 1239
 - setRow, 1239
 - setsubMatrix, 1239
 - size?, 1239
 - square?, 1239
 - squareTop, 1239
 - subMatrix, 1239
 - swapColumns, 1239
 - swapRows, 1239
 - symmetric?, 1239
 - transpose, 1239
 - vertConcat, 1239
 - zero, 1239
- implies
 - BOOLEAN, 322
- in?
 - IDEAL, 2081
- inc
 - INT, 1362

- MINT, [1557](#)
- ROMAN, [2326](#)
- SINT, [2412](#)
- incr
 - DIV, [575](#)
 - SEG, [2359](#)
 - UNISEG, [2896](#)
- increase
 - ATTRBUT, [239](#)
- increasePrecision
 - DFLOAT, [587](#)
 - FLOAT, [908](#)
 - MFLOAT, [1548](#)
- incrementKthElement
 - SETMN, [2377](#)
- INDE, [1217](#)
 - ?<?, [1218](#)
 - ?<=?, [1218](#)
 - ?>?, [1218](#)
 - ?>=?, [1218](#)
 - ?*?, [1218](#)
 - ?+?, [1218](#)
 - ?=?, [1218](#)
 - ?~=?, [1218](#)
 - 0, [1218](#)
 - coerce, [1218](#)
 - hash, [1218](#)
 - latex, [1218](#)
 - leadingCoefficient, [1218](#)
 - leadingSupport, [1218](#)
 - map, [1218](#)
 - max, [1218](#)
 - min, [1218](#)
 - monomial, [1218](#)
 - reductum, [1218](#)
 - sample, [1218](#)
 - subtractIfCan, [1218](#)
 - sup, [1218](#)
 - zero?, [1218](#)
- index
 - ALGFF, [28](#)
 - BOOLEAN, [322](#)
 - CCLASS, [382](#)
 - CHAR, [373](#)
 - COMPLEX, [419](#)
 - DIRPROD, [547](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - HDP, [1173](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - MCMPLEX, [1543](#)
 - MODMON, [1632](#)
 - MODMONOM, [1644](#)
 - MRING, [1659](#)
 - OCT, [1765](#)
 - ODP, [1816](#)
 - OVAR, [1836](#)
 - PACOFF, [2136](#)
 - PF, [2105](#)
 - PRODUCT, [2113](#)
 - RADFF, [2194](#)
 - SAE, [2400](#)
 - SET, [2372](#)
 - SETMN, [2377](#)
 - SHDP, [2509](#)
 - ZMOD, [1368](#)
- index?
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - BITS, [315](#)
 - CDFVEC, [432](#)
 - DFVEC, [605](#)
 - DIRPROD, [547](#)
 - DLIST, [460](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - EQTBL, [699](#)
 - FARRAY, [886](#)
 - GSTBL, [1079](#)
 - HASHTBL, [1120](#)
 - HDP, [1173](#)
 - IARRAY1, [1244](#)

- IBITS, [1199](#)
- IFARRAY, [1222](#)
- ILIST, [1231](#)
- INTABL, [1336](#)
- ISTRING, [1249](#)
- IVECTOR, [1260](#)
- KAFILE, [1414](#)
- LIB, [1429](#)
- LIST, [1504](#)
- NSDPS, [1703](#)
- ODP, [1816](#)
- POINT, [2060](#)
- PRIMARR, [2109](#)
- RESULT, [2301](#)
- ROUTINE, [2331](#)
- SHDP, [2509](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- U16VEC, [2924](#)
- U32VEC, [2931](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- IndexCard, [1193](#)
- IndexedBits, [1199](#)
- IndexedDirectProductAbelianGroup, [1202](#)
- IndexedDirectProductAbelianMonoid, [1205](#)
- IndexedDirectProductObject, [1209](#)
- IndexedDirectProductOrderedAbelianMonoid, [1212](#)
- IndexedDirectProductOrderedAbelianMonoidSup, [1214](#)
- IndexedExponents, [1217](#)
- IndexedFlexibleArray, [1222](#)
- IndexedList, [1231](#)
- IndexedMatrix, [1238](#)
- IndexedOneDimensionalArray, [1243](#)
- IndexedString, [1249](#)
- IndexedTwoDimensionalArray, [1256](#)
- IndexedVector, [1260](#)
- indexName
 - GOPT, [1104](#)
 - GOPT0, [1110](#)
- indices
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - BITS, [315](#)
 - CDFVEC, [432](#)
 - DFVEC, [605](#)
 - DIRPROD, [547](#)
 - DLIST, [460](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - EQTBL, [699](#)
 - FARRAY, [886](#)
 - GSTBL, [1079](#)
 - HASHTBL, [1120](#)
 - HDP, [1173](#)
 - IARRAY1, [1244](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - ILIST, [1231](#)
 - INTABL, [1336](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - KAFILE, [1414](#)
 - LIB, [1429](#)
 - LIST, [1504](#)
 - NSDPS, [1703](#)
 - ODP, [1816](#)
 - POINT, [2060](#)
 - PRIMARR, [2109](#)
 - RESULT, [2301](#)
 - ROUTINE, [2331](#)
 - SHDP, [2509](#)
 - STBL, [2450](#)
 - STREAM, [2582](#)
 - STRING, [2607](#)
 - STRTBL, [2610](#)
 - TABLE, [2664](#)
 - U16VEC, [2924](#)
 - U32VEC, [2931](#)
 - U8VEC, [2917](#)
 - VECTOR, [2942](#)
- inf
 - INTRVL, [1384](#)
- INFCLSPS, [1271](#)
 - ?=, [1271](#)
 - ?~=, [1271](#)
- actualExtensionV, [1271](#)

- chartV, 1271
- coerce, 1271
- create, 1271
- curveV, 1271
- degree, 1271
- excpDivV, 1271
- fullOut, 1271
- fullOutput, 1271
- hash, 1271
- latex, 1271
- localParamV, 1271
- localPointV, 1271
- multV, 1271
- pointV, 1271
- setchart, 1271
- setcurve, 1271
- setexcpDiv, 1271
- setlocalParam, 1271
- setlocalPoint, 1271
- setmult, 1271
- setpoint, 1271
- setsubmult, 1271
- setsymbName, 1271
- subMultV, 1271
- symbNameV, 1271
- INFCLSPT, 1265
 - ?=?, 1265
 - ?~=?, 1265
 - actualExtensionV, 1265
 - chartV, 1265
 - coerce, 1265
 - create, 1265
 - curveV, 1265
 - degree, 1265
 - excpDivV, 1265
 - fullOut, 1265
 - fullOutput, 1265
 - hash, 1265
 - latex, 1265
 - localParamV, 1265
 - localPointV, 1265
 - multV, 1265
 - pointV, 1265
 - setchart, 1265
 - setcurve, 1265
 - setexcpDiv, 1265
 - setlocalParam, 1265
 - setlocalPoint, 1265
 - setmult, 1265
 - setpoint, 1265
 - setsubmult, 1265
 - setsymbName, 1265
 - subMultV, 1265
 - symbNameV, 1265
- InfClsPt, 1190
- infClsPt?
 - BLHN, 317
 - BLQT, 319
- infinite?
 - ONECOMP, 1777
 - ORDCOMP, 1810
- InfiniteTuple, 1262
- InfinitelyClosePoint, 1265
- InfinitelyClosePointOverPseudoAlgebraicClosureOffiniteField, 1271
- infinity
 - ONECOMP, 1777
- infix
 - OUTFORM, 1868
- infix?
 - OUTFORM, 1868
- infLex?
 - SPLNODE, 2512
- INFORM, 1343
 - ?**?, 1343
 - ?*?, 1343
 - ?+?, 1343
 - ?..?, 1343
 - ?/? , 1343
 - ?=?, 1343
 - ?~=?, 1343
 - #?, 1343
 - 0, 1343
 - 1, 1343
 - atom?, 1343
 - binary, 1343
 - car, 1343
 - cdr, 1343
 - coerce, 1343
 - compile, 1343
 - convert, 1343
 - declare, 1343

- destruct, 1343
- eq, 1343
- expr, 1343
- flatten, 1343
- float, 1343
- float?, 1343
- function, 1343
- hash, 1343
- integer, 1343
- integer?, 1343
- interpret, 1343
- lambda, 1343
- latex, 1343
- list?, 1343
- null?, 1343
- pair?, 1343
- parse, 1343
- string, 1343
- string?, 1343
- symbol, 1343
- symbol?, 1343
- unparse, 1343
- infRittWu?
 - GTSET, 1083
 - NSMP, 1714
 - REGSET, 2286
 - RGCHAIN, 2255
 - SREGSET, 2535
 - WUTSET, 2959
- inGroundField?
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866
 - FFP, 851
 - FFX, 846
 - IFF, 1284
 - IPF, 1303
 - PACOFF, 2136
 - PACRAT, 2146
 - PF, 2105
- init
 - ALGFF, 28
 - BINARY, 292
 - BPADICRT, 263
 - COMPLEX, 419
 - DECIMAL, 465
 - EXPEXPAN, 712
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866
 - FFP, 851
 - FFX, 846
 - FRAC, 986
 - HEXADEC, 1143
 - IFF, 1284
 - INT, 1362
 - IPF, 1303
 - MCMPLEX, 1543
 - MINT, 1557
 - MODMON, 1632
 - MYUP, 1695
 - NSMP, 1714
 - NSUP, 1729
 - PACOFF, 2136
 - PADICRAT, 1886
 - PADICRC, 1891
 - PF, 2105
 - RADFF, 2194
 - RADIX, 2206
 - ROMAN, 2326
 - SAE, 2400
 - SINT, 2412
 - SULS, 2457
 - SUP, 2467
 - SUPEXPR, 2481
 - ULS, 2796
 - ULSCONS, 2804
 - UP, 2828
 - ZMOD, 1368
- initial
 - DSMP, 542
 - ODPOL, 1852
 - SDPOL, 2386
- initializeGroupForWordProblem

- PERMGRP, [1960](#)
- initiallyReduce
 - GTSET, [1083](#)
 - NSMP, [1714](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- initiallyReduced?
 - GTSET, [1083](#)
 - NSMP, [1714](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- initials
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- InnerAlgebraicNumber, [1276](#)
- InnerFiniteField, [1283](#)
- InnerFreeAbelianMonoid, [1286](#)
- InnerIndexedTwoDimensionalArray, [1290](#)
- InnerPAdicInteger, [1294](#)
- InnerPrimeField, [1303](#)
- InnerSparseUnivariatePowerSeries, [1310](#)
- InnerTable, [1335](#)
- InnerTaylorSeries, [1338](#)
- input
 - BOP, [274](#)
- InputForm, [1343](#)
- inR?
 - PATTERN, [1928](#)
- inRadical?
 - IDEAL, [2081](#)
- insert
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - BITS, [315](#)
 - CDFVEC, [432](#)
 - DFVEC, [605](#)
 - DLIST, [460](#)
 - FARRAY, [886](#)
 - IARRAY1, [1244](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - ILIST, [1231](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - LIST, [1504](#)
 - NSDPS, [1703](#)
 - POINT, [2060](#)
 - PRIMARR, [2109](#)
 - STREAM, [2582](#)
 - STRING, [2607](#)
 - U16VEC, [2924](#)
 - U32VEC, [2931](#)
 - U8VEC, [2917](#)
 - VECTOR, [2942](#)
- insertMatch
 - PATRES, [1940](#)
- inspect
 - ALIST, [235](#)
 - ASTACK, [65](#)
 - CCLASS, [382](#)
 - DEQUEUE, [512](#)
 - EQTBL, [699](#)
 - GSTBL, [1079](#)
 - HASHTBL, [1120](#)
 - HEAP, [1134](#)
 - INTABL, [1336](#)
 - KAFILE, [1414](#)
 - LIB, [1429](#)
 - LMDICT, [1515](#)
 - MSET, [1670](#)
 - QUEUE, [2184](#)
 - RESULT, [2301](#)
 - ROUTINE, [2331](#)
 - SET, [2372](#)
 - STACK, [2563](#)
 - STBL, [2450](#)
 - STRTBL, [2610](#)
 - TABLE, [2664](#)
- INT, [1361](#)
 - , [1362](#)
 - ?<?, [1362](#)
 - ?<=?, [1362](#)
 - ?>?, [1362](#)
 - ?>=?, [1362](#)
 - ?**?, [1362](#)

- [?*?, 1362](#)
- [?+?, 1362](#)
- [?-?, 1362](#)
- [?=?, 1362](#)
- [?^?, 1362](#)
- [?~=?, 1362](#)
- [?quo?, 1362](#)
- [?rem?, 1362](#)
- [0, 1362](#)
- [1, 1362](#)
- [abs, 1362](#)
- [addmod, 1362](#)
- [associates?, 1362](#)
- [base, 1362](#)
- [binomial, 1362](#)
- [bit?, 1362](#)
- [characteristic, 1362](#)
- [coerce, 1362](#)
- [convert, 1362](#)
- [copy, 1362](#)
- [D, 1362](#)
- [dec, 1362](#)
- [differentiate, 1362](#)
- [divide, 1362](#)
- [euclideanSize, 1362](#)
- [even?, 1362](#)
- [expressIdealMember, 1362](#)
- [exquo, 1362](#)
- [extendedEuclidean, 1362](#)
- [factor, 1362](#)
- [factorial, 1362](#)
- [gcd, 1362](#)
- [gcdPolynomial, 1362](#)
- [hash, 1362](#)
- [inc, 1362](#)
- [init, 1362](#)
- [invmod, 1362](#)
- [latex, 1362](#)
- [lcm, 1362](#)
- [length, 1362](#)
- [mask, 1362](#)
- [max, 1362](#)
- [min, 1362](#)
- [mulmod, 1362](#)
- [multiEuclidean, 1362](#)
- [negative?, 1362](#)
- [nextItem, 1362](#)
- [odd?, 1362](#)
- [OMwrite, 1362](#)
- [one?, 1362](#)
- [patternMatch, 1362](#)
- [permutation, 1362](#)
- [positive?, 1362](#)
- [positiveRemainder, 1362](#)
- [powmod, 1362](#)
- [prime?, 1362](#)
- [principalIdeal, 1362](#)
- [random, 1362](#)
- [rational, 1362](#)
- [rational?, 1362](#)
- [rationalIfCan, 1362](#)
- [recip, 1362](#)
- [reducedSystem, 1362](#)
- [retract, 1362](#)
- [retractIfCan, 1362](#)
- [sample, 1362](#)
- [shift, 1362](#)
- [sign, 1362](#)
- [sizeLess?, 1362](#)
- [squareFree, 1362](#)
- [squareFreePart, 1362](#)
- [submod, 1362](#)
- [subtractIfCan, 1362](#)
- [symmetricRemainder, 1362](#)
- [unit?, 1362](#)
- [unitCanonical, 1362](#)
- [unitNormal, 1362](#)
- [zero?, 1362](#)
- [int](#)
 - [OUTFORM, 1868](#)
- [INTABL, 1335](#)
 - [?., 1336](#)
 - [?=?, 1336](#)
 - [?~=?, 1336](#)
 - [#?, 1336](#)
 - [any?, 1336](#)
 - [bag, 1336](#)
 - [coerce, 1336](#)
 - [construct, 1336](#)
 - [convert, 1336](#)
 - [copy, 1336](#)
 - [count, 1336](#)

- dictionary, [1336](#)
- elt, [1336](#)
- empty, [1336](#)
- empty?, [1336](#)
- entries, [1336](#)
- entry?, [1336](#)
- eq?, [1336](#)
- eval, [1336](#)
- every?, [1336](#)
- extract, [1336](#)
- fill, [1336](#)
- find, [1336](#)
- first, [1336](#)
- hash, [1336](#)
- index?, [1336](#)
- indices, [1336](#)
- insert, [1336](#)
- inspect, [1336](#)
- key?, [1336](#)
- keys, [1336](#)
- latex, [1336](#)
- less?, [1336](#)
- map, [1336](#)
- maxIndex, [1336](#)
- member?, [1336](#)
- members, [1336](#)
- minIndex, [1336](#)
- more?, [1336](#)
- parts, [1336](#)
- qelt, [1336](#)
- qsetelt, [1336](#)
- reduce, [1336](#)
- remove, [1336](#)
- removeDuplicates, [1336](#)
- sample, [1336](#)
- search, [1336](#)
- select, [1336](#)
- setelt, [1336](#)
- size?, [1336](#)
- swap, [1336](#)
- table, [1336](#)
- Integer, [1361](#)
- integer
 - INFORM, [1343](#)
 - SEX, [2392](#)
 - SEXOF, [2394](#)
- integer?
 - FST, [962](#)
 - INFORM, [1343](#)
 - SEX, [2392](#)
 - SEXOF, [2394](#)
- integerDecode
 - DFLOAT, [587](#)
- IntegerMod, [1367](#)
- integral
 - EXPR, [724](#)
 - IR, [1375](#)
- integral?
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- integralAtInfinity?
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- integralBasis
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- integralBasisAtInfinity
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- integralCoordinates
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- integralDerivationMatrix
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- integralMatrix
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- integralMatrixAtInfinity
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- integralRepresents
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- integrate
 - EXPUPXS, [740](#)
 - GSERIES, [1090](#)
 - ISUPS, [1311](#)
 - MODMON, [1632](#)
 - MYUP, [1695](#)
 - NSUP, [1729](#)
 - POLY, [2078](#)

- SMTS, [2441](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXP, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- TS, [2671](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- IntegrationFunctionsTable, [1371](#)
- IntegrationResult, [1375](#)
- intensity
 - VIEW3D, [2710](#)
- internal?
 - SUBSPACE, [2614](#)
- internalAugment
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
- internalZeroSetSplit
 - REGSET, [2286](#)
 - SREGSET, [2535](#)
- interpret
 - INFORM, [1343](#)
- intersect
 - CCLASS, [382](#)
 - IDEAL, [2081](#)
 - MSET, [1670](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SET, [2372](#)
 - SREGSET, [2535](#)
- Interval, [1384](#)
- interval
 - INTRVL, [1384](#)
- INTFTBL, [1371](#)
 - clearTheFTable, [1371](#)
 - entries, [1371](#)
 - entry, [1371](#)
 - fTable, [1371](#)
 - insert, [1371](#)
 - keys, [1371](#)
 - showAttributes, [1371](#)
 - showTheFTable, [1371](#)
- INTRVL, [1384](#)
 - , [1384](#)
 - ?<?, [1384](#)
 - ?<=?, [1384](#)
 - ?>?, [1384](#)
 - ?>=?, [1384](#)
 - ?**?, [1384](#)
 - ?*?, [1384](#)
 - ?+?, [1384](#)
 - ?-?, [1384](#)
 - ?=?, [1384](#)
 - ?^?, [1384](#)
 - ?~=?, [1384](#)
 - 0, [1384](#)
 - 1, [1384](#)
 - acos, [1384](#)
 - acosh, [1384](#)
 - acot, [1384](#)
 - acoth, [1384](#)
 - acsc, [1384](#)
 - acsch, [1384](#)
 - asec, [1384](#)
 - asech, [1384](#)
 - asin, [1384](#)
 - asinh, [1384](#)
 - associates?, [1384](#)
 - atan, [1384](#)
 - atanh, [1384](#)
 - characteristic, [1384](#)
 - coerce, [1384](#)
 - contains?, [1384](#)
 - cos, [1384](#)
 - cosh, [1384](#)
 - cot, [1384](#)
 - coth, [1384](#)
 - csc, [1384](#)
 - csch, [1384](#)
 - exp, [1384](#)
 - exquo, [1384](#)
 - gcd, [1384](#)
 - gcdPolynomial, [1384](#)
 - hash, [1384](#)

- inf, 1384
- interval, 1384
- latex, 1384
- lcm, 1384
- log, 1384
- max, 1384
- min, 1384
- negative?, 1384
- nthRoot, 1384
- one?, 1384
- pi, 1384
- positive?, 1384
- qinterval, 1384
- recip, 1384
- retract, 1384
- retractIfCan, 1384
- sample, 1384
- sec, 1384
- sech, 1384
- sin, 1384
- sinh, 1384
- sqrt, 1384
- subtractIfCan, 1384
- sup, 1384
- tan, 1384
- tanh, 1384
- unit?, 1384
- unitCanonical, 1384
- unitNormal, 1384
- width, 1384
- zero?, 1384
- inv
 - ALGFF, 28
 - AN, 35
 - AUTOMOR, 245
 - BINARY, 292
 - BPADICRT, 263
 - COMPLEX, 419
 - CONTFRAC, 445
 - DECIMAL, 465
 - DFLOAT, 587
 - EMR, 702
 - EQ, 691
 - EXPEXPAN, 712
 - EXPR, 724
 - EXPUPXS, 740
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866
 - FFP, 851
 - FFX, 846
 - FGROUP, 1010
 - FLOAT, 908
 - FRAC, 986
 - FRIDEAL, 995
 - GSERIES, 1090
 - HACKPI, 1978
 - HEXADEC, 1143
 - IAN, 1276
 - IFF, 1284
 - IPF, 1303
 - LEXP, 1435
 - MCMPLEX, 1543
 - MFLOAT, 1548
 - MODFIELD, 1638
 - MODRING, 1641
 - MOEBIUS, 1654
 - MYEXPR, 1688
 - NOTTING, 1744
 - NSDPS, 1703
 - OCT, 1765
 - ODR, 1858
 - PACOFF, 2136
 - PACRAT, 2146
 - PADICRAT, 1886
 - PADICRC, 1891
 - PERM, 1949
 - PF, 2105
 - PFR, 1914
 - PRODUCT, 2113
 - QUAT, 2167
 - RADFF, 2194
 - RADIX, 2206
 - RECLOS, 2237
 - SAE, 2400
 - SULS, 2457
 - SUPXS, 2487
 - ULS, 2796

- ULSCONS, [2804](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- inverse
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- inverseIntegralMatrix
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- inverseIntegralMatrixAtInfinity
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- invertible?
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
- invertibleElseSplit?
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
- invertibleSet
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
- invmod
 - INT, [1362](#)
 - MINT, [1557](#)
 - ROMAN, [2326](#)
 - SINT, [2412](#)
- invmultisect
 - UFPS, [2789](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- iomode
 - BINFILE, [295](#)
 - FILE, [802](#)
 - FTEM, [967](#)
 - KAFILE, [1414](#)
- TEXTFILE, [2692](#)
- IPADIC, [1294](#)
 - , [1294](#)
 - ?**, [1294](#)
 - ?*, [1294](#)
 - ?+?, [1294](#)
 - ?-?, [1294](#)
 - ?=?, [1294](#)
 - ?^?, [1294](#)
 - ?~=?, [1294](#)
 - ?quo?, [1294](#)
 - ?rem?, [1294](#)
 - 0, [1294](#)
 - 1, [1294](#)
 - approximate, [1294](#)
 - associates?, [1294](#)
 - characteristic, [1294](#)
 - coerce, [1294](#)
 - complete, [1294](#)
 - digits, [1294](#)
 - divide, [1294](#)
 - euclideanSize, [1294](#)
 - expressIdealMember, [1294](#)
 - exquo, [1294](#)
 - extend, [1294](#)
 - extendedEuclidean, [1294](#)
 - gcd, [1294](#)
 - gcdPolynomial, [1294](#)
 - hash, [1294](#)
 - latex, [1294](#)
 - lcm, [1294](#)
 - moduloP, [1294](#)
 - modulus, [1294](#)
 - multiEuclidean, [1294](#)
 - one?, [1294](#)
 - order, [1294](#)
 - principalIdeal, [1294](#)
 - quotientByP, [1294](#)
 - recip, [1294](#)
 - root, [1294](#)
 - sample, [1294](#)
 - sizeLess?, [1294](#)
 - sqrt, [1294](#)
 - subtractIfCan, [1294](#)
 - unit?, [1294](#)
 - unitCanonical, [1294](#)

- unitNormal, 1294
- zero?, 1294
- IPF, 1303
 - ?, 1303
 - ?**?, 1303
 - ?*?, 1303
 - ?+?, 1303
 - ?-?, 1303
 - ?/? , 1303
 - ?=?, 1303
 - ?^?, 1303
 - ?~=?, 1303
 - ?quo?, 1303
 - ?rem?, 1303
 - 0, 1303
 - 1, 1303
 - algebraic?, 1303
 - associates?, 1303
 - basis, 1303
 - characteristic, 1303
 - charthRoot, 1303
 - coerce, 1303
 - conditionP, 1303
 - convert, 1303
 - coordinates, 1303
 - createNormalElement, 1303
 - createPrimitiveElement, 1303
 - D, 1303
 - definingPolynomial, 1303
 - degree, 1303
 - differentiate, 1303
 - dimension, 1303
 - discreteLog, 1303
 - divide, 1303
 - euclideanSize, 1303
 - expressIdealMember, 1303
 - exquo, 1303
 - extendedEuclidean, 1303
 - extensionDegree, 1303
 - factor, 1303
 - factorsOfCyclicGroupSize, 1303
 - Frobenius, 1303
 - gcd, 1303
 - gcdPolynomial, 1303
 - generator, 1303
 - hash, 1303
 - index, 1303
 - inGroundField?, 1303
 - init, 1303
 - inv, 1303
 - latex, 1303
 - lcm, 1303
 - linearAssociatedExp, 1303
 - linearAssociatedLog, 1303
 - linearAssociatedOrder, 1303
 - lookup, 1303
 - minimalPolynomial, 1303
 - multiEuclidean, 1303
 - nextItem, 1303
 - norm, 1303
 - normal?, 1303
 - normalElement, 1303
 - one?, 1303
 - order, 1303
 - prime?, 1303
 - primeFrobenius, 1303
 - primitive?, 1303
 - primitiveElement, 1303
 - principalIdeal, 1303
 - random, 1303
 - recip, 1303
 - representationType, 1303
 - represents, 1303
 - retract, 1303
 - retractIfCan, 1303
 - sample, 1303
 - size, 1303
 - sizeLess?, 1303
 - squareFree, 1303
 - squareFreePart, 1303
 - subtractIfCan, 1303
 - tableForDiscreteLogarithm, 1303
 - trace, 1303
 - transcendenceDegree, 1303
 - transcendent?, 1303
 - unit?, 1303
 - unitCanonical, 1303
 - unitNormal, 1303
 - zero?, 1303
- IR, 1375
 - ?, 1375
 - ?*?, 1375

- ?+?, [1375](#)
- ?-?, [1375](#)
- ?=?, [1375](#)
- ?~=?, [1375](#)
- 0, [1375](#)
- coerce, [1375](#)
- differentiate, [1375](#)
- elem?, [1375](#)
- hash, [1375](#)
- integral, [1375](#)
- latex, [1375](#)
- logpart, [1375](#)
- mkAnswer, [1375](#)
- notelem, [1375](#)
- ratpart, [1375](#)
- retract, [1375](#)
- retractIfCan, [1375](#)
- sample, [1375](#)
- subtractIfCan, [1375](#)
- zero?, [1375](#)
- irreducibleFactor
 - FR, [786](#)
- is?
 - AN, [35](#)
 - BOP, [274](#)
 - EXPR, [724](#)
 - FEXPR, [947](#)
 - IAN, [1276](#)
 - KERNEL, [1404](#)
 - MYEXPR, [1688](#)
- isExpt
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPR, [724](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYEXPR, [1688](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - PATTERN, [1928](#)
 - POLY, [2078](#)
 - SDPOL, [2386](#)
- SMP, [2423](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- UP, [2828](#)
- isList
 - PATTERN, [1928](#)
- isMult
 - EXPR, [724](#)
 - MYEXPR, [1688](#)
- isobaric?
 - DSMP, [542](#)
 - ODPOL, [1852](#)
 - SDPOL, [2386](#)
- isOp
 - PATTERN, [1928](#)
- isPlus
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPR, [724](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYEXPR, [1688](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - PATTERN, [1928](#)
 - POLY, [2078](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - UP, [2828](#)
- isPower
 - EXPR, [724](#)
 - MYEXPR, [1688](#)
 - PATTERN, [1928](#)
- isQuotient
 - PATTERN, [1928](#)
- isTimes
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPR, [724](#)
 - GDMP, [1052](#)

- HDMP, 1180
- MODMON, 1632
- MPOLY, 1682
- MYEXPR, 1688
- MYUP, 1695
- NSMP, 1714
- NSUP, 1729
- ODPOL, 1852
- PATTERN, 1928
- POLY, 2078
- SDPOL, 2386
- SMP, 2423
- SUP, 2467
- SUPEXPR, 2481
- UP, 2828
- ISTRING, 1249
 - ?<?, 1249
 - ?<=?, 1249
 - ?>?, 1249
 - ?>=?, 1249
 - ?.?, 1249
 - ?=?, 1249
 - ?~=?, 1249
 - #?, 1249
 - any?, 1249
 - coerce, 1249
 - concat, 1249
 - construct, 1249
 - convert, 1249
 - copy, 1249
 - copyInto, 1249
 - count, 1249
 - delete, 1249
 - elt, 1249
 - empty, 1249
 - empty?, 1249
 - entries, 1249
 - entry?, 1249
 - eq?, 1249
 - eval, 1249
 - every?, 1249
 - fill, 1249
 - find, 1249
 - first, 1249
 - hash, 1249
 - index?, 1249
 - indices, 1249
 - insert, 1249
 - latex, 1249
 - leftTrim, 1249
 - less?, 1249
 - lowerCase, 1249
 - map, 1249
 - match, 1249
 - match?, 1249
 - max, 1249
 - maxIndex, 1249
 - member?, 1249
 - members, 1249
 - merge, 1249
 - min, 1249
 - minIndex, 1249
 - more?, 1249
 - new, 1249
 - parts, 1249
 - position, 1249
 - prefix?, 1249
 - qelt, 1249
 - qsetelt, 1249
 - reduce, 1249
 - remove, 1249
 - removeDuplicates, 1249
 - replace, 1249
 - reverse, 1249
 - rightTrim, 1249
 - sample, 1249
 - select, 1249
 - setelt, 1249
 - size?, 1249
 - sort, 1249
 - sorted?, 1249
 - split, 1249
 - substring?, 1249
 - suffix?, 1249
 - swap, 1249
 - trim, 1249
 - upperCase, 1249
- ISUPS, 1310
 - , 1311
 - ?**?, 1311
 - ?*?, 1311
 - ?+?, 1311

- [?-, 1311](#)
- [?., 1311](#)
- [?/? , 1311](#)
- [?=?, 1311](#)
- [?^?, 1311](#)
- [?~=?, 1311](#)
- [0, 1311](#)
- [1, 1311](#)
- [approximate, 1311](#)
- [associates?, 1311](#)
- [cAcos, 1311](#)
- [cAcosh, 1311](#)
- [cAcot, 1311](#)
- [cAcoth, 1311](#)
- [cAcsc, 1311](#)
- [cAcsch, 1311](#)
- [cAsec, 1311](#)
- [cAsech, 1311](#)
- [cAsin, 1311](#)
- [cAsinh, 1311](#)
- [cAtan, 1311](#)
- [cAtanh, 1311](#)
- [cCos, 1311](#)
- [cCosh, 1311](#)
- [cCot, 1311](#)
- [cCoth, 1311](#)
- [cCsc, 1311](#)
- [cCsch, 1311](#)
- [center, 1311](#)
- [cExp, 1311](#)
- [characteristic, 1311](#)
- [charthRoot, 1311](#)
- [cLog, 1311](#)
- [coefficient, 1311](#)
- [coerce, 1311](#)
- [complete, 1311](#)
- [cPower, 1311](#)
- [cRationalPower, 1311](#)
- [cSec, 1311](#)
- [cSech, 1311](#)
- [cSin, 1311](#)
- [cSinh, 1311](#)
- [cTan, 1311](#)
- [cTanh, 1311](#)
- [D, 1311](#)
- [degree, 1311](#)
- [differentiate, 1311](#)
- [eval, 1311](#)
- [exquo, 1311](#)
- [extend, 1311](#)
- [getRef, 1311](#)
- [getStream, 1311](#)
- [hash, 1311](#)
- [iCompose, 1311](#)
- [iExquo, 1311](#)
- [integrate, 1311](#)
- [latex, 1311](#)
- [leadingCoefficient, 1311](#)
- [leadingMonomial, 1311](#)
- [makeSeries, 1311](#)
- [map, 1311](#)
- [monomial, 1311](#)
- [monomial?, 1311](#)
- [multiplyCoefficients, 1311](#)
- [multiplyExponents, 1311](#)
- [one?, 1311](#)
- [order, 1311](#)
- [pole?, 1311](#)
- [recip, 1311](#)
- [reductum, 1311](#)
- [sample, 1311](#)
- [series, 1311](#)
- [seriesToOutputForm, 1311](#)
- [subtractIfCan, 1311](#)
- [taylorQuoByVar, 1311](#)
- [terms, 1311](#)
- [truncate, 1311](#)
- [unit?, 1311](#)
- [unitCanonical, 1311](#)
- [unitNormal, 1311](#)
- [variable, 1311](#)
- [variables, 1311](#)
- [zero?, 1311](#)
- [ITAYLOR, 1338](#)
- [-?, 1338](#)
- [?**?, 1338](#)
- [?*?, 1338](#)
- [?+?, 1338](#)
- [?-?, 1338](#)
- [?=?, 1338](#)
- [?^?, 1338](#)
- [?~=?, 1338](#)

- 0, [1338](#)
- 1, [1338](#)
- associates?, [1338](#)
- characteristic, [1338](#)
- coefficients, [1338](#)
- coerce, [1338](#)
- exquo, [1338](#)
- hash, [1338](#)
- latex, [1338](#)
- one?, [1338](#)
- order, [1338](#)
- pole?, [1338](#)
- recip, [1338](#)
- sample, [1338](#)
- series, [1338](#)
- subtractIfCan, [1338](#)
- unit?, [1338](#)
- unitCanonical, [1338](#)
- unitNormal, [1338](#)
- zero?, [1338](#)
- iteratedInitials
 - NSMP, [1714](#)
- ITUPLE, [1262](#)
 - coerce, [1262](#)
 - construct, [1262](#)
 - filterUntil, [1262](#)
 - filterWhile, [1262](#)
 - generate, [1262](#)
 - map, [1262](#)
 - select, [1262](#)
- IVECTOR, [1260](#)
 - ?, [1260](#)
 - ?<?, [1260](#)
 - ?<=?, [1260](#)
 - ?>?, [1260](#)
 - ?>=?, [1260](#)
 - ?*?, [1260](#)
 - ?+?, [1260](#)
 - ?-?, [1260](#)
 - ?., [1260](#)
 - ?=?, [1260](#)
 - ?~=?, [1260](#)
 - #?, [1260](#)
 - any?, [1260](#)
 - coerce, [1260](#)
 - concat, [1260](#)
 - construct, [1260](#)
 - convert, [1260](#)
 - copy, [1260](#)
 - copyInto, [1260](#)
 - count, [1260](#)
 - cross, [1260](#)
 - delete, [1260](#)
 - dot, [1260](#)
 - elt, [1260](#)
 - empty, [1260](#)
 - empty?, [1260](#)
 - entries, [1260](#)
 - entry?, [1260](#)
 - eq?, [1260](#)
 - eval, [1260](#)
 - every?, [1260](#)
 - fill, [1260](#)
 - find, [1260](#)
 - first, [1260](#)
 - hash, [1260](#)
 - index?, [1260](#)
 - indices, [1260](#)
 - insert, [1260](#)
 - latex, [1260](#)
 - length, [1260](#)
 - less?, [1260](#)
 - magnitude, [1260](#)
 - map, [1260](#)
 - max, [1260](#)
 - maxIndex, [1260](#)
 - member?, [1260](#)
 - members, [1260](#)
 - merge, [1260](#)
 - min, [1260](#)
 - minIndex, [1260](#)
 - more?, [1260](#)
 - new, [1260](#)
 - outerProduct, [1260](#)
 - parts, [1260](#)
 - position, [1260](#)
 - qelt, [1260](#)
 - qsetelt, [1260](#)
 - reduce, [1260](#)
 - remove, [1260](#)
 - removeDuplicates, [1260](#)
 - reverse, [1260](#)

- sample, 1260
- select, 1260
- setelt, 1260
- size?, 1260
- sort, 1260
- sorted?, 1260
- swap, 1260
- zero, 1260
- jacobiIdentity?
 - ALGSC, 15
 - GCNAALG, 1065
 - JORDAN, 223
 - LIE, 229
 - LSQM, 1456
- JORDAN, 222
 - , 223
 - ***?, 223
 - *?*, 223
 - +?, 223
 - ?-, 223
 - ?.?, 223
 - ?=?, 223
 - ?~=?, 223
 - 0, 223
 - alternative?, 223
 - antiAssociative?, 223
 - antiCommutative?, 223
 - antiCommutator, 223
 - apply, 223
 - associative?, 223
 - associator, 223
 - associatorDependence, 223
 - basis, 223
 - coerce, 223
 - commutative?, 223
 - commutator, 223
 - conditionsForIdempotents, 223
 - convert, 223
 - coordinates, 223
 - flexible?, 223
 - hash, 223
 - jacobiIdentity?, 223
 - jordanAdmissible?, 223
 - jordanAlgebra?, 223
 - latex, 223
 - leftAlternative?, 223
 - leftCharacteristicPolynomial, 223
 - leftDiscriminant, 223
 - leftMinimalPolynomial, 223
 - leftNorm, 223
 - leftPower, 223
 - leftRankPolynomial, 223
 - leftRecip, 223
 - leftRegularRepresentation, 223
 - leftTrace, 223
 - leftTraceMatrix, 223
 - leftUnit, 223
 - leftUnits, 223
 - lieAdmissible?, 223
 - lieAlgebra?, 223
 - noncommutativeJordanAlgebra?, 223
 - plenaryPower, 223
 - powerAssociative?, 223
 - rank, 223
 - recip, 223
 - represents, 223
 - rightAlternative?, 223
 - rightCharacteristicPolynomial, 223
 - rightDiscriminant, 223
 - rightMinimalPolynomial, 223
 - rightNorm, 223
 - rightPower, 223
 - rightRankPolynomial, 223
 - rightRecip, 223
 - rightRegularRepresentation, 223
 - rightTrace, 223
 - rightTraceMatrix, 223
 - rightUnit, 223
 - rightUnits, 223
 - sample, 223
 - someBasis, 223
 - structuralConstants, 223
 - subtractIfCan, 223
 - unit, 223
 - zero?, 223
- jordanAdmissible?
 - ALGSC, 15
 - GCNAALG, 1065
 - JORDAN, 223
 - LIE, 229
 - LSQM, 1456

- jordanAlgebra?
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- KAFfile, [1413](#)
 - ?.?, [1414](#)
 - ?=?, [1414](#)
 - ?~=?, [1414](#)
 - #?, [1414](#)
 - any?, [1414](#)
 - bag, [1414](#)
 - close, [1414](#)
 - coerce, [1414](#)
 - construct, [1414](#)
 - convert, [1414](#)
 - copy, [1414](#)
 - count, [1414](#)
 - dictionary, [1414](#)
 - elt, [1414](#)
 - empty, [1414](#)
 - empty?, [1414](#)
 - entries, [1414](#)
 - entry?, [1414](#)
 - eq?, [1414](#)
 - eval, [1414](#)
 - every?, [1414](#)
 - extract, [1414](#)
 - fill, [1414](#)
 - find, [1414](#)
 - first, [1414](#)
 - hash, [1414](#)
 - index?, [1414](#)
 - indices, [1414](#)
 - insert, [1414](#)
 - inspect, [1414](#)
 - iomode, [1414](#)
 - key?, [1414](#)
 - keys, [1414](#)
 - latex, [1414](#)
 - less?, [1414](#)
 - map, [1414](#)
 - maxIndex, [1414](#)
 - member?, [1414](#)
 - members, [1414](#)
 - minIndex, [1414](#)
 - more?, [1414](#)
 - name, [1414](#)
 - open, [1414](#)
 - pack, [1414](#)
 - parts, [1414](#)
 - qelt, [1414](#)
 - qsetelt, [1414](#)
 - read, [1414](#)
 - reduce, [1414](#)
 - remove, [1414](#)
 - removeDuplicates, [1414](#)
 - reopen, [1414](#)
 - sample, [1414](#)
 - search, [1414](#)
 - select, [1414](#)
 - setelt, [1414](#)
 - size?, [1414](#)
 - swap, [1414](#)
 - table, [1414](#)
 - write, [1414](#)
- karatsubaDivide
 - MODMON, [1632](#)
 - MYUP, [1695](#)
 - NSUP, [1729](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - UP, [2828](#)
- KERNEL, [1404](#)
 - ?<?, [1404](#)
 - ?<=?, [1404](#)
 - ?>?, [1404](#)
 - ?>=?, [1404](#)
 - ?=?, [1404](#)
 - ?~=?, [1404](#)
 - argument, [1404](#)
 - coerce, [1404](#)
 - convert, [1404](#)
 - hash, [1404](#)
 - height, [1404](#)
 - is?, [1404](#)
 - kernel, [1404](#)
 - latex, [1404](#)
 - max, [1404](#)
 - min, [1404](#)

- name, [1404](#)
 - operator, [1404](#)
 - position, [1404](#)
 - setPosition, [1404](#)
 - symbolIfCan, [1404](#)
- Kernel, [1404](#)
- kernel
 - AN, [35](#)
 - EXPR, [724](#)
 - FEXPR, [947](#)
 - IAN, [1276](#)
 - KERNEL, [1404](#)
 - MYEXPR, [1688](#)
- kernels
 - AN, [35](#)
 - EXPR, [724](#)
 - FEXPR, [947](#)
 - IAN, [1276](#)
 - MYEXPR, [1688](#)
- key
 - GRIMAGE, [1094](#)
 - VIEW2d, [2770](#)
 - VIEW3D, [2710](#)
- key?
 - ALIST, [235](#)
 - EQTBL, [699](#)
 - GSTBL, [1079](#)
 - HASHTBL, [1120](#)
 - INTABL, [1336](#)
 - KAFIL, [1414](#)
 - LIB, [1429](#)
 - RESULT, [2301](#)
 - ROUTINE, [2331](#)
 - STBL, [2450](#)
 - STRTBL, [2610](#)
 - TABLE, [2664](#)
- KeyedAccessFile, [1413](#)
- keys
 - ALIST, [235](#)
 - EQTBL, [699](#)
 - GSTBL, [1079](#)
 - HASHTBL, [1120](#)
 - INTABL, [1336](#)
 - INTFTBL, [1371](#)
 - KAFIL, [1414](#)
 - LIB, [1429](#)
- ODEIFTBL, [1768](#)
- RESULT, [2301](#)
- ROUTINE, [2331](#)
- STBL, [2450](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- knownInfBasis
 - ALGFF, [28](#)
- kroneckerDelta
 - CARTEN, [356](#)
- LA, [1520](#)
 - , [1520](#)
 - ?<?, [1520](#)
 - ?<=?, [1520](#)
 - ?>?, [1520](#)
 - ?>=?, [1520](#)
 - ?**?, [1520](#)
 - ?*?, [1520](#)
 - ?+?, [1520](#)
 - ?-?, [1520](#)
 - ?/?, [1520](#)
 - ?=?, [1520](#)
 - ?^?, [1520](#)
 - ?~=?, [1520](#)
 - 0, [1520](#)
 - 1, [1520](#)
 - abs, [1520](#)
 - characteristic, [1520](#)
 - coerce, [1520](#)
 - denom, [1520](#)
 - hash, [1520](#)
 - latex, [1520](#)
 - max, [1520](#)
 - min, [1520](#)
 - negative?, [1520](#)
 - numer, [1520](#)
 - one?, [1520](#)
 - positive?, [1520](#)
 - recip, [1520](#)
 - sample, [1520](#)
 - sign, [1520](#)
 - subtractIfCan, [1520](#)
 - zero?, [1520](#)
- label
 - OUTFORM, [1868](#)

- lagrange
 - UFPS, [2789](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- lambda
 - INFORM, [1343](#)
- lambert
 - UFPS, [2789](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- last
 - ALIST, [235](#)
 - DLIST, [460](#)
 - GTSET, [1083](#)
 - ILIST, [1231](#)
 - LIST, [1504](#)
 - NSDPS, [1703](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - STREAM, [2582](#)
 - WUTSET, [2959](#)
- lastNonNul
 - PROJPL, [2117](#)
 - PROJPLPS, [2119](#)
 - PROJSP, [2122](#)
- lastNonNull
 - PROJPL, [2117](#)
 - PROJPLPS, [2119](#)
 - PROJSP, [2122](#)
- lastSubResultant
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
- lastSubResultantElseSplit
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
- latex
 - AFFPLPS, [7](#)
 - AFFSP, [9](#)
 - ALGFF, [28](#)
 - ALGSC, [15](#)
 - ALIST, [235](#)
 - AN, [35](#)
 - ANON, [38](#)
 - ANTISYM, [40](#)
 - ANY, [50](#)
 - ARRAY1, [1774](#)
 - ARRAY2, [2763](#)
 - ASTACK, [65](#)
 - ATTRBUT, [239](#)
 - AUTOMOR, [245](#)
 - BBTREE, [253](#)
 - BFUNCT, [265](#)
 - BINARY, [292](#)
 - BINFILE, [295](#)
 - BITS, [315](#)
 - BLHN, [317](#)
 - BLQT, [319](#)
 - BOOLEAN, [322](#)
 - BOP, [274](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - BSD, [286](#)
 - BSTREE, [302](#)
 - BTOURN, [306](#)
 - BTREE, [310](#)
 - CARD, [332](#)
 - CARTEN, [356](#)
 - CCLASS, [382](#)
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)
 - CHAR, [373](#)
 - CLIF, [402](#)
 - COLOR, [407](#)
 - COMM, [411](#)
 - COMPLEX, [419](#)
 - COMPPROP, [2624](#)
 - CONTFRAC, [445](#)
 - D01AJFA, [614](#)
 - D01AKFA, [617](#)
 - D01ALFA, [620](#)
 - D01AMFA, [623](#)
 - D01APFA, [629](#), [633](#)
 - D01ASFA, [636](#)
 - D01FCFA, [639](#)
 - D01GBFA, [643](#)
 - D01TRNS, [646](#)
 - D02BBFA, [651](#)

- D02BHFA, [654](#)
- D02CJFA, [658](#)
- D02EJFA, [661](#)
- D03EEFA, [665](#)
- D03FAFA, [668](#)
- D10ANFA, [626](#)
- DBASE, [454](#)
- DECIMAL, [465](#)
- DEQUEUE, [512](#)
- DERHAM, [530](#)
- DFLOAT, [587](#)
- DFMAT, [599](#)
- DFVEC, [605](#)
- DHMATRIX, [491](#)
- DIRPROD, [547](#)
- DIRRING, [564](#)
- DIV, [575](#)
- DLIST, [460](#)
- DMP, [572](#)
- DPMM, [553](#)
- DPMO, [557](#)
- DROPT, [608](#)
- DSMP, [542](#)
- DSTREE, [535](#)
- E04DGFA, [747](#)
- E04FDFA, [750](#)
- E04GCFA, [754](#)
- E04JAFA, [758](#)
- E04MBFA, [762](#)
- E04NAFA, [765](#)
- E04UCFA, [769](#)
- EAB, [743](#)
- EMR, [702](#)
- EQ, [691](#)
- EQTBL, [699](#)
- EXIT, [707](#)
- EXPEXPAN, [712](#)
- EXPR, [724](#)
- EXPUPXS, [740](#)
- FAGROUP, [1004](#)
- FAMONOID, [1007](#)
- FARRAY, [886](#)
- FC, [931](#)
- FCOMP, [975](#)
- FDIV, [813](#)
- FEXPR, [947](#)
- FF, [820](#)
- FFCG, [825](#)
- FFCGP, [835](#)
- FFCGX, [830](#)
- FFNB, [860](#)
- FFNBP, [872](#)
- FFNBX, [866](#)
- FFP, [851](#)
- FFX, [846](#)
- FGROUP, [1010](#)
- FILE, [802](#)
- FLOAT, [908](#)
- FM, [1013](#)
- FM1, [1017](#)
- FMONOID, [1021](#)
- FNAME, [810](#)
- FNLA, [1027](#)
- FORMULA, [2346](#)
- FPARFRAC, [1040](#)
- FR, [786](#)
- FRAC, [986](#)
- FRIDEAL, [995](#)
- FRMOD, [1000](#)
- FSERIES, [978](#)
- FT, [971](#)
- FTEM, [967](#)
- FUNCTION, [1045](#)
- GCNAALG, [1065](#)
- GDMP, [1052](#)
- GMODPOL, [1059](#)
- GOPT, [1104](#)
- GOPT0, [1110](#)
- GPOLSET, [1074](#)
- GRIMAGE, [1094](#)
- GSERIES, [1090](#)
- GSTBL, [1079](#)
- GTSET, [1083](#)
- HACKPI, [1978](#)
- HASHTBL, [1120](#)
- HDMP, [1180](#)
- HDP, [1173](#)
- HEAP, [1134](#)
- HELLFDIV, [1183](#)
- HEXADEC, [1143](#)
- HTMLFORM, [1152](#)
- IAN, [1276](#)

- IARRAY1, [1244](#)
- IARRAY2, [1256](#)
- IBITS, [1199](#)
- IC, [1191](#)
- ICARD, [1193](#)
- IDEAL, [2081](#)
- IDPAG, [1202](#)
- IDPAM, [1206](#)
- IDPO, [1209](#)
- IDPOAM, [1212](#)
- IDPOAMS, [1215](#)
- IFAMON, [1287](#)
- IFARRAY, [1222](#)
- IFF, [1284](#)
- IIARRAY2, [1290](#)
- ILIST, [1231](#)
- IMATRIX, [1239](#)
- INDE, [1218](#)
- INFCLSPS, [1271](#)
- INFCLSPT, [1265](#)
- INFORM, [1343](#)
- INT, [1362](#)
- INTABL, [1336](#)
- INTRVL, [1384](#)
- IPADIC, [1294](#)
- IPF, [1303](#)
- IR, [1375](#)
- ISTRING, [1249](#)
- ISUPS, [1311](#)
- ITAYLOR, [1338](#)
- IVECTOR, [1260](#)
- JORDAN, [223](#)
- KAFILE, [1414](#)
- KERNEL, [1404](#)
- LA, [1520](#)
- LAUPOL, [1422](#)
- LEXP, [1435](#)
- LIB, [1429](#)
- LIE, [229](#)
- LIST, [1504](#)
- LMDICT, [1515](#)
- LMOPS, [1509](#)
- LO, [1523](#)
- LODO, [1469](#)
- LODO1, [1479](#)
- LODO2, [1491](#)
- LPOLY, [1447](#)
- LSQM, [1456](#)
- LWORD, [1532](#)
- M3D, [2703](#)
- MAGMA, [1565](#)
- MATRIX, [1623](#)
- MCMPLEX, [1543](#)
- MFLOAT, [1548](#)
- MINT, [1557](#)
- MKCHSET, [1570](#)
- MMLFORM, [1603](#)
- MODFIELD, [1638](#)
- MODMON, [1632](#)
- MODMONOM, [1644](#)
- MODOP, [1647](#), [1804](#)
- MODRING, [1641](#)
- MOEBIUS, [1654](#)
- MPOLY, [1682](#)
- MRING, [1659](#)
- MSET, [1670](#)
- MYEXPR, [1688](#)
- MYUP, [1695](#)
- NIPROB, [1746](#)
- NNI, [1739](#)
- NONE, [1737](#)
- NOTTING, [1744](#)
- NSDPS, [1703](#)
- NSMP, [1714](#)
- NSUP, [1729](#)
- OCT, [1765](#)
- ODEPROB, [1750](#)
- ODP, [1816](#)
- ODPOL, [1852](#)
- ODR, [1858](#)
- ODVAR, [1855](#)
- OFMONOID, [1829](#)
- OMENC, [1789](#)
- OMERR, [1792](#)
- OMERRK, [1794](#)
- OMLO, [1806](#)
- ONECOMP, [1777](#)
- OPTPROB, [1753](#)
- ORDCOMP, [1810](#)
- ORESUP, [2492](#)
- OREUP, [2872](#)
- OSI, [1864](#)

- OUTFORM, 1868
OVAR, 1836
OWP, 1861
PACOFF, 2136
PACRAT, 2146
PADIC, 1881
PADICRAT, 1886
PADICRC, 1891
PALETTE, 1896
PATLRES, 1937
PATRES, 1940
PATTERN, 1928
PBWLB, 2054
PDEPROB, 1756
PENDTREE, 1945
PERM, 1949
PERMGRP, 1960
PF, 2105
PFR, 1914
PI, 2100
PLACES, 2019
PLACESPS, 2021
POINT, 2060
POLY, 2078
PR, 2092
PRIMARR, 2109
PRODUCT, 2113
PROJPL, 2117
PROJPLPS, 2119
PROJSP, 2122
PRITION, 1923
QALGSET, 2157
QFORM, 2154
QUAT, 2167
QUEUE, 2184
RADFF, 2194
RADIX, 2206
RECLOS, 2237
REF, 2249
REGSET, 2286
RESRING, 2296
RESULT, 2301
RGCHAIN, 2255
RMATRIX, 2246
ROIRC, 2310
ROMAN, 2326
ROUTINE, 2331
RULE, 2305
RULECOLD, 2340
RULESET, 2342
SAE, 2400
SAOS, 2418
SD, 2572
SDPOL, 2386
SDVAR, 2389
SEG, 2359
SEGBIND, 2364
SET, 2372
SETMN, 2377
SEX, 2392
SEXOF, 2394
SHDP, 2509
SINT, 2412
SMP, 2423
SMTS, 2441
SPACE3, 2731
SPLNODE, 2512
SPLTREE, 2518
SQMATRIX, 2548
SREGSET, 2535
STACK, 2563
STBL, 2450
STREAM, 2582
STRING, 2607
STRTBL, 2610
SUBSPACE, 2614
SUCH, 2627
SULS, 2457
SUP, 2467
SUPEXPR, 2481
SUPXS, 2487
SUTS, 2497
SYMBOL, 2640
SYMPOLY, 2654
TABLE, 2664
TEX, 2677
TEXTFILE, 2692
TREE, 2741
TS, 2671
TUPLE, 2752
U16MAT, 2906
U16VEC, 2924

- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UNISEG, [2896](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- VARIABLE, [2936](#)
- VECTOR, [2942](#)
- VIEW2d, [2770](#)
- VIEW3D, [2710](#)
- WP, [2949](#)
- WUTSET, [2959](#)
- XDPOLY, [2967](#)
- XPBWPOLYL, [2987](#)
- XPOLY, [2998](#)
- XPR, [3007](#)
- XPOLY, [3013](#)
- ZMOD, [1368](#)
- LAUPOL, [1421](#)
- ?, [1422](#)
- ?**, [1422](#)
- ?*, [1422](#)
- ?+?, [1422](#)
- ?-?, [1422](#)
- ?=?, [1422](#)
- ?^?, [1422](#)
- ?~=?, [1422](#)
- ?quo?, [1422](#)
- ?rem?, [1422](#)
- 0, [1422](#)
- 1, [1422](#)
- associates?, [1422](#)
- characteristic, [1422](#)
- charthRoot, [1422](#)
- coefficient, [1422](#)
- coerce, [1422](#)
- convert, [1422](#)
- D, [1422](#)
- degree, [1422](#)
- differentiate, [1422](#)
- divide, [1422](#)
- euclideanSize, [1422](#)
- expressIdealMember, [1422](#)
- exquo, [1422](#)
- extendedEuclidean, [1422](#)
- gcd, [1422](#)
- gcdPolynomial, [1422](#)
- hash, [1422](#)
- latex, [1422](#)
- lcm, [1422](#)
- leadingCoefficient, [1422](#)
- monomial, [1422](#)
- monomial?, [1422](#)
- multiEuclidean, [1422](#)
- one?, [1422](#)
- order, [1422](#)
- principalIdeal, [1422](#)
- recip, [1422](#)
- reductum, [1422](#)
- retract, [1422](#)
- retractIfCan, [1422](#)
- sample, [1422](#)
- separate, [1422](#)
- sizeLess?, [1422](#)
- subtractIfCan, [1422](#)
- trailingCoefficient, [1422](#)
- unit?, [1422](#)
- unitCanonical, [1422](#)
- unitNormal, [1422](#)
- zero?, [1422](#)
- laurent
 - SULS, [2457](#)
 - SUPXS, [2487](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
- laurentIfCan
 - SUPXS, [2487](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
- LaurentPolynomial, [1421](#)
- laurentRep
 - SUPXS, [2487](#)

- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- LazardQuotient
 - NSMP, [1714](#)
- LazardQuotient2
 - NSMP, [1714](#)
- lazy?
 - NSDPS, [1703](#)
 - STREAM, [2582](#)
- lazyEvaluate
 - NSDPS, [1703](#)
 - STREAM, [2582](#)
- lazyPquo
 - NSMP, [1714](#)
- lazyPrem
 - NSMP, [1714](#)
- lazyPremWithDefault
 - NSMP, [1714](#)
- lazyPseudoDivide
 - NSMP, [1714](#)
 - NSUP, [1729](#)
- lazyPseudoQuotient
 - NSUP, [1729](#)
- lazyPseudoRemainder
 - NSUP, [1729](#)
- lazyResidueClass
 - NSMP, [1714](#)
 - NSUP, [1729](#)
- lcm
 - ALGFF, [28](#)
 - AN, [35](#)
 - BINARY, [292](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EMR, [702](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FLOAT, [908](#)
 - FR, [786](#)
 - FRAC, [986](#)
 - GDMP, [1052](#)
 - GSERIES, [1090](#)
 - HACKPI, [1978](#)
 - HDMP, [1180](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - IFF, [1284](#)
 - INT, [1362](#)
 - INTRVL, [1384](#)
 - IPADIC, [1294](#)
 - IPF, [1303](#)
 - LAUPOL, [1422](#)
 - MCMPPLX, [1543](#)
 - MFLOAT, [1548](#)
 - MINT, [1557](#)
 - MODFIELD, [1638](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYEXPR, [1688](#)
 - MYUP, [1695](#)
 - NSDPS, [1703](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - ODR, [1858](#)
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
 - PADIC, [1881](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - PF, [2105](#)
 - PFR, [1914](#)
 - POLY, [2078](#)
 - RADFF, [2194](#)
 - RADIX, [2206](#)
 - RECLOS, [2237](#)

- ROMAN, [2326](#)
- SAE, [2400](#)
- SDPOL, [2386](#)
- SINT, [2412](#)
- SMP, [2423](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- LE
 - SWITCH, [2629](#)
- leader
 - DSMP, [542](#)
 - ODPOL, [1852](#)
 - SDPOL, [2386](#)
- leadingBasisTerm
 - ANTISYM, [40](#)
 - DERHAM, [530](#)
- leadingCoefficient
 - ANTISYM, [40](#)
 - DERHAM, [530](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPUPXS, [740](#)
 - FM, [1013](#)
 - FM1, [1017](#)
 - GDMP, [1052](#)
 - GMODPOL, [1059](#)
 - GSERIES, [1090](#)
 - HDMP, [1180](#)
 - IDPAG, [1202](#)
 - IDPAM, [1206](#)
 - IDPO, [1209](#)
 - IDPOAM, [1212](#)
 - IDPOAMS, [1215](#)
 - INDE, [1218](#)
 - ISUPS, [1311](#)
 - LAUPOL, [1422](#)
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - LPOLY, [1447](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MRING, [1659](#)
 - MYUP, [1695](#)
 - NSDPS, [1703](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - OMLO, [1806](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
 - POLY, [2078](#)
 - PR, [2092](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - SYMPOLY, [2654](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UP, [2828](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UPXSING, [2852](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
 - XDPOLY, [2967](#)
 - XPBWPOLYL, [2987](#)
 - XPR, [3007](#)
- leadingExponent
 - GMODPOL, [1059](#)
- leadingIdeal
 - IDEAL, [2081](#)
- leadingIndex
 - GMODPOL, [1059](#)
- leadingMonomial
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPUPXS, [740](#)

- FM1, [1017](#)
- GDMP, [1052](#)
- GMODPOL, [1059](#)
- GSERIES, [1090](#)
- HDMP, [1180](#)
- ISUPS, [1311](#)
- LPOLY, [1447](#)
- MODMON, [1632](#)
- MPOLY, [1682](#)
- MRING, [1659](#)
- MYUP, [1695](#)
- NSDPS, [1703](#)
- NSMP, [1714](#)
- NSUP, [1729](#)
- ODPOL, [1852](#)
- POLY, [2078](#)
- PR, [2092](#)
- SDPOL, [2386](#)
- SMP, [2423](#)
- SMTS, [2441](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXP, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- SYMPOLY, [2654](#)
- TS, [2671](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- XDPOLY, [2967](#)
- XPBWPLYL, [2987](#)
- XPR, [3007](#)
- leadingSupport
 - FM, [1013](#)
 - IDPAG, [1202](#)
 - IDPAM, [1206](#)
 - IDPO, [1209](#)
 - IDPOAM, [1212](#)
 - IDPOAMS, [1215](#)
- INDE, [1218](#)
- leadingTerm
 - FM1, [1017](#)
 - LPOLY, [1447](#)
 - XDPOLY, [2967](#)
 - XPBWPLYL, [2987](#)
 - XPR, [3007](#)
- leaf?
 - ALIST, [235](#)
 - BBTREE, [253](#)
 - BSTREE, [302](#)
 - BTOURN, [306](#)
 - BTREE, [310](#)
 - DLIST, [460](#)
 - DSTREE, [535](#)
 - ILIST, [1231](#)
 - LIST, [1504](#)
 - NSDPS, [1703](#)
 - PENDTREE, [1945](#)
 - PLACES, [2019](#)
 - PLACESPS, [2021](#)
 - SPLTREE, [2518](#)
 - STREAM, [2582](#)
 - SUBSPACE, [2614](#)
 - TREE, [2741](#)
- leastMonomial
 - NSMP, [1714](#)
- leaves
 - ALIST, [235](#)
 - BBTREE, [253](#)
 - BSTREE, [302](#)
 - BTOURN, [306](#)
 - BTREE, [310](#)
 - DLIST, [460](#)
 - DSTREE, [535](#)
 - ILIST, [1231](#)
 - LIST, [1504](#)
 - NSDPS, [1703](#)
 - PENDTREE, [1945](#)
 - SPLTREE, [2518](#)
 - STREAM, [2582](#)
 - TREE, [2741](#)
- left
 - BBTREE, [253](#)
 - BSTREE, [302](#)
 - BTOURN, [306](#)

- BTREE, [310](#)
- LWORD, [1532](#)
- MAGMA, [1565](#)
- OUTFORM, [1868](#)
- PENDTREE, [1945](#)
- ROIRC, [2310](#)
- leftAlternative?
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- leftCharacteristicPolynomial
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- leftDiscriminant
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- leftDivide
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
- leftExactQuotient
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
- leftExtendedGcd
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
- leftGcd
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
- ORESUP, [2492](#)
- OREUP, [2872](#)
- leftLcm
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
- leftMinimalPolynomial
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- leftMult
 - LMOPS, [1509](#)
- leftNorm
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- leftOne
 - EQ, [691](#)
- leftPower
 - ALGSC, [15](#)
 - FNLA, [1027](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- leftQuotient
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
- leftRankPolynomial
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- leftRecip
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)

- JORDAN, 223
- LIE, 229
- LSQM, 1456
- leftRegularRepresentation
 - ALGSC, 15
 - GCNAALG, 1065
 - JORDAN, 223
 - LIE, 229
 - LSQM, 1456
- leftRemainder
 - LODO, 1469
 - LODO1, 1479
 - LODO2, 1491
 - ORESUP, 2492
 - OREUP, 2872
- leftTrace
 - ALGSC, 15
 - GCNAALG, 1065
 - JORDAN, 223
 - LIE, 229
 - LSQM, 1456
- leftTraceMatrix
 - ALGSC, 15
 - GCNAALG, 1065
 - JORDAN, 223
 - LIE, 229
 - LSQM, 1456
- leftTrim
 - ISTRING, 1249
 - STRING, 2607
- leftUnit
 - ALGSC, 15
 - GCNAALG, 1065
 - JORDAN, 223
 - LIE, 229
 - LSQM, 1456
- leftUnits
 - ALGSC, 15
 - GCNAALG, 1065
 - JORDAN, 223
 - LIE, 229
 - LSQM, 1456
- leftZero
 - EQ, 691
- length
 - CDFVEC, 432
- DEQUEUE, 512
- DFVEC, 605
- INT, 1362
- IVECTOR, 1260
- LWORD, 1532
- MAGMA, 1565
- MINT, 1557
- OFMONOID, 1829
- PBWLb, 2054
- POINT, 2060
- QUEUE, 2184
- ROMAN, 2326
- SINT, 2412
- TUPLE, 2752
- VECTOR, 2942
- less?
 - ALIST, 235
 - ARRAY1, 1774
 - ARRAY2, 2763
 - ASTACK, 65
 - BBTREE, 253
 - BITS, 315
 - BSTREE, 302
 - BTOURN, 306
 - BTREE, 310
 - CCLASS, 382
 - CDFMAT, 426
 - CDFVEC, 432
 - DEQUEUE, 512
 - DFMAT, 599
 - DFVEC, 605
 - DHMATRIX, 491
 - DIRPROD, 547
 - DLIST, 460
 - DPMM, 553
 - DPMO, 557
 - DSTREE, 535
 - EQTBL, 699
 - FARRAY, 886
 - GPOLSET, 1074
 - GSTBL, 1079
 - GTSET, 1083
 - HASHTBL, 1120
 - HDP, 1173
 - HEAP, 1134
 - IARRAY1, 1244

- IARRAY2, [1256](#)
- IBITS, [1199](#)
- IFARRAY, [1222](#)
- IIARRAY2, [1290](#)
- ILIST, [1231](#)
- IMATRIX, [1239](#)
- INTABL, [1336](#)
- ISTRING, [1249](#)
- IVECTOR, [1260](#)
- KAFILE, [1414](#)
- LIB, [1429](#)
- LIST, [1504](#)
- LMDICT, [1515](#)
- LSQM, [1456](#)
- M3D, [2703](#)
- MATRIX, [1623](#)
- MSET, [1670](#)
- NSDPS, [1703](#)
- ODP, [1816](#)
- PENDTREE, [1945](#)
- POINT, [2060](#)
- PRIMARR, [2109](#)
- QUEUE, [2184](#)
- REGSET, [2286](#)
- RESULT, [2301](#)
- RGCHAIN, [2255](#)
- RMATRIX, [2246](#)
- ROUTINE, [2331](#)
- SET, [2372](#)
- SHDP, [2509](#)
- SPLTREE, [2518](#)
- SQMATRIX, [2548](#)
- SREGSET, [2535](#)
- STACK, [2563](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRIBL, [2610](#)
- TABLE, [2664](#)
- TREE, [2741](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- WUTSET, [2959](#)
- level
 - SUBSPACE, [2614](#)
- leviCivitaSymbol
 - CARTEN, [356](#)
- lexico
 - LWORD, [1532](#)
 - MAGMA, [1565](#)
 - OFMONOID, [1829](#)
- LEXP, [1435](#)
 - ?**?, [1435](#)
 - ?*?, [1435](#)
 - ?/? , [1435](#)
 - ?=?, [1435](#)
 - ?^?, [1435](#)
 - ?~=?, [1435](#)
 - 1, [1435](#)
 - coerce, [1435](#)
 - commutator, [1435](#)
 - conjugate, [1435](#)
 - exp, [1435](#)
 - hash, [1435](#)
 - identification, [1435](#)
 - inv, [1435](#)
 - latex, [1435](#)
 - listOfTerms, [1435](#)
 - log, [1435](#)
 - LyndonBasis, [1435](#)
 - LyndonCoordinates, [1435](#)
 - mirror, [1435](#)
 - one?, [1435](#)
 - recip, [1435](#)
 - sample, [1435](#)
 - varList, [1435](#)
- lhs
 - EQ, [691](#)
 - RULE, [2305](#)
 - SUCH, [2627](#)
- li
 - EXPR, [724](#)
- LIB, [1428](#)
 - ?.?, [1429](#)
 - ?=?, [1429](#)
 - ?~=?, [1429](#)
 - #?, [1429](#)

any?, 1429
 bag, 1429
 close, 1429
 coerce, 1429
 construct, 1429
 convert, 1429
 copy, 1429
 count, 1429
 dictionary, 1429
 elt, 1429
 empty, 1429
 empty?, 1429
 entries, 1429
 entry?, 1429
 eq?, 1429
 eval, 1429
 every?, 1429
 extract, 1429
 fill, 1429
 find, 1429
 first, 1429
 hash, 1429
 index?, 1429
 indices, 1429
 insert, 1429
 inspect, 1429
 key?, 1429
 keys, 1429
 latex, 1429
 less?, 1429
 library, 1429
 map, 1429
 maxIndex, 1429
 member?, 1429
 members, 1429
 minIndex, 1429
 more?, 1429
 pack, 1429
 parts, 1429
 qelt, 1429
 qsetelt, 1429
 reduce, 1429
 remove, 1429
 removeDuplicates, 1429
 sample, 1429
 search, 1429

select, 1429
 setelt, 1429
 size?, 1429
 swap, 1429
 table, 1429

Library, 1428

library

LIB, 1429

operations

coerce , 1575, 1576

coerce(expr : OutputForm) : String
 , 1574

coerce: E -> S , 1576

coerceL: E -> S , 1577

coerceS: E -> S , 1576

display , 1576

display L , 1581

display(s : String) : Void , 1574

display: S -> Void , 1577

displayF: S -> Void , 1577

displayS , 1581

expres , 1577

expres: E -> S , 1577

precondition , 1577

LIE, 228

-?, 229

?**?, 229

?*?, 229

?+?, 229

?-?, 229

?., 229

?=?, 229

?~=?, 229

0, 229

alternative?, 229

antiAssociative?, 229

antiCommutative?, 229

antiCommutator, 229

apply, 229

associative?, 229

associator, 229

associatorDependence, 229

basis, 229

coerce, 229

commutative?, 229

commutator, 229

- conditionsForIdempotents, 229
- convert, 229
- coordinates, 229
- flexible?, 229
- hash, 229
- jacobiIdentity?, 229
- jordanAdmissible?, 229
- jordanAlgebra?, 229
- latex, 229
- leftAlternative?, 229
- leftCharacteristicPolynomial, 229
- leftDiscriminant, 229
- leftMinimalPolynomial, 229
- leftNorm, 229
- leftPower, 229
- leftRankPolynomial, 229
- leftRecip, 229
- leftRegularRepresentation, 229
- leftTrace, 229
- leftTraceMatrix, 229
- leftUnit, 229
- leftUnits, 229
- lieAdmissible?, 229
- lieAlgebra?, 229
- noncommutativeJordanAlgebra?, 229
- plenaryPower, 229
- powerAssociative?, 229
- rank, 229
- recip, 229
- represents, 229
- rightAlternative?, 229
- rightCharacteristicPolynomial, 229
- rightDiscriminant, 229
- rightMinimalPolynomial, 229
- rightNorm, 229
- rightPower, 229
- rightRankPolynomial, 229
- rightRecip, 229
- rightRegularRepresentation, 229
- rightTrace, 229
- rightTraceMatrix, 229
- rightUnit, 229
- rightUnits, 229
- sample, 229
- someBasis, 229
- structuralConstants, 229
- subtractIfCan, 229
- unit, 229
- zero?, 229
- lieAdmissible?
 - ALGSC, 15
 - GCNAALG, 1065
 - JORDAN, 223
 - LIE, 229
 - LSQM, 1456
- lieAlgebra?
 - ALGSC, 15
 - GCNAALG, 1065
 - JORDAN, 223
 - LIE, 229
 - LSQM, 1456
- LieExponentials, 1435
- LiePoly
 - LPOLY, 1447
- LiePolyIfCan
 - LPOLY, 1447
 - XPBWPOLYL, 2987
- LiePolynomial, 1446
- LieSquareMatrix, 1455
- lift
 - ALGFF, 28
 - COMPLEX, 419
 - MCMPLX, 1543
 - MODMON, 1632
 - PACOFF, 2136
 - PACRAT, 2146
 - RADFF, 2194
 - RESRING, 2296
 - SAE, 2400
- light
 - PALETTE, 1896
- lighting
 - VIEW3D, 2710
- limitPlus
 - EXPEXPAN, 712
 - UPXSSING, 2852
- linearAssociatedExp
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860

- FFNBP, [872](#)
- FFNBX, [866](#)
- FFP, [851](#)
- FFX, [846](#)
- IFF, [1284](#)
- IPF, [1303](#)
- PF, [2105](#)
- linearAssociatedLog
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - PF, [2105](#)
- linearAssociatedOrder
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - PF, [2105](#)
- LinearOrdinaryDifferentialOperator, [1469](#)
- LinearOrdinaryDifferentialOperator1, [1479](#)
- LinearOrdinaryDifferentialOperator2, [1491](#)
- LIST, [1504](#)
 - ?<?, [1504](#)
 - ?<=?, [1504](#)
 - ?>?, [1504](#)
 - ?>=?, [1504](#)
 - ?.?, [1504](#)
 - ?..first, [1504](#)
 - ?..last, [1504](#)
 - ?..rest, [1504](#)
 - ?..value, [1504](#)
 - ?=?, [1504](#)
 - ?~=?, [1504](#)
 - #?, [1504](#)
 - any?, [1504](#)
 - append, [1504](#)
 - child?, [1504](#)
 - children, [1504](#)
 - coerce, [1504](#)
 - concat, [1504](#)
 - cons, [1504](#)
 - construct, [1504](#)
 - convert, [1504](#)
 - copy, [1504](#)
 - copyInto, [1504](#)
 - count, [1504](#)
 - cycleEntry, [1504](#)
 - cycleLength, [1504](#)
 - cycleSplit, [1504](#)
 - cycleTail, [1504](#)
 - cyclic?, [1504](#)
 - delete, [1504](#)
 - distance, [1504](#)
 - elt, [1504](#)
 - empty, [1504](#)
 - empty?, [1504](#)
 - entries, [1504](#)
 - entry?, [1504](#)
 - eq?, [1504](#)
 - eval, [1504](#)
 - every?, [1504](#)
 - explicitlyFinite?, [1504](#)
 - fill, [1504](#)
 - find, [1504](#)
 - first, [1504](#)
 - hash, [1504](#)
 - index?, [1504](#)
 - indices, [1504](#)
 - insert, [1504](#)
 - last, [1504](#)
 - latex, [1504](#)
 - leaf?, [1504](#)
 - leaves, [1504](#)
 - less?, [1504](#)
 - list, [1504](#)
 - map, [1504](#)
 - max, [1504](#)

- maxIndex, [1504](#)
- member?, [1504](#)
- members, [1504](#)
- merge, [1504](#)
- min, [1504](#)
- minIndex, [1504](#)
- more?, [1504](#)
- new, [1504](#)
- nil, [1504](#)
- node?, [1504](#)
- nodes, [1504](#)
- null, [1504](#)
- OMwrite, [1504](#)
- parts, [1504](#)
- position, [1504](#)
- possiblyInfinite?, [1504](#)
- qelt, [1504](#)
- qsetelt, [1504](#)
- reduce, [1504](#)
- remove, [1504](#)
- removeDuplicates, [1504](#)
- rest, [1504](#)
- reverse, [1504](#)
- sample, [1504](#)
- second, [1504](#)
- select, [1504](#)
- setchildren, [1504](#)
- setDifference, [1504](#)
- setelt, [1504](#)
- setfirst, [1504](#)
- setIntersection, [1504](#)
- setlast, [1504](#)
- setrest, [1504](#)
- setUnion, [1504](#)
- setvalue, [1504](#)
- size?, [1504](#)
- sort, [1504](#)
- sorted?, [1504](#)
- split, [1504](#)
- swap, [1504](#)
- tail, [1504](#)
- third, [1504](#)
- value, [1504](#)
- List, [1504](#)
- list
 - AFFPLPS, [7](#)
 - AFFSP, [9](#)
 - ALIST, [235](#)
 - DLIST, [460](#)
 - ILIST, [1231](#)
 - LIST, [1504](#)
 - PROJPL, [2117](#)
 - PROJPLPS, [2119](#)
 - PROJSP, [2122](#)
 - SYMBOL, [2640](#)
- list?
 - INFORM, [1343](#)
 - SEX, [2392](#)
 - SEXOF, [2394](#)
- listBranches
 - ACPLOT, [1992](#)
 - PLOT, [2029](#)
 - PLOT3D, [2042](#)
- listLoops
 - TUBE, [2749](#)
- ListMonoidOps, [1509](#)
- ListMultiDictionary, [1515](#)
- listOfLists
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - RMATRIX, [2246](#)
 - SQMATRIX, [2548](#)
 - TABLEAU, [2666](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- listOfMonoms
 - LMOPS, [1509](#)
- listOfTerms
 - FM1, [1017](#)
 - LEXP, [1435](#)
 - LPOLY, [1447](#)
 - PBWLBS, [2054](#)
 - XDPOLY, [2967](#)
 - XPBWPOLYL, [2987](#)
 - XPR, [3007](#)
- listRepresentation
 - PERM, [1949](#)

- lists
 - PATLRES, [1937](#)
- listSD
 - SD, [2572](#)
- llip
 - SPACE3, [2731](#)
- llp
 - SPACE3, [2731](#)
- llprop
 - SPACE3, [2731](#)
- LMDICT, [1515](#)
 - ?=?, [1515](#)
 - ?~=?, [1515](#)
 - #?, [1515](#)
 - any?, [1515](#)
 - bag, [1515](#)
 - coerce, [1515](#)
 - construct, [1515](#)
 - convert, [1515](#)
 - copy, [1515](#)
 - count, [1515](#)
 - dictionary, [1515](#)
 - duplicates, [1515](#)
 - duplicates?, [1515](#)
 - empty, [1515](#)
 - empty?, [1515](#)
 - eq?, [1515](#)
 - eval, [1515](#)
 - every?, [1515](#)
 - extract, [1515](#)
 - find, [1515](#)
 - hash, [1515](#)
 - insert, [1515](#)
 - inspect, [1515](#)
 - latex, [1515](#)
 - less?, [1515](#)
 - map, [1515](#)
 - member?, [1515](#)
 - members, [1515](#)
 - more?, [1515](#)
 - parts, [1515](#)
 - reduce, [1515](#)
 - remove, [1515](#)
 - removeDuplicates, [1515](#)
 - sample, [1515](#)
 - select, [1515](#)
 - size?, [1515](#)
 - substitute, [1515](#)
- LMOPS, [1509](#)
 - ?=?, [1509](#)
 - ?~=?, [1509](#)
 - coerce, [1509](#)
 - commutativeEquality, [1509](#)
 - hash, [1509](#)
 - latex, [1509](#)
 - leftMult, [1509](#)
 - listOfMonoms, [1509](#)
 - makeMulti, [1509](#)
 - makeTerm, [1509](#)
 - makeUnit, [1509](#)
 - mapExpon, [1509](#)
 - mapGen, [1509](#)
 - nthExpon, [1509](#)
 - nthFactor, [1509](#)
 - outputForm, [1509](#)
 - plus, [1509](#)
 - retract, [1509](#)
 - retractIfCan, [1509](#)
 - reverse, [1509](#)
 - rightMult, [1509](#)
 - size, [1509](#)
- LO, [1522](#)
 - , [1523](#)
 - ?<?, [1523](#)
 - ?<=?, [1523](#)
 - ?>?, [1523](#)
 - ?>=?, [1523](#)
 - ?*?, [1523](#)
 - ?+?, [1523](#)
 - ?-?, [1523](#)
 - ?/? , [1523](#)
 - ?=?, [1523](#)
 - ?~=?, [1523](#)
 - 0, [1523](#)
 - coerce, [1523](#)
 - denom, [1523](#)
 - hash, [1523](#)
 - latex, [1523](#)
 - max, [1523](#)
 - min, [1523](#)
 - numer, [1523](#)
 - sample, [1523](#)

- subtractIfCan, 1523
 - zero?, 1523
 - lo
 - SEG, 2359
 - UNISEG, 2896
 - LocalAlgebra, 1520
 - Localize, 1522
 - localParam
 - PLACES, 2019
 - PLACESPS, 2021
 - localParamV
 - IC, 1191
 - INFCLSPS, 1271
 - INFCLSPT, 1265
 - localPointV
 - IC, 1191
 - INFCLSPS, 1271
 - INFCLSPT, 1265
 - LODO, 1469
 - .?, 1469
 - ?**?, 1469
 - ?*?, 1469
 - ?+?, 1469
 - ?-?, 1469
 - ?..?, 1469
 - ?=?, 1469
 - ?^?, 1469
 - ?~=?, 1469
 - 0, 1469
 - 1, 1469
 - adjoint, 1469
 - apply, 1469
 - characteristic, 1469
 - coefficient, 1469
 - coefficients, 1469
 - coerce, 1469
 - content, 1469
 - D, 1469
 - degree, 1469
 - directSum, 1469
 - exquo, 1469
 - hash, 1469
 - latex, 1469
 - leadingCoefficient, 1469
 - leftDivide, 1469
 - leftExactQuotient, 1469
 - leftExtendedGcd, 1469
 - leftGcd, 1469
 - leftLcm, 1469
 - leftQuotient, 1469
 - leftRemainder, 1469
 - minimumDegree, 1469
 - monicLeftDivide, 1469
 - monicRightDivide, 1469
 - monomial, 1469
 - one?, 1469
 - primitivePart, 1469
 - recip, 1469
 - reductum, 1469
 - retract, 1469
 - retractIfCan, 1469
 - rightDivide, 1469
 - rightExactQuotient, 1469
 - rightExtendedGcd, 1469
 - rightGcd, 1469
 - rightLcm, 1469
 - rightQuotient, 1469
 - rightRemainder, 1469
 - sample, 1469
 - subtractIfCan, 1469
 - symmetricPower, 1469
 - symmetricProduct, 1469
 - symmetricSquare, 1469
 - zero?, 1469
- LODO1, 1479
 - .?, 1479
 - ?**?, 1479
 - ?*?, 1479
 - ?+?, 1479
 - ?-?, 1479
 - ?..?, 1479
 - ?=?, 1479
 - ?^?, 1479
 - ?~=?, 1479
 - 0, 1479
 - 1, 1479
 - adjoint, 1479
 - apply, 1479
 - characteristic, 1479
 - coefficient, 1479
 - coefficients, 1479
 - coerce, 1479

- content, [1479](#)
- D, [1479](#)
- degree, [1479](#)
- directSum, [1479](#)
- exquo, [1479](#)
- hash, [1479](#)
- latex, [1479](#)
- leadingCoefficient, [1479](#)
- leftDivide, [1479](#)
- leftExactQuotient, [1479](#)
- leftExtendedGcd, [1479](#)
- leftGcd, [1479](#)
- leftLcm, [1479](#)
- leftQuotient, [1479](#)
- leftRemainder, [1479](#)
- minimumDegree, [1479](#)
- monicLeftDivide, [1479](#)
- monicRightDivide, [1479](#)
- monomial, [1479](#)
- one?, [1479](#)
- primitivePart, [1479](#)
- recip, [1479](#)
- reductum, [1479](#)
- retract, [1479](#)
- retractIfCan, [1479](#)
- rightDivide, [1479](#)
- rightExactQuotient, [1479](#)
- rightExtendedGcd, [1479](#)
- rightGcd, [1479](#)
- rightLcm, [1479](#)
- rightQuotient, [1479](#)
- rightRemainder, [1479](#)
- sample, [1479](#)
- subtractIfCan, [1479](#)
- symmetricPower, [1479](#)
- symmetricProduct, [1479](#)
- symmetricSquare, [1479](#)
- zero?, [1479](#)
- LODO2, [1491](#)
- ?, [1491](#)
- ?**, [1491](#)
- *?*, [1491](#)
- ?+?, [1491](#)
- ?-?, [1491](#)
- ?.?, [1491](#)
- ?=?, [1491](#)
- ?^?, [1491](#)
- ?~=?, [1491](#)
- 0, [1491](#)
- 1, [1491](#)
- adjoint, [1491](#)
- apply, [1491](#)
- characteristic, [1491](#)
- coefficient, [1491](#)
- coefficients, [1491](#)
- coerce, [1491](#)
- content, [1491](#)
- D, [1491](#)
- degree, [1491](#)
- directSum, [1491](#)
- exquo, [1491](#)
- hash, [1491](#)
- latex, [1491](#)
- leadingCoefficient, [1491](#)
- leftDivide, [1491](#)
- leftExactQuotient, [1491](#)
- leftExtendedGcd, [1491](#)
- leftGcd, [1491](#)
- leftLcm, [1491](#)
- leftQuotient, [1491](#)
- leftRemainder, [1491](#)
- minimumDegree, [1491](#)
- monicLeftDivide, [1491](#)
- monicRightDivide, [1491](#)
- monomial, [1491](#)
- one?, [1491](#)
- primitivePart, [1491](#)
- recip, [1491](#)
- reductum, [1491](#)
- retract, [1491](#)
- retractIfCan, [1491](#)
- rightDivide, [1491](#)
- rightExactQuotient, [1491](#)
- rightExtendedGcd, [1491](#)
- rightGcd, [1491](#)
- rightLcm, [1491](#)
- rightQuotient, [1491](#)
- rightRemainder, [1491](#)
- sample, [1491](#)
- subtractIfCan, [1491](#)
- symmetricPower, [1491](#)
- symmetricProduct, [1491](#)

- symmetricSquare, [1491](#)
- zero?, [1491](#)
- log
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FEXPR, [947](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - LEXP, [1435](#)
 - MCMLPX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXP, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
 - XPBWPOLYL, [2987](#)
- log10
 - DFLOAT, [587](#)
 - FEXPR, [947](#)
 - FLOAT, [908](#)
- log2
 - DFLOAT, [587](#)
 - FLOAT, [908](#)
- logical?
 - FST, [962](#)
- logIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- logpart
 - IR, [1375](#)
- lookup
 - ALGFF, [28](#)
 - BOOLEAN, [322](#)
 - CCLASS, [382](#)
 - CHAR, [373](#)
 - COMPLEX, [419](#)
 - DIRPROD, [547](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - HDP, [1173](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - MCMLPX, [1543](#)
 - MODMON, [1632](#)
 - MRING, [1659](#)
 - OCT, [1765](#)
 - ODP, [1816](#)
 - OVAR, [1836](#)
 - PACOFF, [2136](#)
 - PF, [2105](#)
 - PRODUCT, [2113](#)
 - RADFF, [2194](#)
 - SAE, [2400](#)
 - SET, [2372](#)
 - SETMN, [2377](#)
 - SHDP, [2509](#)
 - ZMOD, [1368](#)
- low
 - SEG, [2359](#)
 - UNISEG, [2896](#)
- lowerCase
 - CCLASS, [382](#)
 - CHAR, [373](#)
 - ISTRING, [1249](#)
 - STRING, [2607](#)
- lowerCase?
 - CHAR, [373](#)
- lp
 - SPACE3, [2731](#)
- LPOLY, [1446](#)

- ?, [1447](#)
- ?*?, [1447](#)
- ?+?, [1447](#)
- ?-?, [1447](#)
- ?/? , [1447](#)
- ?=?, [1447](#)
- ?~=?, [1447](#)
- 0, [1447](#)
- coef, [1447](#)
- coefficient, [1447](#)
- coefficients, [1447](#)
- coerce, [1447](#)
- construct, [1447](#)
- degree, [1447](#)
- eval, [1447](#)
- hash, [1447](#)
- latex, [1447](#)
- leadingCoefficient, [1447](#)
- leadingMonomial, [1447](#)
- leadingTerm, [1447](#)
- LiePoly, [1447](#)
- LiePolyIfCan, [1447](#)
- listOfTerms, [1447](#)
- lquo, [1447](#)
- map, [1447](#)
- mirror, [1447](#)
- monom, [1447](#)
- monomial?, [1447](#)
- monomials, [1447](#)
- numberOfMonomials, [1447](#)
- reductum, [1447](#)
- retract, [1447](#)
- retractIfCan, [1447](#)
- rquo, [1447](#)
- sample, [1447](#)
- subtractIfCan, [1447](#)
- trunc, [1447](#)
- varList, [1447](#)
- zero?, [1447](#)
- lprop
 - SPACE3, [2731](#)
- lquo
 - FMONOID, [1021](#)
 - LPOLY, [1447](#)
 - OFMONOID, [1829](#)
 - XDPOLY, [2967](#)
 - XPBWPOLYL, [2987](#)
 - XPOLY, [2998](#)
 - XRPOLY, [3013](#)
- lSpaceBasis
 - FDIV, [813](#)
- LSQM, [1455](#)
 - ?, [1456](#)
 - ?**?, [1456](#)
 - ?*?, [1456](#)
 - ?+?, [1456](#)
 - ?-?, [1456](#)
 - ?..?, [1456](#)
 - ?/? , [1456](#)
 - ?=?, [1456](#)
 - ?^?, [1456](#)
 - ?~=?, [1456](#)
 - #?, [1456](#)
 - 0, [1456](#)
 - 1, [1456](#)
 - alternative?, [1456](#)
 - antiAssociative?, [1456](#)
 - antiCommutator, [1456](#)
 - antisymmetric?, [1456](#)
 - any?, [1456](#)
 - apply, [1456](#)
 - associative?, [1456](#)
 - associator, [1456](#)
 - associatorDependence, [1456](#)
 - basis, [1456](#)
 - characteristic, [1456](#)
 - coerce, [1456](#)
 - column, [1456](#)
 - commutative?, [1456](#)
 - commutator, [1456](#)
 - conditionsForIdempotents, [1456](#)
 - convert, [1456](#)
 - coordinates, [1456](#)
 - copy, [1456](#)
 - count, [1456](#)
 - D, [1456](#)
 - determinant, [1456](#)
 - diagonal, [1456](#)
 - diagonal?, [1456](#)
 - diagonalMatrix, [1456](#)
 - diagonalProduct, [1456](#)
 - differentiate, [1456](#)

elt, 1456
 empty, 1456
 empty?, 1456
 eq?, 1456
 eval, 1456
 every?, 1456
 exquo, 1456
 flexible?, 1456
 hash, 1456
 inverse, 1456
 jacobiIdentity?, 1456
 jordanAdmissible?, 1456
 jordanAlgebra?, 1456
 latex, 1456
 leftAlternative?, 1456
 leftCharacteristicPolynomial, 1456
 leftDiscriminant, 1456
 leftMinimalPolynomial, 1456
 leftNorm, 1456
 leftPower, 1456
 leftRankPolynomial, 1456
 leftRecip, 1456
 leftRegularRepresentation, 1456
 leftTrace, 1456
 leftTraceMatrix, 1456
 leftUnit, 1456
 leftUnits, 1456
 less?, 1456
 lieAdmissible?, 1456
 lieAlgebra?, 1456
 listOfLists, 1456
 map, 1456
 matrix, 1456
 maxColIndex, 1456
 maxRowIndex, 1456
 member?, 1456
 members, 1456
 minColIndex, 1456
 minordet, 1456
 minRowIndex, 1456
 more?, 1456
 ncols, 1456
 noncommutativeJordanAlgebra?, 1456
 nrows, 1456
 nullity, 1456
 nullSpace, 1456

one?, 1456
 parts, 1456
 plenaryPower, 1456
 powerAssociative?, 1456
 qelt, 1456
 rank, 1456
 recip, 1456
 reducedSystem, 1456
 represents, 1456
 retract, 1456
 retractIfCan, 1456
 rightAlternative?, 1456
 rightCharacteristicPolynomial, 1456
 rightDiscriminant, 1456
 rightMinimalPolynomial, 1456
 rightNorm, 1456
 rightPower, 1456
 rightRankPolynomial, 1456
 rightRecip, 1456
 rightRegularRepresentation, 1456
 rightTrace, 1456
 rightTraceMatrix, 1456
 rightUnit, 1456
 rightUnits, 1456
 row, 1456
 rowEchelon, 1456
 sample, 1456
 scalarMatrix, 1456
 size?, 1456
 someBasis, 1456
 square?, 1456
 structuralConstants, 1456
 subtractIfCan, 1456
 symmetric?, 1456
 trace, 1456
 unit, 1456
 zero?, 1456

LT

SWITCH, 2629

LWORD, 1531

?<?, 1532
 ?<=?, 1532
 ?>?, 1532
 ?>=?, 1532
 ?=?, 1532
 ?~=?, 1532

- coerce, [1532](#)
- factor, [1532](#)
- hash, [1532](#)
- latex, [1532](#)
- left, [1532](#)
- length, [1532](#)
- lexico, [1532](#)
- lyndon, [1532](#)
- lyndon?, [1532](#)
- lyndonIfCan, [1532](#)
- LyndonWordsList, [1532](#)
- LyndonWordsList1, [1532](#)
- max, [1532](#)
- min, [1532](#)
- retract, [1532](#)
- retractable?, [1532](#)
- retractIfCan, [1532](#)
- right, [1532](#)
- varList, [1532](#)
- lyndon
 - LWORD, [1532](#)
- lyndon?
 - LWORD, [1532](#)
- LyndonBasis
 - LEXP, [1435](#)
- LyndonCoordinates
 - LEXP, [1435](#)
- lyndonIfCan
 - LWORD, [1532](#)
- LyndonWord, [1531](#)
- LyndonWordsList
 - LWORD, [1532](#)
- LyndonWordsList1
 - LWORD, [1532](#)
- M3D, [2702](#)
 - ?=?, [2703](#)
 - ?~=?, [2703](#)
 - #?, [2703](#)
 - any?, [2703](#)
 - coerce, [2703](#)
 - construct, [2703](#)
 - copy, [2703](#)
 - count, [2703](#)
 - elt, [2703](#)
 - empty, [2703](#)
 - empty?, [2703](#)
 - eq?, [2703](#)
 - eval, [2703](#)
 - every?, [2703](#)
 - hash, [2703](#)
 - identityMatrix, [2703](#)
 - latex, [2703](#)
 - less?, [2703](#)
 - map, [2703](#)
 - matrixConcat3D, [2703](#)
 - matrixDimensions, [2703](#)
 - member?, [2703](#)
 - members, [2703](#)
 - more?, [2703](#)
 - parts, [2703](#)
 - plus, [2703](#)
 - sample, [2703](#)
 - setelt, [2703](#)
 - size?, [2703](#)
 - zeroMatrix, [2703](#)
- MachineComplex, [1542](#)
- MachineFloat, [1547](#)
- machineFraction
 - DFLOAT, [587](#)
- MachineInteger, [1557](#)
- MAGMA, [1565](#)
 - ?<?, [1565](#)
 - ?<=?, [1565](#)
 - ?>?, [1565](#)
 - ?>=?, [1565](#)
 - ?*?, [1565](#)
 - ?=?, [1565](#)
 - ?~=?, [1565](#)
 - coerce, [1565](#)
 - first, [1565](#)
 - hash, [1565](#)
 - latex, [1565](#)
 - left, [1565](#)
 - length, [1565](#)
 - lexico, [1565](#)
 - max, [1565](#)
 - min, [1565](#)
 - mirror, [1565](#)
 - rest, [1565](#)
 - retract, [1565](#)
 - retractable?, [1565](#)

- retractIfCan, 1565
 - right, 1565
 - varList, 1565
- Magma, 1565
- magnitude
 - CDFVEC, 432
 - DFVEC, 605
 - IVECTOR, 1260
 - POINT, 2060
 - VECTOR, 2942
- mainCharacterization
 - RECLOS, 2237
- mainCoefficients
 - NSMP, 1714
- mainContent
 - NSMP, 1714
- mainDefiningPolynomial
 - RECLOS, 2237
- mainForm
 - RECLOS, 2237
- mainKernel
 - AN, 35
 - EXPR, 724
 - FEXPR, 947
 - IAN, 1276
 - MYEXPR, 1688
- mainMonomial
 - NSMP, 1714
- mainMonomials
 - NSMP, 1714
- mainPrimitivePart
 - NSMP, 1714
- mainSquareFreePart
 - NSMP, 1714
- mainValue
 - RECLOS, 2237
- mainVariable
 - DMP, 572
 - GDMP, 1052
 - HDMP, 1180
 - MODMON, 1632
 - MPOLY, 1682
 - MYUP, 1695
 - NSMP, 1714
 - NSUP, 1729
 - ODPOL, 1852
 - POLY, 2078
 - SDPOL, 2386
 - SMP, 2423
 - SUP, 2467
 - SUPEXPR, 2481
 - UP, 2828
- mainVariable?
 - GPOLSET, 1074
 - GTSET, 1083
 - REGSET, 2286
 - RGCHAIN, 2255
 - SREGSET, 2535
 - WUTSET, 2959
- mainVariables
 - GPOLSET, 1074
 - GTSET, 1083
 - REGSET, 2286
 - RGCHAIN, 2255
 - SREGSET, 2535
 - WUTSET, 2959
- MakeCachableSet, 1570
- makeCos
 - Fseries, 978
- makeFR
 - FR, 786
- makeGraphImage
 - GRIMAGE, 1094
- makeMulti
 - LMOPS, 1509
- makeop
 - MODOP, 1647, 1804
- makeprod
 - PRODUCT, 2113
- makeResult
 - PATLRES, 1937
- makeSeries
 - ISUPS, 1311
- makeSin
 - Fseries, 978
- makeSketch
 - ACPLOT, 1992
- makeSUP
 - MODMON, 1632
 - MYUP, 1695
 - NSUP, 1729
 - SUP, 2467

- SUPEXPR, 2481
- UP, 2828
- makeTerm
 - LMOPS, 1509
- makeUnit
 - LMOPS, 1509
- makeVariable
 - DSMP, 542
 - ODVAR, 1855
 - SDPOL, 2386
 - SDVAR, 2389
- makeViewport2D
 - VIEW2d, 2770
- makeViewport3D
 - VIEW3D, 2710
- mantissa
 - DFLOAT, 587
 - FLOAT, 908
 - MFLOAT, 1548
- map
 - ALIST, 235
 - AN, 35
 - ANTISYM, 40
 - ARRAY1, 1774
 - ARRAY2, 2763
 - ASTACK, 65
 - BBTREE, 253
 - BINARY, 292
 - BITS, 315
 - BPADICRT, 263
 - BSTREE, 302
 - BTOURN, 306
 - BTREE, 310
 - CCLASS, 382
 - CDFMAT, 426
 - CDFVEC, 432
 - COMPLEX, 419
 - DECIMAL, 465
 - DEQUEUE, 512
 - DERHAM, 530
 - DFMAT, 599
 - DFVEC, 605
 - DHMATRIX, 491
 - DIRPROD, 547
 - DLIST, 460
 - DMP, 572
 - DPMM, 553
 - DPMO, 557
 - DSMP, 542
 - DSTREE, 535
 - EQ, 691
 - EQTBL, 699
 - EXPEXPAN, 712
 - EXPR, 724
 - EXPUPXS, 740
 - FARRAY, 886
 - FEXPR, 947
 - FM, 1013
 - FM1, 1017
 - FR, 786
 - FRAC, 986
 - GDMP, 1052
 - GPOLSET, 1074
 - GSERIES, 1090
 - GSTBL, 1079
 - GTSET, 1083
 - HASHTBL, 1120
 - HDMP, 1180
 - HDP, 1173
 - HEAP, 1134
 - HEXADEC, 1143
 - IAN, 1276
 - IARRAY1, 1244
 - IARRAY2, 1256
 - IBITS, 1199
 - IDPAG, 1202
 - IDPAM, 1206
 - IDPO, 1209
 - IDPOAM, 1212
 - IDPOAMS, 1215
 - IFARRAY, 1222
 - IARRAY2, 1290
 - ILIST, 1231
 - IMATRIX, 1239
 - INDE, 1218
 - INTABL, 1336
 - ISTRING, 1249
 - ISUPS, 1311
 - ITUPLE, 1262
 - KAFILE, 1414
 - LIB, 1429
 - LIST, 1504

- LMDICT, [1515](#)
- LPOLY, [1447](#)
- LSQM, [1456](#)
- M3D, [2703](#)
- MATRIX, [1623](#)
- MCMPLEX, [1543](#)
- MODMON, [1632](#)
- MPOLY, [1682](#)
- MRING, [1659](#)
- MSET, [1670](#)
- MYEXPR, [1688](#)
- MYUP, [1695](#)
- NSDPS, [1703](#)
- NSMP, [1714](#)
- NSUP, [1729](#)
- OCT, [1765](#)
- ODP, [1816](#)
- ODPOL, [1852](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PENDTREE, [1945](#)
- POINT, [2060](#)
- POLY, [2078](#)
- PR, [2092](#)
- PRIMARR, [2109](#)
- QUAT, [2167](#)
- QUEUE, [2184](#)
- RADIX, [2206](#)
- REGSET, [2286](#)
- RESULT, [2301](#)
- RGCHAIN, [2255](#)
- RMATRIX, [2246](#)
- ROUTINE, [2331](#)
- SDPOL, [2386](#)
- SEG, [2359](#)
- SET, [2372](#)
- SHDP, [2509](#)
- SMP, [2423](#)
- SMTS, [2441](#)
- SPLTREE, [2518](#)
- SQMATRIX, [2548](#)
- SREGSET, [2535](#)
- STACK, [2563](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- SYMPOLY, [2654](#)
- TABLE, [2664](#)
- TREE, [2741](#)
- TS, [2671](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UNISEG, [2896](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- VECTOR, [2942](#)
- WUTSET, [2959](#)
- XDPOLY, [2967](#)
- XPBWPOLYL, [2987](#)
- XPOLY, [2998](#)
- XPR, [3007](#)
- XPOLY, [3013](#)
- mapCoef
 - DIV, [575](#)
 - FAGROUP, [1004](#)
 - FAMONOID, [1007](#)
 - IFAMON, [1287](#)
- mapExpon
 - FGROUP, [1010](#)
 - FMONOID, [1021](#)
 - LMOPS, [1509](#)
- mapExponents
 - DMP, [572](#)
 - DSMP, [542](#)
 - GDMP, [1052](#)

- HDMP, 1180
- MODMON, 1632
- MPOLY, 1682
- MYUP, 1695
- NSMP, 1714
- NSUP, 1729
- ODPOL, 1852
- POLY, 2078
- PR, 2092
- SDPOL, 2386
- SMP, 2423
- SUP, 2467
- SUPEXP, 2481
- SYMPOLY, 2654
- UP, 2828
- UPXSSING, 2852
- mapGen
 - DIV, 575
 - FAGROUP, 1004
 - FAMONOID, 1007
 - FGROUP, 1010
 - FMONOID, 1021
 - IFAMON, 1287
 - LMOPS, 1509
- mask
 - INT, 1362
 - MINT, 1557
 - ROMAN, 2326
 - SINT, 2412
- match
 - ISTRING, 1249
 - STRING, 2607
- match?
 - ISTRING, 1249
 - STRING, 2607
- MathMLForm, 1603
- MATRIX, 1622
 - , 1623
 - ***?, 1623
 - *?*, 1623
 - ?+?, 1623
 - ?-?, 1623
 - ?/?, 1623
 - ?=?, 1623
 - ?~=?, 1623
 - #?, 1623
 - antisymmetric?, 1623
 - any?, 1623
 - coerce, 1623
 - column, 1623
 - convert, 1623
 - copy, 1623
 - count, 1623
 - determinant, 1623
 - diagonal?, 1623
 - diagonalMatrix, 1623
 - elt, 1623
 - empty, 1623
 - empty?, 1623
 - eq?, 1623
 - eval, 1623
 - every?, 1623
 - exquo, 1623
 - fill, 1623
 - hash, 1623
 - horizConcat, 1623
 - inverse, 1623
 - latex, 1623
 - less?, 1623
 - listOfLists, 1623
 - map, 1623
 - matrix, 1623
 - maxColIndex, 1623
 - maxRowIndex, 1623
 - member?, 1623
 - members, 1623
 - minColIndex, 1623
 - minordet, 1623
 - minRowIndex, 1623
 - more?, 1623
 - ncols, 1623
 - new, 1623
 - nrows, 1623
 - nullity, 1623
 - nullSpace, 1623
 - parts, 1623
 - qelt, 1623
 - qsetelt, 1623
 - rank, 1623
 - row, 1623
 - rowEchelon, 1623
 - sample, 1623

- scalarMatrix, [1623](#)
- setColumn, [1623](#)
- setelt, [1623](#)
- setRow, [1623](#)
- setsubMatrix, [1623](#)
- size?, [1623](#)
- square?, [1623](#)
- squareTop, [1623](#)
- subMatrix, [1623](#)
- swapColumns, [1623](#)
- swapRows, [1623](#)
- symmetric?, [1623](#)
- transpose, [1623](#)
- vertConcat, [1623](#)
- zero, [1623](#)
- Matrix, [1622](#)
- matrix
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - OUTFORM, [1868](#)
 - QFORM, [2154](#)
 - RMATRIX, [2246](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- matrixConcat3D
 - M3D, [2703](#)
- matrixDimensions
 - M3D, [2703](#)
- max
 - ALIST, [235](#)
 - AN, [35](#)
 - ARRAY1, [1774](#)
 - BINARY, [292](#)
 - BITS, [315](#)
 - BOOLEAN, [322](#)
 - BOP, [274](#)
 - BPADICRT, [263](#)
 - BSD, [286](#)
 - CARD, [332](#)
 - CCLASS, [382](#)
 - CDFVEC, [432](#)
 - CHAR, [373](#)
 - COMPLEX, [419](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - DFVEC, [605](#)
 - DIRPROD, [547](#)
 - DLIST, [460](#)
 - DMP, [572](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DSMP, [542](#)
 - EAB, [743](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FAGROUP, [1004](#)
 - FARRAY, [886](#)
 - FCOMP, [975](#)
 - FEXPR, [947](#)
 - FLOAT, [908](#)
 - FMONOID, [1021](#)
 - FRAC, [986](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - HDP, [1173](#)
 - HEAP, [1134](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - IARRAY1, [1244](#)
 - IBITS, [1199](#)
 - ICARD, [1193](#)
 - IDPOAM, [1212](#)
 - IDPOAMS, [1215](#)
 - IFARRAY, [1222](#)
 - ILIST, [1231](#)
 - INDE, [1218](#)
 - INT, [1362](#)
 - INTRVL, [1384](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - KERNEL, [1404](#)
 - LA, [1520](#)
 - LIST, [1504](#)
 - LO, [1523](#)
 - LWORD, [1532](#)

- MAGMA, [1565](#)
- MCMPLX, [1543](#)
- MFLOAT, [1548](#)
- MINT, [1557](#)
- MKCHSET, [1570](#)
- MODMON, [1632](#)
- MODMONOM, [1644](#)
- MPOLY, [1682](#)
- MYEXPR, [1688](#)
- MYUP, [1695](#)
- NNI, [1739](#)
- NSMP, [1714](#)
- NSUP, [1729](#)
- OCT, [1765](#)
- ODP, [1816](#)
- ODPOL, [1852](#)
- ODVAR, [1855](#)
- OFMONOID, [1829](#)
- ONECOMP, [1777](#)
- ORDCOMP, [1810](#)
- OSI, [1864](#)
- OVAR, [1836](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PBWL, [2054](#)
- PERM, [1949](#)
- PI, [2100](#)
- POINT, [2060](#)
- POLY, [2078](#)
- PRIMARR, [2109](#)
- PRODUCT, [2113](#)
- PRITION, [1923](#)
- QUAT, [2167](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- ROMAN, [2326](#)
- SAOS, [2418](#)
- SDPOL, [2386](#)
- SDVAR, [2389](#)
- SET, [2372](#)
- SHDP, [2509](#)
- SINT, [2412](#)
- SMP, [2423](#)
- STRING, [2607](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SYMBOL, [2640](#)
- U16VEC, [2924](#)
- U32VEC, [2931](#)
- U8VEC, [2917](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- VECTOR, [2942](#)
- maxCollIndex
 - ARRAY2, [2763](#)
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IARRAY2, [1256](#)
 - IIARRAY2, [1290](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - RMATRIX, [2246](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- maxdeg
 - XDPOLY, [2967](#)
 - XPBWPOLYL, [2987](#)
 - XPOLY, [2998](#)
 - XPR, [3007](#)
 - XPOLY, [3013](#)
- maxDegree
 - GOPT, [1104](#)
 - GOPT0, [1110](#)
- maxDerivative
 - GOPT, [1104](#)
 - GOPT0, [1110](#)
- maximumExponent
 - MFLOAT, [1548](#)
- maxIndex
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - BITS, [315](#)
 - CDFVEC, [432](#)
 - DFVEC, [605](#)
 - DIRPROD, [547](#)
 - DLIST, [460](#)

- DPM, [553](#)
- DPMO, [557](#)
- EQTBL, [699](#)
- FARRAY, [886](#)
- GSTBL, [1079](#)
- HASHTBL, [1120](#)
- HDP, [1173](#)
- IARRAY1, [1244](#)
- IBITS, [1199](#)
- IFARRAY, [1222](#)
- ILIST, [1231](#)
- INTABL, [1336](#)
- ISTRING, [1249](#)
- IVECTOR, [1260](#)
- KAFIL, [1414](#)
- LIB, [1429](#)
- LIST, [1504](#)
- NSDPS, [1703](#)
- ODP, [1816](#)
- POINT, [2060](#)
- PRIMARR, [2109](#)
- RESULT, [2301](#)
- ROUTINE, [2331](#)
- SHDP, [2509](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- U16VEC, [2924](#)
- U32VEC, [2931](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- maxint
 - MINT, [1557](#)
- maxLevel
 - GOPT, [1104](#)
 - GOPT0, [1110](#)
- maxMixedDegree
 - GOPT, [1104](#)
 - GOPT0, [1110](#)
- maxPoints
 - PLOT, [2029](#)
- maxPoints3D
 - PLOT3D, [2042](#)
- maxPower
 - GOPT, [1104](#)
 - GOPT0, [1110](#)
- maxRowIndex
 - ARRAY2, [2763](#)
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IARRAY2, [1256](#)
 - IIARRAY2, [1290](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - RMATRIX, [2246](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- maxShift
 - GOPT, [1104](#)
 - GOPT0, [1110](#)
- maxSubst
 - GOPT, [1104](#)
 - GOPT0, [1110](#)
- maxTower
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
- MCMPLEX, [1542](#)
- , [1543](#)
- ?<?, [1543](#)
- ?<=?, [1543](#)
- ?>?, [1543](#)
- ?>=?, [1543](#)
- ?**?, [1543](#)
- ?*?, [1543](#)
- ?+?, [1543](#)
- ?-?, [1543](#)
- ?., [1543](#)
- ?/?, [1543](#)
- ?=?, [1543](#)
- ?^?, [1543](#)
- ?~=?, [1543](#)
- ?quo?, [1543](#)
- ?rem?, [1543](#)
- 0, [1543](#)
- 1, [1543](#)
- abs, [1543](#)

- [acos](#), 1543
- [acosh](#), 1543
- [acot](#), 1543
- [acoth](#), 1543
- [acsc](#), 1543
- [acsch](#), 1543
- [argument](#), 1543
- [asec](#), 1543
- [asech](#), 1543
- [asin](#), 1543
- [asinh](#), 1543
- [associates?](#), 1543
- [atan](#), 1543
- [atanh](#), 1543
- [basis](#), 1543
- [characteristic](#), 1543
- [characteristicPolynomial](#), 1543
- [charthRoot](#), 1543
- [coerce](#), 1543
- [complex](#), 1543
- [conditionP](#), 1543
- [conjugate](#), 1543
- [convert](#), 1543
- [coordinates](#), 1543
- [cos](#), 1543
- [cosh](#), 1543
- [cot](#), 1543
- [coth](#), 1543
- [createPrimitiveElement](#), 1543
- [csc](#), 1543
- [csch](#), 1543
- [D](#), 1543
- [definingPolynomial](#), 1543
- [derivationCoordinates](#), 1543
- [differentiate](#), 1543
- [discreteLog](#), 1543
- [discriminant](#), 1543
- [divide](#), 1543
- [euclideanSize](#), 1543
- [eval](#), 1543
- [exp](#), 1543
- [expressIdealMember](#), 1543
- [exquo](#), 1543
- [extendedEuclidean](#), 1543
- [factor](#), 1543
- [factorPolynomial](#), 1543
- [factorsOfCyclicGroupSize](#), 1543
- [factorSquareFreePolynomial](#), 1543
- [gcd](#), 1543
- [gcdPolynomial](#), 1543
- [generator](#), 1543
- [hash](#), 1543
- [imag](#), 1543
- [imaginary](#), 1543
- [index](#), 1543
- [init](#), 1543
- [inv](#), 1543
- [latex](#), 1543
- [lcm](#), 1543
- [lift](#), 1543
- [log](#), 1543
- [lookup](#), 1543
- [map](#), 1543
- [max](#), 1543
- [min](#), 1543
- [minimalPolynomial](#), 1543
- [multiEuclidean](#), 1543
- [nextItem](#), 1543
- [norm](#), 1543
- [nthRoot](#), 1543
- [one?](#), 1543
- [order](#), 1543
- [patternMatch](#), 1543
- [pi](#), 1543
- [polarCoordinates](#), 1543
- [prime?](#), 1543
- [primeFrobenius](#), 1543
- [primitive?](#), 1543
- [primitiveElement](#), 1543
- [principalIdeal](#), 1543
- [random](#), 1543
- [rank](#), 1543
- [rational](#), 1543
- [rational?](#), 1543
- [rationalIfCan](#), 1543
- [real](#), 1543
- [recip](#), 1543
- [reduce](#), 1543
- [reducedSystem](#), 1543
- [regularRepresentation](#), 1543
- [representationType](#), 1543
- [represents](#), 1543

- retract, [1543](#)
- retractIfCan, [1543](#)
- sample, [1543](#)
- sec, [1543](#)
- sech, [1543](#)
- sin, [1543](#)
- sinh, [1543](#)
- size, [1543](#)
- sizeLess?, [1543](#)
- solveLinearPolynomialEquation, [1543](#)
- sqrt, [1543](#)
- squareFree, [1543](#)
- squareFreePart, [1543](#)
- squareFreePolynomial, [1543](#)
- subtractIfCan, [1543](#)
- tableForDiscreteLogarithm, [1543](#)
- tan, [1543](#)
- tanh, [1543](#)
- trace, [1543](#)
- traceMatrix, [1543](#)
- unit?, [1543](#)
- unitCanonical, [1543](#)
- unitNormal, [1543](#)
- zero?, [1543](#)
- mdeg
 - NSMP, [1714](#)
- measure
 - D01AJFA, [614](#)
 - D01AKFA, [617](#)
 - D01ALFA, [620](#)
 - D01AMFA, [623](#)
 - D01APFA, [629](#), [633](#)
 - D01ASFA, [636](#)
 - D01FCFA, [639](#)
 - D01GBFA, [643](#)
 - D01TRNS, [646](#)
 - D02BBFA, [651](#)
 - D02BHFA, [654](#)
 - D02CJFA, [658](#)
 - D02EJFA, [661](#)
 - D03EEFA, [665](#)
 - D03FAFA, [668](#)
 - D10ANFA, [626](#)
 - E04DGFA, [747](#)
 - E04FDFA, [750](#)
 - E04GCFA, [754](#)
 - E04JAFa, [758](#)
 - E04MBFA, [762](#)
 - E04NAFA, [765](#)
 - E04UCFA, [769](#)
- medialSet
 - WUTSET, [2959](#)
- member?
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - ARRAY2, [2763](#)
 - ASTACK, [65](#)
 - BBTREE, [253](#)
 - BITS, [315](#)
 - BSTREE, [302](#)
 - BTOURN, [306](#)
 - BTREE, [310](#)
 - CCLASS, [382](#)
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)
 - DEQUEUE, [512](#)
 - DFMAT, [599](#)
 - DFVEC, [605](#)
 - DHMATRIX, [491](#)
 - DIRPROD, [547](#)
 - DLIST, [460](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DSTREE, [535](#)
 - EQTBL, [699](#)
 - FARRAY, [886](#)
 - GPOLSET, [1074](#)
 - GSTBL, [1079](#)
 - GTSET, [1083](#)
 - HASHTBL, [1120](#)
 - HDP, [1173](#)
 - HEAP, [1134](#)
 - IARRAY1, [1244](#)
 - IARRAY2, [1256](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - IIARRAY2, [1290](#)
 - ILIST, [1231](#)
 - IMATRIX, [1239](#)
 - INTABL, [1336](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)

- KAFILE, 1414
- LIB, 1429
- LIST, 1504
- LMDICT, 1515
- LSQM, 1456
- M3D, 2703
- MATRIX, 1623
- MSET, 1670
- NSDPS, 1703
- ODP, 1816
- PENDTREE, 1945
- PERMGRP, 1960
- POINT, 2060
- PRIMARR, 2109
- QUEUE, 2184
- REGSET, 2286
- RESULT, 2301
- RGCHAIN, 2255
- RMATRIX, 2246
- ROUTINE, 2331
- SET, 2372
- SETMN, 2377
- SHDP, 2509
- SPLTREE, 2518
- SQMATRIX, 2548
- SREGSET, 2535
- STACK, 2563
- STBL, 2450
- STREAM, 2582
- STRING, 2607
- STRTBL, 2610
- TABLE, 2664
- TREE, 2741
- U16MAT, 2906
- U16VEC, 2924
- U32MAT, 2911
- U32VEC, 2931
- U8MAT, 2902
- U8VEC, 2917
- VECTOR, 2942
- WUTSET, 2959
- members
 - ALIST, 235
 - ARRAY1, 1774
 - ARRAY2, 2763
 - ASTACK, 65
 - BBTREE, 253
 - BITS, 315
 - BSTREE, 302
 - BTOURN, 306
 - BTREE, 310
 - CCLASS, 382
 - CDFMAT, 426
 - CDFVEC, 432
 - DEQUEUE, 512
 - DFMAT, 599
 - DFVEC, 605
 - DHMATRIX, 491
 - DIRPROD, 547
 - DLIST, 460
 - DPMM, 553
 - DPMO, 557
 - DSTREE, 535
 - EQTBL, 699
 - FARRAY, 886
 - GPOLSET, 1074
 - GSTBL, 1079
 - GTSET, 1083
 - HASHTBL, 1120
 - HDP, 1173
 - HEAP, 1134
 - IARRAY1, 1244
 - IARRAY2, 1256
 - IBITS, 1199
 - IFARRAY, 1222
 - IIARRAY2, 1290
 - ILIST, 1231
 - IMATRIX, 1239
 - INTABL, 1336
 - ISTRING, 1249
 - IVECTOR, 1260
 - KAFILE, 1414
 - LIB, 1429
 - LIST, 1504
 - LMDICT, 1515
 - LSQM, 1456
 - M3D, 2703
 - MATRIX, 1623
 - MSET, 1670
 - NSDPS, 1703
 - ODP, 1816
 - PENDTREE, 1945

- POINT, 2060
- PRIMARR, 2109
- QUEUE, 2184
- REGSET, 2286
- RESULT, 2301
- RGCHAIN, 2255
- RMATRIX, 2246
- ROUTINE, 2331
- SET, 2372
- SHDP, 2509
- SPLTREE, 2518
- SQMATRIX, 2548
- SREGSET, 2535
- STACK, 2563
- STBL, 2450
- STREAM, 2582
- STRING, 2607
- STRTBL, 2610
- TABLE, 2664
- TREE, 2741
- U16MAT, 2906
- U16VEC, 2924
- U32MAT, 2911
- U32VEC, 2931
- U8MAT, 2902
- U8VEC, 2917
- VECTOR, 2942
- WUTSET, 2959
- merge
 - ALIST, 235
 - ARRAY1, 1774
 - BITS, 315
 - CDFVEC, 432
 - DFVEC, 605
 - DLIST, 460
 - FARRAY, 886
 - HEAP, 1134
 - IARRAY1, 1244
 - IBITS, 1199
 - IFARRAY, 1222
 - ILIST, 1231
 - ISTRING, 1249
 - IVECTOR, 1260
 - LIST, 1504
 - POINT, 2060
 - PRIMARR, 2109
 - SPACE3, 2731
 - STRING, 2607
 - SUBSPACE, 2614
 - U16VEC, 2924
 - U32VEC, 2931
 - U8VEC, 2917
 - VECTOR, 2942
 - mesh
 - SPACE3, 2731
 - mesh?
 - SPACE3, 2731
 - message
 - OUTFORM, 1868
 - messagePrint
 - OUTFORM, 1868
 - MFLOAT, 1547
 - , 1548
 - ?<?, 1548
 - ?<=?, 1548
 - ?>?, 1548
 - ?>=?, 1548
 - ?**?, 1548
 - ?*?, 1548
 - ?+?, 1548
 - ?-?, 1548
 - ?/?, 1548
 - ?=?, 1548
 - ?^?, 1548
 - ?~=?, 1548
 - ?quo?, 1548
 - ?rem?, 1548
 - 0, 1548
 - 1, 1548
 - abs, 1548
 - associates?, 1548
 - base, 1548
 - bits, 1548
 - ceiling, 1548
 - changeBase, 1548
 - characteristic, 1548
 - coerce, 1548
 - convert, 1548
 - decreasePrecision, 1548
 - digits, 1548
 - divide, 1548
 - euclideanSize, 1548

- exponent, 1548
- expressIdealMember, 1548
- exquo, 1548
- extendedEuclidean, 1548
- factor, 1548
- float, 1548
- floor, 1548
- fractionPart, 1548
- gcd, 1548
- gcdPolynomial, 1548
- hash, 1548
- increasePrecision, 1548
- inv, 1548
- latex, 1548
- lcm, 1548
- mantissa, 1548
- max, 1548
- maximumExponent, 1548
- min, 1548
- minimumExponent, 1548
- multiEuclidean, 1548
- negative?, 1548
- norm, 1548
- nthRoot, 1548
- one?, 1548
- order, 1548
- patternMatch, 1548
- positive?, 1548
- precision, 1548
- prime?, 1548
- principalIdeal, 1548
- recip, 1548
- retract, 1548
- retractIfCan, 1548
- round, 1548
- sample, 1548
- sign, 1548
- sizeLess?, 1548
- sqrt, 1548
- squareFree, 1548
- squareFreePart, 1548
- subtractIfCan, 1548
- truncate, 1548
- unit?, 1548
- unitCanonical, 1548
- unitNormal, 1548
- wholePart, 1548
- zero?, 1548
- middle
 - ROIRC, 2310
- mightHaveRoots
 - ROIRC, 2310
- min
 - ALIST, 235
 - AN, 35
 - ARRAY1, 1774
 - BINARY, 292
 - BITS, 315
 - BOOLEAN, 322
 - BOP, 274
 - BPADICRT, 263
 - BSD, 286
 - CARD, 332
 - CCLASS, 382
 - CDFVEC, 432
 - CHAR, 373
 - COMPLEX, 419
 - DECIMAL, 465
 - DFLOAT, 587
 - DFVEC, 605
 - DIRPROD, 547
 - DLIST, 460
 - DMP, 572
 - DPMM, 553
 - DPMO, 557
 - DSMP, 542
 - EAB, 743
 - EXPEXPAN, 712
 - EXPR, 724
 - EXPUPXS, 740
 - FAGROUP, 1004
 - FARRAY, 886
 - FCOMP, 975
 - FEXPR, 947
 - FLOAT, 908
 - FMONOID, 1021
 - FRAC, 986
 - GDMP, 1052
 - HDMP, 1180
 - HDP, 1173
 - HEXADEC, 1143
 - IAN, 1276

IARRAY1, [1244](#)
IBITS, [1199](#)
ICARD, [1193](#)
IDPOAM, [1212](#)
IDPOAMS, [1215](#)
IFARRAY, [1222](#)
ILIST, [1231](#)
INDE, [1218](#)
INT, [1362](#)
INTRVL, [1384](#)
ISTRING, [1249](#)
IVECTOR, [1260](#)
KERNEL, [1404](#)
LA, [1520](#)
LIST, [1504](#)
LO, [1523](#)
LWORD, [1532](#)
MAGMA, [1565](#)
MCMPLX, [1543](#)
MFLOAT, [1548](#)
MINT, [1557](#)
MKCHSET, [1570](#)
MODMON, [1632](#)
MODMONOM, [1644](#)
MPOLY, [1682](#)
MYEXPR, [1688](#)
MYUP, [1695](#)
NNI, [1739](#)
NSMP, [1714](#)
NSUP, [1729](#)
OCT, [1765](#)
ODP, [1816](#)
ODPOL, [1852](#)
ODVAR, [1855](#)
OFMONOID, [1829](#)
ONECOMP, [1777](#)
ORDCOMP, [1810](#)
OSI, [1864](#)
OVAR, [1836](#)
PADICRAT, [1886](#)
PADICRC, [1891](#)
PBWLB, [2054](#)
PERM, [1949](#)
PI, [2100](#)
POINT, [2060](#)
POLY, [2078](#)

PRIMARR, [2109](#)
PRODUCT, [2113](#)
PRITION, [1923](#)
QUAT, [2167](#)
RADIX, [2206](#)
RECLOS, [2237](#)
ROMAN, [2326](#)
SAOS, [2418](#)
SDPOL, [2386](#)
SDVAR, [2389](#)
SET, [2372](#)
SHDP, [2509](#)
SINT, [2412](#)
SMP, [2423](#)
STRING, [2607](#)
SULS, [2457](#)
SUP, [2467](#)
SUPEXPR, [2481](#)
SYMBOL, [2640](#)
U16VEC, [2924](#)
U32VEC, [2931](#)
U8VEC, [2917](#)
ULS, [2796](#)
ULSCONS, [2804](#)
UP, [2828](#)
VECTOR, [2942](#)
minColIndex
 ARRAY2, [2763](#)
 CDFMAT, [426](#)
 DFMAT, [599](#)
 DHMATRIX, [491](#)
 IARRAY2, [1256](#)
 IIARRAY2, [1290](#)
 IMATRIX, [1239](#)
 LSQM, [1456](#)
 MATRIX, [1623](#)
 RMATRIX, [2246](#)
 SQMATRIX, [2548](#)
 U16MAT, [2906](#)
 U32MAT, [2911](#)
 U8MAT, [2902](#)
mindeg
 XDPOLY, [2967](#)
 XPBWPLYL, [2987](#)
 XPOLY, [2998](#)
 XPR, [3007](#)

- XRPOLY, 3013
- mindegTerm
 - XDPOLY, 2967
 - XPBWPLY, 2987
 - XPOLY, 2998
 - XRPOLY, 3013
- minimalPolynomial
 - ALGFF, 28
 - COMPLEX, 419
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866
 - FFP, 851
 - FFX, 846
 - IFF, 1284
 - IPF, 1303
 - MCMPLEX, 1543
 - PF, 2105
 - RADFF, 2194
 - SAE, 2400
- minimize
 - FRIDEAL, 995
- minimumDegree
 - DMP, 572
 - DSMP, 542
 - GDMP, 1052
 - HDMP, 1180
 - LODO, 1469
 - LODO1, 1479
 - LODO2, 1491
 - MODMON, 1632
 - MPOLY, 1682
 - MYUP, 1695
 - NSMP, 1714
 - NSUP, 1729
 - ODPOL, 1852
 - OMLO, 1806
 - ORESUP, 2492
 - OREUP, 2872
 - POLY, 2078
 - PR, 2092
 - SDPOL, 2386
 - SMP, 2423
 - SUP, 2467
 - SUPEXP, 2481
 - SYMPOLY, 2654
 - UP, 2828
 - UPXSSING, 2852
- minimumExponent
 - MFLOAT, 1548
- minIndex
 - ALIST, 235
 - ARRAY1, 1774
 - BITS, 315
 - CDFVEC, 432
 - DFVEC, 605
 - DIRPROD, 547
 - DLIST, 460
 - DPMM, 553
 - DPMO, 557
 - EQTBL, 699
 - FARRAY, 886
 - GSTBL, 1079
 - HASHTBL, 1120
 - HDP, 1173
 - IARRAY1, 1244
 - IBITS, 1199
 - IFARRAY, 1222
 - ILIST, 1231
 - INTABL, 1336
 - ISTRING, 1249
 - IVECTOR, 1260
 - KAFILE, 1414
 - LIB, 1429
 - LIST, 1504
 - NSDPS, 1703
 - ODP, 1816
 - POINT, 2060
 - PRIMARR, 2109
 - RESULT, 2301
 - ROUTINE, 2331
 - SHDP, 2509
 - STBL, 2450
 - STREAM, 2582
 - STRING, 2607
 - STRTBL, 2610
 - TABLE, 2664
 - U16VEC, 2924

- U32VEC, [2931](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- minordet
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- minPoints
 - PLOT, [2029](#)
- minPoints3D
 - PLOT3D, [2042](#)
- minPoly
 - AN, [35](#)
 - EXPR, [724](#)
 - FEXPR, [947](#)
 - IAN, [1276](#)
 - MYEXPR, [1688](#)
- minRowIndex
 - ARRAY2, [2763](#)
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IARRAY2, [1256](#)
 - IIARRAY2, [1290](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - RMATRIX, [2246](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- MINT, [1557](#)
- ?, [1557](#)
- ?<?, [1557](#)
- ?<=?, [1557](#)
- ?>?, [1557](#)
- ?>=?, [1557](#)
- ?**?, [1557](#)
- ?*?, [1557](#)
- ?+?, [1557](#)
- ?-?, [1557](#)
- ?=?, [1557](#)
- ?^?, [1557](#)
- ?~=?, [1557](#)
- ?quo?, [1557](#)
- ?rem?, [1557](#)
- 0, [1557](#)
- 1, [1557](#)
- abs, [1557](#)
- addmod, [1557](#)
- associates?, [1557](#)
- base, [1557](#)
- binomial, [1557](#)
- bit?, [1557](#)
- characteristic, [1557](#)
- coerce, [1557](#)
- convert, [1557](#)
- copy, [1557](#)
- D, [1557](#)
- dec, [1557](#)
- differentiate, [1557](#)
- divide, [1557](#)
- euclideanSize, [1557](#)
- even?, [1557](#)
- expressIdealMember, [1557](#)
- exquo, [1557](#)
- extendedEuclidean, [1557](#)
- factor, [1557](#)
- factorial, [1557](#)
- gcd, [1557](#)
- gcdPolynomial, [1557](#)
- hash, [1557](#)
- inc, [1557](#)
- init, [1557](#)
- invmod, [1557](#)
- latex, [1557](#)
- lcm, [1557](#)
- length, [1557](#)
- mask, [1557](#)
- max, [1557](#)
- maxint, [1557](#)
- min, [1557](#)
- mulmod, [1557](#)
- multiEuclidean, [1557](#)

- negative?, 1557
- nextItem, 1557
- odd?, 1557
- one?, 1557
- patternMatch, 1557
- permutation, 1557
- positive?, 1557
- positiveRemainder, 1557
- powmod, 1557
- prime?, 1557
- principalIdeal, 1557
- random, 1557
- rational, 1557
- rational?, 1557
- rationalIfCan, 1557
- recip, 1557
- reducedSystem, 1557
- retract, 1557
- retractIfCan, 1557
- sample, 1557
- shift, 1557
- sign, 1557
- sizeLess?, 1557
- squareFree, 1557
- squareFreePart, 1557
- submod, 1557
- subtractIfCan, 1557
- symmetricRemainder, 1557
- unit?, 1557
- unitCanonical, 1557
- unitNormal, 1557
- zero?, 1557
- minusInfinity
 - ORDCOMP, 1810
- mirror
 - LEXP, 1435
 - LPOLY, 1447
 - MAGMA, 1565
 - OFMONOID, 1829
 - XDPOLY, 2967
 - XPBWPOLYL, 2987
 - XPOLY, 2998
 - XPOLY, 3013
- mkAnswer
 - IR, 1375
- MKCHSET, 1570
- ?<?, 1570
- ?<=?, 1570
- ?>?, 1570
- ?>=?, 1570
- ?=?, 1570
- ?~=?, 1570
- coerce, 1570
- hash, 1570
- latex, 1570
- max, 1570
- min, 1570
- position, 1570
- setPosition, 1570
- mkcomm
 - COMM, 411
- MMLFORM, 1603
- ?=?, 1603
- ?~=?, 1603
- coerce, 1603
- coerceL, 1603
- coerceS, 1603
- display, 1603
- expres, 1603
- hash, 1603
- latex, 1603
- MODFIELD, 1638
- , 1638
- ?**?, 1638
- ?*?, 1638
- ?+?, 1638
- ?-?, 1638
- ?/?, 1638
- ?=?, 1638
- ?^?, 1638
- ?~=?, 1638
- ?quo?, 1638
- ?rem?, 1638
- 0, 1638
- 1, 1638
- associates?, 1638
- characteristic, 1638
- coerce, 1638
- divide, 1638
- euclideanSize, 1638
- expressIdealMember, 1638
- exQuo, 1638

- exquo, 1638
- extendedEuclidean, 1638
- factor, 1638
- gcd, 1638
- gcdPolynomial, 1638
- hash, 1638
- inv, 1638
- latex, 1638
- lcm, 1638
- modulus, 1638
- multiEuclidean, 1638
- one?, 1638
- prime?, 1638
- principalIdeal, 1638
- recip, 1638
- reduce, 1638
- sample, 1638
- sizeLess?, 1638
- squareFree, 1638
- squareFreePart, 1638
- subtractIfCan, 1638
- unit?, 1638
- unitCanonical, 1638
- unitNormal, 1638
- zero?, 1638
- modifyPoint
 - SUBSPACE, 2614
- modifyPointData
 - SPACE3, 2731
 - VIEW3D, 2710
- MODMON, 1631
- , 1632
- ?<?, 1632
- ?<=?, 1632
- ?>?, 1632
- ?>=?, 1632
- ?**?, 1632
- ?*?, 1632
- ?+?, 1632
- ?-?, 1632
- ?., 1632
- ?/? , 1632
- ?=?, 1632
- ?^?, 1632
- ?~=?, 1632
- ?quo?, 1632
- ?rem?, 1632
- 0, 1632
- 1, 1632
- An, 1632
- associates?, 1632
- binomThmExpt, 1632
- characteristic, 1632
- charthRoot, 1632
- coefficient, 1632
- coefficients, 1632
- coerce, 1632
- composite, 1632
- computePowers, 1632
- conditionP, 1632
- content, 1632
- convert, 1632
- D, 1632
- degree, 1632
- differentiate, 1632
- discriminant, 1632
- divide, 1632
- divideExponents, 1632
- elt, 1632
- euclideanSize, 1632
- eval, 1632
- expressIdealMember, 1632
- exquo, 1632
- extendedEuclidean, 1632
- factor, 1632
- factorPolynomial, 1632
- factorSquareFreePolynomial, 1632
- frobenius, 1632
- gcd, 1632
- gcdPolynomial, 1632
- ground, 1632
- ground?, 1632
- hash, 1632
- index, 1632
- init, 1632
- integrate, 1632
- isExpt, 1632
- isPlus, 1632
- isTimes, 1632
- karatsubaDivide, 1632
- latex, 1632
- lcm, 1632

leadingCoefficient, 1632
 leadingMonomial, 1632
 lift, 1632
 lookup, 1632
 mainVariable, 1632
 makeSUP, 1632
 map, 1632
 mapExponents, 1632
 max, 1632
 min, 1632
 minimumDegree, 1632
 modulus, 1632
 monicDivide, 1632
 monomial, 1632
 monomial?, 1632
 monomials, 1632
 multiEuclidean, 1632
 multiplyExponents, 1632
 multivariate, 1632
 nextItem, 1632
 numberOfMonomials, 1632
 one?, 1632
 order, 1632
 patternMatch, 1632
 pomopo, 1632
 pow, 1632
 prime?, 1632
 primitiveMonomials, 1632
 primitivePart, 1632
 principalIdeal, 1632
 pseudoDivide, 1632
 pseudoQuotient, 1632
 pseudoRemainder, 1632
 random, 1632
 recip, 1632
 reduce, 1632
 reducedSystem, 1632
 reductum, 1632
 resultant, 1632
 retract, 1632
 retractIfCan, 1632
 sample, 1632
 separate, 1632
 setPoly, 1632
 shiftLeft, 1632
 shiftRight, 1632

size, 1632
 sizeLess?, 1632
 solveLinearPolynomialEquation, 1632
 squareFree, 1632
 squareFreePart, 1632
 squareFreePolynomial, 1632
 subResultantGcd, 1632
 subtractIfCan, 1632
 totalDegree, 1632
 unit?, 1632
 unitCanonical, 1632
 unitNormal, 1632
 univariate, 1632
 unmakeSUP, 1632
 UnVectorise, 1632
 variables, 1632
 Vectorise, 1632
 vectorise, 1632
 zero?, 1632
 ModMonic, 1631
 MODMONOM, 1644
 ?<?, 1644
 ?<=?, 1644
 ?>?, 1644
 ?>=?, 1644
 ?=?, 1644
 ?~=?, 1644
 coerce, 1644
 construct, 1644
 exponent, 1644
 hash, 1644
 index, 1644
 latex, 1644
 max, 1644
 min, 1644
 MODOP, 1647
 -?, 1647, 1804
 ?**?, 1647, 1804
 ?*?, 1647, 1804
 ?+?, 1647, 1804
 ?-?, 1647, 1804
 ?..?, 1647, 1804
 ?=?, 1647, 1804
 ?^?, 1647, 1804
 ?~=?, 1647, 1804
 0, 1647, 1804

- 1, [1647](#), [1804](#)
- adjoint, [1647](#), [1804](#)
- characteristic, [1647](#), [1804](#)
- charthRoot, [1647](#), [1804](#)
- coerce, [1647](#), [1804](#)
- conjug, [1647](#), [1804](#)
- evaluate, [1647](#), [1804](#)
- evaluateInverse, [1647](#), [1804](#)
- hash, [1647](#), [1804](#)
- latex, [1647](#), [1804](#)
- makeop, [1647](#), [1804](#)
- one?, [1647](#), [1804](#)
- opeval, [1647](#), [1804](#)
- recip, [1647](#), [1804](#)
- retract, [1647](#), [1804](#)
- retractIfCan, [1647](#), [1804](#)
- sample, [1647](#), [1804](#)
- subtractIfCan, [1647](#), [1804](#)
- zero?, [1647](#), [1804](#)
- MODRING, [1641](#)
 - ?, [1641](#)
 - ?**?, [1641](#)
 - ?*?, [1641](#)
 - ?+?, [1641](#)
 - ?-?, [1641](#)
 - ?=?, [1641](#)
 - ?^?, [1641](#)
 - ?~=?, [1641](#)
 - 0, [1641](#)
 - 1, [1641](#)
 - characteristic, [1641](#)
 - coerce, [1641](#)
 - exQuo, [1641](#)
 - hash, [1641](#)
 - inv, [1641](#)
 - latex, [1641](#)
 - modulus, [1641](#)
 - one?, [1641](#)
 - recip, [1641](#)
 - reduce, [1641](#)
 - sample, [1641](#)
 - subtractIfCan, [1641](#)
 - zero?, [1641](#)
- ModularField, [1638](#)
- ModularRing, [1641](#)
- module
 - FRMOD, [1000](#)
 - ModuleMonomial, [1644](#)
 - ModuleOperator, [1647](#)
 - moduloP
 - BPADIC, [258](#)
 - IPADIC, [1294](#)
 - PADIC, [1881](#)
 - modulus
 - BPADIC, [258](#)
 - EMR, [702](#)
 - IPADIC, [1294](#)
 - MODFIELD, [1638](#)
 - MODMON, [1632](#)
 - MODRING, [1641](#)
 - PADIC, [1881](#)
 - MOEBIUS, [1654](#)
 - ?**?, [1654](#)
 - ?*?, [1654](#)
 - ?/? , [1654](#)
 - ?=?, [1654](#)
 - ?^?, [1654](#)
 - ?~=?, [1654](#)
 - 1, [1654](#)
 - coerce, [1654](#)
 - commutator, [1654](#)
 - conjugate, [1654](#)
 - eval, [1654](#)
 - hash, [1654](#)
 - inv, [1654](#)
 - latex, [1654](#)
 - moebius, [1654](#)
 - one?, [1654](#)
 - recip, [1654](#)
 - sample, [1654](#)
 - scale, [1654](#)
 - shift, [1654](#)
 - moebius
 - MOEBIUS, [1654](#)
 - MoebiusTransform, [1654](#)
 - monic?
 - NSMP, [1714](#)
 - monicDivide
 - DMP, [572](#)
 - DSMP, [542](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)

- MODMON, [1632](#)
- MPOLY, [1682](#)
- MYUP, [1695](#)
- NSMP, [1714](#)
- NSUP, [1729](#)
- ODPOL, [1852](#)
- POLY, [2078](#)
- SDPOL, [2386](#)
- SMP, [2423](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- UP, [2828](#)
- monicLeftDivide
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
- monicModulo
 - NSMP, [1714](#)
 - NSUP, [1729](#)
- monicRightDivide
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
- MonoidRing, [1658](#)
- monom
 - FM1, [1017](#)
 - LPOLY, [1447](#)
 - XDPOLY, [2967](#)
 - XPBWPOLYL, [2987](#)
 - XPOLY, [2998](#)
 - XPR, [3007](#)
 - XPOLY, [3013](#)
- monomial
 - CLIF, [402](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPUPXS, [740](#)
 - FM, [1013](#)
 - GDMP, [1052](#)
 - GMODPOL, [1059](#)
 - GSERIES, [1090](#)
 - HDMP, [1180](#)
 - IDPAG, [1202](#)
 - IDPAM, [1206](#)
 - IDPO, [1209](#)
 - IDPOAM, [1212](#)
 - IDPOAMS, [1215](#)
 - INDE, [1218](#)
 - ISUPS, [1311](#)
 - LAUPOL, [1422](#)
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MRING, [1659](#)
 - MYUP, [1695](#)
 - NSDPS, [1703](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - OMLO, [1806](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
 - POLY, [2078](#)
 - PR, [2092](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - SYMPOLY, [2654](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UP, [2828](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UPXSING, [2852](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- monomial2series
 - NSDPS, [1703](#)
- monomial?

- DMP, [572](#)
- DSMP, [542](#)
- EXPUPXS, [740](#)
- FM1, [1017](#)
- GDMP, [1052](#)
- GSERIES, [1090](#)
- HDMP, [1180](#)
- ISUPS, [1311](#)
- LAUPOL, [1422](#)
- LPOLY, [1447](#)
- MODMON, [1632](#)
- MPOLY, [1682](#)
- MRING, [1659](#)
- MYUP, [1695](#)
- NSDPS, [1703](#)
- NSMP, [1714](#)
- NSUP, [1729](#)
- ODPOL, [1852](#)
- POLY, [2078](#)
- PR, [2092](#)
- SDPOL, [2386](#)
- SMP, [2423](#)
- SMTS, [2441](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- SYMPOLY, [2654](#)
- TS, [2671](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- XDPOLY, [2967](#)
- XPBWPLYL, [2987](#)
- XPOLY, [2998](#)
- XPR, [3007](#)
- XPOLY, [3013](#)
- monomials
 - DMP, [572](#)
 - DSMP, [542](#)
 - FM1, [1017](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - LPOLY, [1447](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MRING, [1659](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - POLY, [2078](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - UP, [2828](#)
 - XDPOLY, [2967](#)
 - XPBWPLYL, [2987](#)
 - XPR, [3007](#)
- MonteCarlo
 - GOPT0, [1110](#)
- more?
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - ARRAY2, [2763](#)
 - ASTACK, [65](#)
 - BBTREE, [253](#)
 - BITS, [315](#)
 - BSTREE, [302](#)
 - BTourn, [306](#)
 - BTREE, [310](#)
 - CCLASS, [382](#)
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)
 - DEQUEUE, [512](#)
 - DFMAT, [599](#)
 - DFVEC, [605](#)
 - DHMATRIX, [491](#)
 - DIRPROD, [547](#)
 - DLIST, [460](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DSTREE, [535](#)
 - EQTBL, [699](#)

- FARRAY, [886](#)
- GPOLSET, [1074](#)
- GSTBL, [1079](#)
- GTSET, [1083](#)
- HASHTBL, [1120](#)
- HDP, [1173](#)
- HEAP, [1134](#)
- IARRAY1, [1244](#)
- IARRAY2, [1256](#)
- IBITS, [1199](#)
- IFARRAY, [1222](#)
- IIARRAY2, [1290](#)
- ILIST, [1231](#)
- IMATRIX, [1239](#)
- INTABL, [1336](#)
- ISTRING, [1249](#)
- IVECTOR, [1260](#)
- KAFILE, [1414](#)
- LIB, [1429](#)
- LIST, [1504](#)
- LMDICT, [1515](#)
- LSQM, [1456](#)
- M3D, [2703](#)
- MATRIX, [1623](#)
- MSET, [1670](#)
- NSDPS, [1703](#)
- ODP, [1816](#)
- PENDTREE, [1945](#)
- POINT, [2060](#)
- PRIMARR, [2109](#)
- QUEUE, [2184](#)
- REGSET, [2286](#)
- RESULT, [2301](#)
- RGCHAIN, [2255](#)
- RMATRIX, [2246](#)
- ROUTINE, [2331](#)
- SET, [2372](#)
- SHDP, [2509](#)
- SPLTREE, [2518](#)
- SQMATRIX, [2548](#)
- SREGSET, [2535](#)
- STACK, [2563](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- TREE, [2741](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- WUTSET, [2959](#)
- morphism
 - AUTOMOR, [245](#)
- move
 - VIEW2d, [2770](#)
 - VIEW3D, [2710](#)
- movedPoints
 - PERM, [1949](#)
 - PERMGRP, [1960](#)
- MPOLY, [1681](#)
 - , [1682](#)
 - ?<?, [1682](#)
 - ?<=?, [1682](#)
 - ?>?, [1682](#)
 - ?>=?, [1682](#)
 - ?**?, [1682](#)
 - ?*?, [1682](#)
 - ?+?, [1682](#)
 - ?-?, [1682](#)
 - ?/?, [1682](#)
 - ?=?, [1682](#)
 - ?^?, [1682](#)
 - ?~=?, [1682](#)
 - 0, [1682](#)
 - 1, [1682](#)
 - associates?, [1682](#)
 - binomThmExpt, [1682](#)
 - characteristic, [1682](#)
 - charthRoot, [1682](#)
 - coefficient, [1682](#)
 - coefficients, [1682](#)
 - coerce, [1682](#)
 - conditionP, [1682](#)
 - content, [1682](#)
 - convert, [1682](#)
 - D, [1682](#)
 - degree, [1682](#)

- differentiate, [1682](#)
- discriminant, [1682](#)
- eval, [1682](#)
- exquo, [1682](#)
- factor, [1682](#)
- factorPolynomial, [1682](#)
- factorSquareFreePolynomial, [1682](#)
- gcd, [1682](#)
- gcdPolynomial, [1682](#)
- ground, [1682](#)
- ground?, [1682](#)
- hash, [1682](#)
- isExpt, [1682](#)
- isPlus, [1682](#)
- isTimes, [1682](#)
- latex, [1682](#)
- lcm, [1682](#)
- leadingCoefficient, [1682](#)
- leadingMonomial, [1682](#)
- mainVariable, [1682](#)
- map, [1682](#)
- mapExponents, [1682](#)
- max, [1682](#)
- min, [1682](#)
- minimumDegree, [1682](#)
- monicDivide, [1682](#)
- monomial, [1682](#)
- monomial?, [1682](#)
- monomials, [1682](#)
- multivariate, [1682](#)
- numberOfMonomials, [1682](#)
- one?, [1682](#)
- patternMatch, [1682](#)
- pomopo, [1682](#)
- prime?, [1682](#)
- primitiveMonomials, [1682](#)
- primitivePart, [1682](#)
- recip, [1682](#)
- reducedSystem, [1682](#)
- reductum, [1682](#)
- resultant, [1682](#)
- retract, [1682](#)
- retractIfCan, [1682](#)
- sample, [1682](#)
- solveLinearPolynomialEquation, [1682](#)
- squareFree, [1682](#)
- squareFreePart, [1682](#)
- squareFreePolynomial, [1682](#)
- subtractIfCan, [1682](#)
- totalDegree, [1682](#)
- unit?, [1682](#)
- unitCanonical, [1682](#)
- unitNormal, [1682](#)
- univariate, [1682](#)
- variables, [1682](#)
- zero?, [1682](#)
- MRING, [1658](#)
- ?, [1659](#)
- ?**?, [1659](#)
- ?*?, [1659](#)
- ?+?, [1659](#)
- ?-?, [1659](#)
- ?=?, [1659](#)
- ?^?, [1659](#)
- ?~=?, [1659](#)
- 0, [1659](#)
- 1, [1659](#)
- characteristic, [1659](#)
- charthRoot, [1659](#)
- coefficient, [1659](#)
- coefficients, [1659](#)
- coerce, [1659](#)
- hash, [1659](#)
- index, [1659](#)
- latex, [1659](#)
- leadingCoefficient, [1659](#)
- leadingMonomial, [1659](#)
- lookup, [1659](#)
- map, [1659](#)
- monomial, [1659](#)
- monomial?, [1659](#)
- monomials, [1659](#)
- numberOfMonomials, [1659](#)
- one?, [1659](#)
- random, [1659](#)
- recip, [1659](#)
- reductum, [1659](#)
- retract, [1659](#)
- retractIfCan, [1659](#)
- sample, [1659](#)
- size, [1659](#)
- subtractIfCan, [1659](#)

- terms, 1659
- zero?, 1659
- MSET, 1670
 - ?<?, 1670
 - ?=?, 1670
 - ?~=?, 1670
 - #?, 1670
 - any?, 1670
 - bag, 1670
 - brace, 1670
 - coerce, 1670
 - construct, 1670
 - convert, 1670
 - copy, 1670
 - count, 1670
 - dictionary, 1670
 - difference, 1670
 - duplicates, 1670
 - empty, 1670
 - empty?, 1670
 - eq?, 1670
 - eval, 1670
 - every?, 1670
 - extract, 1670
 - find, 1670
 - hash, 1670
 - insert, 1670
 - inspect, 1670
 - intersect, 1670
 - latex, 1670
 - less?, 1670
 - map, 1670
 - member?, 1670
 - members, 1670
 - more?, 1670
 - multiset, 1670
 - parts, 1670
 - reduce, 1670
 - remove, 1670
 - removeDuplicates, 1670
 - sample, 1670
 - select, 1670
 - set, 1670
 - size?, 1670
 - subset?, 1670
 - symmetricDifference, 1670
 - union, 1670
- mulmod
 - INT, 1362
 - MINT, 1557
 - ROMAN, 2326
 - SINT, 2412
- multiEuclidean
 - ALGFF, 28
 - AN, 35
 - BINARY, 292
 - BPADIC, 258
 - BPADICRT, 263
 - COMPLEX, 419
 - CONTFRAC, 445
 - DECIMAL, 465
 - DFLOAT, 587
 - EMR, 702
 - EXPEXPAN, 712
 - EXPR, 724
 - EXPUPXS, 740
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866
 - FFP, 851
 - FFX, 846
 - FLOAT, 908
 - FRAC, 986
 - GSERIES, 1090
 - HACKPI, 1978
 - HEXADEC, 1143
 - IAN, 1276
 - IFF, 1284
 - INT, 1362
 - IPADIC, 1294
 - IPF, 1303
 - LAUPOL, 1422
 - MCMPLEX, 1543
 - MFLOAT, 1548
 - MINT, 1557
 - MODFIELD, 1638
 - MODMON, 1632
 - MYEXPR, 1688

- MYUP, [1695](#)
- NSDPS, [1703](#)
- NSUP, [1729](#)
- ODR, [1858](#)
- PACOFF, [2136](#)
- PACRAT, [2146](#)
- PADIC, [1881](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PF, [2105](#)
- PFR, [1914](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- ROMAN, [2326](#)
- SAE, [2400](#)
- SINT, [2412](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- multiple?
 - PATTERN, [1928](#)
- multiplicative?
 - DIRRING, [564](#)
- multiplyCoefficients
 - ISUPS, [1311](#)
 - SULS, [2457](#)
 - SUTS, [2497](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- multiplyExponents
 - EXPUPXS, [740](#)
 - GSERIES, [1090](#)
 - ISUPS, [1311](#)
 - MODMON, [1632](#)
 - MYUP, [1695](#)
 - NSDPS, [1703](#)
 - NSUP, [1729](#)
 - SULS, [2457](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - UP, [2828](#)
- multisect
 - UFPS, [2789](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- Multiset, [1670](#)
- multiset
 - MSET, [1670](#)
- multivariate
 - DMP, [572](#)
 - DSMP, [542](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - POLY, [2078](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - UP, [2828](#)
- MultivariatePolynomial, [1681](#)
- multMonom
 - GMODPOL, [1059](#)
- multV
 - IC, [1191](#)
 - INFCLSPS, [1271](#)
 - INFCLSPT, [1265](#)
- mvar

- GPOLSET, 1074
- GTSET, 1083
- NSMP, 1714
- REGSET, 2286
- RGCHAIN, 2255
- SREGSET, 2535
- WUTSET, 2959
- MYEXPR, 1687
- ?, 1688
- ?<?, 1688
- ?<=?, 1688
- ?>?, 1688
- ?>=?, 1688
- ?**?, 1688
- ?*?, 1688
- ?+?, 1688
- ?-?, 1688
- ?/?, 1688
- ?=?, 1688
- ?^?, 1688
- ?~=?, 1688
- ?quo?, 1688
- ?rem?, 1688
- 0, 1688
- 1, 1688
- applyQuote, 1688
- associates?, 1688
- belong?, 1688
- binomial, 1688
- box, 1688
- characteristic, 1688
- charthRoot, 1688
- coerce, 1688
- commutator, 1688
- conjugate, 1688
- convert, 1688
- D, 1688
- definingPolynomial, 1688
- denom, 1688
- denominator, 1688
- differentiate, 1688
- distribute, 1688
- divide, 1688
- elt, 1688
- euclideanSize, 1688
- eval, 1688
- even?, 1688
- expressIdealMember, 1688
- exquo, 1688
- extendedEuclidean, 1688
- factor, 1688
- factorial, 1688
- factorials, 1688
- freeOf?, 1688
- gcd, 1688
- gcdPolynomial, 1688
- ground, 1688
- ground?, 1688
- hash, 1688
- height, 1688
- inv, 1688
- is?, 1688
- isExpt, 1688
- isMult, 1688
- isPlus, 1688
- isPower, 1688
- isTimes, 1688
- kernel, 1688
- kernels, 1688
- latex, 1688
- lcm, 1688
- mainKernel, 1688
- map, 1688
- max, 1688
- min, 1688
- minPoly, 1688
- multiEuclidean, 1688
- numer, 1688
- numerator, 1688
- odd?, 1688
- one?, 1688
- operator, 1688
- operators, 1688
- paren, 1688
- patternMatch, 1688
- permutation, 1688
- prime?, 1688
- principalIdeal, 1688
- product, 1688
- recip, 1688
- reducedSystem, 1688
- retract, 1688

- retractIfCan, 1688
- sample, 1688
- sizeLess?, 1688
- squareFree, 1688
- squareFreePart, 1688
- subst, 1688
- subtractIfCan, 1688
- summation, 1688
- tower, 1688
- unit?, 1688
- unitCanonical, 1688
- unitNormal, 1688
- univariate, 1688
- variables, 1688
- zero?, 1688
- MyExpression, 1687
- MyUnivariatePolynomial, 1694
- MYUP, 1694
- , 1695
- ?<?, 1695
- ?<=?, 1695
- ?>?, 1695
- ?>=?, 1695
- ?**?, 1695
- ?*?, 1695
- ?+?, 1695
- ?-?, 1695
- ?., 1695
- ?/?, 1695
- ?=?, 1695
- ?^?, 1695
- ?~=?, 1695
- ?quo?, 1695
- ?rem?, 1695
- 0, 1695
- 1, 1695
- associates?, 1695
- binomThmExpt, 1695
- characteristic, 1695
- charthRoot, 1695
- coefficient, 1695
- coefficients, 1695
- coerce, 1695
- composite, 1695
- conditionP, 1695
- content, 1695
- convert, 1695
- D, 1695
- degree, 1695
- differentiate, 1695
- discriminant, 1695
- divide, 1695
- divideExponents, 1695
- elt, 1695
- euclideanSize, 1695
- eval, 1695
- expressIdealMember, 1695
- exquo, 1695
- extendedEuclidean, 1695
- factor, 1695
- factorPolynomial, 1695
- factorSquareFreePolynomial, 1695
- finecg, 1695
- gcd, 1695
- gcdPolynomial, 1695
- ground, 1695
- ground?, 1695
- hash, 1695
- init, 1695
- integrate, 1695
- isExpt, 1695
- isPlus, 1695
- isTimes, 1695
- karatsubaDivide, 1695
- latex, 1695
- lcm, 1695
- leadingCoefficient, 1695
- leadingMonomial, 1695
- mainVariable, 1695
- makeSUP, 1695
- map, 1695
- mapExponents, 1695
- max, 1695
- min, 1695
- minimumDegree, 1695
- monicDivide, 1695
- monomial, 1695
- monomial?, 1695
- monomials, 1695
- multiEuclidean, 1695
- multiplyExponents, 1695
- multivariate, 1695

- nextItem, 1695
- numberOfMonomials, 1695
- one?, 1695
- order, 1695
- patternMatch, 1695
- pomopo, 1695
- prime?, 1695
- primitiveMonomials, 1695
- primitivePart, 1695
- principalIdeal, 1695
- pseudoDivide, 1695
- pseudoQuotient, 1695
- pseudoRemainder, 1695
- recip, 1695
- reducedSystem, 1695
- reductum, 1695
- resultant, 1695
- retract, 1695
- retractIfCan, 1695
- sample, 1695
- separate, 1695
- shiftLeft, 1695
- shiftRight, 1695
- sizeLess?, 1695
- solveLinearPolynomialEquation, 1695
- squareFree, 1695
- squareFreePart, 1695
- squareFreePolynomial, 1695
- subResultantGcd, 1695
- subtractIfCan, 1695
- totalDegree, 1695
- unit?, 1695
- unitCanonical, 1695
- unitNormal, 1695
- univariate, 1695
- unmakeSUP, 1695
- variables, 1695
- vectorise, 1695
- zero?, 1695
- name
 - BINFILE, 295
 - BOP, 274
 - FILE, 802
 - FNAME, 810
 - FTEM, 967
 - FUNCTION, 1045
 - KAFILE, 1414
 - KERNEL, 1404
 - RULECOLD, 2340
 - SYMBOL, 2640
 - TEXTFILE, 2692
- nand
 - BITS, 315
 - BOOLEAN, 322
 - IBITS, 1199
- nary?
 - BOP, 274
- ncols
 - ARRAY2, 2763
 - CDFMAT, 426
 - DFMAT, 599
 - DHMATRIX, 491
 - IARRAY2, 1256
 - IIARRAY2, 1290
 - IMATRIX, 1239
 - LSQM, 1456
 - MATRIX, 1623
 - RMATRIX, 2246
 - SQMATRIX, 2548
 - U16MAT, 2906
 - U32MAT, 2911
 - U8MAT, 2902
- negative?
 - BINARY, 292
 - BPADICRT, 263
 - DECIMAL, 465
 - DFLOAT, 587
 - DIRPROD, 547
 - DPMM, 553
 - DPMO, 557
 - EXPEXPAN, 712
 - FLOAT, 908
 - FRAC, 986
 - HDP, 1173
 - HEXADEC, 1143
 - INT, 1362
 - INTRVL, 1384
 - LA, 1520
 - MFLOAT, 1548
 - MINT, 1557
 - ODP, 1816

- ONECOMP, [1777](#)
- ORDCOMP, [1810](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- ROIRC, [2310](#)
- ROMAN, [2326](#)
- SHDP, [2509](#)
- SINT, [2412](#)
- SULS, [2457](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- NeitherSparseOrDensePowerSeries, [1701](#)
- new
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - ARRAY2, [2763](#)
 - BITS, [315](#)
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)
 - COMPPROP, [2624](#)
 - DFMAT, [599](#)
 - DFVEC, [605](#)
 - DHMATRIX, [491](#)
 - DLIST, [460](#)
 - FARRAY, [886](#)
 - FNAME, [810](#)
 - FORMULA, [2346](#)
 - IARRAY1, [1244](#)
 - IARRAY2, [1256](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - IIARRAY2, [1290](#)
 - ILIST, [1231](#)
 - IMATRIX, [1239](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - LIST, [1504](#)
 - MATRIX, [1623](#)
 - NSDPS, [1703](#)
 - PATLRES, [1937](#)
 - PATRES, [1940](#)
 - POINT, [2060](#)
 - PRIMARR, [2109](#)
 - STREAM, [2582](#)
 - STRING, [2607](#)
 - SUBSPACE, [2614](#)
 - SYMBOL, [2640](#)
 - TEX, [2677](#)
 - U16MAT, [2906](#)
 - U16VEC, [2924](#)
 - U32MAT, [2911](#)
 - U32VEC, [2931](#)
 - U8MAT, [2902](#)
 - U8VEC, [2917](#)
 - VECTOR, [2942](#)
- newElement
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
- NewSparseMultivariatePolynomial, [1713](#)
- NewSparseUnivariatePolynomial, [1728](#)
- newSubProgram
 - SYMS, [2696](#)
- newTypeLists
 - SYMTAB, [2648](#)
- nextItem
 - ALGFF, [28](#)
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - DECIMAL, [465](#)
 - EXPEXPAN, [712](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FRAC, [986](#)
 - HEXADEC, [1143](#)
 - IFF, [1284](#)
 - INT, [1362](#)
 - IPF, [1303](#)
 - MCMPLEX, [1543](#)
 - MINT, [1557](#)
 - MODMON, [1632](#)
 - MYUP, [1695](#)
 - NSUP, [1729](#)

- PACOFF, [2136](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PF, [2105](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- ROMAN, [2326](#)
- SAE, [2400](#)
- SINT, [2412](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- ZMOD, [1368](#)
- nextsubResultant2
 - NSMP, [1714](#)
- nil
 - LIST, [1504](#)
- nilFactor
 - FR, [786](#)
- NIPROB, [1746](#)
 - ?=?, [1746](#)
 - ?~=?, [1746](#)
 - coerce, [1746](#)
 - hash, [1746](#)
 - latex, [1746](#)
 - retract, [1746](#)
- NNI, [1739](#)
 - ?<?, [1739](#)
 - ?<=?, [1739](#)
 - ?>?, [1739](#)
 - ?>=?, [1739](#)
 - ?**?, [1739](#)
 - ?*?, [1739](#)
 - ?+?, [1739](#)
 - ?=?, [1739](#)
 - ?^?, [1739](#)
 - ?~=?, [1739](#)
 - ?quo?, [1739](#)
 - ?rem?, [1739](#)
 - 0, [1739](#)
 - 1, [1739](#)
 - coerce, [1739](#)
 - divide, [1739](#)
 - exquo, [1739](#)
 - gcd, [1739](#)
 - hash, [1739](#)
 - latex, [1739](#)
 - max, [1739](#)
 - min, [1739](#)
 - one?, [1739](#)
 - random, [1739](#)
 - recip, [1739](#)
 - sample, [1739](#)
 - shift, [1739](#)
 - subtractIfCan, [1739](#)
 - sup, [1739](#)
 - zero?, [1739](#)
- node
 - BBTREE, [253](#)
 - BSTREE, [302](#)
 - BTOURN, [306](#)
 - BTREE, [310](#)
- node?
 - ALIST, [235](#)
 - BBTREE, [253](#)
 - BSTREE, [302](#)
 - BTOURN, [306](#)
 - BTREE, [310](#)
 - DLIST, [460](#)
 - DSTREE, [535](#)
 - ILIST, [1231](#)
 - LIST, [1504](#)
 - NSDPS, [1703](#)
 - PENDTREE, [1945](#)
 - SPLTREE, [2518](#)
 - STREAM, [2582](#)
 - TREE, [2741](#)
- nodeOf?
 - SPLTREE, [2518](#)
- nodes
 - ALIST, [235](#)
 - BBTREE, [253](#)
 - BSTREE, [302](#)
 - BTOURN, [306](#)
 - BTREE, [310](#)
 - DLIST, [460](#)
 - DSTREE, [535](#)
 - ILIST, [1231](#)
 - LIST, [1504](#)

- NSDPS, [1703](#)
- PENDTREE, [1945](#)
- SPLTREE, [2518](#)
- STREAM, [2582](#)
- TREE, [2741](#)
- noncommutativeJordanAlgebra?
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- NONE, [1737](#)
 - ?=?, [1737](#)
 - ?~=?, [1737](#)
 - coerce, [1737](#)
 - hash, [1737](#)
 - latex, [1737](#)
- None, [1737](#)
- NonNegativeInteger, [1739](#)
- nonSingularModel
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- nor
 - BITS, [315](#)
 - BOOLEAN, [322](#)
 - IBITS, [1199](#)
- norm
 - ALGFF, [28](#)
 - AN, [35](#)
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FLOAT, [908](#)
 - FRIDEAL, [995](#)
 - FRMOD, [1000](#)
 - IAN, [1276](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - MCMPLX, [1543](#)
 - MFLOAT, [1548](#)
 - OCT, [1765](#)
 - PF, [2105](#)
 - QUAT, [2167](#)
 - RADFF, [2194](#)
 - SAE, [2400](#)
- normal?
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - PF, [2105](#)
- normalElement
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - PF, [2105](#)
- normalize
 - FLOAT, [908](#)
- normalizeAtInfinity
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- normalized?
 - GTSET, [1083](#)
 - NSMP, [1714](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- NOT

- SWITCH, 2629
- Not
 - IBITS, 1199
 - SINT, 2412
- not?
 - BITS, 315
 - BOOLEAN, 322
 - IBITS, 1199
 - OUTFORM, 1868
 - SINT, 2412
- notelem
 - IR, 1375
- NOTTING, 1744
 - *, 1744
 - **, 1744
 - /, 1744
 - =, 1744
 - ^, 1744
 - ^=, 1744
 - 1, 1744
 - coerce, 1744
 - commutator, 1744
 - conjugate, 1744
 - hash, 1744
 - inv, 1744
 - latex, 1744
 - one?, 1744
 - recip, 1744
 - retract, 1744
 - sample, 1744
- NottinghamGroup, 1744
- nrows
 - ARRAY2, 2763
 - CDFMAT, 426
 - DFMAT, 599
 - DHMATRIX, 491
 - IARRAY2, 1256
 - IIARRAY2, 1290
 - IMATRIX, 1239
 - LSQM, 1456
 - MATRIX, 1623
 - RMATRIX, 2246
 - SQMATRIX, 2548
 - U16MAT, 2906
 - U32MAT, 2911
 - U8MAT, 2902
- NSDPS, 1701
 - , 1703
 - ?**, 1703
 - ?*, 1703
 - ?+?, 1703
 - ?-, 1703
 - ?.?, 1703
 - ?.first, 1703
 - ?.last, 1703
 - ?.rest, 1703
 - ?.value, 1703
 - ?/?, 1703
 - ?=?, 1703
 - ?^?, 1703
 - ?~=?, 1703
 - ?quo?, 1703
 - ?rem?, 1703
 - #?, 1703
 - 0, 1703
 - 1, 1703
 - any?, 1703
 - approximate, 1703
 - associates?, 1703
 - center, 1703
 - characteristic, 1703
 - charthRoot, 1703
 - child?, 1703
 - children, 1703
 - coefficient, 1703
 - coefOfFirstNonZeroTerm, 1703
 - coerce, 1703
 - complete, 1703
 - concat, 1703
 - construct, 1703
 - convert, 1703
 - copy, 1703
 - count, 1703
 - cycleEntry, 1703
 - cycleLength, 1703
 - cycleSplit, 1703
 - cycleTail, 1703
 - cyclic?, 1703
 - D, 1703
 - degree, 1703
 - delay, 1703
 - delete, 1703

- differentiate, 1703
- distance, 1703
- divide, 1703
- elt, 1703
- empty, 1703
- empty?, 1703
- entries, 1703
- entry?, 1703
- eq?, 1703
- euclideanSize, 1703
- eval, 1703
- every?, 1703
- explicitEntries?, 1703
- explicitlyEmpty?, 1703
- explicitlyFinite?, 1703
- expressIdealMember, 1703
- exquo, 1703
- extend, 1703
- extendedEuclidean, 1703
- factor, 1703
- fill, 1703
- filterUpTo, 1703
- find, 1703
- findCoef, 1703
- findTerm, 1703
- first, 1703
- frst, 1703
- gcd, 1703
- gcdPolynomial, 1703
- hash, 1703
- index?, 1703
- indices, 1703
- insert, 1703
- inv, 1703
- last, 1703
- latex, 1703
- lazy?, 1703
- lazyEvaluate, 1703
- lcm, 1703
- leadingCoefficient, 1703
- leadingMonomial, 1703
- leaf?, 1703
- leaves, 1703
- less?, 1703
- map, 1703
- maxIndex, 1703
- member?, 1703
- members, 1703
- minIndex, 1703
- monomial, 1703
- monomial2series, 1703
- monomial?, 1703
- more?, 1703
- multiEuclidean, 1703
- multiplyExponents, 1703
- new, 1703
- node?, 1703
- nodes, 1703
- numberOfComputedEntries, 1703
- one?, 1703
- order, 1703
- orderIfNegative, 1703
- parts, 1703
- pole?, 1703
- posExpnPart, 1703
- possiblyInfinite?, 1703
- prime?, 1703
- principalIdeal, 1703
- printInfo, 1703
- qelt, 1703
- qsetelt, 1703
- recip, 1703
- reduce, 1703
- reductum, 1703
- remove, 1703
- removeDuplicates, 1703
- removeFirstZeroes, 1703
- removeZeroes, 1703
- rest, 1703
- rst, 1703
- sample, 1703
- sbt, 1703
- second, 1703
- select, 1703
- series, 1703
- setchildren, 1703
- setelt, 1703
- setfirst, 1703
- setlast, 1703
- setrest, 1703
- setvalue, 1703
- shift, 1703

- size?, 1703
- sizeLess?, 1703
- split, 1703
- squareFree, 1703
- squareFreePart, 1703
- subtractIfCan, 1703
- swap, 1703
- tail, 1703
- terms, 1703
- third, 1703
- truncate, 1703
- unit?, 1703
- unitCanonical, 1703
- unitNormal, 1703
- value, 1703
- variable, 1703
- variables, 1703
- zero?, 1703
- NSMP, 1713
- ?, 1714
- ?<?, 1714
- ?<=?, 1714
- ?>?, 1714
- ?>=?, 1714
- ?**?, 1714
- ?*?, 1714
- ?+?, 1714
- ?-?, 1714
- ?/?, 1714
- ?=?, 1714
- ?^?, 1714
- ?~=?, 1714
- 0, 1714
- 1, 1714
- associates?, 1714
- binomThmExpt, 1714
- characteristic, 1714
- charthRoot, 1714
- coefficient, 1714
- coefficients, 1714
- coerce, 1714
- conditionP, 1714
- content, 1714
- D, 1714
- deepestInitial, 1714
- deepestTail, 1714
- degree, 1714
- differentiate, 1714
- discriminant, 1714
- eval, 1714
- exactQuotient, 1714
- exquo, 1714
- extendedSubResultantGcd, 1714
- factor, 1714
- factorPolynomial, 1714
- factorSquareFreePolynomial, 1714
- gcd, 1714
- gcdPolynomial, 1714
- ground, 1714
- ground?, 1714
- halfExtendedSubResultantGcd1, 1714
- halfExtendedSubResultantGcd2, 1714
- hash, 1714
- head, 1714
- headReduce, 1714
- headReduced?, 1714
- infRittWu?, 1714
- init, 1714
- initiallyReduce, 1714
- initiallyReduced?, 1714
- isExpt, 1714
- isPlus, 1714
- isTimes, 1714
- iteratedInitials, 1714
- lastSubResultant, 1714
- latex, 1714
- LazardQuotient, 1714
- LazardQuotient2, 1714
- lazyPquo, 1714
- lazyPrem, 1714
- lazyPremWithDefault, 1714
- lazyPseudoDivide, 1714
- lazyResidueClass, 1714
- lcm, 1714
- leadingCoefficient, 1714
- leadingMonomial, 1714
- leastMonomial, 1714
- mainCoefficients, 1714
- mainContent, 1714
- mainMonomial, 1714
- mainMonomials, 1714
- mainPrimitivePart, 1714

- mainSquareFreePart, 1714
- mainVariable, 1714
- map, 1714
- mapExponents, 1714
- max, 1714
- mdeg, 1714
- min, 1714
- minimumDegree, 1714
- monic?, 1714
- monicDivide, 1714
- monicModulo, 1714
- monomial, 1714
- monomial?, 1714
- monomials, 1714
- multivariate, 1714
- mvar, 1714
- nextsubResultant2, 1714
- normalized?, 1714
- numberOfMonomials, 1714
- one?, 1714
- patternMatch, 1714
- pomopo, 1714
- pquo, 1714
- prem, 1714
- prime?, 1714
- primitiveMonomials, 1714
- primitivePart, 1714
- primPartElseUnitCanonical, 1714
- pseudoDivide, 1714
- quasiMonic?, 1714
- recip, 1714
- reduced?, 1714
- reducedSystem, 1714
- reductum, 1714
- resultant, 1714
- retract, 1714
- retractIfCan, 1714
- RittWuCompare, 1714
- sample, 1714
- solveLinearPolynomialEquation, 1714
- squareFree, 1714
- squareFreePart, 1714
- squareFreePolynomial, 1714
- subResultantChain, 1714
- subResultantGcd, 1714
- subtractIfCan, 1714
- supRittWu?, 1714
- tail, 1714
- totalDegree, 1714
- unit?, 1714
- unitCanonical, 1714
- unitNormal, 1714
- univariate, 1714
- variables, 1714
- zero?, 1714
- NSUP, 1728
- ?, 1729
- ?<?, 1729
- ?<=?, 1729
- ?>?, 1729
- ?>=?, 1729
- ?**?, 1729
- ?*?, 1729
- ?+?, 1729
- ?-?, 1729
- ?., 1729
- ?/?, 1729
- ?=, 1729
- ?^?, 1729
- ?~=?, 1729
- ?quo?, 1729
- ?rem?, 1729
- 0, 1729
- 1, 1729
- associates?, 1729
- binomThmExpt, 1729
- characteristic, 1729
- charthRoot, 1729
- coefficient, 1729
- coefficients, 1729
- coerce, 1729
- composite, 1729
- conditionP, 1729
- content, 1729
- convert, 1729
- D, 1729
- degree, 1729
- differentiate, 1729
- discriminant, 1729
- divide, 1729
- divideExponents, 1729
- elt, 1729

- euclideanSize, 1729
- eval, 1729
- expressIdealMember, 1729
- exquo, 1729
- extendedEuclidean, 1729
- extendedResultant, 1729
- extendedSubResultantGcd, 1729
- factor, 1729
- factorPolynomial, 1729
- factorSquareFreePolynomial, 1729
- fmeq, 1729
- gcd, 1729
- gcdPolynomial, 1729
- ground, 1729
- ground?, 1729
- halfExtendedResultant1, 1729
- halfExtendedResultant2, 1729
- halfExtendedSubResultantGcd1, 1729
- halfExtendedSubResultantGcd2, 1729
- hash, 1729
- init, 1729
- integrate, 1729
- isExpt, 1729
- isPlus, 1729
- isTimes, 1729
- karatsubaDivide, 1729
- lastSubResultant, 1729
- latex, 1729
- lazyPseudoDivide, 1729
- lazyPseudoQuotient, 1729
- lazyPseudoRemainder, 1729
- lazyResidueClass, 1729
- lcm, 1729
- leadingCoefficient, 1729
- leadingMonomial, 1729
- mainVariable, 1729
- makeSUP, 1729
- map, 1729
- mapExponents, 1729
- max, 1729
- min, 1729
- minimumDegree, 1729
- monicDivide, 1729
- monicModulo, 1729
- monomial, 1729
- monomial?, 1729
- monomials, 1729
- multiEuclidean, 1729
- multiplyExponents, 1729
- multivariate, 1729
- nextItem, 1729
- numberOfMonomials, 1729
- one?, 1729
- order, 1729
- patternMatch, 1729
- pop, 1729
- prime?, 1729
- primitiveMonomials, 1729
- primitivePart, 1729
- principalIdeal, 1729
- pseudoDivide, 1729
- pseudoQuotient, 1729
- pseudoRemainder, 1729
- recip, 1729
- reducedSystem, 1729
- reductum, 1729
- resultant, 1729
- retract, 1729
- retractIfCan, 1729
- sample, 1729
- separate, 1729
- shiftLeft, 1729
- shiftRight, 1729
- sizeLess?, 1729
- solveLinearPolynomialEquation, 1729
- squareFree, 1729
- squareFreePart, 1729
- squareFreePolynomial, 1729
- subResultantGcd, 1729
- subResultantsChain, 1729
- subtractIfCan, 1729
- totalDegree, 1729
- unit?, 1729
- unitCanonical, 1729
- unitNormal, 1729
- univariate, 1729
- unmakeSUP, 1729
- variables, 1729
- vectorise, 1729
- zero?, 1729
- nthCoef
 - DIV, 575

- FAGROUP, [1004](#)
- FAMONOID, [1007](#)
- IFAMON, [1287](#)
- nthExpon
 - FGROUP, [1010](#)
 - FMONOID, [1021](#)
 - LMOPS, [1509](#)
 - OFMONOID, [1829](#)
- nthExponent
 - FR, [786](#)
- nthFactor
 - DIV, [575](#)
 - FAGROUP, [1004](#)
 - FAMONOID, [1007](#)
 - FGROUP, [1010](#)
 - FMONOID, [1021](#)
 - FR, [786](#)
 - IFAMON, [1287](#)
 - LMOPS, [1509](#)
 - OFMONOID, [1829](#)
- nthFlag
 - FR, [786](#)
- nthFractionalTerm
 - PFR, [1914](#)
- nthRoot
 - AN, [35](#)
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - IAN, [1276](#)
 - INTRVL, [1384](#)
 - MCMPLEX, [1543](#)
 - MFLOAT, [1548](#)
 - RECLOS, [2237](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- nthRootIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- Nul
 - EAB, [743](#)
- null
 - LIST, [1504](#)
- null?
 - INFORM, [1343](#)
 - SEX, [2392](#)
 - SEXOF, [2394](#)
- nullary?
 - BOP, [274](#)
- nullity
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - RMATRIX, [2246](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- nullSpace
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - RMATRIX, [2246](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- number?
 - EXPR, [724](#)
- numberOfChildren
 - SUBSPACE, [2614](#)
- numberOfComponents
 - ALGFF, [28](#)

- RADFF, [2194](#)
- SPACE3, [2731](#)
- numberOfComputedEntries
 - NSDPS, [1703](#)
 - STREAM, [2582](#)
- numberOfCycles
 - PERM, [1949](#)
- numberOfFactors
 - FR, [786](#)
- numberOfFractionalTerms
 - PFR, [1914](#)
- numberOfHues
 - COLOR, [407](#)
- numberOfMonomials
 - DMP, [572](#)
 - DSMP, [542](#)
 - FM1, [1017](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - LPOLY, [1447](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MRING, [1659](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - POLY, [2078](#)
 - PR, [2092](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - SYMPOLY, [2654](#)
 - UP, [2828](#)
 - UPXSSING, [2852](#)
 - XDPOLY, [2967](#)
 - XPBWPLYL, [2987](#)
 - XPR, [3007](#)
- numer
 - AN, [35](#)
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - DECIMAL, [465](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - FRAC, [986](#)
 - FRIDEAL, [995](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - LA, [1520](#)
 - LO, [1523](#)
 - MYEXPR, [1688](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - RADIX, [2206](#)
 - SULS, [2457](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
- numerator
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - DECIMAL, [465](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - FRAC, [986](#)
 - HEXADEC, [1143](#)
 - MYEXPR, [1688](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - RADIX, [2206](#)
 - SULS, [2457](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
- numerators
 - CONTFRAC, [445](#)
- numericalIntegration
 - D01AJFA, [614](#)
 - D01AKFA, [617](#)
 - D01ALFA, [620](#)
 - D01AMFA, [623](#)
 - D01APFA, [629](#), [633](#)
 - D01ASFA, [636](#)
 - D01FCFA, [639](#)
 - D01GBFA, [643](#)
 - D01TRNS, [646](#)
 - D10ANFA, [626](#)
- NumericalIntegrationProblem, [1746](#)
- NumericalODEProblem, [1750](#)
- numericalOptimization
 - E04DGFA, [747](#)
 - E04FDFA, [750](#)

- E04GCFA, [754](#)
- E04JAFA, [758](#)
- E04MBFA, [762](#)
- E04NAFA, [765](#)
- E04UCFA, [769](#)
- NumericalOptimizationProblem, [1753](#)
- NumericalPDEProblem, [1756](#)
- numFunEvals
 - PLOT, [2029](#)
- numFunEvals3D
 - PLOT3D, [2042](#)
- obj
 - ANY, [50](#)
- objectOf
 - ANY, [50](#)
- objects
 - SPACE3, [2731](#)
- OCT, [1765](#)
 - , [1765](#)
 - ?<?, [1765](#)
 - ?<=?, [1765](#)
 - ?>?, [1765](#)
 - ?>=?, [1765](#)
 - ?**?, [1765](#)
 - ?*?, [1765](#)
 - ?+?, [1765](#)
 - ?-?, [1765](#)
 - ?..?, [1765](#)
 - ?=?, [1765](#)
 - ?^?, [1765](#)
 - ?~=?, [1765](#)
 - 0, [1765](#)
 - 1, [1765](#)
 - abs, [1765](#)
 - characteristic, [1765](#)
 - charthRoot, [1765](#)
 - coerce, [1765](#)
 - conjugate, [1765](#)
 - convert, [1765](#)
 - eval, [1765](#)
 - hash, [1765](#)
 - imageE, [1765](#)
 - imagI, [1765](#)
 - imagi, [1765](#)
 - imagJ, [1765](#)
 - imagk, [1765](#)
 - imagK, [1765](#)
 - imagj, [1765](#)
 - index, [1765](#)
 - inv, [1765](#)
 - latex, [1765](#)
 - lookup, [1765](#)
 - map, [1765](#)
 - max, [1765](#)
 - min, [1765](#)
 - norm, [1765](#)
 - octon, [1765](#)
 - one?, [1765](#)
 - random, [1765](#)
 - rational, [1765](#)
 - rational?, [1765](#)
 - rationalIfCan, [1765](#)
 - real, [1765](#)
 - recip, [1765](#)
 - retract, [1765](#)
 - retractIfCan, [1765](#)
 - sample, [1765](#)
 - size, [1765](#)
 - subtractIfCan, [1765](#)
 - zero?, [1765](#)
- octon
 - OCT, [1765](#)
- Octonion, [1765](#)
- odd?
 - AN, [35](#)
 - EXPR, [724](#)
 - FEXPR, [947](#)
 - IAN, [1276](#)
 - INT, [1362](#)
 - MINT, [1557](#)
 - MYEXPR, [1688](#)
 - PERM, [1949](#)
 - ROMAN, [2326](#)
 - SINT, [2412](#)
- oddlambert
 - UFPS, [2789](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- ODEIFTBL, [1768](#)
 - clearTheIFTable, [1768](#)
 - iFTable, [1768](#)

- insert, 1768
- keys, 1768
- showIntensityFunctions, 1768
- showTheIFTable, 1768
- ODEIntensityFunctionsTable, 1768
- ODEPROB, 1750
 - ?=?, 1750
 - ?~=?, 1750
 - coerce, 1750
 - hash, 1750
 - latex, 1750
 - retract, 1750
- ODESolve
 - D02BBFA, 651
 - D02BHFA, 654
 - D02CJFA, 658
 - D02EJFA, 661
- ODP, 1816
 - , 1816
 - ?<?, 1816
 - ?<=?, 1816
 - ?>?, 1816
 - ?>=?, 1816
 - ?**?, 1816
 - ?*?, 1816
 - ?+?, 1816
 - ?-?, 1816
 - ?.?, 1816
 - ?/? , 1816
 - ?=?, 1816
 - ?^?, 1816
 - ?~=?, 1816
 - #?, 1816
 - 0, 1816
 - 1, 1816
 - abs, 1816
 - any?, 1816
 - characteristic, 1816
 - coerce, 1816
 - copy, 1816
 - count, 1816
 - D, 1816
 - differentiate, 1816
 - dimension, 1816
 - directProduct, 1816
 - dot, 1816
 - elt, 1816
 - empty, 1816
 - empty?, 1816
 - entries, 1816
 - entry?, 1816
 - eq?, 1816
 - eval, 1816
 - every?, 1816
 - fill, 1816
 - first, 1816
 - hash, 1816
 - index, 1816
 - index?, 1816
 - indices, 1816
 - latex, 1816
 - less?, 1816
 - lookup, 1816
 - map, 1816
 - max, 1816
 - maxIndex, 1816
 - member?, 1816
 - members, 1816
 - min, 1816
 - minIndex, 1816
 - more?, 1816
 - negative?, 1816
 - one?, 1816
 - parts, 1816
 - positive?, 1816
 - qelt, 1816
 - qsetelt, 1816
 - random, 1816
 - recip, 1816
 - reducedSystem, 1816
 - retract, 1816
 - retractIfCan, 1816
 - sample, 1816
 - setelt, 1816
 - sign, 1816
 - size, 1816
 - size?, 1816
 - subtractIfCan, 1816
 - sup, 1816
 - swap, 1816
 - unitVector, 1816
 - zero?, 1816

- ODPOL, 1851
- ?, 1852
- ?<?, 1852
- ?<=?, 1852
- ?>?, 1852
- ?>=?, 1852
- ?**?, 1852
- ?*?, 1852
- ?+?, 1852
- ?-?, 1852
- ?/?, 1852
- ?=?, 1852
- ?^?, 1852
- ?~=?, 1852
- 0, 1852
- 1, 1852
- associates?, 1852
- binomThmExpt, 1852
- characteristic, 1852
- charthRoot, 1852
- coefficient, 1852
- coefficients, 1852
- coerce, 1852
- conditionP, 1852
- content, 1852
- D, 1852
- degree, 1852
- differentialVariables, 1852
- differentiate, 1852
- discriminant, 1852
- eval, 1852
- exquo, 1852
- factor, 1852
- factorPolynomial, 1852
- factorSquareFreePolynomial, 1852
- gcd, 1852
- gcdPolynomial, 1852
- ground, 1852
- ground?, 1852
- hash, 1852
- initial, 1852
- isExpt, 1852
- isobaric?, 1852
- isPlus, 1852
- isTimes, 1852
- latex, 1852
- lcm, 1852
- leader, 1852
- leadingCoefficient, 1852
- leadingMonomial, 1852
- mainVariable, 1852
- map, 1852
- mapExponents, 1852
- max, 1852
- min, 1852
- minimumDegree, 1852
- monicDivide, 1852
- monomial, 1852
- monomial?, 1852
- monomials, 1852
- multivariate, 1852
- numberOfMonomials, 1852
- one?, 1852
- order, 1852
- patternMatch, 1852
- pomopo, 1852
- prime?, 1852
- primitiveMonomials, 1852
- primitivePart, 1852
- recip, 1852
- reducedSystem, 1852
- reductum, 1852
- resultant, 1852
- retract, 1852
- retractIfCan, 1852
- sample, 1852
- separant, 1852
- solveLinearPolynomialEquation, 1852
- squareFree, 1852
- squareFreePart, 1852
- squareFreePolynomial, 1852
- subtractIfCan, 1852
- totalDegree, 1852
- unit?, 1852
- unitCanonical, 1852
- unitNormal, 1852
- univariate, 1852
- variables, 1852
- weight, 1852
- weights, 1852
- zero?, 1852
- ODR, 1858

- ?, 1858
- ?**, 1858
- ?*, 1858
- ?+, 1858
- ?-, 1858
- ?/?, 1858
- ?=?, 1858
- ?^?, 1858
- ?~=?, 1858
- ?quo?, 1858
- ?rem?, 1858
- 0, 1858
- 1, 1858
- associates?, 1858
- characteristic, 1858
- coerce, 1858
- D, 1858
- differentiate, 1858
- divide, 1858
- euclideanSize, 1858
- expressIdealMember, 1858
- exquo, 1858
- extendedEuclidean, 1858
- factor, 1858
- gcd, 1858
- gcdPolynomial, 1858
- hash, 1858
- inv, 1858
- latex, 1858
- lcm, 1858
- multiEuclidean, 1858
- one?, 1858
- prime?, 1858
- principalIdeal, 1858
- recip, 1858
- sample, 1858
- sizeLess?, 1858
- squareFree, 1858
- squareFreePart, 1858
- subtractIfCan, 1858
- unit?, 1858
- unitCanonical, 1858
- unitNormal, 1858
- zero?, 1858
- ODVAR, 1855
- ?<?, 1855
- ?<=?, 1855
- ?>?, 1855
- ?>=?, 1855
- ?=?, 1855
- ?~=?, 1855
- coerce, 1855
- differentiate, 1855
- hash, 1855
- latex, 1855
- makeVariable, 1855
- max, 1855
- min, 1855
- order, 1855
- retract, 1855
- retractIfCan, 1855
- variable, 1855
- weight, 1855
- OFMONOID, 1829
- ?<?, 1829
- ?<=?, 1829
- ?>?, 1829
- ?>=?, 1829
- ?**, 1829
- ?*, 1829
- ?=?, 1829
- ?^?, 1829
- ?~=?, 1829
- ?div?, 1829
- 1, 1829
- coerce, 1829
- factors, 1829
- first, 1829
- hash, 1829
- hclf, 1829
- hcrf, 1829
- latex, 1829
- length, 1829
- lexico, 1829
- lquo, 1829
- max, 1829
- min, 1829
- mirror, 1829
- nthExpon, 1829
- nthFactor, 1829
- one?, 1829
- overlap, 1829

- recip, [1829](#)
 - rest, [1829](#)
 - retract, [1829](#)
 - retractIfCan, [1829](#)
 - rquo, [1829](#)
 - sample, [1829](#)
 - size, [1829](#)
 - varList, [1829](#)
- OMbindTCP
 - OMCONN, [1781](#)
- OMclose
 - OMDEV, [1784](#)
- OMcloseConn
 - OMCONN, [1781](#)
- OMCONN, [1781](#)
 - OMbindTCP, [1781](#)
 - OMcloseConn, [1781](#)
 - OMconnectTCP, [1781](#)
 - OMconnInDevice, [1781](#)
 - OMconnOutDevice, [1781](#)
 - OMmakeConn, [1781](#)
- OMconnectTCP
 - OMCONN, [1781](#)
- OMconnInDevice
 - OMCONN, [1781](#)
- OMconnOutDevice
 - OMCONN, [1781](#)
- OMDEV, [1784](#)
 - OMclose, [1784](#)
 - OMgetApp, [1784](#)
 - OMgetAtp, [1784](#)
 - OMgetAttr, [1784](#)
 - OMgetBind, [1784](#)
 - OMgetBVar, [1784](#)
 - OMgetEndApp, [1784](#)
 - OMgetEndAtp, [1784](#)
 - OMgetEndAttr, [1784](#)
 - OMgetEndBind, [1784](#)
 - OMgetEndBVar, [1784](#)
 - OMgetEndError, [1784](#)
 - OMgetEndObject, [1784](#)
 - OMgetError, [1784](#)
 - OMgetFloat, [1784](#)
 - OMgetInteger, [1784](#)
 - OMgetObject, [1784](#)
 - OMgetString, [1784](#)
 - OMgetSymbol, [1784](#)
 - OMgetType, [1784](#)
 - OMgetVariable, [1784](#)
 - OMopenFile, [1784](#)
 - OMopenString, [1784](#)
 - OMputApp, [1784](#)
 - OMputAtp, [1784](#)
 - OMputAttr, [1784](#)
 - OMputBind, [1784](#)
 - OMputBVar, [1784](#)
 - OMputEndApp, [1784](#)
 - OMputEndAtp, [1784](#)
 - OMputEndAttr, [1784](#)
 - OMputEndBind, [1784](#)
 - OMputEndBVar, [1784](#)
 - OMputEndError, [1784](#)
 - OMputEndObject, [1784](#)
 - OMputError, [1784](#)
 - OMputFloat, [1784](#)
 - OMputInteger, [1784](#)
 - OMputObject, [1784](#)
 - OMputString, [1784](#)
 - OMputSymbol, [1784](#)
 - OMputVariable, [1784](#)
 - OMsetEncoding, [1784](#)
- OMENC, [1789](#)
 - ?=?, [1789](#)
 - ?~=?, [1789](#)
 - coerce, [1789](#)
 - hash, [1789](#)
 - latex, [1789](#)
 - OMencodingBinary, [1789](#)
 - OMencodingSGML, [1789](#)
 - OMencodingUnknown, [1789](#)
 - OMencodingXML, [1789](#)
- OMencodingBinary
 - OMENC, [1789](#)
- OMencodingSGML
 - OMENC, [1789](#)
- OMencodingUnknown
 - OMENC, [1789](#)
- OMencodingXML
 - OMENC, [1789](#)
- OMERR, [1792](#)
 - ?=?, [1792](#)
 - ? =?, [1792](#)

- coerce, 1792
- errorInfo, 1792
- errorKind, 1792
- hash, 1792
- latex, 1792
- omError, 1792
- OMERRK, 1794
 - ?=?, 1794
 - ?~=?, 1794
 - coerce, 1794
 - hash, 1794
 - latex, 1794
 - OMParseError?, 1794
 - OMReadError?, 1794
 - OMUnknownCD?, 1794
 - OMUnknownSymbol?, 1794
- omError
 - OMERR, 1792
- OMgetApp
 - OMDEV, 1784
- OMgetAtp
 - OMDEV, 1784
- OMgetAttr
 - OMDEV, 1784
- OMgetBind
 - OMDEV, 1784
- OMgetBVar
 - OMDEV, 1784
- OMgetEndApp
 - OMDEV, 1784
- OMgetEndAtp
 - OMDEV, 1784
- OMgetEndAttr
 - OMDEV, 1784
- OMgetEndBind
 - OMDEV, 1784
- OMgetEndBVar
 - OMDEV, 1784
- OMgetEndError
 - OMDEV, 1784
- OMgetEndObject
 - OMDEV, 1784
- OMgetError
 - OMDEV, 1784
- OMgetFloat
 - OMDEV, 1784
- OMgetInteger
 - OMDEV, 1784
- OMgetObject
 - OMDEV, 1784
- OMgetString
 - OMDEV, 1784
- OMgetSymbol
 - OMDEV, 1784
- OMgetType
 - OMDEV, 1784
- OMgetVariable
 - OMDEV, 1784
- OMLO, 1806
 - , 1806
 - ?**?, 1806
 - ?*?, 1806
 - ?+?, 1806
 - ?-?, 1806
 - ?=?, 1806
 - ?^?, 1806
 - ?~=?, 1806
 - 0, 1806
 - 1, 1806
 - characteristic, 1806
 - coefficient, 1806
 - coerce, 1806
 - D, 1806
 - degree, 1806
 - differentiate, 1806
 - hash, 1806
 - latex, 1806
 - leadingCoefficient, 1806
 - minimumDegree, 1806
 - monomial, 1806
 - one?, 1806
 - op, 1806
 - po, 1806
 - recip, 1806
 - reductum, 1806
 - sample, 1806
 - subtractIfCan, 1806
 - zero?, 1806
- OMmakeConn
 - OMCONN, 1781
- OMopenFile
 - OMDEV, 1784

- OMOpenString
 - OMDEV, [1784](#)
- OMParseError?
 - OMERRK, [1794](#)
- OMputApp
 - OMDEV, [1784](#)
- OMputAtp
 - OMDEV, [1784](#)
- OMputAttr
 - OMDEV, [1784](#)
- OMputBind
 - OMDEV, [1784](#)
- OMputBVar
 - OMDEV, [1784](#)
- OMputEndApp
 - OMDEV, [1784](#)
- OMputEndAtp
 - OMDEV, [1784](#)
- OMputEndAttr
 - OMDEV, [1784](#)
- OMputEndBind
 - OMDEV, [1784](#)
- OMputEndBVar
 - OMDEV, [1784](#)
- OMputEndError
 - OMDEV, [1784](#)
- OMputEndObject
 - OMDEV, [1784](#)
- OMputError
 - OMDEV, [1784](#)
- OMputFloat
 - OMDEV, [1784](#)
- OMputInteger
 - OMDEV, [1784](#)
- OMputObject
 - OMDEV, [1784](#)
- OMputString
 - OMDEV, [1784](#)
- OMputSymbol
 - OMDEV, [1784](#)
- OMputVariable
 - OMDEV, [1784](#)
- OMReadError?
 - OMERRK, [1794](#)
- OMsetEncoding
 - OMDEV, [1784](#)
- OMUnknownCD?
 - OMERRK, [1794](#)
- OMUnknownSymbol?
 - OMERRK, [1794](#)
- OMwrite
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - FLOAT, [908](#)
 - FRAC, [986](#)
 - INT, [1362](#)
 - LIST, [1504](#)
 - SINT, [2412](#)
 - STRING, [2607](#)
 - SYMBOL, [2640](#)
- one
 - GOPT, [1104](#)
 - GOPT0, [1110](#)
- one?
 - ALGFF, [28](#)
 - AN, [35](#)
 - ANTISYM, [40](#)
 - AUTOMOR, [245](#)
 - BINARY, [292](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - CARD, [332](#)
 - CLIF, [402](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DERHAM, [530](#)
 - DFLOAT, [587](#)
 - DIRPROD, [547](#)
 - DIRRING, [564](#)
 - DMP, [572](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DSMP, [542](#)
 - EMR, [702](#)
 - EQ, [691](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FEXPR, [947](#)
 - FF, [820](#)
 - FFCG, [825](#)

- FFCGP, 835
FFCGX, 830
FFNB, 860
FFNBP, 872
FFNBX, 866
FFP, 851
FFX, 846
FGROUP, 1010
FLOAT, 908
FMONOID, 1021
FR, 786
FRAC, 986
FRIDEAL, 995
FRMOD, 1000
FSERIES, 978
GDMP, 1052
GSERIES, 1090
HACKPI, 1978
HDMP, 1180
HDP, 1173
HEXADEC, 1143
IAN, 1276
IDEAL, 2081
IFF, 1284
INT, 1362
INTRVL, 1384
IPADIC, 1294
IPF, 1303
ISUPS, 1311
ITAYLOR, 1338
LA, 1520
LAUPOL, 1422
LEXP, 1435
LODO, 1469
LODO1, 1479
LODO2, 1491
LSQM, 1456
MCMPLEX, 1543
MFLOAT, 1548
MINT, 1557
MODFIELD, 1638
MODMON, 1632
MODOP, 1647, 1804
MODRING, 1641
MOEBIUS, 1654
MPOLY, 1682
MRING, 1659
MYEXPR, 1688
MYUP, 1695
NNI, 1739
NOTTING, 1744
NSDPS, 1703
NSMP, 1714
NSUP, 1729
OCT, 1765
ODP, 1816
ODPOL, 1852
ODR, 1858
OFMONOID, 1829
OMLO, 1806
ONECOMP, 1777
ORDCOMP, 1810
ORESUP, 2492
OREUP, 2872
OWP, 1861
PACOFF, 2136
PACRAT, 2146
PADIC, 1881
PADICRAT, 1886
PADICRC, 1891
PERM, 1949
PF, 2105
PFR, 1914
PI, 2100
POLY, 2078
PR, 2092
PRODUCT, 2113
QUAT, 2167
RADFF, 2194
RADIX, 2206
RECLOS, 2237
RESRING, 2296
ROMAN, 2326
SAE, 2400
SDPOL, 2386
SHDP, 2509
SINT, 2412
SMP, 2423
SMTS, 2441
SQMATRIX, 2548
SULS, 2457
SUP, 2467

- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- SYMPOLY, [2654](#)
- TS, [2671](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- WP, [2949](#)
- XDPOLY, [2967](#)
- XPBWPLYL, [2987](#)
- XPOLY, [2998](#)
- XPR, [3007](#)
- XPOLY, [3013](#)
- ZMOD, [1368](#)
- ONECOMP, [1777](#)
- ?, [1777](#)
- ?<?, [1777](#)
- ?<=?, [1777](#)
- ?>?, [1777](#)
- ?>=?, [1777](#)
- ?**?, [1777](#)
- ?*?, [1777](#)
- ?+?, [1777](#)
- ?-?, [1777](#)
- ?=?, [1777](#)
- ?^?, [1777](#)
- ?~=?, [1777](#)
- 0, [1777](#)
- 1, [1777](#)
- abs, [1777](#)
- characteristic, [1777](#)
- coerce, [1777](#)
- finite?, [1777](#)
- hash, [1777](#)
- infinite?, [1777](#)
- infinity, [1777](#)
- latex, [1777](#)
- max, [1777](#)
- min, [1777](#)
- negative?, [1777](#)
- one?, [1777](#)
- positive?, [1777](#)
- rational, [1777](#)
- rational?, [1777](#)
- rationalIfCan, [1777](#)
- recip, [1777](#)
- retract, [1777](#)
- retractIfCan, [1777](#)
- sample, [1777](#)
- sign, [1777](#)
- subtractIfCan, [1777](#)
- zero?, [1777](#)
- OneDimensionalArray, [1774](#)
- oneDimensionalArray
 - ARRAY1, [1774](#)
- OnePointCompletion, [1777](#)
- OP, [1804](#)
- op
 - OMLO, [1806](#)
- open
 - BINFILE, [295](#)
 - FILE, [802](#)
 - FTEM, [967](#)
 - KAFILE, [1414](#)
 - TEXTFILE, [2692](#)
- open?
 - TUBE, [2749](#)
- OpenMathConnection, [1781](#)
- OpenMathDevice, [1784](#)
- OpenMathEncoding, [1789](#)
- OpenMathError, [1792](#)
- OpenMathErrorKind, [1794](#)
- operation
 - FC, [931](#)
- Operator, [1804](#)
- operator
 - AN, [35](#)
 - BOP, [274](#)
 - EXPR, [724](#)
 - FEXPR, [947](#)
 - IAN, [1276](#)
 - KERNEL, [1404](#)
 - MYEXPR, [1688](#)
- operators
 - AN, [35](#)

- EXPR, 724
- FEXPR, 947
- IAN, 1276
- MYEXPR, 1688
- opeval
 - MODOP, 1647, 1804
- OppositeMonogenicLinearOperator, 1806
- option
 - DROPT, 608
 - GOPT, 1104
- option?
 - DROPT, 608
- optional?
 - PATTERN, 1928
- options
 - VIEW2d, 2770
 - VIEW3D, 2710
- optpair
 - PATTERN, 1928
- OPTPROB, 1753
 - ?=?, 1753
 - ?~=?, 1753
 - coerce, 1753
 - hash, 1753
 - latex, 1753
 - retract, 1753
- OR
 - SWITCH, 2629
- Or
 - IBITS, 1199
 - SINT, 2412
- orbit
 - AFFPLPS, 7
 - AFFSP, 9
 - PERM, 1949
 - PERMGRP, 1960
 - PROJPL, 2117
 - PROJPLPS, 2119
 - PROJSP, 2122
- orbits
 - PERMGRP, 1960
- ord
 - CHAR, 373
- ORDCOMP, 1809
 - , 1810
 - ?<?, 1810
 - ?<=?, 1810
 - ?>?, 1810
 - ?>=?, 1810
 - ?**?, 1810
 - ?*?, 1810
 - ?+?, 1810
 - ?-?, 1810
 - ?=?, 1810
 - ?^?, 1810
 - ?~=?, 1810
 - 0, 1810
 - 1, 1810
 - abs, 1810
 - characteristic, 1810
 - coerce, 1810
 - finite?, 1810
 - hash, 1810
 - infinite?, 1810
 - latex, 1810
 - max, 1810
 - min, 1810
 - minusInfinity, 1810
 - negative?, 1810
 - one?, 1810
 - plusInfinity, 1810
 - positive?, 1810
 - rational, 1810
 - rational?, 1810
 - rationalIfCan, 1810
 - recip, 1810
 - retract, 1810
 - retractIfCan, 1810
 - sample, 1810
 - sign, 1810
 - subtractIfCan, 1810
 - whatInfinity, 1810
 - zero?, 1810
- order
 - ALGFF, 28
 - BPADIC, 258
 - COMPLEX, 419
 - DFLOAT, 587
 - DSMP, 542
 - EXPUPXS, 740
 - FF, 820
 - FFCG, 825

- FFCGP, [835](#)
- FFCGX, [830](#)
- FFNB, [860](#)
- FFNBP, [872](#)
- FFNBX, [866](#)
- FFP, [851](#)
- FFX, [846](#)
- FLOAT, [908](#)
- GSERIES, [1090](#)
- IFF, [1284](#)
- IPADIC, [1294](#)
- IPF, [1303](#)
- ISUPS, [1311](#)
- ITAYLOR, [1338](#)
- LAUPOL, [1422](#)
- MCMPLX, [1543](#)
- MFLOAT, [1548](#)
- MODMON, [1632](#)
- MYUP, [1695](#)
- NSDPS, [1703](#)
- NSUP, [1729](#)
- ODPOL, [1852](#)
- ODVAR, [1855](#)
- PACOFF, [2136](#)
- PACRAT, [2146](#)
- PADIC, [1881](#)
- PERM, [1949](#)
- PERMGRP, [1960](#)
- PF, [2105](#)
- RADFF, [2194](#)
- SAE, [2400](#)
- SDPOL, [2386](#)
- SDVAR, [2389](#)
- SMTS, [2441](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- TS, [2671](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- OrderedCompletion, [1809](#)
- OrderedDirectProduct, [1816](#)
- OrderedFreeMonoid, [1829](#)
- OrderedVariableList, [1836](#)
- orderIfNegative
 - NSDPS, [1703](#)
- OrderlyDifferentialPolynomial, [1851](#)
- OrderlyDifferentialVariable, [1855](#)
- OrdinaryDifferentialRing, [1858](#)
- OrdinaryWeightedPolynomials, [1861](#)
- OrdSetInts, [1864](#)
- ORESUP, [2492](#)
 - , [2492](#)
 - ?**?, [2492](#)
 - ?*?, [2492](#)
 - ?+?, [2492](#)
 - ?-?, [2492](#)
 - ?=?, [2492](#)
 - ?^?, [2492](#)
 - ?~=?, [2492](#)
 - 0, [2492](#)
 - 1, [2492](#)
 - apply, [2492](#)
 - characteristic, [2492](#)
 - coefficient, [2492](#)
 - coefficients, [2492](#)
 - coerce, [2492](#)
 - content, [2492](#)
 - degree, [2492](#)
 - exquo, [2492](#)
 - hash, [2492](#)
 - latex, [2492](#)
 - leadingCoefficient, [2492](#)
 - leftDivide, [2492](#)
 - leftExactQuotient, [2492](#)
 - leftExtendedGcd, [2492](#)
 - leftGcd, [2492](#)
 - leftLcm, [2492](#)
 - leftQuotient, [2492](#)
 - leftRemainder, [2492](#)
 - minimumDegree, [2492](#)
 - monicLeftDivide, [2492](#)
 - monicRightDivide, [2492](#)
 - monomial, [2492](#)

- one?, [2492](#)
- outputForm, [2492](#)
- primitivePart, [2492](#)
- recip, [2492](#)
- reductum, [2492](#)
- retract, [2492](#)
- retractIfCan, [2492](#)
- rightDivide, [2492](#)
- rightExactQuotient, [2492](#)
- rightExtendedGcd, [2492](#)
- rightGcd, [2492](#)
- rightLcm, [2492](#)
- rightQuotient, [2492](#)
- rightRemainder, [2492](#)
- sample, [2492](#)
- subtractIfCan, [2492](#)
- zero?, [2492](#)
- OREUP, [2872](#)
- ?, [2872](#)
- ?**?, [2872](#)
- ?*?, [2872](#)
- ?+?, [2872](#)
- ?-?, [2872](#)
- ?=?, [2872](#)
- ?^?, [2872](#)
- ?~=?, [2872](#)
- 0, [2872](#)
- 1, [2872](#)
- apply, [2872](#)
- characteristic, [2872](#)
- coefficient, [2872](#)
- coefficients, [2872](#)
- coerce, [2872](#)
- content, [2872](#)
- degree, [2872](#)
- exquo, [2872](#)
- hash, [2872](#)
- latex, [2872](#)
- leadingCoefficient, [2872](#)
- leftDivide, [2872](#)
- leftExactQuotient, [2872](#)
- leftExtendedGcd, [2872](#)
- leftGcd, [2872](#)
- leftLcm, [2872](#)
- leftQuotient, [2872](#)
- leftRemainder, [2872](#)
- minimumDegree, [2872](#)
- monicLeftDivide, [2872](#)
- monicRightDivide, [2872](#)
- monomial, [2872](#)
- one?, [2872](#)
- primitivePart, [2872](#)
- recip, [2872](#)
- reductum, [2872](#)
- retract, [2872](#)
- retractIfCan, [2872](#)
- rightDivide, [2872](#)
- rightExactQuotient, [2872](#)
- rightExtendedGcd, [2872](#)
- rightGcd, [2872](#)
- rightLcm, [2872](#)
- rightQuotient, [2872](#)
- rightRemainder, [2872](#)
- sample, [2872](#)
- subtractIfCan, [2872](#)
- zero?, [2872](#)
- origin
 - AFFPLPS, [7](#)
 - AFFSP, [9](#)
- OSI, [1864](#)
- ?<?, [1864](#)
- ?<=?, [1864](#)
- ?>?, [1864](#)
- ?>=?, [1864](#)
- ?=?, [1864](#)
- ?~=?, [1864](#)
- coerce, [1864](#)
- hash, [1864](#)
- latex, [1864](#)
- max, [1864](#)
- min, [1864](#)
- value, [1864](#)
- outerProduct
 - CDFVEC, [432](#)
 - DFVEC, [605](#)
 - IVECTOR, [1260](#)
 - POINT, [2060](#)
 - VECTOR, [2942](#)
- OUTFORM, [1867](#)
- ?, [1868](#)
- ?<?, [1868](#)
- ?<=?, [1868](#)

- [?>?, 1868](#)
- [?>=?, 1868](#)
- [?***?, 1868](#)
- [?*?, 1868](#)
- [?+?, 1868](#)
- [?-?, 1868](#)
- [?...?, 1868](#)
- [?..?, 1868](#)
- [?/? , 1868](#)
- [?=?, 1868](#)
- [?SEGMENT, 1868](#)
- [?^=?, 1868](#)
- [?~=?, 1868](#)
- [?and?, 1868](#)
- [?div?, 1868](#)
- [?or?, 1868](#)
- [?quo?, 1868](#)
- [?rem?, 1868](#)
- [assign, 1868](#)
- [binomial, 1868](#)
- [blankSeparate, 1868](#)
- [box, 1868](#)
- [brace, 1868](#)
- [bracket, 1868](#)
- [center, 1868](#)
- [coerce, 1868](#)
- [commaSeparate, 1868](#)
- [differentiate, 1868](#)
- [dot, 1868](#)
- [empty, 1868](#)
- [exquo, 1868](#)
- [hash, 1868](#)
- [hconcat, 1868](#)
- [height, 1868](#)
- [hspace, 1868](#)
- [infix, 1868](#)
- [infix?, 1868](#)
- [int, 1868](#)
- [label, 1868](#)
- [latex, 1868](#)
- [left, 1868](#)
- [matrix, 1868](#)
- [message, 1868](#)
- [messagePrint, 1868](#)
- [not?, 1868](#)
- [outputForm, 1868](#)
- [over, 1868](#)
- [overbar, 1868](#)
- [overlabel, 1868](#)
- [paren, 1868](#)
- [pile, 1868](#)
- [postfix, 1868](#)
- [prefix, 1868](#)
- [presub, 1868](#)
- [presuper, 1868](#)
- [prime, 1868](#)
- [print, 1868](#)
- [prod, 1868](#)
- [quote, 1868](#)
- [rarrow, 1868](#)
- [right, 1868](#)
- [root, 1868](#)
- [rspace, 1868](#)
- [scripts, 1868](#)
- [semicolonSeparate, 1868](#)
- [slash, 1868](#)
- [string, 1868](#)
- [sub, 1868](#)
- [subHeight, 1868](#)
- [sum, 1868](#)
- [super, 1868](#)
- [superHeight, 1868](#)
- [supersub, 1868](#)
- [vconcat, 1868](#)
- [vspace, 1868](#)
- [width, 1868](#)
- [zag, 1868](#)
- [outlineRender](#)
 - [VIEW3D, 2710](#)
- [output](#)
 - [STREAM, 2582](#)
- [outputAsFortran](#)
 - [ASP1, 71](#)
 - [ASP10, 75](#)
 - [ASP12, 80](#)
 - [ASP19, 85](#)
 - [ASP20, 93, 98](#)
 - [ASP27, 102](#)
 - [ASP28, 108](#)
 - [ASP29, 114](#)
 - [ASP30, 118](#)
 - [ASP31, 122](#)

- ASP33, [127](#)
- ASP34, [130](#)
- ASP35, [134](#)
- ASP4, [140](#)
- ASP41, [144](#)
- ASP42, [151](#)
- ASP49, [158](#)
- ASP50, [164](#)
- ASP55, [170](#)
- ASP6, [176](#)
- ASP7, [181](#)
- ASP73, [186](#)
- ASP74, [191](#)
- ASP77, [197](#)
- ASP78, [201](#)
- ASP8, [206](#)
- ASP80, [211](#)
- ASP9, [216](#)
- FORTRAN, [956](#)
- SFORT, [2405](#)
- outputFixed
 - FLOAT, [908](#)
- outputFloating
 - FLOAT, [908](#)
- OutputForm, [1867](#)
- outputForm
 - LMOPS, [1509](#)
 - ORESUP, [2492](#)
 - OUTFORM, [1868](#)
 - SUP, [2467](#)
- outputGeneral
 - FLOAT, [908](#)
- outputSpacing
 - FLOAT, [908](#)
- OVAR, [1836](#)
 - ?<?, [1836](#)
 - ?<=?, [1836](#)
 - ?>?, [1836](#)
 - ?>=?, [1836](#)
 - ?=?, [1836](#)
 - ?~=?, [1836](#)
 - coerce, [1836](#)
 - convert, [1836](#)
 - hash, [1836](#)
 - index, [1836](#)
 - latex, [1836](#)
 - lookup, [1836](#)
 - max, [1836](#)
 - min, [1836](#)
 - random, [1836](#)
 - size, [1836](#)
 - variable, [1836](#)
- over
 - OUTFORM, [1868](#)
- overbar
 - OUTFORM, [1868](#)
- overlabel
 - OUTFORM, [1868](#)
- overlap
 - FMONOID, [1021](#)
 - OFMONOID, [1829](#)
- OWP, [1861](#)
 - ?, [1861](#)
 - ?**?, [1861](#)
 - ?*?, [1861](#)
 - ?+?, [1861](#)
 - ?-?, [1861](#)
 - ?/?, [1861](#)
 - ?=?, [1861](#)
 - ?^?, [1861](#)
 - ?~=?, [1861](#)
 - 0, [1861](#)
 - 1, [1861](#)
 - changeWeightLevel, [1861](#)
 - characteristic, [1861](#)
 - coerce, [1861](#)
 - hash, [1861](#)
 - latex, [1861](#)
 - one?, [1861](#)
 - recip, [1861](#)
 - sample, [1861](#)
 - subtractIfCan, [1861](#)
 - zero?, [1861](#)
- PACEXT, [2126](#)
- PACOFF, [2135](#)
 - ?, [2136](#)
 - ?**?, [2136](#)
 - ?*?, [2136](#)
 - ?+?, [2136](#)
 - ?-?, [2136](#)
 - ?/?, [2136](#)

- [?=?, 2136](#)
- [?^?, 2136](#)
- [?~=?, 2136](#)
- [?quo?, 2136](#)
- [?rem?, 2136](#)
- [0, 2136](#)
- [1, 2136](#)
- [algebraic?, 2136](#)
- [associates?, 2136](#)
- [characteristic, 2136](#)
- [charthRoot, 2136](#)
- [coerce, 2136](#)
- [conditionP, 2136](#)
- [conjugate, 2136](#)
- [createPrimitiveElement, 2136](#)
- [D, 2136](#)
- [definingPolynomial, 2136](#)
- [degree, 2136](#)
- [differentiate, 2136](#)
- [dimension, 2136](#)
- [discreteLog, 2136](#)
- [distinguishedRootsOf, 2136](#)
- [divide, 2136](#)
- [euclideanSize, 2136](#)
- [expressIdealMember, 2136](#)
- [exquo, 2136](#)
- [extDegree, 2136](#)
- [extendedEuclidean, 2136](#)
- [extensionDegree, 2136](#)
- [factor, 2136](#)
- [factorsOfCyclicGroupSize, 2136](#)
- [Frobenius, 2136](#)
- [fullOutput, 2136](#)
- [gcd, 2136](#)
- [gcdPolynomial, 2136](#)
- [ground?, 2136](#)
- [hash, 2136](#)
- [index, 2136](#)
- [inGroundField?, 2136](#)
- [init, 2136](#)
- [inv, 2136](#)
- [latex, 2136](#)
- [lcm, 2136](#)
- [lift, 2136](#)
- [lookup, 2136](#)
- [maxTower, 2136](#)
- [multiEuclidean, 2136](#)
- [newElement, 2136](#)
- [nextItem, 2136](#)
- [one?, 2136](#)
- [order, 2136](#)
- [previousTower, 2136](#)
- [prime?, 2136](#)
- [primeFrobenius, 2136](#)
- [primitive?, 2136](#)
- [primitiveElement, 2136](#)
- [principalIdeal, 2136](#)
- [random, 2136](#)
- [recip, 2136](#)
- [reduce, 2136](#)
- [representationType, 2136](#)
- [retract, 2136](#)
- [retractIfCan, 2136](#)
- [sample, 2136](#)
- [setTower, 2136](#)
- [size, 2136](#)
- [sizeLess?, 2136](#)
- [squareFree, 2136](#)
- [squareFreePart, 2136](#)
- [subtractIfCan, 2136](#)
- [tableForDiscreteLogarithm, 2136](#)
- [transcendenceDegree, 2136](#)
- [transcendent?, 2136](#)
- [unit?, 2136](#)
- [unitCanonical, 2136](#)
- [unitNormal, 2136](#)
- [vectorise, 2136](#)
- [zero?, 2136](#)
- [PACRAT, 2146](#)
- [-?, 2146](#)
- [?**, 2146](#)
- [?*?, 2146](#)
- [?+?, 2146](#)
- [?-?, 2146](#)
- [?/? , 2146](#)
- [?=?, 2146](#)
- [?^?, 2146](#)
- [?~=?, 2146](#)
- [?quo?, 2146](#)
- [?rem?, 2146](#)
- [0, 2146](#)
- [1, 2146](#)

- algebraic?, 2146
- associates?, 2146
- characteristic, 2146
- charthRoot, 2146
- coerce, 2146
- conjugate, 2146
- definingPolynomial, 2146
- degree, 2146
- dimension, 2146
- discreteLog, 2146
- distinguishedRootsOf, 2146
- divide, 2146
- euclideanSize, 2146
- expressIdealMember, 2146
- exquo, 2146
- extDegree, 2146
- extendedEuclidean, 2146
- extensionDegree, 2146
- factor, 2146
- Frobenius, 2146
- fullOutput, 2146
- gcd, 2146
- gcdPolynomial, 2146
- ground?, 2146
- hash, 2146
- inGroundField?, 2146
- inv, 2146
- latex, 2146
- lcm, 2146
- lift, 2146
- maxTower, 2146
- multiEuclidean, 2146
- newElement, 2146
- one?, 2146
- order, 2146
- previousTower, 2146
- prime?, 2146
- primeFrobenius, 2146
- principalIdeal, 2146
- recip, 2146
- reduce, 2146
- retract, 2146
- retractIfCan, 2146
- sample, 2146
- setTower, 2146
- sizeLess?, 2146
- squareFree, 2146
- squareFreePart, 2146
- subtractIfCan, 2146
- transcendenceDegree, 2146
- transcendent?, 2146
- unit?, 2146
- unitCanonical, 2146
- unitNormal, 2146
- vectorise, 2146
- zero?, 2146
- PADIC, 1881
 - ?, 1881
 - ?**?, 1881
 - ?*?, 1881
 - ?+?, 1881
 - ?-?, 1881
 - ?=?, 1881
 - ?^?, 1881
 - ?~=?, 1881
 - ?quo?, 1881
 - ?rem?, 1881
 - 0, 1881
 - 1, 1881
 - approximate, 1881
 - associates?, 1881
 - characteristic, 1881
 - coerce, 1881
 - complete, 1881
 - digits, 1881
 - divide, 1881
 - euclideanSize, 1881
 - expressIdealMember, 1881
 - exquo, 1881
 - extend, 1881
 - extendedEuclidean, 1881
 - gcd, 1881
 - gcdPolynomial, 1881
 - hash, 1881
 - latex, 1881
 - lcm, 1881
 - moduloP, 1881
 - modulus, 1881
 - multiEuclidean, 1881
 - one?, 1881
 - order, 1881
 - principalIdeal, 1881

- quotientByP, 1881
- recip, 1881
- root, 1881
- sample, 1881
- sizeLess?, 1881
- sqrt, 1881
- subtractIfCan, 1881
- unit?, 1881
- unitCanonical, 1881
- unitNormal, 1881
- zero?, 1881
- padicallyExpand
 - PFR, 1914
- padicFraction
 - PFR, 1914
- PAdicInteger, 1881
- PADICRAT, 1885
 - , 1886
 - ?<?, 1886
 - ?<=?, 1886
 - ?>?, 1886
 - ?>=?, 1886
 - ?**?, 1886
 - ?*?, 1886
 - ?+?, 1886
 - ?-?, 1886
 - ?., 1886
 - ?/? , 1886
 - ?=?, 1886
 - ?^?, 1886
 - ?~=?, 1886
 - ?quo?, 1886
 - ?rem?, 1886
 - 0, 1886
 - 1, 1886
 - abs, 1886
 - approximate, 1886
 - associates?, 1886
 - ceiling, 1886
 - characteristic, 1886
 - charthRoot, 1886
 - coerce, 1886
 - conditionP, 1886
 - continuedFraction, 1886
 - convert, 1886
 - D, 1886
 - denom, 1886
 - denominator, 1886
 - differentiate, 1886
 - divide, 1886
 - euclideanSize, 1886
 - eval, 1886
 - expressIdealMember, 1886
 - exquo, 1886
 - extendedEuclidean, 1886
 - factor, 1886
 - factorPolynomial, 1886
 - factorSquareFreePolynomial, 1886
 - floor, 1886
 - fractionPart, 1886
 - gcd, 1886
 - gcdPolynomial, 1886
 - hash, 1886
 - init, 1886
 - inv, 1886
 - latex, 1886
 - lcm, 1886
 - map, 1886
 - max, 1886
 - min, 1886
 - multiEuclidean, 1886
 - negative?, 1886
 - nextItem, 1886
 - numer, 1886
 - numerator, 1886
 - one?, 1886
 - patternMatch, 1886
 - positive?, 1886
 - prime?, 1886
 - principalIdeal, 1886
 - random, 1886
 - recip, 1886
 - reducedSystem, 1886
 - removeZeroes, 1886
 - retract, 1886
 - retractIfCan, 1886
 - sample, 1886
 - sign, 1886
 - sizeLess?, 1886
 - solveLinearPolynomialEquation, 1886
 - squareFree, 1886
 - squareFreePart, 1886

- squareFreePolynomial, 1886
- subtractIfCan, 1886
- unit?, 1886
- unitCanonical, 1886
- unitNormal, 1886
- wholePart, 1886
- zero?, 1886
- PAdicRational, 1885
- PAdicRationalConstructor, 1890
- PADICRC, 1890
- ?, 1891
- ?<?, 1891
- ?<=?, 1891
- ?>?, 1891
- ?>=?, 1891
- ?**?, 1891
- ?*?, 1891
- ?+?, 1891
- ?-?, 1891
- ?., 1891
- ?/?, 1891
- ?=?, 1891
- ?^?, 1891
- ?~=?, 1891
- ?quo?, 1891
- ?rem?, 1891
- 0, 1891
- 1, 1891
- abs, 1891
- approximate, 1891
- associates?, 1891
- ceiling, 1891
- characteristic, 1891
- charthRoot, 1891
- coerce, 1891
- conditionP, 1891
- continuedFraction, 1891
- convert, 1891
- D, 1891
- denom, 1891
- denominator, 1891
- differentiate, 1891
- divide, 1891
- euclideanSize, 1891
- eval, 1891
- expressIdealMember, 1891
- exquo, 1891
- extendedEuclidean, 1891
- factor, 1891
- factorPolynomial, 1891
- factorSquareFreePolynomial, 1891
- floor, 1891
- fractionPart, 1891
- gcd, 1891
- gcdPolynomial, 1891
- hash, 1891
- init, 1891
- inv, 1891
- latex, 1891
- lcm, 1891
- map, 1891
- max, 1891
- min, 1891
- multiEuclidean, 1891
- negative?, 1891
- nextItem, 1891
- numer, 1891
- numerator, 1891
- one?, 1891
- patternMatch, 1891
- positive?, 1891
- prime?, 1891
- principalIdeal, 1891
- random, 1891
- recip, 1891
- reducedSystem, 1891
- removeZeroes, 1891
- retract, 1891
- retractIfCan, 1891
- sample, 1891
- sign, 1891
- sizeLess?, 1891
- solveLinearPolynomialEquation, 1891
- squareFree, 1891
- squareFreePart, 1891
- squareFreePolynomial, 1891
- subtractIfCan, 1891
- unit?, 1891
- unitCanonical, 1891
- unitNormal, 1891
- wholePart, 1891
- zero?, 1891

- pair?
 - INFORM, [1343](#)
 - SEX, [2392](#)
 - SEXOF, [2394](#)
- PALETTE, [1896](#)
 - ?=?, [1896](#)
 - ?~=?, [1896](#)
 - bright, [1896](#)
 - coerce, [1896](#)
 - dark, [1896](#)
 - dim, [1896](#)
 - hash, [1896](#)
 - hue, [1896](#)
 - latex, [1896](#)
 - light, [1896](#)
 - pastel, [1896](#)
 - shade, [1896](#)
- Palette, [1896](#)
- parametersOf
 - SYMTAB, [2648](#)
- parametric?
 - PLOT, [2029](#)
- ParametricPlaneCurve, [1899](#)
- ParametricSpaceCurve, [1901](#)
- ParametricSurface, [1904](#)
- paren
 - AN, [35](#)
 - EXPR, [724](#)
 - FEXPR, [947](#)
 - IAN, [1276](#)
 - MYEXPR, [1688](#)
 - OUTFORM, [1868](#)
- parent
 - SUBSPACE, [2614](#)
- PARPCURV, [1899](#)
 - coordinate, [1899](#)
 - curve, [1899](#)
- PARSCURV, [1901](#)
 - coordinate, [1901](#)
 - curve, [1901](#)
- parse
 - INFORM, [1343](#)
- PARSURF, [1904](#)
 - coordinate, [1904](#)
 - surface, [1904](#)
- partialDenominators
 - CONTFRAC, [445](#)
- PartialFraction, [1913](#)
- partialFraction
 - PFR, [1914](#)
- partialNumerators
 - CONTFRAC, [445](#)
- partialQuotients
 - CONTFRAC, [445](#)
- Partition, [1923](#)
- partition
 - PRITION, [1923](#)
- parts
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - ARRAY2, [2763](#)
 - ASTACK, [65](#)
 - BBTREE, [253](#)
 - BITS, [315](#)
 - BSTREE, [302](#)
 - BTOURN, [306](#)
 - BTREE, [310](#)
 - CCLASS, [382](#)
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)
 - DEQUEUE, [512](#)
 - DFMAT, [599](#)
 - DFVEC, [605](#)
 - DHMATRIX, [491](#)
 - DIRPROD, [547](#)
 - DLIST, [460](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DSTREE, [535](#)
 - EQTBL, [699](#)
 - FARRAY, [886](#)
 - GPOLSET, [1074](#)
 - GSTBL, [1079](#)
 - GTSET, [1083](#)
 - HASHTBL, [1120](#)
 - HDP, [1173](#)
 - HEAP, [1134](#)
 - IARRAY1, [1244](#)
 - IARRAY2, [1256](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - IIARRAY2, [1290](#)

- ILIST, [1231](#)
- IMATRIX, [1239](#)
- INTABL, [1336](#)
- ISTRING, [1249](#)
- IVECTOR, [1260](#)
- KAFILE, [1414](#)
- LIB, [1429](#)
- LIST, [1504](#)
- LMDICT, [1515](#)
- LSQM, [1456](#)
- M3D, [2703](#)
- MATRIX, [1623](#)
- MSET, [1670](#)
- NSDPS, [1703](#)
- ODP, [1816](#)
- PENDTREE, [1945](#)
- POINT, [2060](#)
- PRIMARR, [2109](#)
- QUEUE, [2184](#)
- REGSET, [2286](#)
- RESULT, [2301](#)
- RGCHAIN, [2255](#)
- RMATRIX, [2246](#)
- ROUTINE, [2331](#)
- SET, [2372](#)
- SHDP, [2509](#)
- SPLTREE, [2518](#)
- SQMATRIX, [2548](#)
- SREGSET, [2535](#)
- STACK, [2563](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- TREE, [2741](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- WUTSET, [2959](#)
- pastel
 - PALETTE, [1896](#)
- PATLRES, [1937](#)
 - ?=?, [1937](#)
 - ?~=?, [1937](#)
 - atoms, [1937](#)
 - coerce, [1937](#)
 - failed, [1937](#)
 - failed?, [1937](#)
 - hash, [1937](#)
 - latex, [1937](#)
 - lists, [1937](#)
 - makeResult, [1937](#)
 - new, [1937](#)
- PATRES, [1940](#)
 - ?=?, [1940](#)
 - ?~=?, [1940](#)
 - addMatch, [1940](#)
 - addMatchRestricted, [1940](#)
 - coerce, [1940](#)
 - construct, [1940](#)
 - destruct, [1940](#)
 - failed, [1940](#)
 - failed?, [1940](#)
 - getMatch, [1940](#)
 - hash, [1940](#)
 - insertMatch, [1940](#)
 - latex, [1940](#)
 - new, [1940](#)
 - satisfy?, [1940](#)
 - union, [1940](#)
- PATTERN, [1928](#)
 - ?**?, [1928](#)
 - ?*?, [1928](#)
 - ?+?, [1928](#)
 - ?/?, [1928](#)
 - ?=?, [1928](#)
 - ?~=?, [1928](#)
 - 0, [1928](#)
 - 1, [1928](#)
 - addBadValue, [1928](#)
 - coerce, [1928](#)
 - constant?, [1928](#)
 - convert, [1928](#)
 - copy, [1928](#)
 - depth, [1928](#)
 - elt, [1928](#)
 - generic?, [1928](#)

- getBadValues, 1928
- hash, 1928
- hasPredicate?, 1928
- hasTopPredicate?, 1928
- inR?, 1928
- isExpt, 1928
- isList, 1928
- isOp, 1928
- isPlus, 1928
- isPower, 1928
- isQuotient, 1928
- isTimes, 1928
- latex, 1928
- multiple?, 1928
- optional?, 1928
- optpair, 1928
- patternVariable, 1928
- predicates, 1928
- quoted?, 1928
- resetBadValues, 1928
- retract, 1928
- retractIfCan, 1928
- setPredicates, 1928
- setTopPredicate, 1928
- symbol?, 1928
- topPredicate, 1928
- variables, 1928
- withPredicates, 1928
- Pattern, 1928
- pattern
 - RULE, 2305
- patternMatch
 - BINARY, 292
 - BPADICRT, 263
 - COMPLEX, 419
 - DECIMAL, 465
 - DFLOAT, 587
 - DMP, 572
 - DSMP, 542
 - EXPEXPAN, 712
 - EXPR, 724
 - FLOAT, 908
 - FRAC, 986
 - GDMP, 1052
 - HDMP, 1180
 - HEXADEC, 1143
 - INT, 1362
 - MCMPLX, 1543
 - MFLOAT, 1548
 - MINT, 1557
 - MODMON, 1632
 - MPOLY, 1682
 - MYEXPR, 1688
 - MYUP, 1695
 - NSMP, 1714
 - NSUP, 1729
 - ODPOL, 1852
 - PADICRAT, 1886
 - PADICRC, 1891
 - POLY, 2078
 - RADIX, 2206
 - ROMAN, 2326
 - SDPOL, 2386
 - SINT, 2412
 - SMP, 2423
 - SULS, 2457
 - SUP, 2467
 - SUPEXP, 2481
 - SYMBOL, 2640
 - ULS, 2796
 - ULSCONS, 2804
 - UP, 2828
- PatternMatchListResult, 1937
- PatternMatchResult, 1940
- patternVariable
 - PATTERN, 1928
- PBWL, 2054
 - ?<?, 2054
 - ?<=?, 2054
 - ?>?, 2054
 - ?>=?, 2054
 - ?=?, 2054
 - ?~=?, 2054
 - 1, 2054
 - coerce, 2054
 - first, 2054
 - hash, 2054
 - latex, 2054
 - length, 2054
 - listOfTerms, 2054
 - max, 2054
 - min, 2054

- rest, 2054
- retract, 2054
- retractable?, 2054
- retractIfCan, 2054
- varList, 2054
- pdct
 - PRITION, 1923
- PDEPROB, 1756
 - ?=, 1756
 - ?~=, 1756
 - coerce, 1756
 - hash, 1756
 - latex, 1756
 - retract, 1756
- PDESolve
 - D03EEFA, 665
 - D03FAFA, 668
- PendantTree, 1944
- PENDTREE, 1944
 - ?.left, 1945
 - ?.right, 1945
 - ?.value, 1945
 - ?=, 1945
 - ?~=, 1945
 - #?, 1945
 - any?, 1945
 - child?, 1945
 - children, 1945
 - coerce, 1945
 - copy, 1945
 - count, 1945
 - cyclic?, 1945
 - distance, 1945
 - empty, 1945
 - empty?, 1945
 - eq?, 1945
 - eval, 1945
 - every?, 1945
 - hash, 1945
 - latex, 1945
 - leaf?, 1945
 - leaves, 1945
 - left, 1945
 - less?, 1945
 - map, 1945
 - member?, 1945
 - members, 1945
 - more?, 1945
 - node?, 1945
 - nodes, 1945
 - parts, 1945
 - ptree, 1945
 - right, 1945
 - sample, 1945
 - setchildren, 1945
 - setelt, 1945
 - setleft, 1945
 - setright, 1945
 - setvalue, 1945
 - size?, 1945
 - value, 1945
- PERM, 1949
 - ?<?, 1949
 - ?<=, 1949
 - ?>?, 1949
 - ?>=, 1949
 - ?**?, 1949
 - ?*?, 1949
 - ?.?, 1949
 - ?/? , 1949
 - ?=, 1949
 - ?^?, 1949
 - ?~=, 1949
 - 1, 1949
 - coerce, 1949
 - coerceImages, 1949
 - coerceListOfPairs, 1949
 - coercePreimagesImages, 1949
 - commutator, 1949
 - conjugate, 1949
 - cycle, 1949
 - cyclePartition, 1949
 - cycles, 1949
 - degree, 1949
 - eval, 1949
 - even?, 1949
 - fixedPoints, 1949
 - hash, 1949
 - inv, 1949
 - latex, 1949
 - listRepresentation, 1949
 - max, 1949

- min, 1949
- movedPoints, 1949
- numberOfCycles, 1949
- odd?, 1949
- one?, 1949
- orbit, 1949
- order, 1949
- recip, 1949
- sample, 1949
- sign, 1949
- sort, 1949
- PERMGRP, 1959
 - ?<?, 1960
 - ?<=?, 1960
 - ?.?, 1960
 - ?=?, 1960
 - ?~=?, 1960
 - base, 1960
 - coerce, 1960
 - degree, 1960
 - generators, 1960
 - hash, 1960
 - initializeGroupForWordProblem, 1960
 - latex, 1960
 - member?, 1960
 - movedPoints, 1960
 - orbit, 1960
 - orbits, 1960
 - order, 1960
 - permutationGroup, 1960
 - random, 1960
 - strongGenerators, 1960
 - wordInGenerators, 1960
 - wordInStrongGenerators, 1960
 - wordsForStrongGenerators, 1960
- Permutation, 1949
- permutation
 - EXPR, 724
 - INT, 1362
 - MINT, 1557
 - MYEXPR, 1688
 - ROMAN, 2326
 - SINT, 2412
- PermutationGroup, 1959
- permutationGroup
 - PERMGRP, 1960
- perspective
 - VIEW3D, 2710
- PF, 2104
 - ?, 2105
 - ?**?, 2105
 - ?*?, 2105
 - ?+?, 2105
 - ?-?, 2105
 - ?/?, 2105
 - ?=?, 2105
 - ?^?, 2105
 - ?~=?, 2105
 - ?quo?, 2105
 - ?rem?, 2105
 - 0, 2105
 - 1, 2105
 - algebraic?, 2105
 - associates?, 2105
 - basis, 2105
 - characteristic, 2105
 - charthRoot, 2105
 - coerce, 2105
 - conditionP, 2105
 - convert, 2105
 - coordinates, 2105
 - createNormalElement, 2105
 - createPrimitiveElement, 2105
 - D, 2105
 - definingPolynomial, 2105
 - degree, 2105
 - differentiate, 2105
 - dimension, 2105
 - discreteLog, 2105
 - divide, 2105
 - euclideanSize, 2105
 - expressIdealMember, 2105
 - exquo, 2105
 - extendedEuclidean, 2105
 - extensionDegree, 2105
 - factor, 2105
 - factorsOfCyclicGroupSize, 2105
 - Frobenius, 2105
 - gcd, 2105
 - gcdPolynomial, 2105
 - generator, 2105
 - hash, 2105

- index, 2105
- inGroundField?, 2105
- init, 2105
- inv, 2105
- latex, 2105
- lcm, 2105
- linearAssociatedExp, 2105
- linearAssociatedLog, 2105
- linearAssociatedOrder, 2105
- lookup, 2105
- minimalPolynomial, 2105
- multiEuclidean, 2105
- nextItem, 2105
- norm, 2105
- normal?, 2105
- normalElement, 2105
- one?, 2105
- order, 2105
- prime?, 2105
- primeFrobenius, 2105
- primitive?, 2105
- primitiveElement, 2105
- principalIdeal, 2105
- random, 2105
- recip, 2105
- representationType, 2105
- represents, 2105
- retract, 2105
- retractIfCan, 2105
- sample, 2105
- size, 2105
- sizeLess?, 2105
- squareFree, 2105
- squareFreePart, 2105
- subtractIfCan, 2105
- tableForDiscreteLogarithm, 2105
- trace, 2105
- transcendenceDegree, 2105
- transcendent?, 2105
- unit?, 2105
- unitCanonical, 2105
- unitNormal, 2105
- zero?, 2105
- pfaffian
 - CDFMAT, 426
 - DFMAT, 599
 - U16MAT, 2906
 - U32MAT, 2911
 - U8MAT, 2902
- PFR, 1913
 - , 1914
 - ?**?, 1914
 - ?*?, 1914
 - ?+?, 1914
 - ?-?, 1914
 - ?/?, 1914
 - ?=?, 1914
 - ?^?, 1914
 - ?~=?, 1914
 - ?quo?, 1914
 - ?rem?, 1914
 - 0, 1914
 - 1, 1914
 - associates?, 1914
 - characteristic, 1914
 - coerce, 1914
 - compactFraction, 1914
 - divide, 1914
 - euclideanSize, 1914
 - expressIdealMember, 1914
 - exquo, 1914
 - extendedEuclidean, 1914
 - factor, 1914
 - firstDenom, 1914
 - firstNumer, 1914
 - gcd, 1914
 - gcdPolynomial, 1914
 - hash, 1914
 - inv, 1914
 - latex, 1914
 - lcm, 1914
 - multiEuclidean, 1914
 - nthFractionalTerm, 1914
 - numberOfFractionalTerms, 1914
 - one?, 1914
 - padicallyExpand, 1914
 - padicFraction, 1914
 - partialFraction, 1914
 - prime?, 1914
 - principalIdeal, 1914
 - recip, 1914
 - sample, 1914

- sizeLess?, 1914
- squareFree, 1914
- squareFreePart, 1914
- subtractIfCan, 1914
- unit?, 1914
- unitCanonical, 1914
- unitNormal, 1914
- wholePart, 1914
- zero?, 1914
- physicalLength
 - FARRAY, 886
 - IFARRAY, 1222
- PI, 2100
 - ?<?, 2100
 - ?<=?, 2100
 - ?>?, 2100
 - ?>=?, 2100
 - ?**?, 2100
 - ?*?, 2100
 - ?+?, 2100
 - ?=?, 2100
 - ?^?, 2100
 - ?~=?, 2100
 - 1, 2100
 - coerce, 2100
 - gcd, 2100
 - hash, 2100
 - latex, 2100
 - max, 2100
 - min, 2100
 - one?, 2100
 - recip, 2100
 - sample, 2100
- Pi, 1977
- pi
 - COMPLEX, 419
 - DFLOAT, 587
 - EXPR, 724
 - EXPUPXS, 740
 - FEXPR, 947
 - FLOAT, 908
 - GSERIES, 1090
 - HACKPI, 1978
 - INTRVL, 1384
 - MCMPLEX, 1543
 - SMTS, 2441
 - SULS, 2457
 - SUPEXPR, 2481
 - SUPXS, 2487
 - SUTS, 2497
 - TS, 2671
 - UFPS, 2789
 - ULS, 2796
 - ULSCONS, 2804
 - UPXS, 2834
 - UPXSCONS, 2842
 - UTS, 2877
 - UTSZ, 2887
- pile
 - OUTFORM, 1868
- PLACES, 2019
 - .?, 2019
 - ?*?, 2019
 - ?+?, 2019
 - ?-?, 2019
 - ?., 2019
 - ?=?, 2019
 - ?~=?, 2019
 - coerce, 2019
 - create, 2019
 - degree, 2019
 - foundPlaces, 2019
 - hash, 2019
 - itsALeaf, 2019
 - latex, 2019
 - leaf?, 2019
 - localParam, 2019
 - reduce, 2019
 - setDegree, 2019
 - setFoundPlacesToEmpty, 2019
 - setParam, 2019
- Places, 2019
- PlacesOverPseudoAlgebraicClosureOfFiniteField, 2021
- PLACESPS, 2021
 - .?, 2021
 - ?*?, 2021
 - ?+?, 2021
 - ?-?, 2021
 - ?., 2021
 - ?=?, 2021
 - ?~=?, 2021

- coerce, [2021](#)
- create, [2021](#)
- degree, [2021](#)
- foundPlaces, [2021](#)
- hash, [2021](#)
- itsALeaf, [2021](#)
- latex, [2021](#)
- leaf?, [2021](#)
- localParam, [2021](#)
- reduce, [2021](#)
- setDegree, [2021](#)
- setFoundPlacesToEmpty, [2021](#)
- setParam, [2021](#)
- PlaneAlgebraicCurvePlot, [1992](#)
- PLCS, [2023](#)
- Plcs, [2023](#)
- plenaryPower
 - ALGSC, [15](#)
 - FNLA, [1027](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- PLOT, [2029](#)
 - adaptive?, [2029](#)
 - coerce, [2029](#)
 - debug, [2029](#)
 - listBranches, [2029](#)
 - maxPoints, [2029](#)
 - minPoints, [2029](#)
 - numFunEvals, [2029](#)
 - parametric?, [2029](#)
 - plot, [2029](#)
 - plotPolar, [2029](#)
 - pointPlot, [2029](#)
 - refine, [2029](#)
 - screenResolution, [2029](#)
 - setAdaptive, [2029](#)
 - setMaxPoints, [2029](#)
 - setMinPoints, [2029](#)
 - setScreenResolution, [2029](#)
 - tRange, [2029](#)
 - xRange, [2029](#)
 - yRange, [2029](#)
 - zoom, [2029](#)
- Plot, [2029](#)
- plot
 - PLOT, [2029](#)
 - PLOT3D, [2042](#)
- PLOT3D, [2042](#)
 - adaptive3D?, [2042](#)
 - coerce, [2042](#)
 - debug3D, [2042](#)
 - listBranches, [2042](#)
 - maxPoints3D, [2042](#)
 - minPoints3D, [2042](#)
 - numFunEvals3D, [2042](#)
 - plot, [2042](#)
 - pointPlot, [2042](#)
 - refine, [2042](#)
 - screenResolution3D, [2042](#)
 - setAdaptive3D, [2042](#)
 - setMaxPoints3D, [2042](#)
 - setMinPoints3D, [2042](#)
 - setScreenResolution3D, [2042](#)
 - tRange, [2042](#)
 - tValues, [2042](#)
 - xRange, [2042](#)
 - yRange, [2042](#)
 - zoom, [2042](#)
 - zRange, [2042](#)
- Plot3D, [2042](#)
- plotPolar
 - PLOT, [2029](#)
- plus
 - LMOPS, [1509](#)
 - M3D, [2703](#)
- plusInfinity
 - ORDCOMP, [1810](#)
- po
 - OMLO, [1806](#)
- PoincareBirkhoffWittLyndonBasis, [2054](#)
- POINT, [2059](#)
 - , [2060](#)
 - ?<?, [2060](#)
 - ?<=?, [2060](#)
 - ?>?, [2060](#)
 - ?>=?, [2060](#)
 - ?*?, [2060](#)
 - ?+?, [2060](#)
 - ?-?, [2060](#)
 - ?..?, [2060](#)

- [?=?, 2060](#)
- [?~=?, 2060](#)
- [#?, 2060](#)
- [any?, 2060](#)
- [coerce, 2060](#)
- [concat, 2060](#)
- [construct, 2060](#)
- [convert, 2060](#)
- [copy, 2060](#)
- [copyInto, 2060](#)
- [count, 2060](#)
- [cross, 2060](#)
- [delete, 2060](#)
- [dimension, 2060](#)
- [dot, 2060](#)
- [elt, 2060](#)
- [empty, 2060](#)
- [empty?, 2060](#)
- [entries, 2060](#)
- [entry?, 2060](#)
- [eq?, 2060](#)
- [eval, 2060](#)
- [every?, 2060](#)
- [extend, 2060](#)
- [fill, 2060](#)
- [find, 2060](#)
- [first, 2060](#)
- [hash, 2060](#)
- [index?, 2060](#)
- [indices, 2060](#)
- [insert, 2060](#)
- [latex, 2060](#)
- [length, 2060](#)
- [less?, 2060](#)
- [magnitude, 2060](#)
- [map, 2060](#)
- [max, 2060](#)
- [maxIndex, 2060](#)
- [member?, 2060](#)
- [members, 2060](#)
- [merge, 2060](#)
- [min, 2060](#)
- [minIndex, 2060](#)
- [more?, 2060](#)
- [new, 2060](#)
- [outerProduct, 2060](#)
- [parts, 2060](#)
- [point, 2060](#)
- [position, 2060](#)
- [qelt, 2060](#)
- [qsetelt, 2060](#)
- [reduce, 2060](#)
- [remove, 2060](#)
- [removeDuplicates, 2060](#)
- [reverse, 2060](#)
- [sample, 2060](#)
- [select, 2060](#)
- [setelt, 2060](#)
- [size?, 2060](#)
- [sort, 2060](#)
- [sorted?, 2060](#)
- [swap, 2060](#)
- [zero, 2060](#)
- [Point, 2059](#)
- [point](#)
 - [GRIMAGE, 1094](#)
 - [POINT, 2060](#)
 - [SPACE3, 2731](#)
- [point?](#)
 - [SPACE3, 2731](#)
- [pointColor](#)
 - [DROPT, 608](#)
- [pointData](#)
 - [SUBSPACE, 2614](#)
- [pointLists](#)
 - [GRIMAGE, 1094](#)
- [pointPlot](#)
 - [PLOT, 2029](#)
 - [PLOT3D, 2042](#)
- [points](#)
 - [VIEW2d, 2770](#)
- [pointV](#)
 - [IC, 1191](#)
 - [INFCLSPS, 1271](#)
 - [INFCLSPT, 1265](#)
- [pointValue](#)
 - [AFFPLPS, 7](#)
 - [AFFSP, 9](#)
 - [PROJPL, 2117](#)
 - [PROJPLPS, 2119](#)
 - [PROJSP, 2122](#)
- [polarCoordinates](#)

- COMPLEX, [419](#)
- MCMPLX, [1543](#)
- pole?
 - EXPUPXS, [740](#)
 - GSERIES, [1090](#)
 - ISUPS, [1311](#)
 - ITAYLOR, [1338](#)
 - NSDPS, [1703](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- POLY, [2077](#)
 - , [2078](#)
 - ?<?, [2078](#)
 - ?<=?, [2078](#)
 - ?>?, [2078](#)
 - ?>=?, [2078](#)
 - ?**?, [2078](#)
 - ?*?, [2078](#)
 - ?+?, [2078](#)
 - ?-?, [2078](#)
 - ?/?, [2078](#)
 - ?=?, [2078](#)
 - ?^?, [2078](#)
 - ?~=?, [2078](#)
 - 0, [2078](#)
 - 1, [2078](#)
 - associates?, [2078](#)
 - binomThmExpt, [2078](#)
 - characteristic, [2078](#)
 - charthRoot, [2078](#)
 - coefficient, [2078](#)
 - coefficients, [2078](#)
 - coerce, [2078](#)
 - conditionP, [2078](#)
 - content, [2078](#)
 - convert, [2078](#)
 - D, [2078](#)
 - degree, [2078](#)
 - differentiate, [2078](#)
 - discriminant, [2078](#)
 - eval, [2078](#)
 - exquo, [2078](#)
 - factor, [2078](#)
 - factorPolynomial, [2078](#)
 - factorSquareFreePolynomial, [2078](#)
 - gcd, [2078](#)
 - gcdPolynomial, [2078](#)
 - ground, [2078](#)
 - ground?, [2078](#)
 - hash, [2078](#)
 - integrate, [2078](#)
 - isExpt, [2078](#)
 - isPlus, [2078](#)
 - isTimes, [2078](#)
 - latex, [2078](#)
 - lcm, [2078](#)
 - leadingCoefficient, [2078](#)
 - leadingMonomial, [2078](#)
 - mainVariable, [2078](#)
 - map, [2078](#)
 - mapExponents, [2078](#)
 - max, [2078](#)
 - min, [2078](#)
 - minimumDegree, [2078](#)
 - monicDivide, [2078](#)
 - monomial, [2078](#)
 - monomial?, [2078](#)
 - monomials, [2078](#)
 - multivariate, [2078](#)
 - numberOfMonomials, [2078](#)
 - one?, [2078](#)
 - patternMatch, [2078](#)
 - pomopo, [2078](#)
 - prime?, [2078](#)
 - primitiveMonomials, [2078](#)
 - primitivePart, [2078](#)
 - recip, [2078](#)
 - reducedSystem, [2078](#)
 - reductum, [2078](#)
 - resultant, [2078](#)
 - retract, [2078](#)
 - retractIfCan, [2078](#)

- sample, [2078](#)
- solveLinearPolynomialEquation, [2078](#)
- squareFree, [2078](#)
- squareFreePart, [2078](#)
- squareFreePolynomial, [2078](#)
- subtractIfCan, [2078](#)
- totalDegree, [2078](#)
- unit?, [2078](#)
- unitCanonical, [2078](#)
- unitNormal, [2078](#)
- univariate, [2078](#)
- variables, [2078](#)
- zero?, [2078](#)
- polygamma
 - DFLOAT, [587](#)
 - EXPR, [724](#)
- polygon
 - SPACE3, [2731](#)
- polygon?
 - SPACE3, [2731](#)
- Polynomial, [2077](#)
- polynomial
 - SMTS, [2441](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- PolynomialIdeals, [2081](#)
- PolynomialRing, [2092](#)
- polyPart
 - FPARFRAC, [1040](#)
- posExpnPart
 - NSDPS, [1703](#)
- position
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - BINFILE, [295](#)
 - BITS, [315](#)
 - CDFVEC, [432](#)
 - DFVEC, [605](#)
 - DLIST, [460](#)
 - FARRAY, [886](#)
 - IARRAY1, [1244](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - ILIST, [1231](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - KERNEL, [1404](#)
 - LIST, [1504](#)
 - MKCHSET, [1570](#)
 - POINT, [2060](#)
 - PRIMARR, [2109](#)
 - STRING, [2607](#)
 - U16VEC, [2924](#)
 - U32VEC, [2931](#)
 - U8VEC, [2917](#)
 - VECTOR, [2942](#)
- positive?
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - DIRPROD, [547](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - EXPEXPAN, [712](#)
 - FLOAT, [908](#)
 - FRAC, [986](#)
 - HDP, [1173](#)
 - HEXADEC, [1143](#)
 - INT, [1362](#)
 - INTRVL, [1384](#)
 - LA, [1520](#)
 - MFLOAT, [1548](#)
 - MINT, [1557](#)
 - ODP, [1816](#)
 - ONECOMP, [1777](#)
 - ORDCOMP, [1810](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - RADIX, [2206](#)
 - RECLOS, [2237](#)
 - ROIRC, [2310](#)
 - ROMAN, [2326](#)
 - SHDP, [2509](#)
 - SINT, [2412](#)
 - SULS, [2457](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
- PositiveInteger, [2100](#)

- positiveRemainder
 - INT, [1362](#)
 - MINT, [1557](#)
 - ROMAN, [2326](#)
 - SINT, [2412](#)
- possiblyInfinite?
 - ALIST, [235](#)
 - DLIST, [460](#)
 - ILIST, [1231](#)
 - LIST, [1504](#)
 - NSDPS, [1703](#)
 - STREAM, [2582](#)
- postfix
 - OUTFORM, [1868](#)
- pow
 - MODMON, [1632](#)
- powerAssociative?
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- powers
 - PRTITION, [1923](#)
- powmod
 - INT, [1362](#)
 - MINT, [1557](#)
 - ROMAN, [2326](#)
 - SINT, [2412](#)
- pquo
 - NSMP, [1714](#)
- PR, [2092](#)
 - ?, [2092](#)
 - ?**?, [2092](#)
 - ?*?, [2092](#)
 - ?+?, [2092](#)
 - ?-?, [2092](#)
 - ?/?, [2092](#)
 - ?=?, [2092](#)
 - ?^?, [2092](#)
 - ?~=?, [2092](#)
 - 0, [2092](#)
 - 1, [2092](#)
 - associates?, [2092](#)
 - binomThmExpt, [2092](#)
 - characteristic, [2092](#)
 - charthRoot, [2092](#)
 - coefficient, [2092](#)
 - coefficients, [2092](#)
 - coerce, [2092](#)
 - content, [2092](#)
 - degree, [2092](#)
 - exquo, [2092](#)
 - fmeceg, [2092](#)
 - ground, [2092](#)
 - ground?, [2092](#)
 - hash, [2092](#)
 - latex, [2092](#)
 - leadingCoefficient, [2092](#)
 - leadingMonomial, [2092](#)
 - map, [2092](#)
 - mapExponents, [2092](#)
 - minimumDegree, [2092](#)
 - monomial, [2092](#)
 - monomial?, [2092](#)
 - numberOfMonomials, [2092](#)
 - one?, [2092](#)
 - pomopo, [2092](#)
 - primitivePart, [2092](#)
 - recip, [2092](#)
 - reductum, [2092](#)
 - retract, [2092](#)
 - retractIfCan, [2092](#)
 - sample, [2092](#)
 - subtractIfCan, [2092](#)
 - unit?, [2092](#)
 - unitCanonical, [2092](#)
 - unitNormal, [2092](#)
 - zero?, [2092](#)
- precision
 - DFLOAT, [587](#)
 - FLOAT, [908](#)
 - MFLOAT, [1548](#)
- predicates
 - PATTERN, [1928](#)
- prefix
 - OUTFORM, [1868](#)
- prefix?
 - ISTRING, [1249](#)
 - STRING, [2607](#)
- prefixRagits
 - RADIX, [2206](#)

- prem
 - NSMP, [1714](#)
- preprocess
 - REGSET, [2286](#)
 - SREGSET, [2535](#)
- presub
 - OUTFORM, [1868](#)
- presuper
 - OUTFORM, [1868](#)
- previousTower
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
- PRIMARR, [2109](#)
 - ?<?, [2109](#)
 - ?<=?, [2109](#)
 - ?>?, [2109](#)
 - ?>=?, [2109](#)
 - ?.?, [2109](#)
 - ?=?, [2109](#)
 - ?~=?, [2109](#)
 - #?, [2109](#)
 - any?, [2109](#)
 - coerce, [2109](#)
 - concat, [2109](#)
 - construct, [2109](#)
 - convert, [2109](#)
 - copy, [2109](#)
 - copyInto, [2109](#)
 - count, [2109](#)
 - delete, [2109](#)
 - elt, [2109](#)
 - empty, [2109](#)
 - empty?, [2109](#)
 - entries, [2109](#)
 - entry?, [2109](#)
 - eq?, [2109](#)
 - eval, [2109](#)
 - every?, [2109](#)
 - fill, [2109](#)
 - find, [2109](#)
 - first, [2109](#)
 - hash, [2109](#)
 - index?, [2109](#)
 - indices, [2109](#)
 - insert, [2109](#)
 - latex, [2109](#)
 - less?, [2109](#)
 - map, [2109](#)
 - max, [2109](#)
 - maxIndex, [2109](#)
 - member?, [2109](#)
 - members, [2109](#)
 - merge, [2109](#)
 - min, [2109](#)
 - minIndex, [2109](#)
 - more?, [2109](#)
 - new, [2109](#)
 - parts, [2109](#)
 - position, [2109](#)
 - qelt, [2109](#)
 - qsetelt, [2109](#)
 - reduce, [2109](#)
 - remove, [2109](#)
 - removeDuplicates, [2109](#)
 - reverse, [2109](#)
 - sample, [2109](#)
 - select, [2109](#)
 - setelt, [2109](#)
 - size?, [2109](#)
 - sort, [2109](#)
 - sorted?, [2109](#)
 - swap, [2109](#)
- prime
 - OUTFORM, [1868](#)
- prime?
 - ALGFF, [28](#)
 - AN, [35](#)
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)

- FFNB, 860
- FFNBP, 872
- FFNBX, 866
- FFP, 851
- FFX, 846
- FLOAT, 908
- FR, 786
- FRAC, 986
- GDMP, 1052
- GSERIES, 1090
- HACKPI, 1978
- HDMP, 1180
- HEXADEC, 1143
- IAN, 1276
- IFF, 1284
- INT, 1362
- IPF, 1303
- MCMPLEX, 1543
- MFLOAT, 1548
- MINT, 1557
- MODFIELD, 1638
- MODMON, 1632
- MPOLY, 1682
- MYEXPR, 1688
- MYUP, 1695
- NSDPS, 1703
- NSMP, 1714
- NSUP, 1729
- ODPOL, 1852
- ODR, 1858
- PACOFF, 2136
- PACRAT, 2146
- PADICRAT, 1886
- PADICRC, 1891
- PF, 2105
- PFR, 1914
- POLY, 2078
- RADFF, 2194
- RADIX, 2206
- RECLOS, 2237
- ROMAN, 2326
- SAE, 2400
- SDPOL, 2386
- SINT, 2412
- SMP, 2423
- SULS, 2457
- SUP, 2467
- SUPEXPR, 2481
- SUPXS, 2487
- ULS, 2796
- ULSCONS, 2804
- UP, 2828
- UPXS, 2834
- UPXSCONS, 2842
- primeFactor
 - FR, 786
- PrimeField, 2104
- primeFrobenius
 - ALGFF, 28
 - COMPLEX, 419
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866
 - FFP, 851
 - FFX, 846
 - IFF, 1284
 - IPF, 1303
 - MCMPLEX, 1543
 - PACOFF, 2136
 - PACRAT, 2146
 - PF, 2105
 - RADFF, 2194
 - SAE, 2400
- primitive?
 - ALGFF, 28
 - COMPLEX, 419
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866
 - FFP, 851
 - FFX, 846
 - IFF, 1284
 - IPF, 1303
 - MCMPLEX, 1543

- PACOFF, [2136](#)
- PF, [2105](#)
- RADFF, [2194](#)
- SAE, [2400](#)
- PrimitiveArray, [2109](#)
- primitiveElement
 - ALGFF, [28](#)
 - COMPLEX, [419](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - MCMPLX, [1543](#)
 - PACOFF, [2136](#)
 - PF, [2105](#)
 - RADFF, [2194](#)
 - SAE, [2400](#)
- primitiveMonomials
 - DMP, [572](#)
 - DSMP, [542](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - POLY, [2078](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - UP, [2828](#)
- primitivePart
 - ALGFF, [28](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
 - POLY, [2078](#)
 - PR, [2092](#)
 - RADFF, [2194](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - SYMPOLY, [2654](#)
 - UP, [2828](#)
 - UPXSSING, [2852](#)
- primPartElseUnitCanonical
 - NSMP, [1714](#)
- principal?
 - FDIV, [813](#)
 - HELLFDIV, [1183](#)
- principalIdeal
 - ALGFF, [28](#)
 - AN, [35](#)
 - BINARY, [292](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - EMR, [702](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)

- FFNBP, [872](#)
- FFNBX, [866](#)
- FFP, [851](#)
- FFX, [846](#)
- FLOAT, [908](#)
- FRAC, [986](#)
- GSERIES, [1090](#)
- HACKPI, [1978](#)
- HEXADEC, [1143](#)
- IAN, [1276](#)
- IFF, [1284](#)
- INT, [1362](#)
- IPADIC, [1294](#)
- IPF, [1303](#)
- LAUPOL, [1422](#)
- MCMPLX, [1543](#)
- MFLOAT, [1548](#)
- MINT, [1557](#)
- MODFIELD, [1638](#)
- MODMON, [1632](#)
- MYEXPR, [1688](#)
- MYUP, [1695](#)
- NSDPS, [1703](#)
- NSUP, [1729](#)
- ODR, [1858](#)
- PACOFF, [2136](#)
- PACRAT, [2146](#)
- PADIC, [1881](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PF, [2105](#)
- PFR, [1914](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- ROMAN, [2326](#)
- SAE, [2400](#)
- SINT, [2412](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- print
 - OUTFORM, [1868](#)
- printCode
 - FC, [931](#)
- printHeader
 - SYMS, [2696](#)
- printInfo
 - NSDPS, [1703](#)
- printStatement
 - FC, [931](#)
- printTypes
 - SYMS, [2696](#)
 - SYMTAB, [2648](#)
- processTemplate
 - FTEM, [967](#)
- prod
 - OUTFORM, [1868](#)
- PRODUCT, [2112](#)
 - , [2113](#)
 - ?<?, [2113](#)
 - ?<=?, [2113](#)
 - ?>?, [2113](#)
 - ?>=?, [2113](#)
 - ?**?, [2113](#)
 - ?*?, [2113](#)
 - ?+?, [2113](#)
 - ?-?, [2113](#)
 - ?/?, [2113](#)
 - ?=?, [2113](#)
 - ?^?, [2113](#)
 - ?~=?, [2113](#)
 - 0, [2113](#)
 - 1, [2113](#)
 - coerce, [2113](#)
 - commutator, [2113](#)
 - conjugate, [2113](#)
 - hash, [2113](#)
 - index, [2113](#)
 - inv, [2113](#)
 - latex, [2113](#)
 - lookup, [2113](#)
 - makeprod, [2113](#)
 - max, [2113](#)
 - min, [2113](#)
 - one?, [2113](#)

- random, [2113](#)
- recip, [2113](#)
- sample, [2113](#)
- selectfirst, [2113](#)
- selectsecond, [2113](#)
- size, [2113](#)
- subtractIfCan, [2113](#)
- sup, [2113](#)
- zero?, [2113](#)
- Product, [2112](#)
- product
 - CARTEN, [356](#)
 - EXPR, [724](#)
 - MYEXPR, [1688](#)
 - XPBWPOLYL, [2987](#)
- ProjectivePlane, [2117](#)
- ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField, [2119](#)
- projectivePoint
 - PROJPL, [2117](#)
 - PROJPLPS, [2119](#)
 - PROJSP, [2122](#)
- ProjectiveSpace, [2121](#)
- PROJPL, [2117](#)
 - ?.?, [2117](#)
 - ?=?, [2117](#)
 - ?~=?, [2117](#)
 - coerce, [2117](#)
 - conjugate, [2117](#)
 - definingField, [2117](#)
 - degree, [2117](#)
 - hash, [2117](#)
 - homogenize, [2117](#)
 - lastNonNul, [2117](#)
 - lastNonNull, [2117](#)
 - latex, [2117](#)
 - list, [2117](#)
 - orbit, [2117](#)
 - pointValue, [2117](#)
 - projectivePoint, [2117](#)
 - rational?, [2117](#)
 - removeConjugate, [2117](#)
 - setelt, [2117](#)
- PROJPLPS, [2119](#)
 - ?.?, [2119](#)
 - ?=?, [2119](#)
- ?~=?, [2119](#)
- coerce, [2119](#)
- conjugate, [2119](#)
- definingField, [2119](#)
- degree, [2119](#)
- hash, [2119](#)
- homogenize, [2119](#)
- lastNonNul, [2119](#)
- lastNonNull, [2119](#)
- latex, [2119](#)
- list, [2119](#)
- orbit, [2119](#)
- pointValue, [2119](#)
- projectivePoint, [2119](#)
- rational?, [2119](#)
- removeConjugate, [2119](#)
- setelt, [2119](#)
- PROJSP, [2121](#)
 - ?.?, [2122](#)
 - ?=?, [2122](#)
 - ?~=?, [2122](#)
 - coerce, [2122](#)
 - conjugate, [2122](#)
 - definingField, [2122](#)
 - degree, [2122](#)
 - hash, [2122](#)
 - homogenize, [2122](#)
 - lastNonNul, [2122](#)
 - lastNonNull, [2122](#)
 - latex, [2122](#)
 - list, [2122](#)
 - orbit, [2122](#)
 - pointValue, [2122](#)
 - projectivePoint, [2122](#)
 - rational?, [2122](#)
 - removeConjugate, [2122](#)
 - setelt, [2122](#)
- prologue
 - FORMULA, [2346](#)
 - TEX, [2677](#)
- properties
 - BOP, [274](#)
- property
 - BOP, [274](#)
- PRTITION, [1923](#)
 - ?<?, [1923](#)

- ?<=?, 1923
- ?>?, 1923
- ?>=?, 1923
- ?*?, 1923
- ?+?, 1923
- ?=?, 1923
- ?~=?, 1923
- 0, 1923
- coerce, 1923
- conjugate, 1923
- convert, 1923
- hash, 1923
- latex, 1923
- max, 1923
- min, 1923
- partition, 1923
- pdct, 1923
- powers, 1923
- sample, 1923
- subtractIfCan, 1923
- zero?, 1923
- PseudoAlgebraicClosureOfAlgExtOfRationalNumber, 2126
- PseudoAlgebraicClosureOfFiniteField, 2135
- PseudoAlgebraicClosureOfRationalNumber, 2146
- pseudoDivide
 - MODMON, 1632
 - MYUP, 1695
 - NSMP, 1714
 - NSUP, 1729
 - SUP, 2467
 - SUEXPR, 2481
 - UP, 2828
- pseudoQuotient
 - MODMON, 1632
 - MYUP, 1695
 - NSUP, 1729
 - SUP, 2467
 - SUEXPR, 2481
 - UP, 2828
- pseudoRemainder
 - MODMON, 1632
 - MYUP, 1695
 - NSUP, 1729
 - SUP, 2467
 - SUEXPR, 2481
- UP, 2828
- ptree
 - PENDTREE, 1945
- puiseux
 - SUPXS, 2487
 - UPXS, 2834
 - UPXSCONS, 2842
- purelyAlgebraic?
 - REGSET, 2286
 - RGCHAIN, 2255
 - SREGSET, 2535
- purelyAlgebraicLeadingMonomial?
 - REGSET, 2286
 - RGCHAIN, 2255
 - SREGSET, 2535
- purelyTranscendental?
 - REGSET, 2286
 - RGCHAIN, 2255
 - SREGSET, 2535
- putColorInfo
 - GRIMAGE, 1094
- putGraph
 - VIEW2d, 2770
- QALGSET, 2157
 - ?=?, 2157
 - ?~=?, 2157
 - coerce, 2157
 - definingEquations, 2157
 - definingInequation, 2157
 - empty, 2157
 - empty?, 2157
 - hash, 2157
 - idealSimplify, 2157
 - latex, 2157
 - quasiAlgebraicSet, 2157
 - setStatus, 2157
 - simplify, 2157
 - status, 2157
- qelt
 - ALIST, 235
 - ARRAY1, 1774
 - ARRAY2, 2763
 - BITS, 315
 - CDFMAT, 426
 - CDFVEC, 432

- DFMAT, [599](#)
- DFVEC, [605](#)
- DHMATRIX, [491](#)
- DIRPROD, [547](#)
- DLIST, [460](#)
- DPMM, [553](#)
- DPMO, [557](#)
- EQTBL, [699](#)
- FARRAY, [886](#)
- GSTBL, [1079](#)
- HASHTBL, [1120](#)
- HDP, [1173](#)
- IARRAY1, [1244](#)
- IARRAY2, [1256](#)
- IBITS, [1199](#)
- IFARRAY, [1222](#)
- IIARRAY2, [1290](#)
- ILIST, [1231](#)
- IMATRIX, [1239](#)
- INTABL, [1336](#)
- ISTRING, [1249](#)
- IVECTOR, [1260](#)
- KAFILE, [1414](#)
- LIB, [1429](#)
- LIST, [1504](#)
- LSQM, [1456](#)
- MATRIX, [1623](#)
- NSDPS, [1703](#)
- ODP, [1816](#)
- POINT, [2060](#)
- PRIMARR, [2109](#)
- RESULT, [2301](#)
- RMATRIX, [2246](#)
- ROUTINE, [2331](#)
- SHDP, [2509](#)
- SQMATRIX, [2548](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- QEQUAT, [2169](#)
 - coerce, [2169](#)
 - equation, [2169](#)
 - value, [2169](#)
 - variable, [2169](#)
- QFORM, [2154](#)
 - ?, [2154](#)
 - ?*?, [2154](#)
 - ?+?, [2154](#)
 - ?-?, [2154](#)
 - ?., [2154](#)
 - ?=?, [2154](#)
 - ?~=?, [2154](#)
 - 0, [2154](#)
 - coerce, [2154](#)
 - hash, [2154](#)
 - latex, [2154](#)
 - matrix, [2154](#)
 - quadraticForm, [2154](#)
 - sample, [2154](#)
 - subtractIfCan, [2154](#)
 - zero?, [2154](#)
- qinterval
 - INTRVL, [1384](#)
- qnew
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)
 - DFMAT, [599](#)
 - DFVEC, [605](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- QuadraticForm, [2154](#)
- quadraticForm
 - QFORM, [2154](#)
- QuasiAlgebraicSet, [2157](#)
- quasiAlgebraicSet
 - QALGSET, [2157](#)
- quasiComponent
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)

- quasiMonic?
 - NSMP, [1714](#)
- quasiRegular
 - XDPLY, [2967](#)
 - XPBWPLY, [2987](#)
 - XPOLY, [2998](#)
 - XPR, [3007](#)
 - XPOLY, [3013](#)
- quasiRegular?
 - XDPLY, [2967](#)
 - XPBWPLY, [2987](#)
 - XPOLY, [2998](#)
 - XPR, [3007](#)
 - XPOLY, [3013](#)
- QUAT, [2167](#)
 - ?, [2167](#)
 - ?<?, [2167](#)
 - ?<=?, [2167](#)
 - ?>?, [2167](#)
 - ?>=?, [2167](#)
 - ?**?, [2167](#)
 - ?*?, [2167](#)
 - ?+?, [2167](#)
 - ?-?, [2167](#)
 - ?., [2167](#)
 - ?=, [2167](#)
 - ?^?, [2167](#)
 - ?~=?, [2167](#)
 - 0, [2167](#)
 - 1, [2167](#)
 - abs, [2167](#)
 - characteristic, [2167](#)
 - charthRoot, [2167](#)
 - coerce, [2167](#)
 - conjugate, [2167](#)
 - convert, [2167](#)
 - D, [2167](#)
 - differentiate, [2167](#)
 - eval, [2167](#)
 - hash, [2167](#)
 - imagI, [2167](#)
 - imagJ, [2167](#)
 - imagK, [2167](#)
 - inv, [2167](#)
 - latex, [2167](#)
 - map, [2167](#)
 - max, [2167](#)
 - min, [2167](#)
 - norm, [2167](#)
 - one?, [2167](#)
 - quatern, [2167](#)
 - rational, [2167](#)
 - rational?, [2167](#)
 - rationalIfCan, [2167](#)
 - real, [2167](#)
 - recip, [2167](#)
 - reducedSystem, [2167](#)
 - retract, [2167](#)
 - retractIfCan, [2167](#)
 - sample, [2167](#)
 - subtractIfCan, [2167](#)
 - zero?, [2167](#)
- quatern
 - QUAT, [2167](#)
- Quaternion, [2167](#)
- QueryEquation, [2169](#)
- QUEUE, [2184](#)
 - ?=, [2184](#)
 - ?~=?, [2184](#)
 - #?, [2184](#)
 - any?, [2184](#)
 - back, [2184](#)
 - bag, [2184](#)
 - coerce, [2184](#)
 - copy, [2184](#)
 - count, [2184](#)
 - dequeue, [2184](#)
 - empty, [2184](#)
 - empty?, [2184](#)
 - enqueue, [2184](#)
 - eq?, [2184](#)
 - eval, [2184](#)
 - every?, [2184](#)
 - extract, [2184](#)
 - front, [2184](#)
 - hash, [2184](#)
 - insert, [2184](#)
 - inspect, [2184](#)
 - latex, [2184](#)
 - length, [2184](#)
 - less?, [2184](#)
 - map, [2184](#)

- member?, 2184
- members, 2184
- more?, 2184
- parts, 2184
- queue, 2184
- rotate, 2184
- sample, 2184
- size?, 2184
- Queue, 2184
- queue
 - QUEUE, 2184
- quoByVar
 - SUTS, 2497
 - UFPS, 2789
 - UTS, 2877
 - UTSZ, 2887
- quote
 - CHAR, 373
 - OUTFORM, 1868
- quoted?
 - PATTERN, 1928
- quotedOperators
 - RULE, 2305
- quotient
 - IDEAL, 2081
- quotientByP
 - BPADIC, 258
 - IPADIC, 1294
 - PADIC, 1881
- quotValuation
 - BLHN, 317
 - BLQT, 319
- RADFF, 2193
 - , 2194
 - **, 2194
 - *, 2194
 - +, 2194
 - , 2194
 - /, 2194
 - =, 2194
 - ^, 2194
 - ~=, 2194
 - ?quo?, 2194
 - ?rem?, 2194
 - 0, 2194
 - 1, 2194
 - absolutelyIrreducible?, 2194
 - algSplitSimple, 2194
 - associates?, 2194
 - basis, 2194
 - branchPoint?, 2194
 - branchPointAtInfinity?, 2194
 - characteristic, 2194
 - characteristicPolynomial, 2194
 - charthRoot, 2194
 - coerce, 2194
 - complementaryBasis, 2194
 - conditionP, 2194
 - convert, 2194
 - coordinates, 2194
 - createPrimitiveElement, 2194
 - D, 2194
 - definingPolynomial, 2194
 - derivationCoordinates, 2194
 - differentiate, 2194
 - discreteLog, 2194
 - discriminant, 2194
 - divide, 2194
 - elliptic, 2194
 - elt, 2194
 - euclideanSize, 2194
 - expressIdealMember, 2194
 - exquo, 2194
 - extendedEuclidean, 2194
 - factor, 2194
 - factorsOfCyclicGroupSize, 2194
 - gcd, 2194
 - gcdPolynomial, 2194
 - generator, 2194
 - genus, 2194
 - hash, 2194
 - hyperelliptic, 2194
 - index, 2194
 - init, 2194
 - integral?, 2194
 - integralAtInfinity?, 2194
 - integralBasis, 2194
 - integralBasisAtInfinity, 2194
 - integralCoordinates, 2194
 - integralDerivationMatrix, 2194
 - integralMatrix, 2194

- integralMatrixAtInfinity, 2194
- integralRepresents, 2194
- inv, 2194
- inverseIntegralMatrix, 2194
- inverseIntegralMatrixAtInfinity, 2194
- latex, 2194
- lcm, 2194
- lift, 2194
- lookup, 2194
- minimalPolynomial, 2194
- multiEuclidean, 2194
- nextItem, 2194
- nonSingularModel, 2194
- norm, 2194
- normalizeAtInfinity, 2194
- numberOfComponents, 2194
- one?, 2194
- order, 2194
- prime?, 2194
- primeFrobenius, 2194
- primitive?, 2194
- primitiveElement, 2194
- primitivePart, 2194
- principalIdeal, 2194
- ramified?, 2194
- ramifiedAtInfinity?, 2194
- random, 2194
- rank, 2194
- rationalPoint?, 2194
- rationalPoints, 2194
- recip, 2194
- reduce, 2194
- reduceBasisAtInfinity, 2194
- reducedSystem, 2194
- regularRepresentation, 2194
- representationType, 2194
- represents, 2194
- retract, 2194
- retractIfCan, 2194
- sample, 2194
- singular?, 2194
- singularAtInfinity?, 2194
- size, 2194
- sizeLess?, 2194
- squareFree, 2194
- squareFreePart, 2194
- subtractIfCan, 2194
- tableForDiscreteLogarithm, 2194
- trace, 2194
- traceMatrix, 2194
- unit?, 2194
- unitCanonical, 2194
- unitNormal, 2194
- yCoordinates, 2194
- zero?, 2194
- RadicalFunctionField, 2193
- RADIX, 2205
 - ?, 2206
 - ?<?, 2206
 - ?<=?, 2206
 - ?>?, 2206
 - ?>=?, 2206
 - ?**?, 2206
 - ?*?, 2206
 - ?+?, 2206
 - ?-?, 2206
 - ?., 2206
 - ?/?, 2206
 - ?=?, 2206
 - ?^?, 2206
 - ?~=?, 2206
 - ?quo?, 2206
 - ?rem?, 2206
 - 0, 2206
 - 1, 2206
 - abs, 2206
 - associates?, 2206
 - ceiling, 2206
 - characteristic, 2206
 - charthRoot, 2206
 - coerce, 2206
 - conditionP, 2206
 - convert, 2206
 - cycleRagits, 2206
 - D, 2206
 - denom, 2206
 - denominator, 2206
 - differentiate, 2206
 - divide, 2206
 - euclideanSize, 2206
 - eval, 2206
 - expressIdealMember, 2206

- exquo, [2206](#)
- extendedEuclidean, [2206](#)
- factor, [2206](#)
- factorPolynomial, [2206](#)
- factorSquareFreePolynomial, [2206](#)
- floor, [2206](#)
- fractionPart, [2206](#)
- fractRadix, [2206](#)
- fractRagits, [2206](#)
- gcd, [2206](#)
- gcdPolynomial, [2206](#)
- hash, [2206](#)
- init, [2206](#)
- inv, [2206](#)
- latex, [2206](#)
- lcm, [2206](#)
- map, [2206](#)
- max, [2206](#)
- min, [2206](#)
- multiEuclidean, [2206](#)
- negative?, [2206](#)
- nextItem, [2206](#)
- numer, [2206](#)
- numerator, [2206](#)
- one?, [2206](#)
- patternMatch, [2206](#)
- positive?, [2206](#)
- prefixRagits, [2206](#)
- prime?, [2206](#)
- principalIdeal, [2206](#)
- random, [2206](#)
- recip, [2206](#)
- reducedSystem, [2206](#)
- retract, [2206](#)
- retractIfCan, [2206](#)
- sample, [2206](#)
- sign, [2206](#)
- sizeLess?, [2206](#)
- solveLinearPolynomialEquation, [2206](#)
- squareFree, [2206](#)
- squareFreePart, [2206](#)
- squareFreePolynomial, [2206](#)
- subtractIfCan, [2206](#)
- unit?, [2206](#)
- unitCanonical, [2206](#)
- unitNormal, [2206](#)
- wholePart, [2206](#)
- wholeRadix, [2206](#)
- wholeRagits, [2206](#)
- zero?, [2206](#)
- RadixExpansion, [2205](#)
- ramified?
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- ramifiedAtInfinity?
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- ramifMult
 - BLHN, [317](#)
 - BLQT, [319](#)
- random
 - ALGFF, [28](#)
 - BINARY, [292](#)
 - BOOLEAN, [322](#)
 - BPADICRT, [263](#)
 - CCLASS, [382](#)
 - CHAR, [373](#)
 - COMPLEX, [419](#)
 - DECIMAL, [465](#)
 - DIRPROD, [547](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - EXPEXPAN, [712](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FRAC, [986](#)
 - HDP, [1173](#)
 - HEXADEC, [1143](#)
 - IFF, [1284](#)
 - INT, [1362](#)
 - IPF, [1303](#)
 - MCMPLEX, [1543](#)
 - MINT, [1557](#)
 - MODMON, [1632](#)
 - MRING, [1659](#)

- NNI, [1739](#)
- OCT, [1765](#)
- ODP, [1816](#)
- OVAR, [1836](#)
- PACOFF, [2136](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PERMGRP, [1960](#)
- PF, [2105](#)
- PRODUCT, [2113](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- ROMAN, [2326](#)
- SAE, [2400](#)
- SET, [2372](#)
- SETMN, [2377](#)
- SHDP, [2509](#)
- SINT, [2412](#)
- SULS, [2457](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- ZMOD, [1368](#)
- randomLC
 - FRIDEAL, [995](#)
- range
 - DROPT, [608](#)
- ranges
 - DROPT, [608](#)
 - GRIMAGE, [1094](#)
- rank
 - ALGFF, [28](#)
 - ALGSC, [15](#)
 - CARTEN, [356](#)
 - CDFMAT, [426](#)
 - COMPLEX, [419](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - GCNAALG, [1065](#)
 - IMATRIX, [1239](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - MCMPLEX, [1543](#)
 - RADFF, [2194](#)
 - RMATRIX, [2246](#)
 - SAE, [2400](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- rarrow
 - OUTFORM, [1868](#)
- rational
 - COMPLEX, [419](#)
 - FR, [786](#)
 - INT, [1362](#)
 - MCMPLEX, [1543](#)
 - MINT, [1557](#)
 - OCT, [1765](#)
 - ONECOMP, [1777](#)
 - ORDCOMP, [1810](#)
 - QUAT, [2167](#)
 - ROMAN, [2326](#)
 - SINT, [2412](#)
- rational?
 - AFFPLPS, [7](#)
 - AFFSP, [9](#)
 - COMPLEX, [419](#)
 - FR, [786](#)
 - INT, [1362](#)
 - MCMPLEX, [1543](#)
 - MINT, [1557](#)
 - OCT, [1765](#)
 - ONECOMP, [1777](#)
 - ORDCOMP, [1810](#)
 - PROJPL, [2117](#)
 - PROJPLPS, [2119](#)
 - PROJSP, [2122](#)
 - QUAT, [2167](#)
 - ROMAN, [2326](#)
 - SINT, [2412](#)
- rationalApproximation
 - DFLOAT, [587](#)
 - FLOAT, [908](#)
- rationalFunction
 - SULS, [2457](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
- rationalIfCan
 - COMPLEX, [419](#)
 - FR, [786](#)

- INT, [1362](#)
- MCMPLX, [1543](#)
- MINT, [1557](#)
- OCT, [1765](#)
- ONECOMP, [1777](#)
- ORDCOMP, [1810](#)
- QUAT, [2167](#)
- ROMAN, [2326](#)
- SINT, [2412](#)
- rationalPoint?
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- rationalPoints
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- rationalPower
 - SUPXS, [2487](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
- ratpart
 - IR, [1375](#)
- ravel
 - CARTEN, [356](#)
- readable?
 - FNAME, [810](#)
- real
 - COMPLEX, [419](#)
 - MCMPLX, [1543](#)
 - OCT, [1765](#)
 - QUAT, [2167](#)
- real?
 - FST, [962](#)
- RealClosure, [2236](#)
- recip
 - ALGFF, [28](#)
 - ALGSC, [15](#)
 - AN, [35](#)
 - ANTISYM, [40](#)
 - AUTOMOR, [245](#)
 - BINARY, [292](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - CARD, [332](#)
 - CLIF, [402](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DERHAM, [530](#)
 - DFLOAT, [587](#)
 - DIRPROD, [547](#)
 - DIRRING, [564](#)
 - DMP, [572](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DSMP, [542](#)
 - EMR, [702](#)
 - EQ, [691](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FEXPR, [947](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FGROUP, [1010](#)
 - FLOAT, [908](#)
 - FMONOID, [1021](#)
 - FR, [786](#)
 - FRAC, [986](#)
 - FRIDEAL, [995](#)
 - FRMOD, [1000](#)
 - FSERIES, [978](#)
 - GCNAALG, [1065](#)
 - GDMP, [1052](#)
 - GSERIES, [1090](#)
 - HACKPI, [1978](#)
 - HDMP, [1180](#)
 - HDP, [1173](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - IFF, [1284](#)
 - INT, [1362](#)
 - INTRVL, [1384](#)
 - IPADIC, [1294](#)
 - IPF, [1303](#)
 - ISUPS, [1311](#)

- ITAYLOR, 1338
 JORDAN, 223
 LA, 1520
 LAUPOL, 1422
 LEXP, 1435
 LIE, 229
 LODO, 1469
 LODO1, 1479
 LODO2, 1491
 LSQM, 1456
 MCMPLEX, 1543
 MFLOAT, 1548
 MINT, 1557
 MODFIELD, 1638
 MODMON, 1632
 MODOP, 1647, 1804
 MODRING, 1641
 MOEBIUS, 1654
 MPOLY, 1682
 MRING, 1659
 MYEXPR, 1688
 MYUP, 1695
 NNI, 1739
 NOTTING, 1744
 NSDPS, 1703
 NSMP, 1714
 NSUP, 1729
 OCT, 1765
 ODP, 1816
 ODPOL, 1852
 ODR, 1858
 OFMONOID, 1829
 OMLO, 1806
 ONECOMP, 1777
 ORDCOMP, 1810
 ORESUP, 2492
 OREUP, 2872
 OWP, 1861
 PACOFF, 2136
 PACRAT, 2146
 PADIC, 1881
 PADICRAT, 1886
 PADICRC, 1891
 PERM, 1949
 PF, 2105
 PFR, 1914
 PI, 2100
 POLY, 2078
 PR, 2092
 PRODUCT, 2113
 QUAT, 2167
 RADFF, 2194
 RADIX, 2206
 RECLOS, 2237
 RESRING, 2296
 ROIRC, 2310
 ROMAN, 2326
 SAE, 2400
 SDPOL, 2386
 SHDP, 2509
 SINT, 2412
 SMP, 2423
 SMTS, 2441
 SQMATRIX, 2548
 SULS, 2457
 SUP, 2467
 SUPEXPR, 2481
 SUPXS, 2487
 SUTS, 2497
 SYMPOLY, 2654
 TS, 2671
 UFPS, 2789
 ULS, 2796
 ULSCONS, 2804
 UP, 2828
 UPXS, 2834
 UPXSCONS, 2842
 UPXSING, 2852
 UTS, 2877
 UTSZ, 2887
 WP, 2949
 XDPOLY, 2967
 XPBWPOLYL, 2987
 XPOLY, 2998
 XPR, 3007
 XRPOLY, 3013
 ZMOD, 1368
 RECLOS, 2236
 -?, 2237
 ?<?, 2237
 ?<=?, 2237
 ?>?, 2237

- [?>=?](#), 2237
- [?***?](#), 2237
- [?*?](#), 2237
- [?+?](#), 2237
- [?-?](#), 2237
- [?/?](#), 2237
- [?=?](#), 2237
- [?^?](#), 2237
- [?~=?](#), 2237
- [?quo?](#), 2237
- [?rem?](#), 2237
- [0](#), 2237
- [1](#), 2237
- [abs](#), 2237
- [algebraicOf](#), 2237
- [allRootsOf](#), 2237
- [approximate](#), 2237
- [associates?](#), 2237
- [characteristic](#), 2237
- [coerce](#), 2237
- [divide](#), 2237
- [euclideanSize](#), 2237
- [expressIdealMember](#), 2237
- [exquo](#), 2237
- [extendedEuclidean](#), 2237
- [factor](#), 2237
- [gcd](#), 2237
- [gcdPolynomial](#), 2237
- [hash](#), 2237
- [inv](#), 2237
- [latex](#), 2237
- [lcm](#), 2237
- [mainCharacterization](#), 2237
- [mainDefiningPolynomial](#), 2237
- [mainForm](#), 2237
- [mainValue](#), 2237
- [max](#), 2237
- [min](#), 2237
- [multiEuclidean](#), 2237
- [negative?](#), 2237
- [nthRoot](#), 2237
- [one?](#), 2237
- [positive?](#), 2237
- [prime?](#), 2237
- [principalIdeal](#), 2237
- [recip](#), 2237
- [relativeApprox](#), 2237
- [rename](#), 2237
- [retract](#), 2237
- [retractIfCan](#), 2237
- [rootOf](#), 2237
- [sample](#), 2237
- [sign](#), 2237
- [sizeLess?](#), 2237
- [sqrt](#), 2237
- [squareFree](#), 2237
- [squareFreePart](#), 2237
- [subtractIfCan](#), 2237
- [unit?](#), 2237
- [unitCanonical](#), 2237
- [unitNormal](#), 2237
- [zero?](#), 2237
- [recoverAfterFail](#)
 - [ROUTINE](#), 2331
- [RectangularMatrix](#), 2245
- [rectangularMatrix](#)
 - [RMATRIX](#), 2246
- [red](#)
 - [COLOR](#), 407
- [reduce](#)
 - [ALGFF](#), 28
 - [ALIST](#), 235
 - [AN](#), 35
 - [ARRAY1](#), 1774
 - [BITS](#), 315
 - [CCLASS](#), 382
 - [CDFVEC](#), 432
 - [COMPLEX](#), 419
 - [DFVEC](#), 605
 - [DLIST](#), 460
 - [EMR](#), 702
 - [EQTBL](#), 699
 - [EXPR](#), 724
 - [FARRAY](#), 886
 - [FDIV](#), 813
 - [GPOLSET](#), 1074
 - [GSTBL](#), 1079
 - [GTSET](#), 1083
 - [HASHTBL](#), 1120
 - [HELLFDIV](#), 1183
 - [IAN](#), 1276
 - [IARRAY1](#), 1244

- IBITS, [1199](#)
- IFARRAY, [1222](#)
- ILIST, [1231](#)
- INTABL, [1336](#)
- ISTRING, [1249](#)
- IVECTOR, [1260](#)
- KAFILE, [1414](#)
- LIB, [1429](#)
- LIST, [1504](#)
- LMDICT, [1515](#)
- MCMPLEX, [1543](#)
- MODFIELD, [1638](#)
- MODMON, [1632](#)
- MODRING, [1641](#)
- MSET, [1670](#)
- NSDPS, [1703](#)
- PACOFF, [2136](#)
- PACRAT, [2146](#)
- PLACES, [2019](#)
- PLACESPS, [2021](#)
- POINT, [2060](#)
- PRIMARR, [2109](#)
- RADFF, [2194](#)
- REGSET, [2286](#)
- RESRING, [2296](#)
- RESULT, [2301](#)
- RGCHAIN, [2255](#)
- ROUTINE, [2331](#)
- SAE, [2400](#)
- SET, [2372](#)
- SREGSET, [2535](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- U16VEC, [2924](#)
- U32VEC, [2931](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- WUTSET, [2959](#)
- reduceBasisAtInfinity
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- reduceByQuasiMonic
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- reduced?
 - GTSET, [1083](#)
 - NSMP, [1714](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- reducedContinuedFraction
 - CONTFRAC, [445](#)
- reducedForm
 - CONTFRAC, [445](#)
- reducedSystem
 - ALGFF, [28](#)
 - AN, [35](#)
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - DECIMAL, [465](#)
 - DIRPROD, [547](#)
 - DMP, [572](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DSMP, [542](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - FRAC, [986](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - HDP, [1173](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - INT, [1362](#)
 - LSQM, [1456](#)
 - MCMPLEX, [1543](#)
 - MINT, [1557](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYEXPR, [1688](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODP, [1816](#)

- ODPOL, [1852](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- POLY, [2078](#)
- QUAT, [2167](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- ROMAN, [2326](#)
- SAE, [2400](#)
- SDPOL, [2386](#)
- SHDP, [2509](#)
- SINT, [2412](#)
- SMP, [2423](#)
- SQMATRIX, [2548](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- reductum
 - ANTISYM, [40](#)
 - DERHAM, [530](#)
 - DIV, [575](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPUPXS, [740](#)
 - FM, [1013](#)
 - FM1, [1017](#)
 - GDMP, [1052](#)
 - GMODPOL, [1059](#)
 - GSERIES, [1090](#)
 - HDMP, [1180](#)
 - IDPAG, [1202](#)
 - IDPAM, [1206](#)
 - IDPO, [1209](#)
 - IDPOAM, [1212](#)
 - IDPOAMS, [1215](#)
 - INDE, [1218](#)
 - ISUPS, [1311](#)
 - LAUPOL, [1422](#)
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - LPOLY, [1447](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MRING, [1659](#)
 - MYUP, [1695](#)
 - NSDPS, [1703](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - OMLO, [1806](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
 - POLY, [2078](#)
 - PR, [2092](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - SYMPOLY, [2654](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UP, [2828](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UPXSING, [2852](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
 - XDPOLY, [2967](#)
 - XPBWPOLYL, [2987](#)
 - XPR, [3007](#)
- REF, [2249](#)
 - ?=?, [2249](#)
 - ?~=?, [2249](#)
 - coerce, [2249](#)
 - deref, [2249](#)
 - elt, [2249](#)
 - hash, [2249](#)
 - latex, [2249](#)
 - ref, [2249](#)
 - setelt, [2249](#)
 - setref, [2249](#)
- ref

- REF, 2249
- Reference, 2249
- refine
 - ACPLOT, 1992
 - PLOT, 2029
 - PLOT3D, 2042
 - ROIRC, 2310
- region
 - VIEW2d, 2770
- REGSET, 2285
 - ?=?, 2286
 - ?~=?, 2286
 - #?, 2286
 - algebraic?, 2286
 - algebraicCoefficients?, 2286
 - algebraicVariables, 2286
 - any?, 2286
 - augment, 2286
 - autoReduced?, 2286
 - basicSet, 2286
 - coerce, 2286
 - coHeight, 2286
 - collect, 2286
 - collectQuasiMonic, 2286
 - collectUnder, 2286
 - collectUpper, 2286
 - construct, 2286
 - convert, 2286
 - copy, 2286
 - count, 2286
 - degree, 2286
 - empty, 2286
 - empty?, 2286
 - eq?, 2286
 - eval, 2286
 - every?, 2286
 - extend, 2286
 - extendIfCan, 2286
 - find, 2286
 - first, 2286
 - hash, 2286
 - headReduce, 2286
 - headReduced?, 2286
 - headRemainder, 2286
 - infRittWu?, 2286
 - initiallyReduce, 2286
 - initiallyReduced?, 2286
 - initials, 2286
 - internalAugment, 2286
 - internalZeroSetSplit, 2286
 - intersect, 2286
 - invertible?, 2286
 - invertibleElseSplit?, 2286
 - invertibleSet, 2286
 - last, 2286
 - lastSubResultant, 2286
 - lastSubResultantElseSplit, 2286
 - latex, 2286
 - less?, 2286
 - mainVariable?, 2286
 - mainVariables, 2286
 - map, 2286
 - member?, 2286
 - members, 2286
 - more?, 2286
 - mvar, 2286
 - normalized?, 2286
 - parts, 2286
 - preprocess, 2286
 - purelyAlgebraic?, 2286
 - purelyAlgebraicLeadingMonomial?, 2286
 - purelyTranscendental?, 2286
 - quasiComponent, 2286
 - reduce, 2286
 - reduceByQuasiMonic, 2286
 - reduced?, 2286
 - remainder, 2286
 - remove, 2286
 - removeDuplicates, 2286
 - removeZero, 2286
 - rest, 2286
 - retract, 2286
 - retractIfCan, 2286
 - rewriteIdealWithHeadRemainder, 2286
 - rewriteIdealWithRemainder, 2286
 - rewriteSetWithReduction, 2286
 - roughBase?, 2286
 - roughEqualIdeals?, 2286
 - roughSubIdeal?, 2286
 - roughUnitIdeal?, 2286
 - sample, 2286
 - select, 2286

- size?, 2286
- sort, 2286
- squareFreePart, 2286
- stronglyReduce, 2286
- stronglyReduced?, 2286
- triangular?, 2286
- trivialIdeal?, 2286
- variables, 2286
- zeroSetSplit, 2286
- zeroSetSplitIntoTriangularSystems, 2286
- RegularChain, 2254
- regularRepresentation
 - ALGFF, 28
 - COMPLEX, 419
 - MCMPLX, 1543
 - RADFF, 2194
 - SAE, 2400
- RegularTriangularSet, 2285
- reindex
 - CARTEN, 356
- relationsIdeal
 - IDEAL, 2081
- relativeApprox
 - RECLOS, 2237
 - ROIRC, 2310
- releror
 - FLOAT, 908
- remainder
 - GPOLSET, 1074
 - GTSET, 1083
 - REGSET, 2286
 - RGCHAIN, 2255
 - SREGSET, 2535
 - WUTSET, 2959
- RemainderList
 - XPOLY, 2998
 - XRPOLY, 3013
- remove
 - ALIST, 235
 - ARRAY1, 1774
 - BITS, 315
 - CCLASS, 382
 - CDFVEC, 432
 - DFVEC, 605
 - DLIST, 460
 - EQTBL, 699
 - FARRAY, 886
 - GPOLSET, 1074
 - GSTBL, 1079
 - GTSET, 1083
 - HASHTBL, 1120
 - IARRAY1, 1244
 - IFARRAY, 1222
 - ILIST, 1231
 - INTABL, 1336
 - ISTRING, 1249
 - IVECTOR, 1260
 - KAFILE, 1414
 - LIB, 1429
 - LIST, 1504
 - LMDICT, 1515
 - MSET, 1670
 - NSDPS, 1703
 - POINT, 2060
 - PRIMARR, 2109
 - REGSET, 2286
 - RESULT, 2301
 - RGCHAIN, 2255
 - ROUTINE, 2331
 - SET, 2372
 - SPLTREE, 2518
 - SREGSET, 2535
 - STBL, 2450
 - STREAM, 2582
 - STRTBL, 2610
 - TABLE, 2664
 - U16VEC, 2924
 - U32VEC, 2931
 - U8VEC, 2917
 - VECTOR, 2942
 - WUTSET, 2959
- removeConjugate
 - AFFPLPS, 7
 - AFFSP, 9
 - PROJPL, 2117
 - PROJPLPS, 2119
 - PROJSP, 2122
- removeDuplicates
 - ALIST, 235
 - ARRAY1, 1774
 - BITS, 315
 - CCLASS, 382

- CDFVEC, [432](#)
- DFVEC, [605](#)
- DLIST, [460](#)
- EQTBL, [699](#)
- FARRAY, [886](#)
- GPOLSET, [1074](#)
- GSTBL, [1079](#)
- GTSET, [1083](#)
- HASHTBL, [1120](#)
- IARRAY1, [1244](#)
- IBITS, [1199](#)
- IFARRAY, [1222](#)
- ILIST, [1231](#)
- INTABL, [1336](#)
- ISTRING, [1249](#)
- IVECTOR, [1260](#)
- KAFILE, [1414](#)
- LIB, [1429](#)
- LIST, [1504](#)
- LM_DICT, [1515](#)
- MSET, [1670](#)
- NSDPS, [1703](#)
- POINT, [2060](#)
- PRIMARR, [2109](#)
- REGSET, [2286](#)
- RESULT, [2301](#)
- RGCHAIN, [2255](#)
- ROUTINE, [2331](#)
- SET, [2372](#)
- SREGSET, [2535](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- U16VEC, [2924](#)
- U32VEC, [2931](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- WUTSET, [2959](#)
- removeFirstZeroes
 - NSDPS, [1703](#)
- removeZero
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- removeZeroes
 - BPADICRT, [263](#)
 - NSDPS, [1703](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - SULS, [2457](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
- rename
 - RECLOS, [2237](#)
- reorder
 - DMP, [572](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
- repeating
 - STREAM, [2582](#)
- repeating?
 - STREAM, [2582](#)
- repeatUntilLoop
 - FC, [931](#)
- replace
 - ISTRING, [1249](#)
 - STRING, [2607](#)
- replaceKthElement
 - SETMN, [2377](#)
- representationType
 - ALGFF, [28](#)
 - COMPLEX, [419](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - MCMPLEX, [1543](#)
 - PACOFF, [2136](#)
 - PF, [2105](#)
 - RADFF, [2194](#)
 - SAE, [2400](#)

represents

ALGFF, 28
 ALGSC, 15
 COMPLEX, 419
 FF, 820
 FFCG, 825
 FFCGP, 835
 FFCGX, 830
 FFNB, 860
 FFNB, 872
 FFNBX, 866
 FFP, 851
 FFX, 846
 GCNAALG, 1065
 IFF, 1284
 IPF, 1303
 JORDAN, 223
 LIE, 229
 LSQM, 1456
 MCMPLEX, 1543
 PF, 2105
 RADFF, 2194
 SAE, 2400

reset

VIEW2d, 2770
 VIEW3D, 2710

resetAttributeButtons

ATTRIBUT, 239

resetBadValues

PATTERN, 1928

resetNew

SYMBOL, 2640

ResidueRing, 2296

resize

VIEW2d, 2770
 VIEW3D, 2710

RESRING, 2296

-, 2296
 ?**?, 2296
 ?*?, 2296
 ?+?, 2296
 ?-?, 2296
 ?=?, 2296
 ?^?, 2296
 ?~=?, 2296
 0, 2296

1, 2296

characteristic, 2296

coerce, 2296

hash, 2296

latex, 2296

lift, 2296

one?, 2296

recip, 2296

reduce, 2296

sample, 2296

subtractIfCan, 2296

zero?, 2296

rest

ALIST, 235

DLIST, 460

GTSET, 1083

ILIST, 1231

LIST, 1504

MAGMA, 1565

NSDPS, 1703

OFMONOID, 1829

PBWL, 2054

REGSET, 2286

RGCHAIN, 2255

SREGSET, 2535

STREAM, 2582

WUTSET, 2959

RESULT, 2300

?.?, 2301

?=?, 2301

?~=?, 2301

#?, 2301

any?, 2301

bag, 2301

coerce, 2301

construct, 2301

convert, 2301

copy, 2301

count, 2301

dictionary, 2301

elt, 2301

empty, 2301

empty?, 2301

entries, 2301

entry?, 2301

eq?, 2301

- eval, [2301](#)
- every?, [2301](#)
- extract, [2301](#)
- fill, [2301](#)
- find, [2301](#)
- first, [2301](#)
- hash, [2301](#)
- index?, [2301](#)
- indices, [2301](#)
- insert, [2301](#)
- inspect, [2301](#)
- key?, [2301](#)
- keys, [2301](#)
- latex, [2301](#)
- less?, [2301](#)
- map, [2301](#)
- maxIndex, [2301](#)
- member?, [2301](#)
- members, [2301](#)
- minIndex, [2301](#)
- more?, [2301](#)
- parts, [2301](#)
- qelt, [2301](#)
- qsetelt, [2301](#)
- reduce, [2301](#)
- remove, [2301](#)
- removeDuplicates, [2301](#)
- sample, [2301](#)
- search, [2301](#)
- select, [2301](#)
- setelt, [2301](#)
- showArrayValues, [2301](#)
- showScalarValues, [2301](#)
- size?, [2301](#)
- swap, [2301](#)
- table, [2301](#)
- Result, [2300](#)
- result
 - SPLTREE, [2518](#)
- resultant
 - DMP, [572](#)
 - DSMP, [542](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - POLY, [2078](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - UP, [2828](#)
- retract
 - ALGFF, [28](#)
 - AN, [35](#)
 - ANTISYM, [40](#)
 - ASP1, [71](#)
 - ASP10, [75](#)
 - ASP19, [85](#)
 - ASP20, [93](#), [98](#)
 - ASP31, [122](#)
 - ASP35, [134](#)
 - ASP4, [140](#)
 - ASP41, [144](#)
 - ASP42, [151](#)
 - ASP49, [158](#)
 - ASP50, [164](#)
 - ASP55, [170](#)
 - ASP6, [176](#)
 - ASP7, [181](#)
 - ASP73, [186](#)
 - ASP74, [191](#)
 - ASP77, [197](#)
 - ASP78, [201](#)
 - ASP80, [211](#)
 - ASP9, [216](#)
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - CARD, [332](#)
 - CARTEN, [356](#)
 - COMPLEX, [419](#)
 - DECIMAL, [465](#)
 - DERHAM, [530](#)
 - DFLOAT, [587](#)
 - DIRPROD, [547](#)
 - DIV, [575](#)
 - DMP, [572](#)
 - DPMM, [553](#)

- DPMO, 557
DSMP, 542
EXPEXPAN, 712
EXPR, 724
FAGROUP, 1004
FAMONOID, 1007
FEXPR, 947
FF, 820
FFCG, 825
FFCGP, 835
FFCGX, 830
FFNB, 860
FFNBP, 872
FFNBX, 866
FFP, 851
FFX, 846
FGROUP, 1010
FLOAT, 908
FM1, 1017
FMONOID, 1021
FR, 786
FRAC, 986
GDMP, 1052
GPOLSET, 1074
GTSET, 1083
HACKPI, 1978
HDMP, 1180
HDP, 1173
HEXADEC, 1143
IAN, 1276
IFAMON, 1287
IFF, 1284
INT, 1362
INTRVL, 1384
IPF, 1303
IR, 1375
LAUPOL, 1422
LMOPS, 1509
LODO, 1469
LODO1, 1479
LODO2, 1491
LPOLY, 1447
LSQM, 1456
LWORD, 1532
MAGMA, 1565
MCMPLEX, 1543
MFLOAT, 1548
MINT, 1557
MODMON, 1632
MODOP, 1647, 1804
MPOLY, 1682
MRING, 1659
MYEXPR, 1688
MYUP, 1695
NIPROB, 1746
NOTTING, 1744
NSMP, 1714
NSUP, 1729
OCT, 1765
ODEPROB, 1750
ODP, 1816
ODPOL, 1852
ODVAR, 1855
OFMONOID, 1829
ONECOMP, 1777
OPTPROB, 1753
ORDCOMP, 1810
ORESUP, 2492
OREUP, 2872
PACOFF, 2136
PACRAT, 2146
PADICRAT, 1886
PADICRC, 1891
PATTERN, 1928
PBWLB, 2054
PDEPROB, 1756
PF, 2105
POLY, 2078
PR, 2092
QUAT, 2167
RADFF, 2194
RADIX, 2206
RECLOS, 2237
REGSET, 2286
RGCHAIN, 2255
ROMAN, 2326
RULE, 2305
SAE, 2400
SD, 2572
SDPOL, 2386
SDVAR, 2389
SHDP, 2509

- SINT, [2412](#)
- SMP, [2423](#)
- SQMATRIX, [2548](#)
- SREGSET, [2535](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- SYMPOLY, [2654](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSING, [2852](#)
- WUTSET, [2959](#)
- XDPOLY, [2967](#)
- XPBWPLY, [2987](#)
- XPOLY, [2998](#)
- XPR, [3007](#)
- XPOLY, [3013](#)
- retractable?
 - ANTISYM, [40](#)
 - DERHAM, [530](#)
 - LWORD, [1532](#)
 - MAGMA, [1565](#)
 - PBWL, [2054](#)
- retractIfCan
 - ALGFF, [28](#)
 - AN, [35](#)
 - ANTISYM, [40](#)
 - ASP1, [71](#)
 - ASP10, [75](#)
 - ASP19, [85](#)
 - ASP20, [93](#), [98](#)
 - ASP31, [122](#)
 - ASP35, [134](#)
 - ASP4, [140](#)
 - ASP41, [144](#)
 - ASP42, [151](#)
 - ASP49, [158](#)
 - ASP50, [164](#)
 - ASP55, [170](#)
 - ASP6, [176](#)
 - ASP7, [181](#)
 - ASP73, [186](#)
 - ASP74, [191](#)
 - ASP77, [197](#)
 - ASP78, [201](#)
 - ASP80, [211](#)
 - ASP9, [216](#)
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - CARD, [332](#)
 - CARTEN, [356](#)
 - COMPLEX, [419](#)
 - DECIMAL, [465](#)
 - DERHAM, [530](#)
 - DFLOAT, [587](#)
 - DIRPROD, [547](#)
 - DIV, [575](#)
 - DMP, [572](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - DSMP, [542](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - FAGROUP, [1004](#)
 - FAMONOID, [1007](#)
 - FEXPR, [947](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FGROUP, [1010](#)
 - FLOAT, [908](#)
 - FM1, [1017](#)
 - FMONOID, [1021](#)
 - FR, [786](#)
 - FRAC, [986](#)
 - GDMP, [1052](#)
 - GPOLSET, [1074](#)
 - GTSET, [1083](#)
 - HACKPI, [1978](#)
 - HDMP, [1180](#)
 - HDP, [1173](#)
 - HEXADEC, [1143](#)

- IAN, [1276](#)
- IFAMON, [1287](#)
- IFF, [1284](#)
- INT, [1362](#)
- INTRVL, [1384](#)
- IPF, [1303](#)
- IR, [1375](#)
- LAUPOL, [1422](#)
- LMOPS, [1509](#)
- LODO, [1469](#)
- LODO1, [1479](#)
- LODO2, [1491](#)
- LPOLY, [1447](#)
- LSQM, [1456](#)
- LWORD, [1532](#)
- MAGMA, [1565](#)
- MCMPLX, [1543](#)
- MFLOAT, [1548](#)
- MINT, [1557](#)
- MODMON, [1632](#)
- MODOP, [1647](#), [1804](#)
- MPOLY, [1682](#)
- MRING, [1659](#)
- MYEXPR, [1688](#)
- MYUP, [1695](#)
- NSMP, [1714](#)
- NSUP, [1729](#)
- OCT, [1765](#)
- ODP, [1816](#)
- ODPOL, [1852](#)
- ODVAR, [1855](#)
- OFMONOID, [1829](#)
- ONECOMP, [1777](#)
- ORDCOMP, [1810](#)
- ORESUP, [2492](#)
- OREUP, [2872](#)
- PACOFF, [2136](#)
- PACRAT, [2146](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PATTERN, [1928](#)
- PBWLb, [2054](#)
- PF, [2105](#)
- POLY, [2078](#)
- PR, [2092](#)
- QUAT, [2167](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- REGSET, [2286](#)
- RGCHAIN, [2255](#)
- ROMAN, [2326](#)
- RULE, [2305](#)
- SAE, [2400](#)
- SD, [2572](#)
- SDPOL, [2386](#)
- SDVAR, [2389](#)
- SHDP, [2509](#)
- SINT, [2412](#)
- SMP, [2423](#)
- SQMATRIX, [2548](#)
- SREGSET, [2535](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXP, [2481](#)
- SUPXS, [2487](#)
- SYMPOLY, [2654](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSsing, [2852](#)
- WUTSET, [2959](#)
- XDPOLY, [2967](#)
- XPBWPOLYL, [2987](#)
- XPOLY, [2998](#)
- XPR, [3007](#)
- XPOLY, [3013](#)
- returns
 - FC, [931](#)
- returnTypeOf
 - SYMS, [2696](#)
- reverse
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - BITS, [315](#)
 - CDFVEC, [432](#)
 - DFVEC, [605](#)
 - DLIST, [460](#)
 - FARRAY, [886](#)
 - IARRAY1, [1244](#)

- IBITS, 1199
- IFARRAY, 1222
- ILIST, 1231
- ISTRING, 1249
- IVECTOR, 1260
- LIST, 1504
- LMOPS, 1509
- POINT, 2060
- PRIMARR, 2109
- STRING, 2607
- U16VEC, 2924
- U32VEC, 2931
- U8VEC, 2917
- VECTOR, 2942
- revert
 - UFPS, 2789
 - UTS, 2877
 - UTSZ, 2887
- rewriteIdealWithHeadRemainder
 - GPOLSET, 1074
 - GTSET, 1083
 - REGSET, 2286
 - RGCHAIN, 2255
 - SREGSET, 2535
 - WUTSET, 2959
- rewriteIdealWithRemainder
 - GPOLSET, 1074
 - GTSET, 1083
 - REGSET, 2286
 - RGCHAIN, 2255
 - SREGSET, 2535
 - WUTSET, 2959
- RewriteRule, 2305
- rewriteSetWithReduction
 - GTSET, 1083
 - REGSET, 2286
 - RGCHAIN, 2255
 - SREGSET, 2535
 - WUTSET, 2959
- RGCHAIN, 2254
 - ?=?, 2255
 - ?~=?, 2255
 - #?, 2255
 - algebraic?, 2255
 - algebraicCoefficients?, 2255
 - algebraicVariables, 2255
 - any?, 2255
 - augment, 2255
 - autoReduced?, 2255
 - basicSet, 2255
 - coerce, 2255
 - coHeight, 2255
 - collect, 2255
 - collectQuasiMonic, 2255
 - collectUnder, 2255
 - collectUpper, 2255
 - construct, 2255
 - convert, 2255
 - copy, 2255
 - count, 2255
 - degree, 2255
 - empty, 2255
 - empty?, 2255
 - eq?, 2255
 - eval, 2255
 - every?, 2255
 - extend, 2255
 - extendIfCan, 2255
 - find, 2255
 - first, 2255
 - hash, 2255
 - headReduce, 2255
 - headReduced?, 2255
 - headRemainder, 2255
 - infrittWu?, 2255
 - initiallyReduce, 2255
 - initiallyReduced?, 2255
 - initials, 2255
 - internalAugment, 2255
 - intersect, 2255
 - invertible?, 2255
 - invertibleElseSplit?, 2255
 - invertibleSet, 2255
 - last, 2255
 - lastSubResultant, 2255
 - lastSubResultantElseSplit, 2255
 - latex, 2255
 - less?, 2255
 - mainVariable?, 2255
 - mainVariables, 2255
 - map, 2255
 - member?, 2255

- members, [2255](#)
- more?, [2255](#)
- mvar, [2255](#)
- normalized?, [2255](#)
- parts, [2255](#)
- purelyAlgebraic?, [2255](#)
- purelyAlgebraicLeadingMonomial?, [2255](#)
- purelyTranscendental?, [2255](#)
- quasiComponent, [2255](#)
- reduce, [2255](#)
- reduceByQuasiMonic, [2255](#)
- reduced?, [2255](#)
- remainder, [2255](#)
- remove, [2255](#)
- removeDuplicates, [2255](#)
- removeZero, [2255](#)
- rest, [2255](#)
- retract, [2255](#)
- retractIfCan, [2255](#)
- rewriteIdealWithHeadRemainder, [2255](#)
- rewriteIdealWithRemainder, [2255](#)
- rewriteSetWithReduction, [2255](#)
- roughBase?, [2255](#)
- roughEqualIdeals?, [2255](#)
- roughSubIdeal?, [2255](#)
- roughUnitIdeal?, [2255](#)
- sample, [2255](#)
- select, [2255](#)
- size?, [2255](#)
- sort, [2255](#)
- squareFreePart, [2255](#)
- stronglyReduce, [2255](#)
- stronglyReduced?, [2255](#)
- triangular?, [2255](#)
- trivialIdeal?, [2255](#)
- variables, [2255](#)
- zeroSetSplit, [2255](#)
- zeroSetSplitIntoTriangularSystems, [2255](#)
- rhs
 - EQ, [691](#)
 - RULE, [2305](#)
 - SUCH, [2627](#)
- right
 - BTREE, [310](#)
 - LWORD, [1532](#)
 - MAGMA, [1565](#)
 - OUTFORM, [1868](#)
 - PENDTREE, [1945](#)
 - ROIRC, [2310](#)
 - rightAlternative?
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
 - rightCharacteristicPolynomial
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
 - rightDiscriminant
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
 - rightDivide
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
 - rightExactQuotient
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
 - rightExtendedGcd
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
 - rightGcd
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)

- ORESUP, [2492](#)
 - OREUP, [2872](#)
- rightLcm
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
- rightMinimalPolynomial
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- rightMult
 - LMOPS, [1509](#)
- rightNorm
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- rightOne
 - EQ, [691](#)
- RightOpenIntervalRootCharacterization, [2310](#)
- rightPower
 - ALGSC, [15](#)
 - FNLA, [1027](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- rightQuotient
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
- rightRankPolynomial
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- rightRecip
 - ALGSC, [15](#)
- GCNAALG, [1065](#)
- JORDAN, [223](#)
- LIE, [229](#)
- LSQM, [1456](#)
- rightRegularRepresentation
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- rightRemainder
 - LODO, [1469](#)
 - LODO1, [1479](#)
 - LODO2, [1491](#)
 - ORESUP, [2492](#)
 - OREUP, [2872](#)
- rightTrace
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- rightTraceMatrix
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- rightTrim
 - ISTRING, [1249](#)
 - STRING, [2607](#)
- rightUnit
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- rightUnits
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- rightZero
 - EQ, [691](#)
- RittWuCompare

- NSMP, [1714](#)
- RMATRIX, [2245](#)
 - ?, [2246](#)
 - ?*?, [2246](#)
 - ?+?, [2246](#)
 - ?-?, [2246](#)
 - ?/? , [2246](#)
 - ?=?, [2246](#)
 - ?~=?, [2246](#)
 - #?, [2246](#)
 - 0, [2246](#)
 - antisymmetric?, [2246](#)
 - any?, [2246](#)
 - coerce, [2246](#)
 - column, [2246](#)
 - convert, [2246](#)
 - copy, [2246](#)
 - count, [2246](#)
 - diagonal?, [2246](#)
 - dimension, [2246](#)
 - elt, [2246](#)
 - empty, [2246](#)
 - empty?, [2246](#)
 - eq?, [2246](#)
 - eval, [2246](#)
 - every?, [2246](#)
 - exquo, [2246](#)
 - hash, [2246](#)
 - latex, [2246](#)
 - less?, [2246](#)
 - listOfLists, [2246](#)
 - map, [2246](#)
 - matrix, [2246](#)
 - maxColIndex, [2246](#)
 - maxRowIndex, [2246](#)
 - member?, [2246](#)
 - members, [2246](#)
 - minColIndex, [2246](#)
 - minRowIndex, [2246](#)
 - more?, [2246](#)
 - ncols, [2246](#)
 - nrows, [2246](#)
 - nullity, [2246](#)
 - nullSpace, [2246](#)
 - parts, [2246](#)
 - qelt, [2246](#)
 - rank, [2246](#)
 - rectangularMatrix, [2246](#)
 - row, [2246](#)
 - rowEchelon, [2246](#)
 - sample, [2246](#)
 - size?, [2246](#)
 - square?, [2246](#)
 - subtractIfCan, [2246](#)
 - symmetric?, [2246](#)
 - zero?, [2246](#)
- ROIRC, [2310](#)
 - ?=?, [2310](#)
 - ?~=?, [2310](#)
 - allRootsOf, [2310](#)
 - approximate, [2310](#)
 - coerce, [2310](#)
 - definingPolynomial, [2310](#)
 - hash, [2310](#)
 - latex, [2310](#)
 - left, [2310](#)
 - middle, [2310](#)
 - mightHaveRoots, [2310](#)
 - negative?, [2310](#)
 - positive?, [2310](#)
 - recip, [2310](#)
 - refine, [2310](#)
 - relativeApprox, [2310](#)
 - right, [2310](#)
 - rootOf, [2310](#)
 - sign, [2310](#)
 - size, [2310](#)
 - zero?, [2310](#)
- ROMAN, [2326](#)
 - ?, [2326](#)
 - ?<?, [2326](#)
 - ?<=?, [2326](#)
 - ?>?, [2326](#)
 - ?>=?, [2326](#)
 - ?**?, [2326](#)
 - ?*?, [2326](#)
 - ?+?, [2326](#)
 - ?-?, [2326](#)
 - ?=?, [2326](#)
 - ?^?, [2326](#)
 - ?~=?, [2326](#)
 - ?quo?, [2326](#)

- [?rem?](#), [2326](#)
- [0](#), [2326](#)
- [1](#), [2326](#)
- [abs](#), [2326](#)
- [addmod](#), [2326](#)
- [associates?](#), [2326](#)
- [base](#), [2326](#)
- [binomial](#), [2326](#)
- [bit?](#), [2326](#)
- [characteristic](#), [2326](#)
- [coerce](#), [2326](#)
- [convert](#), [2326](#)
- [copy](#), [2326](#)
- [D](#), [2326](#)
- [dec](#), [2326](#)
- [differentiate](#), [2326](#)
- [divide](#), [2326](#)
- [euclideanSize](#), [2326](#)
- [even?](#), [2326](#)
- [expressIdealMember](#), [2326](#)
- [exquo](#), [2326](#)
- [extendedEuclidean](#), [2326](#)
- [factor](#), [2326](#)
- [factorial](#), [2326](#)
- [gcd](#), [2326](#)
- [gcdPolynomial](#), [2326](#)
- [hash](#), [2326](#)
- [inc](#), [2326](#)
- [init](#), [2326](#)
- [invmod](#), [2326](#)
- [latex](#), [2326](#)
- [lcm](#), [2326](#)
- [length](#), [2326](#)
- [mask](#), [2326](#)
- [max](#), [2326](#)
- [min](#), [2326](#)
- [mulmod](#), [2326](#)
- [multiEuclidean](#), [2326](#)
- [negative?](#), [2326](#)
- [nextItem](#), [2326](#)
- [odd?](#), [2326](#)
- [one?](#), [2326](#)
- [patternMatch](#), [2326](#)
- [permutation](#), [2326](#)
- [positive?](#), [2326](#)
- [positiveRemainder](#), [2326](#)
- [powmod](#), [2326](#)
- [prime?](#), [2326](#)
- [principalIdeal](#), [2326](#)
- [random](#), [2326](#)
- [rational](#), [2326](#)
- [rational?](#), [2326](#)
- [rationalIfCan](#), [2326](#)
- [recip](#), [2326](#)
- [reducedSystem](#), [2326](#)
- [retract](#), [2326](#)
- [retractIfCan](#), [2326](#)
- [roman](#), [2326](#)
- [sample](#), [2326](#)
- [shift](#), [2326](#)
- [sign](#), [2326](#)
- [sizeLess?](#), [2326](#)
- [squareFree](#), [2326](#)
- [squareFreePart](#), [2326](#)
- [submod](#), [2326](#)
- [subtractIfCan](#), [2326](#)
- [symmetricRemainder](#), [2326](#)
- [unit?](#), [2326](#)
- [unitCanonical](#), [2326](#)
- [unitNormal](#), [2326](#)
- [zero?](#), [2326](#)
- [roman](#)
 - [ROMAN](#), [2326](#)
- [RomanNumeral](#), [2326](#)
- [root](#)
 - [BPADIC](#), [258](#)
 - [IPADIC](#), [1294](#)
 - [OUTFORM](#), [1868](#)
 - [PADIC](#), [1881](#)
- [root?](#)
 - [SUBSPACE](#), [2614](#)
- [rootOf](#)
 - [AN](#), [35](#)
 - [EXPR](#), [724](#)
 - [IAN](#), [1276](#)
 - [RECLOS](#), [2237](#)
 - [ROIRC](#), [2310](#)
- [rootsOf](#)
 - [AN](#), [35](#)
 - [EXPR](#), [724](#)
 - [IAN](#), [1276](#)
- [rotate](#)

- VIEW3D, [2710](#)
- rotatex
 - DHMATRIX, [491](#)
- rotatey
 - DHMATRIX, [491](#)
- rotatez
 - DHMATRIX, [491](#)
- roughBase?
 - GPOLSET, [1074](#)
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- roughEqualIdeals?
 - GPOLSET, [1074](#)
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- roughSubIdeal?
 - GPOLSET, [1074](#)
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- roughUnitIdeal?
 - GPOLSET, [1074](#)
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- round
 - DFLOAT, [587](#)
 - FLOAT, [908](#)
 - MFLOAT, [1548](#)
- ROUTINE, [2330](#)
- ?.?, [2331](#)
- ?=?, [2331](#)
- ?~=?, [2331](#)
- #?, [2331](#)
- any?, [2331](#)
- bag, [2331](#)
- changeMeasure, [2331](#)
- changeThreshold, [2331](#)
- coerce, [2331](#)
- concat, [2331](#)
- construct, [2331](#)
- convert, [2331](#)
- copy, [2331](#)
- count, [2331](#)
- deleteRoutine, [2331](#)
- dictionary, [2331](#)
- elt, [2331](#)
- empty, [2331](#)
- empty?, [2331](#)
- entries, [2331](#)
- entry?, [2331](#)
- eq?, [2331](#)
- eval, [2331](#)
- every?, [2331](#)
- extract, [2331](#)
- fill, [2331](#)
- find, [2331](#)
- first, [2331](#)
- getExplanations, [2331](#)
- getMeasure, [2331](#)
- hash, [2331](#)
- index?, [2331](#)
- indices, [2331](#)
- insert, [2331](#)
- inspect, [2331](#)
- key?, [2331](#)
- keys, [2331](#)
- latex, [2331](#)
- less?, [2331](#)
- map, [2331](#)
- maxIndex, [2331](#)
- member?, [2331](#)
- members, [2331](#)
- minIndex, [2331](#)
- more?, [2331](#)
- parts, [2331](#)
- qelt, [2331](#)
- qsetelt, [2331](#)
- recoverAfterFail, [2331](#)
- reduce, [2331](#)
- remove, [2331](#)
- removeDuplicates, [2331](#)

- routines, [2331](#)
 - sample, [2331](#)
 - search, [2331](#)
 - select, [2331](#)
 - selectFiniteRoutines, [2331](#)
 - selectIntegrationRoutines, [2331](#)
 - selectMultiDimensionalRoutines, [2331](#)
 - selectNonFiniteRoutines, [2331](#)
 - selectODEIVPRoutines, [2331](#)
 - selectOptimizationRoutines, [2331](#)
 - selectPDERoutines, [2331](#)
 - selectSumOfSquaresRoutines, [2331](#)
 - setelt, [2331](#)
 - showTheRoutinesTable, [2331](#)
 - size?, [2331](#)
 - swap, [2331](#)
 - table, [2331](#)
- routines
 - ROUTINE, [2331](#)
- RoutinesTable, [2330](#)
- row
 - ARRAY2, [2763](#)
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IARRAY2, [1256](#)
 - IIARRAY2, [1290](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - RMATRIX, [2246](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- rowEchelon
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - RMATRIX, [2246](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- rquo
 - FMONOID, [1021](#)
 - LPOLY, [1447](#)
 - OFMONOID, [1829](#)
 - XDPOLY, [2967](#)
 - XPBWPOLYL, [2987](#)
 - XPOLY, [2998](#)
 - XRPOLY, [3013](#)
- rspace
 - OUTFORM, [1868](#)
- rst
 - NSDPS, [1703](#)
 - STREAM, [2582](#)
- RULE, [2305](#)
 - ?.?, [2305](#)
 - ?=?, [2305](#)
 - ?~=?, [2305](#)
 - coerce, [2305](#)
 - elt, [2305](#)
 - hash, [2305](#)
 - latex, [2305](#)
 - lhs, [2305](#)
 - pattern, [2305](#)
 - quotedOperators, [2305](#)
 - retract, [2305](#)
 - retractIfCan, [2305](#)
 - rhs, [2305](#)
 - rule, [2305](#)
 - suchThat, [2305](#)
- rule
 - RULE, [2305](#)
- RuleCalled, [2340](#)
- RULECOLD, [2340](#)
 - ?=?, [2340](#)
 - ?~=?, [2340](#)
 - coerce, [2340](#)
 - hash, [2340](#)
 - latex, [2340](#)
 - name, [2340](#)
- rules
 - RULESET, [2342](#)
- RULESET, [2342](#)
 - ?.?, [2342](#)
 - ?=?, [2342](#)
 - ?~=?, [2342](#)

- coerce, 2342
- elt, 2342
- hash, 2342
- latex, 2342
- rules, 2342
- ruleset, 2342
- Ruleset, 2342
- ruleset
 - RULESET, 2342
- SAE, 2399
- ?, 2400
- ?**?, 2400
- ?*?, 2400
- ?+?, 2400
- ?-?, 2400
- ?/?, 2400
- ?=?, 2400
- ?^?, 2400
- ?~=?, 2400
- ?quo?, 2400
- ?rem?, 2400
- 0, 2400
- 1, 2400
- associates?, 2400
- basis, 2400
- characteristic, 2400
- characteristicPolynomial, 2400
- charthRoot, 2400
- coerce, 2400
- conditionP, 2400
- convert, 2400
- coordinates, 2400
- createPrimitiveElement, 2400
- D, 2400
- definingPolynomial, 2400
- derivationCoordinates, 2400
- differentiate, 2400
- discreteLog, 2400
- discriminant, 2400
- divide, 2400
- euclideanSize, 2400
- expressIdealMember, 2400
- exquo, 2400
- extendedEuclidean, 2400
- factor, 2400
- factorsOfCyclicGroupSize, 2400
- gcd, 2400
- gcdPolynomial, 2400
- generator, 2400
- hash, 2400
- index, 2400
- init, 2400
- inv, 2400
- latex, 2400
- lcm, 2400
- lift, 2400
- lookup, 2400
- minimalPolynomial, 2400
- multiEuclidean, 2400
- nextItem, 2400
- norm, 2400
- one?, 2400
- order, 2400
- prime?, 2400
- primeFrobenius, 2400
- primitive?, 2400
- primitiveElement, 2400
- principalIdeal, 2400
- random, 2400
- rank, 2400
- recip, 2400
- reduce, 2400
- reducedSystem, 2400
- regularRepresentation, 2400
- representationType, 2400
- represents, 2400
- retract, 2400
- retractIfCan, 2400
- sample, 2400
- size, 2400
- sizeLess?, 2400
- squareFree, 2400
- squareFreePart, 2400
- subtractIfCan, 2400
- tableForDiscreteLogarithm, 2400
- trace, 2400
- traceMatrix, 2400
- unit?, 2400
- unitCanonical, 2400
- unitNormal, 2400
- zero?, 2400

safety

GOPT, 1104
GOPT0, 1110

sample

ALGFF, 28
ALGSC, 15
ALIST, 235
AN, 35
ANTISYM, 40
ARRAY1, 1774
ARRAY2, 2763
ASTACK, 65
AUTOMOR, 245
BBTREE, 253
BINARY, 292
BITS, 315
BPADIC, 258
BPADICRT, 263
BSTREE, 302
BTourn, 306
BTREE, 310
CARD, 332
CARTEN, 356
CCLASS, 382
CDFMAT, 426
CDFVEC, 432
CLIF, 402
COMPLEX, 419
CONTFRAC, 445
DECIMAL, 465
DEQUEUE, 512
DERHAM, 530
DFLOAT, 587
DFMAT, 599
DFVEC, 605
DHMATRIX, 491
DIRPROD, 547
DIRRING, 564
DIV, 575
DLIST, 460
DMP, 572
DPMM, 553
DPMO, 557
DSMP, 542
DSTREE, 535
EMR, 702

EQ, 691
EQTBL, 699
EXPEXPAN, 712
EXPR, 724
EXPUPXS, 740
FAGROUP, 1004
FAMONOID, 1007
FARRAY, 886
FDIV, 813
FEXPR, 947
FF, 820
FFCG, 825
FFCGP, 835
FFCGX, 830
FFNB, 860
FFNBP, 872
FFNBX, 866
FFP, 851
FFX, 846
FGROUP, 1010
FLOAT, 908
FM, 1013
FM1, 1017
FMONOID, 1021
FNLA, 1027
FR, 786
FRAC, 986
FRIDEAL, 995
FRMOD, 1000
FSERIES, 978
GCNAALG, 1065
GDMP, 1052
GMODPOL, 1059
GPOLSET, 1074
GSERIES, 1090
GSTBL, 1079
GTSET, 1083
HACKPI, 1978
HASHTBL, 1120
HDMP, 1180
HDP, 1173
HEAP, 1134
HELLFDIV, 1183
HEXADEC, 1143
IAN, 1276
IARRAY1, 1244

- IARRAY2, [1256](#)
- IBITS, [1199](#)
- IDPAG, [1202](#)
- IDPAM, [1206](#)
- IDPOAM, [1212](#)
- IDPOAMS, [1215](#)
- IFAMON, [1287](#)
- IFARRAY, [1222](#)
- IFF, [1284](#)
- IIARRAY2, [1290](#)
- ILIST, [1231](#)
- IMATRIX, [1239](#)
- INDE, [1218](#)
- INT, [1362](#)
- INTABL, [1336](#)
- INTRVL, [1384](#)
- IPADIC, [1294](#)
- IPF, [1303](#)
- IR, [1375](#)
- ISTRING, [1249](#)
- ISUPS, [1311](#)
- ITAYLOR, [1338](#)
- IVECTOR, [1260](#)
- JORDAN, [223](#)
- KAFILE, [1414](#)
- LA, [1520](#)
- LAUPOL, [1422](#)
- LEXP, [1435](#)
- LIB, [1429](#)
- LIE, [229](#)
- LIST, [1504](#)
- LMDICT, [1515](#)
- LO, [1523](#)
- LODO, [1469](#)
- LODO1, [1479](#)
- LODO2, [1491](#)
- LPOLY, [1447](#)
- LSQM, [1456](#)
- M3D, [2703](#)
- MATRIX, [1623](#)
- MCMPLEX, [1543](#)
- MFLOAT, [1548](#)
- MINT, [1557](#)
- MODFIELD, [1638](#)
- MODMON, [1632](#)
- MODOP, [1647](#), [1804](#)
- MODRING, [1641](#)
- MOEBIUS, [1654](#)
- MPOLY, [1682](#)
- MRING, [1659](#)
- MSET, [1670](#)
- MYEXPR, [1688](#)
- MYUP, [1695](#)
- NNI, [1739](#)
- NOTTING, [1744](#)
- NSDPS, [1703](#)
- NSMP, [1714](#)
- NSUP, [1729](#)
- OCT, [1765](#)
- ODP, [1816](#)
- ODPOL, [1852](#)
- ODR, [1858](#)
- OFMONOID, [1829](#)
- OMLO, [1806](#)
- ONECOMP, [1777](#)
- ORDCOMP, [1810](#)
- ORESUP, [2492](#)
- OREUP, [2872](#)
- OWP, [1861](#)
- PACOFF, [2136](#)
- PACRAT, [2146](#)
- PADIC, [1881](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PENDTREE, [1945](#)
- PERM, [1949](#)
- PF, [2105](#)
- PFR, [1914](#)
- PI, [2100](#)
- POINT, [2060](#)
- POLY, [2078](#)
- PR, [2092](#)
- PRIMARR, [2109](#)
- PRODUCT, [2113](#)
- PRITITION, [1923](#)
- QFORM, [2154](#)
- QUAT, [2167](#)
- QUEUE, [2184](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- REGSET, [2286](#)

- RESRING, [2296](#)
- RESULT, [2301](#)
- RGCHAIN, [2255](#)
- RMATRIX, [2246](#)
- ROMAN, [2326](#)
- ROUTINE, [2331](#)
- SAE, [2400](#)
- SD, [2572](#)
- SDPOL, [2386](#)
- SET, [2372](#)
- SHDP, [2509](#)
- SINT, [2412](#)
- SMP, [2423](#)
- SMTS, [2441](#)
- SPLTREE, [2518](#)
- SQMATRIX, [2548](#)
- SREGSET, [2535](#)
- STACK, [2563](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUEXPR, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- SYMBOL, [2640](#)
- SYMPOLY, [2654](#)
- TABLE, [2664](#)
- TREE, [2741](#)
- TS, [2671](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- VECTOR, [2942](#)
- WP, [2949](#)
- WUTSET, [2959](#)
- XDPOLY, [2967](#)
- XPBWPOLYL, [2987](#)
- XPOLY, [2998](#)
- XPR, [3007](#)
- XPOLY, [3013](#)
- ZMOD, [1368](#)
- SAOS, [2418](#)
 - ?<?, [2418](#)
 - ?<=?, [2418](#)
 - ?>?, [2418](#)
 - ?>=?, [2418](#)
 - ?=?, [2418](#)
 - ?~=?, [2418](#)
 - coerce, [2418](#)
 - convert, [2418](#)
 - create, [2418](#)
 - hash, [2418](#)
 - latex, [2418](#)
 - max, [2418](#)
 - min, [2418](#)
- satisfy?
 - PATRES, [1940](#)
- saturate
 - IDEAL, [2081](#)
- save
 - FC, [931](#)
- sbt
 - NSDPS, [1703](#)
- scalarMatrix
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- scalarTypeOf
 - FT, [971](#)
- scale

- DHMATRIX, [491](#)
- MOEBIUS, [1654](#)
- VIEW2d, [2770](#)
- screenResolution
 - PLOT, [2029](#)
- screenResolution3D
 - PLOT3D, [2042](#)
- script
 - SYMBOL, [2640](#)
- scripted?
 - SYMBOL, [2640](#)
- ScriptFormulaFormat, [2346](#)
- scripts
 - OUTFORM, [1868](#)
 - SYMBOL, [2640](#)
- SD, [2572](#)
 - , [2572](#)
 - ***?, [2572](#)
 - *?*, [2572](#)
 - ?+?, [2572](#)
 - ?-?, [2572](#)
 - ?/? , [2572](#)
 - ?=?, [2572](#)
 - ?^?, [2572](#)
 - ?~=?, [2572](#)
 - 0, [2572](#)
 - alterDrift, [2572](#)
 - alterQuadVar, [2572](#)
 - coefficient, [2572](#)
 - coerce, [2572](#)
 - copyDrift, [2572](#)
 - copyQuadVar, [2572](#)
 - drift, [2572](#)
 - equation, [2572](#)
 - freeOf?, [2572](#)
 - hash, [2572](#)
 - latex, [2572](#)
 - listSD, [2572](#)
 - retract, [2572](#)
 - retractIfCan, [2572](#)
 - sample, [2572](#)
 - statusIto, [2572](#)
 - subtractIfCan, [2572](#)
 - uncorrelated?, [2572](#)
 - zero?, [2572](#)
- SDPOL, [2385](#)
- , [2386](#)
- ?<?, [2386](#)
- ?<=?, [2386](#)
- ?>?, [2386](#)
- ?>=?, [2386](#)
- ***?, [2386](#)
- *?*, [2386](#)
- ?+?, [2386](#)
- ?-?, [2386](#)
- ?/? , [2386](#)
- ?=?, [2386](#)
- ?^?, [2386](#)
- ?~=?, [2386](#)
- 0, [2386](#)
- 1, [2386](#)
- associates?, [2386](#)
- binomThmExpt, [2386](#)
- characteristic, [2386](#)
- charthRoot, [2386](#)
- coefficient, [2386](#)
- coefficients, [2386](#)
- coerce, [2386](#)
- conditionP, [2386](#)
- content, [2386](#)
- convert, [2386](#)
- D, [2386](#)
- degree, [2386](#)
- differentialVariables, [2386](#)
- differentiate, [2386](#)
- discriminant, [2386](#)
- eval, [2386](#)
- exquo, [2386](#)
- factor, [2386](#)
- factorPolynomial, [2386](#)
- factorSquareFreePolynomial, [2386](#)
- gcd, [2386](#)
- gcdPolynomial, [2386](#)
- ground, [2386](#)
- ground?, [2386](#)
- hash, [2386](#)
- initial, [2386](#)
- isExpt, [2386](#)
- isobaric?, [2386](#)
- isPlus, [2386](#)
- isTimes, [2386](#)
- latex, [2386](#)

- lcm, [2386](#)
- leader, [2386](#)
- leadingCoefficient, [2386](#)
- leadingMonomial, [2386](#)
- mainVariable, [2386](#)
- makeVariable, [2386](#)
- map, [2386](#)
- mapExponents, [2386](#)
- max, [2386](#)
- min, [2386](#)
- minimumDegree, [2386](#)
- monicDivide, [2386](#)
- monomial, [2386](#)
- monomial?, [2386](#)
- monomials, [2386](#)
- multivariate, [2386](#)
- numberOfMonomials, [2386](#)
- one?, [2386](#)
- order, [2386](#)
- patternMatch, [2386](#)
- pomopo, [2386](#)
- prime?, [2386](#)
- primitiveMonomials, [2386](#)
- primitivePart, [2386](#)
- recip, [2386](#)
- reducedSystem, [2386](#)
- reductum, [2386](#)
- resultant, [2386](#)
- retract, [2386](#)
- retractIfCan, [2386](#)
- sample, [2386](#)
- separant, [2386](#)
- solveLinearPolynomialEquation, [2386](#)
- squareFree, [2386](#)
- squareFreePart, [2386](#)
- squareFreePolynomial, [2386](#)
- subtractIfCan, [2386](#)
- totalDegree, [2386](#)
- unit?, [2386](#)
- unitCanonical, [2386](#)
- unitNormal, [2386](#)
- univariate, [2386](#)
- variables, [2386](#)
- weight, [2386](#)
- weights, [2386](#)
- zero?, [2386](#)
- SDVAR, [2389](#)
 - ?<?, [2389](#)
 - ?<=?, [2389](#)
 - ?>?, [2389](#)
 - ?>=?, [2389](#)
 - ?=?, [2389](#)
 - ?~=?, [2389](#)
 - coerce, [2389](#)
 - differentiate, [2389](#)
 - hash, [2389](#)
 - latex, [2389](#)
 - makeVariable, [2389](#)
 - max, [2389](#)
 - min, [2389](#)
 - order, [2389](#)
 - retract, [2389](#)
 - retractIfCan, [2389](#)
 - variable, [2389](#)
 - weight, [2389](#)
- search
 - ALIST, [235](#)
 - EQTBL, [699](#)
 - GSTBL, [1079](#)
 - HASHTBL, [1120](#)
 - INTABL, [1336](#)
 - KAFILE, [1414](#)
 - LIB, [1429](#)
 - RESULT, [2301](#)
 - ROUTINE, [2331](#)
 - STBL, [2450](#)
 - STRTBL, [2610](#)
 - TABLE, [2664](#)
- sec
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLEX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXPR, [2481](#)

- SUPXS, 2487
- SUTS, 2497
- TS, 2671
- UFPS, 2789
- ULS, 2796
- ULSCONS, 2804
- UPXS, 2834
- UPXSCONS, 2842
- UTS, 2877
- UTSZ, 2887
- sech
 - COMPLEX, 419
 - DFLOAT, 587
 - EFULS, 672
 - EFUPXS, 681
 - EXPR, 724
 - EXPUPXS, 740
 - FLOAT, 908
 - GSERIES, 1090
 - INTRVL, 1384
 - MCMPLX, 1543
 - SMTS, 2441
 - SULS, 2457
 - SUPEXPR, 2481
 - SUPXS, 2487
 - SUTS, 2497
 - TS, 2671
 - UFPS, 2789
 - ULS, 2796
 - ULSCONS, 2804
 - UPXS, 2834
 - UPXSCONS, 2842
 - UTS, 2877
 - UTSZ, 2887
- sechIfCan
 - EFULS, 672
 - EFUPXS, 681
- secIfCan
 - EFULS, 672
 - EFUPXS, 681
- second
 - ALIST, 235
 - DLIST, 460
 - ILIST, 1231
 - LIST, 1504
 - NSDPS, 1703
 - STREAM, 2582
- SEG, 2359
 - ?..?, 2359
 - ?=?, 2359
 - ?~=?, 2359
 - BY, 2359
 - coerce, 2359
 - convert, 2359
 - expand, 2359
 - hash, 2359
 - hi, 2359
 - high, 2359
 - incr, 2359
 - latex, 2359
 - lo, 2359
 - low, 2359
 - map, 2359
 - segment, 2359
- SEGBIND, 2364
 - ?=?, 2364
 - ?~=?, 2364
 - coerce, 2364
 - equation, 2364
 - hash, 2364
 - latex, 2364
 - segment, 2364
 - variable, 2364
- Segment, 2359
- segment
 - SEG, 2359
 - SEGBIND, 2364
 - UNISEG, 2896
- SegmentBinding, 2364
- select
 - ALIST, 235
 - ARRAY1, 1774
 - CCLASS, 382
 - CDFVEC, 432
 - DFVEC, 605
 - DLIST, 460
 - EQTBL, 699
 - FARRAY, 886
 - GPOLSET, 1074
 - GSTBL, 1079
 - GTSET, 1083
 - HASHTBL, 1120

- IARRAY1, [1244](#)
- IBITS, [1199](#)
- IFARRAY, [1222](#)
- ILIST, [1231](#)
- INTABL, [1336](#)
- ISTRING, [1249](#)
- ITUPLE, [1262](#)
- IVECTOR, [1260](#)
- KAFILE, [1414](#)
- LIB, [1429](#)
- LIST, [1504](#)
- LM_DICT, [1515](#)
- MSET, [1670](#)
- NSDPS, [1703](#)
- POINT, [2060](#)
- PRIMARR, [2109](#)
- REGSET, [2286](#)
- RESULT, [2301](#)
- RGCHAIN, [2255](#)
- ROUTINE, [2331](#)
- SET, [2372](#)
- SREGSET, [2535](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- TUPLE, [2752](#)
- U16VEC, [2924](#)
- U32VEC, [2931](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- WUTSET, [2959](#)
- selectFiniteRoutines
 - ROUTINE, [2331](#)
- selectfirst
 - PRODUCT, [2113](#)
- selectIntegrationRoutines
 - ROUTINE, [2331](#)
- selectMultiDimensionalRoutines
 - ROUTINE, [2331](#)
- selectNonFiniteRoutines
 - ROUTINE, [2331](#)
- selectODEIVPRoutines
 - ROUTINE, [2331](#)
- selectOptimizationRoutines
 - ROUTINE, [2331](#)
- selectPDERoutines
 - ROUTINE, [2331](#)
- selectsecond
 - PRODUCT, [2113](#)
- selectSumOfSquaresRoutines
 - ROUTINE, [2331](#)
- semicolonSeparate
 - OUTFORM, [1868](#)
- separant
 - DSMP, [542](#)
 - ODPOL, [1852](#)
 - SDPOL, [2386](#)
- separate
 - LAUPOL, [1422](#)
 - MODMON, [1632](#)
 - MYUP, [1695](#)
 - NSUP, [1729](#)
 - SUBSPACE, [2614](#)
 - SUP, [2467](#)
 - SUPEXP, [2481](#)
 - UP, [2828](#)
- SequentialDifferentialPolynomial, [2385](#)
- SequentialDifferentialVariable, [2389](#)
- series
 - EXPUPXS, [740](#)
 - GSERIES, [1090](#)
 - ISUPS, [1311](#)
 - ITAYLOR, [1338](#)
 - NSDPS, [1703](#)
 - SULS, [2457](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- seriesToOutputForm
 - ISUPS, [1311](#)
- SET, [2372](#)
 - ?<?, [2372](#)
 - ?=?, [2372](#)
 - ?~=?, [2372](#)

- [#?](#), [2372](#)
- [any?](#), [2372](#)
- [bag](#), [2372](#)
- [brace](#), [2372](#)
- [cardinality](#), [2372](#)
- [coerce](#), [2372](#)
- [complement](#), [2372](#)
- [construct](#), [2372](#)
- [convert](#), [2372](#)
- [copy](#), [2372](#)
- [count](#), [2372](#)
- [dictionary](#), [2372](#)
- [difference](#), [2372](#)
- [empty](#), [2372](#)
- [empty?](#), [2372](#)
- [eq?](#), [2372](#)
- [eval](#), [2372](#)
- [every?](#), [2372](#)
- [extract](#), [2372](#)
- [find](#), [2372](#)
- [hash](#), [2372](#)
- [index](#), [2372](#)
- [insert](#), [2372](#)
- [inspect](#), [2372](#)
- [intersect](#), [2372](#)
- [latex](#), [2372](#)
- [less?](#), [2372](#)
- [lookup](#), [2372](#)
- [map](#), [2372](#)
- [max](#), [2372](#)
- [member?](#), [2372](#)
- [members](#), [2372](#)
- [min](#), [2372](#)
- [more?](#), [2372](#)
- [parts](#), [2372](#)
- [random](#), [2372](#)
- [reduce](#), [2372](#)
- [remove](#), [2372](#)
- [removeDuplicates](#), [2372](#)
- [sample](#), [2372](#)
- [select](#), [2372](#)
- [set](#), [2372](#)
- [size](#), [2372](#)
- [size?](#), [2372](#)
- [subset?](#), [2372](#)
- [symmetricDifference](#), [2372](#)
- [union](#), [2372](#)
- [universe](#), [2372](#)
- [Set](#), [2372](#)
- [set](#)
 - [CCLASS](#), [382](#)
 - [MSET](#), [1670](#)
 - [SET](#), [2372](#)
- [setAdaptive](#)
 - [PLOT](#), [2029](#)
- [setAdaptive3D](#)
 - [PLOT3D](#), [2042](#)
- [setAttributeButtonStep](#)
 - [ATTRBUT](#), [239](#)
- [setButtonValue](#)
 - [ATTRBUT](#), [239](#)
- [setClosed](#)
 - [TUBE](#), [2749](#)
- [setDifference](#)
 - [LIST](#), [1504](#)
- [setelt](#)
 - [AFFPLPS](#), [7](#)
 - [AFFSP](#), [9](#)
 - [ALIST](#), [235](#)
 - [ARRAY1](#), [1774](#)
 - [ARRAY2](#), [2763](#)
 - [BBTREE](#), [253](#)
 - [BITS](#), [315](#)
 - [BSTREE](#), [302](#)
 - [BTourn](#), [306](#)
 - [BTREE](#), [310](#)
 - [CDFMAT](#), [426](#)
 - [CDFVEC](#), [432](#)
 - [DFMAT](#), [599](#)
 - [DFVEC](#), [605](#)
 - [DHMATRIX](#), [491](#)
 - [DIRPROD](#), [547](#)
 - [DLIST](#), [460](#)
 - [DPMM](#), [553](#)
 - [DPMO](#), [557](#)
 - [DSTREE](#), [535](#)
 - [EQTBL](#), [699](#)
 - [FARRAY](#), [886](#)
 - [GSTBL](#), [1079](#)
 - [HASHTBL](#), [1120](#)
 - [HDP](#), [1173](#)
 - [IARRAY1](#), [1244](#)

- IARRAY2, [1256](#)
- IFARRAY, [1222](#)
- IIARRAY2, [1290](#)
- ILIST, [1231](#)
- IMATRIX, [1239](#)
- INTABL, [1336](#)
- ISTRING, [1249](#)
- IVECTOR, [1260](#)
- KAFILE, [1414](#)
- LIB, [1429](#)
- LIST, [1504](#)
- MATRIX, [1623](#)
- NSDPS, [1703](#)
- ODP, [1816](#)
- PENDTREE, [1945](#)
- POINT, [2060](#)
- PRIMARR, [2109](#)
- PROJPL, [2117](#)
- PROJPLPS, [2119](#)
- PROJSP, [2122](#)
- REF, [2249](#)
- RESULT, [2301](#)
- ROUTINE, [2331](#)
- SHDP, [2509](#)
- SPLTREE, [2518](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- TREE, [2741](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- setFoundPlacesToEmpty
 - PLACES, [2019](#)
 - PLACESPS, [2021](#)
- setIntersection
 - LIST, [1504](#)
- setLabelValue
 - FC, [931](#)
- setMaxPoints
 - PLOT, [2029](#)
- setMaxPoints3D
 - PLOT3D, [2042](#)
- setMinPoints
 - PLOT, [2029](#)
- setMinPoints3D
 - PLOT3D, [2042](#)
- SETMN, [2377](#)
 - ?=?, [2377](#)
 - ?~=?, [2377](#)
 - coerce, [2377](#)
 - delta, [2377](#)
 - elements, [2377](#)
 - enumerate, [2377](#)
 - hash, [2377](#)
 - incrementKthElement, [2377](#)
 - index, [2377](#)
 - latex, [2377](#)
 - lookup, [2377](#)
 - member?, [2377](#)
 - random, [2377](#)
 - replaceKthElement, [2377](#)
 - setOfMinN, [2377](#)
 - size, [2377](#)
- setOfMinN
 - SETMN, [2377](#)
- SetOfMIntegersInOneToN, [2377](#)
- setPoly
 - MODMON, [1632](#)
- setPosition
 - KERNEL, [1404](#)
 - MKCHSET, [1570](#)
- setPredicates
 - PATTERN, [1928](#)
- setProperty
 - BOP, [274](#)
- setProperty
 - BOP, [274](#)
- setref
 - REF, [2249](#)
- setScreenResolution
 - PLOT, [2029](#)
- setScreenResolution3D
 - PLOT3D, [2042](#)
- setStatus
 - QALGSET, [2157](#)

- setTopPredicate
 - PATTERN, 1928
- setUnion
 - LIST, 1504
- SEX, 2391
 - ?.?, 2392
 - ?=?, 2392
 - ?~=?, 2392
 - #?, 2392
 - atom?, 2392
 - car, 2392
 - cdr, 2392
 - coerce, 2392
 - convert, 2392
 - destruct, 2392
 - eq, 2392
 - expr, 2392
 - float, 2392
 - float?, 2392
 - hash, 2392
 - integer, 2392
 - integer?, 2392
 - latex, 2392
 - list?, 2392
 - null?, 2392
 - pair?, 2392
 - string, 2392
 - string?, 2392
 - symbol, 2392
 - symbol?, 2392
- SEXOF, 2394
 - ?.?, 2394
 - ?=?, 2394
 - ?~=?, 2394
 - #?, 2394
 - atom?, 2394
 - car, 2394
 - cdr, 2394
 - coerce, 2394
 - convert, 2394
 - destruct, 2394
 - eq, 2394
 - expr, 2394
 - float, 2394
 - float?, 2394
 - hash, 2394
 - integer, 2394
 - integer?, 2394
 - latex, 2394
 - list?, 2394
 - null?, 2394
 - pair?, 2394
 - string, 2394
 - string?, 2394
 - symbol, 2394
 - symbol?, 2394
- SExpression, 2391
- SExpressionOf, 2394
- SFORT, 2405
 - coerce, 2405
 - fortran, 2405
 - outputAsFortran, 2405
- sh
 - XDPOLY, 2967
 - XPBWPLY, 2987
 - XPOLY, 2998
 - XRPOLY, 3013
- shade
 - PALETTE, 1896
- shallowCopy
 - SUBSPACE, 2614
- shallowExpand
 - FNLA, 1027
- SHDP, 2508
 - .?, 2509
 - ?<?, 2509
 - ?<=?, 2509
 - ?>?, 2509
 - ?>=?, 2509
 - ?**?, 2509
 - ?*?, 2509
 - ?+?, 2509
 - ?-?, 2509
 - ?.?, 2509
 - ?/? , 2509
 - ?=?, 2509
 - ?^?, 2509
 - ?~=?, 2509
 - #?, 2509
 - 0, 2509
 - 1, 2509
 - abs, 2509

- any?, 2509
- characteristic, 2509
- coerce, 2509
- copy, 2509
- count, 2509
- D, 2509
- differentiate, 2509
- dimension, 2509
- directProduct, 2509
- dot, 2509
- elt, 2509
- empty, 2509
- empty?, 2509
- entries, 2509
- entry?, 2509
- eq?, 2509
- eval, 2509
- every?, 2509
- fill, 2509
- first, 2509
- hash, 2509
- index, 2509
- index?, 2509
- indices, 2509
- latex, 2509
- less?, 2509
- lookup, 2509
- map, 2509
- max, 2509
- maxIndex, 2509
- member?, 2509
- members, 2509
- min, 2509
- minIndex, 2509
- more?, 2509
- negative?, 2509
- one?, 2509
- parts, 2509
- positive?, 2509
- qelt, 2509
- qsetelt, 2509
- random, 2509
- recip, 2509
- reducedSystem, 2509
- retract, 2509
- retractIfCan, 2509
- sample, 2509
- setelt, 2509
- sign, 2509
- size, 2509
- size?, 2509
- subtractIfCan, 2509
- sup, 2509
- swap, 2509
- unitVector, 2509
- zero?, 2509
- shift
 - FLOAT, 908
 - INT, 1362
 - MINT, 1557
 - MOEBIUS, 1654
 - NNI, 1739
 - NSDPS, 1703
 - ROMAN, 2326
 - SINT, 2412
- shiftLeft
 - MODMON, 1632
 - MYUP, 1695
 - NSUP, 1729
 - SUP, 2467
 - SUPEXPR, 2481
 - UP, 2828
- shiftRight
 - MODMON, 1632
 - MYUP, 1695
 - NSUP, 1729
 - SUP, 2467
 - SUPEXPR, 2481
 - UP, 2828
- show
 - VIEW2d, 2770
- showAll?
 - STREAM, 2582
- showAllElements
 - STREAM, 2582
- showArrayValues
 - RESULT, 2301
- showAttributes
 - INTFTBL, 1371
- showClipRegion
 - VIEW3D, 2710
- showIntensityFunctions

- ODEIFTBL, [1768](#)
- showRegion
 - VIEW3D, [2710](#)
- showScalarValues
 - RESULT, [2301](#)
- showTheFTable
 - INTFTBL, [1371](#)
- showTheIFTable
 - ODEIFTBL, [1768](#)
- showTheRoutinesTable
 - ROUTINE, [2331](#)
- showTheSymbolTable
 - SYMS, [2696](#)
- showTypeInOutput
 - ANY, [50](#)
- shrinkable
 - FARRAY, [886](#)
 - IFARRAY, [1222](#)
- Si
 - EXPR, [724](#)
- sign
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - DIRPROD, [547](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - EXPEXPAN, [712](#)
 - FLOAT, [908](#)
 - FRAC, [986](#)
 - HDP, [1173](#)
 - HEXADEC, [1143](#)
 - INT, [1362](#)
 - LA, [1520](#)
 - MFLOAT, [1548](#)
 - MINT, [1557](#)
 - ODP, [1816](#)
 - ONECOMP, [1777](#)
 - ORDCOMP, [1810](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - PERM, [1949](#)
 - RADIX, [2206](#)
 - RECLOS, [2237](#)
 - ROIRC, [2310](#)
 - ROMAN, [2326](#)
 - SHDP, [2509](#)
 - SINT, [2412](#)
 - SULS, [2457](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
- SimpleAlgebraicExtension, [2399](#)
- SimpleFortranProgram, [2405](#)
- simplify
 - QALGSET, [2157](#)
- simplifyPower
 - EXPR, [724](#)
- sin
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FCOMP, [975](#)
 - FEXPR, [947](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLEX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXP, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- sin?
 - FCOMP, [975](#)
- SingleInteger, [2411](#)
- SingletonAsOrderedSet, [2418](#)
- singular?
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- singularAtInfinity?

- ALGFF, [28](#)
- RADFF, [2194](#)
- sinh
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FEXPR, [947](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXP, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- sinhIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- sinIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- SINT, [2411](#)
 - ?, [2412](#)
 - ?<?, [2412](#)
 - ?<=?, [2412](#)
 - ?>?, [2412](#)
 - ?>=?, [2412](#)
 - ? $\Gamma E30F$ /? , [2412](#)
 - ?**?, [2412](#)
 - ?*?, [2412](#)
 - ?+?, [2412](#)
 - ?-?, [2412](#)
 - ?/ $\Gamma E30F$?, [2412](#)
 - ?=?, [2412](#)
 - ?^?, [2412](#)
 - ?~=?, [2412](#)
 - ?quo?, [2412](#)
 - ?rem?, [2412](#)
 - ?, [2412](#)
 - 0, [2412](#)
 - 1, [2412](#)
 - abs, [2412](#)
 - addmod, [2412](#)
 - And, [2412](#)
 - associates?, [2412](#)
 - base, [2412](#)
 - binomial, [2412](#)
 - bit?, [2412](#)
 - characteristic, [2412](#)
 - coerce, [2412](#)
 - convert, [2412](#)
 - copy, [2412](#)
 - D, [2412](#)
 - dec, [2412](#)
 - differentiate, [2412](#)
 - divide, [2412](#)
 - euclideanSize, [2412](#)
 - even?, [2412](#)
 - expressIdealMember, [2412](#)
 - exquo, [2412](#)
 - extendedEuclidean, [2412](#)
 - factor, [2412](#)
 - factorial, [2412](#)
 - gcd, [2412](#)
 - gcdPolynomial, [2412](#)
 - hash, [2412](#)
 - inc, [2412](#)
 - init, [2412](#)
 - invmod, [2412](#)
 - latex, [2412](#)
 - lcm, [2412](#)
 - length, [2412](#)
 - mask, [2412](#)
 - max, [2412](#)
 - min, [2412](#)
 - mulmod, [2412](#)
 - multiEuclidean, [2412](#)
 - negative?, [2412](#)
 - nextItem, [2412](#)
 - Not, [2412](#)

- not?, [2412](#)
- odd?, [2412](#)
- OMwrite, [2412](#)
- one?, [2412](#)
- Or, [2412](#)
- patternMatch, [2412](#)
- permutation, [2412](#)
- positive?, [2412](#)
- positiveRemainder, [2412](#)
- powmod, [2412](#)
- prime?, [2412](#)
- principalIdeal, [2412](#)
- random, [2412](#)
- rational, [2412](#)
- rational?, [2412](#)
- rationalIfCan, [2412](#)
- recip, [2412](#)
- reducedSystem, [2412](#)
- retract, [2412](#)
- retractIfCan, [2412](#)
- sample, [2412](#)
- shift, [2412](#)
- sign, [2412](#)
- sizeLess?, [2412](#)
- squareFree, [2412](#)
- squareFreePart, [2412](#)
- submod, [2412](#)
- subtractIfCan, [2412](#)
- symmetricRemainder, [2412](#)
- unit?, [2412](#)
- unitCanonical, [2412](#)
- unitNormal, [2412](#)
- xor, [2412](#)
- zero?, [2412](#)
- size
 - ALGFF, [28](#)
 - BOOLEAN, [322](#)
 - CCLASS, [382](#)
 - CHAR, [373](#)
 - COMPLEX, [419](#)
 - DIRPROD, [547](#)
 - DIV, [575](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - FAGROUP, [1004](#)
 - FAMONOID, [1007](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FGROUP, [1010](#)
 - FMONOID, [1021](#)
 - HDP, [1173](#)
 - IFAMON, [1287](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - LMOPS, [1509](#)
 - MCMPLEX, [1543](#)
 - MODMON, [1632](#)
 - MRING, [1659](#)
 - OCT, [1765](#)
 - ODP, [1816](#)
 - OFMONOID, [1829](#)
 - OVAR, [1836](#)
 - PACOFF, [2136](#)
 - PF, [2105](#)
 - PRODUCT, [2113](#)
 - RADFF, [2194](#)
 - ROIRC, [2310](#)
 - SAE, [2400](#)
 - SET, [2372](#)
 - SETMN, [2377](#)
 - SHDP, [2509](#)
 - ZMOD, [1368](#)
- size?
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - ARRAY2, [2763](#)
 - ASTACK, [65](#)
 - BBTREE, [253](#)
 - BITS, [315](#)
 - BSTREE, [302](#)
 - BTourn, [306](#)
 - BTREE, [310](#)
 - CCLASS, [382](#)
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)

- DEQUEUE, [512](#)
- DFMAT, [599](#)
- DFVEC, [605](#)
- DHMATRIX, [491](#)
- DIRPROD, [547](#)
- DLIST, [460](#)
- DPMM, [553](#)
- DPMO, [557](#)
- DSTREE, [535](#)
- EQTBL, [699](#)
- FARRAY, [886](#)
- GPOLSET, [1074](#)
- GSTBL, [1079](#)
- GTSET, [1083](#)
- HASHTBL, [1120](#)
- HDP, [1173](#)
- HEAP, [1134](#)
- IARRAY1, [1244](#)
- IARRAY2, [1256](#)
- IBITS, [1199](#)
- IFARRAY, [1222](#)
- IIARRAY2, [1290](#)
- ILIST, [1231](#)
- IMATRIX, [1239](#)
- INTABL, [1336](#)
- ISTRING, [1249](#)
- IVECTOR, [1260](#)
- KAFIL, [1414](#)
- LIB, [1429](#)
- LIST, [1504](#)
- LMDICT, [1515](#)
- LSQM, [1456](#)
- M3D, [2703](#)
- MATRIX, [1623](#)
- MSET, [1670](#)
- NSDPS, [1703](#)
- ODP, [1816](#)
- PENDTREE, [1945](#)
- POINT, [2060](#)
- PRIMARR, [2109](#)
- QUEUE, [2184](#)
- REGSET, [2286](#)
- RESULT, [2301](#)
- RGCHAIN, [2255](#)
- RMATRIX, [2246](#)
- ROUTINE, [2331](#)
- SET, [2372](#)
- SHDP, [2509](#)
- SPLTREE, [2518](#)
- SQMATRIX, [2548](#)
- SREGSET, [2535](#)
- STACK, [2563](#)
- STBL, [2450](#)
- STREAM, [2582](#)
- STRING, [2607](#)
- STRTBL, [2610](#)
- TABLE, [2664](#)
- TREE, [2741](#)
- U16MAT, [2906](#)
- U16VEC, [2924](#)
- U32MAT, [2911](#)
- U32VEC, [2931](#)
- U8MAT, [2902](#)
- U8VEC, [2917](#)
- VECTOR, [2942](#)
- WUTSET, [2959](#)
- sizeLess?
 - ALGFF, [28](#)
 - AN, [35](#)
 - BINARY, [292](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - EMR, [702](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FLOAT, [908](#)
 - FRAC, [986](#)
 - GSERIES, [1090](#)

- HACKPI, [1978](#)
- HEXADEC, [1143](#)
- IAN, [1276](#)
- IFF, [1284](#)
- INT, [1362](#)
- IPADIC, [1294](#)
- IPF, [1303](#)
- LAUPOL, [1422](#)
- MCMPLX, [1543](#)
- MFLOAT, [1548](#)
- MINT, [1557](#)
- MODFIELD, [1638](#)
- MODMON, [1632](#)
- MYEXPR, [1688](#)
- MYUP, [1695](#)
- NSDPS, [1703](#)
- NSUP, [1729](#)
- ODR, [1858](#)
- PACOFF, [2136](#)
- PACRAT, [2146](#)
- PADIC, [1881](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PF, [2105](#)
- PFR, [1914](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- ROMAN, [2326](#)
- SAE, [2400](#)
- SINT, [2412](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- sizeMultiplication
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
- slash
 - OUTFORM, [1868](#)
- SMP, [2422](#)
 - , [2423](#)
 - ?<?, [2423](#)
 - ?<=?, [2423](#)
 - ?>?, [2423](#)
 - ?>=?, [2423](#)
 - ?**?, [2423](#)
 - ?*?, [2423](#)
 - ?+?, [2423](#)
 - ?-?, [2423](#)
 - ?/?, [2423](#)
 - ?=?, [2423](#)
 - ?^?, [2423](#)
 - ?~=?, [2423](#)
 - 0, [2423](#)
 - 1, [2423](#)
 - associates?, [2423](#)
 - binomThmExpt, [2423](#)
 - characteristic, [2423](#)
 - charthRoot, [2423](#)
 - coefficient, [2423](#)
 - coefficients, [2423](#)
 - coerce, [2423](#)
 - conditionP, [2423](#)
 - content, [2423](#)
 - convert, [2423](#)
 - D, [2423](#)
 - degree, [2423](#)
 - differentiate, [2423](#)
 - discriminant, [2423](#)
 - eval, [2423](#)
 - exquo, [2423](#)
 - factor, [2423](#)
 - factorPolynomial, [2423](#)
 - factorSquareFreePolynomial, [2423](#)
 - gcd, [2423](#)
 - gcdPolynomial, [2423](#)
 - ground, [2423](#)
 - ground?, [2423](#)
 - hash, [2423](#)
 - isExpt, [2423](#)
 - isPlus, [2423](#)
 - isTimes, [2423](#)
 - latex, [2423](#)
 - lcm, [2423](#)
 - leadingCoefficient, [2423](#)

- leadingMonomial, [2423](#)
- mainVariable, [2423](#)
- map, [2423](#)
- mapExponents, [2423](#)
- max, [2423](#)
- min, [2423](#)
- minimumDegree, [2423](#)
- monicDivide, [2423](#)
- monomial, [2423](#)
- monomial?, [2423](#)
- monomials, [2423](#)
- multivariate, [2423](#)
- numberOfMonomials, [2423](#)
- one?, [2423](#)
- patternMatch, [2423](#)
- pomopo, [2423](#)
- prime?, [2423](#)
- primitiveMonomials, [2423](#)
- primitivePart, [2423](#)
- recip, [2423](#)
- reducedSystem, [2423](#)
- reductum, [2423](#)
- resultant, [2423](#)
- retract, [2423](#)
- retractIfCan, [2423](#)
- sample, [2423](#)
- solveLinearPolynomialEquation, [2423](#)
- squareFree, [2423](#)
- squareFreePart, [2423](#)
- squareFreePolynomial, [2423](#)
- subtractIfCan, [2423](#)
- totalDegree, [2423](#)
- unit?, [2423](#)
- unitCanonical, [2423](#)
- unitNormal, [2423](#)
- univariate, [2423](#)
- variables, [2423](#)
- zero?, [2423](#)
- SMTS, [2440](#)
- ?, [2441](#)
- ?**, [2441](#)
- ?*, [2441](#)
- ?+?, [2441](#)
- ?-?, [2441](#)
- ?=?, [2441](#)
- ?^?, [2441](#)
- ?~=?, [2441](#)
- 0, [2441](#)
- 1, [2441](#)
- acos, [2441](#)
- acosh, [2441](#)
- acot, [2441](#)
- acoth, [2441](#)
- acsc, [2441](#)
- acsch, [2441](#)
- asec, [2441](#)
- asech, [2441](#)
- asin, [2441](#)
- asinh, [2441](#)
- associates?, [2441](#)
- atan, [2441](#)
- atanh, [2441](#)
- characteristic, [2441](#)
- charthRoot, [2441](#)
- coefficient, [2441](#)
- coerce, [2441](#)
- complete, [2441](#)
- cos, [2441](#)
- cosh, [2441](#)
- cot, [2441](#)
- coth, [2441](#)
- csc, [2441](#)
- csch, [2441](#)
- csubst, [2441](#)
- D, [2441](#)
- degree, [2441](#)
- differentiate, [2441](#)
- eval, [2441](#)
- exp, [2441](#)
- exquo, [2441](#)
- extend, [2441](#)
- fintegrate, [2441](#)
- hash, [2441](#)
- integrate, [2441](#)
- latex, [2441](#)
- leadingCoefficient, [2441](#)
- leadingMonomial, [2441](#)
- log, [2441](#)
- map, [2441](#)
- monomial, [2441](#)
- monomial?, [2441](#)
- nthRoot, [2441](#)

- one?, [2441](#)
- order, [2441](#)
- pi, [2441](#)
- pole?, [2441](#)
- polynomial, [2441](#)
- recip, [2441](#)
- reductum, [2441](#)
- sample, [2441](#)
- sec, [2441](#)
- sech, [2441](#)
- sin, [2441](#)
- sinh, [2441](#)
- sqrt, [2441](#)
- subtractIfCan, [2441](#)
- tan, [2441](#)
- tanh, [2441](#)
- unit?, [2441](#)
- unitCanonical, [2441](#)
- unitNormal, [2441](#)
- variables, [2441](#)
- zero?, [2441](#)
- solid
 - COMPPROP, [2624](#)
- solid?
 - COMPPROP, [2624](#)
- solveLinearPolynomialEquation
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - DECIMAL, [465](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPEXPAN, [712](#)
 - FRAC, [986](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - HEXADEC, [1143](#)
 - MCMPPLX, [1543](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - POLY, [2078](#)
 - RADIX, [2206](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SULS, [2457](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UP, [2828](#)
- someBasis
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- Somos
 - GOPT, [1104](#)
 - GOPT0, [1110](#)
- sort
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - BITS, [315](#)
 - CDFVEC, [432](#)
 - DFVEC, [605](#)
 - DLIST, [460](#)
 - FARRAY, [886](#)
 - GPOLSET, [1074](#)
 - GTSET, [1083](#)
 - IARRAY1, [1244](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - ILIST, [1231](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - LIST, [1504](#)
 - PERM, [1949](#)
 - POINT, [2060](#)
 - PRIMARR, [2109](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - STRING, [2607](#)
 - U16VEC, [2924](#)
 - U32VEC, [2931](#)
 - U8VEC, [2917](#)

- VECTOR, [2942](#)
- WUTSET, [2959](#)
- sorted?
 - ALIST, [235](#)
 - ARRAY1, [1774](#)
 - BITS, [315](#)
 - CDFVEC, [432](#)
 - DFVEC, [605](#)
 - DLIST, [460](#)
 - FARRAY, [886](#)
 - IARRAY1, [1244](#)
 - IBITS, [1199](#)
 - IFARRAY, [1222](#)
 - ILIST, [1231](#)
 - ISTRING, [1249](#)
 - IVECTOR, [1260](#)
 - LIST, [1504](#)
 - POINT, [2060](#)
 - PRIMARR, [2109](#)
 - STRING, [2607](#)
 - U16VEC, [2924](#)
 - U32VEC, [2931](#)
 - U8VEC, [2917](#)
 - VECTOR, [2942](#)
- space
 - CHAR, [373](#)
 - DROPT, [608](#)
- SPACE3, [2731](#)
 - ?=?, [2731](#)
 - ?~=?, [2731](#)
 - check, [2731](#)
 - closedCurve, [2731](#)
 - closedCurve?, [2731](#)
 - coerce, [2731](#)
 - components, [2731](#)
 - composite, [2731](#)
 - composites, [2731](#)
 - copy, [2731](#)
 - create3Space, [2731](#)
 - curve, [2731](#)
 - curve?, [2731](#)
 - enterPointData, [2731](#)
 - hash, [2731](#)
 - latex, [2731](#)
 - lllip, [2731](#)
 - lllp, [2731](#)
 - llprop, [2731](#)
 - lp, [2731](#)
 - lprop, [2731](#)
 - merge, [2731](#)
 - mesh, [2731](#)
 - mesh?, [2731](#)
 - modifyPointData, [2731](#)
 - numberOfComponents, [2731](#)
 - objects, [2731](#)
 - point, [2731](#)
 - point?, [2731](#)
 - polygon, [2731](#)
 - polygon?, [2731](#)
 - subspace, [2731](#)
- SparseMultivariatePolynomial, [2422](#)
- SparseMultivariateTaylorSeries, [2440](#)
- SparseTable, [2450](#)
- SparseUnivariateLaurentSeries, [2456](#)
- SparseUnivariatePolynomial, [2466](#)
- SparseUnivariatePolynomialExpressions, [2480](#)
- SparseUnivariatePuisseuxSeries, [2487](#)
- SparseUnivariateSkewPolynomial, [2492](#)
- SparseUnivariateTaylorSeries, [2496](#)
- split
 - BSTREE, [302](#)
 - DIV, [575](#)
 - ISTRING, [1249](#)
 - STRING, [2607](#)
- SplitHomogeneousDirectProduct, [2508](#)
- SplittingNode, [2512](#)
- SplittingTree, [2518](#)
- SPLNODE, [2512](#)
 - ?=?, [2512](#)
 - ?~=?, [2512](#)
 - coerce, [2512](#)
 - condition, [2512](#)
 - construct, [2512](#)
 - copy, [2512](#)
 - empty, [2512](#)
 - empty?, [2512](#)
 - hash, [2512](#)
 - inflex?, [2512](#)
 - latex, [2512](#)
 - setCondition, [2512](#)
 - setEmpty, [2512](#)
 - setStatus, [2512](#)

- setValue, [2512](#)
 - status, [2512](#)
 - subNode?, [2512](#)
 - value, [2512](#)
- SPLTREE, [2518](#)
 - ?.value, [2518](#)
 - ?=?, [2518](#)
 - ?~=?, [2518](#)
 - #?, [2518](#)
 - any?, [2518](#)
 - child?, [2518](#)
 - children, [2518](#)
 - coerce, [2518](#)
 - conditions, [2518](#)
 - construct, [2518](#)
 - copy, [2518](#)
 - count, [2518](#)
 - cyclic?, [2518](#)
 - distance, [2518](#)
 - empty, [2518](#)
 - empty?, [2518](#)
 - eq?, [2518](#)
 - eval, [2518](#)
 - every?, [2518](#)
 - extractSplittingLeaf, [2518](#)
 - hash, [2518](#)
 - latex, [2518](#)
 - leaf?, [2518](#)
 - leaves, [2518](#)
 - less?, [2518](#)
 - map, [2518](#)
 - member?, [2518](#)
 - members, [2518](#)
 - more?, [2518](#)
 - node?, [2518](#)
 - nodeOf?, [2518](#)
 - nodes, [2518](#)
 - parts, [2518](#)
 - remove, [2518](#)
 - result, [2518](#)
 - sample, [2518](#)
 - setchildren, [2518](#)
 - setelt, [2518](#)
 - setvalue, [2518](#)
 - size?, [2518](#)
 - splitNodeOf, [2518](#)
 - subNodeOf?, [2518](#)
 - updateStatus, [2518](#)
 - value, [2518](#)
- sqrFactor
 - FR, [786](#)
- SQMATRIX, [2547](#)
 - , [2548](#)
 - ?**?, [2548](#)
 - ?*?, [2548](#)
 - ?+?, [2548](#)
 - ?-?, [2548](#)
 - ?/? , [2548](#)
 - ?=?, [2548](#)
 - ?^?, [2548](#)
 - ?~=?, [2548](#)
 - #?, [2548](#)
 - 0, [2548](#)
 - 1, [2548](#)
 - antisymmetric?, [2548](#)
 - any?, [2548](#)
 - characteristic, [2548](#)
 - coerce, [2548](#)
 - column, [2548](#)
 - convert, [2548](#)
 - copy, [2548](#)
 - count, [2548](#)
 - D, [2548](#)
 - determinant, [2548](#)
 - diagonal, [2548](#)
 - diagonal?, [2548](#)
 - diagonalMatrix, [2548](#)
 - diagonalProduct, [2548](#)
 - differentiate, [2548](#)
 - elt, [2548](#)
 - empty, [2548](#)
 - empty?, [2548](#)
 - eq?, [2548](#)
 - eval, [2548](#)
 - every?, [2548](#)
 - exquo, [2548](#)
 - hash, [2548](#)
 - inverse, [2548](#)
 - latex, [2548](#)
 - less?, [2548](#)
 - listOfLists, [2548](#)
 - map, [2548](#)

- matrix, [2548](#)
- maxColIndex, [2548](#)
- maxRowIndex, [2548](#)
- member?, [2548](#)
- members, [2548](#)
- minColIndex, [2548](#)
- minordet, [2548](#)
- minRowIndex, [2548](#)
- more?, [2548](#)
- ncols, [2548](#)
- nrows, [2548](#)
- nullity, [2548](#)
- nullSpace, [2548](#)
- one?, [2548](#)
- parts, [2548](#)
- qelt, [2548](#)
- rank, [2548](#)
- recip, [2548](#)
- reducedSystem, [2548](#)
- retract, [2548](#)
- retractIfCan, [2548](#)
- row, [2548](#)
- rowEchelon, [2548](#)
- sample, [2548](#)
- scalarMatrix, [2548](#)
- size?, [2548](#)
- square?, [2548](#)
- squareMatrix, [2548](#)
- subtractIfCan, [2548](#)
- symmetric?, [2548](#)
- trace, [2548](#)
- transpose, [2548](#)
- zero?, [2548](#)
- sqrt
 - AN, [35](#)
 - BPADIC, [258](#)
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FEXPR, [947](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - IAN, [1276](#)
 - INTRVL, [1384](#)
 - IPADIC, [1294](#)
 - MCMPLX, [1543](#)
 - MFLOAT, [1548](#)
 - PADIC, [1881](#)
 - RECLOS, [2237](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
 - square?
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IMATRIX, [1239](#)
 - LSQM, [1456](#)
 - MATRIX, [1623](#)
 - RMATRIX, [2246](#)
 - SQMATRIX, [2548](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
 - squareFree
 - ALGFF, [28](#)
 - AN, [35](#)
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)

- FFNB, 860
- FFNBP, 872
- FFNBX, 866
- FFP, 851
- FFX, 846
- FLOAT, 908
- FR, 786
- FRAC, 986
- GDMP, 1052
- GSERIES, 1090
- HACKPI, 1978
- HDMP, 1180
- HEXADEC, 1143
- IAN, 1276
- IFF, 1284
- INT, 1362
- IPF, 1303
- MCMPLEX, 1543
- MFLOAT, 1548
- MINT, 1557
- MODFIELD, 1638
- MODMON, 1632
- MPOLY, 1682
- MYEXPR, 1688
- MYUP, 1695
- NSDPS, 1703
- NSMP, 1714
- NSUP, 1729
- ODPOL, 1852
- ODR, 1858
- PACOFF, 2136
- PACRAT, 2146
- PADICRAT, 1886
- PADICRC, 1891
- PF, 2105
- PFR, 1914
- POLY, 2078
- RADFF, 2194
- RADIX, 2206
- RECLOS, 2237
- ROMAN, 2326
- SAE, 2400
- SDPOL, 2386
- SINT, 2412
- SMP, 2423
- SULS, 2457
- SUP, 2467
- SUPEXPR, 2481
- SUPXS, 2487
- ULS, 2796
- ULSCONS, 2804
- UP, 2828
- UPXS, 2834
- UPXSCONS, 2842
- squareFreePart
 - ALGFF, 28
 - AN, 35
 - BINARY, 292
 - BPADICRT, 263
 - COMPLEX, 419
 - CONTFRAC, 445
 - DECIMAL, 465
 - DFLOAT, 587
 - DMP, 572
 - DSMP, 542
 - EXPEXPAN, 712
 - EXPR, 724
 - EXPUPXS, 740
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866
 - FFP, 851
 - FFX, 846
 - FLOAT, 908
 - FR, 786
 - FRAC, 986
 - GDMP, 1052
 - GSERIES, 1090
 - HACKPI, 1978
 - HDMP, 1180
 - HEXADEC, 1143
 - IAN, 1276
 - IFF, 1284
 - INT, 1362
 - IPF, 1303
 - MCMPLEX, 1543
 - MFLOAT, 1548
 - MINT, 1557

- MODFIELD, [1638](#)
- MODMON, [1632](#)
- MPOLY, [1682](#)
- MYEXPR, [1688](#)
- MYUP, [1695](#)
- NSDPS, [1703](#)
- NSMP, [1714](#)
- NSUP, [1729](#)
- ODPOL, [1852](#)
- ODR, [1858](#)
- PACOFF, [2136](#)
- PACRAT, [2146](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PF, [2105](#)
- PFR, [1914](#)
- POLY, [2078](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- REGSET, [2286](#)
- RGCHAIN, [2255](#)
- ROMAN, [2326](#)
- SAE, [2400](#)
- SDPOL, [2386](#)
- SINT, [2412](#)
- SMP, [2423](#)
- SREGSET, [2535](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUEXPR, [2481](#)
- SUPXS, [2487](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- squareFreePolynomial
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - DECIMAL, [465](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - FRAC, [986](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - HEXADEC, [1143](#)
 - MCMPLX, [1543](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - POLY, [2078](#)
 - RADIX, [2206](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SULS, [2457](#)
 - SUP, [2467](#)
 - SUEXPR, [2481](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UP, [2828](#)
- SquareFreeRegularTriangularSet, [2534](#)
- SquareMatrix, [2547](#)
- squareMatrix
 - SQMATRIX, [2548](#)
- squareTop
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IMATRIX, [1239](#)
 - MATRIX, [1623](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- SREGSET, [2534](#)
- ?=?, [2535](#)
- ?~=?, [2535](#)
- #?, [2535](#)
- algebraic?, [2535](#)
- algebraicCoefficients?, [2535](#)
- algebraicVariables, [2535](#)
- any?, [2535](#)
- augment, [2535](#)
- autoReduced?, [2535](#)

- basicSet, [2535](#)
- coerce, [2535](#)
- coHeight, [2535](#)
- collect, [2535](#)
- collectQuasiMonic, [2535](#)
- collectUnder, [2535](#)
- collectUpper, [2535](#)
- construct, [2535](#)
- convert, [2535](#)
- copy, [2535](#)
- count, [2535](#)
- degree, [2535](#)
- empty, [2535](#)
- empty?, [2535](#)
- eq?, [2535](#)
- eval, [2535](#)
- every?, [2535](#)
- extend, [2535](#)
- extendIfCan, [2535](#)
- find, [2535](#)
- first, [2535](#)
- hash, [2535](#)
- headReduce, [2535](#)
- headReduced?, [2535](#)
- headRemainder, [2535](#)
- inFRittWu?, [2535](#)
- initiallyReduce, [2535](#)
- initiallyReduced?, [2535](#)
- initials, [2535](#)
- internalAugment, [2535](#)
- internalZeroSetSplit, [2535](#)
- intersect, [2535](#)
- invertible?, [2535](#)
- invertibleElseSplit?, [2535](#)
- invertibleSet, [2535](#)
- last, [2535](#)
- lastSubResultant, [2535](#)
- lastSubResultantElseSplit, [2535](#)
- latex, [2535](#)
- less?, [2535](#)
- mainVariable?, [2535](#)
- mainVariables, [2535](#)
- map, [2535](#)
- member?, [2535](#)
- members, [2535](#)
- more?, [2535](#)
- mvar, [2535](#)
- normalized?, [2535](#)
- parts, [2535](#)
- preprocess, [2535](#)
- purelyAlgebraic?, [2535](#)
- purelyAlgebraicLeadingMonomial?, [2535](#)
- purelyTranscendental?, [2535](#)
- quasiComponent, [2535](#)
- reduce, [2535](#)
- reduceByQuasiMonic, [2535](#)
- reduced?, [2535](#)
- remainder, [2535](#)
- remove, [2535](#)
- removeDuplicates, [2535](#)
- removeZero, [2535](#)
- rest, [2535](#)
- retract, [2535](#)
- retractIfCan, [2535](#)
- rewriteIdealWithHeadRemainder, [2535](#)
- rewriteIdealWithRemainder, [2535](#)
- rewriteSetWithReduction, [2535](#)
- roughBase?, [2535](#)
- roughEqualIdeals?, [2535](#)
- roughSubIdeal?, [2535](#)
- roughUnitIdeal?, [2535](#)
- sample, [2535](#)
- select, [2535](#)
- size?, [2535](#)
- sort, [2535](#)
- squareFreePart, [2535](#)
- stronglyReduce, [2535](#)
- stronglyReduced?, [2535](#)
- triangular?, [2535](#)
- trivialIdeal?, [2535](#)
- variables, [2535](#)
- zeroSetSplit, [2535](#)
- zeroSetSplitIntoTriangularSystems, [2535](#)
- STACK, [2563](#)
- =? , [2563](#)
- ?~=? , [2563](#)
- #? , [2563](#)
- any? , [2563](#)
- bag, [2563](#)
- coerce, [2563](#)
- copy, [2563](#)
- count, [2563](#)

- depth, 2563
- empty, 2563
- empty?, 2563
- eq?, 2563
- eval, 2563
- every?, 2563
- extract, 2563
- hash, 2563
- insert, 2563
- inspect, 2563
- latex, 2563
- less?, 2563
- map, 2563
- member?, 2563
- members, 2563
- more?, 2563
- parts, 2563
- pop, 2563
- push, 2563
- sample, 2563
- size?, 2563
- stack, 2563
- top, 2563
- Stack, 2563
- stack
 - STACK, 2563
- status
 - QALGSET, 2157
 - SPLNODE, 2512
- statusIto
 - SD, 2572
- STBL, 2450
 - ?.?, 2450
 - ?=?, 2450
 - ?~=?, 2450
 - #?, 2450
 - any?, 2450
 - bag, 2450
 - coerce, 2450
 - construct, 2450
 - convert, 2450
 - copy, 2450
 - count, 2450
 - dictionary, 2450
 - elt, 2450
 - empty, 2450
 - empty?, 2450
 - entries, 2450
 - entry?, 2450
 - eq?, 2450
 - eval, 2450
 - every?, 2450
 - extract, 2450
 - fill, 2450
 - find, 2450
 - first, 2450
 - hash, 2450
 - index?, 2450
 - indices, 2450
 - insert, 2450
 - inspect, 2450
 - key?, 2450
 - keys, 2450
 - latex, 2450
 - less?, 2450
 - map, 2450
 - maxIndex, 2450
 - member?, 2450
 - members, 2450
 - minIndex, 2450
 - more?, 2450
 - parts, 2450
 - qelt, 2450
 - qsetelt, 2450
 - reduce, 2450
 - remove, 2450
 - removeDuplicates, 2450
 - sample, 2450
 - search, 2450
 - select, 2450
 - setelt, 2450
 - size?, 2450
 - swap, 2450
 - table, 2450
- StochasticDifferential, 2572
- stop
 - FC, 931
- STREAM, 2581
 - ?.?, 2582
 - ?..first, 2582
 - ?..last, 2582
 - ?..rest, 2582

- [?.value, 2582](#)
- [?=?, 2582](#)
- [?~=?, 2582](#)
- [#?, 2582](#)
- [any?, 2582](#)
- [child?, 2582](#)
- [children, 2582](#)
- [coerce, 2582](#)
- [complete, 2582](#)
- [concat, 2582](#)
- [cons, 2582](#)
- [construct, 2582](#)
- [convert, 2582](#)
- [copy, 2582](#)
- [count, 2582](#)
- [cycleEntry, 2582](#)
- [cycleLength, 2582](#)
- [cycleSplit, 2582](#)
- [cycleTail, 2582](#)
- [cyclic?, 2582](#)
- [delay, 2582](#)
- [delete, 2582](#)
- [distance, 2582](#)
- [elt, 2582](#)
- [empty, 2582](#)
- [empty?, 2582](#)
- [entries, 2582](#)
- [entry?, 2582](#)
- [eq?, 2582](#)
- [eval, 2582](#)
- [every?, 2582](#)
- [explicitEntries?, 2582](#)
- [explicitlyEmpty?, 2582](#)
- [explicitlyFinite?, 2582](#)
- [extend, 2582](#)
- [fill, 2582](#)
- [filterUntil, 2582](#)
- [filterWhile, 2582](#)
- [find, 2582](#)
- [findCycle, 2582](#)
- [first, 2582](#)
- [frst, 2582](#)
- [generate, 2582](#)
- [hash, 2582](#)
- [index?, 2582](#)
- [indices, 2582](#)
- [insert, 2582](#)
- [last, 2582](#)
- [latex, 2582](#)
- [lazy?, 2582](#)
- [lazyEvaluate, 2582](#)
- [leaf?, 2582](#)
- [leaves, 2582](#)
- [less?, 2582](#)
- [map, 2582](#)
- [maxIndex, 2582](#)
- [member?, 2582](#)
- [members, 2582](#)
- [minIndex, 2582](#)
- [more?, 2582](#)
- [new, 2582](#)
- [node?, 2582](#)
- [nodes, 2582](#)
- [numberOfComputedEntries, 2582](#)
- [output, 2582](#)
- [parts, 2582](#)
- [possiblyInfinite?, 2582](#)
- [qelt, 2582](#)
- [qsetelt, 2582](#)
- [reduce, 2582](#)
- [remove, 2582](#)
- [removeDuplicates, 2582](#)
- [repeating, 2582](#)
- [repeating?, 2582](#)
- [rest, 2582](#)
- [rst, 2582](#)
- [sample, 2582](#)
- [second, 2582](#)
- [select, 2582](#)
- [setchildren, 2582](#)
- [setelt, 2582](#)
- [setfirst, 2582](#)
- [setlast, 2582](#)
- [setrest, 2582](#)
- [setvalue, 2582](#)
- [showAll?, 2582](#)
- [showAllElements, 2582](#)
- [size?, 2582](#)
- [split, 2582](#)
- [swap, 2582](#)
- [tail, 2582](#)
- [third, 2582](#)

- value, 2582
- Stream, 2581
- STRING, 2606
 - ?<?, 2607
 - ?<=?, 2607
 - ?>?, 2607
 - ?>=?, 2607
 - ?..?, 2607
 - ?=?, 2607
 - ?~=?, 2607
 - #?, 2607
 - any?, 2607
 - coerce, 2607
 - concat, 2607
 - construct, 2607
 - convert, 2607
 - copy, 2607
 - copyInto, 2607
 - count, 2607
 - delete, 2607
 - elt, 2607
 - empty, 2607
 - empty?, 2607
 - entries, 2607
 - entry?, 2607
 - eq?, 2607
 - eval, 2607
 - every?, 2607
 - fill, 2607
 - find, 2607
 - first, 2607
 - hash, 2607
 - index?, 2607
 - indices, 2607
 - insert, 2607
 - latex, 2607
 - leftTrim, 2607
 - less?, 2607
 - lowerCase, 2607
 - map, 2607
 - match, 2607
 - match?, 2607
 - max, 2607
 - maxIndex, 2607
 - member?, 2607
 - members, 2607
 - merge, 2607
 - min, 2607
 - minIndex, 2607
 - more?, 2607
 - new, 2607
 - OMwrite, 2607
 - parts, 2607
 - position, 2607
 - prefix?, 2607
 - qelt, 2607
 - qsetelt, 2607
 - reduce, 2607
 - removeDuplicates, 2607
 - replace, 2607
 - reverse, 2607
 - rightTrim, 2607
 - sample, 2607
 - select, 2607
 - setelt, 2607
 - size?, 2607
 - sort, 2607
 - sorted?, 2607
 - split, 2607
 - string, 2607
 - substring?, 2607
 - suffix?, 2607
 - swap, 2607
 - trim, 2607
 - upperCase, 2607
- String, 2606
- string
 - INFORM, 1343
 - OUTFORM, 1868
 - SEX, 2392
 - SEXOF, 2394
 - STRING, 2607
 - SYMBOL, 2640
- string?
 - INFORM, 1343
 - SEX, 2392
 - SEXOF, 2394
- StringTable, 2610
- strongGenerators
 - PERMGRP, 1960
- stronglyReduce
 - GTSET, 1083

- REGSET, [2286](#)
- RGCHAIN, [2255](#)
- SREGSET, [2535](#)
- WUTSET, [2959](#)
- stronglyReduced?
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- STRTBL, [2610](#)
- ?.?, [2610](#)
- ?=?, [2610](#)
- ?~=?, [2610](#)
- #?, [2610](#)
- any?, [2610](#)
- bag, [2610](#)
- coerce, [2610](#)
- construct, [2610](#)
- convert, [2610](#)
- copy, [2610](#)
- count, [2610](#)
- dictionary, [2610](#)
- elt, [2610](#)
- empty, [2610](#)
- empty?, [2610](#)
- entries, [2610](#)
- entry?, [2610](#)
- eq?, [2610](#)
- eval, [2610](#)
- every?, [2610](#)
- extract, [2610](#)
- fill, [2610](#)
- find, [2610](#)
- first, [2610](#)
- hash, [2610](#)
- index?, [2610](#)
- indices, [2610](#)
- insert, [2610](#)
- inspect, [2610](#)
- key?, [2610](#)
- keys, [2610](#)
- latex, [2610](#)
- less?, [2610](#)
- map, [2610](#)
- maxIndex, [2610](#)
- member?, [2610](#)
- members, [2610](#)
- minIndex, [2610](#)
- more?, [2610](#)
- parts, [2610](#)
- qelt, [2610](#)
- qsetelt, [2610](#)
- reduce, [2610](#)
- remove, [2610](#)
- removeDuplicates, [2610](#)
- sample, [2610](#)
- search, [2610](#)
- select, [2610](#)
- setelt, [2610](#)
- size?, [2610](#)
- swap, [2610](#)
- table, [2610](#)
- structuralConstants
 - ALGSC, [15](#)
 - GCNAALG, [1065](#)
 - JORDAN, [223](#)
 - LIE, [229](#)
 - LSQM, [1456](#)
- style
 - DROPT, [608](#)
- sub
 - OUTFORM, [1868](#)
- subHeight
 - OUTFORM, [1868](#)
- subMatrix
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IMATRIX, [1239](#)
 - MATRIX, [1623](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- submod
 - INT, [1362](#)
 - MINT, [1557](#)
 - ROMAN, [2326](#)
 - SINT, [2412](#)
- subMultV
 - IC, [1191](#)
 - INFCLSPS, [1271](#)

- INFCLSPT, 1265
- subNode?
 - SPLNODE, 2512
- subNodeOf?
 - SPLTREE, 2518
- subResultantChain
 - NSMP, 1714
- subResultantGcd
 - MODMON, 1632
 - MYUP, 1695
 - NSMP, 1714
 - NSUP, 1729
 - SUP, 2467
 - SUPEXPR, 2481
 - UP, 2828
- subResultantsChain
 - NSUP, 1729
- subscript
 - SYMBOL, 2640
- subset?
 - CCLASS, 382
 - MSET, 1670
 - SET, 2372
- SUBSPACE, 2614
 - ?=?, 2614
 - ? =?, 2614
 - addPoint, 2614
 - addPoint2, 2614
 - addPointLast, 2614
 - birth, 2614
 - child, 2614
 - children, 2614
 - closeComponent, 2614
 - coerce, 2614
 - deepCopy, 2614
 - defineProperty, 2614
 - extractClosed, 2614
 - extractIndex, 2614
 - extractPoint, 2614
 - extractProperty, 2614
 - hash, 2614
 - internal?, 2614
 - latex, 2614
 - leaf?, 2614
 - level, 2614
 - merge, 2614
 - modifyPoint, 2614
 - new, 2614
 - numberOfChildren, 2614
 - parent, 2614
 - pointData, 2614
 - root?, 2614
 - separate, 2614
 - shallowCopy, 2614
 - subspace, 2614
 - traverse, 2614
- SubSpace, 2614
- subspace
 - SPACE3, 2731
 - SUBSPACE, 2614
 - VIEW3D, 2710
- SubSpaceComponentProperty, 2624
- subst
 - AN, 35
 - EQ, 691
 - EXPR, 724
 - FEXPR, 947
 - IAN, 1276
 - MYEXPR, 1688
- substitute
 - LMDICT, 1515
- substring?
 - ISTRING, 1249
 - STRING, 2607
- subtractIfCan
 - ALGFF, 28
 - ALGSC, 15
 - AN, 35
 - ANTISYM, 40
 - BINARY, 292
 - BPADIC, 258
 - BPADICRT, 263
 - CLIF, 402
 - COMPLEX, 419
 - CONTFRAC, 445
 - DECIMAL, 465
 - DERHAM, 530
 - DFLOAT, 587
 - DIRPROD, 547
 - DIRRING, 564
 - DIV, 575
 - DMP, 572

- DPMM, 553
DPMO, 557
DSMP, 542
EMR, 702
EQ, 691
EXPEXPAN, 712
EXPR, 724
EXPUPXS, 740
FAGROUP, 1004
FAMONOID, 1007
FDIV, 813
FEXPR, 947
FF, 820
FFCG, 825
FFCGP, 835
FFCGX, 830
FFNB, 860
FFNBP, 872
FFNBX, 866
FFP, 851
FFX, 846
FLOAT, 908
FM, 1013
FM1, 1017
FNLA, 1027
FR, 786
FRAC, 986
FSERIES, 978
GCNAALG, 1065
GDMP, 1052
GMODPOL, 1059
GSERIES, 1090
HACKPI, 1978
HDMP, 1180
HDP, 1173
HELLFDIV, 1183
HEXADEC, 1143
IAN, 1276
IDPAG, 1202
IDPOAMS, 1215
IFAMON, 1287
IFF, 1284
INDE, 1218
INT, 1362
INTRVL, 1384
IPADIC, 1294
IPF, 1303
IR, 1375
ISUPS, 1311
ITAYLOR, 1338
JORDAN, 223
LA, 1520
LAUPOL, 1422
LIE, 229
LO, 1523
LODO, 1469
LODO1, 1479
LODO2, 1491
LPOLY, 1447
LSQM, 1456
MCMPLEX, 1543
MFLOAT, 1548
MINT, 1557
MODFIELD, 1638
MODMON, 1632
MODOP, 1647, 1804
MODRING, 1641
MPOLY, 1682
MRING, 1659
MYEXPR, 1688
MYUP, 1695
NNI, 1739
NSDPS, 1703
NSMP, 1714
NSUP, 1729
OCT, 1765
ODP, 1816
ODPOL, 1852
ODR, 1858
OMLO, 1806
ONECOMP, 1777
ORDCOMP, 1810
ORESUP, 2492
OREUP, 2872
OWP, 1861
PACOFF, 2136
PACRAT, 2146
PADIC, 1881
PADICRAT, 1886
PADICRC, 1891
PF, 2105
PFR, 1914

- POLY, [2078](#)
- PR, [2092](#)
- PRODUCT, [2113](#)
- PRITION, [1923](#)
- QFORM, [2154](#)
- QUAT, [2167](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- RESRING, [2296](#)
- RMATRIX, [2246](#)
- ROMAN, [2326](#)
- SAE, [2400](#)
- SD, [2572](#)
- SDPOL, [2386](#)
- SHDP, [2509](#)
- SINT, [2412](#)
- SMP, [2423](#)
- SMTS, [2441](#)
- SQMATRIX, [2548](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXPR, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- SYMPOLY, [2654](#)
- TS, [2671](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- WP, [2949](#)
- XDPLY, [2967](#)
- XPBWPLYL, [2987](#)
- XPOLY, [2998](#)
- XPR, [3007](#)
- XPOLY, [3013](#)
- ZMOD, [1368](#)
- SUCH, [2627](#)
 - ?=, [2627](#)
 - ?~=, [2627](#)
- coerce, [2627](#)
- construct, [2627](#)
- hash, [2627](#)
- latex, [2627](#)
- lhs, [2627](#)
- rhs, [2627](#)
- SuchThat, [2627](#)
- suchThat
 - RULE, [2305](#)
- suffix?
 - ISTRING, [1249](#)
 - STRING, [2607](#)
- SULS, [2456](#)
 - , [2457](#)
 - ?<?, [2457](#)
 - ?<=, [2457](#)
 - ?>?, [2457](#)
 - ?>=, [2457](#)
 - ?**?, [2457](#)
 - ?*?, [2457](#)
 - ?+?, [2457](#)
 - ?-?, [2457](#)
 - ?., [2457](#)
 - ?/?, [2457](#)
 - ?=, [2457](#)
 - ?^?, [2457](#)
 - ?~=, [2457](#)
 - ?quo?, [2457](#)
 - ?rem?, [2457](#)
 - 0, [2457](#)
 - 1, [2457](#)
 - abs, [2457](#)
 - acos, [2457](#)
 - acosh, [2457](#)
 - acot, [2457](#)
 - acoth, [2457](#)
 - acsc, [2457](#)
 - acsch, [2457](#)
 - approximate, [2457](#)
 - asec, [2457](#)
 - asech, [2457](#)
 - asin, [2457](#)
 - asinh, [2457](#)
 - associates?, [2457](#)
 - atan, [2457](#)
 - atanh, [2457](#)

- ceiling, [2457](#)
- center, [2457](#)
- characteristic, [2457](#)
- charthRoot, [2457](#)
- coefficient, [2457](#)
- coerce, [2457](#)
- complete, [2457](#)
- conditionP, [2457](#)
- convert, [2457](#)
- cos, [2457](#)
- cosh, [2457](#)
- cot, [2457](#)
- coth, [2457](#)
- csc, [2457](#)
- csch, [2457](#)
- D, [2457](#)
- degree, [2457](#)
- denom, [2457](#)
- denominator, [2457](#)
- differentiate, [2457](#)
- divide, [2457](#)
- euclideanSize, [2457](#)
- eval, [2457](#)
- exp, [2457](#)
- expressIdealMember, [2457](#)
- exquo, [2457](#)
- extend, [2457](#)
- extendedEuclidean, [2457](#)
- factor, [2457](#)
- factorPolynomial, [2457](#)
- factorSquareFreePolynomial, [2457](#)
- floor, [2457](#)
- fractionPart, [2457](#)
- gcd, [2457](#)
- gcdPolynomial, [2457](#)
- hash, [2457](#)
- init, [2457](#)
- integrate, [2457](#)
- inv, [2457](#)
- latex, [2457](#)
- laurent, [2457](#)
- lcm, [2457](#)
- leadingCoefficient, [2457](#)
- leadingMonomial, [2457](#)
- log, [2457](#)
- map, [2457](#)
- max, [2457](#)
- min, [2457](#)
- monomial, [2457](#)
- monomial?, [2457](#)
- multiEuclidean, [2457](#)
- multiplyCoefficients, [2457](#)
- multiplyExponents, [2457](#)
- negative?, [2457](#)
- nextItem, [2457](#)
- nthRoot, [2457](#)
- numer, [2457](#)
- numerator, [2457](#)
- one?, [2457](#)
- order, [2457](#)
- patternMatch, [2457](#)
- pi, [2457](#)
- pole?, [2457](#)
- positive?, [2457](#)
- prime?, [2457](#)
- principalIdeal, [2457](#)
- random, [2457](#)
- rationalFunction, [2457](#)
- recip, [2457](#)
- reducedSystem, [2457](#)
- reductum, [2457](#)
- removeZeroes, [2457](#)
- retract, [2457](#)
- retractIfCan, [2457](#)
- sample, [2457](#)
- sec, [2457](#)
- sech, [2457](#)
- series, [2457](#)
- sign, [2457](#)
- sin, [2457](#)
- sinh, [2457](#)
- sizeLess?, [2457](#)
- solveLinearPolynomialEquation, [2457](#)
- sqrt, [2457](#)
- squareFree, [2457](#)
- squareFreePart, [2457](#)
- squareFreePolynomial, [2457](#)
- subtractIfCan, [2457](#)
- tan, [2457](#)
- tanh, [2457](#)
- taylor, [2457](#)
- taylorIfCan, [2457](#)

- taylorRep, 2457
- terms, 2457
- truncate, 2457
- unit?, 2457
- unitCanonical, 2457
- unitNormal, 2457
- variable, 2457
- variables, 2457
- wholePart, 2457
- zero?, 2457
- sum
 - OUTFORM, 1868
- summation
 - EXPR, 724
 - MYEXPR, 1688
- SUP, 2466
 - , 2467
 - ?<?, 2467
 - ?<=?, 2467
 - ?>?, 2467
 - ?>=?, 2467
 - ?**?, 2467
 - ?*?, 2467
 - ?+?, 2467
 - ?-?, 2467
 - ?., 2467
 - ?/? , 2467
 - ?=?, 2467
 - ?^?, 2467
 - ?~=?, 2467
 - ?quo?, 2467
 - ?rem?, 2467
 - 0, 2467
 - 1, 2467
 - associates?, 2467
 - binomThmExpt, 2467
 - characteristic, 2467
 - charthRoot, 2467
 - coefficient, 2467
 - coefficients, 2467
 - coerce, 2467
 - composite, 2467
 - conditionP, 2467
 - content, 2467
 - convert, 2467
 - D, 2467
 - degree, 2467
 - differentiate, 2467
 - discriminant, 2467
 - divide, 2467
 - divideExponents, 2467
 - elt, 2467
 - euclideanSize, 2467
 - eval, 2467
 - expressIdealMember, 2467
 - exquo, 2467
 - extendedEuclidean, 2467
 - factor, 2467
 - factorPolynomial, 2467
 - factorSquareFreePolynomial, 2467
 - fmeqg, 2467
 - gcd, 2467
 - gcdPolynomial, 2467
 - ground, 2467
 - ground?, 2467
 - hash, 2467
 - init, 2467
 - integrate, 2467
 - isExpt, 2467
 - isPlus, 2467
 - isTimes, 2467
 - karatsubaDivide, 2467
 - latex, 2467
 - lcm, 2467
 - leadingCoefficient, 2467
 - leadingMonomial, 2467
 - mainVariable, 2467
 - makeSUP, 2467
 - map, 2467
 - mapExponents, 2467
 - max, 2467
 - min, 2467
 - minimumDegree, 2467
 - monicDivide, 2467
 - monomial, 2467
 - monomial?, 2467
 - monomials, 2467
 - multiEuclidean, 2467
 - multiplyExponents, 2467
 - multivariate, 2467
 - nextItem, 2467
 - numberOfMonomials, 2467

- one?, [2467](#)
- order, [2467](#)
- outputForm, [2467](#)
- patternMatch, [2467](#)
- pomopo, [2467](#)
- prime?, [2467](#)
- primitiveMonomials, [2467](#)
- primitivePart, [2467](#)
- principalIdeal, [2467](#)
- pseudoDivide, [2467](#)
- pseudoQuotient, [2467](#)
- pseudoRemainder, [2467](#)
- recip, [2467](#)
- reducedSystem, [2467](#)
- reductum, [2467](#)
- resultant, [2467](#)
- retract, [2467](#)
- retractIfCan, [2467](#)
- sample, [2467](#)
- separate, [2467](#)
- shiftLeft, [2467](#)
- shiftRight, [2467](#)
- sizeLess?, [2467](#)
- solveLinearPolynomialEquation, [2467](#)
- squareFree, [2467](#)
- squareFreePart, [2467](#)
- squareFreePolynomial, [2467](#)
- subResultantGcd, [2467](#)
- subtractIfCan, [2467](#)
- totalDegree, [2467](#)
- unit?, [2467](#)
- unitCanonical, [2467](#)
- unitNormal, [2467](#)
- univariate, [2467](#)
- unmakeSUP, [2467](#)
- variables, [2467](#)
- vectorise, [2467](#)
- zero?, [2467](#)
- sup
 - DIRPROD, [547](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - HDP, [1173](#)
 - IDPOAMS, [1215](#)
 - INDE, [1218](#)
 - INTRVL, [1384](#)
 - NNI, [1739](#)
 - ODP, [1816](#)
 - PRODUCT, [2113](#)
 - SHDP, [2509](#)
 - super
 - OUTFORM, [1868](#)
 - superHeight
 - OUTFORM, [1868](#)
 - superscript
 - SYMBOL, [2640](#)
 - supersub
 - OUTFORM, [1868](#)
 - SUPEXPR, [2480](#)
 - , [2481](#)
 - ?<?, [2481](#)
 - ?<=?, [2481](#)
 - ?>?, [2481](#)
 - ?>=?, [2481](#)
 - ?**?, [2481](#)
 - ?*?, [2481](#)
 - ?+?, [2481](#)
 - ?-?, [2481](#)
 - ?., [2481](#)
 - ?/?, [2481](#)
 - ?=?, [2481](#)
 - ?^?, [2481](#)
 - ?~=?, [2481](#)
 - ?quo?, [2481](#)
 - ?rem?, [2481](#)
 - 0, [2481](#)
 - 1, [2481](#)
 - acos, [2481](#)
 - acosh, [2481](#)
 - acot, [2481](#)
 - acoth, [2481](#)
 - acsc, [2481](#)
 - acsch, [2481](#)
 - asec, [2481](#)
 - asech, [2481](#)
 - asin, [2481](#)
 - asinh, [2481](#)
 - associates?, [2481](#)
 - atan, [2481](#)
 - atanh, [2481](#)
 - binomThmExpt, [2481](#)
 - characteristic, [2481](#)

- charthRoot, 2481
- coefficient, 2481
- coefficients, 2481
- coerce, 2481
- composite, 2481
- conditionP, 2481
- content, 2481
- convert, 2481
- cos, 2481
- cosh, 2481
- cot, 2481
- coth, 2481
- csc, 2481
- csch, 2481
- D, 2481
- degree, 2481
- differentiate, 2481
- discriminant, 2481
- divide, 2481
- divideExponents, 2481
- elt, 2481
- euclideanSize, 2481
- eval, 2481
- exp, 2481
- expressIdealMember, 2481
- exquo, 2481
- extendedEuclidean, 2481
- factor, 2481
- factorPolynomial, 2481
- factorSquareFreePolynomial, 2481
- gcd, 2481
- gcdPolynomial, 2481
- ground, 2481
- ground?, 2481
- hash, 2481
- init, 2481
- integrate, 2481
- isExpt, 2481
- isPlus, 2481
- isTimes, 2481
- karatsubaDivide, 2481
- latex, 2481
- lcm, 2481
- leadingCoefficient, 2481
- leadingMonomial, 2481
- log, 2481
- mainVariable, 2481
- makeSUP, 2481
- map, 2481
- mapExponents, 2481
- max, 2481
- min, 2481
- minimumDegree, 2481
- monicDivide, 2481
- monomial, 2481
- monomial?, 2481
- monomials, 2481
- multiEuclidean, 2481
- multiplyExponents, 2481
- multivariate, 2481
- nextItem, 2481
- numberOfMonomials, 2481
- one?, 2481
- order, 2481
- patternMatch, 2481
- pi, 2481
- pomopo, 2481
- prime?, 2481
- primitiveMonomials, 2481
- primitivePart, 2481
- principalIdeal, 2481
- pseudoDivide, 2481
- pseudoQuotient, 2481
- pseudoRemainder, 2481
- recip, 2481
- reducedSystem, 2481
- reductum, 2481
- resultant, 2481
- retract, 2481
- retractIfCan, 2481
- sample, 2481
- sec, 2481
- sech, 2481
- separate, 2481
- shiftLeft, 2481
- shiftRight, 2481
- sin, 2481
- sinh, 2481
- sizeLess?, 2481
- solveLinearPolynomialEquation, 2481
- squareFree, 2481
- squareFreePart, 2481

- squareFreePolynomial, [2481](#)
- subResultantGcd, [2481](#)
- subtractIfCan, [2481](#)
- tan, [2481](#)
- tanh, [2481](#)
- totalDegree, [2481](#)
- unit?, [2481](#)
- unitCanonical, [2481](#)
- unitNormal, [2481](#)
- univariate, [2481](#)
- unmakeSUP, [2481](#)
- variables, [2481](#)
- vectorise, [2481](#)
- zero?, [2481](#)
- supp
 - DIV, [575](#)
- suppOfPole
 - DIV, [575](#)
- suppOfZero
 - DIV, [575](#)
- supRittWu?
 - NSMP, [1714](#)
- SUPXS, [2487](#)
- , [2487](#)
- ?**?, [2487](#)
- ?*?, [2487](#)
- ?+?, [2487](#)
- ?-?, [2487](#)
- ?., [2487](#)
- ?/?, [2487](#)
- ?=?, [2487](#)
- ?^?, [2487](#)
- ?~=?, [2487](#)
- ?quo?, [2487](#)
- ?rem?, [2487](#)
- 0, [2487](#)
- 1, [2487](#)
- acos, [2487](#)
- acosh, [2487](#)
- acot, [2487](#)
- acoth, [2487](#)
- acsc, [2487](#)
- acsch, [2487](#)
- approximate, [2487](#)
- asec, [2487](#)
- asech, [2487](#)
- asin, [2487](#)
- asinh, [2487](#)
- associates?, [2487](#)
- atan, [2487](#)
- atanh, [2487](#)
- center, [2487](#)
- characteristic, [2487](#)
- charthRoot, [2487](#)
- coefficient, [2487](#)
- coerce, [2487](#)
- complete, [2487](#)
- cos, [2487](#)
- cosh, [2487](#)
- cot, [2487](#)
- coth, [2487](#)
- csc, [2487](#)
- csch, [2487](#)
- D, [2487](#)
- degree, [2487](#)
- differentiate, [2487](#)
- divide, [2487](#)
- euclideanSize, [2487](#)
- eval, [2487](#)
- exp, [2487](#)
- expressIdealMember, [2487](#)
- exquo, [2487](#)
- extend, [2487](#)
- extendedEuclidean, [2487](#)
- factor, [2487](#)
- gcd, [2487](#)
- gcdPolynomial, [2487](#)
- hash, [2487](#)
- integrate, [2487](#)
- inv, [2487](#)
- latex, [2487](#)
- laurent, [2487](#)
- laurentIfCan, [2487](#)
- laurentRep, [2487](#)
- lcm, [2487](#)
- leadingCoefficient, [2487](#)
- leadingMonomial, [2487](#)
- log, [2487](#)
- map, [2487](#)
- monomial, [2487](#)
- monomial?, [2487](#)
- multiEuclidean, [2487](#)

- multiplyExponents, 2487
- nthRoot, 2487
- one?, 2487
- order, 2487
- pi, 2487
- pole?, 2487
- prime?, 2487
- principalIdeal, 2487
- puiseux, 2487
- rationalPower, 2487
- recip, 2487
- reductum, 2487
- retract, 2487
- retractIfCan, 2487
- sample, 2487
- sec, 2487
- sech, 2487
- series, 2487
- sin, 2487
- sinh, 2487
- sizeLess?, 2487
- sqrt, 2487
- squareFree, 2487
- squareFreePart, 2487
- subtractIfCan, 2487
- tan, 2487
- tanh, 2487
- terms, 2487
- truncate, 2487
- unit?, 2487
- unitCanonical, 2487
- unitNormal, 2487
- variable, 2487
- variables, 2487
- zero?, 2487
- surface
 - PARSURF, 1904
- SUTS, 2496
 - , 2497
 - ?**, 2497
 - *?, 2497
 - ?+, 2497
 - ?-, 2497
 - ?.?, 2497
 - ?/?, 2497
 - ?=, 2497
 - ?^?, 2497
 - ?~=, 2497
 - 0, 2497
 - 1, 2497
 - acos, 2497
 - acosh, 2497
 - acot, 2497
 - acoth, 2497
 - acsc, 2497
 - acsch, 2497
 - approximate, 2497
 - asec, 2497
 - asech, 2497
 - asin, 2497
 - asinh, 2497
 - associates?, 2497
 - atan, 2497
 - atanh, 2497
 - center, 2497
 - characteristic, 2497
 - charthRoot, 2497
 - coefficient, 2497
 - coefficients, 2497
 - coerce, 2497
 - complete, 2497
 - cos, 2497
 - cosh, 2497
 - cot, 2497
 - coth, 2497
 - csc, 2497
 - csch, 2497
 - D, 2497
 - degree, 2497
 - differentiate, 2497
 - eval, 2497
 - exp, 2497
 - exquo, 2497
 - extend, 2497
 - hash, 2497
 - integrate, 2497
 - latex, 2497
 - leadingCoefficient, 2497
 - leadingMonomial, 2497
 - log, 2497
 - map, 2497
 - monomial, 2497

- monomial?, [2497](#)
- multiplyCoefficients, [2497](#)
- multiplyExponents, [2497](#)
- nthRoot, [2497](#)
- one?, [2497](#)
- order, [2497](#)
- pi, [2497](#)
- pole?, [2497](#)
- polynomial, [2497](#)
- quoByVar, [2497](#)
- recip, [2497](#)
- reductum, [2497](#)
- sample, [2497](#)
- sec, [2497](#)
- sech, [2497](#)
- series, [2497](#)
- sin, [2497](#)
- sinh, [2497](#)
- sqrt, [2497](#)
- subtractIfCan, [2497](#)
- tan, [2497](#)
- tanh, [2497](#)
- terms, [2497](#)
- truncate, [2497](#)
- unit?, [2497](#)
- unitCanonical, [2497](#)
- unitNormal, [2497](#)
- univariatePolynomial, [2497](#)
- variable, [2497](#)
- variables, [2497](#)
- zero?, [2497](#)
- swap
 - EQ, [691](#)
- SWITCH, [2629](#)
 - AND, [2629](#)
 - coerce, [2629](#)
 - EQ, [2629](#)
 - GE, [2629](#)
 - GT, [2629](#)
 - LE, [2629](#)
 - LT, [2629](#)
 - NOT, [2629](#)
 - OR, [2629](#)
- Switch, [2629](#)
- symbNameV
 - IC, [1191](#)
- INFCLSPS, [1271](#)
- INFCLSPT, [1265](#)
- SYMBOL, [2640](#)
 - ?<?, [2640](#)
 - ?<=?, [2640](#)
 - ?>?, [2640](#)
 - ?>=?, [2640](#)
 - ?..?, [2640](#)
 - ?=?, [2640](#)
 - ?~=?, [2640](#)
 - argscript, [2640](#)
 - coerce, [2640](#)
 - convert, [2640](#)
 - hash, [2640](#)
 - latex, [2640](#)
 - list, [2640](#)
 - max, [2640](#)
 - min, [2640](#)
 - name, [2640](#)
 - new, [2640](#)
 - OMwrite, [2640](#)
 - patternMatch, [2640](#)
 - resetNew, [2640](#)
 - sample, [2640](#)
 - script, [2640](#)
 - scripted?, [2640](#)
 - scripts, [2640](#)
 - string, [2640](#)
 - subscript, [2640](#)
 - superscript, [2640](#)
- Symbol, [2640](#)
- symbol
 - INFORM, [1343](#)
 - SEX, [2392](#)
 - SEXOF, [2394](#)
- symbol?
 - INFORM, [1343](#)
 - PATTERN, [1928](#)
 - SEX, [2392](#)
 - SEXOF, [2394](#)
- symbolIfCan
 - KERNEL, [1404](#)
- SymbolTable, [2648](#)
- symbolTable
 - SYMTAB, [2648](#)
- symbolTableOf

- SYMS, 2696
- symmetric?
 - CDFMAT, 426
 - DFMAT, 599
 - DHMATRIX, 491
 - IMATRIX, 1239
 - LSQM, 1456
 - MATRIX, 1623
 - RMATRIX, 2246
 - SQMATRIX, 2548
 - U16MAT, 2906
 - U32MAT, 2911
 - U8MAT, 2902
- symmetricDifference
 - CCLASS, 382
 - MSET, 1670
 - SET, 2372
- SymmetricPolynomial, 2654
- symmetricPower
 - LODO, 1469
 - LODO1, 1479
 - LODO2, 1491
- symmetricProduct
 - LODO, 1469
 - LODO1, 1479
 - LODO2, 1491
- symmetricRemainder
 - INT, 1362
 - MINT, 1557
 - ROMAN, 2326
 - SINT, 2412
- symmetricSquare
 - LODO, 1469
 - LODO1, 1479
 - LODO2, 1491
- SYMPOLY, 2654
 - , 2654
 - ?**?, 2654
 - ?*?, 2654
 - ?+?, 2654
 - ?-?, 2654
 - ?/?, 2654
 - ?=?, 2654
 - ?^?, 2654
 - ?~=?, 2654
 - 0, 2654
 - 1, 2654
 - associates?, 2654
 - binomThmExpt, 2654
 - characteristic, 2654
 - charthRoot, 2654
 - coefficient, 2654
 - coefficients, 2654
 - coerce, 2654
 - content, 2654
 - degree, 2654
 - exquo, 2654
 - fmeq, 2654
 - ground, 2654
 - ground?, 2654
 - hash, 2654
 - latex, 2654
 - leadingCoefficient, 2654
 - leadingMonomial, 2654
 - map, 2654
 - mapExponents, 2654
 - minimumDegree, 2654
 - monomial, 2654
 - monomial?, 2654
 - numberOfMonomials, 2654
 - one?, 2654
 - pomopo, 2654
 - primitivePart, 2654
 - recip, 2654
 - reductum, 2654
 - retract, 2654
 - retractIfCan, 2654
 - sample, 2654
 - subtractIfCan, 2654
 - unit?, 2654
 - unitCanonical, 2654
 - unitNormal, 2654
 - zero?, 2654
- SYMS, 2696
 - argumentList, 2696
 - argumentListOf, 2696
 - clearTheSymbolTable, 2696
 - coerce, 2696
 - currentSubProgram, 2696
 - declare, 2696
 - empty, 2696
 - endSubProgram, 2696

- newSubProgram, [2696](#)
- printHeader, [2696](#)
- printTypes, [2696](#)
- returnType, [2696](#)
- returnTypeOf, [2696](#)
- showTheSymbolTable, [2696](#)
- symbolTableOf, [2696](#)
- SYMTAB, [2648](#)
 - coerce, [2648](#)
 - declare, [2648](#)
 - empty, [2648](#)
 - externalList, [2648](#)
 - fortranTypeOf, [2648](#)
 - newTypeLists, [2648](#)
 - parametersOf, [2648](#)
 - printTypes, [2648](#)
 - symbolTable, [2648](#)
 - typeList, [2648](#)
 - typeLists, [2648](#)
- TABLE, [2663](#)
 - ?.?, [2664](#)
 - ?=?, [2664](#)
 - ?~=?, [2664](#)
 - #?, [2664](#)
 - any?, [2664](#)
 - bag, [2664](#)
 - coerce, [2664](#)
 - construct, [2664](#)
 - convert, [2664](#)
 - copy, [2664](#)
 - count, [2664](#)
 - dictionary, [2664](#)
 - elt, [2664](#)
 - empty, [2664](#)
 - empty?, [2664](#)
 - entries, [2664](#)
 - entry?, [2664](#)
 - eq?, [2664](#)
 - eval, [2664](#)
 - every?, [2664](#)
 - extract, [2664](#)
 - fill, [2664](#)
 - find, [2664](#)
 - first, [2664](#)
 - hash, [2664](#)
 - index?, [2664](#)
 - indices, [2664](#)
 - insert, [2664](#)
 - inspect, [2664](#)
 - key?, [2664](#)
 - keys, [2664](#)
 - latex, [2664](#)
 - less?, [2664](#)
 - map, [2664](#)
 - maxIndex, [2664](#)
 - member?, [2664](#)
 - members, [2664](#)
 - minIndex, [2664](#)
 - more?, [2664](#)
 - parts, [2664](#)
 - qelt, [2664](#)
 - qsetelt, [2664](#)
 - reduce, [2664](#)
 - remove, [2664](#)
 - removeDuplicates, [2664](#)
 - sample, [2664](#)
 - search, [2664](#)
 - select, [2664](#)
 - setelt, [2664](#)
 - size?, [2664](#)
 - swap, [2664](#)
 - table, [2664](#)
- Table, [2663](#)
- table
 - ALIST, [235](#)
 - EQTBL, [699](#)
 - GSTBL, [1079](#)
 - HASHTBL, [1120](#)
 - INTABL, [1336](#)
 - KAFILE, [1414](#)
 - LIB, [1429](#)
 - RESULT, [2301](#)
 - ROUTINE, [2331](#)
 - STBL, [2450](#)
 - STRTBL, [2610](#)
 - TABLE, [2664](#)
- TABLEAU, [2666](#)
 - coerce, [2666](#)
 - listOfLists, [2666](#)
 - tableau, [2666](#)
- Tableau, [2666](#)

- tableau
 - TABLEAU, [2666](#)
- tableForDiscreteLogarithm
 - ALGFF, [28](#)
 - COMPLEX, [419](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - MCMPLEX, [1543](#)
 - PACOFF, [2136](#)
 - PF, [2105](#)
 - RADFF, [2194](#)
 - SAE, [2400](#)
- tail
 - ALIST, [235](#)
 - DLIST, [460](#)
 - ILIST, [1231](#)
 - LIST, [1504](#)
 - NSDPS, [1703](#)
 - NSMP, [1714](#)
 - STREAM, [2582](#)
- tan
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FEXPR, [947](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLEX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
- TS, [2671](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- tanh
 - COMPLEX, [419](#)
 - DFLOAT, [587](#)
 - EFULS, [672](#)
 - EFUPXS, [681](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FEXPR, [947](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - INTRVL, [1384](#)
 - MCMPLEX, [1543](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- tanhIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- tanIfCan
 - EFULS, [672](#)
 - EFUPXS, [681](#)
- taylor
 - SULS, [2457](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
- taylorIfCan
 - SULS, [2457](#)
 - ULS, [2796](#)

- ULSCONS, [2804](#)
- taylorQuoByVar
 - ISUPS, [1311](#)
- taylorRep
 - SULS, [2457](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
- TaylorSeries, [2670](#)
- terms
 - DIV, [575](#)
 - EXPUPXS, [740](#)
 - FAGROUP, [1004](#)
 - FAMONOID, [1007](#)
 - GSERIES, [1090](#)
 - IFAMON, [1287](#)
 - ISUPS, [1311](#)
 - MRING, [1659](#)
 - NSDPS, [1703](#)
 - SULS, [2457](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UPXS, [2834](#)
 - UPXSCONS, [2842](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- test
 - BOOLEAN, [322](#)
- TEX, [2676](#)
 - ?=?, [2677](#)
 - ?~=?, [2677](#)
 - coerce, [2677](#)
 - convert, [2677](#)
 - display, [2677](#)
 - epilogue, [2677](#)
 - hash, [2677](#)
 - latex, [2677](#)
 - new, [2677](#)
 - prologue, [2677](#)
 - setEpilogue, [2677](#)
 - setPrologue, [2677](#)
 - setTex, [2677](#)
 - tex, [2677](#)
- tex
 - TEX, [2677](#)
- TexFormat, [2676](#)
- TEXTFILE, [2692](#)
 - ?=?, [2692](#)
 - ?~=?, [2692](#)
 - close, [2692](#)
 - coerce, [2692](#)
 - endOfFile?, [2692](#)
 - hash, [2692](#)
 - iomode, [2692](#)
 - latex, [2692](#)
 - name, [2692](#)
 - open, [2692](#)
 - read, [2692](#)
 - readIfCan, [2692](#)
 - readLine, [2692](#)
 - readLineIfCan, [2692](#)
 - reopen, [2692](#)
 - write, [2692](#)
 - writeLine, [2692](#)
- TextFile, [2692](#)
- TheSymbolTable, [2696](#)
- third
 - ALIST, [235](#)
 - DLIST, [460](#)
 - ILIST, [1231](#)
 - LIST, [1504](#)
 - NSDPS, [1703](#)
 - STREAM, [2582](#)
- ThreeDimensionalMatrix, [2702](#)
- ThreeDimensionalViewport, [2710](#)
- ThreeSpace, [2731](#)
- title
 - DROPT, [608](#)
 - VIEW2d, [2770](#)
 - VIEW3D, [2710](#)
- top
 - ASTACK, [65](#)
 - DEQUEUE, [512](#)
 - STACK, [2563](#)
- topPredicate
 - PATTERN, [1928](#)
- toScale
 - DROPT, [608](#)
- totalDegree
 - DMP, [572](#)

- DSMP, [542](#)
- GDMP, [1052](#)
- HDMP, [1180](#)
- MODMON, [1632](#)
- MPOLY, [1682](#)
- MYUP, [1695](#)
- NSMP, [1714](#)
- NSUP, [1729](#)
- ODPOL, [1852](#)
- POLY, [2078](#)
- SDPOL, [2386](#)
- SMP, [2423](#)
- SUP, [2467](#)
- SUPEXP, [2481](#)
- UP, [2828](#)
- totalDifferential
 - DERHAM, [530](#)
- tower
 - AN, [35](#)
 - EXPR, [724](#)
 - FEXPR, [947](#)
 - IAN, [1276](#)
 - MYEXPR, [1688](#)
- trace
 - ALGFF, [28](#)
 - COMPLEX, [419](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - LSQM, [1456](#)
 - MCMPLX, [1543](#)
 - PF, [2105](#)
 - RADFF, [2194](#)
 - SAE, [2400](#)
 - SQMATRIX, [2548](#)
- traceMatrix
 - ALGFF, [28](#)
 - COMPLEX, [419](#)
 - MCMPLX, [1543](#)
 - RADFF, [2194](#)
 - SAE, [2400](#)
- trailingCoefficient
 - LAUPOL, [1422](#)
- tRange
 - PLOT, [2029](#)
 - PLOT3D, [2042](#)
- transcendenceDegree
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
 - PF, [2105](#)
- transcendent?
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - IFF, [1284](#)
 - IPF, [1303](#)
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
 - PF, [2105](#)
- transCoord
 - BLHN, [317](#)
 - BLQT, [319](#)
- translate
 - DHMATRIX, [491](#)
 - VIEW2d, [2770](#)
 - VIEW3D, [2710](#)
- transpose

- CARTEN, [356](#)
- CDFMAT, [426](#)
- DFMAT, [599](#)
- DHMATRIX, [491](#)
- IMATRIX, [1239](#)
- MATRIX, [1623](#)
- SQMATRIX, [2548](#)
- U16MAT, [2906](#)
- U32MAT, [2911](#)
- U8MAT, [2902](#)
- traverse
 - SUBSPACE, [2614](#)
- TREE, [2740](#)
 - ?.value, [2741](#)
 - ?=?, [2741](#)
 - ?~=?, [2741](#)
 - #?, [2741](#)
 - any?, [2741](#)
 - child?, [2741](#)
 - children, [2741](#)
 - coerce, [2741](#)
 - copy, [2741](#)
 - count, [2741](#)
 - cyclic?, [2741](#)
 - cyclicCopy, [2741](#)
 - cyclicEntries, [2741](#)
 - cyclicEqual?, [2741](#)
 - cyclicParents, [2741](#)
 - distance, [2741](#)
 - empty, [2741](#)
 - empty?, [2741](#)
 - eq?, [2741](#)
 - eval, [2741](#)
 - every?, [2741](#)
 - hash, [2741](#)
 - latex, [2741](#)
 - leaf?, [2741](#)
 - leaves, [2741](#)
 - less?, [2741](#)
 - map, [2741](#)
 - member?, [2741](#)
 - members, [2741](#)
 - more?, [2741](#)
 - node?, [2741](#)
 - nodes, [2741](#)
 - parts, [2741](#)
 - sample, [2741](#)
 - setchildren, [2741](#)
 - setelt, [2741](#)
 - setvalue, [2741](#)
 - size?, [2741](#)
 - tree, [2741](#)
 - value, [2741](#)
- Tree, [2740](#)
- tree
 - DSTREE, [535](#)
 - TREE, [2741](#)
- triangular?
 - GPOLSET, [1074](#)
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- trim
 - ISTRING, [1249](#)
 - STRING, [2607](#)
- trivialIdeal?
 - GPOLSET, [1074](#)
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- true
 - BOOLEAN, [322](#)
- trueEqual
 - IAN, [1276](#)
- trunc
 - LPOLY, [1447](#)
 - XDPOLY, [2967](#)
 - XPBWPOLYL, [2987](#)
 - XPOLY, [2998](#)
 - XPOLY, [3013](#)
- truncate
 - DFLOAT, [587](#)
 - EXPUPXS, [740](#)
 - FLOAT, [908](#)
 - GSERIES, [1090](#)
 - ISUPS, [1311](#)
 - MFLOAT, [1548](#)
 - NSDPS, [1703](#)

- SULS, [2457](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- TS, [2670](#)
 - ?, [2671](#)
 - ?**?, [2671](#)
 - ?*?, [2671](#)
 - ?+?, [2671](#)
 - ?-?, [2671](#)
 - ?/?, [2671](#)
 - ?=?, [2671](#)
 - ?^?, [2671](#)
 - ?~=?, [2671](#)
 - 0, [2671](#)
 - 1, [2671](#)
 - acos, [2671](#)
 - acosh, [2671](#)
 - acot, [2671](#)
 - acoth, [2671](#)
 - acsc, [2671](#)
 - acsch, [2671](#)
 - asec, [2671](#)
 - asech, [2671](#)
 - asin, [2671](#)
 - asinh, [2671](#)
 - associates?, [2671](#)
 - atan, [2671](#)
 - atanh, [2671](#)
 - characteristic, [2671](#)
 - charthRoot, [2671](#)
 - coefficient, [2671](#)
 - coerce, [2671](#)
 - complete, [2671](#)
 - cos, [2671](#)
 - cosh, [2671](#)
 - cot, [2671](#)
 - coth, [2671](#)
 - csc, [2671](#)
 - csch, [2671](#)
 - D, [2671](#)
 - degree, [2671](#)
 - differentiate, [2671](#)
 - eval, [2671](#)
 - exp, [2671](#)
 - exquo, [2671](#)
 - extend, [2671](#)
 - fintegrate, [2671](#)
 - hash, [2671](#)
 - integrate, [2671](#)
 - latex, [2671](#)
 - leadingCoefficient, [2671](#)
 - leadingMonomial, [2671](#)
 - log, [2671](#)
 - map, [2671](#)
 - monomial, [2671](#)
 - monomial?, [2671](#)
 - nthRoot, [2671](#)
 - one?, [2671](#)
 - order, [2671](#)
 - pi, [2671](#)
 - pole?, [2671](#)
 - polynomial, [2671](#)
 - recip, [2671](#)
 - reductum, [2671](#)
 - sample, [2671](#)
 - sec, [2671](#)
 - sech, [2671](#)
 - sin, [2671](#)
 - sinh, [2671](#)
 - sqrt, [2671](#)
 - subtractIfCan, [2671](#)
 - tan, [2671](#)
 - tanh, [2671](#)
 - unit?, [2671](#)
 - unitCanonical, [2671](#)
 - unitNormal, [2671](#)
 - variables, [2671](#)
 - zero?, [2671](#)
- TUBE, [2749](#)
 - closed?, [2749](#)
 - getCurve, [2749](#)
 - listLoops, [2749](#)
 - open?, [2749](#)
 - setClosed, [2749](#)
 - tube, [2749](#)

- tube
 - TUBE, [2749](#)
- TubePlot, [2749](#)
- tubePoints
 - DROPT, [608](#)
- tubeRadius
 - DROPT, [608](#)
- TUPLE, [2752](#)
 - ?=?, [2752](#)
 - ?~=?, [2752](#)
 - coerce, [2752](#)
 - hash, [2752](#)
 - latex, [2752](#)
 - length, [2752](#)
 - select, [2752](#)
- Tuple, [2752](#)
- tValues
 - PLOT3D, [2042](#)
- TwoDimensionalArray, [2763](#)
- TwoDimensionalViewport, [2769](#)
- type
 - BLHN, [317](#)
 - BLQT, [319](#)
- typeList
 - SYMTAB, [2648](#)
- typeLists
 - SYMTAB, [2648](#)
- U16MAT, [2906](#)
 - .?, [2906](#)
 - ?**?, [2906](#)
 - ?*?, [2906](#)
 - ?+?, [2906](#)
 - ?-?, [2906](#)
 - ?/?, [2906](#)
 - ?=?, [2906](#)
 - ?~=?, [2906](#)
 - #?, [2906](#)
 - antisymmetric?, [2906](#)
 - any?, [2906](#)
 - coerce, [2906](#)
 - column, [2906](#)
 - columnSpace, [2906](#)
 - copy, [2906](#)
 - count, [2906](#)
 - determinant, [2906](#)
 - diagonal?, [2906](#)
 - diagonalMatrix, [2906](#)
 - elt, [2906](#)
 - empty, [2906](#)
 - empty?, [2906](#)
 - eq?, [2906](#)
 - eval, [2906](#)
 - every?, [2906](#)
 - exquo, [2906](#)
 - fill, [2906](#)
 - hash, [2906](#)
 - horizConcat, [2906](#)
 - inverse, [2906](#)
 - latex, [2906](#)
 - less?, [2906](#)
 - listOfLists, [2906](#)
 - map, [2906](#)
 - matrix, [2906](#)
 - maxColIndex, [2906](#)
 - maxRowIndex, [2906](#)
 - member?, [2906](#)
 - members, [2906](#)
 - minColIndex, [2906](#)
 - minordet, [2906](#)
 - minRowIndex, [2906](#)
 - more?, [2906](#)
 - ncols, [2906](#)
 - new, [2906](#)
 - nrows, [2906](#)
 - nullity, [2906](#)
 - nullSpace, [2906](#)
 - parts, [2906](#)
 - pfaffian, [2906](#)
 - qelt, [2906](#)
 - qnew, [2906](#)
 - qsetelt, [2906](#)
 - rank, [2906](#)
 - row, [2906](#)
 - rowEchelon, [2906](#)
 - sample, [2906](#)
 - scalarMatrix, [2906](#)
 - setColumn, [2906](#)
 - setelt, [2906](#)
 - setRow, [2906](#)
 - setsubMatrix, [2906](#)
 - size?, [2906](#)

- square?, 2906
- squareTop, 2906
- subMatrix, 2906
- swapColumns, 2906
- swapRows, 2906
- symmetric?, 2906
- transpose, 2906
- vertConcat, 2906
- zero, 2906
- U16Matrix, 2906
- U16VEC, 2924
 - ?<?, 2924
 - ?<=?, 2924
 - ?>?, 2924
 - ?>=?, 2924
 - ?..?, 2924
 - ?=?, 2924
 - ?~=?, 2924
 - #?, 2924
 - any?, 2924
 - coerce, 2924
 - concat, 2924
 - construct, 2924
 - convert, 2924
 - copy, 2924
 - copyInto, 2924
 - count, 2924
 - delete, 2924
 - elt, 2924
 - empty, 2924
 - empty?, 2924
 - entries, 2924
 - entry?, 2924
 - eq?, 2924
 - eval, 2924
 - every?, 2924
 - fill, 2924
 - find, 2924
 - first, 2924
 - hash, 2924
 - index?, 2924
 - indices, 2924
 - insert, 2924
 - latex, 2924
 - less?, 2924
 - map, 2924
 - max, 2924
 - maxIndex, 2924
 - member?, 2924
 - members, 2924
 - merge, 2924
 - min, 2924
 - minIndex, 2924
 - more?, 2924
 - new, 2924
 - parts, 2924
 - position, 2924
 - qelt, 2924
 - qsetelt, 2924
 - reduce, 2924
 - remove, 2924
 - removeDuplicates, 2924
 - reverse, 2924
 - sample, 2924
 - select, 2924
 - setelt, 2924
 - size?, 2924
 - sort, 2924
 - sorted?, 2924
 - swap, 2924
- U16Vector, 2924
- U32MAT, 2910
 - , 2911
 - ?**?, 2911
 - ?*?, 2911
 - ?+?, 2911
 - ?-?, 2911
 - ?/?, 2911
 - ?=?, 2911
 - ?~=?, 2911
 - #?, 2911
 - antisymmetric?, 2911
 - any?, 2911
 - coerce, 2911
 - column, 2911
 - columnSpace, 2911
 - copy, 2911
 - count, 2911
 - determinant, 2911
 - diagonal?, 2911
 - diagonalMatrix, 2911
 - elt, 2911

- empty, 2911
- empty?, 2911
- eq?, 2911
- eval, 2911
- every?, 2911
- exquo, 2911
- fill, 2911
- hash, 2911
- horizConcat, 2911
- inverse, 2911
- latex, 2911
- less?, 2911
- listOfLists, 2911
- map, 2911
- matrix, 2911
- maxColIndex, 2911
- maxRowIndex, 2911
- member?, 2911
- members, 2911
- minColIndex, 2911
- minordet, 2911
- minRowIndex, 2911
- more?, 2911
- ncols, 2911
- new, 2911
- nrows, 2911
- nullity, 2911
- nullSpace, 2911
- parts, 2911
- pfaffian, 2911
- qelt, 2911
- qnew, 2911
- qsetelt, 2911
- rank, 2911
- row, 2911
- rowEchelon, 2911
- sample, 2911
- scalarMatrix, 2911
- setColumn, 2911
- setelt, 2911
- setRow, 2911
- setsubMatrix, 2911
- size?, 2911
- square?, 2911
- squareTop, 2911
- subMatrix, 2911
- swapColumns, 2911
- swapRows, 2911
- symmetric?, 2911
- transpose, 2911
- vertConcat, 2911
- zero, 2911
- U32Matrix, 2910
- U32VEC, 2931
 - ?<?, 2931
 - ?<=?, 2931
 - ?>?, 2931
 - ?>=?, 2931
 - ?..?, 2931
 - ?=?, 2931
 - ?~=?, 2931
 - #?, 2931
 - any?, 2931
 - coerce, 2931
 - concat, 2931
 - construct, 2931
 - convert, 2931
 - copy, 2931
 - copyInto, 2931
 - count, 2931
 - delete, 2931
 - elt, 2931
 - empty, 2931
 - empty?, 2931
 - entries, 2931
 - entry?, 2931
 - eq?, 2931
 - eval, 2931
 - every?, 2931
 - fill, 2931
 - find, 2931
 - first, 2931
 - hash, 2931
 - index?, 2931
 - indices, 2931
 - insert, 2931
 - latex, 2931
 - less?, 2931
 - map, 2931
 - max, 2931
 - maxIndex, 2931
 - member?, 2931

- members, [2931](#)
- merge, [2931](#)
- min, [2931](#)
- minIndex, [2931](#)
- more?, [2931](#)
- new, [2931](#)
- parts, [2931](#)
- position, [2931](#)
- qelt, [2931](#)
- qsetelt, [2931](#)
- reduce, [2931](#)
- remove, [2931](#)
- removeDuplicates, [2931](#)
- reverse, [2931](#)
- sample, [2931](#)
- select, [2931](#)
- setelt, [2931](#)
- size?, [2931](#)
- sort, [2931](#)
- sorted?, [2931](#)
- swap, [2931](#)
- U32Vector, [2931](#)
- U8MAT, [2901](#)
- ?, [2902](#)
- ?**?, [2902](#)
- ?*?, [2902](#)
- ?+?, [2902](#)
- ?-?, [2902](#)
- ?/?, [2902](#)
- ?=?, [2902](#)
- ?~=?, [2902](#)
- #?, [2902](#)
- antisymmetric?, [2902](#)
- any?, [2902](#)
- coerce, [2902](#)
- column, [2902](#)
- columnSpace, [2902](#)
- copy, [2902](#)
- count, [2902](#)
- determinant, [2902](#)
- diagonal?, [2902](#)
- diagonalMatrix, [2902](#)
- elt, [2902](#)
- empty, [2902](#)
- empty?, [2902](#)
- eq?, [2902](#)
- eval, [2902](#)
- every?, [2902](#)
- exquo, [2902](#)
- fill, [2902](#)
- hash, [2902](#)
- horizConcat, [2902](#)
- inverse, [2902](#)
- latex, [2902](#)
- less?, [2902](#)
- listOfLists, [2902](#)
- map, [2902](#)
- matrix, [2902](#)
- maxColIndex, [2902](#)
- maxRowIndex, [2902](#)
- member?, [2902](#)
- members, [2902](#)
- minColIndex, [2902](#)
- minordet, [2902](#)
- minRowIndex, [2902](#)
- more?, [2902](#)
- ncols, [2902](#)
- new, [2902](#)
- nrows, [2902](#)
- nullity, [2902](#)
- nullSpace, [2902](#)
- parts, [2902](#)
- pfaffian, [2902](#)
- qelt, [2902](#)
- qnew, [2902](#)
- qsetelt, [2902](#)
- rank, [2902](#)
- row, [2902](#)
- rowEchelon, [2902](#)
- sample, [2902](#)
- scalarMatrix, [2902](#)
- setColumn, [2902](#)
- setelt, [2902](#)
- setRow, [2902](#)
- setsubMatrix, [2902](#)
- size?, [2902](#)
- square?, [2902](#)
- squareTop, [2902](#)
- subMatrix, [2902](#)
- swapColumns, [2902](#)
- swapRows, [2902](#)
- symmetric?, [2902](#)

- transpose, 2902
- vertConcat, 2902
- zero, 2902
- U8Matrix, 2901
- U8VEC, 2917
 - ?<?, 2917
 - ?<=?, 2917
 - ?>?, 2917
 - ?>=?, 2917
 - ?.?, 2917
 - ?=?, 2917
 - ?~=?, 2917
 - #?, 2917
 - any?, 2917
 - coerce, 2917
 - concat, 2917
 - construct, 2917
 - convert, 2917
 - copy, 2917
 - copyInto, 2917
 - count, 2917
 - delete, 2917
 - elt, 2917
 - empty, 2917
 - empty?, 2917
 - entries, 2917
 - entry?, 2917
 - eq?, 2917
 - eval, 2917
 - every?, 2917
 - fill, 2917
 - find, 2917
 - first, 2917
 - hash, 2917
 - index?, 2917
 - indices, 2917
 - insert, 2917
 - latex, 2917
 - less?, 2917
 - map, 2917
 - max, 2917
 - maxIndex, 2917
 - member?, 2917
 - members, 2917
 - merge, 2917
 - min, 2917
 - minIndex, 2917
 - more?, 2917
 - new, 2917
 - parts, 2917
 - position, 2917
 - qelt, 2917
 - qsetelt, 2917
 - reduce, 2917
 - remove, 2917
 - removeDuplicates, 2917
 - reverse, 2917
 - sample, 2917
 - select, 2917
 - setelt, 2917
 - size?, 2917
 - sort, 2917
 - sorted?, 2917
 - swap, 2917
- U8Vector, 2917
- UFPS, 2788
 - .?, 2789
 - ?**?, 2789
 - ?*?, 2789
 - ?+?, 2789
 - ?-?, 2789
 - ?.?, 2789
 - ?/? , 2789
 - ?=?, 2789
 - ?^?, 2789
 - ?~=?, 2789
 - 0, 2789
 - 1, 2789
 - acos, 2789
 - acosh, 2789
 - acot, 2789
 - acoth, 2789
 - acsc, 2789
 - acsch, 2789
 - approximate, 2789
 - asec, 2789
 - asech, 2789
 - asin, 2789
 - asinh, 2789
 - associates?, 2789
 - atan, 2789
 - atanh, 2789

- center, 2789
- characteristic, 2789
- charthRoot, 2789
- coefficient, 2789
- coefficients, 2789
- coerce, 2789
- complete, 2789
- cos, 2789
- cosh, 2789
- cot, 2789
- coth, 2789
- csc, 2789
- csch, 2789
- D, 2789
- degree, 2789
- differentiate, 2789
- eval, 2789
- evenlambert, 2789
- exp, 2789
- exquo, 2789
- extend, 2789
- generalLambert, 2789
- hash, 2789
- integrate, 2789
- invmultisect, 2789
- lagrange, 2789
- lambert, 2789
- latex, 2789
- leadingCoefficient, 2789
- leadingMonomial, 2789
- log, 2789
- map, 2789
- monomial, 2789
- monomial?, 2789
- multiplyCoefficients, 2789
- multiplyExponents, 2789
- multisect, 2789
- nthRoot, 2789
- oddlambert, 2789
- one?, 2789
- order, 2789
- pi, 2789
- pole?, 2789
- polynomial, 2789
- quoByVar, 2789
- recip, 2789
- reductum, 2789
- revert, 2789
- sample, 2789
- sec, 2789
- sech, 2789
- series, 2789
- sin, 2789
- sinh, 2789
- sqrt, 2789
- subtractIfCan, 2789
- tan, 2789
- tanh, 2789
- terms, 2789
- truncate, 2789
- unit?, 2789
- unitCanonical, 2789
- unitNormal, 2789
- univariatePolynomial, 2789
- variable, 2789
- variables, 2789
- zero?, 2789
- ULS, 2795
- , 2796
- ?<?, 2796
- ?<=?, 2796
- ?>?, 2796
- ?>=?, 2796
- ?**?, 2796
- ?*?, 2796
- ?+?, 2796
- ?-?, 2796
- ?., 2796
- ?/?, 2796
- ?=?, 2796
- ?^?, 2796
- ?~=?, 2796
- ?quo?, 2796
- ?rem?, 2796
- 0, 2796
- 1, 2796
- abs, 2796
- acos, 2796
- acosh, 2796
- acot, 2796
- acoth, 2796
- acsc, 2796

- acsch, 2796
- approximate, 2796
- asec, 2796
- asech, 2796
- asin, 2796
- asinh, 2796
- associates?, 2796
- atan, 2796
- atanh, 2796
- ceiling, 2796
- center, 2796
- characteristic, 2796
- charthRoot, 2796
- coefficient, 2796
- coerce, 2796
- complete, 2796
- conditionP, 2796
- convert, 2796
- cos, 2796
- cosh, 2796
- cot, 2796
- coth, 2796
- csc, 2796
- csch, 2796
- D, 2796
- degree, 2796
- denom, 2796
- denominator, 2796
- differentiate, 2796
- divide, 2796
- euclideanSize, 2796
- eval, 2796
- exp, 2796
- expressIdealMember, 2796
- exquo, 2796
- extend, 2796
- extendedEuclidean, 2796
- factor, 2796
- factorPolynomial, 2796
- factorSquareFreePolynomial, 2796
- floor, 2796
- fractionPart, 2796
- gcd, 2796
- gcdPolynomial, 2796
- hash, 2796
- init, 2796
- integrate, 2796
- inv, 2796
- latex, 2796
- laurent, 2796
- lcm, 2796
- leadingCoefficient, 2796
- leadingMonomial, 2796
- log, 2796
- map, 2796
- max, 2796
- min, 2796
- monomial, 2796
- monomial?, 2796
- multiEuclidean, 2796
- multiplyCoefficients, 2796
- multiplyExponents, 2796
- negative?, 2796
- nextItem, 2796
- nthRoot, 2796
- numer, 2796
- numerator, 2796
- one?, 2796
- order, 2796
- patternMatch, 2796
- pi, 2796
- pole?, 2796
- positive?, 2796
- prime?, 2796
- principalIdeal, 2796
- random, 2796
- rationalFunction, 2796
- recip, 2796
- reducedSystem, 2796
- reductum, 2796
- removeZeroes, 2796
- retract, 2796
- retractIfCan, 2796
- sample, 2796
- sec, 2796
- sech, 2796
- series, 2796
- sign, 2796
- sin, 2796
- sinh, 2796
- sizeLess?, 2796
- solveLinearPolynomialEquation, 2796

- sqrt, [2796](#)
- squareFree, [2796](#)
- squareFreePart, [2796](#)
- squareFreePolynomial, [2796](#)
- subtractIfCan, [2796](#)
- tan, [2796](#)
- tanh, [2796](#)
- taylor, [2796](#)
- taylorIfCan, [2796](#)
- taylorRep, [2796](#)
- terms, [2796](#)
- truncate, [2796](#)
- unit?, [2796](#)
- unitCanonical, [2796](#)
- unitNormal, [2796](#)
- variable, [2796](#)
- variables, [2796](#)
- wholePart, [2796](#)
- zero?, [2796](#)
- ULSCONS, [2803](#)
- ?, [2804](#)
- ?<?, [2804](#)
- ?<=?, [2804](#)
- ?>?, [2804](#)
- ?>=?, [2804](#)
- ?**?, [2804](#)
- ?*?, [2804](#)
- ?+?, [2804](#)
- ?-?, [2804](#)
- ?., [2804](#)
- ?/? , [2804](#)
- ?=?, [2804](#)
- ?^?, [2804](#)
- ?~=?, [2804](#)
- ?quo?, [2804](#)
- ?rem?, [2804](#)
- 0, [2804](#)
- 1, [2804](#)
- abs, [2804](#)
- acos, [2804](#)
- acosh, [2804](#)
- acot, [2804](#)
- acoth, [2804](#)
- acsc, [2804](#)
- acsch, [2804](#)
- approximate, [2804](#)
- asec, [2804](#)
- asech, [2804](#)
- asin, [2804](#)
- asinh, [2804](#)
- associates?, [2804](#)
- atan, [2804](#)
- atanh, [2804](#)
- ceiling, [2804](#)
- center, [2804](#)
- characteristic, [2804](#)
- charthRoot, [2804](#)
- coefficient, [2804](#)
- coerce, [2804](#)
- complete, [2804](#)
- conditionP, [2804](#)
- convert, [2804](#)
- cos, [2804](#)
- cosh, [2804](#)
- cot, [2804](#)
- coth, [2804](#)
- csc, [2804](#)
- csch, [2804](#)
- D, [2804](#)
- degree, [2804](#)
- denom, [2804](#)
- denominator, [2804](#)
- differentiate, [2804](#)
- divide, [2804](#)
- euclideanSize, [2804](#)
- eval, [2804](#)
- exp, [2804](#)
- expressIdealMember, [2804](#)
- exquo, [2804](#)
- extend, [2804](#)
- extendedEuclidean, [2804](#)
- factor, [2804](#)
- factorPolynomial, [2804](#)
- factorSquareFreePolynomial, [2804](#)
- floor, [2804](#)
- fractionPart, [2804](#)
- gcd, [2804](#)
- gcdPolynomial, [2804](#)
- hash, [2804](#)
- init, [2804](#)
- integrate, [2804](#)
- inv, [2804](#)

- latex, [2804](#)
- laurent, [2804](#)
- lcm, [2804](#)
- leadingCoefficient, [2804](#)
- leadingMonomial, [2804](#)
- log, [2804](#)
- map, [2804](#)
- max, [2804](#)
- min, [2804](#)
- monomial, [2804](#)
- monomial?, [2804](#)
- multiEuclidean, [2804](#)
- multiplyCoefficients, [2804](#)
- multiplyExponents, [2804](#)
- negative?, [2804](#)
- nextItem, [2804](#)
- nthRoot, [2804](#)
- numer, [2804](#)
- numerator, [2804](#)
- one?, [2804](#)
- order, [2804](#)
- patternMatch, [2804](#)
- pi, [2804](#)
- pole?, [2804](#)
- positive?, [2804](#)
- prime?, [2804](#)
- principalIdeal, [2804](#)
- random, [2804](#)
- rationalFunction, [2804](#)
- recip, [2804](#)
- reducedSystem, [2804](#)
- reductum, [2804](#)
- removeZeroes, [2804](#)
- retract, [2804](#)
- retractIfCan, [2804](#)
- sample, [2804](#)
- sec, [2804](#)
- sech, [2804](#)
- series, [2804](#)
- sign, [2804](#)
- sin, [2804](#)
- sinh, [2804](#)
- sizeLess?, [2804](#)
- solveLinearPolynomialEquation, [2804](#)
- sqrt, [2804](#)
- squareFree, [2804](#)
- squareFreePart, [2804](#)
- squareFreePolynomial, [2804](#)
- subtractIfCan, [2804](#)
- tan, [2804](#)
- tanh, [2804](#)
- taylor, [2804](#)
- taylorIfCan, [2804](#)
- taylorRep, [2804](#)
- terms, [2804](#)
- truncate, [2804](#)
- unit?, [2804](#)
- unitCanonical, [2804](#)
- unitNormal, [2804](#)
- variable, [2804](#)
- variables, [2804](#)
- wholePart, [2804](#)
- zero?, [2804](#)
- unary?
 - BOP, [274](#)
- uncorrelated?
 - SD, [2572](#)
- unexpand
 - XPOLY, [2998](#)
 - XPOLY, [3013](#)
- union
 - CCLASS, [382](#)
 - MSET, [1670](#)
 - PATRES, [1940](#)
 - SET, [2372](#)
- UNISEG, [2896](#)
 - ?..?, [2896](#)
 - ?=?, [2896](#)
 - ?SEGMENT, [2896](#)
 - ?~=?, [2896](#)
 - BY, [2896](#)
 - coerce, [2896](#)
 - convert, [2896](#)
 - expand, [2896](#)
 - hash, [2896](#)
 - hasHi, [2896](#)
 - hi, [2896](#)
 - high, [2896](#)
 - incr, [2896](#)
 - latex, [2896](#)
 - lo, [2896](#)
 - low, [2896](#)

- map, 2896
- segment, 2896
- unit
 - ALGSC, 15
 - DROPT, 608
 - FR, 786
 - GCNAALG, 1065
 - JORDAN, 223
 - LIE, 229
 - LSQM, 1456
- unit?
 - ALGFF, 28
 - AN, 35
 - BINARY, 292
 - BPADIC, 258
 - BPADICRT, 263
 - COMPLEX, 419
 - CONTFRAC, 445
 - DECIMAL, 465
 - DFLOAT, 587
 - DIRRING, 564
 - DMP, 572
 - DSMP, 542
 - EMR, 702
 - EXPEXPAN, 712
 - EXPR, 724
 - EXPUPXS, 740
 - FF, 820
 - FFCG, 825
 - FFCGP, 835
 - FFCGX, 830
 - FFNB, 860
 - FFNBP, 872
 - FFNBX, 866
 - FFP, 851
 - FFX, 846
 - FLOAT, 908
 - FR, 786
 - FRAC, 986
 - GDMP, 1052
 - GSERIES, 1090
 - HACKPI, 1978
 - HDMP, 1180
 - HEXADEC, 1143
 - IAN, 1276
 - IFF, 1284
 - INT, 1362
 - INTRVL, 1384
 - IPADIC, 1294
 - IPF, 1303
 - ISUPS, 1311
 - ITAYLOR, 1338
 - LAUPOL, 1422
 - MCMPLX, 1543
 - MFLOAT, 1548
 - MINT, 1557
 - MODFIELD, 1638
 - MODMON, 1632
 - MPOLY, 1682
 - MYEXPR, 1688
 - MYUP, 1695
 - NSDPS, 1703
 - NSMP, 1714
 - NSUP, 1729
 - ODPOL, 1852
 - ODR, 1858
 - PACOFF, 2136
 - PACRAT, 2146
 - PADIC, 1881
 - PADICRAT, 1886
 - PADICRC, 1891
 - PF, 2105
 - PFR, 1914
 - POLY, 2078
 - PR, 2092
 - RADFF, 2194
 - RADIX, 2206
 - RECLOS, 2237
 - ROMAN, 2326
 - SAE, 2400
 - SDPOL, 2386
 - SINT, 2412
 - SMP, 2423
 - SMTS, 2441
 - SULS, 2457
 - SUP, 2467
 - SUPEXP, 2481
 - SUPXS, 2487
 - SUTS, 2497
 - SYMPOLY, 2654
 - TS, 2671
 - UFPS, 2789

- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- unitCanonical
 - ALGFF, [28](#)
 - AN, [35](#)
 - BINARY, [292](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - DIRRING, [564](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EMR, [702](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FLOAT, [908](#)
 - FR, [786](#)
 - FRAC, [986](#)
 - GDMP, [1052](#)
 - GSERIES, [1090](#)
 - HACKPI, [1978](#)
 - HDMP, [1180](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - IFF, [1284](#)
 - INT, [1362](#)
 - INTRVL, [1384](#)
 - IPADIC, [1294](#)
 - IPF, [1303](#)
 - ISUPS, [1311](#)
 - ITAYLOR, [1338](#)
 - LAUPOL, [1422](#)
 - MCMPLX, [1543](#)
 - MFLOAT, [1548](#)
 - MINT, [1557](#)
 - MODFIELD, [1638](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYEXPR, [1688](#)
 - MYUP, [1695](#)
 - NSDPS, [1703](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - ODR, [1858](#)
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
 - PADIC, [1881](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - PF, [2105](#)
 - PFR, [1914](#)
 - POLY, [2078](#)
 - PR, [2092](#)
 - RADFF, [2194](#)
 - RADIX, [2206](#)
 - RECLOS, [2237](#)
 - ROMAN, [2326](#)
 - SAE, [2400](#)
 - SDPOL, [2386](#)
 - SINT, [2412](#)
 - SMP, [2423](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - SYMPOLY, [2654](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)

- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- unitNormal
 - ALGFF, [28](#)
 - AN, [35](#)
 - BINARY, [292](#)
 - BPADIC, [258](#)
 - BPADICRT, [263](#)
 - COMPLEX, [419](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - DIRRING, [564](#)
 - DMP, [572](#)
 - DSMP, [542](#)
 - EMR, [702](#)
 - EXPEXPAN, [712](#)
 - EXPR, [724](#)
 - EXPUPXS, [740](#)
 - FF, [820](#)
 - FFCG, [825](#)
 - FFCGP, [835](#)
 - FFCGX, [830](#)
 - FFNB, [860](#)
 - FFNBP, [872](#)
 - FFNBX, [866](#)
 - FFP, [851](#)
 - FFX, [846](#)
 - FLOAT, [908](#)
 - FR, [786](#)
 - FRAC, [986](#)
 - GDMP, [1052](#)
 - GSERIES, [1090](#)
 - HACKPI, [1978](#)
 - HDMP, [1180](#)
 - HEXADEC, [1143](#)
 - IAN, [1276](#)
 - IFF, [1284](#)
 - INT, [1362](#)
 - INTRVL, [1384](#)
 - IPADIC, [1294](#)
 - IPF, [1303](#)
 - ISUPS, [1311](#)
 - ITAYLOR, [1338](#)
 - LAUPOL, [1422](#)
 - MCMPPLX, [1543](#)
 - MFLOAT, [1548](#)
 - MINT, [1557](#)
 - MODFIELD, [1638](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYEXPR, [1688](#)
 - MYUP, [1695](#)
 - NSDPS, [1703](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - ODR, [1858](#)
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
 - PADIC, [1881](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - PF, [2105](#)
 - PFR, [1914](#)
 - POLY, [2078](#)
 - PR, [2092](#)
 - RADFF, [2194](#)
 - RADIX, [2206](#)
 - RECLOS, [2237](#)
 - ROMAN, [2326](#)
 - SAE, [2400](#)
 - SDPOL, [2386](#)
 - SINT, [2412](#)
 - SMP, [2423](#)
 - SMTS, [2441](#)
 - SULS, [2457](#)
 - SUP, [2467](#)
 - SUPEXP, [2481](#)
 - SUPXS, [2487](#)
 - SUTS, [2497](#)
 - SYMPOLY, [2654](#)
 - TS, [2671](#)
 - UFPS, [2789](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
 - UP, [2828](#)
 - UPXS, [2834](#)

- UPXSCONS, [2842](#)
- UPXSSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- unitNormalize
 - FR, [786](#)
- units
 - GRIMAGE, [1094](#)
 - VIEW2d, [2770](#)
- unitVector
 - DIRPROD, [547](#)
 - DPMM, [553](#)
 - DPMO, [557](#)
 - GMODPOL, [1059](#)
 - HDP, [1173](#)
 - ODP, [1816](#)
 - SHDP, [2509](#)
- univariate
 - DMP, [572](#)
 - DSMP, [542](#)
 - EXPR, [724](#)
 - GDMP, [1052](#)
 - HDMP, [1180](#)
 - MODMON, [1632](#)
 - MPOLY, [1682](#)
 - MYEXPR, [1688](#)
 - MYUP, [1695](#)
 - NSMP, [1714](#)
 - NSUP, [1729](#)
 - ODPOL, [1852](#)
 - POLY, [2078](#)
 - SDPOL, [2386](#)
 - SMP, [2423](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - UP, [2828](#)
- UnivariateFormalPowerSeries, [2788](#)
- UnivariateLaurentSeries, [2795](#)
- UnivariateLaurentSeriesConstructor, [2803](#)
- UnivariatePolynomial, [2827](#)
- univariatePolynomial
 - SUTS, [2497](#)
 - UFPS, [2789](#)
 - UTS, [2877](#)
 - UTSZ, [2887](#)
- UnivariatePuisseuxSeries, [2833](#)
- UnivariatePuisseuxSeriesConstructor, [2841](#)
- UnivariatePuisseuxSeriesWithExponentialSingularity, [2852](#)
- UnivariateSkewPolynomial, [2872](#)
- UnivariateTaylorSeries, [2876](#)
- UnivariateTaylorSeriesCZero, [2886](#)
- UniversalSegment, [2896](#)
- universe
 - CCLASS, [382](#)
 - SET, [2372](#)
- unmakeSUP
 - MODMON, [1632](#)
 - MYUP, [1695](#)
 - NSUP, [1729](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - UP, [2828](#)
- unparse
 - INFORM, [1343](#)
- unravel
 - CARTEN, [356](#)
- UnVectorise
 - MODMON, [1632](#)
- UP, [2827](#)
- ?, [2828](#)
- ?<?, [2828](#)
- ?<=?, [2828](#)
- ?>?, [2828](#)
- ?>=?, [2828](#)
- ?**?, [2828](#)
- ?*?, [2828](#)
- ?+?, [2828](#)
- ?-?, [2828](#)
- ?., [2828](#)
- ?/?, [2828](#)
- ?=?, [2828](#)
- ?^?, [2828](#)
- ?~=?, [2828](#)
- ?quo?, [2828](#)
- ?rem?, [2828](#)
- 0, [2828](#)
- 1, [2828](#)
- associates?, [2828](#)
- binomThmExpt, [2828](#)
- characteristic, [2828](#)
- charthRoot, [2828](#)

- coefficient, 2828
- coefficients, 2828
- coerce, 2828
- composite, 2828
- conditionP, 2828
- content, 2828
- convert, 2828
- D, 2828
- degree, 2828
- differentiate, 2828
- discriminant, 2828
- divide, 2828
- divideExponents, 2828
- elt, 2828
- euclideanSize, 2828
- eval, 2828
- expressIdealMember, 2828
- exquo, 2828
- extendedEuclidean, 2828
- factor, 2828
- factorPolynomial, 2828
- factorSquareFreePolynomial, 2828
- fmeq, 2828
- gcd, 2828
- gcdPolynomial, 2828
- ground, 2828
- ground?, 2828
- hash, 2828
- init, 2828
- integrate, 2828
- isExpt, 2828
- isPlus, 2828
- isTimes, 2828
- karatsubaDivide, 2828
- latex, 2828
- lcm, 2828
- leadingCoefficient, 2828
- leadingMonomial, 2828
- mainVariable, 2828
- makeSUP, 2828
- map, 2828
- mapExponents, 2828
- max, 2828
- min, 2828
- minimumDegree, 2828
- monicDivide, 2828
- monomial, 2828
- monomial?, 2828
- monomials, 2828
- multiEuclidean, 2828
- multiplyExponents, 2828
- multivariate, 2828
- nextItem, 2828
- numberOfMonomials, 2828
- one?, 2828
- order, 2828
- patternMatch, 2828
- pomopo, 2828
- prime?, 2828
- primitiveMonomials, 2828
- primitivePart, 2828
- principalIdeal, 2828
- pseudoDivide, 2828
- pseudoQuotient, 2828
- pseudoRemainder, 2828
- recip, 2828
- reducedSystem, 2828
- reductum, 2828
- resultant, 2828
- retract, 2828
- retractIfCan, 2828
- sample, 2828
- separate, 2828
- shiftLeft, 2828
- shiftRight, 2828
- sizeLess?, 2828
- solveLinearPolynomialEquation, 2828
- squareFree, 2828
- squareFreePart, 2828
- squareFreePolynomial, 2828
- subResultantGcd, 2828
- subtractIfCan, 2828
- totalDegree, 2828
- unit?, 2828
- unitCanonical, 2828
- unitNormal, 2828
- univariate, 2828
- unmakeSUP, 2828
- variables, 2828
- vectorise, 2828
- zero?, 2828
- update

- VIEW2d, 2770
- upperCase
 - CCLASS, 382
 - CHAR, 373
 - ISTRING, 1249
 - STRING, 2607
- upperCase?
 - CHAR, 373
- UPXS, 2833
- , 2834
- ?**, 2834
- ?*, 2834
- ?+, 2834
- ?-, 2834
- ?., 2834
- ?/?, 2834
- ?=?, 2834
- ?^?, 2834
- ?~=?, 2834
- ?quo?, 2834
- ?rem?, 2834
- 0, 2834
- 1, 2834
- acos, 2834
- acosh, 2834
- acot, 2834
- acoth, 2834
- acsc, 2834
- acsch, 2834
- approximate, 2834
- asec, 2834
- asech, 2834
- asin, 2834
- asinh, 2834
- associates?, 2834
- atan, 2834
- atanh, 2834
- center, 2834
- characteristic, 2834
- charthRoot, 2834
- coefficient, 2834
- coerce, 2834
- complete, 2834
- cos, 2834
- cosh, 2834
- cot, 2834
- coth, 2834
- csc, 2834
- csch, 2834
- D, 2834
- degree, 2834
- differentiate, 2834
- divide, 2834
- euclideanSize, 2834
- eval, 2834
- exp, 2834
- expressIdealMember, 2834
- exquo, 2834
- extend, 2834
- extendedEuclidean, 2834
- factor, 2834
- gcd, 2834
- gcdPolynomial, 2834
- hash, 2834
- integrate, 2834
- inv, 2834
- latex, 2834
- laurent, 2834
- laurentIfCan, 2834
- laurentRep, 2834
- lcm, 2834
- leadingCoefficient, 2834
- leadingMonomial, 2834
- log, 2834
- map, 2834
- monomial, 2834
- monomial?, 2834
- multiEuclidean, 2834
- multiplyExponents, 2834
- nthRoot, 2834
- one?, 2834
- order, 2834
- pi, 2834
- pole?, 2834
- prime?, 2834
- principalIdeal, 2834
- puiseux, 2834
- rationalPower, 2834
- recip, 2834
- reductum, 2834
- retract, 2834
- retractIfCan, 2834

- sample, [2834](#)
- sec, [2834](#)
- sech, [2834](#)
- series, [2834](#)
- sin, [2834](#)
- sinh, [2834](#)
- sizeLess?, [2834](#)
- sqrt, [2834](#)
- squareFree, [2834](#)
- squareFreePart, [2834](#)
- subtractIfCan, [2834](#)
- tan, [2834](#)
- tanh, [2834](#)
- terms, [2834](#)
- truncate, [2834](#)
- unit?, [2834](#)
- unitCanonical, [2834](#)
- unitNormal, [2834](#)
- variable, [2834](#)
- variables, [2834](#)
- zero?, [2834](#)
- UPXSCONS, [2841](#)
- ?, [2842](#)
- ?**?, [2842](#)
- ?*?, [2842](#)
- ?+?, [2842](#)
- ?-?, [2842](#)
- ?., [2842](#)
- ?/?, [2842](#)
- ?=?, [2842](#)
- ?^?, [2842](#)
- ?~=?, [2842](#)
- ?quo?, [2842](#)
- ?rem?, [2842](#)
- 0, [2842](#)
- 1, [2842](#)
- acos, [2842](#)
- acosh, [2842](#)
- acot, [2842](#)
- acoth, [2842](#)
- acsc, [2842](#)
- acsch, [2842](#)
- approximate, [2842](#)
- asec, [2842](#)
- asech, [2842](#)
- asin, [2842](#)
- asinh, [2842](#)
- associates?, [2842](#)
- atan, [2842](#)
- atanh, [2842](#)
- center, [2842](#)
- characteristic, [2842](#)
- charthRoot, [2842](#)
- coefficient, [2842](#)
- coerce, [2842](#)
- complete, [2842](#)
- cos, [2842](#)
- cosh, [2842](#)
- cot, [2842](#)
- coth, [2842](#)
- csc, [2842](#)
- csch, [2842](#)
- D, [2842](#)
- degree, [2842](#)
- differentiate, [2842](#)
- divide, [2842](#)
- euclideanSize, [2842](#)
- eval, [2842](#)
- exp, [2842](#)
- expressIdealMember, [2842](#)
- exquo, [2842](#)
- extend, [2842](#)
- extendedEuclidean, [2842](#)
- factor, [2842](#)
- gcd, [2842](#)
- gcdPolynomial, [2842](#)
- hash, [2842](#)
- integrate, [2842](#)
- inv, [2842](#)
- latex, [2842](#)
- laurent, [2842](#)
- laurentIfCan, [2842](#)
- laurentRep, [2842](#)
- lcm, [2842](#)
- leadingCoefficient, [2842](#)
- leadingMonomial, [2842](#)
- log, [2842](#)
- map, [2842](#)
- monomial, [2842](#)
- monomial?, [2842](#)
- multiEuclidean, [2842](#)
- multiplyExponents, [2842](#)

- nthRoot, [2842](#)
- one?, [2842](#)
- order, [2842](#)
- pi, [2842](#)
- pole?, [2842](#)
- prime?, [2842](#)
- principalIdeal, [2842](#)
- puiseux, [2842](#)
- rationalPower, [2842](#)
- recip, [2842](#)
- reductum, [2842](#)
- retract, [2842](#)
- retractIfCan, [2842](#)
- sample, [2842](#)
- sec, [2842](#)
- sech, [2842](#)
- series, [2842](#)
- sin, [2842](#)
- sinh, [2842](#)
- sizeLess?, [2842](#)
- sqrt, [2842](#)
- squareFree, [2842](#)
- squareFreePart, [2842](#)
- subtractIfCan, [2842](#)
- tan, [2842](#)
- tanh, [2842](#)
- terms, [2842](#)
- truncate, [2842](#)
- unit?, [2842](#)
- unitCanonical, [2842](#)
- unitNormal, [2842](#)
- variable, [2842](#)
- variables, [2842](#)
- zero?, [2842](#)
- UPXSSING, [2852](#)
- ?, [2852](#)
- ***?, [2852](#)
- *?*, [2852](#)
- ?+?, [2852](#)
- ?-?, [2852](#)
- ?/? , [2852](#)
- ?=?, [2852](#)
- ?^?, [2852](#)
- ?~=?, [2852](#)
- 0, [2852](#)
- 1, [2852](#)
- associates?, [2852](#)
- binomThmExpt, [2852](#)
- characteristic, [2852](#)
- charthRoot, [2852](#)
- coefficient, [2852](#)
- coefficients, [2852](#)
- coerce, [2852](#)
- content, [2852](#)
- degree, [2852](#)
- dominantTerm, [2852](#)
- exquo, [2852](#)
- ground, [2852](#)
- ground?, [2852](#)
- hash, [2852](#)
- latex, [2852](#)
- leadingCoefficient, [2852](#)
- leadingMonomial, [2852](#)
- limitPlus, [2852](#)
- map, [2852](#)
- mapExponents, [2852](#)
- minimumDegree, [2852](#)
- monomial, [2852](#)
- monomial?, [2852](#)
- numberOfMonomials, [2852](#)
- one?, [2852](#)
- pomopo, [2852](#)
- primitivePart, [2852](#)
- recip, [2852](#)
- reductum, [2852](#)
- retract, [2852](#)
- retractIfCan, [2852](#)
- sample, [2852](#)
- subtractIfCan, [2852](#)
- unit?, [2852](#)
- unitCanonical, [2852](#)
- unitNormal, [2852](#)
- zero?, [2852](#)
- useNagFunctions
- FEXPR, [947](#)
- UTS, [2876](#)
- ?, [2877](#)
- ***?, [2877](#)
- *?*, [2877](#)
- ?+?, [2877](#)
- ?-?, [2877](#)
- ?..?, [2877](#)

- [?=?, 2877](#)
- [?^?, 2877](#)
- [?~=?, 2877](#)
- [0, 2877](#)
- [1, 2877](#)
- [acos, 2877](#)
- [acosh, 2877](#)
- [acot, 2877](#)
- [acoth, 2877](#)
- [acsc, 2877](#)
- [acsch, 2877](#)
- [approximate, 2877](#)
- [asec, 2877](#)
- [asech, 2877](#)
- [asin, 2877](#)
- [asinh, 2877](#)
- [associates?, 2877](#)
- [atan, 2877](#)
- [atanh, 2877](#)
- [center, 2877](#)
- [characteristic, 2877](#)
- [charthRoot, 2877](#)
- [coefficient, 2877](#)
- [coefficients, 2877](#)
- [coerce, 2877](#)
- [complete, 2877](#)
- [cos, 2877](#)
- [cosh, 2877](#)
- [cot, 2877](#)
- [coth, 2877](#)
- [csc, 2877](#)
- [csch, 2877](#)
- [D, 2877](#)
- [degree, 2877](#)
- [differentiate, 2877](#)
- [eval, 2877](#)
- [evenlambert, 2877](#)
- [exp, 2877](#)
- [exquo, 2877](#)
- [extend, 2877](#)
- [generalLambert, 2877](#)
- [hash, 2877](#)
- [integrate, 2877](#)
- [invmultisect, 2877](#)
- [lagrange, 2877](#)
- [lambert, 2877](#)
- [latex, 2877](#)
- [leadingCoefficient, 2877](#)
- [leadingMonomial, 2877](#)
- [log, 2877](#)
- [map, 2877](#)
- [monomial, 2877](#)
- [monomial?, 2877](#)
- [multiplyCoefficients, 2877](#)
- [multiplyExponents, 2877](#)
- [multisect, 2877](#)
- [nthRoot, 2877](#)
- [oddlambert, 2877](#)
- [one?, 2877](#)
- [order, 2877](#)
- [pi, 2877](#)
- [pole?, 2877](#)
- [polynomial, 2877](#)
- [quoByVar, 2877](#)
- [recip, 2877](#)
- [reductum, 2877](#)
- [revert, 2877](#)
- [sample, 2877](#)
- [sec, 2877](#)
- [sech, 2877](#)
- [series, 2877](#)
- [sin, 2877](#)
- [sinh, 2877](#)
- [sqrt, 2877](#)
- [subtractIfCan, 2877](#)
- [tan, 2877](#)
- [tanh, 2877](#)
- [terms, 2877](#)
- [truncate, 2877](#)
- [unit?, 2877](#)
- [unitCanonical, 2877](#)
- [unitNormal, 2877](#)
- [univariatePolynomial, 2877](#)
- [variable, 2877](#)
- [variables, 2877](#)
- [zero?, 2877](#)
- [UTSZ, 2886](#)
- [-?, 2887](#)
- [?**?, 2887](#)
- [?*?, 2887](#)
- [?+?, 2887](#)
- [?-?, 2887](#)

- ?.?, 2887
- ?/?, 2887
- ?=?, 2887
- ?^?, 2887
- ?~=?, 2887
- 0, 2887
- 1, 2887
- acos, 2887
- acosh, 2887
- acot, 2887
- acoth, 2887
- acsc, 2887
- acsch, 2887
- approximate, 2887
- asec, 2887
- asech, 2887
- asin, 2887
- asinh, 2887
- associates?, 2887
- atan, 2887
- atanh, 2887
- center, 2887
- characteristic, 2887
- charthRoot, 2887
- coefficient, 2887
- coefficients, 2887
- coerce, 2887
- complete, 2887
- cos, 2887
- cosh, 2887
- cot, 2887
- coth, 2887
- csc, 2887
- csch, 2887
- D, 2887
- degree, 2887
- differentiate, 2887
- eval, 2887
- evenlambert, 2887
- exp, 2887
- exquo, 2887
- extend, 2887
- generalLambert, 2887
- hash, 2887
- integrate, 2887
- invmultisect, 2887
- lagrange, 2887
- lambert, 2887
- latex, 2887
- leadingCoefficient, 2887
- leadingMonomial, 2887
- log, 2887
- map, 2887
- monomial, 2887
- monomial?, 2887
- multiplyCoefficients, 2887
- multiplyExponents, 2887
- multisect, 2887
- nthRoot, 2887
- oddlambert, 2887
- one?, 2887
- order, 2887
- pi, 2887
- pole?, 2887
- polynomial, 2887
- quoByVar, 2887
- recip, 2887
- reductum, 2887
- revert, 2887
- sample, 2887
- sec, 2887
- sech, 2887
- series, 2887
- sin, 2887
- sinh, 2887
- sqrt, 2887
- subtractIfCan, 2887
- tan, 2887
- tanh, 2887
- terms, 2887
- truncate, 2887
- unit?, 2887
- unitCanonical, 2887
- unitNormal, 2887
- univariatePolynomial, 2887
- variable, 2887
- variables, 2887
- zero?, 2887
- value
 - ALIST, 235
 - BBTREE, 253

- BSTREE, 302
- BTOURN, 306
- BTREE, 310
- DLIST, 460
- DSTREE, 535
- ILIST, 1231
- LIST, 1504
- NSDPS, 1703
- OSI, 1864
- PENDTREE, 1945
- QEQUAT, 2169
- SPLNODE, 2512
- SPLTREE, 2518
- STREAM, 2582
- TREE, 2741
- var1Steps
 - DROPT, 608
- var2Steps
 - DROPT, 608
- VARIABLE, 2936
 - ?=?, 2936
 - ?~=?, 2936
 - coerce, 2936
 - hash, 2936
 - latex, 2936
 - variable, 2936
- Variable, 2936
- variable
 - EXPUPXS, 740
 - GSERIES, 1090
 - ISUPS, 1311
 - NSDPS, 1703
 - ODVAR, 1855
 - OVAR, 1836
 - QEQUAT, 2169
 - SDVAR, 2389
 - SEGBIND, 2364
 - SULS, 2457
 - SUPXS, 2487
 - SUTS, 2497
 - UFPS, 2789
 - ULS, 2796
 - ULSCONS, 2804
 - UPXS, 2834
 - UPXSCONS, 2842
 - UTS, 2877
 - UTSZ, 2887
 - VARIABLE, 2936
- variableName
 - GOPT, 1104
 - GOPT0, 1110
- variables
 - DMP, 572
 - DSMP, 542
 - EXPR, 724
 - EXPUPXS, 740
 - FEXPR, 947
 - GDMP, 1052
 - GPOLSET, 1074
 - GSERIES, 1090
 - GTSET, 1083
 - HDMP, 1180
 - ISUPS, 1311
 - MODMON, 1632
 - MPOLY, 1682
 - MYEXPR, 1688
 - MYUP, 1695
 - NSDPS, 1703
 - NSMP, 1714
 - NSUP, 1729
 - ODPOL, 1852
 - PATTERN, 1928
 - POLY, 2078
 - REGSET, 2286
 - RGCHAIN, 2255
 - SDPOL, 2386
 - SMP, 2423
 - SMTS, 2441
 - SREGSET, 2535
 - SULS, 2457
 - SUP, 2467
 - SUPEXPR, 2481
 - SUPXS, 2487
 - SUTS, 2497
 - TS, 2671
 - UFPS, 2789
 - ULS, 2796
 - ULSCONS, 2804
 - UP, 2828
 - UPXS, 2834
 - UPXSCONS, 2842
 - UTS, 2877

- UTSZ, 2887
- WUTSET, 2959
- varList
 - LEXP, 1435
 - LPOLY, 1447
 - LWORD, 1532
 - MAGMA, 1565
 - OFMONOID, 1829
 - PBWLb, 2054
 - XDPOLY, 2967
 - XPBWPOLYL, 2987
 - XPOLY, 2998
 - XRPOLY, 3013
- vconcat
 - OUTFORM, 1868
- VECTOR, 2941
 - , 2942
 - ?<?, 2942
 - ?<=?, 2942
 - ?>?, 2942
 - ?>=?, 2942
 - ?*?, 2942
 - ?+?, 2942
 - ?-?, 2942
 - ?., 2942
 - ?=?, 2942
 - ?~=?, 2942
 - #?, 2942
 - any?, 2942
 - coerce, 2942
 - concat, 2942
 - construct, 2942
 - convert, 2942
 - copy, 2942
 - copyInto, 2942
 - count, 2942
 - cross, 2942
 - delete, 2942
 - dot, 2942
 - elt, 2942
 - empty, 2942
 - empty?, 2942
 - entries, 2942
 - entry?, 2942
 - eq?, 2942
 - eval, 2942
 - every?, 2942
 - fill, 2942
 - find, 2942
 - first, 2942
 - hash, 2942
 - index?, 2942
 - indices, 2942
 - insert, 2942
 - latex, 2942
 - length, 2942
 - less?, 2942
 - magnitude, 2942
 - map, 2942
 - max, 2942
 - maxIndex, 2942
 - member?, 2942
 - members, 2942
 - merge, 2942
 - min, 2942
 - minIndex, 2942
 - more?, 2942
 - new, 2942
 - outerProduct, 2942
 - parts, 2942
 - position, 2942
 - qelt, 2942
 - qsetelt, 2942
 - reduce, 2942
 - remove, 2942
 - removeDuplicates, 2942
 - reverse, 2942
 - sample, 2942
 - select, 2942
 - setelt, 2942
 - size?, 2942
 - sort, 2942
 - sorted?, 2942
 - swap, 2942
 - vector, 2942
 - zero, 2942
- Vector, 2941
- vector
 - CDFVEC, 432
 - VECTOR, 2942
- Vectorise
 - MODMON, 1632

- vectorise
 - MODMON, [1632](#)
 - MYUP, [1695](#)
 - NSUP, [1729](#)
 - PACOFF, [2136](#)
 - PACRAT, [2146](#)
 - SUP, [2467](#)
 - SUPEXPR, [2481](#)
 - UP, [2828](#)
- vertConcat
 - CDFMAT, [426](#)
 - DFMAT, [599](#)
 - DHMATRIX, [491](#)
 - IMATRIX, [1239](#)
 - MATRIX, [1623](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
- VIEW2D, [2769](#)
- VIEW2d
 - ?=?, [2770](#)
 - ?~=?, [2770](#)
 - axes, [2770](#)
 - close, [2770](#)
 - coerce, [2770](#)
 - connect, [2770](#)
 - controlPanel, [2770](#)
 - dimensions, [2770](#)
 - getGraph, [2770](#)
 - getPickedPoints, [2770](#)
 - graphs, [2770](#)
 - graphState, [2770](#)
 - graphStates, [2770](#)
 - hash, [2770](#)
 - key, [2770](#)
 - latex, [2770](#)
 - makeViewport2D, [2770](#)
 - move, [2770](#)
 - options, [2770](#)
 - points, [2770](#)
 - putGraph, [2770](#)
 - region, [2770](#)
 - reset, [2770](#)
 - resize, [2770](#)
 - scale, [2770](#)
 - show, [2770](#)
 - title, [2770](#)
 - translate, [2770](#)
 - units, [2770](#)
 - update, [2770](#)
 - viewport2D, [2770](#)
 - write, [2770](#)
- VIEW3D, [2710](#)
 - ?=?, [2710](#)
 - ?~=?, [2710](#)
 - axes, [2710](#)
 - clipSurface, [2710](#)
 - close, [2710](#)
 - coerce, [2710](#)
 - colorDef, [2710](#)
 - controlPanel, [2710](#)
 - diagonals, [2710](#)
 - dimensions, [2710](#)
 - drawStyle, [2710](#)
 - eyeDistance, [2710](#)
 - hash, [2710](#)
 - hitherPlane, [2710](#)
 - intensity, [2710](#)
 - key, [2710](#)
 - latex, [2710](#)
 - lighting, [2710](#)
 - makeViewport3D, [2710](#)
 - modifyPointData, [2710](#)
 - move, [2710](#)
 - options, [2710](#)
 - outlineRender, [2710](#)
 - perspective, [2710](#)
 - reset, [2710](#)
 - resize, [2710](#)
 - rotate, [2710](#)
 - showClipRegion, [2710](#)
 - showRegion, [2710](#)
 - subspace, [2710](#)
 - title, [2710](#)
 - translate, [2710](#)
 - viewDeltaXDefault, [2710](#)
 - viewDeltaYDefault, [2710](#)
 - viewPhiDefault, [2710](#)
 - viewpoint, [2710](#)
 - viewport3D, [2710](#)
 - viewThetaDefault, [2710](#)
 - viewZoomDefault, [2710](#)

- write, [2710](#)
 - zoom, [2710](#)
- viewDeltaXDefault
 - VIEW3D, [2710](#)
- viewDeltaYDefault
 - VIEW3D, [2710](#)
- viewPhiDefault
 - VIEW3D, [2710](#)
- viewpoint
 - DROPT, [608](#)
 - VIEW3D, [2710](#)
- viewport2D
 - VIEW2d, [2770](#)
- viewport3D
 - VIEW3D, [2710](#)
- viewThetaDefault
 - VIEW3D, [2710](#)
- viewZoomDefault
 - VIEW3D, [2710](#)
- VOID, [2945](#)
 - coerce, [2945](#)
 - void, [2945](#)
- Void, [2945](#)
- void
 - VOID, [2945](#)
- vspace
 - OUTFORM, [1868](#)
- weight
 - BOP, [274](#)
 - DSMP, [542](#)
 - ODPOL, [1852](#)
 - ODVAR, [1855](#)
 - SDPOL, [2386](#)
 - SDVAR, [2389](#)
- WeightedPolynomials, [2949](#)
- weights
 - DSMP, [542](#)
 - ODPOL, [1852](#)
 - SDPOL, [2386](#)
- whatInfinity
 - ORDCOMP, [1810](#)
- whileLoop
 - FC, [931](#)
- wholePart
 - BINARY, [292](#)
 - BPADICRT, [263](#)
 - CONTFRAC, [445](#)
 - DECIMAL, [465](#)
 - DFLOAT, [587](#)
 - EXPEXPAN, [712](#)
 - FLOAT, [908](#)
 - FRAC, [986](#)
 - HEXADEC, [1143](#)
 - MFLOAT, [1548](#)
 - PADICRAT, [1886](#)
 - PADICRC, [1891](#)
 - PFR, [1914](#)
 - RADIX, [2206](#)
 - SULS, [2457](#)
 - ULS, [2796](#)
 - ULSCONS, [2804](#)
- wholeRadix
 - RADIX, [2206](#)
- wholeRagits
 - RADIX, [2206](#)
- width
 - INTRVL, [1384](#)
 - OUTFORM, [1868](#)
- withPredicates
 - PATTERN, [1928](#)
- wordInGenerators
 - PERMGRP, [1960](#)
- wordInStrongGenerators
 - PERMGRP, [1960](#)
- wordsForStrongGenerators
 - PERMGRP, [1960](#)
- WP, [2949](#)
 - , [2949](#)
 - ?**, [2949](#)
 - ?*, [2949](#)
 - ?+?, [2949](#)
 - ?-?, [2949](#)
 - ?/?, [2949](#)
 - ?=?, [2949](#)
 - ?^?, [2949](#)
 - ?~=?, [2949](#)
 - 0, [2949](#)
 - 1, [2949](#)
 - changeWeightLevel, [2949](#)
 - characteristic, [2949](#)
 - coerce, [2949](#)

- hash, 2949
- latex, 2949
- one?, 2949
- recip, 2949
- sample, 2949
- subtractIfCan, 2949
- zero?, 2949
- writable?
 - FNAME, 810
- write
 - VIEW2d, 2770
 - VIEW3D, 2710
- WUTSET, 2958
 - ?=?, 2959
 - ?~=?, 2959
 - #?, 2959
 - algebraic?, 2959
 - algebraicVariables, 2959
 - any?, 2959
 - autoReduced?, 2959
 - basicSet, 2959
 - characteristicSerie, 2959
 - characteristicSet, 2959
 - coerce, 2959
 - coHeight, 2959
 - collect, 2959
 - collectQuasiMonic, 2959
 - collectUnder, 2959
 - collectUpper, 2959
 - construct, 2959
 - convert, 2959
 - copy, 2959
 - count, 2959
 - degree, 2959
 - empty, 2959
 - empty?, 2959
 - eq?, 2959
 - eval, 2959
 - every?, 2959
 - extend, 2959
 - extendIfCan, 2959
 - find, 2959
 - first, 2959
 - hash, 2959
 - headReduce, 2959
 - headReduced?, 2959
 - headRemainder, 2959
 - infRittWu?, 2959
 - initiallyReduce, 2959
 - initiallyReduced?, 2959
 - initials, 2959
 - last, 2959
 - latex, 2959
 - less?, 2959
 - mainVariable?, 2959
 - mainVariables, 2959
 - map, 2959
 - medialSet, 2959
 - member?, 2959
 - members, 2959
 - more?, 2959
 - mvar, 2959
 - normalized?, 2959
 - parts, 2959
 - quasiComponent, 2959
 - reduce, 2959
 - reduceByQuasiMonic, 2959
 - reduced?, 2959
 - remainder, 2959
 - remove, 2959
 - removeDuplicates, 2959
 - removeZero, 2959
 - rest, 2959
 - retract, 2959
 - retractIfCan, 2959
 - rewriteIdealWithHeadRemainder, 2959
 - rewriteIdealWithRemainder, 2959
 - rewriteSetWithReduction, 2959
 - roughBase?, 2959
 - roughEqualIdeals?, 2959
 - roughSubIdeal?, 2959
 - roughUnitIdeal?, 2959
 - sample, 2959
 - select, 2959
 - size?, 2959
 - sort, 2959
 - stronglyReduce, 2959
 - stronglyReduced?, 2959
 - triangular?, 2959
 - trivialIdeal?, 2959
 - variables, 2959
 - zeroSetSplit, 2959

- zeroSetSplitIntoTriangularSystems, 2959
- WuWenTsunTriangularSet, 2958
- XDistributedPolynomial, 2967
- XDPOLY, 2967
 - ?, 2967
 - ?**?, 2967
 - ?*?, 2967
 - ?+?, 2967
 - ?-?, 2967
 - ?=?, 2967
 - ?^?, 2967
 - ?~=?, 2967
 - 0, 2967
 - 1, 2967
 - characteristic, 2967
 - coef, 2967
 - coefficient, 2967
 - coefficients, 2967
 - coerce, 2967
 - constant, 2967
 - constant?, 2967
 - degree, 2967
 - hash, 2967
 - latex, 2967
 - leadingCoefficient, 2967
 - leadingMonomial, 2967
 - leadingTerm, 2967
 - listOfTerms, 2967
 - lquo, 2967
 - map, 2967
 - maxdeg, 2967
 - mindeg, 2967
 - mindegTerm, 2967
 - mirror, 2967
 - monom, 2967
 - monomial?, 2967
 - monomials, 2967
 - numberOfMonomials, 2967
 - one?, 2967
 - quasiRegular, 2967
 - quasiRegular?, 2967
 - recip, 2967
 - reductum, 2967
 - retract, 2967
 - retractIfCan, 2967
 - rquo, 2967
 - sample, 2967
 - sh, 2967
 - subtractIfCan, 2967
 - trunc, 2967
 - varList, 2967
 - zero?, 2967
- xor
 - BITS, 315
 - BOOLEAN, 322
 - IBITS, 1199
 - SINT, 2412
- XPBWPOLY, 2987
- XPBWPOLYL
 - ?, 2987
 - ?**?, 2987
 - ?*?, 2987
 - ?+?, 2987
 - ?-?, 2987
 - ?=?, 2987
 - ?^?, 2987
 - ?~=?, 2987
 - 0, 2987
 - 1, 2987
 - characteristic, 2987
 - coef, 2987
 - coefficient, 2987
 - coefficients, 2987
 - coerce, 2987
 - constant, 2987
 - constant?, 2987
 - degree, 2987
 - exp, 2987
 - hash, 2987
 - latex, 2987
 - leadingCoefficient, 2987
 - leadingMonomial, 2987
 - leadingTerm, 2987
 - LiePolyIfCan, 2987
 - listOfTerms, 2987
 - log, 2987
 - lquo, 2987
 - map, 2987
 - maxdeg, 2987
 - mindeg, 2987
 - mindegTerm, 2987

mirror, [2987](#)
 monom, [2987](#)
 monomial?, [2987](#)
 monomials, [2987](#)
 numberOfMonomials, [2987](#)
 one?, [2987](#)
 product, [2987](#)
 quasiRegular, [2987](#)
 quasiRegular?, [2987](#)
 recip, [2987](#)
 reductum, [2987](#)
 retract, [2987](#)
 retractIfCan, [2987](#)
 rquo, [2987](#)
 sample, [2987](#)
 sh, [2987](#)
 subtractIfCan, [2987](#)
 trunc, [2987](#)
 varList, [2987](#)
 zero?, [2987](#)
 XPBWPolynomial, [2987](#)
 XPOLY, [2998](#)
 -?, [2998](#)
 ?**?, [2998](#)
 ?*?, [2998](#)
 ?+?, [2998](#)
 ?-?, [2998](#)
 ?=?, [2998](#)
 ?^?, [2998](#)
 ?~=?, [2998](#)
 0, [2998](#)
 1, [2998](#)
 characteristic, [2998](#)
 coef, [2998](#)
 coerce, [2998](#)
 constant, [2998](#)
 constant?, [2998](#)
 degree, [2998](#)
 expand, [2998](#)
 hash, [2998](#)
 latex, [2998](#)
 lquo, [2998](#)
 map, [2998](#)
 maxdeg, [2998](#)
 mindeg, [2998](#)
 mindegTerm, [2998](#)

mirror, [2998](#)
 monom, [2998](#)
 monomial?, [2998](#)
 one?, [2998](#)
 quasiRegular, [2998](#)
 quasiRegular?, [2998](#)
 recip, [2998](#)
 RemainderList, [2998](#)
 retract, [2998](#)
 retractIfCan, [2998](#)
 rquo, [2998](#)
 sample, [2998](#)
 sh, [2998](#)
 subtractIfCan, [2998](#)
 trunc, [2998](#)
 unexpand, [2998](#)
 varList, [2998](#)
 zero?, [2998](#)
 XPolynomial, [2998](#)
 XPolynomialRing, [3007](#)
 XPR, [3007](#)
 -?, [3007](#)
 ?**?, [3007](#)
 ?*?, [3007](#)
 ?+?, [3007](#)
 ?-?, [3007](#)
 ?=?, [3007](#)
 ?^?, [3007](#)
 ?~=?, [3007](#)
 #?, [3007](#)
 0, [3007](#)
 1, [3007](#)
 characteristic, [3007](#)
 coef, [3007](#)
 coefficient, [3007](#)
 coefficients, [3007](#)
 coerce, [3007](#)
 constant, [3007](#)
 constant?, [3007](#)
 hash, [3007](#)
 latex, [3007](#)
 leadingCoefficient, [3007](#)
 leadingMonomial, [3007](#)
 leadingTerm, [3007](#)
 listOfTerms, [3007](#)
 map, [3007](#)

- maxdeg, [3007](#)
- mindeg, [3007](#)
- monom, [3007](#)
- monomial?, [3007](#)
- monomials, [3007](#)
- numberOfMonomials, [3007](#)
- one?, [3007](#)
- quasiRegular, [3007](#)
- quasiRegular?, [3007](#)
- recip, [3007](#)
- reductum, [3007](#)
- retract, [3007](#)
- retractIfCan, [3007](#)
- sample, [3007](#)
- subtractIfCan, [3007](#)
- zero?, [3007](#)
- xRange
 - ACPLOT, [1992](#)
 - PLOT, [2029](#)
 - PLOT3D, [2042](#)
- XRecursivePolynomial, [3013](#)
- XRPOLY, [3013](#)
 - , [3013](#)
 - ?**?, [3013](#)
 - ?*?, [3013](#)
 - ?+?, [3013](#)
 - ?-?, [3013](#)
 - ?=?, [3013](#)
 - ?^?, [3013](#)
 - ?~=?, [3013](#)
 - 0, [3013](#)
 - 1, [3013](#)
 - characteristic, [3013](#)
 - coef, [3013](#)
 - coerce, [3013](#)
 - constant, [3013](#)
 - constant?, [3013](#)
 - degree, [3013](#)
 - expand, [3013](#)
 - hash, [3013](#)
 - latex, [3013](#)
 - lquo, [3013](#)
 - map, [3013](#)
 - maxdeg, [3013](#)
 - mindeg, [3013](#)
 - mindegTerm, [3013](#)
 - mirror, [3013](#)
 - monom, [3013](#)
 - monomial?, [3013](#)
 - one?, [3013](#)
 - quasiRegular, [3013](#)
 - quasiRegular?, [3013](#)
 - recip, [3013](#)
 - RemainderList, [3013](#)
 - retract, [3013](#)
 - retractIfCan, [3013](#)
 - rquo, [3013](#)
 - sample, [3013](#)
 - sh, [3013](#)
 - subtractIfCan, [3013](#)
 - trunc, [3013](#)
 - unexpand, [3013](#)
 - varList, [3013](#)
 - zero?, [3013](#)
- yCoordinates
 - ALGFF, [28](#)
 - RADFF, [2194](#)
- yellow
 - COLOR, [407](#)
- yRange
 - ACPLOT, [1992](#)
 - PLOT, [2029](#)
 - PLOT3D, [2042](#)
- zag
 - OUTFORM, [1868](#)
- zero
 - CDFMAT, [426](#)
 - CDFVEC, [432](#)
 - DFMAT, [599](#)
 - DFVEC, [605](#)
 - DHMATRIX, [491](#)
 - IMATRIX, [1239](#)
 - IVECTOR, [1260](#)
 - MATRIX, [1623](#)
 - POINT, [2060](#)
 - U16MAT, [2906](#)
 - U32MAT, [2911](#)
 - U8MAT, [2902](#)
 - VECTOR, [2942](#)
- zero?

- ALGFF, 28
ALGSC, 15
AN, 35
ANTISYM, 40
BINARY, 292
BPADIC, 258
BPADICRT, 263
CARD, 332
CLIF, 402
COMPLEX, 419
CONTFRAC, 445
DECIMAL, 465
DERHAM, 530
DFLOAT, 587
DIRPROD, 547
DIRRING, 564
DIV, 575
DMP, 572
DPMM, 553
DPMO, 557
DSMP, 542
EMR, 702
EQ, 691
EXPEXPAN, 712
EXPR, 724
EXPUPXS, 740
FAGROUP, 1004
FAMONOID, 1007
FDIV, 813
FEXPR, 947
FF, 820
FFCG, 825
FFCGP, 835
FFCGX, 830
FFNB, 860
FFNBP, 872
FFNBX, 866
FFP, 851
FFX, 846
FLOAT, 908
FM, 1013
FM1, 1017
FNLA, 1027
FR, 786
FRAC, 986
FSERIES, 978
GCNAALG, 1065
GDMP, 1052
GMODPOL, 1059
GSERIES, 1090
HACKPI, 1978
HDMP, 1180
HDP, 1173
HELLFDIV, 1183
HEXADEC, 1143
IAN, 1276
IDEAL, 2081
IDPAG, 1202
IDPAM, 1206
IDPOAM, 1212
IDPOAMS, 1215
IFAMON, 1287
IFF, 1284
INDE, 1218
INT, 1362
INTRVL, 1384
IPADIC, 1294
IPF, 1303
IR, 1375
ISUPS, 1311
ITAYLOR, 1338
JORDAN, 223
LA, 1520
LAUPOL, 1422
LIE, 229
LO, 1523
LODO, 1469
LODO1, 1479
LODO2, 1491
LPOLY, 1447
LSQM, 1456
MCMLPX, 1543
MFLOAT, 1548
MINT, 1557
MODFIELD, 1638
MODMON, 1632
MODOP, 1647, 1804
MODRING, 1641
MPOLY, 1682
MRING, 1659
MYEXPR, 1688
MYUP, 1695

- NNI, [1739](#)
- NSDPS, [1703](#)
- NSMP, [1714](#)
- NSUP, [1729](#)
- OCT, [1765](#)
- ODP, [1816](#)
- ODPOL, [1852](#)
- ODR, [1858](#)
- OMLO, [1806](#)
- ONECOMP, [1777](#)
- ORDCOMP, [1810](#)
- ORESUP, [2492](#)
- OREUP, [2872](#)
- OWP, [1861](#)
- PACOFF, [2136](#)
- PACRAT, [2146](#)
- PADIC, [1881](#)
- PADICRAT, [1886](#)
- PADICRC, [1891](#)
- PF, [2105](#)
- PFR, [1914](#)
- POLY, [2078](#)
- PR, [2092](#)
- PRODUCT, [2113](#)
- PRITION, [1923](#)
- QFORM, [2154](#)
- QUAT, [2167](#)
- RADFF, [2194](#)
- RADIX, [2206](#)
- RECLOS, [2237](#)
- RESRING, [2296](#)
- RMATRIX, [2246](#)
- ROIRC, [2310](#)
- ROMAN, [2326](#)
- SAE, [2400](#)
- SD, [2572](#)
- SDPOL, [2386](#)
- SHDP, [2509](#)
- SINT, [2412](#)
- SMP, [2423](#)
- SMTS, [2441](#)
- SQMATRIX, [2548](#)
- SULS, [2457](#)
- SUP, [2467](#)
- SUPEXP, [2481](#)
- SUPXS, [2487](#)
- SUTS, [2497](#)
- SYMPOLY, [2654](#)
- TS, [2671](#)
- UFPS, [2789](#)
- ULS, [2796](#)
- ULSCONS, [2804](#)
- UP, [2828](#)
- UPXS, [2834](#)
- UPXSCONS, [2842](#)
- UPXSING, [2852](#)
- UTS, [2877](#)
- UTSZ, [2887](#)
- WP, [2949](#)
- XDPLY, [2967](#)
- XPBWPOLYL, [2987](#)
- XPOLY, [2998](#)
- XPR, [3007](#)
- XPOLY, [3013](#)
- ZMOD, [1368](#)
- zeroDim?
 - IDEAL, [2081](#)
- zeroMatrix
 - M3D, [2703](#)
- zeroOf
 - AN, [35](#)
 - EXPR, [724](#)
 - IAN, [1276](#)
- zeroSetSplit
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- zeroSetSplitIntoTriangularSystems
 - GTSET, [1083](#)
 - REGSET, [2286](#)
 - RGCHAIN, [2255](#)
 - SREGSET, [2535](#)
 - WUTSET, [2959](#)
- zerosOf
 - AN, [35](#)
 - EXPR, [724](#)
 - IAN, [1276](#)
- zeta
 - DIRRING, [564](#)
- ZMOD, [1367](#)

- ?, [1368](#)
- ?**?, [1368](#)
- ?*?, [1368](#)
- ?+?, [1368](#)
- ?-?, [1368](#)
- ?=?, [1368](#)
- ?^?, [1368](#)
- ?~=?, [1368](#)
- 0, [1368](#)
- 1, [1368](#)
- characteristic, [1368](#)
- coerce, [1368](#)
- convert, [1368](#)
- hash, [1368](#)
- index, [1368](#)
- init, [1368](#)
- latex, [1368](#)
- lookup, [1368](#)
- nextItem, [1368](#)
- one?, [1368](#)
- random, [1368](#)
- recip, [1368](#)
- sample, [1368](#)
- size, [1368](#)
- subtractIfCan, [1368](#)
- zero?, [1368](#)
- zoom
 - PLOT, [2029](#)
 - PLOT3D, [2042](#)
 - VIEW3D, [2710](#)
- zRange
 - PLOT3D, [2042](#)