axiom™

The 30 Year Horizon

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

Volume 9: Axiom Compiler

Portions Copyright (c) 2005 Timothy Daly

The Blue Bayou image Copyright (c) 2004 Jocelyn Guidry

Portions Copyright (c) 2004 Martin Dunstan

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.

Cyril Alberga	Roy Adler	Richard Anderson
George Andrews	Henry Baker	Stephen Balzac
Yurij Baransky	David R. Barton	Gerald Baumgartner
Gilbert Baumslag	Fred Blair	Vladimir Bondarenko
Mark Botch	Alexandre Bouyer	Peter A. Broadbery
Martin Brock	Manuel Bronstein	Florian Bundschuh
William Burge	Quentin Carpent	Bob Caviness
Bruce Char	Cheekai Chin	David V. Chudnovsky
Gregory V. Chudnovsky	Josh Cohen	Christophe Conil
Don Coppersmith	George Corliss	Robert Corless
Gary Cornell	Meino Cramer	Claire Di Crescenzo
Timothy Daly Sr.	Timothy Daly Jr.	James H. Davenport
Jean Della Dora	Gabriel Dos Reis	Michael Dewar
Claire DiCrescendo	Sam Dooley	Lionel Ducos
Martin Dunstan	Brian Dupee	Dominique Duval
Robert Edwards	Heow Eide-Goodman	Lars Erickson
Richard Fateman	Bertfried Fauser	Stuart Feldman
Brian Ford	Albrecht Fortenbacher	George Frances
Constantine Frangos	Timothy Freeman	Korrinn Fu
Marc Gaetano	Rudiger Gebauer	Kathy Gerber
Patricia Gianni	Holger Gollan	Teresa Gomez-Diaz
Laureano Gonzalez-Vega	Stephen Gortler	Johannes Grabmeier
Matt Grayson	James Griesmer	Vladimir Grinberg
Oswald Gschnitzer	Jocelyn Guidry	Steve Hague
Vilya Harvey	Satoshi Hamaguchi	Martin Hassner
Ralf Hemmecke	Henderson	Antoine Hersen
Pietro Iglio	Richard Jenks	Kai Kaminski
Grant Keady	Tony Kennedy	Paul Kosinski
Klaus Kusche	Bernhard Kutzler	Larry Lambe
Frederic Lehobey	Michel Levaud	Howard Levy
Rudiger Loos	Michael Lucks	Richard Luczak
Camm Maguire	Bob McElrath	Michael McGettrick
Ian Meikle	David Mentre	Victor S. Miller
Gerard Milmeister	Mohammed Mobarak	H. Michael Moeller
Michael Monagan	Marc Moreno-Maza	Scott Morrison
Mark Murray	William Naylor	C. Andrew Neff
John Nelder	Godfrey Nolan	Arthur Norman
Jinzhong Niu	Michael O'Connor	Kostas Oikonomou
Julian A. Padget	Bill Page	Jaap Weel
Susan Pelzel	Michel Petitot	Didier Pinchon
Claude Quitte	Norman Ramsey	Michael Richardson
Renaud Rioboo	Jean Rivlin	Nicolas Robidoux
Simon Robinson	Michael Rothstein	Martin Rubey
Philip Santas	Alfred Scheerhorn	William Schelter
Gerhard Schneider	Martin Schoenert	Marshall Schor
Fritz Schwarz	Nick Simicich	William Sit
Elena Smirnova	Jonathan Steinbach	Christine Sundaresan
Robert Sutor	Moss E. Sweedler	Eugene Surowitz
James Thatcher	Baldir Thomas	Mike Thomas
Dylan Thurston	Barry Trager	Themos T. Tsikas
Gregory Vanuxem	Bernhard Wall	Stephen Watt
Juergen Weiss	M. Weller	Mark Wegman
James Wen	Thorsten Werther	Michael Wester
John M. Wiley	Berhard Will	Clifton J. Williamson
Stephen Wilson	Shmuel Winograd	Robert Wisbauer
Sandra Wityak	Waldemar Wiwianka	Knut Wolf
Clifford Yapp	David Yun	Richard Zippel
Evelyn Zoernack	Bruno Zuercher	Dan Zwillinger

Contents

0.1 Makefile 1

1 Overview **3**
1.1 The Input 4
1.2 The Output, the EQ.nrlib directory 8
1.3 The code.lsp and EQ.lsp files 9
1.4 The code.o file 23
1.5 The info file 23
1.6 The EQ.fn file 26
1.7 The index.kaf file 31
1.7.1 The index offset byte 33
1.7.2 The "loadTimeStuff" 33
1.7.3 The "compilerInfo" 35
1.7.4 The "constructorForm" 42
1.7.5 The "constructorKind" 42
1.7.6 The "constructorModemap" 42
1.7.7 The "constructorCategory" 44
1.7.8 The "sourceFile" 45
1.7.9 The "modemaps" 45
1.7.10 The "operationAlist" 47
1.7.11 The "superDomain" 49
1.7.12 The "signaturesAndLocals" 49
1.7.13 The "attributes" 49
1.7.14 The "predicates" 50
1.7.15 The "abbreviation" 51
1.7.16 The "parents" 51
1.7.17 The "ancestors" 51
1.7.18 The "documentation" 51
1.7.19 The "slotInfo" 53
1.7.20 The "index" 55

2 Compiler top level **57**
2.1 Global Data Structures 57
2.2 Pratt Parsing 57
2.3)compile 58

2.3.1 Spad compiler . . . 61
2.4 Operator Precedence Table Initialization . . . 62
2.4.1 LED and NUD Tables . . . 63
2.5 Gliph Table . . . 65
2.5.1 Rename Token Table . . . 66
2.5.2 Generic function table . . . 66
2.6 Giant steps, Baby steps . . . 66

3 The Parser **67**
3.1 EQ.spad . . . 67
3.2 preparse . . . 71
3.2.1 defvar $index . . . 72
3.2.2 defvar $linelist . . . 72
3.2.3 defvar $echolinestack . . . 72
3.2.4 defvar $preparse-last-line . . . 72
3.3 Parsing routines . . . 72
3.3.1 defun initialize-preparse . . . 73
3.3.2 defun preparse . . . 76
3.3.3 defun Build the lines from the input for piles . . . 81
3.3.4 defun parsepiles . . . 86
3.3.5 defun add-parens-and-semis-to-line . . . 87
3.3.6 defun preparseReadLine . . . 88
3.3.7 defun skip-ifblock . . . 88
3.3.8 defun preparseReadLine1 . . . 89
3.4 I/O Handling . . . 90
3.4.1 defun preparse-echo . . . 90
3.4.2 defvar $current-fragment . . . 90
3.4.3 defun read-a-line . . . 91
3.5 Line Handling . . . 91
3.5.1 Line Buffer . . . 91
3.5.2 defstruct $line . . . 92
3.5.3 defvar $current-line . . . 92
3.5.4 defmacro line-clear . . . 92
3.5.5 defun line-print . . . 93
3.5.6 defun line-at-end-p . . . 93
3.5.7 defun line-past-end-p . . . 93
3.5.8 defun line-next-char . . . 93
3.5.9 defun line-advance-char . . . 94
3.5.10 defun line-current-segment . . . 94
3.5.11 defun line-new-line . . . 94
3.5.12 defun next-line . . . 95
3.5.13 defun Advance-Char . . . 95
3.5.14 defun storeblanks . . . 95
3.5.15 defun initial-substring . . . 96
3.5.16 defun get-a-line . . . 96
3.5.17 defun make-string-adjustable . . . 96

3.5.18 Parsing stack . . . 97
3.5.19 defstruct $stack . . . 97
3.5.20 defun stack-load . . . 97
3.5.21 defun stack-clear . . . 97
3.5.22 defmacro stack-/-empty . . . 98
3.5.23 defun stack-push . . . 98
3.5.24 defun stack-pop . . . 98
3.5.25 Parsing token . . . 99
3.5.26 defstruct $token . . . 99
3.5.27 defvar $prior-token . . . 99
3.5.28 defvar $nonblank . . . 99
3.5.29 defvar $current-token . . . 100
3.5.30 defvar $next-token . . . 100
3.5.31 defvar $valid-tokens . . . 100
3.5.32 defun token-install . . . 100
3.5.33 defun token-print . . . 101
3.5.34 Parsing reduction . . . 101
3.5.35 defstruct $reduction . . . 101

4 Parse Transformers 103
4.1 Direct called parse routines . . . 103
4.1.1 defun parseTransform . . . 103
4.1.2 defun parseTran . . . 103
4.1.3 defun parseAtom . . . 104
4.1.4 defun parseTranList . . . 105
4.1.5 defun parseConstruct . . . 105
4.1.6 defun parseConstruct . . . 105
4.2 Indirect called parse routines . . . 106
4.2.1 defun parseAnd . . . 107
4.2.2 defun parseAnd . . . 107
4.2.3 defun parseAtSign . . . 107
4.2.4 defun parseAtSign . . . 108
4.2.5 defun parseType . . . 108
4.2.6 defun parseCategory . . . 108
4.2.7 defun parseCategory . . . 109
4.2.8 defun parseDropAssertions . . . 109
4.2.9 defun parseCoerce . . . 109
4.2.10 defun parseCoerce . . . 110
4.2.11 defun parseColon . . . 110
4.2.12 defun parseColon . . . 110
4.2.13 defun parseDEF . . . 111
4.2.14 defun parseDEF . . . 111
4.2.15 defun parseLhs . . . 112
4.2.16 defun transIs . . . 112
4.2.17 defun transIs1 . . . 112
4.2.18 defun isListConstructor . . . 113

4.2.19 defun parseDollarGreaterthan . . . 114
4.2.20 defun parseDollarGreaterThan . . . 114
4.2.21 defun parseDollarGreaterEqual . . . 114
4.2.22 defun parseDollarGreaterEqual . . . 114
4.2.23 defun parseDollarLessEqual . . . 115
4.2.24 defun parseDollarNotEqual . . . 115
4.2.25 defun parseDollarNotEqual . . . 115
4.2.26 defun parseEquivalence . . . 116
4.2.27 defun parseEquivalence . . . 116
4.2.28 defun parseExit . . . 116
4.2.29 defun parseExit . . . 117
4.2.30 defun parseGreaterEqual . . . 117
4.2.31 defun parseGreaterEqual . . . 117
4.2.32 defun parseGreaterThan . . . 118
4.2.33 defun parseGreaterThan . . . 118
4.2.34 defun parseHas . . . 118
4.2.35 defun parseHas . . . 118
4.2.36 defun parseHasRhs . . . 120
4.2.37 defun parseIf,ifTran . . . 121
4.2.38 defun parseIf . . . 123
4.2.39 defun parseIf . . . 123
4.2.40 defun parseImplies . . . 123
4.2.41 defun parseImplies . . . 124
4.2.42 defun parseIn . . . 124
4.2.43 defun parseIn . . . 124
4.2.44 defun parseInBy . . . 125
4.2.45 defun parseInBy . . . 125
4.2.46 defun parseIs . . . 126
4.2.47 defun parseIs . . . 126
4.2.48 defun parseIsnt . . . 126
4.2.49 defun parseIsnt . . . 127
4.2.50 defun parseJoin . . . 127
4.2.51 defun parseJoin . . . 127
4.2.52 defun parseLeave . . . 128
4.2.53 defun parseLeave . . . 128
4.2.54 defun parseLessEqual . . . 128
4.2.55 defun parseLessEqual . . . 129
4.2.56 defun parseLET . . . 129
4.2.57 defun parseLET . . . 129
4.2.58 defun parseLETD . . . 130
4.2.59 defun parseLETD . . . 130
4.2.60 defun parseMDEF . . . 130
4.2.61 defun parseMDEF . . . 131
4.2.62 defun parseNot . . . 131
4.2.63 defun parseNot . . . 131
4.2.64 defun parseNot . . . 132

4.2.65 defun parseNotEqual . . . 132
4.2.66 defun parseNotEqual . . . 132
4.2.67 defun parseOr . . . 132
4.2.68 defun parseOr . . . 133
4.2.69 defun parsePretend . . . 133
4.2.70 defun parsePretend . . . 133
4.2.71 defun parseReturn . . . 134
4.2.72 defun parseReturn . . . 134
4.2.73 defun parseSegment . . . 135
4.2.74 defun parseSegment . . . 135
4.2.75 defun parseSeq . . . 135
4.2.76 defun parseSeq . . . 135
4.2.77 defun parseVCONS . . . 136
4.2.78 defun parseVCONS . . . 136
4.2.79 defun parseWhere . . . 136
4.2.80 defun parseWhere . . . 137

5 Compile Transformers 139
5.1 Direct called comp routines . . . 139
5.2 Indirect called comp routines . . . 139
5.2.1 defun compAtSign . . . 140
5.2.2 defun compAdd . . . 140
5.2.3 defun compAtSign . . . 142
5.2.4 defun compAtSign . . . 142
5.2.5 defun compCapsule . . . 143
5.2.6 defun compCapsule . . . 143
5.2.7 defun compCapsuleInner . . . 144
5.2.8 defun compCase . . . 144
5.2.9 defun compCase . . . 145
5.2.10 defun compCase1 . . . 145
5.2.11 defun compCat . . . 146
5.2.12 defun compCat . . . 146
5.2.13 defun compCat . . . 146
5.2.14 defun compCat . . . 147
5.2.15 defun compCategory . . . 147
5.2.16 defun compCategory . . . 147
5.2.17 defun compCoerce . . . 148
5.2.18 defun compCoerce . . . 148
5.2.19 defun compCoerce1 . . . 149
5.2.20 defun compColon . . . 150
5.2.21 defun compColon . . . 150
5.2.22 defun compCons . . . 154
5.2.23 defun compCons . . . 154
5.2.24 defun compCons1 . . . 154
5.2.25 defun compConstructorCategory . . . 155
5.2.26 defun compConstructorCategory . . . 155

5.2.27 defun compConstructorCategory . . . 156
5.2.28 defun compConstructorCategory . . . 156
5.2.29 defun compConstructorCategory . . . 156
5.2.30 defun compConstruct . . . 156
5.2.31 defun compConstruct . . . 157
5.2.32 defun compDefine . . . 158
5.2.33 defun compDefine . . . 158
5.2.34 defun compDefine1 . . . 158
5.2.35 defun compElt . . . 160
5.2.36 defun compElt . . . 160
5.2.37 defun compExit . . . 162
5.2.38 defun compExit . . . 162
5.2.39 defun compHas . . . 163
5.2.40 defun compHas . . . 163
5.2.41 defun compIf . . . 163
5.2.42 defun compIf . . . 164
5.2.43 defun compImport . . . 165
5.2.44 defun compImport . . . 165
5.2.45 defun compIs . . . 165
5.2.46 defun compIs . . . 165
5.2.47 defun compJoin . . . 166
5.2.48 defun compJoin . . . 166
5.2.49 defun compLambda . . . 168
5.2.50 defun compLambda . . . 168
5.2.51 defun compLeave . . . 169
5.2.52 defun compLeave . . . 169
5.2.53 defun compMacro . . . 170
5.2.54 defun compMacro . . . 170
5.2.55 defun compPretend . . . 171
5.2.56 defun compPretend . . . 171
5.2.57 defun compQuote . . . 172
5.2.58 defun compQuote . . . 172
5.2.59 defun compRepeatOrCollect . . . 172
5.2.60 defun compRepeatOrCollect . . . 173
5.2.61 defun compRepeatOrCollect . . . 173
5.2.62 defun compReduce . . . 175
5.2.63 defun compReduce . . . 175
5.2.64 defun compReduce1 . . . 175
5.2.65 defun compReturn . . . 177
5.2.66 defun compReturn . . . 177
5.2.67 defun compSeq . . . 178
5.2.68 defun compSeq . . . 179
5.2.69 defun compSeq1 . . . 179
5.2.70 defun compSeqItem . . . 180
5.2.71 defun compSetq . . . 180
5.2.72 defun compSetq . . . 180

5.2.73 defun compSetq . . . 180
5.2.74 defun compSetq1 . . . 181
5.2.75 defun setqSetelt . . . 181
5.2.76 defun setqSingle . . . 182
5.2.77 defun compString . . . 183
5.2.78 defun compString . . . 184
5.2.79 defun compSubDomain . . . 184
5.2.80 defun compSubDomain . . . 184
5.2.81 defun compSubDomain1 . . . 185
5.2.82 defun compSubsetCategory . . . 186
5.2.83 defun compSubsetCategory . . . 186
5.2.84 defun compSuchthat . . . 186
5.2.85 defun compSuchthat . . . 187
5.2.86 defun compVector . . . 187
5.2.87 defun compVector . . . 187
5.2.88 defun compWhere . . . 188
5.2.89 defun compWhere . . . 188

6 Post Transformers 191
6.1 Direct called postparse routines . . . 191
6.1.1 defun postTransform . . . 191
6.1.2 defun postTran . . . 192
6.1.3 defun postOp . . . 193
6.1.4 defun postAtom . . . 193
6.1.5 defun postTranList . . . 194
6.1.6 defun postScriptsForm . . . 194
6.1.7 defun postTranScripts . . . 194
6.1.8 defun postTransformCheck . . . 195
6.1.9 defun postcheck . . . 195
6.1.10 defun postError . . . 196
6.1.11 defun postForm . . . 196
6.2 Indirect called postparse routines . . . 197
6.2.1 defun postAdd . . . 198
6.2.2 defun postAdd . . . 198
6.2.3 defun postCapsule . . . 199
6.2.4 defun postBlockItemList . . . 199
6.2.5 defun postAtSign . . . 200
6.2.6 defun postAtSign . . . 200
6.2.7 defun postType . . . 200
6.2.8 defun postBigFloat . . . 201
6.2.9 defun postBigFloat . . . 201
6.2.10 defun postBlock . . . 201
6.2.11 defun postBlock . . . 202
6.2.12 defun postCategory . . . 202
6.2.13 defun postCategory . . . 202
6.2.14 defun postCollect,finish . . . 203

6.2.15 defun postMakeCons . . . 204
6.2.16 defun postCollect . . . 204
6.2.17 defun postCollect . . . 204
6.2.18 defun postIteratorList . . . 205
6.2.19 defun postColon . . . 206
6.2.20 defun postColon . . . 206
6.2.21 defun postColonColon . . . 206
6.2.22 defun postColonColon . . . 207
6.2.23 defun postComma . . . 207
6.2.24 defun postComma . . . 207
6.2.25 defun comma2Tuple . . . 207
6.2.26 defun postFlatten . . . 208
6.2.27 defun postConstruct . . . 208
6.2.28 defun postConstruct . . . 208
6.2.29 defun postTranSegment . . . 209
6.2.30 defun postDef . . . 210
6.2.31 defun postDef . . . 210
6.2.32 defun postDefArgs . . . 211
6.2.33 defun postExit . . . 212
6.2.34 defun postExit . . . 212
6.2.35 defun postIf . . . 212
6.2.36 defun postIf . . . 213
6.2.37 defun postin . . . 213
6.2.38 defun postin . . . 213
6.2.39 defun postInSeq . . . 214
6.2.40 defun postIn . . . 214
6.2.41 defun postIn . . . 214
6.2.42 defun postJoin . . . 215
6.2.43 defun postJoin . . . 215
6.2.44 defun postMapping . . . 215
6.2.45 defun postMapping . . . 216
6.2.46 defun postMDef . . . 216
6.2.47 defun postMDef . . . 216
6.2.48 defun postPretend . . . 217
6.2.49 defun postPretend . . . 218
6.2.50 defun postQUOTE . . . 218
6.2.51 defun postQUOTE . . . 218
6.2.52 defun postReduce . . . 218
6.2.53 defun postReduce . . . 219
6.2.54 defun postRepeat . . . 219
6.2.55 defun postRepeat . . . 219
6.2.56 defun postScripts . . . 220
6.2.57 defun postScripts . . . 220
6.2.58 defun postSemiColon . . . 220
6.2.59 defun postSemiColon . . . 220
6.2.60 defun postFlattenLeft . . . 221

6.2.61 defun postSignature . . . 221
6.2.62 defun postSignature . . . 221
6.2.63 defun killColons . . . 222
6.2.64 defun postSlash . . . 222
6.2.65 defun postSlash . . . 222
6.2.66 defun postTuple . . . 223
6.2.67 defun postTuple . . . 223
6.2.68 defun postTupleCollect . . . 223
6.2.69 defun postTupleCollect . . . 224
6.2.70 defun postWhere . . . 224
6.2.71 defun postWhere . . . 224
6.2.72 defun postWith . . . 225
6.2.73 defun postWith . . . 225
6.3 Support routines . . . 225
6.3.1 defun setDefOp . . . 225
6.3.2 defun aplTran . . . 226
6.3.3 defun aplTran1 . . . 226
6.3.4 defun aplTranList . . . 228
6.3.5 defun hasAplExtension . . . 228
6.3.6 defun deepestExpression . . . 229
6.3.7 defun containsBang . . . 229
6.3.8 defun getScriptName . . . 230
6.3.9 defun decodeScripts . . . 230

7 DEF forms 231
7.0.10 defvar $defstack . . . 231
7.0.11 defvar $is-spill . . . 231
7.0.12 defvar $is-spill-list . . . 231
7.0.13 defvar $vl . . . 232
7.0.14 defvar $is-gensymlist . . . 232
7.0.15 defvar $initial-gensym . . . 232
7.0.16 defvar $is-eqlist . . . 232
7.0.17 defun hackforis . . . 232
7.0.18 defun hackforis1 . . . 233
7.0.19 defun unTuple . . . 233
7.0.20 defun errhuh . . . 233

8 PARSE forms 235
8.1 The original meta specification . . . 235
8.2 The PARSE code . . . 240
8.2.1 defvar $tmptok . . . 240
8.2.2 defvar $tok . . . 240
8.2.3 defvar $ParseMode . . . 241
8.2.4 defvar $definition-name . . . 241
8.2.5 defvar $lablasoc . . . 241
8.2.6 defun PARSE-NewExpr . . . 241

8.2.7 defun PARSE-Command . . . 242
8.2.8 defun PARSE-SpecialKeyWord . . . 242
8.2.9 defun PARSE-SpecialCommand . . . 243
8.2.10 defun PARSE-TokenCommandTail . . . 243
8.2.11 defun PARSE-TokenOption . . . 244
8.2.12 defun PARSE-TokenList . . . 244
8.2.13 defun PARSE-CommandTail . . . 245
8.2.14 defun PARSE-PrimaryOrQM . . . 245
8.2.15 defun PARSE-Option . . . 246
8.2.16 defun PARSE-Statement . . . 246
8.2.17 defun PARSE-InfixWith . . . 247
8.2.18 defun PARSE-With . . . 247
8.2.19 defun PARSE-Category . . . 247
8.2.20 defun PARSE-Expression . . . 249
8.2.21 defun PARSE-Import . . . 249
8.2.22 defun PARSE-Expr . . . 250
8.2.23 defun PARSE-LedPart . . . 250
8.2.24 defun PARSE-NudPart . . . 250
8.2.25 defun PARSE-Operation . . . 251
8.2.26 defun PARSE-leftBindingPowerOf . . . 251
8.2.27 defun PARSE-rightBindingPowerOf . . . 252
8.2.28 defun PARSE-getSemanticForm . . . 252
8.2.29 defun PARSE-Prefix . . . 252
8.2.30 defun PARSE-Infix . . . 253
8.2.31 defun PARSE-TokTail . . . 254
8.2.32 defun PARSE-Qualification . . . 254
8.2.33 defun PARSE-Reduction . . . 255
8.2.34 defun PARSE-ReductionOp . . . 255
8.2.35 defun PARSE-Form . . . 255
8.2.36 defun PARSE-Application . . . 256
8.2.37 defun PARSE-Label . . . 257
8.2.38 defun PARSE-Selector . . . 257
8.2.39 defun PARSE-PrimaryNoFloat . . . 258
8.2.40 defun PARSE-Primary . . . 258
8.2.41 defun PARSE-Primary1 . . . 258
8.2.42 defun PARSE-Float . . . 259
8.2.43 defun PARSE-FloatBase . . . 260
8.2.44 defun PARSE-FloatBasePart . . . 260
8.2.45 defun PARSE-FloatExponent . . . 261
8.2.46 defun PARSE-Enclosure . . . 262
8.2.47 defun PARSE-IntegerTok . . . 262
8.2.48 defun PARSE-FormalParameter . . . 263
8.2.49 defun PARSE-FormalParameterTok . . . 263
8.2.50 defun PARSE-Quad . . . 263
8.2.51 defun PARSE-String . . . 263
8.2.52 defun PARSE-VarForm . . . 264

8.2.53 defun PARSE-Scripts . . . 264
8.2.54 defun PARSE-ScriptItem . . . 265
8.2.55 defun PARSE-Name . . . 265
8.2.56 defun PARSE-Data . . . 266
8.2.57 defun PARSE-Sexpr . . . 266
8.2.58 defun PARSE-Sexpr1 . . . 266
8.2.59 defun PARSE-NBGliphTok . . . 267
8.2.60 defun PARSE-GliphTok . . . 268
8.2.61 defun PARSE-AnyId . . . 268
8.2.62 defun PARSE-Sequence . . . 269
8.2.63 defun PARSE-Sequence1 . . . 269
8.2.64 defun PARSE-OpenBracket . . . 270
8.2.65 defun PARSE-OpenBrace . . . 270
8.2.66 defun PARSE-IteratorTail . . . 271
8.2.67 defun PARSE-Iterator . . . 271
8.2.68 The PARSE implicit routines . . . 272
8.2.69 defun PARSE-Suffix . . . 272
8.2.70 defun PARSE-SemiColon . . . 273
8.2.71 defun PARSE-Return . . . 273
8.2.72 defun PARSE-Exit . . . 273
8.2.73 defun PARSE-Leave . . . 274
8.2.74 defun PARSE-Seg . . . 274
8.2.75 defun PARSE-Conditional . . . 275
8.2.76 defun PARSE-ElseClause . . . 275
8.2.77 defun PARSE-Loop . . . 276
8.2.78 defun PARSE-LabelExpr . . . 276
8.2.79 defun PARSE-FloatTok . . . 277
8.3 The PARSE support routines . . . 277
8.3.1 String grabbing . . . 278
8.3.2 defun match-string . . . 278
8.3.3 defun skip-blanks . . . 278
8.3.4 defun token-lookahead-type . . . 279
8.3.5 defun match-advance-string . . . 279
8.3.6 defun initial-substring-p . . . 280
8.3.7 defun quote-if-string . . . 280
8.3.8 defun escape-keywords . . . 281
8.3.9 defun isTokenDelimiter . . . 281
8.3.10 defun underscore . . . 282
8.3.11 Token Handling . . . 282
8.3.12 defun getToken . . . 282
8.3.13 defun unget-tokens . . . 282
8.3.14 defun match-current-token . . . 283
8.3.15 defun match-token . . . 284
8.3.16 defun match-next-token . . . 284
8.3.17 defun current-symbol . . . 284
8.3.18 defun make-symbol-of . . . 284

8.3.19 defun current-token . . . 285
8.3.20 defun try-get-token . . . 285
8.3.21 defun next-token . . . 286
8.3.22 defun advance-token . . . 286
8.3.23 defvar $XTokenReader . . . 287
8.3.24 defun get-token . . . 287
8.3.25 Character handling . . . 287
8.3.26 defun current-char . . . 287
8.3.27 defun next-char . . . 287
8.3.28 defun char-eq . . . 288
8.3.29 defun char-ne . . . 288
8.3.30 Error handling . . . 288
8.3.31 defvar $meta-error-handler . . . 288
8.3.32 defun meta-syntax-error . . . 289
8.3.33 Floating Point Support . . . 289
8.3.34 defun floatexpid . . . 289
8.3.35 Dollar Translation . . . 289
8.3.36 defun dollarTran . . . 289
8.3.37 Applying metagrammatical elements of a production (e.g., Star). . . . 290
8.3.38 defmacro Bang . . . 290
8.3.39 defmacro must . . . 290
8.3.40 defun action . . . 291
8.3.41 defun optional . . . 291
8.3.42 defmacro star . . . 291
8.3.43 Stacking and retrieving reductions of rules. . . . 292
8.3.44 defvar $reduce-stack . . . 292
8.3.45 defmacro reduce-stack-clear . . . 292
8.3.46 defun push-reduction . . . 292

9 Utility Functions 295
9.0.47 defun translabel . . . 295
9.0.48 defun translabel1 . . . 295
9.0.49 defun displayPreCompilationErrors . . . 296
9.0.50 defun bumperrorcount . . . 297
9.0.51 defun parseTranCheckForRecord . . . 297
9.0.52 defun new2OldLisp . . . 298
9.0.53 defun makeSimplePredicateOrNil . . . 298
9.0.54 defun parse-spadstring . . . 298
9.0.55 defun parse-string . . . 299
9.0.56 defun parse-identifier . . . 299
9.0.57 defun parse-number . . . 300
9.0.58 defun parse-keyword . . . 300
9.0.59 defun parse-argument-designator . . . 301
9.0.60 defun print-package . . . 301
9.0.61 defun checkWarning . . . 301

9.0.62 defun tuple2List . . . 302
9.0.63 defmacro pop-stack-1 . . . 302
9.0.64 defmacro pop-stack-2 . . . 303
9.0.65 defmacro pop-stack-3 . . . 303
9.0.66 defmacro pop-stack-4 . . . 303
9.0.67 defmacro nth-stack . . . 304
9.0.68 defun Pop-Reduction . . . 304
9.0.69 defun addclose . . . 304
9.0.70 defun blankp . . . 305
9.0.71 defun drop . . . 305
9.0.72 defun escaped . . . 305
9.0.73 defvar $comblocklist . . . 305
9.0.74 defun fincomblock . . . 306
9.0.75 defun indent-pos . . . 306
9.0.76 defun infixtok . . . 307
9.0.77 defun is-console . . . 307
9.0.78 defun next-tab-loc . . . 307
9.0.79 defun nonblankloc . . . 308
9.0.80 defun parseprint . . . 308
9.0.81 defun skip-to-endif . . . 308

10 The Compiler 309
10.1 Compiling EQ.spad . . . 309
10.1.1 The top level compiler command . . . 312
10.1.2 The Spad compiler top level function . . . 314
10.1.3 defun compilerDoit . . . 318
10.1.4 defun /RQ,LIB . . . 319
10.1.5 defun /rf-1 . . . 320
10.1.6 defun spad . . . 329
10.1.7 defun Interpreter interface to the compiler . . . 330
10.1.8 defun print-defun . . . 333
10.1.9 defun def-rename . . . 333
10.1.10 defun def-rename1 . . . 333
10.1.11 defun compTopLevel . . . 334
10.1.12 defun compOrCroak . . . 335
10.1.13 defun compOrCroak1 . . . 336
10.1.14 defun comp . . . 337
10.1.15 defun compNoStacking . . . 338
10.1.16 defun compNoStacking1 . . . 338
10.1.17 defun comp2 . . . 339
10.1.18 defun comp3 . . . 339
10.1.19 defun compTypeOf . . . 342
10.1.20 defun compColonInside . . . 342
10.1.21 defun compAtom . . . 343
10.1.22 defun convert . . . 344
10.1.23 defun primitiveType . . . 345

10.1.24 defun compSymbol 345
10.1.25 defun compList 347
10.1.26 defun compExpression 347
10.1.27 defun compForm 348
10.1.28 defun compForm1 348
10.1.29 defun compForm2 350
10.1.30 defun compArgumentsAndTryAgain 352
10.1.31 defun compWithMappingMode 353
10.1.32 defun compWithMappingMode1 353
10.1.33 defun extractCodeAndConstructTriple 360
10.1.34 defun hasFormalMapVariable 361
10.1.35 defun argsToSig 361
10.1.36 defun compMakeDeclaration 362
10.1.37 defun modifyModeStack 363
10.1.38 defun Create a list of unbound symbols 363
10.1.39 defun compOrCroak1,compactify 364
10.1.40 defun Compiler/Interpreter interface 365
10.1.41 defun compileSpadLispCmd 365
10.1.42 defun recompile-lib-file-if-necessary 366
10.1.43 defun spad-fixed-arg 367
10.1.44 defun compile-lib-file 367
10.1.45 defun compileFileQuietly 368
10.1.46 defvar $byConstructors 368
10.1.47 defvar $constructorsSeen 368

11 Index **381**

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))

0.1 Makefile

This book is actually a literate program[2] and contains executable source code. In particular, the Makefile for this book is part of the source of the book and is included below. Axiom uses the "noweb" literate programming system by Norman Ramsey[6].

Chapter 1

Overview

The Spad language is a mathematically oriented language intended for writing computational mathematics. It derives its logical structure from abstract algebra. It features ideas that are still not available in general purpose programming languages, such as selecting overloaded procedures based on the return type as well as the types of the arguments.

The Spad language is heavily influenced by Barbara Liskov's work. It features encapsulation (aka objects), inheritance, and overloading. It has categories which are defined by the exports. Categories are parameterized functors that take arguments which define their behavior.

More details on the language and its high level concepts is available in the Programmers Guide, Volume 3.

The Spad compiler accepts the Spad language and generates a set of files used by the interpreter, detailed in Volume 5.

The compiler does not produce stand-alone executable code. It assumes that it will run inside the interpreter and that the code it generates will be loaded into the interpreter.

Some of the routines are common to both the compiler and the interpreter. Where this happens we have favored the interpreter volume (Volume 5) as the official source location. In each case we will make reference to that volume and the code in it. Thus, the compiler volume should be considered as an extension of the interpreter document.

This volume will go into painful detail of every aspect of compiling Spad code. We will start by defining the input to, and output from the compiler so we know what we are trying to achieve.

Next we will look at the top level data structures used by the compiler. Unfortunately, the compiler uses a large number of "global variables" to pass information and alter control flow. Some of these are used by many routines and some of

these are very local to a small subset or a recursion. We will cover the minor ones as they arise.

Next we examine the Pratt parser idea and the Led and Nud concepts, which is used to drive the low level parsing.

Following that we journey deep into the code, trying our best not to get lost in the details. The code is introduced based on "motivation" rather than in strict execution order or related concept order. We do this to try to make the compiler a "readable novel" rather than a mud-march through the code. The goal is to keep the reader's interest while trying to be exact. Sometimes this will require detours to discuss subtopics.

"Motivating" a piece of software is a not-very-well established form of narrative writing so we assume your forgiveness if we get it wrong. Worse yet, some of the pieces of the system are "legacy", in that they are no longer used and should be removed. Other parts of the system may have very weak descriptions because we simply do not understand them either. Since this is a living document and the code for the system is actually the code you are reading we will expand parts as we go.

1.1 The Input

```
)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
++ Basic Operations: =
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: equation
++ Examples:
++ References:
++ 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(eqn) returns the left hand side of equation eqn.
rhs: $ -> S
    ++ rhs(eqn) 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(eqn, x=f) replaces x by f in equation eqn.
       eval: ($, List $) -> $
        ++ eval(eqn, [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
        l:S   * eqn:$ == l       * eqn.lhs = l       * eqn.rhs
        l:S * eqn:$  ==  l * eqn.lhs    =    l * eqn.rhs
        eqn:$ * l:S  ==  eqn.lhs * l    =    eqn.rhs * l
        -- 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)]
```

1.2 The Output, the EQ.nrlib directory

The Spad compiler generates several files in a directory named after the input abbreviation. The input file contains an abbreviation line:

```
)abbrev domain EQ Equation
```

for each category, domain, or package. The abbreviation line has 3 parts.

- one of “category”, “domain”, or “package”

- the abbreviation for this domain (8 Uppercase Characters maximum)
- the name of this domain

Since the abbreviation for the Equation domain is EQ, the compiler will put all of its output into a subdirectory called "EQ.nrlib". The "nrlib" is a port of a very old VMLisp file format, simulated with directories.

For the EQ input file, the compiler will create the following output files, each of which we will explain in detail below.

```
/research/test/int/algebra/EQ.nrlib:
used 216 available 4992900
drwxr-xr-x    2 root root  4096 2010-12-09 11:20 .
drwxr-xr-x 1259 root root 73728 2010-12-09 11:43 ..
-rw-r--r--    1 root root 19228 2010-12-09 11:20 code.lsp
-rw-r--r--    1 root root 34074 2010-12-09 11:20 code.o
-rw-r--r--    1 root root 13543 2010-12-09 11:20 EQ.fn
-rw-r--r--    1 root root 19228 2010-12-09 11:20 EQ.lsp
-rw-r--r--    1 root root 36148 2010-12-09 11:20 index.kaf
-rw-r--r--    1 root root  6236 2010-12-09 11:20 info
```

1.3 The code.lsp and EQ.lsp files

```
(/VERSIONCHECK 2)

(DEFUN |EQ;factorAndSplit;$L;1| (|eq| $)
  (PROG (|eq0| #:G1403 |rcf| #:G1404)
    (RETURN
      (SEQ (COND
             (((|HasSignature| (QREFELT $ 6)
                  (LIST '|factor|
                        (LIST (LIST '|Factored|
                                    (|devaluate| (QREFELT $ 6)))
                              (|devaluate| (QREFELT $ 6)))))
              (SEQ (LETT |eq0| (SPADCALL |eq| (QREFELT $ 8))
                         |EQ;factorAndSplit;$L;1|)
                   (EXIT (PROGN
                           (LETT #:G1403 NIL |EQ;factorAndSplit;$L;1|)
                           (SEQ (LETT |rcf| NIL
                                      |EQ;factorAndSplit;$L;1|)
                                (LETT #:G1404
                                      (SPADCALL
                                       (SPADCALL
                                        (SPADCALL |eq0| (QREFELT $ 9))
                                        (QREFELT $ 11))
                                       (QREFELT $ 15))
                                      |EQ;factorAndSplit;$L;1|)
```

```
                                G190
                                (COND
                                  ((OR (ATOM #:G1404)
                                    (PROGN
                                      (LETT |rcf| (CAR #:G1404)
                                       |EQ;factorAndSplit;$L;1|)
                                      NIL))
                                   (GO G191)))
                                (SEQ (EXIT
                                      (LETT #:G1403
                                       (CONS
                                        (SPADCALL (QCAR |rcf|)
                                         (|spadConstant| $ 16)
                                         (QREFELT $ 17))
                                        #:G1403)
                                       |EQ;factorAndSplit;$L;1|)))
                                (LETT #:G1404 (CDR #:G1404)
                                      |EQ;factorAndSplit;$L;1|)
                                (GO G190) G191
                                (EXIT (NREVERSE0 #:G1403)))))))
            ('T (LIST |eq|)))))))

(PUT (QUOTE |EQ;=;2S$;2|) (QUOTE |SPADreplace|) (QUOTE CONS))

(DEFUN |EQ;=;2S$;2| (|l| |r| $) (CONS |l| |r|))

(PUT (QUOTE |EQ;equation;2S$;3|) (QUOTE |SPADreplace|) (QUOTE CONS))

(DEFUN |EQ;equation;2S$;3| (|l| |r| $) (CONS |l| |r|))

(PUT (QUOTE |EQ;lhs;$S;4|) (QUOTE |SPADreplace|) (QUOTE QCAR))

(DEFUN |EQ;lhs;$S;4| (|eqn| $) (QCAR |eqn|))

(PUT (QUOTE |EQ;rhs;$S;5|) (QUOTE |SPADreplace|) (QUOTE QCDR))

(DEFUN |EQ;rhs;$S;5| (|eqn| $) (QCDR |eqn|))

(DEFUN |EQ;swap;2$;6| (|eqn| $) (CONS (SPADCALL |eqn| (QREFELT $ 21))
 (SPADCALL |eqn| (QREFELT $ 9))))

(DEFUN |EQ;map;M2$;7| (|fn| |eqn| $)
 (SPADCALL
  (SPADCALL (QCAR |eqn|) |fn|)
  (SPADCALL (QCDR |eqn|) |fn|)
  (QREFELT $ 17)))

(DEFUN |EQ;eval;$SS$;8| (|eqn| |s| |x| $)
 (SPADCALL
  (SPADCALL (QCAR |eqn|) |s| |x| (QREFELT $ 26))
```

```
  (SPADCALL (QCDR |eqn|) |s| |x| (QREFELT $ 26))
  (QREFELT $ 20)))

(DEFUN |EQ;eval;$LL$;9| (|eqn| |ls| |lx| $)
 (SPADCALL
  (SPADCALL (QCAR |eqn|) |ls| |lx| (QREFELT $ 30))
  (SPADCALL (QCDR |eqn|) |ls| |lx| (QREFELT $ 30))
  (QREFELT $ 20)))

(DEFUN |EQ;eval;3$;10| (|eqn1| |eqn2| $)
 (SPADCALL
  (SPADCALL (QCAR |eqn1|) |eqn2| (QREFELT $ 33))
  (SPADCALL (QCDR |eqn1|) |eqn2| (QREFELT $ 33))
  (QREFELT $ 20)))

(DEFUN |EQ;eval;$L$;11| (|eqn1| |leqn2| $)
 (SPADCALL
  (SPADCALL (QCAR |eqn1|) |leqn2| (QREFELT $ 36))
  (SPADCALL (QCDR |eqn1|) |leqn2| (QREFELT $ 36))
  (QREFELT $ 20)))

(DEFUN |EQ;=;2$B;12| (|eq1| |eq2| $)
 (COND
  ((SPADCALL (QCAR |eq1|) (QCAR |eq2|) (QREFELT $ 39))
   (SPADCALL (QCDR |eq1|) (QCDR |eq2|) (QREFELT $ 39)))
  ((QUOTE T) (QUOTE NIL))))

(DEFUN |EQ;coerce;$Of;13| (|eqn| $)
 (SPADCALL
  (SPADCALL (QCAR |eqn|) (QREFELT $ 42))
  (SPADCALL (QCDR |eqn|) (QREFELT $ 42))
  (QREFELT $ 43)))

(DEFUN |EQ;coerce;$B;14| (|eqn| $)
 (SPADCALL (QCAR |eqn|) (QCDR |eqn|) (QREFELT $ 39)))

(DEFUN |EQ;+;3$;15| (|eq1| |eq2| $)
 (SPADCALL
  (SPADCALL (QCAR |eq1|) (QCAR |eq2|) (QREFELT $ 46))
  (SPADCALL (QCDR |eq1|) (QCDR |eq2|) (QREFELT $ 46))
  (QREFELT $ 20)))

(DEFUN |EQ;+;S2$;16| (|s| |eq2| $)
 (SPADCALL (CONS |s| |s|) |eq2| (QREFELT $ 47)))

(DEFUN |EQ;+;$S$;17| (|eq1| |s| $)
 (SPADCALL |eq1| (CONS |s| |s|) (QREFELT $ 47)))

(DEFUN |EQ;-;2$;18| (|eq| $)
 (SPADCALL
```

```
  (SPADCALL (SPADCALL |eq| (QREFELT $ 9)) (QREFELT $ 50))
  (SPADCALL (SPADCALL |eq| (QREFELT $ 21)) (QREFELT $ 50))
  (QREFELT $ 20)))

(DEFUN |EQ;-;S2$;19| (|s| |eq2| $)
 (SPADCALL (CONS |s| |s|) |eq2| (QREFELT $ 52)))

(DEFUN |EQ;-;$S$;20| (|eq1| |s| $)
 (SPADCALL |eq1| (CONS |s| |s|) (QREFELT $ 52)))

(DEFUN |EQ;leftZero;2$;21| (|eq| $)
 (SPADCALL
  (|spadConstant| $ 16)
  (SPADCALL
   (SPADCALL |eq| (QREFELT $ 21))
   (SPADCALL |eq| (QREFELT $ 9))
   (QREFELT $ 56))
  (QREFELT $ 20)))

(DEFUN |EQ;rightZero;2$;22| (|eq| $)
 (SPADCALL
  (SPADCALL
   (SPADCALL |eq| (QREFELT $ 9))
   (SPADCALL |eq| (QREFELT $ 21))
   (QREFELT $ 56))
  (|spadConstant| $ 16)
  (QREFELT $ 20)))

(DEFUN |EQ;Zero;$;23| ($)
 (SPADCALL (|spadConstant| $ 16) (|spadConstant| $ 16) (QREFELT $ 17)))

(DEFUN |EQ;-;3$;24| (|eq1| |eq2| $)
 (SPADCALL
  (SPADCALL (QCAR |eq1|) (QCAR |eq2|) (QREFELT $ 56))
  (SPADCALL (QCDR |eq1|) (QCDR |eq2|) (QREFELT $ 56))
  (QREFELT $ 20)))

(DEFUN |EQ;*;3$;25| (|eq1| |eq2| $)
 (SPADCALL
  (SPADCALL (QCAR |eq1|) (QCAR |eq2|) (QREFELT $ 58))
  (SPADCALL (QCDR |eq1|) (QCDR |eq2|) (QREFELT $ 58))
  (QREFELT $ 20)))

(DEFUN |EQ;*;S2$;26| (|l| |eqn| $)
 (SPADCALL
  (SPADCALL |l| (QCAR |eqn|) (QREFELT $ 58))
  (SPADCALL |l| (QCDR |eqn|) (QREFELT $ 58))
  (QREFELT $ 20)))

(DEFUN |EQ;*;S2$;27| (|l| |eqn| $)
```

```
 (SPADCALL
  (SPADCALL |l| (QCAR |eqn|) (QREFELT $ 58))
  (SPADCALL |l| (QCDR |eqn|) (QREFELT $ 58))
  (QREFELT $ 20)))

(DEFUN |EQ;*;$S$;28| (|eqn| |l| $)
 (SPADCALL
  (SPADCALL (QCAR |eqn|) |l| (QREFELT $ 58))
  (SPADCALL (QCDR |eqn|) |l| (QREFELT $ 58))
  (QREFELT $ 20)))

(DEFUN |EQ;One;$;29| ($)
 (SPADCALL (|spadConstant| $ 62) (|spadConstant| $ 62) (QREFELT $ 17)))

(DEFUN |EQ;recip;$U;30| (|eq| $)
 (PROG (|lh| |rh|)
  (RETURN
   (SEQ
    (LETT |lh|
     (SPADCALL (SPADCALL |eq| (QREFELT $ 9)) (QREFELT $ 65))
     |EQ;recip;$U;30|)
    (EXIT
     (COND
      ((QEQCAR |lh| 1) (CONS 1 "failed"))
      ('T
       (SEQ
        (LETT |rh|
         (SPADCALL (SPADCALL |eq| (QREFELT $ 21)) (QREFELT $ 65))
         |EQ;recip;$U;30|)
        (EXIT
         (COND
          ((QEQCAR |rh| 1) (CONS 1 "failed"))
          ('T
            (CONS 0
             (CONS (QCDR |lh|) (QCDR |rh|)))))))))))))))

(DEFUN |EQ;leftOne;$U;31| (|eq| $)
 (PROG (|re|)
  (RETURN
   (SEQ
    (LETT |re|
     (SPADCALL (SPADCALL |eq| (QREFELT $ 9)) (QREFELT $ 65))
     |EQ;leftOne;$U;31|)
    (EXIT
     (COND
      ((QEQCAR |re| 1) (CONS 1 "failed"))
      ('T
       (CONS 0
        (SPADCALL
         (|spadConstant| $ 62)
```

```
         (SPADCALL (SPADCALL |eq| (QREFELT $ 21)) (QCDR |re|) (QREFELT $ 58))
         (QREFELT $ 20))))))))))


(DEFUN |EQ;rightOne;$U;32| (|eq| $)
 (PROG (|re|)
  (RETURN
   (SEQ
    (LETT |re|
     (SPADCALL (SPADCALL |eq| (QREFELT $ 21)) (QREFELT $ 65))
     |EQ;rightOne;$U;32|)
    (EXIT
     (COND
      ((QEQCAR |re| 1) (CONS 1 "failed"))
      ('T
       (CONS 0
        (SPADCALL
         (SPADCALL (SPADCALL |eq| (QREFELT $ 9)) (QCDR |re|) (QREFELT $ 58))
         (|spadConstant| $ 62)
         (QREFELT $ 20)))))))))))


(DEFUN |EQ;inv;2$;33| (|eq| $)
  (CONS (SPADCALL (SPADCALL |eq| (QREFELT $ 9)) (QREFELT $ 69))
        (SPADCALL (SPADCALL |eq| (QREFELT $ 21)) (QREFELT $ 69))))

(DEFUN |EQ;leftOne;$U;34| (|eq| $)
  (CONS 0
        (SPADCALL (|spadConstant| $ 62)
            (SPADCALL (SPADCALL |eq| (QREFELT $ 21))
                (SPADCALL (SPADCALL |eq| (QREFELT $ 21))
                    (QREFELT $ 69))
                (QREFELT $ 58))
            (QREFELT $ 20))))

(DEFUN |EQ;rightOne;$U;35| (|eq| $)
  (CONS 0
        (SPADCALL
            (SPADCALL (SPADCALL |eq| (QREFELT $ 9))
                (SPADCALL (SPADCALL |eq| (QREFELT $ 21))
                    (QREFELT $ 69))
                (QREFELT $ 58))
            (|spadConstant| $ 62) (QREFELT $ 20))))

(DEFUN |EQ;characteristic;Nni;36| ($) (SPADCALL (QREFELT $ 72)))

(DEFUN |EQ;*;I2$;37| (|i| |eq| $)
 (SPADCALL (SPADCALL |i| (QREFELT $ 75)) |eq| (QREFELT $ 60)))
```

```
(DEFUN |EQ;factorAndSplit;$L;38| (|eq| $)
  (PROG (#:G1488 #:G1489 |eq0| |p| #:G1490 |rcf| #:G1491)
    (RETURN
      (SEQ (COND
             ((|HasSignature| (QREFELT $ 6)
                  (LIST '|factor|
                        (LIST (LIST '|Factored|
                                    (|devaluate| (QREFELT $ 6)))
                              (|devaluate| (QREFELT $ 6)))))
              (SEQ (LETT |eq0| (SPADCALL |eq| (QREFELT $ 8))
                         |EQ;factorAndSplit;$L;38|)
                   (EXIT (PROGN
                           (LETT #:G1488 NIL |EQ;factorAndSplit;$L;38|)
                           (SEQ (LETT |rcf| NIL
                                      |EQ;factorAndSplit;$L;38|)
                                (LETT #:G1489
                                      (SPADCALL
                                       (SPADCALL
                                        (SPADCALL |eq0| (QREFELT $ 9))
                                        (QREFELT $ 11))
                                       (QREFELT $ 15))
                                      |EQ;factorAndSplit;$L;38|)
                                G190
                                (COND
                                  ((OR (ATOM #:G1489)
                                       (PROGN
                                         (LETT |rcf| (CAR #:G1489)
                                          |EQ;factorAndSplit;$L;38|)
                                         NIL))
                                   (GO G191)))
                                (SEQ (EXIT
                                      (LETT #:G1488
                                       (CONS
                                        (SPADCALL (QCAR |rcf|)
                                         (|spadConstant| $ 16)
                                         (QREFELT $ 17))
                                        #:G1488)
                                       |EQ;factorAndSplit;$L;38|)))
                                (LETT #:G1489 (CDR #:G1489)
                                      |EQ;factorAndSplit;$L;38|)
                                (GO G190) G191
                                (EXIT (NREVERSE0 #:G1488)))))))
             ((EQUAL (QREFELT $ 6) (|Polynomial| (|Integer|)))
              (SEQ (LETT |eq0| (SPADCALL |eq| (QREFELT $ 8))
                         |EQ;factorAndSplit;$L;38|)
                   (LETT |p| (SPADCALL |eq0| (QREFELT $ 9))
                         |EQ;factorAndSplit;$L;38|)
                   (EXIT (PROGN
                           (LETT #:G1490 NIL |EQ;factorAndSplit;$L;38|)
                           (SEQ (LETT |rcf| NIL
```

```
                                    |EQ;factorAndSplit;$L;38|)
                                  (LETT #:G1491
                                    (SPADCALL
                                     (SPADCALL |p| (QREFELT $ 80))
                                     (QREFELT $ 83))
                                    |EQ;factorAndSplit;$L;38|)
                                  G190
                                  (COND
                                    ((OR (ATOM #:G1491)
                                      (PROGN
                                        (LETT |rcf| (CAR #:G1491)
                                         |EQ;factorAndSplit;$L;38|)
                                        NIL))
                                     (GO G191)))
                                  (SEQ (EXIT
                                    (LETT #:G1490
                                     (CONS
                                      (SPADCALL (QCAR |rcf|)
                                       (|spadConstant| $ 16)
                                       (QREFELT $ 17))
                                      #:G1490)
                                     |EQ;factorAndSplit;$L;38|)))
                                  (LETT #:G1491 (CDR #:G1491)
                                    |EQ;factorAndSplit;$L;38|)
                                  (GO G190) G191
                                  (EXIT (NREVERSE0 #:G1490)))))))
            ('T (LIST |eq|)))))))

(DEFUN |EQ;differentiate;$S$;39| (|eq| |sym| $)
  (CONS (SPADCALL (SPADCALL |eq| (QREFELT $ 9)) |sym| (QREFELT $ 84))
        (SPADCALL (SPADCALL |eq| (QREFELT $ 21)) |sym| (QREFELT $ 84))))


(DEFUN |EQ;dimension;Cn;40| ($) (SPADCALL 2 (QREFELT $ 87)))

(DEFUN |EQ;/;3$;41| (|eq1| |eq2| $)
  (SPADCALL (SPADCALL (QCAR |eq1|) (QCAR |eq2|) (QREFELT $ 89))
      (SPADCALL (QCDR |eq1|) (QCDR |eq2|) (QREFELT $ 89))
      (QREFELT $ 20)))

(DEFUN |EQ;inv;2$;42| (|eq| $)
  (CONS (SPADCALL (SPADCALL |eq| (QREFELT $ 9)) (QREFELT $ 69))
        (SPADCALL (SPADCALL |eq| (QREFELT $ 21)) (QREFELT $ 69))))

(DEFUN |EQ;subst;3$;43| (|eq1| |eq2| $)
  (PROG (|eq3|)
    (RETURN
      (SEQ (LETT |eq3| |eq2| |EQ;subst;3$;43|)
           (EXIT (CONS (SPADCALL (SPADCALL |eq1| (QREFELT $ 9)) |eq3|
                       (QREFELT $ 92))
```

```
                    (SPADCALL (SPADCALL |eq1| (QREFELT $ 21)) |eq3|
                        (QREFELT $ 92))))))))

(DEFUN |Equation| (#:G1503)
 (PROG ()
  (RETURN
   (PROG (#:G1504)
    (RETURN
     (COND
      ((LETT #:G1504
        (|lassocShiftWithFunction|
          (LIST (|devaluate| #:G1503))
          (HGET |$ConstructorCache| '|Equation|)
          '|domainEqualList|)
        |Equation|)
       (|CDRwithIncrement| #:G1504))
      ('T
       (UNWIND-PROTECT
        (PROG1 (|Equation;| #:G1503)
         (LETT #:G1504 T |Equation|))
        (COND
         ((NOT #:G1504) (HREM |$ConstructorCache| '|Equation|)))))))))))

(DEFUN |Equation;| (|#1|)
  (PROG (DV$1 |dv$| $ #:G1502 #:G1501 #:G1500 #:G1499 #:G1498 |pv$|)
    (RETURN
      (PROGN
        (LETT DV$1 (|devaluate| |#1|) |Equation|)
        (LETT |dv$| (LIST '|Equation| DV$1) |Equation|)
        (LETT $ (GETREFV 98) |Equation|)
        (QSETREFV $ 0 |dv$|)
        (QSETREFV $ 3
            (LETT |pv$|
                  (|buildPredVector| 0 0
                      (LIST (|HasCategory| |#1| '(|Field|))
                            (|HasCategory| |#1| '(|SetCategory|))
                            (|HasCategory| |#1| '(|Ring|))
                            (|HasCategory| |#1|
                                '(|PartialDifferentialRing| (|Symbol|)))
                            (OR (|HasCategory| |#1|
                                    '(|PartialDifferentialRing|
                                      (|Symbol|)))
                                (|HasCategory| |#1| '(|Ring|)))
                            (|HasCategory| |#1| '(|Group|))
                            (|HasCategory| |#1|
                                (LIST '|InnerEvalable| '(|Symbol|)
                                      (|devaluate| |#1|)))
                            (AND (|HasCategory| |#1|
                                     (LIST '|Evalable|
                                      (|devaluate| |#1|)))
```

```
     (|HasCategory| |#1| '(|SetCategory|)))
(|HasCategory| |#1| '(|IntegralDomain|))
(|HasCategory| |#1| '(|ExpressionSpace|))
(OR (|HasCategory| |#1| '(|Field|))
    (|HasCategory| |#1| '(|Group|)))
(OR (|HasCategory| |#1| '(|Group|))
    (|HasCategory| |#1| '(|Ring|)))
(LETT #:G1502
      (|HasCategory| |#1|
          '(|CommutativeRing|))
      |Equation|)
(OR #:G1502 (|HasCategory| |#1| '(|Field|))
    (|HasCategory| |#1| '(|Ring|)))
(OR #:G1502
    (|HasCategory| |#1| '(|Field|)))
(LETT #:G1501
      (|HasCategory| |#1| '(|Monoid|))
      |Equation|)
(OR (|HasCategory| |#1| '(|Group|))
    #:G1501)
(LETT #:G1500
      (|HasCategory| |#1| '(|SemiGroup|))
      |Equation|)
(OR (|HasCategory| |#1| '(|Group|)) #:G1501
    #:G1500)
(LETT #:G1499
      (|HasCategory| |#1|
          '(|AbelianGroup|))
      |Equation|)
(OR (|HasCategory| |#1|
        '(|PartialDifferentialRing|
          (|Symbol|)))
    #:G1499 #:G1502
    (|HasCategory| |#1| '(|Field|))
    (|HasCategory| |#1| '(|Ring|)))
(OR #:G1499 #:G1501)
(LETT #:G1498
      (|HasCategory| |#1|
          '(|AbelianSemiGroup|))
      |Equation|)
(OR (|HasCategory| |#1|
        '(|PartialDifferentialRing|
          (|Symbol|)))
    #:G1499 #:G1498 #:G1502
    (|HasCategory| |#1| '(|Field|))
    (|HasCategory| |#1| '(|Ring|)))
(OR (|HasCategory| |#1|
        '(|PartialDifferentialRing|
          (|Symbol|)))
    #:G1499 #:G1498 #:G1502
```

```
                        (|HasCategory| |#1| '(|Field|))
                        (|HasCategory| |#1| '(|Group|)) #:G1501
                        (|HasCategory| |#1| '(|Ring|)) #:G1500
                        (|HasCategory| |#1| '(|SetCategory|)))))
          |Equation|))
(|haddProp| |$ConstructorCache| '|Equation| (LIST DV$1)
    (CONS 1 $))
(|stuffDomainSlots| $)
(QSETREFV $ 6 |#1|)
(QSETREFV $ 7 (|Record| (|:| |lhs| |#1|) (|:| |rhs| |#1|)))
(COND
  ((|testBitVector| |pv$| 9)
   (QSETREFV $ 19
       (CONS (|dispatchFunction| |EQ;factorAndSplit;$L;1|) $))))
(COND
  ((|testBitVector| |pv$| 7)
   (PROGN
     (QSETREFV $ 27
         (CONS (|dispatchFunction| |EQ;eval;$SS$;8|) $))
     (QSETREFV $ 31
         (CONS (|dispatchFunction| |EQ;eval;$LL$;9|) $)))))
(COND
  ((|HasCategory| |#1| (LIST '|Evalable| (|devaluate| |#1|)))
   (PROGN
     (QSETREFV $ 34
         (CONS (|dispatchFunction| |EQ;eval;3$;10|) $))
     (QSETREFV $ 37
         (CONS (|dispatchFunction| |EQ;eval;$L$;11|) $)))))
(COND
  ((|testBitVector| |pv$| 2)
   (PROGN
     (QSETREFV $ 40
         (CONS (|dispatchFunction| |EQ;=;2$B;12|) $))
     (QSETREFV $ 44
         (CONS (|dispatchFunction| |EQ;coerce;$Of;13|) $))
     (QSETREFV $ 45
         (CONS (|dispatchFunction| |EQ;coerce;$B;14|) $)))))
(COND
  ((|testBitVector| |pv$| 23)
   (PROGN
     (QSETREFV $ 47 (CONS (|dispatchFunction| |EQ;+;3$;15|) $))
     (QSETREFV $ 48
         (CONS (|dispatchFunction| |EQ;+;S2$;16|) $))
     (QSETREFV $ 49
         (CONS (|dispatchFunction| |EQ;+;$S$;17|) $)))))
(COND
  ((|testBitVector| |pv$| 20)
   (PROGN
     (QSETREFV $ 51 (CONS (|dispatchFunction| |EQ;-;2$;18|) $))
     (QSETREFV $ 53
```

```
          (CONS (|dispatchFunction| |EQ;-;S2$;19|) $))
      (QSETREFV $ 54
          (CONS (|dispatchFunction| |EQ;-;$S$;20|) $))
      (QSETREFV $ 57
          (CONS (|dispatchFunction| |EQ;leftZero;2$;21|) $))
      (QSETREFV $ 8
          (CONS (|dispatchFunction| |EQ;rightZero;2$;22|) $))
      (QSETREFV $ 55
          (CONS IDENTITY
                (FUNCALL (|dispatchFunction| |EQ;Zero;$;23|) $)))
      (QSETREFV $ 52 (CONS (|dispatchFunction| |EQ;-;3$;24|) $)))))
(COND
  ((|testBitVector| |pv$| 18)
   (PROGN
     (QSETREFV $ 59 (CONS (|dispatchFunction| |EQ;*;3$;25|) $))
     (QSETREFV $ 60
          (CONS (|dispatchFunction| |EQ;*;S2$;26|) $))
     (QSETREFV $ 60
          (CONS (|dispatchFunction| |EQ;*;S2$;27|) $))
     (QSETREFV $ 61
          (CONS (|dispatchFunction| |EQ;*;$S$;28|) $)))))
(COND
  ((|testBitVector| |pv$| 16)
   (PROGN
     (QSETREFV $ 63
          (CONS IDENTITY
                (FUNCALL (|dispatchFunction| |EQ;One;$;29|) $)))
     (QSETREFV $ 66
          (CONS (|dispatchFunction| |EQ;recip;$U;30|) $))
     (QSETREFV $ 67
          (CONS (|dispatchFunction| |EQ;leftOne;$U;31|) $))
     (QSETREFV $ 68
          (CONS (|dispatchFunction| |EQ;rightOne;$U;32|) $)))))
(COND
  ((|testBitVector| |pv$| 6)
   (PROGN
     (QSETREFV $ 70
          (CONS (|dispatchFunction| |EQ;inv;2$;33|) $))
     (QSETREFV $ 67
          (CONS (|dispatchFunction| |EQ;leftOne;$U;34|) $))
     (QSETREFV $ 68
          (CONS (|dispatchFunction| |EQ;rightOne;$U;35|) $)))))
(COND
  ((|testBitVector| |pv$| 3)
   (PROGN
     (QSETREFV $ 73
          (CONS (|dispatchFunction| |EQ;characteristic;Nni;36|)
                $))
     (QSETREFV $ 76
          (CONS (|dispatchFunction| |EQ;*;I2$;37|) $)))))
```

```
        (COND
          ((|testBitVector| |pv$| 9)
           (QSETREFV $ 19
               (CONS (|dispatchFunction| |EQ;factorAndSplit;$L;38|) $))))
        (COND
          ((|testBitVector| |pv$| 4)
           (QSETREFV $ 85
               (CONS (|dispatchFunction| |EQ;differentiate;$S$;39|) $))))
        (COND
          ((|testBitVector| |pv$| 1)
           (PROGN
             (QSETREFV $ 88
                 (CONS (|dispatchFunction| |EQ;dimension;Cn;40|) $))
             (QSETREFV $ 90 (CONS (|dispatchFunction| |EQ;/;3$;41|) $))
             (QSETREFV $ 70
                 (CONS (|dispatchFunction| |EQ;inv;2$;42|) $)))))
        (COND
          ((|testBitVector| |pv$| 10)
           (QSETREFV $ 93
               (CONS (|dispatchFunction| |EQ;subst;3$;43|) $))))
        $))))

(MAKEPROP '|Equation| '|infovec|
    (LIST '#(NIL NIL NIL NIL NIL NIL (|local| |#1|) '|Rep|
             (0 . |rightZero|) |EQ;lhs;$S;4| (|Factored| $)
             (5 . |factor|)
             (|Record| (|:| |factor| 6) (|:| |exponent| 74))
             (|List| 12) (|Factored| 6) (10 . |factors|) (15 . |Zero|)
             |EQ;equation;2S$;3| (|List| $) (19 . |factorAndSplit|)
             |EQ;=;2S$;2| |EQ;rhs;$S;5| |EQ;swap;2$;6| (|Mapping| 6 6)
             |EQ;map;M2$;7| (|Symbol|) (24 . |eval|) (31 . |eval|)
             (|List| 25) (|List| 6) (38 . |eval|) (45 . |eval|)
             (|Equation| 6) (52 . |eval|) (58 . |eval|) (|List| 32)
             (64 . |eval|) (70 . |eval|) (|Boolean|) (76 . =) (82 . =)
             (|OutputForm|) (88 . |coerce|) (93 . =) (99 . |coerce|)
             (104 . |coerce|) (109 . +) (115 . +) (121 . +) (127 . +)
             (133 . -) (138 . -) (143 . -) (149 . -) (155 . -)
             (161 . |Zero|) (165 . -) (171 . |leftZero|) (176 . *)
             (182 . *) (188 . *) (194 . *) (200 . |One|) (204 . |One|)
             (|Union| $ '"failed") (208 . |recip|) (213 . |recip|)
             (218 . |leftOne|) (223 . |rightOne|) (228 . |inv|)
             (233 . |inv|) (|NonNegativeInteger|)
             (238 . |characteristic|) (242 . |characteristic|)
             (|Integer|) (246 . |coerce|) (251 . *) (|Factored| 78)
             (|Polynomial| 74)
             (|MultivariateFactorize| 25 (|IndexedExponents| 25) 74 78)
             (257 . |factor|)
             (|Record| (|:| |factor| 78) (|:| |exponent| 74))
             (|List| 81) (262 . |factors|) (267 . |differentiate|)
             (273 . |differentiate|) (|CardinalNumber|)
```

```
   (279 . |coerce|) (284 . |dimension|) (288 . /) (294 . /)
   (|Equation| $) (300 . |subst|) (306 . |subst|)
   (|PositiveInteger|) (|List| 71) (|SingleInteger|)
   (|String|))
'#(~= 312 |zero?| 318 |swap| 323 |subtractIfCan| 328 |subst|
   334 |sample| 340 |rightZero| 344 |rightOne| 349 |rhs| 354
   |recip| 359 |one?| 364 |map| 369 |lhs| 375 |leftZero| 380
   |leftOne| 385 |latex| 390 |inv| 395 |hash| 400
   |factorAndSplit| 405 |eval| 410 |equation| 436 |dimension|
   442 |differentiate| 446 |conjugate| 472 |commutator| 478
   |coerce| 484 |characteristic| 499 ^ 503 |Zero| 521 |One|
   525 D 529 = 555 / 567 - 579 + 602 ** 620 * 638)
'((|unitsKnown| . 12) (|rightUnitary| . 3)
  (|leftUnitary| . 3))
(CONS (|makeByteWordVec2| 25
          '(1 15 4 14 5 14 3 5 3 21 21 6 21 17 24 19 25 0 2
            25 2 7))
      (CONS '#(|VectorSpace&| |Module&|
               |PartialDifferentialRing&| NIL |Ring&| NIL NIL
               NIL NIL |AbelianGroup&| NIL |Group&|
               |AbelianMonoid&| |Monoid&| |AbelianSemiGroup&|
               |SemiGroup&| |SetCategory&| NIL NIL
               |BasicType&| NIL |InnerEvalable&|)
            (CONS '#((|VectorSpace| 6) (|Module| 6)
                     (|PartialDifferentialRing| 25)
                     (|BiModule| 6 6) (|Ring|)
                     (|LeftModule| 6) (|RightModule| 6)
                     (|Rng|) (|LeftModule| $$)
                     (|AbelianGroup|)
                     (|CancellationAbelianMonoid|) (|Group|)
                     (|AbelianMonoid|) (|Monoid|)
                     (|AbelianSemiGroup|) (|SemiGroup|)
                     (|SetCategory|) (|Type|)
                     (|CoercibleTo| 41) (|BasicType|)
                     (|CoercibleTo| 38)
                     (|InnerEvalable| 25 6))
                  (|makeByteWordVec2| 97
                      '(1 0 0 0 8 1 6 10 0 11 1 14 13 0 15 0
                        6 0 16 1 0 18 0 19 3 6 0 0 25 6 26 3
                        0 0 0 25 6 27 3 6 0 0 28 29 30 3 0 0
                        0 28 29 31 2 6 0 0 32 33 2 0 0 0 0 34
                        2 6 0 0 35 36 2 0 0 0 18 37 2 6 38 0
                        0 39 2 0 38 0 0 40 1 6 41 0 42 2 41 0
                        0 0 43 1 0 41 0 44 1 0 38 0 45 2 6 0
                        0 0 46 2 0 0 0 0 47 2 0 0 6 0 48 2 0
                        0 0 6 49 1 6 0 0 50 1 0 0 0 51 2 0 0
                        0 0 52 2 0 0 6 0 53 2 0 0 0 6 54 0 0
                        0 55 2 6 0 0 0 56 1 0 0 0 57 2 6 0 0
                        0 58 2 0 0 0 0 59 2 0 0 6 0 60 2 0 0
                        0 6 61 0 6 0 62 0 0 0 63 1 6 64 0 65
```

```
                                1 0 64 0 66 1 0 64 0 67 1 0 64 0 68 1
                                6 0 0 69 1 0 0 0 70 0 6 71 72 0 0 71
                                73 1 6 0 74 75 2 0 0 74 0 76 1 79 77
                                78 80 1 77 82 0 83 2 6 0 0 25 84 2 0
                                0 0 25 85 1 86 0 71 87 0 0 86 88 2 6
                                0 0 0 89 2 0 0 0 0 90 2 6 0 0 91 92 2
                                0 0 0 0 93 2 2 38 0 0 1 1 20 38 0 1 1
                                0 0 0 22 2 20 64 0 0 1 2 10 0 0 0 93
                                0 22 0 1 1 20 0 0 8 1 16 64 0 68 1 0
                                6 0 21 1 16 64 0 66 1 16 38 0 1 2 0 0
                                23 0 24 1 0 6 0 9 1 20 0 0 57 1 16 64
                                0 67 1 2 97 0 1 1 11 0 0 70 1 2 96 0
                                1 1 9 18 0 19 2 8 0 0 0 34 2 8 0 0 18
                                37 3 7 0 0 25 6 27 3 7 0 0 28 29 31 2
                                0 0 6 6 17 0 1 86 88 2 4 0 0 28 1 2 4
                                0 0 25 85 3 4 0 0 28 95 1 3 4 0 0 25
                                71 1 2 6 0 0 0 1 2 6 0 0 0 1 1 3 0 74
                                1 1 2 41 0 44 1 2 38 0 45 0 3 71 73 2
                                6 0 0 74 1 2 16 0 0 71 1 2 18 0 0 94
                                1 0 20 0 55 0 16 0 63 2 4 0 0 28 1 2
                                4 0 0 25 1 3 4 0 0 28 95 1 3 4 0 0 25
                                71 1 2 2 38 0 0 40 2 0 0 6 6 20 2 11
                                0 0 0 90 2 1 0 0 6 1 1 20 0 0 51 2 20
                                0 0 0 52 2 20 0 6 0 53 2 20 0 0 6 54
                                2 23 0 0 0 47 2 23 0 6 0 48 2 23 0 0
                                6 49 2 6 0 0 74 1 2 16 0 0 71 1 2 18
                                0 0 94 1 2 20 0 71 0 1 2 20 0 74 0 76
                                2 23 0 94 0 1 2 18 0 0 0 59 2 18 0 0
                                6 61 2 18 0 6 0 60)))))
        '|lookupComplete|))
```

1.4 The code.o file

The Spad compiler translates the Spad language into Common Lisp. It eventually invokes the Common Lisp "compile-file" command to output files in binary. Depending on the lisp system this filename can vary (e.g "code.fasl"). The details of how these are used depends on the Common Lisp in use.

By default, Axiom uses Gnu Common Lisp (GCL), which generates ".o" files.

1.5 The info file

```
((* (($ $ $) (|arguments| (|eq2| . $) (|eq1| . $)) (S (* S S S))
     ($ (= $ S S)))
    (($ $ S) (|arguments| (|l| . S) (|eqn| . $)) (S (* S S S))
     ($ (= $ S S)))
```

```
   (($ #0=(|Integer|) $) (|arguments| (|i| . #0#) (|eq| . $))
    (S (|coerce| S (|Integer|))) ($ (* $ S $)))
   (($ S $) (|arguments| (|l| . S) (|eqn| . $)) (S (* S S S))
    ($ (= $ S S))))
(+ (($ $ $) (|arguments| (|eq2| . $) (|eq1| . $)) (S (+ S S S))
    ($ (= $ S S)))
   (($ $ S) (|arguments| (|s| . S) (|eq1| . $)) ($ (+ $ $ $)))
   (($ S $) (|arguments| (|s| . S) (|eq2| . $)) ($ (+ $ $ $))))
(- (($ $ $) (|arguments| (|eq2| . $) (|eq1| . $)) (S (- S S S))
    ($ (= $ S S)))
   (($ $ S) (|arguments| (|s| . S) (|eq1| . $)) ($ (- $ $ $)))
   (($ $) (|arguments| (|eq| . $)) (S (- S S))
    ($ (|rhs| S $) (|lhs| S $) (= $ S S)))
   (($ S $) (|arguments| (|s| . S) (|eq2| . $)) ($ (- $ $ $))))
(/ (($ $ $) (|arguments| (|eq2| . $) (|eq1| . $)) (S (/ S S S))
    ($ (= $ S S))))
(= (($ S S) (|arguments| (|r| . S) (|l| . S)))
   ((((|Boolean|) $ $) ((|Boolean|) (|false| (|Boolean|)))
    (|locals| (#:G1393 |Boolean|))
    (|arguments| (|eq2| . $) (|eq1| . $)) (S (= (|Boolean|) S S))))
(|One| (($) (S (|One| S)) ($ (|equation| $ S S))))
(|Zero| (($) (S (|Zero| S)) ($ (|equation| $ S S))))
(|characteristic|
    ((((|NonNegativeInteger|))
     (S (|characteristic| (|NonNegativeInteger|)))))
(|coerce|
    ((((|Boolean|) $) (|arguments| (|eqn| . $))
     (S (= (|Boolean|) S S)))
    ((((|OutputForm|) $)
     ((|OutputForm|) (= (|OutputForm|) (|OutputForm|) (|OutputForm|)))
     (|arguments| (|eqn| . $)) (S (|coerce| (|OutputForm|) S))))
(|constructor|
    (NIL (|locals|
             (|Rep| |Join| (|SetCategory|)
                    (CATEGORY |domain|
                        (SIGNATURE |construct|
                            ((|Record| (|:| |lhs| S) (|:| |rhs| S)) S
                             S))
                        (SIGNATURE |coerce|
                            ((|OutputForm|)
                             (|Record| (|:| |lhs| S) (|:| |rhs| S))))
                        (SIGNATURE |elt|
                            (S (|Record| (|:| |lhs| S) (|:| |rhs| S))
                               "lhs"))
                        (SIGNATURE |elt|
                            (S (|Record| (|:| |lhs| S) (|:| |rhs| S))
                               "rhs"))
                        (SIGNATURE |setelt|
                            (S (|Record| (|:| |lhs| S) (|:| |rhs| S))
                               "lhs" S))
```

```
                        (SIGNATURE |setelt|
                            (S (|Record| (|:| |lhs| S) (|:| |rhs| S))
                               "rhs" S))
                        (SIGNATURE |copy|
                            ((|Record| (|:| |lhs| S) (|:| |rhs| S))
                             (|Record| (|:| |lhs| S) (|:| |rhs| S)))))))))
(|differentiate|
    (($ $ #1=(|Symbol|)) (|arguments| (|sym| . #1#) (|eq| . $))
     (S (|differentiate| S S (|Symbol|))) ($ (|rhs| S $) (|lhs| S $))))
(|dimension|
    ((#2=(|CardinalNumber|))
     (#2# (|coerce| (|CardinalNumber|) (|NonNegativeInteger|)))))
(|equation| (($ S S) (|arguments| (|r| . S) (|l| . S))))
(|eval| (($ $ $) (|arguments| (|eqn2| . $) (|eqn1| . $))
         (S (|eval| S S (|Equation| S))) ($ (= $ S S)))
        (($ $ #3=(|List| $))
         (|arguments| (|leqn2| . #3#) (|eqn1| . $))
         (S (|eval| S S (|List| (|Equation| S)))) ($ (= $ S S)))
        (($ $ #4=(|List| #5=(|Symbol|)) #6=(|List| S))
         (|arguments| (|lx| . #6#) (|ls| . #4#) (|eqn| . $))
         (S (|eval| S S (|List| (|Symbol|)) (|List| S)))
         ($ (= $ S S)))
        (($ $ #5# S) (|arguments| (|x| . S) (|s| . #5#) (|eqn| . $))
         (S (|eval| S S (|Symbol|) S)) ($ (= $ S S))))
(|factorAndSplit|
    ((((|List| $) $)
     ((|MultivariateFactorize| (|Symbol|)
          (|IndexedExponents| (|Symbol|)) (|Integer|)
          (|Polynomial| (|Integer|)))
      (|factor| (|Factored| (|Polynomial| (|Integer|)))
          (|Polynomial| (|Integer|))))
     ((|Factored| S)
      (|factors|
          (|List| (|Record| (|:| |factor| S)
                      (|:| |exponent| (|Integer|))))
          (|Factored| S)))
     ((|Factored| (|Polynomial| (|Integer|)))
      (|factors|
          (|List| (|Record| (|:| |factor| (|Polynomial| (|Integer|)))
                      (|:| |exponent| (|Integer|))))
          (|Factored| (|Polynomial| (|Integer|)))))
     (|locals| (|p| |Polynomial| (|Integer|)) (|eq0| . $))
     (|arguments| (|eq| . $))
     (S (|factor| (|Factored| S) S) (|Zero| S))
     ($ (|rightZero| $ $) (|lhs| S $) (|equation| $ S S))))
(|inv| (($ $) (|arguments| (|eq| . $)) (S (|inv| S S))
        ($ (|rhs| S $) (|lhs| S $))))
(|leftOne|
    ((((|Union| $ "failed") $) (|locals| (|re| |Union| S "failed"))
     (|arguments| (|eq| . $))
```

```
      (S (|recip| (|Union| S "failed") S) (|inv| S S) (|One| S)
         (* S S S))
      ($ (|rhs| S $) (|lhs| S $) (|One| $) (= $ S S))))
 (|leftZero|
     (($ $) (|arguments| (|eq| . $)) (S (|Zero| S) (- S S S))
      ($ (|rhs| S $) (|lhs| S $) (|Zero| $) (= $ S S))))
 (|lhs| ((S $) (|arguments| (|eqn| . $))))
 (|map| (($ #7=(|Mapping| S S) $)
         (|arguments| (|fn| . #7#) (|eqn| . $)) ($ (|equation| $ S S))))
 (|recip| ((((|Union| $ "failed") $)
            (|locals| (|rh| |Union| S "failed")
                (|lh| |Union| S "failed"))
            (|arguments| (|eq| . $))
            (S (|recip| (|Union| S "failed") S))
            ($ (|rhs| S $) (|lhs| S $))))
 (|rhs| ((S $) (|arguments| (|eqn| . $))))
 (|rightOne|
     ((((|Union| $ "failed") $) (|locals| (|re| |Union| S "failed"))
      (|arguments| (|eq| . $))
      (S (|recip| (|Union| S "failed") S) (|inv| S S) (|One| S)
         (* S S S))
      ($ (|rhs| S $) (|lhs| S $) (= $ S S))))
 (|rightZero|
     (($ $) (|arguments| (|eq| . $)) (S (|Zero| S) (- S S S))
      ($ (|rhs| S $) (|lhs| S $) (= $ S S))))
 (|subst| (($ $ $) (|locals| (|eq3| |Equation| S))
           (|arguments| (|eq2| . $) (|eq1| . $))
           (S (|subst| S S (|Equation| S)))
           ($ (|rhs| S $) (|lhs| S $))))
 (|swap| (($ $) (|arguments| (|eqn| . $)) ($ (|rhs| S $) (|lhs| S $)))))
```

1.6 The EQ.fn file

```
(in-package 'compiler)(init-fn)
(ADD-FN-DATA '(
#S(FN NAME BOOT::|EQ;*;S2$;26| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;rightOne;$U;32| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES
      (BOOT::|spadConstant| VMLISP:QCDR CONS VMLISP:QCAR EQL
          BOOT::QEQCAR COND VMLISP:EXIT CDR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL BOOT::LETT VMLISP:SEQ RETURN)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (BOOT::|spadConstant| VMLISP:QCDR VMLISP:QCAR BOOT::QEQCAR COND
```

```
          VMLISP:EXIT VMLISP:QREFELT BOOT:SPADCALL BOOT::LETT
          VMLISP:SEQ RETURN))
#S(FN NAME BOOT::|EQ;lhs;$S;4| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CAR VMLISP:QCAR) RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT
      NIL MACROS (VMLISP:QCAR))
#S(FN NAME BOOT::|EQ;+;3$;15| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;dimension;Cn;40| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;rightZero;2$;22| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES
      (BOOT::|spadConstant| CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (BOOT::|spadConstant| VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;coerce;$Of;13| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;One;$;29| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (CDR BOOT::|spadConstant| CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T) NO-EMIT NIL MACROS
      (BOOT::|spadConstant| VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;inv;2$;42| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL CONS)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;-;$S$;20| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CDR CONS CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;=;2$B;12| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL COND)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL COND))
#S(FN NAME BOOT::|EQ;/;3$;41| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
```

```
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;recip;$U;30| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR LIST* CONS VMLISP:QCAR EQL BOOT::QEQCAR COND
          VMLISP:EXIT CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL
          BOOT::LETT VMLISP:SEQ RETURN)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR BOOT::QEQCAR COND VMLISP:EXIT
          VMLISP:QREFELT BOOT:SPADCALL BOOT::LETT VMLISP:SEQ RETURN))
#S(FN NAME BOOT::|EQ;-;3$;24| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;eval;$L$;11| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;leftZero;2$;21| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES
      (CDR BOOT::|spadConstant| CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (BOOT::|spadConstant| VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;*;S2$;27| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;*;I2$;37| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL) RETURN-TYPE
      NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;eval;3$;10| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;eval;$SS$;8| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;factorAndSplit;$L;38| DEF DEFUN VALUE-TYPE T
      FUN-VALUES NIL CALLEES
```

```
      (BOOT:|Integer| BOOT:|Polynomial| EQUAL BOOT:NREVERSE0
          BOOT::|spadConstant| VMLISP:QCAR CONS ATOM VMLISP:EXIT CDR
          CAR BOOT:SPADCALL BOOT::LETT BOOT::|devaluate| LIST SVREF
          VMLISP:QREFELT BOOT::|HasSignature| COND VMLISP:SEQ RETURN)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (BOOT::|spadConstant| VMLISP:QCAR VMLISP:EXIT BOOT:SPADCALL
          BOOT::LETT VMLISP:QREFELT COND VMLISP:SEQ RETURN))
#S(FN NAME BOOT::|EQ;differentiate;$S$;39| DEF DEFUN VALUE-TYPE T
      FUN-VALUES NIL CALLEES
      (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL CONS) RETURN-TYPE NIL
      ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;eval;$LL$;9| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;leftOne;$U;34| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES
      (CDR BOOT::|spadConstant| CAR SVREF VMLISP:QREFELT BOOT:SPADCALL
           CONS)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (BOOT::|spadConstant| VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;map;M2$;7| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;-;S2$;19| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CDR CONS CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;equation;2S$;3| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES (CONS) RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL
      MACROS NIL)
#S(FN NAME BOOT::|EQ;+;$S$;17| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CDR CONS CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;factorAndSplit;$L;1| DEF DEFUN VALUE-TYPE T
      FUN-VALUES NIL CALLEES
      (BOOT:NREVERSE0 BOOT::|spadConstant| VMLISP:QCAR CONS ATOM
          VMLISP:EXIT CDR CAR BOOT:SPADCALL BOOT::LETT
          BOOT::|devaluate| LIST SVREF VMLISP:QREFELT
          BOOT::|HasSignature| COND VMLISP:SEQ RETURN)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (BOOT::|spadConstant| VMLISP:QCAR VMLISP:EXIT BOOT:SPADCALL
          BOOT::LETT VMLISP:QREFELT COND VMLISP:SEQ RETURN))
```

```
#S(FN NAME BOOT::|EQ;*;3$;25| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;Zero;$;23| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (CDR BOOT::|spadConstant| CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T) NO-EMIT NIL MACROS
      (BOOT::|spadConstant| VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;characteristic;Nni;36| DEF DEFUN VALUE-TYPE T
      FUN-VALUES NIL CALLEES
      (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL) RETURN-TYPE NIL
      ARG-TYPES (T) NO-EMIT NIL MACROS (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;leftOne;$U;31| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES
      (VMLISP:QCDR BOOT::|spadConstant| CONS VMLISP:QCAR EQL
          BOOT::QEQCAR COND VMLISP:EXIT CDR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL BOOT::LETT VMLISP:SEQ RETURN)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR BOOT::|spadConstant| VMLISP:QCAR BOOT::QEQCAR COND
          VMLISP:EXIT VMLISP:QREFELT BOOT:SPADCALL BOOT::LETT
          VMLISP:SEQ RETURN))
#S(FN NAME BOOT::|EQ;swap;2$;6| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL CONS)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;-;2$;18| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL) RETURN-TYPE
      NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;subst;3$;43| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL CONS VMLISP:EXIT
           BOOT::LETT VMLISP:SEQ RETURN)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL VMLISP:EXIT BOOT::LETT VMLISP:SEQ
          RETURN))
#S(FN NAME BOOT::|EQ;=;2S$;2| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CONS) RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL
      MACROS NIL)
#S(FN NAME BOOT::|EQ;*;$S$;28| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;+;S2$;16| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CDR CONS CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
```

```
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|Equation;| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (BOOT::|EQ;One;$;29| BOOT::|EQ;Zero;$;23|
          BOOT::|dispatchFunction| BOOT::|testBitVector| COND
          BOOT::|Record0| BOOT::|Record| BOOT::|stuffDomainSlots| CONS
          BOOT::|haddProp| BOOT::|HasCategory| BOOT::|buildPredVector|
          SYSTEM:SVSET SETF VMLISP:QSETREFV VMLISP:GETREFV LIST
          BOOT::|devaluate| BOOT::LETT RETURN)
      RETURN-TYPE NIL ARG-TYPES (T) NO-EMIT NIL MACROS
      (BOOT::|dispatchFunction| COND BOOT::|Record| SETF
          VMLISP:QSETREFV BOOT::LETT RETURN))
#S(FN NAME BOOT::|EQ;coerce;$B;14| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES
      (CDR VMLISP:QCDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
           BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;rhs;$S;5| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CDR VMLISP:QCDR) RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT
      NIL MACROS (VMLISP:QCDR))
#S(FN NAME OTHER-FORM DEF NIL VALUE-TYPE NIL FUN-VALUES NIL CALLEES NIL
      RETURN-TYPE NIL ARG-TYPES NIL NO-EMIT NIL MACROS NIL)
#S(FN NAME BOOT::|EQ;inv;2$;33| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL CONS)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;rightOne;$U;35| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES
      (BOOT::|spadConstant| CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL
          CONS)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (BOOT::|spadConstant| VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|Equation| DEF DEFUN VALUE-TYPE T FUN-VALUES
      (SINGLE-VALUE) CALLEES
      (REMHASH VMLISP:HREM BOOT::|Equation;| PROG1
               BOOT::|CDRwithIncrement| GETHASH VMLISP:HGET
               BOOT::|devaluate| LIST BOOT::|lassocShiftWithFunction|
               BOOT::LETT COND RETURN)
      RETURN-TYPE NIL ARG-TYPES (T) NO-EMIT NIL MACROS
      (VMLISP:HREM PROG1 VMLISP:HGET BOOT::LETT COND RETURN)) ))
```

1.7 The index.kaf file

Each constructor (e.g. EQ) had one library directory (e.g. EQ.nrlib). This directory contained a random access file called the index.kaf file. These files contain runtime information such as the operationAlist and the Constructor-

Modemap. At system build time we merge all of these .nrlib/index.kaf files into one database, INTERP.daase. Requests to get information from this database are cached so that multiple references do not cause additional disk i/o.

Before getting into the contents, we need to understand the format of an index.kaf file. The kaf file is a random access file, originally used as a database. In the current system we make a pass to combine these files at build time to construct the various daase files.

This is just a file of lisp objects, one after another, in (read) format.

A kaf file starts with an integer, in this case, 35695. This integer gives the byte offset to the index. Due to the way the file is constructed, the index is at the end of the file. To read a kaf file, first read the integer, then seek to that location in the file, and do a (read). This will return the index, in this case:

```
(("slot1Info" 0 32444)
 ("documentation" 0 29640)
 ("ancestors" 0 28691)
 ("parents" 0 28077)
 ("abbreviation" 0 28074)
 ("predicates" 0 25442)
 ("attributes" 0 25304)
 ("signaturesAndLocals" 0 23933)
 ("superDomain" 0 NIL)
 ("operationAlist" 0 20053)
 ("modemaps" 0 17216)
 ("sourceFile" 0 17179)
 ("constructorCategory" 0 15220)
 ("constructorModemap" 0 13215)
 ("constructorKind" 0 13206)
 ("constructorForm" 0 13191)
 ("compilerInfo" 0 4433)
 ("loadTimeStuff" 0 20))
```

This is a list of triples. The first item in each triple is a string that is used as a lookup key (e.g. "operationAlist"). The second element is no longer used. The third element is the byte offset from the beginning of the file.

So to read the "operationAlist" from this file you would:

1. open the index.kaf file
2. (read) the integer
3. (seek) to the integer offset from the beginning of the file
4. (read) the index of triples
5. find the keyword (e.g. "operationAlist") triple
6. select the third element, an integer

7. (seek) to the integer offset from the beginning of the file
8. (read) the "operationAlist"

Note that the information below has been reformatted to fit this document. In order to save space the index.kaf file is does not use prettyprint since it is normally only read by machine.

1.7.1 The index offset byte

```
35695
```

1.7.2 The "loadTimeStuff"

```
(MAKEPROP '|Equation| '|infovec|
    (LIST '#(NIL NIL NIL NIL NIL NIL (|local| |#1|) '|Rep|
             (0 . |rightZero|) |EQ;lhs;$S;4| (|Factored| $)
             (5 . |factor|)
             (|Record| (|:| |factor| 6) (|:| |exponent| 74))
             (|List| 12) (|Factored| 6) (10 . |factors|) (15 . |Zero|)
             |EQ;equation;2S$;3| (|List| $) (19 . |factorAndSplit|)
             |EQ;=;2S$;2| |EQ;rhs;$S;5| |EQ;swap;2$;6| (|Mapping| 6 6)
             |EQ;map;M2$;7| (|Symbol|) (24 . |eval|) (31 . |eval|)
             (|List| 25) (|List| 6) (38 . |eval|) (45 . |eval|)
             (|Equation| 6) (52 . |eval|) (58 . |eval|) (|List| 32)
             (64 . |eval|) (70 . |eval|) (|Boolean|) (76 . =) (82 . =)
             (|OutputForm|) (88 . |coerce|) (93 . =) (99 . |coerce|)
             (104 . |coerce|) (109 . +) (115 . +) (121 . +) (127 . +)
             (133 . -) (138 . -) (143 . -) (149 . -) (155 . -)
             (161 . |Zero|) (165 . -) (171 . |leftZero|) (176 . *)
             (182 . *) (188 . *) (194 . *) (200 . |One|) (204 . |One|)
             (|Union| $ '"failed") (208 . |recip|) (213 . |recip|)
             (218 . |leftOne|) (223 . |rightOne|) (228 . |inv|)
             (233 . |inv|) (|NonNegativeInteger|)
             (238 . |characteristic|) (242 . |characteristic|)
             (|Integer|) (246 . |coerce|) (251 . *) (|Factored| 78)
             (|Polynomial| 74)
             (|MultivariateFactorize| 25 (|IndexedExponents| 25) 74 78)
             (257 . |factor|)
             (|Record| (|:| |factor| 78) (|:| |exponent| 74))
             (|List| 81) (262 . |factors|) (267 . |differentiate|)
             (273 . |differentiate|) (|CardinalNumber|)
             (279 . |coerce|) (284 . |dimension|) (288 . /) (294 . /)
             (|Equation| $) (300 . |subst|) (306 . |subst|)
             (|PositiveInteger|) (|List| 71) (|SingleInteger|)
             (|String|))
          '#(~= 312 |zero?| 318 |swap| 323 |subtractIfCan| 328 |subst|
             334 |sample| 340 |rightZero| 344 |rightOne| 349 |rhs| 354
```

```
   |recip| 359 |one?| 364 |map| 369 |lhs| 375 |leftZero| 380
   |leftOne| 385 |latex| 390 |inv| 395 |hash| 400
   |factorAndSplit| 405 |eval| 410 |equation| 436 |dimension|
   442 |differentiate| 446 |conjugate| 472 |commutator| 478
   |coerce| 484 |characteristic| 499 ^ 503 |Zero| 521 |One|
   525 D 529 = 555 / 567 - 579 + 602 ** 620 * 638)
'(((|unitsKnown| . 12) (|rightUnitary| . 3)
  (|leftUnitary| . 3))
(CONS (|makeByteWordVec2| 25
         '(1 15 4 14 5 14 3 5 3 21 21 6 21 17 24 19 25 0 2
           25 2 7))
      (CONS '#(|VectorSpace&| |Module&|
               |PartialDifferentialRing&| NIL |Ring&| NIL NIL
               NIL NIL |AbelianGroup&| NIL |Group&|
               |AbelianMonoid&| |Monoid&| |AbelianSemiGroup&|
               |SemiGroup&| |SetCategory&| NIL NIL
               |BasicType&| NIL |InnerEvalable&|)
            (CONS '#((|VectorSpace| 6) (|Module| 6)
                     (|PartialDifferentialRing| 25)
                     (|BiModule| 6 6) (|Ring|)
                     (|LeftModule| 6) (|RightModule| 6)
                     (|Rng|) (|LeftModule| $$)
                     (|AbelianGroup|)
                     (|CancellationAbelianMonoid|) (|Group|)
                     (|AbelianMonoid|) (|Monoid|)
                     (|AbelianSemiGroup|) (|SemiGroup|)
                     (|SetCategory|) (|Type|)
                     (|CoercibleTo| 41) (|BasicType|)
                     (|CoercibleTo| 38)
                     (|InnerEvalable| 25 6))
                  (|makeByteWordVec2| 97
                      '(1 0 0 0 8 1 6 10 0 11 1 14 13 0 15 0
                        6 0 16 1 0 18 0 19 3 6 0 0 25 6 26 3
                        0 0 0 25 6 27 3 6 0 0 28 29 30 3 0 0
                        0 28 29 31 2 6 0 0 32 33 2 0 0 0 0 34
                        2 6 0 0 35 36 2 0 0 0 18 37 2 6 38 0
                        0 39 2 0 38 0 0 40 1 6 41 0 42 2 41 0
                        0 0 43 1 0 41 0 44 1 0 38 0 45 2 6 0
                        0 0 46 2 0 0 0 0 47 2 0 0 6 0 48 2 0
                        0 0 6 49 1 6 0 0 50 1 0 0 0 51 2 0 0
                        0 0 52 2 0 0 6 0 53 2 0 0 0 6 54 0 0
                        0 55 2 6 0 0 0 56 1 0 0 0 57 2 6 0 0
                        0 58 2 0 0 0 0 59 2 0 0 6 0 60 2 0 0
                        0 6 61 0 6 0 62 0 0 0 63 1 6 64 0 65
                        1 0 64 0 66 1 0 64 0 67 1 0 64 0 68 1
                        6 0 0 69 1 0 0 0 70 0 6 71 72 0 0 71
                        73 1 6 0 74 75 2 0 0 74 0 76 1 79 77
                        78 80 1 77 82 0 83 2 6 0 0 25 84 2 0
                        0 0 25 85 1 86 0 71 87 0 0 86 88 2 6
                        0 0 0 89 2 0 0 0 0 90 2 6 0 0 91 92 2
```

```
                                0 0 0 0 93 2 2 38 0 0 1 1 20 38 0 1 1
                                0 0 0 22 2 20 64 0 0 1 2 10 0 0 0 93
                                0 22 0 1 1 20 0 0 8 1 16 64 0 68 1 0
                                6 0 21 1 16 64 0 66 1 16 38 0 1 2 0 0
                                23 0 24 1 0 6 0 9 1 20 0 0 57 1 16 64
                                0 67 1 2 97 0 1 1 11 0 0 70 1 2 96 0
                                1 1 9 18 0 19 2 8 0 0 0 34 2 8 0 0 18
                                37 3 7 0 0 25 6 27 3 7 0 0 28 29 31 2
                                0 0 6 6 17 0 1 86 88 2 4 0 0 28 1 2 4
                                0 0 25 85 3 4 0 0 28 95 1 3 4 0 0 25
                                71 1 2 6 0 0 0 1 2 6 0 0 0 1 1 3 0 74
                                1 1 2 41 0 44 1 2 38 0 45 0 3 71 73 2
                                6 0 0 74 1 2 16 0 0 71 1 2 18 0 0 94
                                1 0 20 0 55 0 16 0 63 2 4 0 0 28 1 2
                                4 0 0 25 1 3 4 0 0 28 95 1 3 4 0 0 25
                                71 1 2 2 38 0 0 40 2 0 0 6 6 20 2 11
                                0 0 0 90 2 1 0 0 6 1 1 20 0 0 51 2 20
                                0 0 0 52 2 20 0 6 0 53 2 20 0 0 6 54
                                2 23 0 0 0 47 2 23 0 6 0 48 2 23 0 0
                                6 49 2 6 0 0 74 1 2 16 0 0 71 1 2 18
                                0 0 94 1 2 20 0 71 0 1 2 20 0 74 0 76
                                2 23 0 94 0 1 2 18 0 0 0 59 2 18 0 0
                                6 61 2 18 0 6 0 60)))))
        '|lookupComplete|))
```

1.7.3 The "compilerInfo"

```
(SETQ |$CategoryFrame|
      (|put| '|Equation| '|isFunctor|
             '((((|eval| ($ $ (|List| (|Symbol|)) (|List| |#1|)))
                 (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|))
                 (ELT $ 31))
                ((|eval| ($ $ (|Symbol|) |#1|))
                 (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|))
                 (ELT $ 27))
                ((~= (($ $)) (|has| |#1| (|SetCategory|)))
                 (ELT $ NIL))
                ((= (($ $)) (|has| |#1| (|SetCategory|)))
                 (ELT $ 40))
                ((|coerce| ((|OutputForm|) $))
                 (|has| |#1| (|SetCategory|)) (ELT $ 44))
                ((|hash| ((|SingleInteger|) $))
                 (|has| |#1| (|SetCategory|)) (ELT $ NIL))
                ((|latex| ((|String|) $)) (|has| |#1| (|SetCategory|))
                 (ELT $ NIL))
                ((|coerce| ((|Boolean|) $)) (|has| |#1| (|SetCategory|))
                 (ELT $ 45))
                ((+ ($ $ $)) (|has| |#1| (|AbelianSemiGroup|))
                 (ELT $ 47))
```

```
((* ($ (|PositiveInteger|) $))
 (|has| |#1| (|AbelianSemiGroup|)) (ELT $ NIL))
((|Zero| ($)) (|has| |#1| (|AbelianGroup|))
 (CONST $ 55))
((|sample| ($))
 (OR (|has| |#1| (|AbelianGroup|))
     (|has| |#1| (|Monoid|)))
 (CONST $ NIL))
((|zero?| ((|Boolean|) $)) (|has| |#1| (|AbelianGroup|))
 (ELT $ NIL))
((* ($ (|NonNegativeInteger|) $))
 (|has| |#1| (|AbelianGroup|)) (ELT $ NIL))
((|subtractIfCan| ((|Union| $ "failed") $ $))
 (|has| |#1| (|AbelianGroup|)) (ELT $ NIL))
((- ($ $)) (|has| |#1| (|AbelianGroup|)) (ELT $ 51))
((- ($ $ $)) (|has| |#1| (|AbelianGroup|)) (ELT $ 52))
((* ($ (|Integer|) $)) (|has| |#1| (|AbelianGroup|))
 (ELT $ 76))
((* ($ $ $)) (|has| |#1| (|SemiGroup|)) (ELT $ 59))
((** ($ $ (|PositiveInteger|)))
 (|has| |#1| (|SemiGroup|)) (ELT $ NIL))
((^ ($ $ (|PositiveInteger|)))
 (|has| |#1| (|SemiGroup|)) (ELT $ NIL))
((|One| ($)) (|has| |#1| (|Monoid|)) (CONST $ 63))
((|one?| ((|Boolean|) $)) (|has| |#1| (|Monoid|))
 (ELT $ NIL))
((** ($ $ (|NonNegativeInteger|)))
 (|has| |#1| (|Monoid|)) (ELT $ NIL))
((^ ($ $ (|NonNegativeInteger|)))
 (|has| |#1| (|Monoid|)) (ELT $ NIL))
((|recip| ((|Union| $ "failed") $))
 (|has| |#1| (|Monoid|)) (ELT $ 66))
((|inv| ($ $))
 (OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|)))
 (ELT $ 70))
((/ ($ $ $))
 (OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|)))
 (ELT $ 90))
((** ($ $ (|Integer|))) (|has| |#1| (|Group|))
 (ELT $ NIL))
((^ ($ $ (|Integer|))) (|has| |#1| (|Group|))
 (ELT $ NIL))
((|conjugate| ($ $ $)) (|has| |#1| (|Group|))
 (ELT $ NIL))
((|commutator| ($ $ $)) (|has| |#1| (|Group|))
 (ELT $ NIL))
((|characteristic| ((|NonNegativeInteger|)))
 (|has| |#1| (|Ring|)) (ELT $ 73))
((|coerce| ($ (|Integer|))) (|has| |#1| (|Ring|))
 (ELT $ NIL))
```

```
((* ($ |#1| $)) (|has| |#1| (|SemiGroup|)) (ELT $ 60))
((* ($ $ |#1|)) (|has| |#1| (|SemiGroup|)) (ELT $ 61))
((|differentiate| ($ $ (|Symbol|)))
 (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
 (ELT $ 85))
((|differentiate| ($ $ (|List| (|Symbol|))))
 (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
 (ELT $ NIL))
((|differentiate|
     ($ $ (|Symbol|) (|NonNegativeInteger|)))
 (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
 (ELT $ NIL))
((|differentiate|
     ($ $ (|List| (|Symbol|))
        (|List| (|NonNegativeInteger|))))
 (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
 (ELT $ NIL))
((D ($ $ (|Symbol|)))
 (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
 (ELT $ NIL))
((D ($ $ (|List| (|Symbol|))))
 (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
 (ELT $ NIL))
((D ($ $ (|Symbol|) (|NonNegativeInteger|)))
 (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
 (ELT $ NIL))
((D ($ $ (|List| (|Symbol|))
       (|List| (|NonNegativeInteger|))))
 (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
 (ELT $ NIL))
((/ ($ $ |#1|)) (|has| |#1| (|Field|)) (ELT $ NIL))
((|dimension| ((|CardinalNumber|)))
 (|has| |#1| (|Field|)) (ELT $ 88))
((|subst| ($ $ $)) (|has| |#1| (|ExpressionSpace|))
 (ELT $ 93))
((|factorAndSplit| ((|List| $) $))
 (|has| |#1| (|IntegralDomain|)) (ELT $ 19))
((|rightOne| ((|Union| $ "failed") $))
 (|has| |#1| (|Monoid|)) (ELT $ 68))
((|leftOne| ((|Union| $ "failed") $))
 (|has| |#1| (|Monoid|)) (ELT $ 67))
((- ($ $ |#1|)) (|has| |#1| (|AbelianGroup|))
 (ELT $ 54))
((- ($ |#1| $)) (|has| |#1| (|AbelianGroup|))
 (ELT $ 53))
((|rightZero| ($ $)) (|has| |#1| (|AbelianGroup|))
 (ELT $ 8))
((|leftZero| ($ $)) (|has| |#1| (|AbelianGroup|))
 (ELT $ 57))
((+ ($ $ |#1|)) (|has| |#1| (|AbelianSemiGroup|))
```

```
   (ELT $ 49))
  ((+ ($ |#1| $)) (|has| |#1| (|AbelianSemiGroup|))
   (ELT $ 48))
  ((|eval| ($ $ (|List| $)))
   (AND (|has| |#1| (|Evalable| |#1|))
        (|has| |#1| (|SetCategory|)))
   (ELT $ 37))
  ((|eval| ($ $ $))
   (AND (|has| |#1| (|Evalable| |#1|))
        (|has| |#1| (|SetCategory|)))
   (ELT $ 34))
  ((|map| ($ (|Mapping| |#1| |#1|) $)) T (ELT $ 24))
  ((|rhs| (|#1| $)) T (ELT $ 21))
  ((|lhs| (|#1| $)) T (ELT $ 9))
  ((|swap| ($ $)) T (ELT $ 22))
  ((|equation| ($ |#1| |#1|)) T (ELT $ 17))
  ((= ($ |#1| |#1|)) T (ELT $ 20)))
(|addModemap| '|Equation| '(|Equation| |#1|)
    '((|Join| (|Type|)
              (CATEGORY |domain|
                  (SIGNATURE = ($ |#1| |#1|))
                  (SIGNATURE |equation| ($ |#1| |#1|))
                  (SIGNATURE |swap| ($ $))
                  (SIGNATURE |lhs| (|#1| $))
                  (SIGNATURE |rhs| (|#1| $))
                  (SIGNATURE |map|
                      ($ (|Mapping| |#1| |#1|) $))
                  (IF (|has| |#1|
                       (|InnerEvalable| (|Symbol|) |#1|))
                      (ATTRIBUTE
                       (|InnerEvalable| (|Symbol|) |#1|))
                      |noBranch|)
                  (IF (|has| |#1| (|SetCategory|))
                      (PROGN
                        (ATTRIBUTE (|SetCategory|))
                        (ATTRIBUTE
                         (|CoercibleTo| (|Boolean|)))
                        (IF (|has| |#1| (|Evalable| |#1|))
                         (PROGN
                           (SIGNATURE |eval| ($ $ $))
                           (SIGNATURE |eval|
                            ($ $ (|List| $))))
                         |noBranch|))
                      |noBranch|)
                  (IF (|has| |#1| (|AbelianSemiGroup|))
                      (PROGN
                        (ATTRIBUTE (|AbelianSemiGroup|))
                        (SIGNATURE + ($ |#1| $))
                        (SIGNATURE + ($ $ |#1|)))
                      |noBranch|)
```

```
(IF (|has| |#1| (|AbelianGroup|))
    (PROGN
      (ATTRIBUTE (|AbelianGroup|))
      (SIGNATURE |leftZero| ($ $))
      (SIGNATURE |rightZero| ($ $))
      (SIGNATURE - ($ |#1| $))
      (SIGNATURE - ($ $ |#1|)))
    |noBranch|)
(IF (|has| |#1| (|SemiGroup|))
    (PROGN
      (ATTRIBUTE (|SemiGroup|))
      (SIGNATURE * ($ |#1| $))
      (SIGNATURE * ($ $ |#1|)))
    |noBranch|)
(IF (|has| |#1| (|Monoid|))
    (PROGN
      (ATTRIBUTE (|Monoid|))
      (SIGNATURE |leftOne|
       ((|Union| $ "failed") $))
      (SIGNATURE |rightOne|
       ((|Union| $ "failed") $)))
    |noBranch|)
(IF (|has| |#1| (|Group|))
    (PROGN
      (ATTRIBUTE (|Group|))
      (SIGNATURE |leftOne|
       ((|Union| $ "failed") $))
      (SIGNATURE |rightOne|
       ((|Union| $ "failed") $)))
    |noBranch|)
(IF (|has| |#1| (|Ring|))
    (PROGN
      (ATTRIBUTE (|Ring|))
      (ATTRIBUTE (|BiModule| |#1| |#1|)))
    |noBranch|)
(IF (|has| |#1| (|CommutativeRing|))
    (ATTRIBUTE (|Module| |#1|))
    |noBranch|)
(IF (|has| |#1| (|IntegralDomain|))
    (SIGNATURE |factorAndSplit|
     ((|List| $) $))
    |noBranch|)
(IF (|has| |#1|
     (|PartialDifferentialRing|
      (|Symbol|)))
    (ATTRIBUTE
     (|PartialDifferentialRing|
      (|Symbol|)))
    |noBranch|)
(IF (|has| |#1| (|Field|))
```

```
                        (PROGN
                          (ATTRIBUTE (|VectorSpace| |#1|))
                          (SIGNATURE / ($ $ $))
                          (SIGNATURE |inv| ($ $)))
                        |noBranch|)
                    (IF (|has| |#1| (|ExpressionSpace|))
                        (SIGNATURE |subst| ($ $ $))
                        |noBranch|)))
      (|Type|))
    T '|Equation|
    (|put| '|Equation| '|mode|
           '(|Mapping|
                (|Join| (|Type|)
                        (CATEGORY |domain|
                         (SIGNATURE = ($ |#1| |#1|))
                         (SIGNATURE |equation|
                          ($ |#1| |#1|))
                         (SIGNATURE |swap| ($ $))
                         (SIGNATURE |lhs| (|#1| $))
                         (SIGNATURE |rhs| (|#1| $))
                         (SIGNATURE |map|
                          ($ (|Mapping| |#1| |#1|) $))
                         (IF
                          (|has| |#1|
                           (|InnerEvalable| (|Symbol|)
                            |#1|))
                          (ATTRIBUTE
                           (|InnerEvalable| (|Symbol|)
                            |#1|))
                          |noBranch|)
                         (IF (|has| |#1| (|SetCategory|))
                          (PROGN
                            (ATTRIBUTE (|SetCategory|))
                            (ATTRIBUTE
                             (|CoercibleTo| (|Boolean|)))
                            (IF
                             (|has| |#1|
                              (|Evalable| |#1|))
                             (PROGN
                               (SIGNATURE |eval| ($ $ $))
                               (SIGNATURE |eval|
                                ($ $ (|List| $))))
                             |noBranch|))
                          |noBranch|)
                         (IF
                          (|has| |#1|
                           (|AbelianSemiGroup|))
                          (PROGN
                            (ATTRIBUTE
                             (|AbelianSemiGroup|))
```

```
   (SIGNATURE + ($ |#1| $))
   (SIGNATURE + ($ $ |#1|)))
 |noBranch|)
(IF (|has| |#1| (|AbelianGroup|))
 (PROGN
   (ATTRIBUTE (|AbelianGroup|))
   (SIGNATURE |leftZero| ($ $))
   (SIGNATURE |rightZero| ($ $))
   (SIGNATURE - ($ |#1| $))
   (SIGNATURE - ($ $ |#1|)))
 |noBranch|)
(IF (|has| |#1| (|SemiGroup|))
 (PROGN
   (ATTRIBUTE (|SemiGroup|))
   (SIGNATURE * ($ |#1| $))
   (SIGNATURE * ($ $ |#1|)))
 |noBranch|)
(IF (|has| |#1| (|Monoid|))
 (PROGN
   (ATTRIBUTE (|Monoid|))
   (SIGNATURE |leftOne|
    ((|Union| $ "failed") $))
   (SIGNATURE |rightOne|
    ((|Union| $ "failed") $)))
 |noBranch|)
(IF (|has| |#1| (|Group|))
 (PROGN
   (ATTRIBUTE (|Group|))
   (SIGNATURE |leftOne|
    ((|Union| $ "failed") $))
   (SIGNATURE |rightOne|
    ((|Union| $ "failed") $)))
 |noBranch|)
(IF (|has| |#1| (|Ring|))
 (PROGN
   (ATTRIBUTE (|Ring|))
   (ATTRIBUTE
    (|BiModule| |#1| |#1|)))
 |noBranch|)
(IF
 (|has| |#1| (|CommutativeRing|))
 (ATTRIBUTE (|Module| |#1|))
 |noBranch|)
(IF
 (|has| |#1| (|IntegralDomain|))
 (SIGNATURE |factorAndSplit|
  ((|List| $) $))
 |noBranch|)
(IF
 (|has| |#1|
```

```
                                    (|PartialDifferentialRing|
                                     (|Symbol|)))
                                   (ATTRIBUTE
                                    (|PartialDifferentialRing|
                                     (|Symbol|)))
                                   |noBranch|)
                                  (IF (|has| |#1| (|Field|))
                                   (PROGN
                                     (ATTRIBUTE
                                      (|VectorSpace| |#1|))
                                     (SIGNATURE / ($ $ $))
                                     (SIGNATURE |inv| ($ $)))
                                   |noBranch|)
                                  (IF
                                   (|has| |#1| (|ExpressionSpace|))
                                   (SIGNATURE |subst| ($ $ $))
                                   |noBranch|)))
                          (|Type|))
                    |$CategoryFrame|))))
```

1.7.4 The "constructorForm"

```
(|Equation| S)
```

1.7.5 The "constructorKind"

```
|domain|
```

1.7.6 The "constructorModemap"

```
((((|Equation| |#1|)
  (|Join| (|Type|)
          (CATEGORY |domain| (SIGNATURE = ($ |#1| |#1|))
              (SIGNATURE |equation| ($ |#1| |#1|))
              (SIGNATURE |swap| ($ $)) (SIGNATURE |lhs| (|#1| $))
              (SIGNATURE |rhs| (|#1| $))
              (SIGNATURE |map| ($ (|Mapping| |#1| |#1|) $))
              (IF (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|))
                  (ATTRIBUTE (|InnerEvalable| (|Symbol|) |#1|))
                  |noBranch|)
              (IF (|has| |#1| (|SetCategory|))
                  (PROGN
                    (ATTRIBUTE (|SetCategory|))
                    (ATTRIBUTE (|CoercibleTo| (|Boolean|)))
                    (IF (|has| |#1| (|Evalable| |#1|))
                        (PROGN
                          (SIGNATURE |eval| ($ $ $))
```

```
            (SIGNATURE |eval| ($ $ (|List| $))))
          |noBranch|))
    |noBranch|)
(IF (|has| |#1| (|AbelianSemiGroup|))
    (PROGN
      (ATTRIBUTE (|AbelianSemiGroup|))
      (SIGNATURE + ($ |#1| $))
      (SIGNATURE + ($ $ |#1|)))
    |noBranch|)
(IF (|has| |#1| (|AbelianGroup|))
    (PROGN
      (ATTRIBUTE (|AbelianGroup|))
      (SIGNATURE |leftZero| ($ $))
      (SIGNATURE |rightZero| ($ $))
      (SIGNATURE - ($ |#1| $))
      (SIGNATURE - ($ $ |#1|)))
    |noBranch|)
(IF (|has| |#1| (|SemiGroup|))
    (PROGN
      (ATTRIBUTE (|SemiGroup|))
      (SIGNATURE * ($ |#1| $))
      (SIGNATURE * ($ $ |#1|)))
    |noBranch|)
(IF (|has| |#1| (|Monoid|))
    (PROGN
      (ATTRIBUTE (|Monoid|))
      (SIGNATURE |leftOne| ((|Union| $ "failed") $))
      (SIGNATURE |rightOne| ((|Union| $ "failed") $)))
    |noBranch|)
(IF (|has| |#1| (|Group|))
    (PROGN
      (ATTRIBUTE (|Group|))
      (SIGNATURE |leftOne| ((|Union| $ "failed") $))
      (SIGNATURE |rightOne| ((|Union| $ "failed") $)))
    |noBranch|)
(IF (|has| |#1| (|Ring|))
    (PROGN
      (ATTRIBUTE (|Ring|))
      (ATTRIBUTE (|BiModule| |#1| |#1|)))
    |noBranch|)
(IF (|has| |#1| (|CommutativeRing|))
    (ATTRIBUTE (|Module| |#1|)) |noBranch|)
(IF (|has| |#1| (|IntegralDomain|))
    (SIGNATURE |factorAndSplit| ((|List| $) $))
    |noBranch|)
(IF (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
    (ATTRIBUTE (|PartialDifferentialRing| (|Symbol|)))
    |noBranch|)
(IF (|has| |#1| (|Field|))
    (PROGN
```

```
                  (ATTRIBUTE (|VectorSpace| |#1|))
                  (SIGNATURE / ($ $ $))
                  (SIGNATURE |inv| ($ $)))
                |noBranch|)
            (IF (|has| |#1| (|ExpressionSpace|))
                (SIGNATURE |subst| ($ $ $)) |noBranch|)))
 (|Type|))
(T |Equation|))
```

1.7.7 The "constructorCategory"

```
(|Join| (|Type|)
        (CATEGORY |domain| (SIGNATURE = ($ |#1| |#1|))
            (SIGNATURE |equation| ($ |#1| |#1|))
            (SIGNATURE |swap| ($ $)) (SIGNATURE |lhs| (|#1| $))
            (SIGNATURE |rhs| (|#1| $))
            (SIGNATURE |map| ($ (|Mapping| |#1| |#1|) $))
            (IF (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|))
                (ATTRIBUTE (|InnerEvalable| (|Symbol|) |#1|))
                |noBranch|)
            (IF (|has| |#1| (|SetCategory|))
                (PROGN
                  (ATTRIBUTE (|SetCategory|))
                  (ATTRIBUTE (|CoercibleTo| (|Boolean|)))
                  (IF (|has| |#1| (|Evalable| |#1|))
                      (PROGN
                        (SIGNATURE |eval| ($ $ $))
                        (SIGNATURE |eval| ($ $ (|List| $))))
                      |noBranch|))
                |noBranch|)
            (IF (|has| |#1| (|AbelianSemiGroup|))
                (PROGN
                  (ATTRIBUTE (|AbelianSemiGroup|))
                  (SIGNATURE + ($ |#1| $))
                  (SIGNATURE + ($ $ |#1|)))
                |noBranch|)
            (IF (|has| |#1| (|AbelianGroup|))
                (PROGN
                  (ATTRIBUTE (|AbelianGroup|))
                  (SIGNATURE |leftZero| ($ $))
                  (SIGNATURE |rightZero| ($ $))
                  (SIGNATURE - ($ |#1| $))
                  (SIGNATURE - ($ $ |#1|)))
                |noBranch|)
            (IF (|has| |#1| (|SemiGroup|))
                (PROGN
                  (ATTRIBUTE (|SemiGroup|))
                  (SIGNATURE * ($ |#1| $))
                  (SIGNATURE * ($ $ |#1|)))
```

```
        |noBranch|)
    (IF (|has| |#1| (|Monoid|))
        (PROGN
          (ATTRIBUTE (|Monoid|))
          (SIGNATURE |leftOne| ((|Union| $ "failed") $))
          (SIGNATURE |rightOne| ((|Union| $ "failed") $)))
        |noBranch|)
    (IF (|has| |#1| (|Group|))
        (PROGN
          (ATTRIBUTE (|Group|))
          (SIGNATURE |leftOne| ((|Union| $ "failed") $))
          (SIGNATURE |rightOne| ((|Union| $ "failed") $)))
        |noBranch|)
    (IF (|has| |#1| (|Ring|))
        (PROGN
          (ATTRIBUTE (|Ring|))
          (ATTRIBUTE (|BiModule| |#1| |#1|)))
        |noBranch|)
    (IF (|has| |#1| (|CommutativeRing|))
        (ATTRIBUTE (|Module| |#1|)) |noBranch|)
    (IF (|has| |#1| (|IntegralDomain|))
        (SIGNATURE |factorAndSplit| ((|List| $) $)) |noBranch|)
    (IF (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
        (ATTRIBUTE (|PartialDifferentialRing| (|Symbol|)))
        |noBranch|)
    (IF (|has| |#1| (|Field|))
        (PROGN
          (ATTRIBUTE (|VectorSpace| |#1|))
          (SIGNATURE / ($ $ $))
          (SIGNATURE |inv| ($ $)))
        |noBranch|)
    (IF (|has| |#1| (|ExpressionSpace|))
        (SIGNATURE |subst| ($ $ $)) |noBranch|)))
```

1.7.8 The "sourceFile"

```
"/research/test/int/algebra/EQ.spad"
```

1.7.9 The "modemaps"

```
((= (*1 *1 *2 *2)
    (AND (|isDomain| *1 (|Equation| *2)) (|ofCategory| *2 (|Type|))))
 (|equation| (*1 *1 *2 *2)
     (AND (|isDomain| *1 (|Equation| *2)) (|ofCategory| *2 (|Type|))))
 (|swap| (*1 *1 *1)
         (AND (|isDomain| *1 (|Equation| *2))
              (|ofCategory| *2 (|Type|))))
 (|lhs| (*1 *2 *1)
```

```
       (AND (|isDomain| *1 (|Equation| *2))
            (|ofCategory| *2 (|Type|))))
(|rhs| (*1 *2 *1)
       (AND (|isDomain| *1 (|Equation| *2))
            (|ofCategory| *2 (|Type|))))
(|map| (*1 *1 *2 *1)
       (AND (|isDomain| *2 (|Mapping| *3 *3))
            (|ofCategory| *3 (|Type|))
            (|isDomain| *1 (|Equation| *3))))
(|eval| (*1 *1 *1 *1)
        (AND (|ofCategory| *2 (|Evalable| *2))
             (|ofCategory| *2 (|SetCategory|))
             (|ofCategory| *2 (|Type|))
             (|isDomain| *1 (|Equation| *2))))
(|eval| (*1 *1 *1 *2)
        (AND (|isDomain| *2 (|List| (|Equation| *3)))
             (|ofCategory| *3 (|Evalable| *3))
             (|ofCategory| *3 (|SetCategory|))
             (|ofCategory| *3 (|Type|))
             (|isDomain| *1 (|Equation| *3))))
(+ (*1 *1 *2 *1)
   (AND (|isDomain| *1 (|Equation| *2))
        (|ofCategory| *2 (|AbelianSemiGroup|))
        (|ofCategory| *2 (|Type|))))
(+ (*1 *1 *1 *2)
   (AND (|isDomain| *1 (|Equation| *2))
        (|ofCategory| *2 (|AbelianSemiGroup|))
        (|ofCategory| *2 (|Type|))))
(|leftZero| (*1 *1 *1)
    (AND (|isDomain| *1 (|Equation| *2))
         (|ofCategory| *2 (|AbelianGroup|))
         (|ofCategory| *2 (|Type|))))
(|rightZero| (*1 *1 *1)
    (AND (|isDomain| *1 (|Equation| *2))
         (|ofCategory| *2 (|AbelianGroup|))
         (|ofCategory| *2 (|Type|))))
(- (*1 *1 *2 *1)
   (AND (|isDomain| *1 (|Equation| *2))
        (|ofCategory| *2 (|AbelianGroup|)) (|ofCategory| *2 (|Type|))))
(- (*1 *1 *1 *2)
   (AND (|isDomain| *1 (|Equation| *2))
        (|ofCategory| *2 (|AbelianGroup|)) (|ofCategory| *2 (|Type|))))
(|leftOne| (*1 *1 *1)
    (|partial| AND (|isDomain| *1 (|Equation| *2))
        (|ofCategory| *2 (|Monoid|)) (|ofCategory| *2 (|Type|))))
(|rightOne| (*1 *1 *1)
    (|partial| AND (|isDomain| *1 (|Equation| *2))
        (|ofCategory| *2 (|Monoid|)) (|ofCategory| *2 (|Type|))))
(|factorAndSplit| (*1 *2 *1)
    (AND (|isDomain| *2 (|List| (|Equation| *3)))
```

```
          (|isDomain| *1 (|Equation| *3))
          (|ofCategory| *3 (|IntegralDomain|))
          (|ofCategory| *3 (|Type|))))
 (|subst| (*1 *1 *1 *1)
          (AND (|isDomain| *1 (|Equation| *2))
               (|ofCategory| *2 (|ExpressionSpace|))
               (|ofCategory| *2 (|Type|))))
 (* (*1 *1 *1 *2)
    (AND (|isDomain| *1 (|Equation| *2))
         (|ofCategory| *2 (|SemiGroup|)) (|ofCategory| *2 (|Type|))))
 (* (*1 *1 *2 *1)
    (AND (|isDomain| *1 (|Equation| *2))
         (|ofCategory| *2 (|SemiGroup|)) (|ofCategory| *2 (|Type|))))
 (/ (*1 *1 *1 *1)
    (OR (AND (|isDomain| *1 (|Equation| *2))
             (|ofCategory| *2 (|Field|)) (|ofCategory| *2 (|Type|)))
        (AND (|isDomain| *1 (|Equation| *2))
             (|ofCategory| *2 (|Group|)) (|ofCategory| *2 (|Type|)))))
 (|inv| (*1 *1 *1)
        (OR (AND (|isDomain| *1 (|Equation| *2))
                 (|ofCategory| *2 (|Field|))
                 (|ofCategory| *2 (|Type|)))
            (AND (|isDomain| *1 (|Equation| *2))
                 (|ofCategory| *2 (|Group|))
                 (|ofCategory| *2 (|Type|))))))
```

1.7.10 The "operationAlist"

```
((~= (((|Boolean|) $ $) NIL (|has| |#1| (|SetCategory|))))
 (|zero?| (((|Boolean|) $) NIL (|has| |#1| (|AbelianGroup|))))
 (|swap| (($ $) 22))
 (|subtractIfCan|
     (((|Union| $ "failed") $ $) NIL (|has| |#1| (|AbelianGroup|))))
 (|subst| (($ $ $) 93 (|has| |#1| (|ExpressionSpace|))))
 (|sample|
     (($) NIL
      (OR (|has| |#1| (|AbelianGroup|)) (|has| |#1| (|Monoid|))) CONST))
 (|rightZero| (($ $) 8 (|has| |#1| (|AbelianGroup|))))
 (|rightOne| (((|Union| $ "failed") $) 68 (|has| |#1| (|Monoid|))))
 (|rhs| ((|#1| $) 21))
 (|recip| (((|Union| $ "failed") $) 66 (|has| |#1| (|Monoid|))))
 (|one?| (((|Boolean|) $) NIL (|has| |#1| (|Monoid|))))
 (|map| (($ (|Mapping| |#1| |#1|) $) 24)) (|lhs| ((|#1| $) 9))
 (|leftZero| (($ $) 57 (|has| |#1| (|AbelianGroup|))))
 (|leftOne| (((|Union| $ "failed") $) 67 (|has| |#1| (|Monoid|))))
 (|latex| (((|String|) $) NIL (|has| |#1| (|SetCategory|))))
 (|inv| (($ $) 70 (OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|)))))
 (|hash| (((|SingleInteger|) $) NIL (|has| |#1| (|SetCategory|))))
 (|factorAndSplit| (((|List| $) $) 19 (|has| |#1| (|IntegralDomain|))))
```

```
(|eval| (($ $ $) 34
         (AND (|has| |#1| (|Evalable| |#1|))
              (|has| |#1| (|SetCategory|))))
        (($ $ (|List| $)) 37
         (AND (|has| |#1| (|Evalable| |#1|))
              (|has| |#1| (|SetCategory|))))
        (($ $ (|Symbol|) |#1|) 27
         (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|)))
        (($ $ (|List| (|Symbol|)) (|List| |#1|)) 31
         (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|))))
(|equation| (($ |#1| |#1|) 17))
(|dimension| ((|CardinalNumber|)) 88 (|has| |#1| (|Field|)))
(|differentiate|
    (($ $ (|List| (|Symbol|)) (|List| (|NonNegativeInteger|))) NIL
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
    (($ $ (|Symbol|) (|NonNegativeInteger|)) NIL
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
    (($ $ (|List| (|Symbol|))) NIL
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
    (($ $ (|Symbol|)) 85
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))))
(|conjugate| (($ $ $) NIL (|has| |#1| (|Group|))))
(|commutator| (($ $ $) NIL (|has| |#1| (|Group|))))
(|coerce| (($ (|Integer|)) NIL (|has| |#1| (|Ring|)))
    ((($ (|Boolean|)) $) 45 (|has| |#1| (|SetCategory|)))
    ((($ (|OutputForm|)) $) 44 (|has| |#1| (|SetCategory|))))
(|characteristic| ((($ (|NonNegativeInteger|)) 73 (|has| |#1| (|Ring|))))
(^ (($ $ (|Integer|)) NIL (|has| |#1| (|Group|)))
   (($ $ (|NonNegativeInteger|)) NIL (|has| |#1| (|Monoid|)))
   (($ $ (|PositiveInteger|)) NIL (|has| |#1| (|SemiGroup|))))
(|Zero| (($) 55 (|has| |#1| (|AbelianGroup|)) CONST))
(|One| (($) 63 (|has| |#1| (|Monoid|)) CONST))
(D (($ $ (|List| (|Symbol|)) (|List| (|NonNegativeInteger|))) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
   (($ $ (|Symbol|) (|NonNegativeInteger|)) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
   (($ $ (|List| (|Symbol|))) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
   (($ $ (|Symbol|)) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))))
(= (($ |#1| |#1|) 20)
   ((($ (|Boolean|)) $ $) 40 (|has| |#1| (|SetCategory|))))
(/ (($ $ |#1|) NIL (|has| |#1| (|Field|)))
   (($ $ $) 90 (OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|)))))
(- (($ |#1| $) 53 (|has| |#1| (|AbelianGroup|)))
   (($ $ |#1|) 54 (|has| |#1| (|AbelianGroup|)))
   (($ $ $) 52 (|has| |#1| (|AbelianGroup|)))
   (($ $) 51 (|has| |#1| (|AbelianGroup|))))
(+ (($ |#1| $) 48 (|has| |#1| (|AbelianSemiGroup|)))
   (($ $ |#1|) 49 (|has| |#1| (|AbelianSemiGroup|)))
```

```
   (($ $ $) 47 (|has| |#1| (|AbelianSemiGroup|))))
 (** (($ $ (|Integer|)) NIL (|has| |#1| (|Group|)))
    (($ $ (|NonNegativeInteger|)) NIL (|has| |#1| (|Monoid|)))
    (($ $ (|PositiveInteger|)) NIL (|has| |#1| (|SemiGroup|))))
 (* (($ $ |#1|) 61 (|has| |#1| (|SemiGroup|)))
   (($ |#1| $) 60 (|has| |#1| (|SemiGroup|)))
   (($ $ $) 59 (|has| |#1| (|SemiGroup|)))
   (($ (|Integer|) $) 76 (|has| |#1| (|AbelianGroup|)))
   (($ (|NonNegativeInteger|) $) NIL (|has| |#1| (|AbelianGroup|)))
   (($ (|PositiveInteger|) $) NIL (|has| |#1| (|AbelianSemiGroup|)))))
```

1.7.11 The "superDomain"

1.7.12 The "signaturesAndLocals"

```
((|EQ;subst;3$;43| ($ $ $)) (|EQ;inv;2$;42| ($ $))
 (|EQ;/;3$;41| ($ $ $)) (|EQ;dimension;Cn;40| ((|CardinalNumber|)))
 (|EQ;differentiate;$S$;39| ($ $ (|Symbol|)))
 (|EQ;factorAndSplit;$L;38| ((|List| $) $))
 (|EQ;*;I2$;37| ($ (|Integer|) $))
 (|EQ;characteristic;Nni;36| ((|NonNegativeInteger|)))
 (|EQ;rightOne;$U;35| ((|Union| $ "failed") $))
 (|EQ;leftOne;$U;34| ((|Union| $ "failed") $)) (|EQ;inv;2$;33| ($ $))
 (|EQ;rightOne;$U;32| ((|Union| $ "failed") $))
 (|EQ;leftOne;$U;31| ((|Union| $ "failed") $))
 (|EQ;recip;$U;30| ((|Union| $ "failed") $)) (|EQ;One;$;29| ($))
 (|EQ;*;$S$;28| ($ $ S)) (|EQ;*;S2$;27| ($ S $))
 (|EQ;*;S2$;26| ($ S $)) (|EQ;*;3$;25| ($ $ $)) (|EQ;-;3$;24| ($ $ $))
 (|EQ;Zero;$;23| ($)) (|EQ;rightZero;2$;22| ($ $))
 (|EQ;leftZero;2$;21| ($ $)) (|EQ;-;$S$;20| ($ $ S))
 (|EQ;-;S2$;19| ($ S $)) (|EQ;-;2$;18| ($ $)) (|EQ;+;$S$;17| ($ $ S))
 (|EQ;+;S2$;16| ($ S $)) (|EQ;+;3$;15| ($ $ $))
 (|EQ;coerce;$B;14| ((|Boolean|) $))
 (|EQ;coerce;$Of;13| ((|OutputForm|) $))
 (|EQ;=;2$B;12| ((|Boolean|) $ $)) (|EQ;eval;$L$;11| ($ $ (|List| $)))
 (|EQ;eval;3$;10| ($ $ $))
 (|EQ;eval;$LL$;9| ($ $ (|List| (|Symbol|)) (|List| S)))
 (|EQ;eval;$SS$;8| ($ $ (|Symbol|) S))
 (|EQ;map;M2$;7| ($ (|Mapping| S S) $)) (|EQ;swap;2$;6| ($ $))
 (|EQ;rhs;$S;5| (S $)) (|EQ;lhs;$S;4| (S $))
 (|EQ;equation;2S$;3| ($ S S)) (|EQ;=;2S$;2| ($ S S))
 (|EQ;factorAndSplit;$L;1| ((|List| $) $)))
```

1.7.13 The "attributes"

```
((|unitsKnown| OR (|has| |#1| (|Ring|)) (|has| |#1| (|Group|)))
 (|rightUnitary| |has| |#1| (|Ring|))
 (|leftUnitary| |has| |#1| (|Ring|)))
```

1.7.14 The "predicates"

```
((|HasCategory| |#1| '(|Field|)) (|HasCategory| |#1| '(|SetCategory|))
 (|HasCategory| |#1| '(|Ring|))
 (|HasCategory| |#1| (LIST '|PartialDifferentialRing| '(|Symbol|)))
 (OR (|HasCategory| |#1| (LIST '|PartialDifferentialRing| '(|Symbol|)))
     (|HasCategory| |#1| '(|Ring|)))
 (|HasCategory| |#1| '(|Group|))
 (|HasCategory| |#1|
     (LIST '|InnerEvalable| '(|Symbol|) (|devaluate| |#1|)))
 (AND (|HasCategory| |#1| (LIST '|Evalable| (|devaluate| |#1|)))
      (|HasCategory| |#1| '(|SetCategory|)))
 (|HasCategory| |#1| '(|IntegralDomain|))
 (|HasCategory| |#1| '(|ExpressionSpace|))
 (OR (|HasCategory| |#1| '(|Field|)) (|HasCategory| |#1| '(|Group|)))
 (OR (|HasCategory| |#1| '(|Group|)) (|HasCategory| |#1| '(|Ring|)))
 (|HasCategory| |#1| '(|CommutativeRing|))
 (OR (|HasCategory| |#1| '(|CommutativeRing|))
     (|HasCategory| |#1| '(|Field|)) (|HasCategory| |#1| '(|Ring|)))
 (OR (|HasCategory| |#1| '(|CommutativeRing|))
     (|HasCategory| |#1| '(|Field|)))
 (|HasCategory| |#1| '(|Monoid|))
 (OR (|HasCategory| |#1| '(|Group|)) (|HasCategory| |#1| '(|Monoid|)))
 (|HasCategory| |#1| '(|SemiGroup|))
 (OR (|HasCategory| |#1| '(|Group|)) (|HasCategory| |#1| '(|Monoid|))
     (|HasCategory| |#1| '(|SemiGroup|)))
 (|HasCategory| |#1| '(|AbelianGroup|))
 (OR (|HasCategory| |#1| (LIST '|PartialDifferentialRing| '(|Symbol|)))
     (|HasCategory| |#1| '(|AbelianGroup|))
     (|HasCategory| |#1| '(|CommutativeRing|))
     (|HasCategory| |#1| '(|Field|)) (|HasCategory| |#1| '(|Ring|)))
 (OR (|HasCategory| |#1| '(|AbelianGroup|))
     (|HasCategory| |#1| '(|Monoid|)))
 (|HasCategory| |#1| '(|AbelianSemiGroup|))
 (OR (|HasCategory| |#1| (LIST '|PartialDifferentialRing| '(|Symbol|)))
     (|HasCategory| |#1| '(|AbelianGroup|))
     (|HasCategory| |#1| '(|AbelianSemiGroup|))
     (|HasCategory| |#1| '(|CommutativeRing|))
     (|HasCategory| |#1| '(|Field|)) (|HasCategory| |#1| '(|Ring|)))
 (OR (|HasCategory| |#1| (LIST '|PartialDifferentialRing| '(|Symbol|)))
     (|HasCategory| |#1| '(|AbelianGroup|))
     (|HasCategory| |#1| '(|AbelianSemiGroup|))
     (|HasCategory| |#1| '(|CommutativeRing|))
     (|HasCategory| |#1| '(|Field|)) (|HasCategory| |#1| '(|Group|))
     (|HasCategory| |#1| '(|Monoid|)) (|HasCategory| |#1| '(|Ring|))
     (|HasCategory| |#1| '(|SemiGroup|))
     (|HasCategory| |#1| '(|SetCategory|))))
```

1.7.15 The "abbreviation"

```
EQ
```

1.7.16 The "parents"

```
(((|Type|) . T)
 ((|InnerEvalable| (|Symbol|) S) |has| S
  (|InnerEvalable| (|Symbol|) S))
 ((|CoercibleTo| (|Boolean|)) |has| S (|SetCategory|))
 ((|SetCategory|) |has| S (|SetCategory|))
 ((|AbelianSemiGroup|) |has| S (|AbelianSemiGroup|))
 ((|AbelianGroup|) |has| S (|AbelianGroup|))
 ((|SemiGroup|) |has| S (|SemiGroup|)) ((|Monoid|) |has| S (|Monoid|))
 ((|Group|) |has| S (|Group|)) ((|BiModule| S S) |has| S (|Ring|))
 ((|Ring|) |has| S (|Ring|)) ((|Module| S) |has| S (|CommutativeRing|))
 ((|PartialDifferentialRing| (|Symbol|)) |has| S
  (|PartialDifferentialRing| (|Symbol|)))
 ((|VectorSpace| S) |has| S (|Field|)))
```

1.7.17 The "ancestors"

```
(((|AbelianGroup|) |has| S (|AbelianGroup|))
 ((|AbelianMonoid|) |has| S (|AbelianGroup|))
 ((|AbelianSemiGroup|) |has| S (|AbelianSemiGroup|))
 ((|BasicType|) |has| S (|SetCategory|))
 ((|BiModule| S S) |has| S (|Ring|))
 ((|CancellationAbelianMonoid|) |has| S (|AbelianGroup|))
 ((|CoercibleTo| (|OutputForm|)) |has| S (|SetCategory|))
 ((|CoercibleTo| (|Boolean|)) |has| S (|SetCategory|))
 ((|Group|) |has| S (|Group|))
 ((|InnerEvalable| (|Symbol|) S) |has| S
  (|InnerEvalable| (|Symbol|) S))
 ((|LeftModule| $) |has| S (|Ring|))
 ((|LeftModule| S) |has| S (|Ring|))
 ((|Module| S) |has| S (|CommutativeRing|))
 ((|Monoid|) |has| S (|Monoid|))
 ((|PartialDifferentialRing| (|Symbol|)) |has| S
  (|PartialDifferentialRing| (|Symbol|)))
 ((|RightModule| S) |has| S (|Ring|)) ((|Ring|) |has| S (|Ring|))
 ((|Rng|) |has| S (|Ring|)) ((|SemiGroup|) |has| S (|SemiGroup|))
 ((|SetCategory|) |has| S (|SetCategory|)) ((|Type|) . T)
 ((|VectorSpace| S) |has| S (|Field|)))
```

1.7.18 The "documentation"

```
((|constructor|
```

```
    (NIL "Equations as mathematical objects. All properties of the basis
          domain,{} \\spadignore{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."))
(|subst| (($ $ $)
          "\\spad{subst(eq1,{}eq2)} substitutes \\spad{eq2} into both sides
          of \\spad{eq1} the \\spad{lhs} of \\spad{eq2} should be a kernel"))
(|inv| (($ $)
        "\\spad{inv(x)} returns the multiplicative inverse of \\spad{x}."))
(/ (($ $ $)
    "\\spad{e1/e2} produces a new equation by dividing the left and right
     hand sides of equations \\spad{e1} and \\spad{e2}."))
(|factorAndSplit|
    ((($ $) $)
     "\\spad{factorAndSplit(eq)} make the right hand side 0 and factors the
      new left hand side. Each factor is equated to 0 and put into the
      resulting list without repetitions."))
(|rightOne|
    ((($ "failed") $)
     "\\spad{rightOne(eq)} divides by the right hand side.")
    ((($ "failed") $)
     "\\spad{rightOne(eq)} divides by the right hand side,{} if possible."))
(|leftOne|
    ((($ "failed") $)
     "\\spad{leftOne(eq)} divides by the left hand side.")
    ((($ "failed") $)
     "\\spad{leftOne(eq)} divides by the left hand side,{} if possible."))
(* (($ $ |#1|)
    "\\spad{eqn*x} produces a new equation by multiplying both sides of
     equation eqn by \\spad{x}.")
   (($ |#1| $)
    "\\spad{x*eqn} produces a new equation by multiplying both sides of
     equation eqn by \\spad{x}."))
(- (($ $ |#1|)
    "\\spad{eqn-x} produces a new equation by subtracting \\spad{x} from
     both sides of equation eqn.")
   (($ |#1| $)
    "\\spad{x-eqn} produces a new equation by subtracting both sides of
     equation eqn from \\spad{x}."))
(|rightZero|
    (($ $) "\\spad{rightZero(eq)} subtracts the right hand side."))
(|leftZero|
    (($ $) "\\spad{leftZero(eq)} subtracts the left hand side."))
(+ (($ $ |#1|)
    "\\spad{eqn+x} produces a new equation by adding \\spad{x} to both
    sides of equation eqn.")
   (($ |#1| $)
    "\\spad{x+eqn} produces a new equation by adding \\spad{x} to both
    sides of equation eqn."))
(|eval| (($ $ (|List| $))
```

```
         "\\spad{eval(eqn,{} [x1=v1,{} ... xn=vn])} replaces \\spad{xi}
         by \\spad{vi} in equation \\spad{eqn}.")
        (($ $ $)
         "\\spad{eval(eqn,{} x=f)} replaces \\spad{x} by \\spad{f} in
         equation \\spad{eqn}."))
 (|map| (($ (|Mapping| |#1| |#1|) $)
        "\\spad{map(f,{}eqn)} constructs a new equation by applying
         \\spad{f} to both sides of \\spad{eqn}."))
 (|rhs| ((|#1| $)
        "\\spad{rhs(eqn)} returns the right hand side of equation
         \\spad{eqn}."))
 (|lhs| ((|#1| $)
        "\\spad{lhs(eqn)} returns the left hand side of equation
         \\spad{eqn}."))
 (|swap| (($ $)
         "\\spad{swap(eq)} interchanges left and right hand side of
          equation \\spad{eq}."))
 (|equation|
     (($ |#1| |#1|) "\\spad{equation(a,{}b)} creates an equation."))
 (= (($ |#1| |#1|) "\\spad{a=b} creates an equation.")))
```

1.7.19 The "slotInfo"

```
(|Equation|
    (NIL (~= ((38 0 0) NIL (|has| |#1| (|SetCategory|))))
         (|zero?| ((38 0) NIL (|has| |#1| (|AbelianGroup|))))
         (|swap| ((0 0) 22))
         (|subtractIfCan| ((64 0 0) NIL (|has| |#1| (|AbelianGroup|))))
         (|subst| ((0 0 0) 93 (|has| |#1| (|ExpressionSpace|))))
         (|sample|
             ((0) NIL
              (OR (|has| |#1| (|AbelianGroup|))
                  (|has| |#1| (|Monoid|)))
              CONST))
         (|rightZero| ((0 0) 8 (|has| |#1| (|AbelianGroup|))))
         (|rightOne| ((64 0) 68 (|has| |#1| (|Monoid|))))
         (|rhs| ((6 0) 21))
         (|recip| ((64 0) 66 (|has| |#1| (|Monoid|))))
         (|one?| ((38 0) NIL (|has| |#1| (|Monoid|))))
         (|map| ((0 23 0) 24)) (|lhs| ((6 0) 9))
         (|leftZero| ((0 0) 57 (|has| |#1| (|AbelianGroup|))))
         (|leftOne| ((64 0) 67 (|has| |#1| (|Monoid|))))
         (|latex| ((97 0) NIL (|has| |#1| (|SetCategory|))))
         (|inv| ((0 0) 70
                 (OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|)))))
         (|hash| ((96 0) NIL (|has| |#1| (|SetCategory|))))
         (|factorAndSplit| ((18 0) 19 (|has| |#1| (|IntegralDomain|))))
         (|eval| ((0 0 28 29) 31
                  (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|)))
```

```
        ((0 0 25 6) 27
         (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|)))
        ((0 0 18) 37
         (AND (|has| |#1| (|Evalable| |#1|))
              (|has| |#1| (|SetCategory|))))
        ((0 0 0) 34
         (AND (|has| |#1| (|Evalable| |#1|))
              (|has| |#1| (|SetCategory|)))))
(|equation| ((0 6 6) 17))
(|dimension| ((86) 88 (|has| |#1| (|Field|))))
(|differentiate|
    ((0 0 25 71) NIL
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
    ((0 0 28 95) NIL
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
    ((0 0 25) 85
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
    ((0 0 28) NIL
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))))
(|conjugate| ((0 0 0) NIL (|has| |#1| (|Group|))))
(|commutator| ((0 0 0) NIL (|has| |#1| (|Group|))))
(|coerce| ((38 0) 45 (|has| |#1| (|SetCategory|)))
    ((41 0) 44 (|has| |#1| (|SetCategory|)))
    ((0 74) NIL (|has| |#1| (|Ring|))))
(|characteristic| ((71) 73 (|has| |#1| (|Ring|))))
(^ ((0 0 94) NIL (|has| |#1| (|SemiGroup|)))
   ((0 0 71) NIL (|has| |#1| (|Monoid|)))
   ((0 0 74) NIL (|has| |#1| (|Group|))))
(|Zero| ((0) 55 (|has| |#1| (|AbelianGroup|)) CONST))
(|One| ((0) 63 (|has| |#1| (|Monoid|)) CONST))
(D ((0 0 25 71) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
   ((0 0 28 95) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
   ((0 0 25) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
   ((0 0 28) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))))
(= ((0 6 6) 20) ((38 0 0) 40 (|has| |#1| (|SetCategory|))))
(/ ((0 0 6) NIL (|has| |#1| (|Field|)))
   ((0 0 0) 90
    (OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|)))))
(- ((0 0 6) 54 (|has| |#1| (|AbelianGroup|)))
   ((0 6 0) 53 (|has| |#1| (|AbelianGroup|)))
   ((0 0 0) 52 (|has| |#1| (|AbelianGroup|)))
   ((0 0) 51 (|has| |#1| (|AbelianGroup|))))
(+ ((0 0 6) 49 (|has| |#1| (|AbelianSemiGroup|)))
   ((0 6 0) 48 (|has| |#1| (|AbelianSemiGroup|)))
   ((0 0 0) 47 (|has| |#1| (|AbelianSemiGroup|))))
(** ((0 0 94) NIL (|has| |#1| (|SemiGroup|)))
```

```
            ((0 0 71) NIL (|has| |#1| (|Monoid|)))
            ((0 0 74) NIL (|has| |#1| (|Group|))))
         (* ((0 6 0) 60 (|has| |#1| (|SemiGroup|)))
            ((0 0 6) 61 (|has| |#1| (|SemiGroup|)))
            ((0 0 0) 59 (|has| |#1| (|SemiGroup|)))
            ((0 94 0) NIL (|has| |#1| (|AbelianSemiGroup|)))
            ((0 74 0) 76 (|has| |#1| (|AbelianGroup|)))
            ((0 71 0) NIL (|has| |#1| (|AbelianGroup|))))))
```

1.7.20 The "index"

```
(("slot1Info" 0 32444) ("documentation" 0 29640) ("ancestors" 0 28691)
 ("parents" 0 28077) ("abbreviation" 0 28074) ("predicates" 0 25442)
 ("attributes" 0 25304) ("signaturesAndLocals" 0 23933)
 ("superDomain" 0 NIL) ("operationAlist" 0 20053) ("modemaps" 0 17216)
 ("sourceFile" 0 17179) ("constructorCategory" 0 15220)
 ("constructorModemap" 0 13215) ("constructorKind" 0 13206)
 ("constructorForm" 0 13191) ("compilerInfo" 0 4433)
 ("loadTimeStuff" 0 20))
```

Chapter 2

Compiler top level

2.1 Global Data Structures

2.2 Pratt Parsing

Parsing involves understanding the association of symbols and operators. Vaughn Pratt [8] poses the question "Given a substring AEB where A takes a right argument, B a left, and E is an expression, does E associate with A or B?".

Floyd [9] associates a precedence with operators, storing them in a table, called "binding powers". The expression E would associate with the argument position having the highest binding power. This leads to a large set of numbers, one for every situation.

Pratt assigns data types to "classes" and then creates a total order on the classes. He lists, in ascending order, Outcomes, Booleans, Graphs (trees, lists, etc), Strings, Algebraics (e.g. Integer, complex numbers, polynomials, real arrays) and references (e.g. the left hand side of assignments). Thus, Strings ¡ References. The key restriction is "that the class of the type at any argument that might participate in an association problem not be less than the class of the data type of the result of the function taking that argument".

For a less-than comparision ("<") the argument types are Algebraics but the result type is Boolean. Since Algebraics are greater than Boolean we can associate the Algebraics together and apply them as arguments to the Boolean.

In more detail, there an "association" is a function of 4 types:

- a_A – The data type of the right argument
- r_A – The return type of the right argument
- a_B – The data type of the left argument

- r_B – The return type of the left argument

Note that the return types might depend on the type of the expression E. If all 4 are of the same class then the association is to the left.

Using these ideas and given the restriction above, Pratt proves that every association problem has at most one solution consistant with the data types of the associated operators.

Pratt proves that there exists an assignment of integers to the argument positions of each token in the language such that the correct association, if any, is always in the direction of the argument position with the larger number, with ties being broken to the left.

To construct the proper numbers, first assign even integers to the data type classes. Then to each argument position assign an integer lying strictly (where possible) between the integers corresponding to the classes of the argument and result types.

For tokens like "and", "or", +, ∗, and the Booleans and Algebraics can be subdivided into pseudo-classes so that

terms < factors < primaries

Then + is defined over terms, ∗ over factors, and over primaries with coercions allowed from primaries to factors to terms. To be consistent with Algol, the primaries should be a right associative class (e.g. xyz)

2.3)compile

This is the implementation of the)compile command.

You use this command to invoke the new Axiom library compiler or the old Axiom system compiler. The `)compile` system command is actually a combination of Axiom processing and a call to the Aldor compiler. It is performing double-duty, acting as a front-end to both the Aldor compiler and the old Axiom system compiler. (The old Axiom system compiler was written in Lisp and was an integral part of the Axiom environment. The Aldor compiler is written in C and executed by the operating system when called from within Axiom.)

User Level Required: compiler

Command Syntax:

`)compile`

`)compile` *fileName*

`)compile` *fileName*`.spad`

`)compile` *directory/fileName*`.spad`

)compile *fileName*)old

)compile *fileName*)translate

)compile *fileName*)quiet

)compile *fileName*)noquiet

)compile *fileName*)moreargs

)compile *fileName*)onlyargs

)compile *fileName*)break

)compile *fileName*)nobreak

)compile *fileName*)library

)compile *fileName*)nolibrary

)compile *fileName*)vartrace

)compile *fileName*)constructor *nameOrAbbrev*

These command forms invoke the Aldor compiler.

)compile *fileName*.as

)compile *directory/fileName*.as

)compile *fileName*.ao

)compile *directory/fileName*.ao

)compile *fileName*.al

)compile *directory/fileName*.al

)compile *fileName*.lsp

)compile *directory/fileName*.lsp

)compile *fileName*)new

Command Description:

The first thing)compile does is look for a source code filename among its arguments. Thus

```
)compile mycode.spad
)compile /u/jones/mycode.spad
)compile mycode
```

all invoke)compiler on the file /u/jones/mycode.spad if the current Axiom working directory is /u/jones. (Recall that you can set the working directory via the)cd command. If you don't set it explicitly, it is the directory from which you started Axiom.)

If you omit the file extension, the command looks to see if you have specified the)new or)old option. If you have given one of these options, the corresponding compiler is used.

The command first looks in the standard system directories for files with extension *.as, .ao* and *.al* and then files with extension *.spad*. The first file found has the appropriate compiler invoked on it. If the command cannot find a matching file, an error message is displayed and the command terminates.

The first thing)compile does is look for a source code filename among its arguments. Thus

```
)compile mycode
)co mycode
)co mycode.spad
```

all invoke)compiler on the file /u/jones/mycode.spad if the current Axiom working directory is /u/jones. Recall that you can set the working directory via the)cd command. If you don't set it explicitly, it is the directory from which you started Axiom.

This is frequently all you need to compile your file.

This simple command:

1. Invokes the Spad compiler and produces Lisp output.
2. Calls the Lisp compiler if the compilation was successful.
3. Uses the)library command to tell Axiom about the contents of your compiled file and arrange to have those contents loaded on demand.

Should you not want the)library command automatically invoked, call)compile with the)nolibrary option. For example,

```
)compile mycode )nolibrary
```

By default, the)library system command *exposes* all domains and categories it processes. This means that the Axiom intepreter will consider those domains and categories when it is trying to resolve a reference to a function. Sometimes domains and categories should not be exposed. For example, a domain may just be used privately by another domain and may not be meant for top-level use. The)library command should still be used, though, so that the code will be loaded on demand. In this case, you should use the)nolibrary option on)compile and the)noexpose option in the)library command. For example,

```
)compile mycode )nolibrary
)library mycode )noexpose
```

Once you have established your own collection of compiled code, you may find it handy to use the `)dir` option on the `)library` command. This causes `)library` to process all compiled code in the specified directory. For example,

```
)library )dir /u/jones/quantum
```

You must give an explicit directory after `)dir`, even if you want all compiled code in the current working directory processed, e.g.

```
)library )dir .
```

2.3.1 Spad compiler

This command compiles files with file extension `.spad` with the Spad system compiler.

The `)translate` option is used to invoke a special version of the old system compiler that will translate a *.spad* file to a *.as* file. That is, the *.spad* file will be parsed and analyzed and a file using the new syntax will be created.

By default, the *.as* file is created in the same directory as the *.spad* file. If that directory is not writable, the current directory is used. If the current directory is not writable, an error message is given and the command terminates. Note that `)translate` implies the `)old` option so the file extension can safely be omitted. If `)translate` is given, all other options are ignored. Please be aware that the translation is not necessarily one hundred percent complete or correct. You should attempt to compile the output with the Aldor compiler and make any necessary corrections.

You can compile category, domain, and package constructors contained in files with file extension *.spad*. You can compile individual constructors or every constructor in a file.

The full filename is remembered between invocations of this command and `)edit` commands. The sequence of commands

```
)compile matrix.spad
)edit
)compile
```

will call the compiler, edit, and then call the compiler again on the file **matrix.spad.** If you do not specify a *directory,* the working current directory is searched for the file. If the file is not found, the standard system directories are searched.

If you do not give any options, all constructors within a file are compiled. Each constructor should have an `)abbreviation` command in the file in which it is

defined. We suggest that you place the)abbreviation commands at the top of the file in the order in which the constructors are defined.

The)library option causes directories containing the compiled code for each constructor to be created in the working current directory. The name of such a directory consists of the constructor abbreviation and the **.nrlib** file extension. For example, the directory containing the compiled code for the MATRIX constructor is called **MATRIX.nrlib.** The)nolibrary option says that such files should not be created. The default is)library. Note that the semantics of)library and)nolibrary for the new Aldor compiler and for the old system compiler are completely different.

The)vartrace option causes the compiler to generate extra code for the constructor to support conditional tracing of variable assignments. Without this option, this code is suppressed and one cannot use the)vars option for the trace command.

The)constructor option is used to specify a particular constructor to compile. All other constructors in the file are ignored. The constructor name or abbreviation follows)constructor. Thus either

```
)compile matrix.spad )constructor RectangularMatrix
```

or

```
)compile matrix.spad )constructor RMATRIX
```

compiles the RectangularMatrix constructor defined in **matrix.spad.**

The)break and)nobreak options determine what the spad compiler does when it encounters an error.)break is the default and it indicates that processing should stop at the first error. The value of the)set break variable then controls what happens.

2.4 Operator Precedence Table Initialization

```
; PURPOSE: This file sets up properties which are used by the Boot lexical
;          analyzer for bottom-up recognition of operators.  Also certain
;          other character-class definitions are included, as well as
;          table accessing functions.
;
; ORGANIZATION: Each section is organized in terms of Creation and Access code.
;
;               1. Led and Nud Tables
;               2. GLIPH  Table
;               3. RENAMETOK Table
;               4. GENERIC Table
;               5. Character syntax class predicates
```

2.4.1 LED and NUD Tables

```
; **** 1. LED and NUD Tables

; ** TABLE PURPOSE

; Led and Nud have to do with operators. An operator with a Led property takes
; an operand on its left (infix/suffix operator).

; An operator with a Nud takes no operand on its left (prefix/nilfix).
; Some have both (e.g. - ).  This terminology is from the Pratt parser.
; The translator for Scratchpad II is a modification of the Pratt parser which
; branches to special handlers when it is most convenient and practical to
; do so (Pratt's scheme cannot handle local contexts very easily).

; Both LEDs and NUDs have right and left binding powers.  This is meaningful
; for prefix and infix operators.  These powers are stored as the values of
; the LED and NUD properties of an atom, if the atom has such a property.
; The format is:

;       <Operator Left-Binding-Power  Right-Binding-Power <Special-Handler>>

; where the Special-Handler is the name of a function to be evaluated when that
; keyword is encountered.

; The default values of Left and Right Binding-Power are NIL.  NIL is a
; legitimate value signifying no precedence.  If the Special-Handler is NIL,
; this is just an ordinary operator (as opposed to a surfix operator like
; if-then-else).
;
; The Nud value gives the precedence when the operator is a prefix op.
; The Led value gives the precedence when the operator is an infix op.
; Each op has 2 priorities, left and right.
; If the right priority of the first is greater than or equal to the
; left priority of the second then collect the second operator into
; the right argument of the first operator.
```

— LEDNUDTables —

```
; ** TABLE CREATION

(defun makenewop (x y) (makeop x y '|PARSE-NewKEY|))

(defun makeop (x y keyname)
  (if (or (not (cdr x)) (numberp (second x)))
      (setq x (cons (first x) x)))
  (if (and (alpha-char-p (elt (princ-to-string (first x)) 0))
           (not (member (first x) (eval keyname))))
```

```
      (set keyname (cons (first x) (eval keyname))))
  (put (first x) y x)
  (second x))

(setq |PARSE-NewKEY| nil) ;;list of keywords

(mapcar #'(LAMBDA(J) (MAKENEWOP J '|Led|))
        '((* 800 801)   (|rem| 800 801)   (|mod| 800 801)
          (|quo| 800 801)   (|div| 800 801)
          (/ 800 801)    (** 900 901)  (^ 900 901)
          (|exquo| 800 801) (+ 700 701)
          (\- 700 701)    (\-\> 1001 1002)  (\<\- 1001 1002)
          (\: 996 997)    (\:\: 996 997)
          (\@ 996 997)    (|pretend| 995 996)
          (\.)            (\! \! 1002 1001)
          (\, 110 111)
          (\; 81 82 (|PARSE-SemiColon|))
          (\< 400 400)    (\> 400 400)
          (\<\< 400 400)  (\>\> 400 400)
          (\<= 400 400)   (\>= 400 400)
          (= 400 400)     (^= 400 400)
          (\~= 400 400)
          (|in| 400 400)    (|case| 400 400)
          (|add| 400 120)   (|with| 2000 400 (|PARSE-InfixWith|))
          (|has| 400 400)
          (|where| 121 104)     ; must be 121 for SPAD, 126 for boot--> nboot
          (|when| 112 190)
          (|otherwise| 119 190 (|PARSE-Suffix|))
          (|is| 400 400)    (|isnt| 400 400)
          (|and| 250 251)   (|or| 200 201)
          (/\\ 250 251)   (\\/ 200 201)
          (\.\. SEGMENT 401 699 (|PARSE-Seg|))
          (=\> 123 103)
          (+-\> 995 112)
          (== DEF 122 121)
          (==\> MDEF 122 121)
          (\| 108 111)                          ;was 190 190
          (\:- LETD 125 124) (\:= LET 125 124)))

(mapcar #'(LAMBDA (J) (MAKENEWOP J `|Nud|))
        '((|for| 130 350 (|PARSE-Loop|))
          (|while| 130 190 (|PARSE-Loop|))
          (|until| 130 190 (|PARSE-Loop|))
          (|repeat| 130 190 (|PARSE-Loop|))
          (|import| 120 0 (|PARSE-Import|) )
          (|unless|)
          (|add| 900 120)
          (|with| 1000 300 (|PARSE-With|))
          (|has| 400 400)
          (\- 701 700)  ; right-prec. wants to be -1 + left-prec
```

```
;;        (\+ 701 700)
          (\# 999 998)
          (\! 1002 1001)
          (\' 999 999 (|PARSE-Data|))
          (\<\< 122 120 (|PARSE-LabelExpr|))
          (\>\>)
          (^ 260 259 NIL)
          (\-\> 1001 1002)
          (\: 194 195)
          (|not| 260 259 NIL)
          (\~ 260 259 nil)
          (\= 400 700)
          (|return| 202 201 (|PARSE-Return|))
          (|leave| 202 201 (|PARSE-Leave|))
          (|exit| 202 201 (|PARSE-Exit|))
          (|from|)
          (|iterate|)
          (|yield|)
          (|if| 130 0 (|PARSE-Conditional|))    ; was 130
          (\| 0 190)
          (|suchthat|)
          (|then| 0 114)
          (|else| 0 114)))
```

———-

2.5 Gliph Table

Gliphs are symbol clumps. The gliph property of a symbol gives the tree describing the tokens which begin with that symbol. The token reader uses the gliph property to determine the longest token. Thus := is read as one token not as : followed by =.

— **GLIPHTable** —

```
(mapcar #'(lambda (x) (put (car x) 'gliph (cdr x)))
        `(
          ( \| (\))        )
          ( *  (*)         )
          ( \( (<) (\|)    )
          ( +  (- (>))     )
          ( -  (>)         )
          ( <  (=) (<)     )
    ;;    ( /  (\\)        ) breaks */xxx
          ( \\ (/)         )
          ( >  (=) (>) (\)))
```

```
( =  (= (>)) (>) )
( \. (\.)        )
( ^  (=)         )
( \~ (=)         )
( \: (=) (-) (\:))))
```

———-

2.5.1 Rename Token Table

RENAMETOK defines alternate token strings which can be used for different keyboards which define equivalent tokens.

— **RENAMETOKTable** —

```
(mapcar
  #'(lambda (x) (put (car x) 'renametok (cadr x)) (makenewop x nil))
        '((\(\| \[)                    ; (| |) means []
          (\|\) \])
          (\(< \{)                     ; (< >) means {}
          (>\) \})))
```

———-

2.5.2 Generic function table

GENERIC operators be suffixed by $ qualifications in SPAD code. $ is then followed by a domain label, such as I for Integer, which signifies which domain the operator refers to. For example `+$Integer` is + for Integers.

— **GENERICTable** —

```
(mapcar #'(lambda (x) (put x 'generic 'true))
        '(- = * |rem| |mod| |quo| |div| / ** |exquo| + - < > <= >= ^= ))
```

———-

2.6 Giant steps, Baby steps

We will walk through the compiler with the EQ.spad example using a Giant-steps, Baby-steps approach. That is, we will show the large scale (Giant) transformations at each stage of compilation and discuss the details (Baby) in subsequent chapters.

Chapter 3

The Parser

3.1 EQ.spad

We will explain the compilation function using the file `EQ.spad`. We trace the execution of the various functions to understand the actual call parameters and results returned. The `EQ.spad` file is:

```
)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
++ Basic Operations: =
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: equation
++ Examples:
++ References:
++ 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(eqn) returns the left hand side of equation eqn.
rhs: $ -> S
    ++ rhs(eqn) 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(eqn, x=f) replaces x by f in equation eqn.
       eval: ($, List $) -> $
           ++ eval(eqn, [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
    l:S   * eqn:$ == l       * eqn.lhs = l       * eqn.rhs
    l:S * eqn:$  ==  l * eqn.lhs    =    l * eqn.rhs
    eqn:$ * l:S  ==  eqn.lhs * l    =    eqn.rhs * l
    -- 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)]
```

3.2 preparse

The first large transformation of this input occurs in the function preparse . The preparse function reads the source file and breaks the input into a list of pairs. The first part of the pair is the line number of the input file and the second part of the pair is the actual source text as a string.

One feature that is the added semicolons at the end of the strings where the "pile" structure of the code has been converted to a semicolon delimited form.

3.2.1 defvar $index

— **initvars** —

```
(defvar $index 0 "File line number of most recently read line")
```

————-

3.2.2 defvar $linelist

— **initvars** —

```
(defvar $linelist nil "Stack of preparsed lines")
```

————-

3.2.3 defvar $echolinestack

— **initvars** —

```
(defvar $echolinestack nil "Stack of lines to list")
```

————-

3.2.4 defvar $preparse-last-line

— **initvars** —

```
(defvar $preparse-last-line nil "Most recently read line")
```

————-

3.3 Parsing routines

The **initialize-preparse** expects to be called before the **preparse** function. It initializes the state, in particular, it reads a single line from the input stream and

stores it in `$preparse-last-line`. The caller gives a stream and the `$preparse-last-line` variable is initialized as:

```
2> (INITIALIZE-PREPARSE #<input stream "/tmp/EQ.spad">)
<2 (INITIALIZE-PREPARSE ")abbrev domain EQ Equation")
```

3.3.1 defun initialize-preparse

[get-a-line p96]
[$index p72]
[$linelist p72]
[$echolinestack p72]
[$preparse-last-line p72]

— **defun initialize-preparse** —

```
(defun initialize-preparse (strm)
  (setq $index 0)
  (setq $linelist nil)
  (setq $echolinestack nil)
  (setq $preparse-last-line (get-a-line strm)))
```

———-

The **preparse** function returns a list of pairs of the form: ((linenumber . linestring) .... (linenumber . linestring)) For instance, for the file `EQ.spad`, we get:

```
   2> (PREPARSE #<input stream "/tmp/EQ.spad">)
     3> (PREPARSE1 (")abbrev domain EQ Equation"))
       4> (|doSystemCommand| "abbrev domain EQ Equation")
       <4 (|doSystemCommand| NIL)
     <3 (PREPARSE1 ( ...[snip]... )
    <2 (PREPARSE (
(19 . "Equation(S: Type): public == private where")
(20 . " (Ex ==> OutputForm;")
(21 . "  public ==> Type with")
(22 . "   (\"=\": (S, S) -> $;")
(24 . "    equation: (S, S) -> $;")
(26 . "    swap: $ -> $;")
(28 . "    lhs: $ -> S;")
(30 . "    rhs: $ -> S;")
(32 . "    map: (S -> S, $) -> $;")
(35 . "    if S has InnerEvalable(Symbol,S) then")
(36 . "              InnerEvalable(Symbol,S);")
(37 . "    if S has SetCategory then")
(38 . "       (SetCategory;")
```

```
(39 . "        CoercibleTo Boolean;")
(40 . "        if S has Evalable(S) then")
(41 . "          (eval: ($, $) -> $;")
(43 . "           eval: ($, List $) -> $));")
(45 . "    if S has AbelianSemiGroup then")
(46 . "       (AbelianSemiGroup;")
(47 . "        \"+\": (S, $) -> $;")
(50 . "        \"+\": ($, S) -> $);")
(53 . "    if S has AbelianGroup then")
(54 . "       (AbelianGroup;")
(55 . "        leftZero : $ -> $;")
(57 . "        rightZero : $ -> $;")
(59 . "        \"-\": (S, $) -> $;")
(62 . "        \"-\": ($, S) -> $);")
(65 . "    if S has SemiGroup then")
(66 . "       (SemiGroup;")
(67 . "        \"*\": (S, $) -> $;")
(70 . "        \"*\": ($, S) -> $);")
(73 . "    if S has Monoid then")
(74 . "       (Monoid;")
(75 . "        leftOne : $ -> Union($,\"failed\");")
(77 . "        rightOne : $ -> Union($,\"failed\"));")
(79 . "    if S has Group then")
(80 . "       (Group;")
(81 . "        leftOne : $ -> Union($,\"failed\");")
(83 . "        rightOne : $ -> Union($,\"failed\"));")
(85 . "    if S has Ring then")
(86 . "     (Ring;")
(87 . "      BiModule(S,S));")
(88 . "    if S has CommutativeRing then")
(89 . "      Module(S);")
(91 . "    if S has IntegralDomain then")
(92 . "      factorAndSplit : $ -> List $;")
(96 . "    if S has PartialDifferentialRing(Symbol) then")
(97 . "      PartialDifferentialRing(Symbol);")
(98 . "    if S has Field then")
(99 . "     (VectorSpace(S);")
(100 . "      \"/\": ($, $) -> $;")
(103 . "      inv: $ -> $);")
(105 . "    if S has ExpressionSpace then")
(106 . "        subst: ($, $) -> $);")
(109 . "  private ==> add")
(110 . "   (Rep := Record(lhs: S, rhs: S);")
(111 . "    eq1,eq2: $;")
(112 . "    s : S;")
(113 . "    if S has IntegralDomain then")
(114 . "        factorAndSplit eq ==")
(115 . "         ((S has factor : S -> Factored S) =>")
(116 . "           (eq0 := rightZero eq;")
(117 . "            [equation(rcf.factor,0)
```

```
                    for rcf in factors factor lhs eq0]);")
(118 . "          [eq]);")
(119 . "    l:S = r:S      == [l, r];")
(120 . "    equation(l, r) == [l, r];")
(121 . "    lhs eqn        == eqn.lhs;")
(122 . "    rhs eqn        == eqn.rhs;")
(123 . "    swap eqn     == [rhs eqn, lhs eqn];")
(124 . "    map(fn, eqn)   == equation(fn(eqn.lhs), fn(eqn.rhs));")
(125 . "    if S has InnerEvalable(Symbol,S) then")
(126 . "       (s:Symbol;")
(127 . "        ls:List Symbol;")
(128 . "        x:S;")
(129 . "        lx:List S;")
(130 . "        eval(eqn,s,x) == eval(eqn.lhs,s,x) = eval(eqn.rhs,s,x);")
(131 . "        eval(eqn,ls,lx) == eval(eqn.lhs,ls,lx) =
                                   eval(eqn.rhs,ls,lx));")
(132 . "    if S has Evalable(S) then")
(133 . "       (eval(eqn1:$, eqn2:$):$ ==")
(134 . "           eval(eqn1.lhs, eqn2 pretend Equation S) =")
(135 . "               eval(eqn1.rhs, eqn2 pretend Equation S);")
(136 . "        eval(eqn1:$, leqn2:List $):$ ==")
(137 . "           eval(eqn1.lhs, leqn2 pretend List Equation S) =")
(138 . "               eval(eqn1.rhs, leqn2 pretend List Equation S));")
(139 . "    if S has SetCategory then")
(140 . "       (eq1 = eq2 == (eq1.lhs = eq2.lhs)@Boolean and")
(141 . "                     (eq1.rhs = eq2.rhs)@Boolean;")
(142 . "        coerce(eqn:$):Ex == eqn.lhs::Ex = eqn.rhs::Ex;")
(143 . "        coerce(eqn:$):Boolean == eqn.lhs = eqn.rhs);")
(144 . "    if S has AbelianSemiGroup then")
(145 . "       (eq1 + eq2 == eq1.lhs + eq2.lhs = eq1.rhs + eq2.rhs;")
(146 . "        s + eq2 == [s,s] + eq2;")
(147 . "        eq1 + s == eq1 + [s,s]);")
(148 . "    if S has AbelianGroup then")
(149 . "       (- eq == (- lhs eq) = (-rhs eq);")
(150 . "        s - eq2 == [s,s] - eq2;")
(151 . "        eq1 - s == eq1 - [s,s];")
(152 . "        leftZero eq == 0 = rhs eq - lhs eq;")
(153 . "        rightZero eq == lhs eq - rhs eq = 0;")
(154 . "        0 == equation(0$S,0$S);")
(155 . "        eq1 - eq2 == eq1.lhs - eq2.lhs = eq1.rhs - eq2.rhs);")
(156 . "    if S has SemiGroup then")
(157 . "       (eq1:$ * eq2:$ == eq1.lhs * eq2.lhs = eq1.rhs * eq2.rhs;")
(158 . "        l:S   * eqn:$ == l       * eqn.lhs = l       * eqn.rhs;")
(159 . "        l:S * eqn:$  ==  l * eqn.lhs    =    l * eqn.rhs;")
(160 . "        eqn:$ * l:S  ==  eqn.lhs * l    =    eqn.rhs * l);")
(165 . "    if S has Monoid then")
(166 . "       (1 == equation(1$S,1$S);")
(167 . "        recip eq ==")
(168 . "         ((lh := recip lhs eq) case \"failed\" => \"failed\";")
(169 . "          (rh := recip rhs eq) case \"failed\" => \"failed\";")
```

```
(170 . "          [lh :: S, rh :: S]);")
(171 . "        leftOne eq ==")
(172 . "         ((re := recip lhs eq) case \"failed\" => \"failed\";")
(173 . "          1 = rhs eq * re);")
(174 . "        rightOne eq ==")
(175 . "         ((re := recip rhs eq) case \"failed\" => \"failed\";")
(176 . "          lhs eq * re = 1));")
(177 . "    if S has Group then")
(178 . "       (inv eq == [inv lhs eq, inv rhs eq];")
(179 . "        leftOne eq == 1 = rhs eq * inv rhs eq;")
(180 . "        rightOne eq == lhs eq * inv rhs eq = 1);")
(181 . "    if S has Ring then")
(182 . "       (characteristic() == characteristic()$S;")
(183 . "        i:Integer * eq:$ == (i::S) * eq);")
(184 . "    if S has IntegralDomain then")
(185 . "        factorAndSplit eq ==")
(186 . "         ((S has factor : S -> Factored S) =>")
(187 . "           (eq0 := rightZero eq;")
(188 . "            [equation(rcf.factor,0)
                      for rcf in factors factor lhs eq0]);")
(189 . "          (S has Polynomial Integer) =>")
(190 . "           (eq0 := rightZero eq;")
(191 . "            MF ==> MultivariateFactorize(Symbol,
                             IndexedExponents Symbol,
                             Integer, Polynomial Integer);")
(193 . "            p : Polynomial Integer :=
                           (lhs eq0) pretend Polynomial Integer;")
(194 . "            [equation((rcf.factor) pretend S,0)
                         for rcf in factors factor(p)$MF]);")
(195 . "          [eq]);")
(196 . "    if S has PartialDifferentialRing(Symbol) then")
(197 . "        differentiate(eq:$, sym:Symbol):$ ==")
(198 . "           [differentiate(lhs eq, sym), differentiate(rhs eq, sym)];")
(199 . "    if S has Field then")
(200 . "       (dimension() == 2 :: CardinalNumber;")
(201 . "        eq1:$ / eq2:$ == eq1.lhs / eq2.lhs = eq1.rhs / eq2.rhs;")
(202 . "        inv eq == [inv lhs eq, inv rhs eq]);")
(203 . "    if S has ExpressionSpace then")
(204 . "        subst(eq1,eq2) ==")
(205 . "           (eq3 := eq2 pretend Equation S;")
(206 . "            [subst(lhs eq1,eq3),subst(rhs eq1,eq3)])))")))
```

3.3.2 defun preparse

[preparse p76]
[preparse1 p81]
[parseprint p308]
[ifcar p??]

[$comblocklist p305]
[$skipme p??]
[$preparse-last-line p72]
[$index p72]
[$docList p??]
[$preparseReportIfTrue p??]
[$headerDocumentation p??]
[$maxSignatureLineNumber p??]
[$constructorLineNumber p??]

— **defun preparse** —

```
(defun preparse (strm &aux (stack ()))
 (declare (special $comblocklist $skipme $preparse-last-line $index |$docList|
           $preparseReportIfTrue |$headerDocumentation|
           |$maxSignatureLineNumber| |$constructorLineNumber|))
  (setq $comblocklist nil)
  (setq $skipme nil)
  (when $preparse-last-line
   (if (pairp $preparse-last-line)
    (setq stack $preparse-last-line)
    (push $preparse-last-line stack))
   (setq $index (- $index (length stack))))
  (let ((u (preparse1 stack)))
   (if $skipme
    (preparse strm)
    (progn
      (when $preparseReportIfTrue (parseprint u))
      (setq |$headerDocumentation| nil)
      (setq |$docList| nil)
      (setq |$maxSignatureLineNumber| 0)
      (setq |$constructorLineNumber| (ifcar (ifcar u)))
      u))))
```

———-

The **preparse** function returns a list of pairs of the form: ((linenumber . linestring) .... (linenumber . linestring)) For instance, for the file `EQ.spad`, we get:

```
    2> (PREPARSE #<input stream "/tmp/EQ.spad">)
      3> (PREPARSE1 (")abbrev domain EQ Equation"))
        4> (|doSystemCommand| "abbrev domain EQ Equation")
        <4 (|doSystemCommand| NIL)
      <3 (PREPARSE1 (
(19 . "Equation(S: Type): public == private where")
(20 . " (Ex ==> OutputForm;")
(21 . "  public ==> Type with")
```

```
(22 . "   (\"=\": (S, S) -> $;")
(24 . "    equation: (S, S) -> $;")
(26 . "    swap: $ -> $;")
(28 . "    lhs: $ -> S;")
(30 . "    rhs: $ -> S;")
(32 . "    map: (S -> S, $) -> $;")
(35 . "    if S has InnerEvalable(Symbol,S) then")
(36 . "              InnerEvalable(Symbol,S);")
(37 . "    if S has SetCategory then")
(38 . "       (SetCategory;")
(39 . "        CoercibleTo Boolean;")
(40 . "        if S has Evalable(S) then")
(41 . "          (eval: ($, $) -> $;")
(43 . "           eval: ($, List $) -> $));")
(45 . "    if S has AbelianSemiGroup then")
(46 . "       (AbelianSemiGroup;")
(47 . "        \"+\": (S, $) -> $;")
(50 . "        \"+\": ($, S) -> $);")
(53 . "    if S has AbelianGroup then")
(54 . "       (AbelianGroup;")
(55 . "        leftZero : $ -> $;")
(57 . "        rightZero : $ -> $;")
(59 . "        \"-\": (S, $) -> $;")
(62 . "        \"-\": ($, S) -> $);")
(65 . "    if S has SemiGroup then")
(66 . "       (SemiGroup;")
(67 . "        \"*\": (S, $) -> $;")
(70 . "        \"*\": ($, S) -> $);")
(73 . "    if S has Monoid then")
(74 . "       (Monoid;")
(75 . "        leftOne : $ -> Union($,\"failed\");")
(77 . "        rightOne : $ -> Union($,\"failed\"));")
(79 . "    if S has Group then")
(80 . "       (Group;")
(81 . "        leftOne : $ -> Union($,\"failed\");")
(83 . "        rightOne : $ -> Union($,\"failed\"));")
(85 . "    if S has Ring then")
(86 . "     (Ring;")
(87 . "      BiModule(S,S));")
(88 . "    if S has CommutativeRing then")
(89 . "      Module(S);")
(91 . "    if S has IntegralDomain then")
(92 . "      factorAndSplit : $ -> List $;")
(96 . "    if S has PartialDifferentialRing(Symbol) then")
(97 . "      PartialDifferentialRing(Symbol);")
(98 . "    if S has Field then")
(99 . "     (VectorSpace(S);")
(100 . "      \"/\": ($, $) -> $;")
(103 . "      inv: $ -> $);")
(105 . "    if S has ExpressionSpace then")
```

```
(106 . "        subst: ($, $) -> $);")
(109 . "  private ==> add")
(110 . "   (Rep := Record(lhs: S, rhs: S);")
(111 . "    eq1,eq2: $;")
(112 . "    s : S;")
(113 . "    if S has IntegralDomain then")
(114 . "        factorAndSplit eq ==")
(115 . "         ((S has factor : S -> Factored S) =>")
(116 . "           (eq0 := rightZero eq;")
(117 . "            [equation(rcf.factor,0)
                      for rcf in factors factor lhs eq0]);")
(118 . "          [eq]);")
(119 . "    l:S = r:S      == [l, r];")
(120 . "    equation(l, r) == [l, r];")
(121 . "    lhs eqn        == eqn.lhs;")
(122 . "    rhs eqn        == eqn.rhs;")
(123 . "    swap eqn     == [rhs eqn, lhs eqn];")
(124 . "    map(fn, eqn)   == equation(fn(eqn.lhs), fn(eqn.rhs));")
(125 . "    if S has InnerEvalable(Symbol,S) then")
(126 . "       (s:Symbol;")
(127 . "        ls:List Symbol;")
(128 . "        x:S;")
(129 . "        lx:List S;")
(130 . "        eval(eqn,s,x) == eval(eqn.lhs,s,x) = eval(eqn.rhs,s,x);")
(131 . "        eval(eqn,ls,lx) == eval(eqn.lhs,ls,lx) =
                                   eval(eqn.rhs,ls,lx));")
(132 . "    if S has Evalable(S) then")
(133 . "       (eval(eqn1:$, eqn2:$):$ ==")
(134 . "           eval(eqn1.lhs, eqn2 pretend Equation S) =")
(135 . "               eval(eqn1.rhs, eqn2 pretend Equation S);")
(136 . "        eval(eqn1:$, leqn2:List $):$ ==")
(137 . "           eval(eqn1.lhs, leqn2 pretend List Equation S) =")
(138 . "               eval(eqn1.rhs, leqn2 pretend List Equation S));")
(139 . "    if S has SetCategory then")
(140 . "       (eq1 = eq2 == (eq1.lhs = eq2.lhs)@Boolean and")
(141 . "                     (eq1.rhs = eq2.rhs)@Boolean;")
(142 . "        coerce(eqn:$):Ex == eqn.lhs::Ex = eqn.rhs::Ex;")
(143 . "        coerce(eqn:$):Boolean == eqn.lhs = eqn.rhs);")
(144 . "    if S has AbelianSemiGroup then")
(145 . "       (eq1 + eq2 == eq1.lhs + eq2.lhs = eq1.rhs + eq2.rhs;")
(146 . "        s + eq2 == [s,s] + eq2;")
(147 . "        eq1 + s == eq1 + [s,s]);")
(148 . "    if S has AbelianGroup then")
(149 . "       (- eq == (- lhs eq) = (-rhs eq);")
(150 . "        s - eq2 == [s,s] - eq2;")
(151 . "        eq1 - s == eq1 - [s,s];")
(152 . "        leftZero eq == 0 = rhs eq - lhs eq;")
(153 . "        rightZero eq == lhs eq - rhs eq = 0;")
(154 . "        0 == equation(0$S,0$S);")
(155 . "        eq1 - eq2 == eq1.lhs - eq2.lhs = eq1.rhs - eq2.rhs);")
```

```
(156 . "    if S has SemiGroup then")
(157 . "       (eq1:$ * eq2:$ == eq1.lhs * eq2.lhs = eq1.rhs * eq2.rhs;")
(158 . "        l:S   * eqn:$ == l       * eqn.lhs = l       * eqn.rhs;")
(159 . "        l:S * eqn:$  ==  l * eqn.lhs    =    l * eqn.rhs;")
(160 . "        eqn:$ * l:S  ==  eqn.lhs * l    =    eqn.rhs * l);")
(165 . "    if S has Monoid then")
(166 . "       (1 == equation(1$S,1$S);")
(167 . "        recip eq ==")
(168 . "         ((lh := recip lhs eq) case \"failed\" => \"failed\";")
(169 . "          (rh := recip rhs eq) case \"failed\" => \"failed\";")
(170 . "          [lh :: S, rh :: S]);")
(171 . "        leftOne eq ==")
(172 . "         ((re := recip lhs eq) case \"failed\" => \"failed\";")
(173 . "          1 = rhs eq * re);")
(174 . "        rightOne eq ==")
(175 . "         ((re := recip rhs eq) case \"failed\" => \"failed\";")
(176 . "          lhs eq * re = 1));")
(177 . "    if S has Group then")
(178 . "       (inv eq == [inv lhs eq, inv rhs eq];")
(179 . "        leftOne eq == 1 = rhs eq * inv rhs eq;")
(180 . "        rightOne eq == lhs eq * inv rhs eq = 1);")
(181 . "    if S has Ring then")
(182 . "       (characteristic() == characteristic()$S;")
(183 . "        i:Integer * eq:$ == (i::S) * eq);")
(184 . "    if S has IntegralDomain then")
(185 . "        factorAndSplit eq ==")
(186 . "         ((S has factor : S -> Factored S) =>")
(187 . "           (eq0 := rightZero eq;")
(188 . "            [equation(rcf.factor,0)
                      for rcf in factors factor lhs eq0]);")
(189 . "          (S has Polynomial Integer) =>")
(190 . "           (eq0 := rightZero eq;")
(191 . "            MF ==> MultivariateFactorize(Symbol,
                             IndexedExponents Symbol,
                             Integer, Polynomial Integer);")
(193 . "            p : Polynomial Integer :=
                        (lhs eq0) pretend Polynomial Integer;")
(194 . "            [equation((rcf.factor) pretend S,0)
                       for rcf in factors factor(p)$MF]);")
(195 . "          [eq]);")
(196 . "    if S has PartialDifferentialRing(Symbol) then")
(197 . "        differentiate(eq:$, sym:Symbol):$ ==")
(198 . "           [differentiate(lhs eq, sym), differentiate(rhs eq, sym)];")
(199 . "    if S has Field then")
(200 . "       (dimension() == 2 :: CardinalNumber;")
(201 . "        eq1:$ / eq2:$ == eq1.lhs / eq2.lhs = eq1.rhs / eq2.rhs;")
(202 . "        inv eq == [inv lhs eq, inv rhs eq]);")
(203 . "    if S has ExpressionSpace then")
(204 . "        subst(eq1,eq2) ==")
(205 . "           (eq3 := eq2 pretend Equation S;")
```

```
 (206 . "            [subst(lhs eq1,eq3),subst(rhs eq1,eq3)])))")))
```

3.3.3 defun Build the lines from the input for piles

The READLOOP calls preparseReadLine which returns a pair of the form

```
(number . string)
```

[preparseReadLine p88]
[preparse-echo p90]
[fincomblock p306]
[parsepiles p86]
[preparse1 doSystemCommand (vol5)]
[escaped p305]
[indent-pos p306]
[make-full-cvec p??]
[maxindex p??]
[preparse1 strposl (vol5)]
[is-console p307]
[spad-reader p??]
[$linelist p72]
[$echolinestack p72]
[$byConstructors p368]
[$skipme p??]
[$constructorsSeen p368]
[$preparse-last-line p72]

— defun preparse1 —

```
(defun preparse1 (linelist)
 (labels (
  (isSystemCommand (line)
    (and (> (length line) 0) (eq (char line 0) #\) )))
  (executeSystemCommand (line)
   (catch 'spad_reader (|doSystemCommand| (subseq line 1))))
 )
 (prog (($linelist linelist) $echolinestack num line i l psloc
        instring pcount comsym strsym oparsym cparsym n ncomsym
        (sloc -1) continue (parenlev 0) ncomblock lines locs nums functor)
 (declare (special $linelist $echolinestack |$byConstructors| $skipme
           |$constructorsSeen| $preparse-last-line))
READLOOP
  (dcq (num . line) (preparseReadLine linelist))
  (unless (stringp line)
    (preparse-echo linelist)
    (cond
     ((null lines) (return nil))
```

```
     (ncomblock    (fincomblock nil nums locs ncomblock nil)))
    (return
     (pair (nreverse nums) (parsepiles (nreverse locs) (nreverse lines)))))
  (when (and (null lines) (isSystemCommand line))
    (preparse-echo linelist)
    (setq $preparse-last-line nil) ;don't reread this line
    (executeSystemCommand line)
    (go READLOOP))
  (setq l (length line))
  ; if we get a null line, read the next line
  (when (eq l 0) (go READLOOP))
  ; otherwise we have to parse this line
  (setq psloc sloc)
  (setq i 0)
  (setq instring nil)
  (setq pcount 0)
STRLOOP ;; handle things that need ignoring, quoting, or grouping
  ; are we in a comment, quoting, or grouping situation?
  (setq strsym  (or (position #\" line :start  i ) l))
  (setq comsym  (or (search "--"  line :start2 i ) l))
  (setq ncomsym (or (search "++"  line :start2 i ) l))
  (setq oparsym (or (position #\( line :start  i ) l))
  (setq cparsym (or (position #\) line :start  i ) l))
  (setq n (min strsym comsym ncomsym oparsym cparsym))
  (cond
   ; nope, we found no comment, quoting, or grouping
   ((= n l) (go NOCOMS))
   ((escaped line n))
   ; scan until we hit the end of the string
   ((= n strsym) (setq instring (not instring)))
   ; we are in a string, just continue looping
   (instring)
   ;; handle -- comments by ignoring them
   ((= n comsym)
    (setq line (subseq line 0 n))
    (go NOCOMS)) ; discard trailing comment
   ;; handle ++ comments by chunking them together
   ((= n ncomsym)
    (setq sloc (indent-pos line))
    (cond
     ((= sloc n)
      (when (and ncomblock (not (= n (car ncomblock))))
       (fincomblock num nums locs ncomblock linelist)
       (setq ncomblock nil))
      (setq ncomblock (cons n (cons line (ifcdr ncomblock))))
      (setq line ""))
     (t
      (push (strconc (make-full-cvec n " ") (substring line n ())) $linelist)
      (setq $index (1- $index))
      (setq line (subseq line 0 n))))
```

```
    (go NOCOMS))
   ; know how deep we are into parens
   ((= n oparsym) (setq pcount (1+ pcount)))
   ((= n cparsym) (setq pcount (1- pcount))))
  (setq i (1+ n))
  (go STRLOOP)
NOCOMS
  ; remember the indentation level
  (setq sloc (indent-pos line))
  (setq line (string-right-trim " " line))
  (when (null sloc)
   (setq sloc psloc)
   (go READLOOP))
  ; handle line that ends in a continuation character
  (cond
   ((eq (elt line (maxindex line)) #\_)
    (setq continue t)
    (setq line (subseq line (maxindex line))))
   ((setq continue nil)))
  ; test for skipping constructors
  (when (and (null lines) (= sloc 0))
    (if (and |$byConstructors|
             (null (search "==>" line))
             (not
              (member
               (setq functor
                (intern (substring line 0 (strposl ": (=" line 0 nil))))
                |$byConstructors|)))
       (setq $skipme 't)
       (progn
        (push functor |$constructorsSeen|)
        (setq $skipme nil))))
  ; is this thing followed by ++ comments?
  (when (and lines (eql sloc 0))
   (when (and ncomblock (not (zerop (car ncomblock))))
    (fincomblock num nums locs ncomblock linelist))
   (when (not (is-console in-stream))
    (setq $preparse-last-line (nreverse $echolinestack)))
   (return
    (pair (nreverse nums) (parsepiles (nreverse locs) (nreverse lines)))))
  (when (> parenlev 0)
   (push nil locs)
   (setq sloc psloc)
   (go REREAD))
  (when ncomblock
   (fincomblock num nums locs ncomblock linelist)
   (setq ncomblock ()))
  (push sloc locs)
REREAD
  (preparse-echo linelist)
```

```
  (push line lines)
  (push num nums)
  (setq parenlev (+ parenlev pcount))
  (when (and (is-console in-stream) (not continue))
   (setq $preparse-last-line nil)
   (return
    (pair (nreverse nums) (parsepiles (nreverse locs) (nreverse lines)))))
  (go READLOOP))))

;(defun preparse1 (linelist)
; (prog (($linelist linelist) $echolinestack num a i l psloc
;        instring pcount comsym strsym oparsym cparsym n ncomsym
;        (sloc -1) (continue nil)  (parenlev 0) (ncomblock ())
;        (lines ()) (locs ()) (nums ()) functor)
; (declare (special $linelist $echolinestack |$byConstructors| $skipme
;           |$constructorsSeen| $preparse-last-line))
;READLOOP
;  (dcq (num . a) (preparseReadLine linelist))
;  (unless (stringp a)
;    (preparse-echo linelist)
;    (cond
;     ((null lines) (return nil))
;     (ncomblock    (fincomblock nil nums locs ncomblock nil)))
;    (return
;     (pair (nreverse nums) (parsepiles (nreverse locs) (nreverse lines)))))
;  ; this is a command line, don't parse it
;  (when (and (null lines) (> (length a) 0) (eq (char a 0) #\) ))
;    (preparse-echo linelist)
;    (setq $preparse-last-line nil) ;don't reread this line
;    (setq line a)
;    (catch 'spad_reader (|doSystemCommand| (subseq line 1)))
;    (go READLOOP))
;  (setq l (length a))
;  ; if we get a null line, read the next line
;  (when (eq l 0) (go READLOOP))
;  ; otherwise we have to parse this line
;  (setq psloc sloc)
;  (setq i 0)
;  (setq instring nil)
;  (setq pcount 0)
;STRLOOP ;; handle things that need ignoring, quoting, or grouping
;  ; are we in a comment, quoting, or grouping situation?
;  (setq strsym (or (position #\" a :start i ) l))
;  (setq comsym (or (search "--" a :start2 i ) l))
;  (setq ncomsym (or (search "++" a :start2 i ) l))
;  (setq oparsym (or (position #\( a :start i ) l))
;  (setq cparsym (or (position #\) a :start i ) l))
;  (setq n (min strsym comsym ncomsym oparsym cparsym))
;  (cond
;   ; nope, we found no comment, quoting, or grouping
```

```
;   ((= n l) (go NOCOMS))
;   ((escaped a n))
;   ; scan until we hit the end of the string
;   ((= n strsym) (setq instring (not instring)))
;   (instring)
;   ;; handle -- comments by ignoring them
;   ((= n comsym)
;    (setq a (subseq a 0 n))
;    (go NOCOMS)) ; discard trailing comment
;   ;; handle ++ comments by chunking them together
;   ((= n ncomsym)
;    (setq sloc (indent-pos a))
;    (cond
;     ((= sloc n)
;      (when (and ncomblock (not (= n (car ncomblock))))
;       (fincomblock num nums locs ncomblock linelist)
;       (setq ncomblock nil))
;      (setq ncomblock (cons n (cons a (ifcdr ncomblock))))
;      (setq a ""))
;     (t
;      (push (strconc (make-full-cvec n " ") (substring a n ())) $linelist)
;      (setq $index (1- $index))
;      (setq a (subseq a 0 n))))
;    (go NOCOMS))
;   ; know how deep we are into parens
;   ((= n oparsym) (setq pcount (1+ pcount)))
;   ((= n cparsym) (setq pcount (1- pcount))))
;  (setq i (1+ n))
;  (go STRLOOP)
;NOCOMS
;  ; remember the indentation level
;  (setq sloc (indent-pos a))
;  (setq a (string-right-trim " " a))
;  (when (null sloc)
;   (setq sloc psloc)
;   (go READLOOP))
;  ; handle line that ends in a continuation character
;  (cond
;   ((eq (elt a (maxindex a)) #\_)
;    (setq continue t)
;    (setq a (subseq a (maxindex a))))
;   ((setq continue nil)))
;  ; test for skipping constructors
;  (when (and (null lines) (= sloc 0))
;    (if (and |$byConstructors|
;             (null (search "==>" a))
;             (not
;              (member
;               (setq functor
;                (intern (substring a 0 (strposl ": (=" a 0 nil))))
```

```
;                |$byConstructors|)))
;       (setq $skipme 't)
;       (progn
;        (push functor |$constructorsSeen|)
;        (setq $skipme nil))))
;  ; is this thing followed by ++ comments?
;  (when (and lines (eql sloc 0))
;   (when (and ncomblock (not (zerop (car ncomblock))))
;    (fincomblock num nums locs ncomblock linelist))
;   (when (not (is-console in-stream))
;    (setq $preparse-last-line (nreverse $echolinestack)))
;   (return
;    (pair (nreverse nums) (parsepiles (nreverse locs) (nreverse lines)))))
;  (when (> parenlev 0)
;   (push nil locs)
;   (setq sloc psloc)
;   (go REREAD))
;  (when ncomblock
;   (fincomblock num nums locs ncomblock linelist)
;   (setq ncomblock ()))
;  (push sloc locs)
;REREAD
;  (preparse-echo linelist)
;  (push a lines)
;  (push num nums)
;  (setq parenlev (+ parenlev pcount))
;  (when (and (is-console in-stream) (not continue))
;   (setq $preparse-last-line nil)
;   (return
;    (pair (nreverse nums) (parsepiles (nreverse locs) (nreverse lines)))))
;  (go READLOOP)))
```

———-

3.3.4 defun parsepiles

Add parens and semis to lines to aid parsing. [add-parens-and-semis-to-line p87]

— **defun parsepiles** —

```
(defun parsepiles (locs lines)
  (mapl #'add-parens-and-semis-to-line
    (nconc lines '(" ")) (nconc locs '(nil)))
  lines)
```

———-

3.3.5 defun add-parens-and-semis-to-line

The line to be worked on is (CAR SLINES). It's indentation is (CAR SLOCS). There is a notion of current indentation. Then:

- Add open paren to beginning of following line if following line's indentation is greater than current, and add close paren to end of last succeeding line with following line's indentation.
- Add semicolon to end of line if following line's indentation is the same.
- If the entire line consists of the single keyword then or else, leave it alone."

[infixtok p307]
[drop p305]
[addclose p304]
[nonblankloc p308]

— defun add-parens-and-semis-to-line —

```
(defun add-parens-and-semis-to-line (slines slocs)
 (let ((start-column (car slocs)))
  (when (and start-column (> start-column 0))
   (let ((count 0) (i 0))
    (seq
     (mapl #'(lambda (next-lines nlocs)
              (let ((next-line (car next-lines)) (next-column (car nlocs)))
               (incf i)
               (when next-column
                (setq next-column (abs next-column))
                (when (< next-column start-column) (exit nil))
                (cond
                 ((and (eq next-column start-column)
                       (rplaca nlocs (- (car nlocs)))
                       (not (infixtok next-line)))
                   (setq next-lines (drop (1- i) slines))
                   (rplaca next-lines (addclose (car next-lines) #\;))
                   (setq count (1+ count)))))))
               (cdr slines) (cdr slocs)))
     (when (> count 0)
      (setf (char (car slines) (1- (nonblankloc (car slines)))) #\( )
      (setq slines (drop (1- i) slines))
      (rplaca slines (addclose (car slines) #\) )))))))
```

———-

3.3.6 defun preparseReadLine

[dcq p??]
[preparseReadLine1 p89]
[initial-substring p96]
[string2BootTree p??]
[storeblanks p95]
[skip-to-endif p308]
[preparseReadLine p88]

— **defun preparseReadLine** —

```
(defun preparseReadLine (x)
 (let (line ind)
  (dcq (ind . line) (preparseReadLine1))
  (cond
   ((not (stringp line)) (cons ind line))
   ((zerop (size line))  (cons ind line))
   ((char= (elt line 0) #\) )
    (cond
     ((initial-substring ")if" line)
      (if (eval (|string2BootTree| (storeblanks line 3)))
       (preparseReadLine x)
       (skip-ifblock x)))
     ((initial-substring ")elseif" line) (skip-to-endif x))
     ((initial-substring ")else" line)   (skip-to-endif x))
     ((initial-substring ")endif" line)  (preparseReadLine x))
     ((initial-substring ")fin" line)
      (setq *eof* t)
      (cons ind nil)))))
  (cons ind line)))
```

————-

3.3.7 defun skip-ifblock

[preparseReadLine1 p89]
[skip-ifblock p88]
[initial-substring p96]
[string2BootTree p??]
[storeblanks p95]

— **defun skip-ifblock** —

```
(defun skip-ifblock (x)
 (let (line ind)
```

```
  (dcq (ind . line) (preparseReadLine1))
  (cond
   ((not (stringp line))
     (cons ind line))
   ((zerop (size line))
     (skip-ifblock x))
   ((char= (elt line 0) #\) )
    (cond
     ((initial-substring ")if" line)
      (cond
        ((eval (|string2BootTree| (storeblanks line 3)))
          (preparseReadLine X))
        (t (skip-ifblock x))))
     ((initial-substring ")elseif" line)
       (cond
        ((eval (|string2BootTree| (storeblanks line 7)))
          (preparseReadLine X))
        (t (skip-ifblock x))))
     ((initial-substring ")else" line)
       (preparseReadLine x))
     ((initial-substring ")endif" line)
       (preparseReadLine x))
     ((initial-substring ")fin" line)
       (cons ind nil))))
   (t (skip-ifblock x)))))
```

———-

3.3.8 defun preparseReadLine1

[get-a-line p96]
[expand-tabs p??]
[maxindex p??]
[strconc p??]
[preparseReadLine1 p89]
[$linelist p72]
[$preparse-last-line p72]
[$index p72]
[$EchoLineStack p??]

— **defun preparseReadLine1** —

```
(defun preparseReadLine1 ()
 (labels (
  (accumulateLinesWithTrailingEscape (line)
   (let (ind)
   (declare (special $preparse-last-line))
```

```
    (if (and (> (setq ind (maxindex line)) -1) (char= (elt line ind) #\_))
     (setq $preparse-last-line
       (strconc (substring line 0 ind) (cdr (preparseReadLine1))))
     line))))
(let (line)
(declare (special $linelist $preparse-last-line $index $EchoLineStack))
 (setq line
  (if $linelist
   (pop $linelist)
   (expand-tabs (get-a-line in-stream))))
 (setq $preparse-last-line line)
 (if (stringp line)
  (progn
   (incf $index)   ;; $index is the current line number
   (setq line (string-right-trim " " line))
   (push (copy-seq line) $EchoLineStack)
   (cons $index (accumulateLinesWithTrailingEscape line)))
  (cons $index line)))))
```

———-

3.4 I/O Handling

3.4.1 defun preparse-echo

[Echo-Meta p??]
[$EchoLineStack p??]

— **defun preparse-echo** —

```
(defun preparse-echo (linelist)
 (declare (special $EchoLineStack Echo-Meta) (ignore linelist))
 (if Echo-Meta
  (dolist (x (reverse $EchoLineStack))
   (format out-stream "~&;~A~%" x)))
 (setq $EchoLineStack ()))
```

———-

3.4.2 defvar $current-fragment

A string containing remaining chars from readline; needed because Symbolics read-line returns embedded newlines in a c-m-Y.

— **initvars** —

```
(defvar current-fragment nil)
```

———-

3.4.3 defun read-a-line

[subseq p??]
[Line-New-Line p??]
[read-a-line p91]
[*eof* p??]

— **defun read-a-line** —

```
(defun read-a-line (&optional (stream t))
 (let (cp)
 (declare (special *eof*))
  (if (and Current-Fragment (> (length Current-Fragment) 0))
   (let ((line (with-input-from-string
                (s Current-Fragment :index cp :start 0)
                (read-line s nil nil))))
    (setq Current-Fragment (subseq Current-Fragment cp))
    line)
   (prog nil
    (when (stream-eof in-stream)
      (setq File-Closed t)
      (setq *eof* t)
      (Line-New-Line (make-string 0) Current-Line)
      (return nil))
    (when (setq Current-Fragment (read-line stream))
     (return (read-a-line stream)))))))
```

———-

3.5 Line Handling

3.5.1 Line Buffer

The philosophy of lines is that

- NEXT LINE will always return a non-blank line or fail.
- Every line is terminated by a blank character.

Hence there is always a current character, because there is never a non-blank line, and there is always a separator character between tokens on separate lines.

Also, when a line is read, the character pointer is always positioned ON the first character.

3.5.2 defstruct $line

— **initvars** —

```
(defstruct line "Line of input file to parse."
           (buffer (make-string 0) :type string)
           (current-char #\Return :type character)
           (current-index 1 :type fixnum)
           (last-index 0 :type fixnum)
           (number 0 :type fixnum))
```

———-

3.5.3 defvar $current-line

The current input line.

— **initvars** —

```
(defvar current-line (make-line))
```

———-

3.5.4 defmacro line-clear

[$line p92]

— **defmacro line-clear** —

```
(defmacro line-clear (line)
 `(let ((l ,line))
   (setf (line-buffer l) (make-string 0))
   (setf (line-current-char l) #\return)
   (setf (line-current-index l) 1)
   (setf (line-last-index l) 0)
   (setf (line-number l) 0)))
```

———-

3.5.5 defun line-print

[$line p92]

— **defun line-print** —

```
(defun line-print (line)
  (format out-stream "~&~5D> ~A~%" (Line-Number line) (Line-Buffer Line))
  (format out-stream "~v@T^~%" (+ 7 (Line-Current-Index line))))
```

———-

3.5.6 defun line-at-end-p

[$line p92]

— **defun line-at-end-p** —

```
(defun line-at-end-p (line)
  "Tests if line is empty or positioned past the last character."
  (>= (line-current-index line) (line-last-index line)))
```

———-

3.5.7 defun line-past-end-p

[$line p92]

— **defun line-past-end-p** —

```
(defun line-past-end-p (line)
  "Tests if line is empty or positioned past the last character."
  (> (line-current-index line) (line-last-index line)))
```

———-

3.5.8 defun line-next-char

[$line p92]

— **defun line-next-char** —

```
(defun line-next-char (line)
  (elt (line-buffer line) (1+ (line-current-index line))))
```

———-

3.5.9 defun line-advance-char

[$line p92]

— **defun line-advance-char** —

```
(defun line-advance-char (line)
  (setf (line-current-char line)
        (elt (line-buffer line) (incf (line-current-index line)))))
```

———-

3.5.10 defun line-current-segment

[$line p92]

— **defun line-current-segment** —

```
(defun line-current-segment (line)
  "Buffer from current index to last index."
  (if (line-at-end-p line)
   (make-string 0)
   (subseq (line-buffer line)
           (line-current-index line)
           (line-last-index line))))
```

———-

3.5.11 defun line-new-line

[$line p92]

— **defun line-new-line** —

```
(defun line-new-line (string line &optional (linenum nil))
  "Sets string to be the next line stored in line."
  (setf (line-last-index line) (1- (length string)))
  (setf (line-current-index line) 0)
```

```
  (setf (line-current-char line)
        (or (and (> (length string) 0) (elt string 0)) #\Return))
  (setf (line-buffer line) string)
  (setf (line-number line) (or linenum (1+ (line-number line)))))
```

———-

3.5.12 defun next-line

— **defun next-line** —

```
(defun next-line (&optional (in-stream t))
 (funcall Line-Handler in-stream))
```

———-

3.5.13 defun Advance-Char

[Line-At-End-P p??]
[Line-Advance-Char p??]
[next-line p95]
[current-char p287]
[$line p92]

— **defun Advance-Char** —

```
(defun Advance-Char ()
  "Advances IN-STREAM, invoking Next Line if necessary."
 (loop
  (cond
   ((not (Line-At-End-P Current-Line))
    (return (Line-Advance-Char Current-Line)))
   ((next-line in-stream)
    (return (current-char)))
   ((return nil)))))
```

———-

3.5.14 defun storeblanks

— **defun storeblanks** —

```
(defun storeblanks (line n)
 (do ((i 0 (1+ i)))
     ((= i n) line)
  (setf (char line i) #\ )))
```

———-

3.5.15 defun initial-substring

[mismatch p??]

— **defun initial-substring** —

```
(defun initial-substring (pattern line)
   (let ((ind (mismatch pattern line)))
     (or (null ind) (eql ind (size pattern)))))
```

———-

3.5.16 defun get-a-line

[is-console p307]
[get-a-line mkprompt (vol5)]
[read-a-line p91]
[make-string-adjustable p96]

— **defun get-a-line** —

```
(defun get-a-line (stream)
 (when (is-console stream) (princ (mkprompt)))
 (let ((ll (read-a-line stream)))
  (if (stringp ll)
   (make-string-adjustable ll)
   ll)))
```

———-

3.5.17 defun make-string-adjustable

— **defun make-string-adjustable** —

```
(defun make-string-adjustable (s)
 (if (adjustable-array-p s)
  s
  (make-array (array-dimensions s) :element-type 'string-char
                  :adjustable t :initial-contents s)))
```

———-

3.5.18 Parsing stack

3.5.19 defstruct $stack

— **initvars** —

```
(defstruct stack           "A stack"
           (store nil)     ; contents of the stack
           (size 0)        ; number of elements in Store
           (top nil)       ; first element of Store
           (updated nil)   ; whether something has been pushed on the stack
                           ; since this flag was last set to NIL
)
```

———-

3.5.20 defun stack-load

[$stack p97]

— **defun stack-load** —

```
(defun stack-load (list stack)
  (setf (stack-store stack) list)
  (setf (stack-size stack) (length list))
  (setf (stack-top stack) (car list)))
```

———-

3.5.21 defun stack-clear

[$stack p97]

— **defun stack-clear** —

```
(defun stack-clear (stack)
  (setf (stack-store stack) nil)
  (setf (stack-size stack) 0)
  (setf (stack-top stack) nil)
  (setf (stack-updated stack) nil))
```

———-

3.5.22 defmacro stack-/-empty

[$stack p97]

— **defmacro stack-/-empty** —

```
(defmacro stack-/-empty (stack) `(> (stack-size ,stack) 0))
```

———-

3.5.23 defun stack-push

[$stack p97]

— **defun stack-push** —

```
(defun stack-push (x stack)
  (push x (stack-store stack))
  (setf (stack-top stack) x)
  (setf (stack-updated stack) t)
  (incf (stack-size stack))
  x)
```

———-

3.5.24 defun stack-pop

[$stack p97]

— **defun stack-pop** —

```
(defun stack-pop (stack)
  (let ((y (pop (stack-store stack))))
    (decf (stack-size stack))
    (setf (stack-top stack)
```

```
        (if (stack-/-empty stack) (car (stack-store stack))))
    y))
```

———-

3.5.25 Parsing token

3.5.26 defstruct $token

A token is a Symbol with a Type. The type is either NUMBER, IDENTIFIER or SPECIAL-CHAR. NonBlank is true if the token is not preceded by a blank.

— **initvars** —

```
(defstruct token
  (symbol nil)
  (type nil)
  (nonblank t))
```

———-

3.5.27 defvar $prior-token

[$token p99]

— **initvars** —

```
(defvar prior-token (make-token) "What did I see last")
```

———-

3.5.28 defvar $nonblank

— **initvars** —

```
(defvar nonblank t "Is there no blank in front of the current token.")
```

———-

3.5.29 defvar $current-token

Token at head of input stream. [$token p99]

— **initvars** —

```
(defvar current-token (make-token))
```

————-

3.5.30 defvar $next-token

[$token p99]

— **initvars** —

```
(defvar next-token (make-token) "Next token in input stream.")
```

————-

3.5.31 defvar $valid-tokens

[$token p99]

— **initvars** —

```
(defvar valid-tokens 0 "Number of tokens in buffer (0, 1 or 2)")
```

————-

3.5.32 defun token-install

[$token p99]

— **defun token-install** —

```
(defun token-install (symbol type token &optional (nonblank t))
  (setf (token-symbol token) symbol)
  (setf (token-type token) type)
  (setf (token-nonblank token) nonblank)
  token)
```

————-

3.5.33 defun token-print

[$token p99]

— **defun token-print** —

```
(defun token-print (token)
  (format out-stream "(token (symbol ~S) (type ~S))~%"
          (token-symbol token) (token-type token)))
```

———-

3.5.34 Parsing reduction

3.5.35 defstruct $reduction

A reduction of a rule is any S-Expression the rule chooses to stack.

— **initvars** —

```
(defstruct (reduction (:type list))
  (rule nil)            ; Name of rule
  (value nil))
```

———-

Chapter 4

Parse Transformers

4.1 Direct called parse routines

4.1.1 defun parseTransform

[msubst p??]
[parseTran p103]
[$defOp p??]

— **defun parseTransform** —

```
(defun |parseTransform| (x)
 (let (|$defOp|)
 (declare (special |$defOp|))
  (setq |$defOp| nil)
  (setq x (msubst '$ '% x)) ; for new compiler compatibility
  (|parseTran| x)))
```

———-

4.1.2 defun parseTran

[parseAtom p104]
[parseConstruct p105]
[parseTran p103]
[parseTranList p105]
[getl p??]
[$op p??]

— **defun parseTran** —

```
(defun |parseTran| (x)
 (labels (
  (g (op)
   (let (tmp1 tmp2 x)
    (seq
     (if (and (pairp op) (eq (qcar op) '|elt|)
         (progn
          (setq tmp1 (qcdr op))
          (and (pairp tmp1)
               (progn
                (setq op (qcar tmp1))
                (setq tmp2 (qcdr tmp1))
                (and (pairp tmp2)
                     (eq (qcdr tmp2) nil)
                         (progn (setq x (qcar tmp2)) t))))))
        (exit (g x)))
     (exit op)))))
 (let (|$op| argl u r fn)
 (declare (special |$op|))
  (setq |$op| nil)
  (if (atom x)
   (|parseAtom| x)
   (progn
    (setq |$op| (car x))
    (setq argl (cdr x))
    (setq u (g |$op|))
    (cond
     ((eq u '|construct|)
       (setq r (|parseConstruct| argl))
       (if (and (pairp |$op|) (eq (qcar |$op|) '|elt|))
        (cons (|parseTran| |$op|) (cdr r))
        r))
     ((and (atom u) (setq fn (getl u '|parseTran|)))
       (funcall fn argl))
     (t (cons (|parseTran| |$op|) (|parseTranList| argl)))))))))
```

———-

4.1.3 defun parseAtom

[parseLeave p128]
[$NoValue p??]

— defun parseAtom —

```
(defun |parseAtom| (x)
 (declare (special |$NoValue|))
```

```
 (if (eq x '|break|)
  (|parseLeave| (list '|$NoValue|))
  x))
```

———-

4.1.4 defun parseTranList

[parseTran p103]
[parseTranList p105]

— **defun parseTranList** —

```
(defun |parseTranList| (x)
 (if (atom x)
  (|parseTran| x)
  (cons (|parseTran| (car x)) (|parseTranList| (cdr x)))))
```

———-

4.1.5 defun parseConstruct

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|construct| '|parseTran|) '|parseConstruct|))
```

———-

4.1.6 defun parseConstruct

[parseTranList p105]
[$insideConstructIfTrue p**??**]

— **defun parseConstruct** —

```
(defun |parseConstruct| (u)
 (let (|$insideConstructIfTrue| x)
 (declare (special |$insideConstructIfTrue|))
  (setq |$insideConstructIfTrue| t)
  (setq x (|parseTranList| u))
  (cons '|construct| x)))
```

4.2 Indirect called parse routines

In the **parseTran** function there is the code:

```
((and (atom u) (setq fn (getl u '|parseTran|)))
  (funcall fn argl))
```

The functions in this section are called through the symbol-plist of the symbol being parsed. The original list read:

```
  and          parseAnd
  @            parseAtSign
  CATEGORY     parseCategory
  ::           parseCoerce
  \:           parseColon
  construct    parseConstruct
  DEF          parseDEF
  $<=          parseDollarLessEqual
  $>           parseDollarGreaterThan
  $>=          parseDollarGreaterEqual
  $^=          parseDollarNotEqual
  eqv          parseEquivalence
;;xor          parseExclusiveOr
  exit         parseExit
  >            parseGreaterThan
  >=           parseGreaterEqual
  has          parseHas
  IF           parseIf
  implies      parseImplies
  IN           parseIn
  INBY         parseInBy
  is           parseIs
  isnt         parseIsnt
  Join         parseJoin
  leave        parseLeave
;;control-H    parseLeftArrow
  <=           parseLessEqual
  LET          parseLET
  LETD         parseLETD
  MDEF         parseMDEF
  ^            parseNot
  not          parseNot
  ^=           parseNotEqual
  or           parseOr
  pretend      parsePretend
  return       parseReturn
```

```
  SEGMENT      parseSegment
  SEQ          parseSeq
;;control-V    parseUpArrow
  VCONS        parseVCONS
  where        parseWhere
```

4.2.1 defun parseAnd

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|and| '|parseTran|) '|parseAnd|))
```

————-

4.2.2 defun parseAnd

[parseTran p103]
[parseAnd p107]
[parseTranList p105]
[parseIf p123]
[$InteractiveMode p??]

— **defun parseAnd** —

```
(defun |parseAnd| (arg)
 (cond
  (|$InteractiveMode| (cons '|and| (|parseTranList| arg)))
  ((null arg) '|true|)
  ((null (cdr arg)) (car arg))
  (t
   (|parseIf|
    (list (|parseTran| (car arg)) (|parseAnd| (CDR arg)) '|false| )))))
```

————-

4.2.3 defun parseAtSign

— **postvars** —

```
(eval-when (eval load)
 (setf (get '@ '|parseTran|) '|parseAtSign|))
```

———-

4.2.4 defun parseAtSign

[parseTran p103]
[parseType p108]
[$InteractiveMode p??]

— **defun parseAtSign** —

```
(defun |parseAtSign| (arg)
 (declare (special |$InteractiveMode|))
 (if |$InteractiveMode|
  (list '@ (|parseTran| (first arg)) (|parseTran| (|parseType| (second arg))))
  (list '@ (|parseTran| (first arg)) (|parseTran| (second arg)))))
```

———-

4.2.5 defun parseType

[msubst p??]
[parseTran p103]

— **defun parseType** —

```
(defun |parseType| (x)
 (declare (special |$EmptyMode| |$quadSymbol|))
 (setq x (msubst |$EmptyMode| |$quadSymbol| x))
 (if (and (pairp x) (eq (qcar x) '|typeOf|)
          (pairp (qcdr x)) (eq (qcdr (qcdr x)) nil))
    (list '|typeOf| (|parseTran| (qcar (qcdr x))))
    x))
```

———-

4.2.6 defun parseCategory

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'category '|parseTran|) '|parseCategory|))
```

———-

4.2.7 defun parseCategory

[parseTranList p105]
[parseDropAssertions p109]
[contained p??]

— **defun parseCategory** —

```
(defun |parseCategory| (arg)
 (let (z key)
  (setq z (|parseTranList| (|parseDropAssertions| arg)))
  (setq key (if (contained '$ z) '|domain| '|package|))
  (cons 'category (cons key z))))
```

———-

4.2.8 defun parseDropAssertions

[parseDropAssertions p109]

— **defun parseDropAssertions** —

```
(defun |parseDropAssertions| (x)
 (cond
  ((not (pairp x)) x)
  ((and (pairp (qcar x)) (eq (qcar (qcar x)) 'if)
        (pairp (qcdr (qcar x)))
        (eq (qcar (qcdr (qcar x))) '|asserted|))
     (|parseDropAssertions| (qcdr x)))
  (t (cons (qcar x) (|parseDropAssertions| (qcdr x))))))
```

———-

4.2.9 defun parseCoerce

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|::| '|parseTran|) '|parseCoerce|))
```

———-

4.2.10 defun parseCoerce

[parseType p108]
[parseTran p103]
[$InteractiveMode p??]

— **defun parseCoerce** —

```
(defun |parseCoerce| (arg)
 (if |$InteractiveMode|
  (list '|::|
    (|parseTran| (first arg)) (|parseTran| (|parseType| (second arg))))
  (list '|::| (|parseTran| (first arg)) (|parseTran| (second arg)))))
```

———-

4.2.11 defun parseColon

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|:| '|parseTran|) '|parseColon|))
```

———-

4.2.12 defun parseColon

[parseTran p103]
[parseType p108]
[$InteractiveMode p??]
[$insideConstructIfTrue p??]

— **defun parseColon** —

```
(defun |parseColon| (arg)
 (cond
  ((and (pairp arg) (eq (qcdr arg) nil))
   (list '|:| (|parseTran| (first arg))))
  ((and (pairp arg) (pairp (qcdr arg)) (eq (qcdr (qcdr arg)) nil))
    (if |$InteractiveMode|
      (if |$insideConstructIfTrue|
        (list 'tag (|parseTran| (first arg))
                   (|parseTran| (second arg)))
```

```
        (list '|:| (|parseTran| (first arg))
                   (|parseTran| (|parseType| (second arg)))))
      (list '|:| (|parseTran| (first arg))
                 (|parseTran| (second arg)))))))
```

———-

4.2.13 defun parseDEF

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'def '|parseTran|) '|parseDEF|))
```

———-

4.2.14 defun parseDEF

[setDefOp p225]
[parseLhs p112]
[parseTranList p105]
[parseTranCheckForRecord p297]
[opFf p**??**]
[$lhs p**??**]

— **defun parseDEF** —

```
(defun |parseDEF| (arg)
 (let (|$lhs| tList specialList body)
 (declare (special |$lhs|))
  (setq |$lhs| (first arg))
  (setq tList (second arg))
  (setq specialList (third arg))
  (setq body (fourth arg))
  (|setDefOp| |$lhs|)
  (list 'def (|parseLhs| |$lhs|)
             (|parseTranList| tList)
             (|parseTranList| specialList)
             (|parseTranCheckForRecord| body (|opOf| |$lhs|)))))
```

———-

4.2.15 defun parseLhs

[parseTran p103]
[transIs p112]

— **defun parseLhs** —

```
(defun |parseLhs| (x)
 (let (result)
  (cond
   ((atom x) (|parseTran| x))
   ((atom (car x))
     (cons (|parseTran| (car x))
       (dolist (y (cdr x) (nreverse result))
         (push (|transIs| (|parseTran| y)) result))))
   (t (|parseTran| x)))))
```

———-

4.2.16 defun transIs

[isListConstructor p113]
[transIs1 p112]

— **defun transIs** —

```
(defun |transIs| (u)
  (if (|isListConstructor| u)
    (cons '|construct| (|transIs1| u))
    u))
```

———-

4.2.17 defun transIs1

[qcar p**??**]
[qcdr p**??**]
[pairp p**??**]
[nreverse0 p**??**]
[transIs p112]
[transIs1 p112]

— **defun transIs1** —

```
(defun |transIs1| (u)
 (let (x h v tmp3)
  (cond
   ((and (pairp u) (eq (qcar u) '|construct|))
     (dolist (x (qcdr u) (nreverse0 tmp3))
       (push (|transIs| x) tmp3)))
   ((and (pairp u) (eq (qcar u) '|append|) (pairp (qcdr u))
         (pairp (qcdr (qcdr u))) (eq (qcdr (qcdr (qcdr u))) nil))
     (setq x (qcar (qcdr u)))
     (setq h (list '|:| (|transIs| x)))
     (setq v (|transIs1| (qcar (qcdr (qcdr u)))))
     (cond
      ((and (pairp v) (eq (qcar v) '|:|)
            (pairp (qcdr v)) (eq (qcdr (qcdr v)) nil))
         (list h (qcar (qcdr v))))
      ((eq v '|nil|) (car (cdr h)))
      ((atom v) (list h (list '|:| v)))
      (t (cons h v))))
   ((and (pairp u) (eq (qcar u) '|cons|) (pairp (qcdr u))
         (pairp (qcdr (qcdr u))) (eq (qcdr (qcdr (qcdr u))) nil))
     (setq h (|transIs| (qcar (qcdr u))))
     (setq v (|transIs1| (qcar (qcdr (qcdr u)))))
     (cond
      ((and (pairp v) (eq (qcar v) '|:|) (pairp (qcdr v))
            (eq (qcdr (qcdr v)) nil))
         (cons h (list (qcar (qcdr v)))))
      ((eq v '|nil|) (cons h nil))
      ((atom v) (list h (list '|:| v)))
      (t (cons h v))))
   (t u))))
```

———-

4.2.18 defun isListConstructor

[member p**??**]

— **defun isListConstructor** —

```
(defun |isListConstructor| (u)
 (and (pairp u) (|member| (qcar u) '(|construct| |append| |cons|))))
```

———-

4.2.19 defun parseDollarGreaterthan

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|$>| '|parseTran|) '|parseDollarGreaterThan|))
```

————-

4.2.20 defun parseDollarGreaterThan

[msubst p**??**]
[parseTran p103]
[$op p**??**]

— **defun parseDollarGreaterThan** —

```
(defun |parseDollarGreaterThan| (arg)
 (declare (special |$op|))
  (list (msubst '$< '$> |$op|)
        (|parseTran| (second arg))
        (|parseTran| (first arg))))
```

————-

4.2.21 defun parseDollarGreaterEqual

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|$>=| '|parseTran|) '|parseDollarGreaterEqual|))
```

————-

4.2.22 defun parseDollarGreaterEqual

[msubst p**??**]
[parseTran p103]
[$op p**??**]

— **defun parseDollarGreaterEqual** —

```
(defun |parseDollarGreaterEqual| (arg)
 (declare (special |$op|))
  (|parseTran| (list '|not| (cons (msubst '$< '$>= |$op|) arg))))
```

———-

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|$<=| '|parseTran|) '|parseDollarLessEqual|))
```

———-

4.2.23 defun parseDollarLessEqual

[msubst p**??**]
[parseTran p103]
[$op p**??**]

— **defun parseDollarLessEqual** —

```
(defun |parseDollarLessEqual| (arg)
 (declare (special |$op|))
  (|parseTran| (list '|not| (cons (msubst '$> '$<= |$op|) arg))))
```

———-

4.2.24 defun parseDollarNotEqual

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|$^=| '|parseTran|) '|parseDollarNotEqual|))
```

———-

4.2.25 defun parseDollarNotEqual

[parseTran p103]
[msubst p**??**]

[$op p**??**]

— **defun parseDollarNotEqual** —

```
(defun |parseDollarNotEqual| (arg)
 (declare (special |$op|))
 (|parseTran| (list '|not| (cons (msubst '$= '$^= |$op|) arg))))
```

———-

4.2.26 defun parseEquivalence

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|eqv| '|parseTran|) '|parseEquivalence|))
```

———-

4.2.27 defun parseEquivalence

[parseIf p123]

— **defun parseEquivalence** —

```
(defun |parseEquivalence| (arg)
  (|parseIf|
   (list (first arg) (second arg)
     (|parseIf| (cons (second arg) '(|false| |true|))))))
```

———-

4.2.28 defun parseExit

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|exit| '|parseTran|) '|parseExit|))
```

———-

4.2.29 defun parseExit

[parseTran p103]
[moan p??]

— **defun parseExit** —

```
(defun |parseExit| (arg)
 (let (a b)
  (setq a (|parseTran| (car arg)))
  (setq b (|parseTran| (cdr arg)))
  (if b
    (cond
     ((null (integerp a))
       (moan "first arg " a " for exit must be integer")
       (list '|exit| 1 a ))
     (t
      (cons '|exit| (cons a b))))
    (list '|exit| 1 a ))))
```

———-

4.2.30 defun parseGreaterEqual

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|>=| '|parseTran|) '|parseGreaterEqual|))
```

———-

4.2.31 defun parseGreaterEqual

[parseTran p103]
[$op p??]

— **defun parseGreaterEqual** —

```
(defun |parseGreaterEqual| (arg)
 (declare (special |$op|))
 (|parseTran| (list '|not| (cons (msubst '< '>= |$op|) arg))))
```

———-

4.2.32 defun parseGreaterThan

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|>| '|parseTran|) '|parseGreaterThan|))
```

————-

4.2.33 defun parseGreaterThan

[parseTran p103]
 [$op p**??**]

— **defun parseGreaterThan** —

```
(defun |parseGreaterThan| (arg)
 (declare (special |$op|))
 (list (msubst '< '> |$op|)
  (|parseTran| (second arg)) (|parseTran| (first arg))))
```

————-

4.2.34 defun parseHas

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|has| '|parseTran|) '|parseHas|))
```

————-

4.2.35 defun parseHas

[unabbrevAndLoad p**??**]
 [qcar p**??**]
 [qcdr p**??**]
 [getdatabase p**??**]
 [opOf p**??**]
 [makeNonAtomic p**??**]

[parseHasRhs p120]
[member p**??**]
[parseType p108]
[nreverse0 p**??**]
[$InteractiveMode p**??**]
[$CategoryFrame p**??**]

— defun parseHas —

```
(defun |parseHas| (arg)
 (labels (
  (fn (arg)
   (let (tmp4 tmp6 map op kk)
   (declare (special |$InteractiveMode|))
    (when |$InteractiveMode| (setq arg (|unabbrevAndLoad| arg)))
    (cond
     ((and (pairp arg) (eq (qcar arg) '|:|) (pairp (qcdr arg))
               (pairp (qcdr (qcdr arg))) (eq (qcdr (qcdr (qcdr arg))) nil)
               (pairp (qcar (qcdr (qcdr arg))))
               (eq (qcar (qcar (qcdr (qcdr arg)))) '|Mapping|))
        (setq map (rest (third arg)))
        (setq op (second arg))
        (setq op (if (stringp op) (intern op) op))
        (list (list 'signature op map)))
     ((and (pairp arg) (eq (qcar arg) '|Join|))
        (dolist (z (rest arg) tmp4)
         (setq tmp4 (append tmp4 (fn z)))))
     ((and (pairp arg) (eq (qcar arg) 'category))
        (dolist (z (rest arg) tmp6)
          (setq tmp6 (append tmp6 (fn z)))))
     (t
      (setq kk (getdatabase (|opOf| arg) 'constructorkind))
      (cond
       ((or (eq kk '|domain|) (eq kk '|category|))
         (list (|makeNonAtomic| arg)))
       ((and (pairp arg) (eq (qcar arg) 'attribute))
         (list arg))
      ((and (pairp arg) (eq (qcar arg) 'signature))
        (list arg))
      (|$InteractiveMode|
        (|parseHasRhs| arg))
      (t
        (list (list 'attribute arg)))))))))
 (let (tmp1 tmp2 tmp3 x)
 (declare (special |$InteractiveMode| |$CategoryFrame|))
  (setq x (first arg))
  (setq tmp1 (|get| x '|value| |$CategoryFrame|))
  (when |$InteractiveMode|
   (setq x
```

```
    (if (and (pairp tmp1) (pairp (qcdr tmp1)) (pairp (qcdr (qcdr tmp1)))
             (eq (qcdr (qcdr (qcdr tmp1))) nil)
             (|member| (second tmp1)
              '((|Mode|) (|Domain|) (|SubDomain| (|Domain|)))))
      (first tmp1)
      (|parseType| x))))
  (setq tmp2
   (dolist (u (fn (second arg)) (nreverse0 tmp3))
     (push (list '|has| x u ) tmp3)))
  (if (and (pairp tmp2) (eq (qcdr tmp2) nil))
   (qcar tmp2)
   (cons '|and| tmp2)))))
```

———-

4.2.36 defun parseHasRhs

[get p??]
[qcar p??]
[qcdr p??]
[member p??]
[abbreviation? p??]
[loadIfNecessary p??]
[unabbrevAndLoad p??]
[$CategoryFrame p??]

— defun parseHasRhs —

```
(defun |parseHasRhs| (u)
 (let (tmp1 y)
  (declare (special  |$CategoryFrame|))
  (setq tmp1 (|get| u '|value| |$CategoryFrame|))
  (cond
    ((and (pairp tmp1) (pairp (qcdr tmp1))
          (pairp (qcdr (qcdr tmp1))) (eq (qcdr (qcdr (qcdr tmp1))) nil)
          (|member| (second tmp1)
             '((|Mode|) (|Domain|) (|SubDomain| (|Domain|)))))
     (second tmp1))
    ((setq y (|abbreviation?| u))
      (if (|loadIfNecessary| y)
        (list (|unabbrevAndLoad| y))
        (list (list 'attribute u))))
    (t (list (list 'attribute u))))))
```

———-

4.2.37 defun parseIf,ifTran

[parseIf,ifTran p121]
[incExitLevel p**??**]
[makeSimplePredicateOrNil p298]
[incExitLevel p**??**]
[parseTran p103]
[$InteractiveMode p**??**]

— **defun parseIf,ifTran** —

```
(defun |parseIf,ifTran| (p a b)
 (let (pp z ap bp tmp1 tmp2 tmp3 tmp4 tmp5 tmp6 val s)
 (declare (special |$InteractiveMode|))
   (cond
    ((and (null |$InteractiveMode|) (eq p '|true|))
     a)
    ((and (null |$InteractiveMode|) (eq p '|false|))
     b)
    ((and (pairp p) (eq (qcar p) '|not|)
          (pairp (qcdr p)) (eq (qcdr (qcdr p)) nil))
     (|parseIf,ifTran| (second p) b a))
    ((and (pairp p) (eq (qcar p) 'if)
             (progn
              (setq tmp1 (qcdr p))
              (and (pairp tmp1)
                   (progn
                    (setq pp (qcar tmp1))
                    (setq tmp2 (qcdr tmp1))
                    (and (pairp tmp2)
                         (progn
                          (setq ap (qcar tmp2))
                          (setq tmp3 (qcdr tmp2))
                          (and (pairp tmp3)
                               (eq (qcdr tmp3) nil)
                               (progn (setq bp (qcar tmp3)) t))))))))
     (|parseIf,ifTran| pp
       (|parseIf,ifTran| ap (copy a) (copy b))
       (|parseIf,ifTran| bp a b)))
    ((and (pairp p) (eq (qcar p) 'seq)
          (pairp (qcdr p)) (progn (setq tmp2 (reverse (qcdr p))) t)
          (and (pairp tmp2)
               (pairp (qcar tmp2))
               (eq (qcar (qcar tmp2)) '|exit|)
               (progn
                  (setq tmp4 (qcdr (qcar tmp2)))
                  (and (pairp tmp4)
                       (equal (qcar tmp4) 1)
                       (progn
```

```
                    (setq tmp5 (qcdr tmp4))
                    (and (pairp tmp5)
                         (eq (qcdr tmp5) nil)
                         (progn (setq pp (qcar tmp5)) t)))))
               (progn (setq z (qcdr tmp2)) t))
          (progn (setq z (nreverse z)) t))
     (cons 'seq
       (append z
        (list
         (list '|exit| 1 (|parseIf,ifTran| pp
                           (|incExitLevel| a)
                            (|incExitLevel| b)))))))
    ((and (pairp a) (eq (qcar a) 'if) (pairp (qcdr a))
          (equal (qcar (qcdr a)) p) (pairp (qcdr (qcdr a)))
          (pairp (qcdr (qcdr (qcdr a))))
          (eq (qcdr (qcdr (qcdr (qcdr a)))) nil))
     (list 'if p (third a) b))
    ((and (pairp b) (eq (qcar b) 'if)
          (pairp (qcdr b)) (equal (qcar (qcdr b)) p)
          (pairp (qcdr (qcdr b)))
          (pairp (qcdr (qcdr (qcdr b))))
          (eq (qcdr (qcdr (qcdr (qcdr b)))) nil))
     (list 'if p a (fourth b)))
    ((progn
      (setq tmp1 (|makeSimplePredicateOrNil| p))
      (and (pairp tmp1) (eq (qcar tmp1) 'seq)
           (progn
            (setq tmp2 (qcdr tmp1))
            (and (and (pairp tmp2)
                      (progn (setq tmp3 (reverse tmp2)) t))
                 (and (pairp tmp3)
                      (progn
                       (setq tmp4 (qcar tmp3))
                       (and (pairp tmp4) (eq (qcar tmp4) '|exit|)
                            (progn
                             (setq tmp5 (qcdr tmp4))
                             (and (pairp tmp5) (equal (qcar tmp5) 1)
                                  (progn
                                   (setq tmp6 (qcdr tmp5))
                                   (and (pairp tmp6) (eq (qcdr tmp6) nil)
                                        (progn (setq val (qcar tmp6)) t)))))))
                                  (progn (setq s (qcdr tmp3)) t))))))
     (setq s (nreverse s))
     (|parseTran|
      (cons 'seq
       (append s
        (list (list '|exit| 1 (|incExitLevel| (list 'if val a b))))))))
    (t
     (list 'if p a b )))))
```

———-

4.2.38 defun parseIf

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'if '|parseTran|) '|parseIf|))
```

———-

4.2.39 defun parseIf

[parseIf,ifTran p121]
[parseTran p103]

— **defun parseIf** —

```
(defun |parseIf| (arg)
  (if (null (and (pairp arg) (pairp (qcdr arg))
                 (pairp (qcdr (qcdr arg))) (eq (qcdr (qcdr (qcdr arg))) nil)))
    arg
    (|parseIf,ifTran|
      (|parseTran| (first arg))
      (|parseTran| (second arg))
      (|parseTran| (third arg)))))
```

———-

4.2.40 defun parseImplies

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|implies| '|parseTran|) '|parseImplies|))
```

———-

4.2.41 defun parseImplies

[parseIf p123]

— **defun parseImplies** —

```
(defun |parseImplies| (arg)
 (|parseIf| (list (first arg) (second arg) '|true|)))
```

———-

4.2.42 defun parseIn

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'in '|parseTran|) '|parseIn|))
```

———-

4.2.43 defun parseIn

[parseTran p103]
[postError p196]

— **defun parseIn** —

```
(defun |parseIn| (arg)
 (let (i n)
  (setq i (|parseTran| (first arg)))
  (setq n (|parseTran| (second arg)))
  (cond
   ((and (pairp n) (eq (qcar n) 'segment)
         (pairp (qcdr n)) (eq (qcdr (qcdr n)) nil))
    (list 'step i (second n) 1))
   ((and (pairp n) (eq (qcar n) '|reverse|)
         (pairp (qcdr n)) (eq (qcdr (qcdr n)) nil)
         (pairp (qcar (qcdr n))) (eq (qcar (qcar (qcdr n))) 'segment)
         (pairp (qcdr (qcar (qcdr n))))
         (eq (qcdr (qcdr (qcar (qcdr n)))) nil))
    (|postError| (list "  You cannot reverse an infinite sequence." )))
   ((and (pairp n) (eq (qcar n) 'segment)
         (pairp (qcdr n)) (pairp (qcdr (qcdr n)))
         (eq (qcdr (qcdr (qcdr n))) nil))
```

```
    (if (third n)
     (list 'step i (second n) 1 (third n))
     (list 'step i (second n) 1)))
   ((and (pairp n) (eq (qcar n) '|reverse|)
         (pairp (qcdr n)) (eq (qcdr (qcdr n)) nil)
         (pairp (qcar (qcdr n))) (eq (qcar (qcar (qcdr n))) 'segment)
         (pairp (qcdr (qcar (qcdr n))))
         (pairp (qcdr (qcdr (qcar (qcdr n)))))
         (eq (qcdr (qcdr (qcdr (qcar (qcdr n))))) nil))
    (if (third (second n))
     (list 'step i (third (second n)) -1 (second (second n)))
     (|postError| (list "  You cannot reverse an infinite sequence.")))
   ((and (pairp n) (eq (qcar n) '|tails|)
         (pairp (qcdr n)) (eq (qcdr (qcdr n)) nil))
    (list 'on i (second n)))
   (t
    (list 'in i n)))))
```

———-

4.2.44 defun parseInBy

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'inby '|parseTran|) '|parseInBy|))
```

———-

4.2.45 defun parseInBy

[postError p196]
[parseTran p103]
[bright p**??**]
[parseIn p124]

— **defun parseInBy** —

```
(defun |parseInBy| (arg)
 (let (i n inc u)
  (setq i (first arg))
  (setq n (second arg))
  (setq inc (third arg))
  (setq u (|parseIn| (list i n)))
```

```
  (cond
   ((null (and (pairp u) (eq (qcar u) 'step)
               (pairp (qcdr u))
               (pairp (qcdr (qcdr u)))
               (pairp (qcdr (qcdr (qcdr u))))))
    (|postError|
     (cons '|   You cannot use|
      (append (|bright| "by")
       (list "except for an explicitly indexed sequence."))))
   (t
    (setq inc (|parseTran| inc))
    (cons 'step
     (cons (second u)
      (cons (third u)
       (cons (|parseTran| inc) (cddddr u))))))))
```

———-

4.2.46 defun parseIs

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|is| '|parseTran|) '|parseIs|))
```

———-

4.2.47 defun parseIs

[parseTran p103]
[transIs p112]

— **defun parseIs** —

```
(defun |parseIs| (arg)
 (list '|is| (|parseTran| (first arg)) (|transIs| (|parseTran| (second arg)))))
```

———-

4.2.48 defun parseIsnt

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|isnt| '|parseTran|) '|parseIsnt|))
```

———-

4.2.49 defun parseIsnt

[parseTran p103]
[transIs p112]

— **defun parseIsnt** —

```
(defun |parseIsnt| (arg)
 (list '|isnt|
   (|parseTran| (first arg))
   (|transIs| (|parseTran| (second arg)))))
```

———-

4.2.50 defun parseJoin

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|Join| '|parseTran|) '|parseJoin|))
```

———-

4.2.51 defun parseJoin

[parseTranList p105]

— **defun parseJoin** —

```
(defun |parseJoin| (thejoin)
 (labels (
  (fn (arg)
   (cond
    ((null arg)
      nil)
    ((and (pairp arg) (pairp (qcar arg)) (eq (qcar (qcar arg)) '|Join|))
     (append (cdar arg) (fn (rest arg))))
```

```
    (t
     (cons (first arg) (fn (rest arg))))))
 )
 (cons '|Join| (fn (|parseTranList| thejoin)))))
```

———-

4.2.52 defun parseLeave

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|leave| '|parseTran|) '|parseLeave|))
```

———-

4.2.53 defun parseLeave

[parseTran p103]

— **defun parseLeave** —

```
(defun |parseLeave| (arg)
 (let (a b)
    (setq a (|parseTran| (car arg)))
    (setq b (|parseTran| (cdr arg)))
    (cond
     (b
      (cond
       ((null (integerp a))
        (moan "first arg " a " for 'leave' must be integer")
        (list '|leave| 1 a))
       (t (cons '|leave| (cons a b)))))
     (t (list '|leave| 1 a)))))
```

———-

4.2.54 defun parseLessEqual

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|<=| '|parseTran|) '|parseLessEqual|))
```

———-

4.2.55 defun parseLessEqual

[parseTran p103]
[$op p??]

— **defun parseLessEqual** —

```
(defun |parseLessEqual| (arg)
 (declare (special |$op|))
 (|parseTran| (list '|not| (cons (msubst '> '<= |$op|) arg))))
```

———-

4.2.56 defun parseLET

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'let '|parseTran|) '|parseLET|))
```

———-

4.2.57 defun parseLET

[parseTran p103]
[parseTranCheckForRecord p297]
[opOf p??]
[transIs p112]

— **defun parseLET** —

```
(defun |parseLET| (arg)
 (let (p)
  (setq p
   (list 'let (|parseTran| (first arg))
     (|parseTranCheckForRecord| (second arg) (|opOf| (first arg)))))
```

```
  (if (eq (|opOf| (first arg)) '|cons|)
    (list 'let (|transIs| (second p)) (third p))
    p)))
```

———-

4.2.58 defun parseLETD

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'letd '|parseTran|) '|parseLETD|))
```

———-

4.2.59 defun parseLETD

[parseTran p103]
[parseType p108]

— **defun parseLETD** —

```
(defun |parseLETD| (arg)
 (list 'letd
   (|parseTran| (first arg))
   (|parseTran| (|parseType| (second arg)))))
```

———-

4.2.60 defun parseMDEF

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'mdef '|parseTran|) '|parseMDEF|))
```

———-

4.2.61 defun parseMDEF

[parseTran p103]
[parseTranList p105]
[parseTranCheckForRecord p297]
[opOf p??]
[$lhs p??]

— **defun parseMDEF** —

```
(defun |parseMDEF| (arg)
 (let (|$lhs|)
 (declare (special |$lhs|))
  (setq |$lhs| (first arg))
  (list 'mdef
    (|parseTran| |$lhs|)
    (|parseTranList| (second arg))
    (|parseTranList| (third arg))
    (|parseTranCheckForRecord| (fourth arg) (|opOf| |$lhs|)))))
```

———-

4.2.62 defun parseNot

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|not| '|parseTran|) '|parseNot|))
```

———-

4.2.63 defun parseNot

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|^| '|parseTran|) '|parseNot|))
```

———-

4.2.64 defun parseNot

[parseTran p103]
[$InteractiveMode p??]

— **defun parseNot** —

```
(defun |parseNot| (arg)
 (declare (special |$InteractiveMode|))
 (if |$InteractiveMode|
  (list '|not| (|parseTran| (car arg)))
  (|parseTran| (cons 'if (cons (car arg) '(|false| |true|))))))
```

———-

4.2.65 defun parseNotEqual

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|^=| '|parseTran|) '|parseNotEqual|))
```

———-

4.2.66 defun parseNotEqual

[parseTran p103]
[msubst p??]
[$op p??]

— **defun parseNotEqual** —

```
(defun |parseNotEqual| (arg)
 (declare (special |$op|))
 (|parseTran| (list '|not| (cons (msubst '= '^= |$op|) arg))))
```

———-

4.2.67 defun parseOr

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|or| '|parseTran|) '|parseOr|))
```

———-

4.2.68 defun parseOr

[parseTran p103]
[parseTranList p105]
[parseIf p123]
[parseOr p133]

— **defun parseOr** —

```
(defun |parseOr| (arg)
 (let (x)
  (setq x (|parseTran| (car arg)))
  (cond
   (|$InteractiveMode| (cons '|or| (|parseTranList| arg)))
   ((null arg) '|false|)
   ((null (cdr arg)) (car arg))
   ((and (pairp x) (eq (qcar x) '|not|)
         (pairp (qcdr x)) (eq (qcdr (qcdr x)) nil))
     (|parseIf| (list (second x) (|parseOr| (cdr arg)) '|true|)))
   (t
     (|parseIf| (list x '|true| (|parseOr| (cdr arg))))))))
```

———-

4.2.69 defun parsePretend

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|pretend| '|parseTran|) '|parsePretend|))
```

———-

4.2.70 defun parsePretend

[parseTran p103]
[parseType p108]

— **defun parsePretend** —

```
(defun |parsePretend| (arg)
 (if |$InteractiveMode|
  (list '|pretend|
        (|parseTran| (first arg))
        (|parseTran| (|parseType| (second arg))))
  (list '|pretend|
         (|parseTran| (first arg))
         (|parseTran| (second arg)))))
```

———-

4.2.71 defun parseReturn

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|return| '|parseTran|) '|parseReturn|))
```

———-

4.2.72 defun parseReturn

[parseTran p103]
[moan p??]

— **defun parseReturn** —

```
(defun |parseReturn| (arg)
 (let (a b)
  (setq a (|parseTran| (car arg)))
  (setq b (|parseTran| (cdr arg)))
  (cond
   (b
    (when (nequal a 1) (moan "multiple-level 'return' not allowed"))
    (cons '|return| (cons 1 b)))
   (t (list '|return| 1 a)))))
```

———-

4.2.73 defun parseSegment

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'segment '|parseTran|) '|parseSegment|))
```

———-

4.2.74 defun parseSegment

[parseTran p103]

— **defun parseSegment** —

```
(defun |parseSegment| (arg)
  (if (and (pairp arg) (pairp (qcdr arg)) (eq (qcdr (qcdr arg)) nil))
   (if (second arg)
    (list 'segment (|parseTran| (first arg)) (|parseTran| (second arg)))
    (list 'segment (|parseTran| (first arg))))
   (cons 'segment arg)))
```

———-

4.2.75 defun parseSeq

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'seq '|parseTran|) '|parseSeq|))
```

———-

4.2.76 defun parseSeq

[postError p196]
[transSeq p??]
[mapInto p??]
[last p??]

— **defun parseSeq** —

```
(defun |parseSeq| (arg)
 (let (tmp1)
  (when (pairp arg) (setq tmp1 (reverse arg)))
  (if (null (and (pairp arg) (pairp tmp1)
                 (pairp (qcar tmp1)) (eq (qcar (qcar tmp1)) '|exit|)))
    (|postError| (list "   Invalid ending to block: " (|last| arg)))
    (|transSeq| (|mapInto| arg '|parseTran|)))))
```

———-

4.2.77 defun parseVCONS

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'vcons '|parseTran|) '|parseVCONS|))
```

———-

4.2.78 defun parseVCONS

[parseTranList p105]

— **defun parseVCONS** —

```
(defun |parseVCONS| (arg)
 (cons 'vector (|parseTranList| arg)))
```

———-

4.2.79 defun parseWhere

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|where| '|parseTran|) '|parseWhere|))
```

———-

4.2.80 defun parseWhere

[mapInto p??]

— **defun parseWhere** —

```
(defun |parseWhere| (arg)
  (cons '|where| (|mapInto| arg '|parseTran|)))
```

————-

Chapter 5

Compile Transformers

5.1 Direct called comp routines

5.2 Indirect called comp routines

In the **compExpression** function there is the code:

```
(if (and (atom (car x)) (setq fn (getl (car x) 'special)))
  (funcall fn x m e)
  (|compForm| x m e))))
```

The functions in this section are called through the symbol-plist of the symbol being parsed. The original list read:

```
          (|add|    |compAdd|)
;          (\@       |compAtSign|)
          (CAPSULE  |compCapsule|)
          (|case|   |compCase|)
          (|Mapping| |compCat|)
          (|Record| |compCat|)
          (|Union|  |compCat|)
          (CATEGORY |compCategory|)
          (\:\:     |compCoerce|)
          (COLLECTV |compCollectV|)
;          (\:       |compColon|)
          (CONS     |compCons|)
          (|ListCategory| |compConstructorCategory|)
          (|RecordCategory| |compConstructorCategory|)
          (|UnionCategory| |compConstructorCategory|)
          (|VectorCategory| |compConstructorCategory|)
          (|construct| |compConstruct|)
```

```
          (DEF      |compDefine|)
          (|elt|    |compElt|)
          (|exit|   |compExit|)
          (|has|    |compHas|)
          (IF       |compIf|)
          (|import| |compImport|)
          (|is|     |compIs|)
          (|Join|   |compJoin|)
          (|+->|    |compLambda|)
          (|leave|  |compLeave|)
          (MDEF     |compMacro|)
          (QUOTE    |compQuote|)
          (|pretend| |compPretend|)
          (REDUCE   |compReduce|)
          (COLLECT  |compRepeatOrCollect|)
          (REPEAT   |compRepeatOrCollect|)
          (|return| |compReturn|)
          (LET      |compSetq|)
          (SETQ     |compSetq|)
;          (SEQ      |compSeq|)
          (|String| |compString|)
          (|SubDomain| |compSubDomain|)
          (|SubsetCategory| |compSubsetCategory|)
          (\|       |compSuchthat|)
;          (VECTOR   |compVector|)
;          (|where|  |compWhere|)
```

5.2.1 defun compAtSign

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|add| 'special) '|compAdd|))
```

————-

5.2.2 defun compAdd

[comp p337]
[qcdr p**??**]
[qcar p**??**]
[compSubDomain1 p185]
[pairp p**??**]
[nreverse0 p**??**]
[NRTgetLocalIndex p**??**]

[compTuple2Record p**??**]
[compOrCroak p335]
[compCapsule p143]
[/editfile p**??**]
[$addForm p**??**]
[$addFormLhs p**??**]
[$EmptyMode p**??**]
[$NRTaddForm p**??**]
[$packagesUsed p**??**]
[$functorForm p**??**]
[$bootStrapMode p**??**]

— defun compAdd —

```
(defun |compAdd| (arg m e)
 (let (|$addForm| |$addFormLhs| code domainForm predicate tmp3 tmp4)
 (declare (special |$addForm| |$addFormLhs| |$EmptyMode| |$NRTaddForm|
                   |$packagesUsed| |$functorForm| |$bootStrapMode| /editfile))
  (setq |$addForm| (second arg))
  (cond
   ((eq |$bootStrapMode| t)
    (cond
     ((and (pairp |$addForm|) (eq (qcar |$addForm|) '|@Tuple|))
       (setq code nil))
     (t
       (setq tmp3 (|comp| |$addForm| m e))
       (setq code (first tmp3))
       (setq m (second tmp3))
       (setq e (third tmp3)) tmp3))
    (list
      (list 'cond
        (list '|$bootStrapMode| code)
         (list 't
          (list '|systemError|
           (list 'list ''|%b| (mkq (car |$functorForm|)) ''|%d| "from"
                 ''|%b| (mkq (|namestring| /editfile)) ''|%d|
                 "needs to be compiled"))))
        m e))
   (t
    (setq |$addFormLhs| |$addForm|)
    (cond
     ((and (pairp |$addForm|) (eq (qcar |$addForm|) '|SubDomain|)
           (pairp (qcdr |$addForm|)) (pairp (qcdr (qcdr |$addForm|)))
           (eq (qcdr (qcdr (qcdr |$addForm|))) nil))
       (setq domainForm (second |$addForm|))
       (setq predicate (third |$addForm|))
       (setq |$packagesUsed| (cons domainForm |$packagesUsed|))
       (setq |$NRTaddForm| domainForm)
       (|NRTgetLocalIndex| domainForm)
```

```
      ; need to generate slot for add form since all $ go-get
      ; slots will need to access it
      (setq tmp3 (|compSubDomain1| domainForm predicate m e))
      (setq |$addForm| (first tmp3))
      (setq e (third tmp3)) tmp3)
     (t
      (setq |$packagesUsed|
       (if (and (pairp |$addForm|) (eq (qcar |$addForm|) '|@Tuple|))
         (append (qcdr |$addForm|) |$packagesUsed|)
         (cons |$addForm| |$packagesUsed|)))
      (setq |$NRTaddForm| |$addForm|)
      (setq tmp3
       (cond
        ((and (pairp |$addForm|) (eq (qcar |$addForm|) '|@Tuple|))
         (setq |$NRTaddForm|
          (cons '|@Tuple|
           (dolist (x (cdr |$addForm|) (nreverse0 tmp4))
            (push (|NRTgetLocalIndex| x) tmp4))))
         (|compOrCroak| (|compTuple2Record| |$addForm|) |$EmptyMode| e))
        (t
         (|compOrCroak| |$addForm| |$EmptyMode| e))))
      (setq |$addForm| (first tmp3))
      (setq e (third tmp3))
      tmp3))
    (|compCapsule| (third arg) m e)))))
```

———-

5.2.3 defun compAtSign

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|@| 'special) '|compAtSign|))
```

———-

5.2.4 defun compAtSign

[addDomain p**??**]
[comp p337]
[coerce p**??**]

— **defun compAtSign** —

```
(defun |compAtSign| (arg1 m e)
  (let ((x (second arg1)) (mprime (third arg1)) tmp)
   (setq e (|addDomain| mprime e))
   (when (setq tmp (|comp| x mprime e)) (|coerce| tmp m))))
```

———-

5.2.5 defun compCapsule

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'capsule 'special) '|compCapsule|))
```

———-

5.2.6 defun compCapsule

[bootStrapError p**??**]
[compCapsuleInner p144]
[addDomain p**??**]
[editfile p**??**]
[$insideExpressionIfTrue p**??**]
[$functorForm p**??**]
[$bootStrapMode p**??**]

— **defun compCapsule** —

```
(defun |compCapsule| (arg m e)
 (let (|$insideExpressionIfTrue| itemList)
 (declare (special |$insideExpressionIfTrue| |$functorForm| /editfile
                   |$bootStrapMode|))
  (setq itemList (cdr arg))
  (cond
   ((eq |$bootStrapMode| t)
     (list (|bootStrapError| |$functorForm| /editfile) m e))
   (t
    (setq |$insideExpressionIfTrue| nil)
    (|compCapsuleInner| itemList m (|addDomain| '$ e))))))
```

———-

5.2.7 defun compCapsuleInner

[addInformation p??]
[compCapsuleItems p??]
[processFunctorOrPackage p??]
[mkpf p??]
[$getDomainCode p??]
[$signature p??]
[$form p??]
[$addForm p??]
[$insideCategoryPackageIfTrue p??]
[$insideCategoryIfTrue p??]
[$functorLocalParameters p??]

— **defun compCapsuleInner** —

```
(defun |compCapsuleInner| (itemList m e)
 (let (localParList data code)
 (declare (special |$getDomainCode| |$signature| |$form| |$addForm|
                   |$insideCategoryPackageIfTrue| |$insideCategoryIfTrue|
                   |$functorLocalParameters|))
  (setq e (|addInformation| m e))
  (setq data (cons 'progn itemList))
  (setq e (|compCapsuleItems| itemList nil e))
  (setq localParList |$functorLocalParameters|)
  (when |$addForm| (setq data (list '|add| |$addForm| data)))
  (setq code
   (if (and |$insideCategoryIfTrue| (null |$insideCategoryPackageIfTrue|))
    data
    (|processFunctorOrPackage| |$form| |$signature| data localParList m e)))
  (cons (mkpf (append |$getDomainCode| (list code)) 'progn) (list m e))))
```

————-

5.2.8 defun compCase

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|case| 'special) '|compCase|))
```

————-

5.2.9 defun compCase

Will the jerk who commented out these two functions please NOT do so again. These functions ARE needed, and case can NOT be done by modemap alone. The reason is that A case B requires to take A evaluated, but B unevaluated. Therefore a special function is required. You may have thought that you had tested this on "failed" etc., but "failed" evaluates to it's own mode. Try it on x case $ next time.

An angry JHD - August 15th., 1984 [addDomain p**??**]
[compCase1 p145]
[coerce p**??**]

— **defun compCase** —

```
(defun |compCase| (arg m e)
 (let (mp td)
  (setq mp (third arg))
  (setq e (|addDomain| mp e))
  (when (setq td (|compCase1| (second arg) mp e)) (|coerce| td m))))
```

———-

5.2.10 defun compCase1

[comp p337]
[getModemapList p**??**]
[nreverse0 p**??**]
[modeEqual p**??**]
[$Boolean p**??**]
[$EmptyMode p**??**]

— **defun compCase1** —

```
(defun |compCase1| (x m e)
 (let (xp mp ep map tmp3 tmp5 tmp6 u fn)
 (declare (special |$Boolean| |$EmptyMode|))
  (when (setq tmp3 (|comp| x |$EmptyMode| e))
   (setq xp (first tmp3))
   (setq mp (second tmp3))
   (setq ep (third tmp3))
   (when
    (setq u
     (dolist (modemap (|getModemapList| '|case| 2 ep) (nreverse0 tmp5))
        (setq map (first modemap))
        (when
          (and (pairp map) (pairp (qcdr map)) (pairp (qcdr (qcdr map)))
```

```
                (pairp (qcdr (qcdr (qcdr map))))
                (eq (qcdr (qcdr (qcdr (qcdr map)))) nil)
                (|modeEqual| (fourth map) m)
                (|modeEqual| (third map) mp))
            (push (second modemap) tmp5))))
    (when
     (setq fn
      (dolist (onepair u tmp6)
        (when (first onepair) (setq tmp6 (or tmp6 (second onepair))))))
      (list (list '|call| fn xp) |$Boolean| ep))))))
```

———-

5.2.11 defun compCat

— postvars —

```
(eval-when (eval load)
 (setf (get '|Record| 'special) '|compCat|))
```

———-

5.2.12 defun compCat

— postvars —

```
(eval-when (eval load)
 (setf (get '|Mapping| 'special) '|compCat|))
```

———-

5.2.13 defun compCat

— postvars —

```
(eval-when (eval load)
 (setf (get '|Union| 'special) '|compCat|))
```

———-

5.2.14 defun compCat

[getl p??]

— **defun compCat** —

```
(defun |compCat| (form m e)
 (declare (ignore m))
 (let (functorName fn tmp1 tmp2 funList op sig catForm)
  (setq functorName (first form))
  (when (setq fn (getl functorName '|makeFunctionList|))
   (setq tmp1 (funcall fn form form e))
   (setq funList (first tmp1))
   (setq e (second tmp1))
   (setq catForm
    (list '|Join| '(|SetCategory|)
     (cons 'category
      (cons '|domain|
       (dolist (item funList (nreverse0 tmp2))
        (setq op (first item))
        (setq sig (second item))
        (unless (eq op '=) (push (list 'signature op sig) tmp2)))))))
   (list form catForm e))))
```

———-

5.2.15 defun compCategory

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'category 'special) '|compCategory|))
```

———-

5.2.16 defun compCategory

[resolve p??]
[qcar p??]
[qcdr p??]
[compCategoryItem p??]
[mkExplicitCategoryFunction p??]
[systemErrorHere p??]

— **defun compCategory** —

```
(defun |compCategory| (x m e)
 (let ($top_level |$sigList| |$atList| domainOrPackage z rep)
 (declare (special $top_level |$sigList| |$atList|))
  (setq $top_level t)
  (cond
   ((and
      (equal (setq m (|resolve| m (list '|Category|))) (list '|Category|))
      (pairp x)
      (eq (qcar x) 'category)
      (pairp (qcdr x)))
    (setq domainOrPackage (second x))
    (setq z (qcdr (qcdr x)))
    (setq |$sigList| nil)
    (setq |$atList| nil)
    (setq |$sigList| nil)
    (setq |$atList| nil)
    (dolist (x z) (|compCategoryItem| x nil))
    (setq rep
      (|mkExplicitCategoryFunction| domainOrPackage |$sigList| |$atList|))
    (list rep m e))
   (t
    (|systemErrorHere| "compCategory")))))
```

———-

5.2.17 defun compCoerce

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|::| 'special) '|compCoerce|))
```

———-

5.2.18 defun compCoerce

[addDomain p**??**]
[getmode p**??**]
[compCoerce1 p149]
[coerce p**??**]

— **defun compCoerce** —

```
(defun |compCoerce| (arg m e)
 (let (x mp tmp1 tmp4 z td)
  (setq x (second arg))
  (setq mp (third arg))
  (setq e (|addDomain| mp e))
  (setq tmp1 (|getmode| mp e))
  (cond
   ((setq td (|compCoerce1| x mp e))
    (|coerce| td m))
   ((and (pairp tmp1) (eq (qcar tmp1) '|Mapping|)
         (pairp (qcdr tmp1)) (eq (qcdr (qcdr tmp1)) nil)
         (pairp (qcar (qcdr tmp1)))
         (eq (qcar (qcar (qcdr tmp1))) '|UnionCategory|))
    (setq z (qcdr (qcar (qcdr tmp1))))
    (when
     (setq td
      (dolist (m1 z tmp4) (setq tmp4 (or tmp4 (|compCoerce1| x m1 e)))))
     (|coerce| (list (car td) mp (third td)) m))))))
```

———-

5.2.19 defun compCoerce1

[comp p337]
[resolve p??]
[coerce p??]
[coerceByModemap p??]
[msubst p??]
[mkq p??]

— **defun compCoerce1** —

```
(defun |compCoerce1| (x mp e)
 (let (m1 td tp gg pred code)
 (declare (special |$String| |$EmptyMode|))
  (when (setq td (or (|comp| x mp e) (|comp| x |$EmptyMode| e)))
    (setq m1 (if (stringp (second td)) |$String| (second td)))
    (setq mp (|resolve| m1 mp))
    (setq td (list (car td) m1 (third td)))
    (cond
     ((setq tp (|coerce| td mp)) tp)
     ((setq tp (|coerceByModemap| td mp)) tp)
     ((setq pred (|isSubset| mp (second td) e))
       (setq gg (gensym))
       (setq pred (msubst gg '* pred))
       (setq code
        (list 'prog1
```

```
        (list 'let gg (first td))
         (cons '|check-subtype| (cons pred (list (mkq mp) gg)))))
       (list code mp (third td)))))))
```

———-

5.2.20 defun compColon

— postvars —

```
(eval-when (eval load)
 (setf (get '|:| 'special) '|compColon|))
```

———-

5.2.21 defun compColon

```
;compColon([":",f,t],m,e) ==
;  $insideExpressionIfTrue=true => compColonInside(f,m,e,t)
;    --if inside an expression, ":" means to convert to m "on faith"
;  $lhsOfColon: local:= f
;  t:=
;    atom t and (t':= ASSOC(t,getDomainsInScope e)) => t'
;    isDomainForm(t,e) and not $insideCategoryIfTrue =>
;      (if not MEMBER(t,getDomainsInScope e) then e:= addDomain(t,e); t)
;    isDomainForm(t,e) or isCategoryForm(t,e) => t
;    t is ["Mapping",m',:r] => t
;    unknownTypeError t
;    t
;  f is ["LISTOF",:l] =>
;    (for x in l repeat T:= [.,.,e]:= compColon([":",x,t],m,e); T)
;  e:=
;    f is [op,:argl] and not (t is ["Mapping",:.]) =>
;      --for MPOLY--replace parameters by formal arguments: RDJ 3/83
;      newTarget:= EQSUBSTLIST(take(#argl,$FormalMapVariableList),
;        [(x is [":",a,m] => a; x) for x in argl],t)
;      signature:=
;        ["Mapping",newTarget,:
;          [(x is [":",a,m] => m;
;              getmode(x,e) or systemErrorHere '"compColonOld") for x in argl]]
;      put(op,"mode",signature,e)
;    put(f,"mode",t,e)
;  if not $bootStrapMode and $insideFunctorIfTrue and
;    makeCategoryForm(t,e) is [catform,e] then
;        e:= put(f,"value",[genSomeVariable(),t,$noEnv],e)
```

```
;  ["/throwAway",getmode(f,e),e]
```

[compColonInside p342]
[assoc p??]
[getDomainsInScope p??]
[isDomainForm p??]
[compColon member (vol5)]
[addDomain p??]
[isDomainForm p??]
[isCategoryForm p??]
[unknownTypeError p??]
[compColon p150]
[eqsubstlist p??]
[take p??]
[length p??]
[nreverse0 p??]
[getmode p??]
[systemErrorHere p??]
[put p??]
[makeCategoryForm p??]
[genSomeVariable p??]
[$lhsOfColon p??]
[$noEnv p??]
[$insideFunctorIfTrue p??]
[$bootStrapMode p??]
[$FormalMapVariableList p??]
[$insideCategoryIfTrue p??]
[$insideExpressionIfTrue p??]

— defun compColon —

```
(defun |compColon| (arg0 m e)
  (let (|$lhsOfColon| argf argt tprime mprime r td op argl newTarget a
        signature tmp2 catform tmp3 g2 g5)
  (declare (special |$lhsOfColon| |$noEnv| |$insideFunctorIfTrue|
                    |$bootStrapMode| |$FormalMapVariableList|
                    |$insideCategoryIfTrue| |$insideExpressionIfTrue|))
    (setq argf (second arg0))
    (setq argt (third arg0))
    (if |$insideExpressionIfTrue|
      (|compColonInside| argf m e argt)
      (progn
        (setq |$lhsOfColon| argf)
        (setq argt
         (cond
          ((and (atom argt)
                (setq tprime (|assoc| argt (|getDomainsInScope| e))))
```

```
      tprime)
     ((and (|isDomainForm| argt e) (null |$insideCategoryIfTrue|))
      (unless (|member| argt (|getDomainsInScope| e))
         (setq e (|addDomain| argt e)))
      argt)
     ((or (|isDomainForm| argt e) (|isCategoryForm| argt e))
      argt)
     ((and (pairp argt) (eq (qcar argt) '|Mapping|)
           (progn
             (setq tmp2 (qcdr argt))
             (and (pairp tmp2)
                  (progn
                   (setq mprime (qcar tmp2))
                   (setq r (qcdr tmp2))
                   t))))
      argt)
     (t
      (|unknownTypeError| argt)
      argt)))
  (cond
   ((eq (car argf) 'listof)
    (dolist (x (cdr argf) td)
      (setq td (|compColon| (list '|:| x argt) m e))
      (setq e (third td))))
   (t
    (setq e
     (cond
      ((and (pairp argf)
            (progn
             (setq op (qcar argf))
             (setq argl (qcdr argf))
             t)
            (null (and (pairp argt) (eq (qcar argt) '|Mapping|))))
       (setq newTarget
        (eqsubstlist (take (|#| argl) |$FormalMapVariableList|)
        (dolist (x argl (nreverse0 g2))
          (setq g2
           (cons
            (cond
             ((and (pairp x) (eq (qcar x) '|:|)
                   (progn
                    (setq tmp2 (qcdr x))
                    (and (pairp tmp2)
                         (progn
                          (setq a (qcar tmp2))
                          (setq tmp3 (qcdr tmp2))
                          (and (pairp tmp3)
                               (eq (qcdr tmp3) nil)
                               (progn
                                (setq m (qcar tmp3))
```

```
                                t))))))
                 a)
                (t x))
         g2)))
        argt))
       (setq signature
        (cons '|Mapping|
         (cons newTarget
          (dolist (x argl (nreverse0 g5))
            (setq g5
             (cons
              (cond
               ((and (pairp x) (eq (qcar x) '|:|)
                (progn
                 (setq tmp2 (qcdr x))
                 (and (pairp tmp2)
                      (progn
                       (setq a (qcar tmp2))
                       (setq tmp3 (qcdr tmp2))
                       (and (pairp tmp3)
                            (eq (qcdr tmp3) nil)
                            (progn
                             (setq m (qcar tmp3))
                             t))))))
                m)
               (t
                (or (|getmode| x e)
                    (|systemErrorHere| "compColonOld"))))
              g5))))))
     (|put| op '|mode| signature e))
    (t (|put| argf '|mode| argt e))))
(cond
  ((and (null |$bootStrapMode|) |$insideFunctorIfTrue|
        (progn
         (setq tmp2 (|makeCategoryForm| argt e))
         (and (pairp tmp2)
              (progn
               (setq catform (qcar tmp2))
               (setq tmp3 (qcdr tmp2))
               (and (pairp tmp3)
                    (eq (qcdr tmp3) nil)
                    (progn
                      (setq e (qcar tmp3))
                              t))))))
   (setq e
    (|put| argf '|value| (list (|genSomeVariable|) argt |$noEnv|)
     e))))
(list '|/throwAway| (|getmode| argf e) e )))))))
```

———-

5.2.22 defun compCons

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'cons 'special) '|compCons|))
```

———-

5.2.23 defun compCons

[compCons1 p154]
[compForm p348]

— **defun compCons** —

```
(defun |compCons| (form m e)
  (or (|compCons1| form m e) (|compForm| form m e)))
```

———-

5.2.24 defun compCons1

[comp p337]
[convert p344]
[pairp p??]
[qcar p??]
[qcdr p??]
[$EmptyMode p??]

— **defun compCons1** —

```
(defun |compCons1| (arg m e)
 (let (mx y my yt mp mr ytp tmp1 x td)
 (declare (special |$EmptyMode|))
  (setq x (second arg))
  (setq y (third arg))
  (when (setq tmp1 (|comp| x |$EmptyMode| e))
   (setq x (first tmp1))
   (setq mx (second tmp1))
```

```
(setq e (third tmp1))
(cond
 ((null y)
  (|convert| (list (list 'list x) (list '|List| mx) e ) m))
 (t
  (when (setq yt (|comp| y |$EmptyMode| e))
   (setq y (first yt))
   (setq my (second yt))
   (setq e (third yt))
   (setq td
    (cond
     ((and (pairp my) (eq (qcar my) '|List|) (pairp (qcdr my)))
       (setq mp (second my))
       (when (setq mr (list '|List| (|resolve| mp mx)))
        (when (setq ytp (|convert| yt mr))
         (when (setq tmp1 (|convert| (list x mx (third ytp)) (second mr)))
          (setq x (first tmp1))
          (setq e (third tmp1))
          (cond
           ((and (pairp (car ytp)) (eq (qcar (car ytp)) 'list))
            (list (cons 'list (cons x (cdr (car ytp)))) mr e))
           (t
            (list (list 'cons x (car ytp)) mr e)))))))
      (t
       (list (list 'cons x y) (list '|Pair| mx my) e ))))
   (|convert| td m)))))
```

———-

5.2.25 defun compConstructorCategory

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|ListCategory| 'special) '|compConstructorCategory|))
```

———-

5.2.26 defun compConstructorCategory

— **postvars** —

```
(eval-when (eval load)
```

```
 (setf (get '|RecordCategory| 'special) '|compConstructorCategory|))
```

———-

5.2.27 defun compConstructorCategory

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|UnionCategory| 'special) '|compConstructorCategory|))
```

———-

5.2.28 defun compConstructorCategory

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|VectorCategory| 'special) '|compConstructorCategory|))
```

———-

5.2.29 defun compConstructorCategory

[resolve p**??**]
[$Category p**??**]

— **defun compConstructorCategory** —

```
(defun |compConstructorCategory| (x m e)
 (declare (special |$Category|))
 (list x (|resolve| |$Category| m) e))
```

———-

5.2.30 defun compConstruct

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|construct| 'special) '|compConstruct|))
```

———-

5.2.31 defun compConstruct

[modeIsAggregateOf p??]
[compList p347]
[convert p344]
[compForm p348]
[compVector p187]
[getDomainsInScope p??]

— **defun compConstruct** —

```
(defun |compConstruct| (form m e)
 (let (z y td tp)
  (setq z (cdr form))
  (cond
   ((setq y (|modeIsAggregateOf| '|List| m e))
    (if (setq td (|compList| z (list '|List| (cadr y)) e))
      (|convert| td m)
      (|compForm| form m e)))
   ((setq y (|modeIsAggregateOf| '|Vector| m e))
    (if (setq td (|compVector| z (list '|Vector| (cadr y)) e))
      (|convert| td m)
      (|compForm| form m e)))
   ((setq td (|compForm| form m e)) td)
   (t
    (dolist (d (|getDomainsInScope| e))
     (cond
      ((and (setq y (|modeIsAggregateOf| '|List| D e))
            (setq td (|compList| z (list '|List| (cadr y)) e))
            (setq tp (|convert| td m)))
       (return tp))
      ((and (setq y (|modeIsAggregateOf| '|Vector| D e))
            (setq td (|compVector| z (list '|Vector| (cadr y)) e))
            (setq tp (|convert| td m)))
       (return tp)))))))))
```

———-

5.2.32 defun compDefine

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'def 'special) '|compDefine|))
```

————-

5.2.33 defun compDefine

[compDefine1 p158]
[$tripleCache p??]
[$tripleHits p??]
[$macroIfTrue p??]
[$packagesUsed p??]

— **defun compDefine** —

```
(defun |compDefine| (form m e)
 (let (|$tripleCache| |$tripleHits| |$macroIfTrue| |$packagesUsed|)
 (declare (special |$tripleCache| |$tripleHits| |$macroIfTrue|
                   |$packagesUsed|))
  (setq |$tripleCache| nil)
  (setq |$tripleHits| 0)
  (setq |$macroIfTrue| nil)
  (setq |$packagesUsed| nil)
  (|compDefine1| form m e)))
```

————-

5.2.34 defun compDefine1

[macroExpand p??]
[isMacro p??]
[getSignatureFromMode p??]
[compDefine1 p158]
[compInternalFunction p??]
[compDefineAddSignature p??]
[compDefWhereClause p??]
[compDefineCategory p??]
[isDomainForm p??]
[getTargetFromRhs p??]

[giveFormalParametersValues p??]
[addEmptyCapsuleIfNecessary p??]
[compDefineFunctor p??]
[stackAndThrow p??]
[strconc p??]
[getAbbreviation p??]
[length p??]
[compDefineCapsuleFunction p??]
[$insideExpressionIfTrue p??]
[$formalArgList p??]
[$form p??]
[$op p??]
[$prefix p??]
[$insideFunctorIfTrue p??]
[$Category p??]
[$insideCategoryIfTrue p??]
[$insideCapsuleFunctionIfTrue p??]
[$ConstructorNames p??]
[$NoValueMode p??]
[$EmptyMode p??]
[$insideWhereIfTrue p??]
[$insideExpressionIfTrue p??]

— **defun compDefine1** —

```
(defun |compDefine1| (form m e)
 (let (|$insideExpressionIfTrue| lhs specialCases sig signature rhs newPrefix
       (tmp1 t))
 (declare (special |$insideExpressionIfTrue| |$formalArgList| |$form|
                   |$op| |$prefix| |$insideFunctorIfTrue| |$Category|
                   |$insideCategoryIfTrue| |$insideCapsuleFunctionIfTrue|
                   |$ConstructorNames| |$NoValueMode| |$EmptyMode|
                   |$insideWhereIfTrue| |$insideExpressionIfTrue|))
  (setq |$insideExpressionIfTrue| nil)
  (setq form (|macroExpand| form e))
  (setq lhs (second form))
  (setq signature (third form))
  (setq specialCases (fourth form))
  (setq rhs (fifth form))
  (cond
   ((and |$insideWhereIfTrue|
         (|isMacro| form e)
         (or (equal m |$EmptyMode|) (equal m |$NoValueMode|)))
     (list lhs m (|put| (car lhs) '|macro| rhs e)))
   ((and (null (car signature)) (consp rhs)
         (null (member (qcar rhs) |$ConstructorNames|))
         (setq sig (|getSignatureFromMode| lhs e)))
    (|compDefine1|
```

```
    (list 'def lhs (cons (car sig) (cdr signature)) specialCases rhs) m e))
  (|$insideCapsuleFunctionIfTrue| (|compInternalFunction| form m e))
  (t
   (when (equal (car signature) |$Category|) (setq |$insideCategoryIfTrue| t))
   (setq e (|compDefineAddSignature| lhs signature e))
   (cond
    ((null (dolist (x (rest signature) tmp1) (setq tmp1 (and tmp1 (null x)))))
     (|compDefWhereClause| form m e))
    ((equal (car signature) |$Category|)
     (|compDefineCategory| form m e nil |$formalArgList|))
    ((and (|isDomainForm| rhs e) (null |$insideFunctorIfTrue|))
     (when (null (car signature))
       (setq signature
        (cons (|getTargetFromRhs| lhs rhs
                (|giveFormalParametersValues| (cdr lhs) e))
              (cdr signature))))
     (setq rhs (|addEmptyCapsuleIfNecessary| (car signature) rhs))
     (|compDefineFunctor|
       (list 'def lhs signature specialCases rhs) m e NIL |$formalArgList|))
    ((null |$form|)
     (|stackAndThrow| (list "bad == form " form)))
    (t
     (setq newPrefix
      (if |$prefix|
        (intern (strconc (|encodeItem| |$prefix|) "," (|encodeItem| |$op|)))
        (|getAbbreviation| |$op| (|#| (cdr |$form|)))))
     (|compDefineCapsuleFunction| form m e newPrefix |$formalArgList|)))))))
```

———-

5.2.35 defun compElt

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|elt| 'special) '|compElt|))
```

———-

5.2.36 defun compElt

[compForm p348]
[isDomainForm p**??**]
[addDomain p**??**]

[getModemapListFromDomain p??]
[length p??]
[stackMessage p??]
[stackWarning p??]
[convert p344]
[opOf p??]
[getDeltaEntry p??]
[nequal p??]
[$One p??]
[$Zero p??]

— defun compElt —

```
(defun |compElt| (form m e)
 (let (aDomain anOp mmList n modemap sig pred val)
 (declare (special |$One| |$Zero|))
 (setq anOp (third form))
 (setq aDomain (second form))
 (cond
  ((null (and (pairp form) (eq (qcar form) '|elt|)
              (pairp (qcdr form)) (pairp (qcdr (qcdr form)))
              (eq (qcdr (qcdr (qcdr form))) nil)))
   (|compForm| form m e))
  ((eq aDomain '|Lisp|)
   (list (cond
          ((equal anOp |$Zero|) 0)
          ((equal anOp |$One|) 1)
          (t anOp))
      m e))
  ((|isDomainForm| aDomain e)
   (setq e (|addDomain| aDomain e))
   (setq mmList (|getModemapListFromDomain| anOp 0 aDomain e))
   (setq modemap
    (progn
     (setq n (|#| mmList))
     (cond
      ((eql 1 n) (elt mmList 0))
      ((eql 0 n)
        (|stackMessage|
          (list "Operation " '|%b| anOp '|%d| "missing from domain: "
                aDomain nil))
        nil)
      (t
       (|stackWarning|
        (list "more than 1 modemap for: " anOp " with dc="
               aDomain " ===>" mmList ))
        (elt mmList 0)))))
   (when modemap
    (setq sig (first modemap))
```

```
    (setq pred (caadr modemap))
    (setq val (cadadr modemap))
    (unless (and (nequal (|#| sig) 2)
                 (null (and (pairp val) (eq (qcar val) '|elt|))))
       (setq val (|genDeltaEntry| (cons (|opOf| anOp) modemap)))
       (|convert| (list (list '|call| val) (second sig) e) m))))
  (t
    (|compForm| form m e)))))
```

———-

5.2.37 defun compExit

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|exit| 'special) '|compExit|))
```

———-

5.2.38 defun compExit

[comp p337]
[modifyModeStack p363]
[stackMessageIfNone p**??**]
[$exitModeStack p**??**]

— **defun compExit** —

```
(defun |compExit| (arg0 m e)
 (let (x index m1 u)
 (declare (special |$exitModeStack|))
  (setq index (1- (second arg0)))
  (setq x (third arg0))
  (cond
   ((null |$exitModeStack|)
    (|comp| x m e))
   (t
    (setq m1 (elt |$exitModeStack| index))
    (setq u (|comp| x m1 e))
    (cond
     (u
      (|modifyModeStack| (second u) index)
      (list (list '|TAGGEDexit| index u) m e))
```

```
    (t
      (|stackMessageIfNone|
       (list '|cannot compile exit expression| x '|in mode| m1))))))))
```

———-

5.2.39 defun compHas

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|has| 'special) '|compHas|))
```

———-

5.2.40 defun compHas

[chaseInferences p**??**]
[compHasFormat p**??**]
[coerce p**??**]
[$e p**??**]

— **defun compHas** —

```
(defun |compHas| (pred m |$e|)
 (declare (special |$e|))
 (let (a b predCode)
  (setq a (second pred))
  (setq b (third pred))
  (setq |$e| (|chaseInferences| pred |$e|))
  (setq predCode (|compHasFormat| pred))
  (|coerce| (list predCode |$Boolean| |$e|) m)))
```

———-

5.2.41 defun compIf

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'if 'special) '|compIf|))
```

———-

5.2.42 defun compIf

[canReturn p??]
[intersectionEnvironment p??]
[compBoolean p??]
[compFromIf p??]
[resolve p??]
[coerce p??]
[quotify p??]
[$Boolean p??]

— **defun compIf** —

```
(defun |compIf| (arg m e)
 (labels (
  (env (bEnv cEnv b c e)
   (cond
    ((|canReturn| b 0 0 t)
     (if (|canReturn| c 0 0 t) (|intersectionEnvironment| bEnv cEnv) bEnv))
    ((|canReturn| c 0 0 t) cEnv)
    (t e))))
  (let (a b c tmp1 xa ma Ea Einv Tb xb mb Eb Tc xc mc Ec xbp x returnEnv)
  (declare (special |$Boolean|))
   (setq a (second arg))
   (setq b (third arg))
   (setq c (fourth arg))
   (when  (setq tmp1 (|compBoolean| a |$Boolean| e))
    (setq xa (first tmp1))
    (setq ma (second tmp1))
    (setq Ea (third tmp1))
    (setq Einv (fourth tmp1))
    (when (setq Tb (|compFromIf| b m Ea))
     (setq xb (first Tb))
     (setq mb (second Tb))
     (setq Eb (third Tb))
     (when (setq Tc (|compFromIf| c (|resolve| mb m) Einv))
      (setq xc (first Tc))
      (setq mc (second Tc))
      (setq Ec (third Tc))
      (when (setq xbp (|coerce| Tb mc))
       (setq x (list 'if xa (|quotify| (first xbp)) (|quotify| xc)))
       (setq returnEnv (env (third xbp) Ec (first xbp) xc e))
       (list x mc returnEnv))))))))
```

———-

5.2.43 defun compImport

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|import| 'special) '|compImport|))
```

———-

5.2.44 defun compImport

[addDomain p**??**]
[$NoValueMode p**??**]

— **defun compImport** —

```
(defun |compImport| (arg m e)
 (declare (ignore m))
 (declare (special |$NoValueMode|))
  (dolist (dom (cdr arg)) (setq e (|addDomain| dom e)))
  (list '|/throwAway| |$NoValueMode| e))
```

———-

5.2.45 defun compIs

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|is| 'special) '|compIs|))
```

———-

5.2.46 defun compIs

[comp p337]
[coerce p**??**]
[$Boolean p**??**]
[$EmptyMode p**??**]

— **defun compIs** —

```
(defun |compIs| (arg m e)
 (let (a b aval am tmp1 bval bm td)
 (declare (special |$Boolean| |$EmptyMode|))
  (setq a (CADR arg))
  (setq b (CADDR arg))
  (when (setq tmp1 (|comp| a |$EmptyMode| e))
   (setq aval (CAR tmp1))
   (setq am (CADR tmp1))
   (setq e (CADDR tmp1))
   (when (setq tmp1 (|comp| b |$EmptyMode| e))
    (setq bval (CAR tmp1))
    (setq bm (CADR tmp1))
    (setq e (CADDR tmp1))
    (setq td (list (list '|domainEqual| aval bval) |$Boolean| e ))
    (|coerce| td m)))))
```

———-

5.2.47 defun compJoin

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|Join| 'special) '|compJoin|))
```

———-

5.2.48 defun compJoin

[nreverse0 p**??**]
[compForMode p**??**]
[stackSemanticError p**??**]
[nreverse0 p**??**]
[isCategoryForm p**??**]
[union p**??**]
[compJoin,getParms p**??**]
[pairp p**??**]
[qcar p**??**]
[qcdr p**??**]
[wrapDomainSub p**??**]
[convert p344]
[$Category p**??**]

— **defun compJoin** —

```
(defun |compJoin| (arg m e)
 (labels (
  (getParms (y e)
   (cond
    ((atom y)
     (when (|isDomainForm| y e) (list y)))
    ((and (pairp y) (eq (qcar y) 'length)
          (pairp (qcdr y)) (eq (qcdr (qcdr y)) nil))
     (list y (second y)))
    (t (list y)))) )
 (let (argl catList pl tmp3 tmp4 tmp5 body parameters catListp td)
 (declare (special |$Category|))
  (setq argl (cdr arg))
  (setq catList
   (dolist (x argl (nreverse0 tmp3))
    (push (car (or (|compForMode| x |$Category| e) (return '|failed|)))
       tmp3)))
  (cond
   ((eq catList '|failed|)
    (|stackSemanticError| (list '|cannot form Join of: | argl) nil))
   (t
    (setq catListp
     (dolist (x catList (nreverse0 tmp4))
       (setq tmp4
        (cons
         (cond
          ((|isCategoryForm| x e)
           (setq parameters
            (|union|
              (dolist (y (cdr x) tmp5)
               (setq tmp5 (append tmp5 (getParms y e))))
              parameters))
           x)
          ((and (pairp x) (eq (qcar x) '|DomainSubstitutionMacro|)
                (pairp (qcdr x)) (pairp (qcdr (qcdr x)))
                (eq (qcdr (qcdr (qcdr x))) nil))
           (setq pl (second x))
           (setq body (third x))
           (setq parameters (|union| pl parameters)) body)
          ((and (pairp x) (eq (qcar x) '|mkCategory|))
            x)
          ((and (atom x) (equal (|getmode| x e) |$Category|))
            x)
          (t
           (|stackSemanticError| (list '|invalid argument to Join: | x) nil)
           x))
         tmp4))))
    (setq td (list (|wrapDomainSub| parameters (cons '|Join| catListp))
                |$Category| e))
    (|convert| td m))))))
```

———-

5.2.49 defun compLambda

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|+->| 'special) '|compLambda|))
```

———-

5.2.50 defun compLambda

[qcar p**??**]
[qcdr p**??**]
[argsToSig p361]
[compAtSign p142]
[stackAndThrow p**??**]

— **defun compLambda** —

```
(defun |compLambda| (x m e)
 (let (vl body tmp1 tmp2 tmp3 target args arg1 sig1 ress)
  (setq vl (second x))
  (setq body (third x))
  (cond
   ((and (pairp vl) (eq (qcar vl) '|:|)
         (progn
          (setq tmp1 (qcdr vl))
          (and (pairp tmp1)
               (progn
                (setq args (qcar tmp1))
                (setq tmp2 (qcdr tmp1))
                (and (pairp tmp2)
                     (eq (qcdr tmp2) nil)
                     (progn
                      (setq target (qcar tmp2))
                      t))))))
    (when (and (pairp args) (eq (qcar args) '|@Tuple|))
     (setq args (qcdr args)))
    (cond
     ((listp args)
      (setq tmp3 (|argsToSig| args))
```

```
       (setq arg1 (CAR tmp3))
       (setq sig1 (second tmp3))
       (cond
        (sig1
         (setq ress
          (|compAtSign|
           (list '@
            (list '+-> arg1 body)
            (cons '|Mapping| (cons target sig1))) m e))
         ress)
        (t (|stackAndThrow| (list '|compLambda| x )))))
      (t (|stackAndThrow| (list '|compLambda| x )))))
    (t (|stackAndThrow| (list '|compLambda| x ))))))
```

———-

5.2.51 defun compLeave

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|leave| 'special) '|compLeave|))
```

———-

5.2.52 defun compLeave

[comp p337]
[modifyModeStack p363]
[$exitModeStack p**??**]
[$leaveLevelStack p**??**]

— **defun compLeave** —

```
(defun |compLeave| (arg m e)
 (let (level x index u)
  (declare (special |$exitModeStack| |$leaveLevelStack|))
   (setq level (second arg))
   (setq x (third arg))
   (setq index
     (- (1- (|#| |$exitModeStack|)) (elt |$leaveLevelStack| (1- level))))
   (when (setq u (|comp| x (elt |$exitModeStack| index) e))
    (|modifyModeStack| (second u) index)
    (list (list '|TAGGEDexit| index u) m e ))))
```

———-

5.2.53 defun compMacro

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'mdef 'special) '|compMacro|))
```

———-

5.2.54 defun compMacro

[qcar p**??**]
[formatUnabbreviated p**??**]
[sayBrightly p**??**]
[put p**??**]
[macroExpand p**??**]
[$macroIfTrue p**??**]
[$NoValueMode p**??**]
[$EmptyMode p**??**]

— **defun compMacro** —

```
(defun |compMacro| (form m e)
 (let (|$macroIfTrue| lhs signature specialCases rhs prhs)
 (declare (special |$macroIfTrue| |$NoValueMode| |$EmptyMode|))
  (setq |$macroIfTrue| t)
  (setq lhs (second form))
  (setq signature (third form))
  (setq specialCases (fourth form))
  (setq rhs (fifth form))
  (setq prhs
   (cond
    ((and (pairp rhs) (eq (qcar rhs) 'category))
      (list "-- the constructor category"))
    ((and (pairp rhs) (eq (qcar rhs) '|Join|))
      (list "-- the constructor category"))
    ((and (pairp rhs) (eq (qcar rhs) 'capsule))
      (list "-- the constructor capsule"))
    ((and (pairp rhs) (eq (qcar rhs) '|add|))
      (list "-- the constructor capsule"))
    (t (|formatUnabbreviated| rhs))))
  (|sayBrightly|
   (cons "   processing macro definition"
```

```
    (cons '|%b|
     (append (|formatUnabbreviated| lhs)
      (cons " ==> "
       (append prhs (list '|%d|)))))))
  (when (or (equal m |$EmptyMode|) (equal m |$NoValueMode|))
    (list '|/throwAway| |$NoValueMode|
     (|put| (CAR lhs) '|macro| (|macroExpand| rhs e) e)))))
```

———-

5.2.55 defun compPretend

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|pretend| 'special) '|compPretend|))
```

———-

5.2.56 defun compPretend

[addDomain p**??**]
[comp p337]
[opOf p**??**]
[nequal p**??**]
[stackSemanticError p**??**]
[stackWarning p**??**]
[$newCompilerUnionFlag p**??**]
[$EmptyMode p**??**]

— **defun compPretend** —

```
(defun |compPretend| (arg m e)
 (let (x tt warningMessage td tp)
 (declare (special |$newCompilerUnionFlag| |$EmptyMode|))
  (setq x (second arg))
  (setq tt (third arg))
  (setq e (|addDomain| tt e))
  (when (setq td (or (|comp| x tt e) (|comp| x |$EmptyMode| e)))
   (when (equal (second td) tt)
    (setq warningMessage (list '|pretend| tt '| -- should replace by @|)))
   (cond
     ((and |$newCompilerUnionFlag|
           (eq (|opOf| (second td)) '|Union|)
```

```
          (nequal (|opOf| m) '|Union|))
       (|stackSemanticError|
        (list '|cannot pretend | x '| of mode | (second td) '| to mode | m)
         nil))
     (t
      (setq td (list (first td) tt (third td)))
      (when (setq tp (|coerce| td m))
        (when warningMessage (|stackWarning| warningMessage))
        tp))))))
```

———-

5.2.57 defun compQuote

— postvars —

```
(eval-when (eval load)
 (setf (get 'quote 'special) '|compQuote|))
```

———-

5.2.58 defun compQuote

— defun compQuote —

```
(defun |compQuote| (expr m e)
 (list expr m e))
```

———-

5.2.59 defun compRepeatOrCollect

— postvars —

```
(eval-when (eval load)
 (setf (get 'collect 'special) '|compRepeatOrCollect|))
```

———-

5.2.60 defun compRepeatOrCollect

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'repeat 'special) '|compRepeatOrCollect|))
```

———-

5.2.61 defun compRepeatOrCollect

[length p??]
[compIterator p??]
[modeIsAggregateOf p??]
[stackMessage p??]
[compOrCroak p335]
[comp p337]
[msubst p??]
[coerceExit p??]
[p??]
[p??]
[$until p??]
[$Boolean p??]
[$NoValueMode p??]
[$exitModeStack p??]
[$leaveLevelStack p??]
[$formalArgList p??]

— **defun compRepeatOrCollect** —

```
(defun |compRepeatOrCollect| (form m e)
 (labels (
  (fn (form |$exitModeStack| |$leaveLevelStack| |$formalArgList| e)
  (declare (special |$exitModeStack| |$leaveLevelStack| |$formalArgList|))
  (let (|$until| body itl xp targetMode repeatOrCollect bodyMode bodyp mp tmp1
        untilCode ep itlp formp u mpp tmp2)
  (declare (special  |$Boolean| |$until| |$NoValueMode| ))
   (setq |$until| nil)
   (setq repeatOrCollect (car form))
   (setq tmp1 (reverse (cdr form)))
   (setq body (car tmp1))
   (setq itl (nreverse (cdr tmp1)))
   (setq itlp
    (dolist (x itl (nreverse0 tmp2))
     (setq tmp1 (or (|compIterator| x e) (return '|failed|)))
```

```
    (setq xp (first tmp1))
    (setq e (second tmp1))
    (push xp tmp2)))
  (unless (eq itlp '|failed|)
   (setq targetMode (car |$exitModeStack|))
   (setq bodyMode
     (if (eq repeatOrCollect 'collect)
      (cond
       ((eq targetMode '|$EmptyMode|)
         '|$EmptyMode|)
       ((setq u (|modeIsAggregateOf| '|List| targetMode e))
         (second u))
       ((setq u (|modeIsAggregateOf| '|PrimitiveArray| targetMode e))
        (setq repeatOrCollect 'collectv)
        (second u))
       ((setq u (|modeIsAggregateOf| '|Vector| targetMode e))
        (setq repeatOrCollect 'collectvec)
        (second u))
       (t
        (|stackMessage| "Invalid collect bodytype")
        '|failed|))
      |$NoValueMode|))
    (unless (eq bodyMode '|failed|)
     (when (setq tmp1 (|compOrCroak| body bodyMode e))
      (setq bodyp (first tmp1))
      (setq mp (second tmp1))
      (setq ep (third tmp1))
      (when |$until|
        (setq tmp1 (|comp| |$until| |$Boolean| ep))
        (setq untilCode (first tmp1))
        (setq ep (third tmp1))
        (setq itlp (msubst (list 'until untilCode) '|$until| itlp)))
      (setq formp (cons repeatOrCollect (append itlp (list bodyp))))
      (setq mpp
       (cond
        ((eq repeatOrCollect 'collect)
          (if (setq u (|modeIsAggregateOf| '|List| targetMode e))
            (car u)
            (list '|List| mp)))
        ((eq repeatOrCollect 'collectv)
          (if (setq u (|modeIsAggregateOf| '|PrimitiveArray| targetMode e))
            (car u)
            (list '|PrimitiveArray| mp)))
        ((eq repeatOrCollect 'collectvec)
          (if (setq u (|modeIsAggregateOf| '|Vector| targetMode e))
            (car u)
            (list '|Vector| mp)))
        (t mp)))
      (|coerceExit| (list formp  mpp ep) targetMode))))))  )
(declare (special |$exitModeStack| |$leaveLevelStack| |$formalArgList|))
```

```
(fn form
  (cons m |$exitModeStack|)
  (cons (|#| |$exitModeStack|) |$leaveLevelStack|)
  |$formalArgList|
  e)))
```

———-

5.2.62 defun compReduce

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'reduce 'special) '|compReduce|))
```

———-

5.2.63 defun compReduce

[compReduce1 p175]
[$formalArgList p??]

— **defun compReduce** —

```
(defun |compReduce| (form m e)
  (declare (special |$formalArgList|))
  (|compReduce1| form m e |$formalArgList|))
```

———-

5.2.64 defun compReduce1

[systemError p??]
[nreverse0 p??]
[compIterator p??]
[comp p337]
[parseTran p103]
[getIdentity p??]
[msubst p??]
[$sideEffectsList p??]

[$until p??]
[$initList p??]
[$Boolean p??]
[$e p??]
[$endTestList p??]

— defun compReduce1 —

```
(defun |compReduce1| (form m e |$formalArgList|)
 (declare (special |$formalArgList|))
 (let (|$sideEffectsList| |$until| |$initList| |$endTestList| collectForm
       collectOp body op itl acc afterFirst bodyVal part1 part2 part3 id
       identityCode untilCode finalCode tmp1 tmp2)
 (declare (special |$sideEffectsList| |$until| |$initList| |$Boolean| |$e|
                   |$endTestList|))
  (setq op (second form))
  (setq collectForm (fourth form))
  (setq collectOp (first collectForm))
  (setq tmp1 (reverse (cdr collectForm)))
  (setq body (first tmp1))
  (setq itl (nreverse (cdr tmp1)))
  (when (stringp op) (setq op (intern op)))
  (cond
   ((null (member collectOp '(collect collectv collectvec)))
    (|systemError| (list '|illegal reduction form:| form)))
   (t
    (setq |$sideEffectsList| nil)
    (setq |$until| nil)
    (setq |$initList| nil)
    (setq |$endTestList| nil)
    (setq |$e| e)
    (setq itl
     (dolist (x itl (nreverse0 tmp2))
      (setq tmp1 (or (|compIterator| x |$e|) (return '|failed|)))
      (setq |$e| (second tmp1))
      (push (elt tmp1 0) tmp2)))
    (unless (eq itl '|failed|)
     (setq e |$e|)
     (setq acc (gensym))
     (setq afterFirst (gensym))
     (setq bodyVal (gensym))
     (when (setq tmp1 (|comp| (list 'let bodyVal body ) m e))
      (setq part1 (first tmp1))
      (setq m (second tmp1))
      (setq e (third tmp1))
      (when (setq tmp1 (|comp| (list 'let acc bodyVal) m e))
       (setq part2 (first tmp1))
       (setq e (third tmp1))
       (when (setq tmp1
```

```
      (|comp| (list 'let acc (|parseTran| (list op acc bodyVal))) m e))
    (setq part3 (first tmp1))
    (setq e (third tmp1))
    (when (setq identityCode
           (if (setq id (|getIdentity| op e))
            (car (|comp| id m e))
            (list '|IdentityError| (mkq op))))
     (setq finalCode
      (cons 'progn
       (cons (list 'let afterFirst nil)
        (cons
         (cons 'repeat
          (append itl
           (list
            (list 'progn part1
             (list 'if afterFirst part3
              (list 'progn part2 (list 'let afterFirst (mkq t)))) nil))))
             (list (list 'if afterFirst acc identityCode ))))))
     (when |$until|
      (setq tmp1 (|comp| |$until| |$Boolean| e))
      (setq untilCode (first tmp1))
      (setq e (third tmp1))
      (setq finalCode
       (msubst (list 'until untilCode) '|$until| finalCode)))
      (list finalCode m e ))))))))))
```

———-

5.2.65 defun compReturn

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|return| 'special) '|compReturn|))
```

———-

5.2.66 defun compReturn

[stackSemanticError p??]
[nequal p??]
[userError p??]
[resolve p??]
[comp p337]

[modifyModeStack p363]
[$exitModeStack p??]
[$returnMode p??]

— **defun compReturn** —

```
(defun |compReturn| (arg m e)
 (let (level x index u xp mp ep)
 (declare (special |$returnMode| |$exitModeStack|))
  (setq level (second arg))
  (setq x (third arg))
  (cond
   ((null |$exitModeStack|)
    (|stackSemanticError|
      (list '|the return before| '|%b| x '|%d| '|is unneccessary|) nil)
    nil)
   ((nequal level 1)
     (|userError| "multi-level returns not supported"))
   (t
    (setq index (max 0 (1- (|#| |$exitModeStack|))))
    (when (>= index 0)
      (setq |$returnMode|
       (|resolve| (elt |$exitModeStack| index) |$returnMode|)))
    (when (setq u (|comp| x |$returnMode| e))
     (setq xp (first u))
     (setq mp (second u))
     (setq ep (third u))
     (when (>= index 0)
       (setq |$returnMode| (|resolve| mp |$returnMode|))
       (|modifyModeStack| mp index))
     (list (list '|TAGGEDreturn| 0 u) m ep))))))
```

———-

5.2.67 defun compSeq

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'seq 'special) '|compSeq|))
```

———-

5.2.68 defun compSeq

[compSeq1 p179]
[$exitModeStack p??]

— **defun compSeq** —

```
(defun |compSeq| (arg0 m e)
 (declare (special |$exitModeStack|))
 (|compSeq1| (cdr arg0) (cons m |$exitModeStack|) e))
```

———-

5.2.69 defun compSeq1

[nreverse0 p??]
[compSeqItem p180]
[mkq p??]
[replaceExitEtc p??]
[$exitModeStack p??]
[$insideExpressionIfTrue p??]
[$finalEnv p??]
[$NoValueMode p??]

— **defun compSeq1** —

```
(defun |compSeq1| (l |$exitModeStack| e)
 (declare (special |$exitModeStack|))
 (let (|$insideExpressionIfTrue| |$finalEnv| tmp1 tmp2 c catchTag form)
 (declare (special |$insideExpressionIfTrue| |$finalEnv| |$NoValueMode|))
  (setq |$insideExpressionIfTrue| nil)
  (setq |$finalEnv| nil)
  (when
    (setq c (dolist (x l (nreverse0 tmp2))
             (setq |$insideExpressionIfTrue| nil)
             (setq tmp1 (|compSeqItem| x |$NoValueMode| e))
             (unless tmp1 (return nil))
             (setq e (third tmp1))
             (push (first tmp1) tmp2)))
   (setq catchTag (mkq (gensym)))
   (setq form
    (cons 'seq
     (|replaceExitEtc| c catchTag '|TAGGEDexit| (elt |$exitModeStack| 0))))
   (list (list 'catch catchTag form) (elt |$exitModeStack| 0) |$finalEnv|))))
```

———-

5.2.70 defun compSeqItem

[comp p337]
[macroExpand p??]

— **defun compSeqItem** —

```
(defun |compSeqItem| (x m e)
  (|comp| (|macroExpand| x e) m e))
```

————-

5.2.71 defun compSetq

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'let 'special) '|compSetq|))
```

————-

5.2.72 defun compSetq

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'setq 'special) '|compSetq|))
```

————-

5.2.73 defun compSetq

[compSetq1 p181]

— **defun compSetq** —

```
(defun |compSetq| (arg m e)
 (|compSetq1| (second arg) (third arg) m e))
```

————-

5.2.74 defun compSetq1

[setqSingle p182]
[compSetq1 identp (vol5)]
[compMakeDeclaration p362]
[compSetq p180]
[qcar p**??**]
[qcdr p**??**]
[setqMultiple p**??**]
[setqSetelt p181]
[$EmptyMode p**??**]

— **defun compSetq1** —

```
(defun |compSetq1| (form val m e)
 (let (x y ep op z)
 (declare (special |$EmptyMode|))
 (cond
  ((identp form) (|setqSingle| form val m e))
  ((and (pairp form) (eq (qcar form) '|:|) (pairp (qcdr form))
        (pairp (qcdr (qcdr form))) (eq (qcdr (qcdr (qcdr form))) nil))
    (setq x (second form))
    (setq y (third form))
    (setq ep (third (|compMakeDeclaration| form |$EmptyMode| e)))
    (|compSetq| (list 'let x val) m ep))
  ((pairp form)
    (setq op (qcar form))
    (setq z (qcdr form))
    (cond
     ((eq op 'cons)     (|setqMultiple| (|uncons| form) val m e))
     ((eq op '|@Tuple|) (|setqMultiple| z val m e))
     (t                 (|setqSetelt| form val m e)))))))
```

———-

5.2.75 defun setqSetelt

[comp p337]

— **defun setqSetelt** —

```
(defun |setqSetelt| (arg val m e)
 (|comp| (cons '|setelt| (cons (car arg) (append (cdr arg) (list val)))) m e))
```

———-

5.2.76 defun setqSingle

[setqSingle getProplist (vol5)]
[getmode p??]
[get p??]
[nequal p??]
[maxSuperType p??]
[comp p337]
[getmode p??]
[assignError p??]
[convert p344]
[setqSingle identp (vol5)]
[profileRecord p??]
[consProplistOf p??]
[removeEnv p??]
[setqSingle addBinding (vol5)]
[isDomainForm p??]
[isDomainInScope p??]
[stackWarning p??]
[augModemapsFromDomain1 p??]
[NRTassocIndex p??]
[isDomainForm p??]
[outputComp p??]
[$insideSetqSingleIfTrue p??]
[$QuickLet p??]
[$form p??]
[$profileCompiler p??]
[$EmptyMode p??]
[$NoValueMode p??]

— defun setqSingle —

```
(defun |setqSingle| (id val m e)
 (let (|$insideSetqSingleIfTrue| currentProplist mpp maxmpp td x mp tp key
       newProplist ep k form)
 (declare (special |$insideSetqSingleIfTrue| |$QuickLet| |$form|
                   |$profileCompiler| |$EmptyMode| |$NoValueMode|))
  (setq |$insideSetqSingleIfTrue| t)
  (setq currentProplist (|getProplist| id e))
  (setq mpp
   (or (|get| id '|mode| e) (|getmode| id e)
       (if (equal m |$NoValueMode|) |$EmptyMode| m)))
  (when (setq td
         (cond
          ((setq td (|comp| val mpp e))
           td)
          ((and (null (|get| id '|mode| e))
```

```
              (nequal mpp (setq maxmpp (|maxSuperType| mpp e)))
              (setq td (|comp| val maxmpp e)))
         td)
        ((and (setq td (|comp| val |$EmptyMode| e))
              (|getmode| (second td) e))
         (|assignError| val (second td) id mpp))))
  (when (setq tp (|convert| td m))
   (setq x (first tp))
   (setq mp (second tp))
   (setq ep (third tp))
   (when (and |$profileCompiler| (identp id))
     (setq key (if (member id (cdr |$form|)) '|arguments| '|locals|))
     (|profileRecord| key id (second td)))
   (setq newProplist
    (|consProplistOf| id currentProplist '|value|
     (|removeEnv| (cons val (cdr td)))))
   (setq ep (if (pairp id) ep (|addBinding| id newProplist ep)))
   (when (|isDomainForm| val ep)
    (when (|isDomainInScope| id ep)
     (|stackWarning|
      (list '|domain valued variable| '|%b| id '|%d|
            '|has been reassigned within its scope| )))
     (setq ep (|augModemapsFromDomain1| id val ep)))
   (if (setq k (|NRTassocIndex| id))
     (setq form (list 'setelt '$ k x))
     (setq form
      (if |$QuickLet|
       (list 'let id x)
       (list 'let id x
         (if (|isDomainForm| x ep)
           (list 'elt id 0)
           (car (|outputComp| id ep)))))))
   (list form mp ep)))))
```

———-

5.2.77 defun compString

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|String| 'special) '|compString|))
```

———-

5.2.78 defun compString

[resolve p??]
[$StringCategory p??]

— **defun compString** —

```
(defun |compString| (x m e)
 (declare (special |$StringCategory|))
 (list x (|resolve| |$StringCategory| m) e))
```

———-

5.2.79 defun compSubDomain

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|SubDomain| 'special) '|compSubDomain|))
```

———-

5.2.80 defun compSubDomain

[compSubDomain1 p185]
[compCapsule p143]
[$addFormLhs p??]
[$NRTaddForm p??]
[$addForm p??]
[$addFormLhs p??]

— **defun compSubDomain** —

```
(defun |compSubDomain| (arg m e)
 (let (|$addFormLhs| |$addForm| domainForm predicate tmp1)
 (declare (special |$addFormLhs| |$addForm| |$NRTaddForm| |$addFormLhs|))
  (setq domainForm (second arg))
  (setq predicate (third arg))
  (setq |$addFormLhs| domainForm)
  (setq |$addForm| nil)
  (setq |$NRTaddForm| domainForm)
  (setq tmp1 (|compSubDomain1| domainForm predicate m e))
  (setq |$addForm| (first tmp1))
```

```
  (setq e (third tmp1))
  (|compCapsule| (list 'capsule) m e)))
```

———-

5.2.81 defun compSubDomain1

[compMakeDeclaration p362]
[addDomain p??]
[compOrCroak p335]
[stackSemanticError p??]
[lispize p??]
[evalAndRwriteLispForm p??]
[$CategoryFrame p??]
[$op p??]
[$lisplibSuperDomain p??]
[$Boolean p??]
[$EmptyMode p??]

— **defun compSubDomain1** —

```
(defun |compSubDomain1| (domainForm predicate m e)
 (let (u prefixPredicate opp dFp)
 (declare (special |$CategoryFrame| |$op| |$lisplibSuperDomain| |$Boolean|
                   |$EmptyMode|))
  (setq e (third
   (|compMakeDeclaration| (list '|:| '|#1| domainForm)
     |$EmptyMode| (|addDomain| domainForm e))))
   (setq u (|compOrCroak| predicate |$Boolean| e))
   (unless u
    (|stackSemanticError|
     (list '|predicate: | predicate
           '| cannot be interpreted with #1: | domainForm) nil))
   (setq prefixPredicate (|lispize| (first u)))
   (setq |$lisplibSuperDomain| (list domainForm predicate))
   (|evalAndRwriteLispForm| '|evalOnLoad2|
    (list 'setq '|$CategoryFrame|
     (list '|put|
      (setq opp (list 'quote |$op|))
       ''|SuperDomain|
        (setq dFp (list 'quote domainForm))
         (list '|put| dFp ''|SubDomain|
          (list 'cons (list 'quote (cons |$op| prefixPredicate))
          (list 'delasc opp (list '|get| dFp ''|SubDomain| '|$CategoryFrame|)))
          '|$CategoryFrame|))))
   (list domainForm m e)))
```

———-

5.2.82 defun compSubsetCategory

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|SubsetCategory| 'special) '|compSubsetCategory|))
```

———-

5.2.83 defun compSubsetCategory

[put p**??**]
[comp p337]
[msubst p**??**]
[$lhsOfColon p**??**]

— **defun compSubsetCategory** —

```
(defun |compSubsetCategory| (arg m e)
 (let (cat r)
 (declare (special |$lhsOfColon|))
  (setq cat (second arg))
  (setq r (third arg))
  ;  --1. put "Subsets" property on R to allow directly coercion to subset;
  ;  --   allow automatic coercion from subset to R but not vice versa
  (setq e (|put| r '|Subsets| (list (list |$lhsOfColon| '|isFalse|)) e))
  ;  --2. give the subset domain modemaps of cat plus 3 new functions
  (|comp|
    (list '|Join| cat
      (msubst |$lhsOfColon| '$
             (list 'category '|domain|
              (list 'signature '|coerce| (list r '$))
               (list 'signature '|lift| (list r '$))
                (list 'signature '|reduce| (list '$ r)))))
    m e)))
```

———-

5.2.84 defun compSuchthat

— **postvars** —

```
(eval-when (eval load)
 (setf (get '\| 'special) '|compSuchthat|))
```

———-

5.2.85 defun compSuchthat

[comp p337]
[put p??]
[$Boolean p??]

— **defun compSuchthat** —

```
(defun |compSuchthat| (arg m e)
 (let (x p xp mp tmp1 pp)
 (declare (special |$Boolean|))
  (setq x (second arg))
  (setq p (third arg))
  (when (setq tmp1 (|comp| x m e))
   (setq xp (first tmp1))
   (setq mp (second tmp1))
   (setq e (third tmp1))
   (when (setq tmp1 (|comp| p |$Boolean| e))
    (setq pp (first tmp1))
    (setq e (third tmp1))
    (setq e (|put| xp '|condition| pp e))
    (list xp mp e)))))
```

———-

5.2.86 defun compVector

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'vector 'special) '|compVector|))
```

———-

5.2.87 defun compVector

```
;  null l => [$EmptyVector,m,e]
```

```
;  Tl:= [[.,mUnder,e]:= comp(x,mUnder,e) or return "failed" for x in l]
;  Tl="failed" => nil
;  [["VECTOR",:[T.expr for T in Tl]],m,e]
```

[comp p337]
[$EmptyVector p??]

— **defun compVector** —

```
(defun |compVector| (l m e)
 (let (tmp1 tmp2 t0 failed (mUnder (second m)))
  (declare (special |$EmptyVector|))
   (if (null l)
    (list |$EmptyVector| m e)
    (progn
     (setq t0
      (do ((t3 l (cdr t3)) (x nil))
          ((or (atom t3) failed) (unless failed (nreverse0 tmp2)))
        (setq x (car t3))
        (if (setq tmp1 (|comp| x mUnder e))
          (progn
           (setq mUnder (second tmp1))
           (setq e (third tmp1))
           (push tmp1 tmp2))
          (setq failed t))))
     (unless failed
      (list (cons 'vector (loop for texpr in t0 collect (car texpr))) m e))))))
```

———-

5.2.88 defun compWhere

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|where| 'special) '|compWhere|))
```

———-

5.2.89 defun compWhere

[comp p337]
[macroExpand p??]
[deltaContour p??]

[addContour p**??**]
[$insideExpressionIfTrue p**??**]
[$insideWhereIfTrue p**??**]
[$EmptyMode p**??**]

— defun compWhere —

```
(defun |compWhere| (arg0 m eInit)
 (let (|$insideExpressionIfTrue| |$insideWhereIfTrue| form exprList e
       eBefore tmp1 x eAfter del eFinal)
 (declare (special |$insideExpressionIfTrue| |$insideWhereIfTrue|
                   |$EmptyMode|))
  (setq form (second arg0))
  (setq exprlist (cddr arg0))
  (setq |$insideExpressionIfTrue| nil)
  (setq |$insideWhereIfTrue| t)
  (setq e eInit)
  (when (dolist (item exprList t)
          (setq tmp1 (|comp| item |$EmptyMode| e))
          (unless tmp1 (return nil))
          (setq e (third tmp1)))
  (setq |$insideWhereIfTrue| nil)
  (setq tmp1 (|comp| (|macroExpand| form (setq eBefore e)) m e))
  (when tmp1
    (setq x (first tmp1))
    (setq m (second tmp1))
    (setq eAfter (third tmp1))
    (setq del (|deltaContour| eAfter eBefore))
    (if del
      (setq eFinal (|addContour| del eInit))
      (setq eFinal eInit))
    (list x m eFinal)))))
```

———-

Chapter 6

Post Transformers

6.1 Direct called postparse routines

6.1.1 defun postTransform

[postTran p192]
[postTransform identp (vol5)]
[postTransformCheck p195]
[aplTran p226]

— defun postTransform —

```
(defun |postTransform| (y)
 (let (x tmp1 tmp2 tmp3 tmp4 tmp5 tt l u)
  (setq x y)
  (setq u (|postTran| x))
  (when
   (and (pairp u) (eq (qcar u) '|@Tuple|)
        (progn
         (setq tmp1 (qcdr u))
         (and (pairp tmp1)
              (progn (setq tmp2 (reverse tmp1)) t)
              (pairp tmp2)
              (progn
               (setq tmp3 (qcar tmp2))
               (and (pairp tmp3)
                    (eq (qcar tmp3) '|:|)
                    (progn
                     (setq tmp4 (qcdr tmp3))
                     (and (pairp tmp4)
                          (progn
                           (setq y (qcar tmp4))
```

```
                        (setq tmp5 (qcdr tmp4))
                        (and (pairp tmp5)
                             (eq (qcdr tmp5) nil)
                             (progn (setq tt (qcar tmp5)) t)))))))
                   (progn (setq l (qcdr tmp2)) t)
                   (progn (setq l (nreverse l)) t)))
          (dolist (x l t) (unless (identp x) (return nil))))
    (setq u (list '|:| (cons 'listof (append l (list y))) tt)))
  (|postTransformCheck| u)
  (|aplTran| u)))
```

———-

6.1.2 defun postTran

[postAtom p193]
[postTran p192]
[pairp p??]
[qcar p??]
[qcdr p??]
[unTuple p233]
[postTranList p194]
[postForm p196]
[postOp p193]
[postScriptsForm p194]

— defun postTran —

```
(defun |postTran| (x)
 (let (op f tmp1 a tmp2 tmp3 b y)
  (if (atom x)
   (|postAtom| x)
   (progn
    (setq op (car x))
    (cond
     ((and (atom op) (setq f (getl op '|postTran|)))
      (funcall f x))
     ((and (pairp op) (eq (qcar op) '|elt|)
           (progn
            (setq tmp1 (qcdr op))
            (and (pairp tmp1)
                 (progn
                   (setq a (qcar tmp1))
                   (setq tmp2 (qcdr tmp1))
                   (and (pairp tmp2)
                        (eq (qcdr tmp2) nil)
                        (progn (setq b (qcar tmp2)) t))))))
```

```
     (cons (|postTran| op) (cdr (|postTran| (cons b (cdr x))))))
    ((and (pairp op) (eq (qcar op) '|Scripts|))
     (|postScriptsForm| op
      (dolist (y (rest x) tmp3)
       (setq tmp3 (append tmp3 (|unTuple| (|postTran| y)))))))
    ((nequal op (setq y (|postOp| op)))
     (cons y (|postTranList| (cdr x))))
    (t (|postForm| x)))))
```

———-

6.1.3 defun postOp

— **defun postOp** —

```
(defun |postOp| (x)
 (declare (special $boot))
 (cond
  ((eq x '|:=|) (if $boot 'spadlet 'let))
  ((eq x '|:-|) 'letd)
  ((eq x '|Attribute|) 'attribute)
  (t x)))
```

———-

6.1.4 defun postAtom

[$boot p**??**]

— **defun postAtom** —

```
(defun |postAtom| (x)
 (declare (special $boot))
  (cond
    ($boot x)
    ((eql x 0) '(|Zero|))
    ((eql x 1) '(|One|))
    ((eq x t) 't$)
    ((and (identp x) (getdatabase x 'niladic)) (list x))
    (t x)))
```

———-

6.1.5 defun postTranList

[postTran p192]

— **defun postTranList** —

```
(defun |postTranList| (x)
 (loop for y in x collect (|postTran| y)))
```

————-

6.1.6 defun postScriptsForm

[getScriptName p230]
[length p**??**]
[postTranScripts p194]

— **defun postScriptsForm** —

```
(defun |postScriptsForm| (arg0 argl)
 (let ((op (second arg0)) (a (third arg0)))
  (cons (|getScriptName| op a (|#| argl))
        (append (|postTranScripts| a) argl))))
```

————-

6.1.7 defun postTranScripts

[postTranScripts p194]
[postTran p192]

— **defun postTranScripts** —

```
(defun |postTranScripts| (a)
 (labels (
  (fn (x)
   (if (and (pairp x) (eq (qcar x) '|@Tuple|))
    (qcdr x)
    (list x))))
 (let (tmp1 tmp2 tmp3)
  (cond
   ((and (pairp a) (eq (qcar a) '|PrefixSC|)
         (progn
          (setq tmp1 (qcdr a))
```

```
          (and (pairp tmp1) (eq (qcdr tmp1) nil))))
     (|postTranScripts| (qcar tmp1)))
   ((and (pairp a) (eq (qcar a) '|;|))
    (dolist (y (qcdr a) tmp2)
     (setq tmp2 (append tmp2 (|postTranScripts| y)))))
   ((and (pairp a) (eq (qcar a) '|,|))
    (dolist (y (qcdr a) tmp3)
     (setq tmp3 (append tmp3 (fn (|postTran| y))))))
   (t (list (|postTran| a)))))))
```

———-

6.1.8 defun postTransformCheck

[postcheck p195]
[$defOp p??]

— **defun postTransformCheck** —

```
(defun |postTransformCheck| (x)
 (let (|$defOp|)
 (declare (special |$defOp|))
  (setq |$defOp| nil)
  (|postcheck| x)))
```

———-

6.1.9 defun postcheck

[setDefOp p225]
[postcheck p195]

— **defun postcheck** —

```
(defun |postcheck| (x)
 (cond
  ((atom x) nil)
  ((and (pairp x) (eq (qcar x) 'def) (pairp (qcdr x)))
   (|setDefOp|  (qcar (qcdr x)))
   (|postcheck| (qcdr (qcdr x))))
  ((and (pairp x) (eq (qcar x) 'quote)) nil)
  (t (|postcheck| (car x)) (|postcheck| (cdr x)))))
```

———-

6.1.10 defun postError

[nequal p??]
[bumperrorcount p297]
[$defOp p??]
[$InteractiveMode p??]
[$postStack p??]

— **defun postError** —

```
(defun |postError| (msg)
 (let (xmsg)
  (declare (special |$defOp| |$postStack| |$InteractiveMode|))
  (bumperrorcount '|precompilation|)
  (setq xmsg
   (if (and (nequal |$defOp| '|$defOp|) (null |$InteractiveMode|))
    (cons |$defOp| (cons ": " msg))
    msg))
  (push xmsg |$postStack|)
  nil))
```

———-

6.1.11 defun postForm

[postTranList p194]
[internl p??]
[postTran p192]
[postError p196]
[bright p??]
[$boot p??]

— **defun postForm** —

```
(defun |postForm| (u)
 (let (op argl arglp numOfArgs opp x)
 (declare (special $boot))
  (seq
   (setq op (car u))
   (setq argl (cdr u))
   (setq x
    (cond
     ((atom op)
       (setq arglp (|postTranList| argl))
       (setq opp
         (seq
```

```
        (exit op)
        (when $boot (exit op))
        (when (or (getl op '|Led|)  (getl op '|Nud|) (eq op 'in)) (exit op))
        (setq numOfArgs
         (cond
          ((and (pairp arglp) (eq (qcdr arglp) nil) (pairp (qcar arglp))
                (eq (qcar (qcar arglp)) '|@Tuple|))
            (|#| (qcdr (qcar arglp))))
          (t 1)))
        (internl '* (princ-to-string numOfArgs) (pname op))))
    (cons opp arglp))
   ((and (pairp op) (eq (qcar op) '|Scripts|))
    (append (|postTran| op) (|postTranList| argl)))
   (t
    (setq u (|postTranList| u))
    (cond
     ((and (pairp u) (pairp (qcar u)) (eq (qcar (qcar u)) '|@Tuple|))
       (|postError|
        (cons "  "
         (append (|bright| u)
          (list "is illegal because tuples cannot be applied!" '|%l|
                "   Did you misuse infix dot?"))))))
     u)))
(cond
 ((and (pairp x) (pairp (qcdr x)) (eq (qcdr (qcdr x)) nil)
       (pairp (qcar (qcdr x))) (eq (qcar (qcar (qcdr x))) '|@Tuple|))
  (cons (car x) (qcdr (qcar (qcdr x)))))
 (t x)))))
```

———-

6.2 Indirect called postparse routines

In the **postTran** function there is the code:

```
((and (atom op) (setq f (getl op '|postTran|)))
 (funcall f x))
```

The functions in this section are called through the symbol-plist of the symbol being parsed. The original list read:

```
add             postAdd
@               postAtSign
:BF:            postBigFloat
Block           postBlock
CATEGORY        postCategory
```

```
  COLLECT        postCollect
  :              postColon
  ::             postColonColon
  ,              postComma
  construct      postConstruct
  ==             postDef
  =>             postExit
  if             postIf
  in             postin       ;" the infix operator version of in"
  IN             postIn       ;" the iterator form of in"
  Join           postJoin
  ->             postMapping
  ==>            postMDef
  pretend        postPretend
  QUOTE          postQUOTE
  Reduce         postReduce
  REPEAT         postRepeat
  Scripts        postScripts
  ;              postSemiColon
  Signature      postSignature
  /              postSlash
  @Tuple         postTuple
  TupleCollect   postTupleCollect
  where          postWhere
  with           postWith
```

6.2.1 defun postAdd

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|add| '|postTran|) '|postAdd|))
```

————-

6.2.2 defun postAdd

[postTran p192]
[postCapsule p199]

— **defun postAdd** —

```
(defun |postAdd| (arg)
 (if (null (cddr arg))
  (|postCapsule| (second arg))
```

```
  (list '|add| (|postTran| (second arg)) (|postCapsule| (third arg)))))
```

———-

6.2.3 defun postCapsule

[checkWarning p301]
[postBlockItem p??]
[postBlockItemList p199]
[postFlatten p208]

— **defun postCapsule** —

```
(defun |postCapsule| (x)
 (let (op)
  (cond
   ((null (and (pairp x) (progn (setq op (qcar x)) t)))
     (|checkWarning| (list "Apparent indentation error following add")))
   ((or (integerp op) (eq op '==))
     (list 'capsule (|postBlockItem| x)))
   ((eq op '|;|)
     (cons 'capsule (|postBlockItemList| (|postFlatten| x '|;|))))
   ((eq op '|if|)
     (list 'capsule (|postBlockItem| x)))
   (t (|checkWarning| (list "Apparent indentation error following add"))))))
```

———-

6.2.4 defun postBlockItemList

[postBlockItem p??]

— **defun postBlockItemList** —

```
(defun |postBlockItemList| (args)
 (let (result)
  (dolist (item args (nreverse result))
   (push (|postBlockItem| item) result))))
```

———-

6.2.5 defun postAtSign

— **postvars** —

```
(eval-when (eval load)
 (setf (get '@ '|postTran|) '|postAtSign|))
```

————-

6.2.6 defun postAtSign

[postTran p192]
[postType p200]

— **defun postAtSign** —

```
(defun |postAtSign| (arg)
 (cons '@ (cons (|postTran| (second arg)) (|postType| (third arg)))))
```

————-

6.2.7 defun postType

[postTran p192]
[unTuple p233]

— **defun postType** —

```
(defun |postType| (typ)
 (let (source target)
  (cond
   ((and (pairp typ) (eq (qcar typ) '->) (pairp (qcdr typ))
         (pairp (qcdr (qcdr typ))) (eq (qcdr (qcdr (qcdr typ))) nil))
     (setq source (qcar (qcdr typ)))
     (setq target (qcar (qcdr (qcdr typ))))
     (cond
      ((eq source '|constant|)
        (list (list (|postTran| target)) '|constant|))
      (t
        (list (cons '|Mapping|
                (cons (|postTran| target)
                      (|unTuple| (|postTran| source))))))))
   ((and (pairp typ) (eq (qcar typ) '->)
         (pairp (qcdr typ)) (eq (qcdr (qcdr typ)) nil))
```

```
      (list (list '|Mapping| (|postTran| (qcar (qcdr typ))))))
   (t (list (|postTran| typ))))
```

———-

6.2.8 defun postBigFloat

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|:BF:| '|postTran|) '|postBigFloat|))
```

———-

6.2.9 defun postBigFloat

[postTran p192]
[$boot p**??**]
[$InteractiveMode p**??**]

— **defun postBigFloat** —

```
(defun |postBigFloat| (arg)
 (let (mant expon eltword)
  (declare (special $boot |$InteractiveMode|))
  (setq mant (second arg))
  (setq expon (cddr arg))
  (if $boot
   (times (float mant) (expt (float 10) expon))
   (progn
    (setq eltword (if |$InteractiveMode| '|$elt| '|elt|))
    (|postTran|
     (list (list eltword '(|Float|) '|float|)
       (list '|,| (list '|,| mant expon) 10)))))))
```

———-

6.2.10 defun postBlock

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|Block| '|postTran|) '|postBlock|))
```

———-

6.2.11 defun postBlock

[postBlockItemList p199]
[postTran p192]

— **defun postBlock** —

```
(defun |postBlock| (arg)
 (let (tmp1 x y)
  (setq tmp1 (reverse (cdr arg)))
  (setq x (car tmp1))
  (setq y (nreverse (cdr tmp1)))
  (cons 'seq
   (append (|postBlockItemList| y) (list (list '|exit| (|postTran| x)))))))
```

———-

6.2.12 defun postCategory

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'category '|postTran|) '|postCategory|))
```

———-

6.2.13 defun postCategory

[postTran p192]
[nreverse0 p**??**]
[$insidePostCategoryIfTrue p**??**]

— **defun postCategory** —

```
(defun |postCategory| (u)
 (declare (special |$insidePostCategoryIfTrue|))
```

```
(labels (
 (fn (arg)
  (let (|$insidePostCategoryIfTrue|)
  (declare (special |$insidePostCategoryIfTrue|))
   (setq |$insidePostCategoryIfTrue| t)
   (|postTran| arg))) )
(let ((z (cdr u)) op tmp1)
 (if (null z)
  u
  (progn
    (setq op (if |$insidePostCategoryIfTrue| 'progn 'category))
    (cons op (dolist (x z (nreverse0 tmp1)) (push (fn x) tmp1))))))))
```

———-

6.2.14 defun postCollect,finish

[qcar p??]
[qcdr p??]
[postMakeCons p204]
[tuple2List p302]
[postTranList p194]

— **defun postCollect,finish** —

```
(defun |postCollect,finish| (op itl y)
 (let (tmp2 tmp5 newBody)
  (cond
   ((and (pairp y) (eq (qcar y) '|:|)
         (pairp (qcdr y)) (eq (qcdr (qcdr y)) nil))
    (list 'reduce '|append| 0 (cons op (append itl (list (qcar (qcdr y)))))))
   ((and (pairp y) (eq (qcar y) '|Tuple|))
    (setq newBody
     (cond
      ((dolist (x (qcdr y) tmp2)
        (setq tmp2
         (or tmp2 (and (pairp x) (eq (qcar x) '|:|)
                       (pairp (qcdr x)) (eq (qcdr (qcdr x)) nil)))))
       (|postMakeCons| (qcdr y)))
      ((dolist (x (qcdr y) tmp5)
         (setq tmp5 (or tmp5 (and (pairp x) (eq (qcar x) 'segment)))))
        (|tuple2List| (qcdr y)))
      (t (cons '|construct| (|postTranList| (qcdr y))))))
     (list 'reduce '|append| 0 (cons op (append itl (list newBody)))))
   (t (cons op (append itl (list y)))))))
```

———-

6.2.15 defun postMakeCons

[postMakeCons p204]
[postTran p192]

— **defun postMakeCons** —

```
(defun |postMakeCons| (args)
 (let (a b)
  (cond
   ((null args) '|nil|)
   ((and (pairp args) (pairp (qcar args)) (eq (qcar (qcar args)) '|:|)
         (pairp (qcdr (qcar args))) (eq (qcdr (qcdr (qcar args))) nil))
     (setq a (qcar (qcdr (qcar args))))
     (setq b (qcdr args))
     (if b
       (list '|append| (|postTran| a) (|postMakeCons| b))
       (|postTran| a)))
   (t (list '|cons| (|postTran| (car args)) (|postMakeCons| (cdr args)))))))
```

———-

6.2.16 defun postCollect

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'collect '|postTran|) '|postCollect|))
```

———-

6.2.17 defun postCollect

[postCollect,finish p203]
[postCollect p204]
[postIteratorList p205]
[postTran p192]

— **defun postCollect** —

```
(defun |postCollect| (arg)
 (let (constructOp tmp3 m itl x)
  (setq constructOp (car arg))
```

```
  (setq tmp3 (reverse (cdr arg)))
  (setq x (car tmp3))
  (setq m (nreverse (cdr tmp3)))
  (cond
   ((and (pairp x) (pairp (qcar x)) (eq (qcar (qcar x)) '|elt|)
         (pairp (qcdr (qcar x))) (pairp (qcdr (qcdr (qcar x))))
         (eq (qcdr (qcdr (qcdr (qcar x)))) nil)
         (eq (qcar (qcdr (qcdr (qcar x)))) '|construct|))
    (|postCollect|
     (cons (list '|elt| (qcar (qcdr (qcar x))) 'collect)
      (append m (list (cons '|construct| (qcdr x)))))))
   (t
    (setq itl (|postIteratorList| m))
    (setq x
     (if (and (pairp x) (eq (qcar x) '|construct|)
              (pairp (qcdr x)) (eq (qcdr (qcdr x)) nil))
       (qcar (qcdr x))
       x))
    (|postCollect,finish| constructOp itl (|postTran| x))))))
```

———-

6.2.18 defun postIteratorList

[postTran p192]
[postInSeq p214]
[postIteratorList p205]

— **defun postIteratorList** —

```
(defun |postIteratorList| (args)
 (let (z p y u a b)
  (cond
   ((pairp args)
     (setq p (|postTran| (qcar args)))
     (setq z (qcdr args))
     (cond
       ((and (pairp p) (eq (qcar p) 'in) (pairp (qcdr p))
             (pairp (qcdr (qcdr p))) (eq (qcdr (qcdr (qcdr p))) nil))
        (setq y (qcar (qcdr p)))
        (setq u (qcar (qcdr (qcdr p))))
        (cond
          ((and (pairp u) (eq (qcar u) '|\||) (pairp (qcdr u))
                (pairp (qcdr (qcdr u))) (eq (qcdr (qcdr (qcdr u))) nil))
            (setq a (qcar (qcdr u)))
            (setq b (qcar (qcdr (qcdr u))))
            (cons (list 'in y (|postInSeq| a))
```

```
            (cons (list '|\|| b)
              (|postIteratorList| z))))
         (t (cons (list 'in y (|postInSeq| u)) (|postIteratorList| z)))))
      (t (cons p (|postIteratorList| z)))))
  (t args))))
```

———-

6.2.19 defun postColon

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|:| '|postTran|) '|postColon|))
```

———-

6.2.20 defun postColon

[postTran p192]
[postType p200]

— **defun postColon** —

```
(defun |postColon| (u)
 (cond
  ((and (pairp u) (eq (qcar u) '|:|)
        (pairp (qcdr u)) (eq (qcdr (qcdr u)) nil))
    (list '|:| (|postTran| (qcar (qcdr u)))))
  ((and (pairp u) (eq (qcar u) '|:|) (pairp (qcdr u))
        (pairp (qcdr (qcdr u))) (eq (qcdr (qcdr (qcdr u))) nil))
   (cons '|:| (cons (|postTran| (second u)) (|postType| (third u)))))))
```

———-

6.2.21 defun postColonColon

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|::| '|postTran|) '|postColonColon|))
```

———-

6.2.22 defun postColonColon

[postForm p196]
[$boot p??]

— **defun postColonColon** —

```
(defun |postColonColon| (u)
  (if (and $boot (pairp u) (eq (qcar u) '|::|) (pairp (qcdr u))
           (pairp (qcdr (qcdr u))) (eq (qcdr (qcdr (qcdr u))) nil))
   (intern (princ-to-string (third u)) (second u))
   (|postForm| u)))
```

———-

6.2.23 defun postComma

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|,| '|postTran|) '|postComma|))
```

———-

6.2.24 defun postComma

[postTuple p223]
[comma2Tuple p207]

— **defun postComma** —

```
(defun |postComma| (u)
 (|postTuple| (|comma2Tuple| u)))
```

———-

6.2.25 defun comma2Tuple

[postFlatten p208]

— **defun comma2Tuple** —

```
(defun |comma2Tuple| (u)
 (cons '|@Tuple| (|postFlatten| u '|,|)))
```

————-

6.2.26 defun postFlatten

[postFlatten p208]

— **defun postFlatten** —

```
(defun |postFlatten| (x op)
 (let (a b)
  (cond
   ((and (pairp x) (equal (qcar x) op) (pairp (qcdr x))
         (pairp (qcdr (qcdr x))) (eq (qcdr (qcdr (qcdr x))) nil))
     (setq a (qcar (qcdr x)))
     (setq b (qcar (qcdr (qcdr x))))
     (append (|postFlatten| a op) (|postFlatten| b op)))
   (t (list x)))))
```

————-

6.2.27 defun postConstruct

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|construct| '|postTran|) '|postConstruct|))
```

————-

6.2.28 defun postConstruct

[comma2Tuple p207]
[postTranSegment p209]
[postMakeCons p204]
[tuple2List p302]
[postTranList p194]

[postTran p192]

— **defun postConstruct** —

```
(defun |postConstruct| (u)
 (let (b a tmp4 tmp7)
  (cond
   ((and (pairp u) (eq (qcar u) '|construct|)
         (pairp (qcdr u)) (eq (qcdr (qcdr u)) nil))
     (setq b (qcar (qcdr u)))
     (setq a
      (if (and (pairp b) (eq (qcar b) '|,|))
       (|comma2Tuple| b)
       b))
     (cond
      ((and (pairp a) (eq (qcar a) 'segment) (pairp (qcdr a))
            (pairp (qcdr (qcdr a))) (eq (qcdr (qcdr (qcdr a))) nil))
       (list '|construct| (|postTranSegment| (second a) (third a))))
      ((and (pairp a) (eq (qcar a) '|@Tuple|))
        (cond
         ((dolist (x (qcdr a) tmp4)
           (setq tmp4
            (or tmp4
             (and (pairp x) (eq (qcar x) '|:|)
                  (pairp (qcdr x)) (eq (qcdr (qcdr x)) nil)))))
          (|postMakeCons| (qcdr a)))
         ((dolist (x (qcdr a) tmp7)
           (setq tmp7 (or tmp7 (and (pairp x) (eq (qcar x) 'segment)))))
          (|tuple2List| (qcdr a)))
         (t (cons '|construct| (|postTranList| (qcdr a))))))
      (t (list '|construct| (|postTran| a)))))
   (t u))))
```

———-

6.2.29 defun postTranSegment

[postTran p192]

— **defun postTranSegment** —

```
(defun |postTranSegment| (p q)
  (list 'segment (|postTran| p) (when q (|postTran| q))))
```

———-

6.2.30 defun postDef

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|==| '|postTran|) '|postDef|))
```

————-

6.2.31 defun postDef

[postMDef p216]
[recordHeaderDocumentation p??]
[nequal p??]
[postTran p192]
[postDefArgs p211]
[nreverse0 p??]
[$boot p??]
[$maxSignatureLineNumber p??]
[$headerDocumentation p??]
[$docList p??]
[$InteractiveMode p??]

— **defun postDef** —

```
(defun |postDef| (arg)
 (let (defOp rhs lhs targetType tmp1 op argl newLhs
       argTypeList typeList form specialCaseForm tmp4 tmp6 tmp8)
 (declare (special $boot |$maxSignatureLineNumber| |$headerDocumentation|
                   |$docList| |$InteractiveMode|))
  (setq defOp (first arg))
  (setq lhs (second arg))
  (setq rhs (third arg))
  (if (and (pairp lhs) (eq (qcar lhs) '|macro|)
           (pairp (qcdr lhs)) (eq (qcdr (qcdr lhs)) nil))
   (|postMDef| (list '==> (second lhs) rhs))
   (progn
    (unless $boot (|recordHeaderDocumentation| nil))
    (when (nequal |$maxSignatureLineNumber| 0)
      (setq |$docList|
       (cons (cons '|constructor| |$headerDocumentation|) |$docList|))
      (setq |$maxSignatureLineNumber| 0))
    (setq lhs (|postTran| lhs))
    (setq tmp1
     (if (and (pairp lhs) (eq (qcar lhs) '|:|)) (cdr lhs) (list lhs nil)))
```

```
(setq form (first tmp1))
(setq targetType (second tmp1))
(when (and (null |$InteractiveMode|) (atom form)) (setq form (list form)))
(setq newLhs
 (if (atom form)
  form
  (progn
   (setq tmp1
    (dolist (x form (nreverse0 tmp4))
     (push
       (if (and (pairp x) (eq (qcar x) '|:|) (pairp (qcdr x))
                (pairp (qcdr (qcdr x))) (eq (qcdr (qcdr (qcdr x))) nil))
        (second x)
        x)
       tmp4)))
   (setq op (car tmp1))
   (setq argl (cdr tmp1))
   (cons op (|postDefArgs| argl)))))
(setq argTypeList
 (unless (atom form)
  (dolist (x (cdr form) (nreverse0 tmp6))
   (push
     (when (and (pairp x) (eq (qcar x) '|:|) (pairp (qcdr x))
           (pairp (qcdr (qcdr x))) (eq (qcdr (qcdr (qcdr x))) nil))
       (third x))
     tmp6))))
(setq typeList (cons targetType argTypeList))
(when (atom form) (setq form (list form)))
(setq specialCaseForm (dolist (x form (nreverse tmp8)) (push nil tmp8)))
(list 'def newLhs typeList specialCaseForm (|postTran| rhs))))))
```

———-

6.2.32 defun postDefArgs

[postError p196]
[postDefArgs p211]

— **defun postDefArgs** —

```
(defun |postDefArgs| (args)
 (let (a b)
  (cond
   ((null args) args)
   ((and (pairp args) (pairp (qcar args)) (eq (qcar (qcar args)) '|:|)
         (pairp (qcdr (qcar args))) (eq (qcdr (qcdr (qcar args))) nil))
    (setq a (qcar (qcdr (qcar args))))
```

```
    (setq b (qcdr args))
    (cond
     (b (|postError|
         (list  "   Argument" a "of indefinite length must be last")))
     ((or (atom a) (and (pairp a) (eq (qcar a) 'quote)))
       a)
     (t
       (|postError|
         (list "   Argument" a "of indefinite length must be a name")))))
  (t (cons (car args) (|postDefArgs| (cdr args)))))))
```

———-

6.2.33 defun postExit

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|=>| '|postTran|) '|postExit|))
```

———-

6.2.34 defun postExit

[postTran p192]

— **defun postExit** —

```
(defun |postExit| (arg)
 (list 'if (|postTran| (second arg))
  (list '|exit| (|postTran| (third arg)))
   '|noBranch|))
```

———-

6.2.35 defun postIf

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|if| '|postTran|) '|postIf|))
```

———-

6.2.36 defun postIf

[nreverse0 p??]
[postTran p192]
[$boot p??]

— **defun postIf** —

```
(defun |postIf| (arg)
 (let (tmp1)
  (if (null (and (pairp arg) (eq (qcar arg) '|if|)))
   arg
   (cons 'if
    (dolist (x (qcdr arg) (nreverse0 tmp1))
     (push
       (if (and (null (setq x (|postTran| x))) (null $boot)) '|noBranch| x)
       tmp1))))))
```

———-

6.2.37 defun postin

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|in| '|postTran|) '|postin|))
```

———-

6.2.38 defun postin

[systemErrorHere p??]
[postTran p192]
[postInSeq p214]

— **defun postin** —

```
(defun |postin| (arg)
 (if (null (and (pairp arg) (eq (qcar arg) '|in|) (pairp (qcdr arg))
                (pairp (qcdr (qcdr arg))) (eq (qcdr (qcdr (qcdr arg))) nil)))
```

```
   (|systemErrorHere| "postin")
   (list '|in| (|postTran| (second arg)) (|postInSeq| (third arg)))))
```

———-

6.2.39 defun postInSeq

[postTranSegment p209]
[tuple2List p302]
[postTran p192]

— **defun postInSeq** —

```
(defun |postInSeq| (seq)
 (cond
  ((and (pairp seq) (eq (qcar seq) 'segment) (pairp (qcdr seq))
        (pairp (qcdr (qcdr seq))) (eq (qcdr (qcdr (qcdr seq))) nil))
     (|postTranSegment| (second seq) (third seq)))
  ((and (pairp seq) (eq (qcar seq) '|@Tuple|))
     (|tuple2List| (qcdr seq)))
  (t (|postTran| seq))))
```

———-

6.2.40 defun postIn

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'in '|postTran|) '|postIn|))
```

———-

6.2.41 defun postIn

[systemErrorHere p??]
[postTran p192]
[postInSeq p214]

— **defun postIn** —

```
(defun |postIn| (arg)
  (if (null (and (pairp arg) (eq (qcar arg) 'in) (pairp (qcdr arg))
                 (pairp (qcdr (qcdr arg))) (eq (qcdr (qcdr (qcdr arg))) nil)))
    (|systemErrorHere| "postIn")
    (list 'in (|postTran| (second arg)) (|postInSeq| (third arg)))))
```

———-

6.2.42 defun postJoin

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|Join| '|postTran|) '|postJoin|))
```

———-

6.2.43 defun postJoin

[postTran p192]
[postTranList p194]

— **defun postJoin** —

```
(defun |postJoin| (arg)
 (let (a l al)
  (setq a (|postTran| (cadr arg)))
  (setq l (|postTranList| (cddr arg)))
  (when (and (pairp l) (eq (qcdr l) nil) (pairp (qcar l))
             (member (qcar (qcar l)) '(attribute signature)))
    (setq l (list (list 'category (qcar l)))))
  (setq al (if (and (pairp a) (eq (qcar a) '|@Tuple|)) (qcdr a) (list a)))
  (cons '|Join| (append al l))))
```

———-

6.2.44 defun postMapping

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|->| '|postTran|) '|postMapping|))
```

———-

6.2.45 defun postMapping

[postTran p192]
[unTuple p233]

— **defun postMapping** —

```
(defun |postMapping| (u)
 (if (null (and (pairp u) (eq (qcar u) '->) (pairp (qcdr u))
           (pairp (qcdr (qcdr u))) (eq (qcdr (qcdr (qcdr u))) nil)))
   u
   (cons '|Mapping|
    (cons (|postTran| (third u))
          (|unTuple| (|postTran| (second u)))))))
```

———-

6.2.46 defun postMDef

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|==>| '|postTran|) '|postMDef|))
```

———-

6.2.47 defun postMDef

[postTran p192]
[throwkeyedmsg p**??**]
[nreverse0 p**??**]
[$InteractiveMode p**??**]
[$boot p**??**]

— **defun postMDef** —

```
(defun |postMDef| (arg)
 (let (rhs lhs tmp1 targetType form newLhs typeList tmp4 tmp5 tmp8)
 (declare (special |$InteractiveMode| $boot))
  (setq lhs (second arg))
  (setq rhs (third arg))
  (cond
   ((and |$InteractiveMode| (null $boot))
    (setq lhs (|postTran| lhs))
    (if (null (identp lhs))
      (|throwkeyedmsg| 's2ip0001 nil)
      (list 'mdef lhs nil nil (|postTran| rhs))))
   (t
    (setq lhs (|postTran| lhs))
    (setq tmp1
     (if (and (pairp lhs) (eq (qcar lhs) '|:|)) (cdr lhs) (list lhs nil)))
    (setq form (first tmp1))
    (setq targetType (second tmp1))
    (setq form (if (atom form) (list form) form))
    (setq newLhs
     (dolist (x form (nreverse0 tmp4))
      (push
       (if (and (pairp x) (eq (qcar x) '|:|) (pairp (qcdr x))) (second x) x)
       tmp4)))
    (setq typeList
     (cons targetType
      (dolist (x (qcdr form) (nreverse0 tmp5))
       (push
         (when (and (pairp x) (eq (qcar x) '|:|) (pairp (qcdr x))
                    (pairp (qcdr (qcdr x))) (eq (qcdr (qcdr (qcdr x))) nil))
           (third x))
         tmp5))))
    (list 'mdef newLhs typeList
     (dolist (x form (nreverse0 tmp8)) (push nil tmp8))
     (|postTran| rhs))))))
```

———-

6.2.48 defun postPretend

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|pretend| '|postTran|) '|postPretend|))
```

———-

6.2.49 defun postPretend

[postTran p192]
[postType p200]

— **defun postPretend** —

```
(defun |postPretend| (arg)
 (cons '|pretend| (cons (|postTran| (second arg)) (|postType| (third arg)))))
```

————-

6.2.50 defun postQUOTE

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'quote '|postTran|) '|postQUOTE|))
```

————-

6.2.51 defun postQUOTE

— **defun postQUOTE** —

```
(defun |postQUOTE| (arg) arg)
```

————-

6.2.52 defun postReduce

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|Reduce| '|postTran|) '|postReduce|))
```

————-

6.2.53 defun postReduce

[postTran p192]
[postReduce p219]
[$InteractiveMode p??]

— **defun postReduce** —

```
(defun |postReduce| (arg)
 (let (op expr g)
  (setq op (second arg))
  (setq expr (third arg))
  (if (or |$InteractiveMode| (and (pairp expr) (eq (qcar expr) 'collect)))
    (list 'reduce op 0 (|postTran| expr))
    (|postReduce|
     (list '|Reduce| op
      (list 'collect
       (list 'in (setq g (gensym)) expr)
        (list '|construct| g)))))))
```

———-

6.2.54 defun postRepeat

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'repeat '|postTran|) '|postRepeat|))
```

———-

6.2.55 defun postRepeat

[postIteratorList p205]
[postTran p192]

— **defun postRepeat** —

```
(defun |postRepeat| (arg)
 (let (tmp1 x m)
  (setq tmp1 (reverse (cdr arg)))
  (setq x (car tmp1))
  (setq m (nreverse (cdr tmp1)))
  (cons 'repeat (append (|postIteratorList| m) (list (|postTran| x))))))
```

———-

6.2.56 defun postScripts

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|Scripts| '|postTran|) '|postScripts|))
```

———-

6.2.57 defun postScripts

[getScriptName p230]
[postTranScripts p194]

— **defun postScripts** —

```
(defun |postScripts| (arg)
 (cons (|getScriptName| (second arg) (third arg) 0)
       (|postTranScripts| (third arg))))
```

———-

6.2.58 defun postSemiColon

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|;| '|postTran|) '|postSemiColon|))
```

———-

6.2.59 defun postSemiColon

[postBlock p202]
[postFlattenLeft p221]

— **defun postSemiColon** —

```
(defun |postSemiColon| (u)
 (|postBlock| (cons '|Block| (|postFlattenLeft| u '|;|))))
```

———-

6.2.60 defun postFlattenLeft

[postFlattenLeft p221]

— **defun postFlattenLeft** —

```
(defun |postFlattenLeft| (x op)
 (let (a b)
  (cond
   ((and (pairp x) (equal (qcar x) op) (pairp (qcdr x))
         (pairp (qcdr (qcdr x))) (eq (qcdr (qcdr (qcdr x))) nil))
     (setq a (qcar (qcdr x)))
     (setq b (qcar (qcdr (qcdr x))))
     (append (|postFlattenLeft| a op) (list b)))
   (t (list x)))))
```

———-

6.2.61 defun postSignature

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|Signature| '|postTran|) '|postSignature|))
```

———-

6.2.62 defun postSignature

[pairp p**??**]
[postType p200]
[removeSuperfluousMapping p**??**]
[killColons p222]

— **defun postSignature** —

```
(defun |postSignature| (arg)
 (let (sig sig1 op)
  (setq op (second arg))
  (setq sig (third arg))
  (when (and (pairp sig) (eq (qcar sig) '->))
    (setq sig1 (|postType| sig))
    (setq op (|postAtom| (if (stringp op) (setq op (intern op)) op)))
    (cons 'signature
     (cons op (|removeSuperfluousMapping| (|killColons| sig1)))))))
```

———-

6.2.63 defun killColons

[killColons p222]

— **defun killColons** —

```
(defun |killColons| (x)
 (cond
  ((atom x) x)
  ((and (pairp x) (eq (qcar x) '|Record|)) x)
  ((and (pairp x) (eq (qcar x) '|Union|)) x)
  ((and (pairp x) (eq (qcar x) '|:|) (pairp (qcdr x))
        (pairp (qcdr (qcdr x))) (eq (qcdr (qcdr (qcdr x))) nil))
   (|killColons| (third x)))
  (t (cons (|killColons| (car x)) (|killColons| (cdr x))))))
```

———-

6.2.64 defun postSlash

— **postvars** —

```
(eval-when (eval load)
 (setf (get '/ '|postTran|) '|postSlash|))
```

———-

6.2.65 defun postSlash

[postTran p192]

— **defun postSlash** —

```
(defun |postSlash| (arg)
  (if (stringp (second arg))
   (|postTran| (list '|Reduce| (intern (second arg)) (third arg) ))
   (list '/ (|postTran| (second arg)) (|postTran| (third arg)))))
```

———-

6.2.66 defun postTuple

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|@Tuple| '|postTran|) '|postTuple|))
```

———-

6.2.67 defun postTuple

[postTranList p194]

— **defun postTuple** —

```
(defun |postTuple| (arg)
 (cond
  ((and (pairp arg) (eq (qcdr arg) nil) (eq (qcar arg) '|@Tuple|))
   arg)
  ((and (pairp arg) (eq (qcar arg) '|@Tuple|) (pairp (qcdr arg)))
   (cons '|@Tuple| (|postTranList| (cdr arg))))))
```

———-

6.2.68 defun postTupleCollect

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|TupleCollect| '|postTran|) '|postTupleCollect|))
```

———-

6.2.69 defun postTupleCollect

[postCollect p204]

— **defun postTupleCollect** —

```
(defun |postTupleCollect| (arg)
 (let (constructOp tmp1 x m)
  (setq constructOp (car arg))
  (setq tmp1 (reverse (cdr arg)))
  (setq x (car tmp1))
  (setq m (nreverse (cdr tmp1)))
  (|postCollect| (cons constructOp (append m (list (list '|construct| x)))))))
```

————-

6.2.70 defun postWhere

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|where| '|postTran|) '|postWhere|))
```

————-

6.2.71 defun postWhere

[postTran p192]
[postTranList p194]

— **defun postWhere** —

```
(defun |postWhere| (arg)
 (let (b x)
  (setq b (third arg))
  (setq x (if (and (pairp b) (eq (qcar b) '|Block|)) (qcdr b) (list b)))
  (cons '|where| (cons (|postTran| (second arg)) (|postTranList| x)))))
```

————-

6.2.72 defun postWith

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|with| '|postTran|) '|postWith|))
```

————-

6.2.73 defun postWith

[postTran p192]
[$insidePostCategoryIfTrue p??]

— **defun postWith** —

```
(defun |postWith| (arg)
 (let (|$insidePostCategoryIfTrue| a)
 (declare (special |$insidePostCategoryIfTrue|))
  (setq |$insidePostCategoryIfTrue| t)
  (setq a (|postTran| (second arg)))
  (cond
   ((and (pairp a) (member (qcar a) '(signature attribute if)))
     (list 'category a))
   ((and (pairp a) (eq (qcar a) 'progn))
     (cons 'category (qcdr a)))
   (t a))))
```

————-

6.3 Support routines

6.3.1 defun setDefOp

[$defOp p??]
[$topOp p??]

— **defun setDefOp** —

```
(defun |setDefOp| (f)
 (let (tmp1)
  (declare (special |$defOp| |$topOp|))
  (when (and (pairp f) (eq (qcar f) '|:|)
```

```
            (pairp (setq tmp1 (qcdr f))))
    (setq f (qcar tmp1)))
  (unless (atom f) (setq f (car f)))
  (if |$topOp|
   (setq |$defOp| f)
   (setq |$topOp| f))))
```

———-

6.3.2 defun aplTran

[aplTran1 p226]
[containsBang p229]
[$genno p??]
[$boot p??]

— **defun aplTran** —

```
(defun |aplTran| (x)
 (let ($genno u)
 (declare (special $genno $boot))
  (cond
   ($boot x)
   (t
    (setq $genno 0)
    (setq u (|aplTran1| x))
    (cond
     ((|containsBang| u) (|throwKeyedMsg| 's2ip0002 nil))
     (t u))))))
```

———-

6.3.3 defun aplTran1

[aplTranList p228]
[aplTran1 p226]
[hasAplExtension p228]
[nreverse0 p??]
[p??]
[$boot p??]

— **defun aplTran1** —

```
(defun |aplTran1| (x)
```

```
(let (op argl1 argl f y opprime yprime tmp1 arglAssoc futureArgl g)
(declare (special $boot))
 (if (atom x)
  x
  (progn
   (setq op (car x))
   (setq argl1 (cdr x))
   (setq argl (|aplTranList| argl1))
   (cond
    ((eq op '!)
     (cond
      ((and (pairp argl)
            (progn
             (setq f (qcar argl))
             (setq tmp1 (qcdr argl))
             (and (pairp tmp1)
                  (eq (qcdr tmp1) nil)
                  (progn
                   (setq y (qcar tmp1))
                   t))))
       (cond
        ((and (pairp y)
              (progn
               (setq opprime (qcar y))
               (setq yprime (qcdr y))
               t)
              (eq opprime '!))
         (|aplTran1| (cons op (cons op (cons f yprime)))))
        ($boot
         (cons 'collect
          (cons
           (list 'in (setq g (genvar)) (|aplTran1| y))
            (list (list f g ) ))))
        (t
         (list '|map| f (|aplTran1| y) ))))
       (t x)))
    ((progn
      (setq tmp1 (|hasAplExtension| argl))
      (and (pairp tmp1)
           (progn
            (setq arglAssoc (qcar tmp1))
            (setq futureArgl (qcdr tmp1))
            t)))
      (cons '|reshape|
       (cons
        (cons 'collect
         (append
          (do ((tmp3 arglAssoc (cdr tmp3)) (tmp4 nil))
              ((or (atom tmp3)
                   (progn (setq tmp4 (car tmp3)) nil)
```

```
                  (progn
                    (setq g (car tmp4))
                    (setq a (cdr tmp4))
                    nil))
                 (nreverse0 tmp2))
             (push (list 'in g (list '|ravel| a))) tmp2))
          (list (|aplTran1| (cons op futureArgl)))))
         (list (cdar arglAssoc))))
     (t (cons op argl)))))))
```

———-

6.3.4 defun aplTranList

[aplTran1 p226]
[aplTranList p228]

— **defun aplTranList** —

```
(defun |aplTranList| (x)
 (if (atom x)
  x
  (cons (|aplTran1| (car x)) (|aplTranList| (cdr x)))))
```

———-

6.3.5 defun hasAplExtension

[nreverse0 p**??**]
[deepestExpression p229]
[genvar p**??**]
[aplTran1 p226]
[msubst p**??**]

— **defun hasAplExtension** —

```
(defun |hasAplExtension| (argl)
 (let (tmp2 tmp3 y z g arglAssoc u)
  (when
   (dolist (x argl tmp2)
      (setq tmp2 (or tmp2 (and (pairp x) (eq (qcar x) '!))))))
    (setq u
     (dolist (x argl (nreverse0 tmp3))
      (push
        (if (and (pairp x) (eq (qcar x) '!)
```

```
              (pairp (qcdr x)) (eq (qcdr (qcdr x)) nil))
         (progn
           (setq y (qcar (qcdr x)))
           (setq z (|deepestExpression| y))
           (setq arglAssoc
            (cons (cons (setq g (genvar)) (|aplTran1| z)) arglAssoc))
           (msubst g z y))
          x)
         tmp3)))
    (cons arglAssoc u))))
```

———-

6.3.6 defun deepestExpression

[deepestExpression p229]

— **defun deepestExpression** —

```
(defun |deepestExpression| (x)
 (if (and (pairp x) (eq (qcar x) '!)
          (pairp (qcdr x)) (eq (qcdr (qcdr x)) nil))
   (|deepestExpression| (qcar (qcdr x)))
   x))
```

———-

6.3.7 defun containsBang

[containsBang p229]

— **defun containsBang** —

```
(defun |containsBang| (u)
 (let (tmp2)
  (cond
   ((atom u) (eq u '!))
   ((and (pairp u) (equal (qcar u) 'quote)
         (pairp (qcdr u)) (eq (qcdr (qcdr u)) nil))
     nil)
   (t
     (dolist (x u tmp2)
       (setq tmp2 (or tmp2 (|containsBang| x)))))))))
```

———-

6.3.8 defun getScriptName

[getScriptName identp (vol5)]
[postError p196]
[internl p??]
[decodeScripts p230]
[getScriptName pname (vol5)]

— **defun getScriptName** —

```
(defun |getScriptName| (op a numberOfFunctionalArgs)
 (when (null (identp op))
   (|postError| (list "   " op " cannot have scripts" )))
 (internl '* (princ-to-string numberOfFunctionalArgs)
             (|decodeScripts| a) (pname op)))
```

———-

6.3.9 defun decodeScripts

[qcar p??]
[qcdr p??]
[strconc p??]
[decodeScripts p230]

— **defun decodeScripts** —

```
(defun |decodeScripts| (a)
 (labels (
  (fn (a)
   (let ((tmp1 0))
    (if (and (pairp a) (eq (qcar a) '|,|))
     (dolist (x (qcdr a) tmp1) (setq tmp1 (+ tmp1 (fn x))))
     1))))
  (cond
   ((and (pairp a) (eq (qcar a) '|PrefixSC|)
         (pairp (qcdr a)) (eq (qcdr (qcdr a)) nil))
    (strconc (princ-to-string 0) (|decodeScripts| (qcar (qcdr a)))))
   ((and (pairp a) (eq (qcar a) '|;|))
    (apply 'strconc (loop for x in (qcdr a) collect (|decodeScripts| x))))
   ((and (pairp a) (eq (qcar a) '|,|))
    (princ-to-string (fn a)))
   (t
    (princ-to-string 1)))))
```

———-

Chapter 7

DEF forms

7.0.10 defvar $defstack

— **initvars** —

```
(defvar $defstack nil)
```

———-

7.0.11 defvar $is-spill

— **initvars** —

```
(defvar $is-spill nil)
```

———-

7.0.12 defvar $is-spill-list

— **initvars** —

```
(defvar $is-spill-list nil)
```

———-

7.0.13 defvar $vl

— **initvars** —

```
(defvar $vl nil)
```

————-

7.0.14 defvar $is-gensymlist

— **initvars** —

```
(defvar $is-gensymlist nil)
```

————-

7.0.15 defvar $initial-gensym

— **initvars** —

```
(defvar initial-gensym (list (gensym)))
```

————-

7.0.16 defvar $is-eqlist

— **initvars** —

```
(defvar $is-eqlist nil)
```

————-

7.0.17 defun hackforis

[hackforis1 p233]

— **defun hackforis** —

```
(defun hackforis (l) (mapcar #'hackforis1 L))
```

———-

7.0.18 defun hackforis1

[kar p??]
[eqcar p??]

— **defun hackforis1** —

```
(defun hackforis1 (x)
 (if (and (member (kar x) '(in on)) (eqcar (second x) 'is))
  (cons (first x) (cons (cons 'spadlet (cdadr x)) (cddr x)))
  x))
```

———-

7.0.19 defun unTuple

— **defun unTuple** —

```
(defun |unTuple| (x)
 (if (and (pairp x) (eq (qcar x) '|@Tuple|))
  (qcdr x)
  (list x)))
```

———-

7.0.20 defun errhuh

[systemError p??]

— **defun errhuh** —

```
(defun errhuh ()
 (|systemError| "problem with BOOT to LISP translation"))
```

———-

Chapter 8

PARSE forms

8.1 The original meta specification

This package provides routines to support the Metalanguage translator writing system. Metalanguage is described in META/LISP, R.D. Jenks, Tech Report, IBM T.J. Watson Research Center, 1969. Familiarity with this document is assumed.

Note that META/LISP and the meta parser/generator were removed from Axiom. This information is only for documentation purposes.

```
%       Scratchpad II Boot Language Grammar, Common Lisp Version
%       IBM Thomas J. Watson Research Center
%       Summer, 1986
%
%       NOTE: Substantially different from VM/LISP version, due to
%             different parser and attempt to render more within META proper.

.META(New NewExpr Process)
.PACKAGE 'BOOT'
.DECLARE(tmptok TOK ParseMode DEFINITION-NAME LABLASOC)
.PREFIX 'PARSE-'

NewExpr:        =')' .(processSynonyms) Command
              / .(SETQ DEFINITION-NAME (CURRENT-SYMBOL)) Statement ;

Command:        ')' SpecialKeyWord SpecialCommand +() ;

SpecialKeyWord: =(MATCH-CURRENT-TOKEN "IDENTIFIER)
                .(SETF (TOKEN-SYMBOL (CURRENT-TOKEN)) (unAbbreviateKeyword (CURRENT-SYMBOL))) ;

SpecialCommand: 'show' <'?' / Expression>! +(show #1) CommandTail
              / ?(MEMBER (CURRENT-SYMBOL) \$noParseCommands)
```

```
                .(FUNCALL (CURRENT-SYMBOL))
              / ?(MEMBER (CURRENT-SYMBOL) \$tokenCommands) TokenList
                  TokenCommandTail
              / PrimaryOrQM* CommandTail ;

TokenList:      (^?(isTokenDelimiter) +=(CURRENT-SYMBOL) .(ADVANCE-TOKEN))* ;

TokenCommandTail:
                <TokenOption*>! ?(atEndOfLine) +(#2 -#1) .(systemCommand #1) ;

TokenOption:    ')' TokenList ;

CommandTail:    <Option*>! ?(atEndOfLine) +(#2 -#1) .(systemCommand #1) ;

PrimaryOrQM:    '?' +\? / Primary ;

Option:         ')' PrimaryOrQM* ;

Statement:      Expr{0} <(',' Expr{0})* +(Series #2 -#1)>;

InfixWith:      With +(Join #2 #1) ;

With:           'with' Category +(with #1) ;

Category:       'if' Expression 'then' Category <'else' Category>! +(if #3 #2 #1)
              / '(' Category <(';' Category)*>! ')' +(CATEGORY #2 -#1)
              / .(SETQ $1 (LINE-NUMBER CURRENT-LINE)) Application
                   ( ':' Expression +(Signature #2 #1)
                              .(recordSignatureDocumentation ##1 $1)
                             / +(Attribute #1)
                               .(recordAttributeDocumentation ##1 $1));

Expression:   Expr{(PARSE-rightBindingPowerOf (MAKE-SYMBOL-OF PRIOR-TOKEN) ParseMode)}
                +#1 ;

Import:         'import' Expr{1000} <(',' Expr{1000})*>! +(import #2 -#1) ;

Infix:          =TRUE +=(CURRENT-SYMBOL) .(ADVANCE-TOKEN) <TokTail>
                Expression +(#2 #2 #1) ;

Prefix:         =TRUE +=(CURRENT-SYMBOL) .(ADVANCE-TOKEN) <TokTail>
                Expression +(#2 #1) ;

Suffix:         +=(CURRENT-SYMBOL) .(ADVANCE-TOKEN) <TokTail> +(#1 #1) ;

TokTail:        ?(AND (NULL \$BOOT) (EQ (CURRENT-SYMBOL) "\$)
                       (OR (ALPHA-CHAR-P (CURRENT-CHAR))
                           (CHAR-EQ (CURRENT-CHAR) '$')
                           (CHAR-EQ (CURRENT-CHAR) '\%')
                           (CHAR-EQ (CURRENT-CHAR) '(')))
```

```
                .(SETQ $1 (COPY-TOKEN PRIOR-TOKEN)) Qualification
                .(SETQ PRIOR-TOKEN $1) ;

Qualification:  '$' Primary1 +=(dollarTran #1 #1) ;

SemiColon:      ';' (Expr{82} / + \/throwAway) +(\; #2 #1) ;

Return:         'return' Expression +(return #1) ;

Exit:           'exit' (Expression / +\$NoValue) +(exit #1) ;

Leave:          'leave' ( Expression / +\$NoValue )
                ('from' Label +(leaveFrom #1 #1) / +(leave #1)) ;

Seg:            GliphTok{"\.\.} <Expression>! +(SEGMENT #2 #1) ;

Conditional:    'if' Expression 'then' Expression <'else' ElseClause>!
                   +(if #3 #2 #1) ;

ElseClause:     ?(EQ (CURRENT-SYMBOL) "if) Conditional / Expression ;

Loop:           Iterator* 'repeat' Expr{110} +(REPEAT -#2 #1)
              / 'repeat' Expr{110} +(REPEAT #1) ;

Iterator:       'for' Primary 'in' Expression
                ( 'by' Expr{200} +(INBY #3 #2 #1) / +(IN #2 #1) )
                < '\|' Expr{111} +(\| #1) >
              / 'while' Expr{190} +(WHILE #1)
              / 'until' Expr{190} +(UNTIL #1) ;

Expr{RBP}:      NudPart{RBP} <LedPart{RBP}>* +#1;

LabelExpr:      Label Expr{120} +(LABEL #2 #1) ;

Label:          '@<<' Name '>>' ;

LedPart{RBP}:   Operation{"Led RBP} +#1;

NudPart{RBP}:   (Operation{"Nud RBP} / Reduction / Form) +#1 ;

Operation{ParseMode RBP}:
        ^?(MATCH-CURRENT-TOKEN "IDENTIFIER)
        ?(GETL (SETQ tmptok (CURRENT-SYMBOL)) ParseMode)
        ?(LT RBP (PARSE-leftBindingPowerOf tmptok ParseMode))
        .(SETQ RBP (PARSE-rightBindingPowerOf tmptok ParseMode))
        getSemanticForm{tmptok ParseMode (ELEMN (GETL tmptok ParseMode) 5 NIL)} ;

% Binding powers stored under the Led and Red properties of an operator
% are set up by the file BOTTOMUP.LISP.  The format for a Led property
% is <Operator Left-Power Right-Power>, and the same for a Nud, except that
```

```
% it may also have a fourth component <Special-Handler>. ELEMN attempts to
% get the Nth indicator, counting from 1.

leftBindingPowerOf{X IND}: =(LET ((Y (GETL X IND))) (IF Y (ELEMN Y 3 0) 0)) ;

rightBindingPowerOf{X IND}: =(LET ((Y (GETL X IND))) (IF Y (ELEMN Y 4 105) 105)) ;

getSemanticForm{X IND Y}:
               ?(AND Y (EVAL Y)) / ?(EQ IND "Nud) Prefix / ?(EQ IND "Led) Infix ;


Reduction:      ReductionOp Expr{1000} +(Reduce #2 #1) ;

ReductionOp:    ?(AND (GETL (CURRENT-SYMBOL) "Led)
                      (MATCH-NEXT-TOKEN "SPECIAL-CHAR (CODE-CHAR 47))) % Forgive me!
                +=(CURRENT-SYMBOL) .(ADVANCE-TOKEN) .(ADVANCE-TOKEN) ;

Form:           'iterate' < 'from' Label +(#1) >! +(iterate -#1)
              / 'yield' Application +(yield #1)
              / Application ;

Application: Primary <Selector>* <Application +(#2 #1)>;

Selector: ?NONBLANK ?(EQ (CURRENT-SYMBOL) "\.) ?(CHAR-NE (CURRENT-CHAR) "\ )
                '.' PrimaryNoFloat (=\$BOOT +(ELT #2 #1)/ +(#2 #1))
          / (Float /'.' Primary) (=\$BOOT +(ELT #2 #1)/ +(#2 #1));

PrimaryNoFloat: Primary1 <TokTail> ;

Primary: Float /PrimaryNoFloat ;

Primary1: VarForm <=(AND NONBLANK (EQ (CURRENT-SYMBOL) "\()) Primary1 +(#2 #1)>
        /Quad
        /String
        /IntegerTok
        /FormalParameter
        /='\'' (?\$BOOT Data / '\'' Expr{999} +(QUOTE #1))
        /Sequence
        /Enclosure ;

Float: FloatBase (?NONBLANK FloatExponent / +0) +=(MAKE-FLOAT #4 #2 #2 #1) ;

FloatBase: ?(FIXP (CURRENT-SYMBOL)) ?(CHAR-EQ (CURRENT-CHAR) '.')
           ?(CHAR-NE (NEXT-CHAR) '.')
              IntegerTok FloatBasePart
          /?(FIXP (CURRENT-SYMBOL)) ?(CHAR-EQ (CHAR-UPCASE (CURRENT-CHAR)) "E)
              IntegerTok +0 +0
         /?(DIGITP (CURRENT-CHAR)) ?(EQ (CURRENT-SYMBOL) "\.)
              +0 FloatBasePart ;
```

```
FloatBasePart: '.'
  (?(DIGITP (CURRENT-CHAR)) +=(TOKEN-NONBLANK (CURRENT-TOKEN)) IntegerTok
  / +0 +0);


FloatExponent: =(AND (MEMBER (CURRENT-SYMBOL) "(E e))
                     (FIND (CURRENT-CHAR) '+-'))
                 .(ADVANCE-TOKEN)
        (IntegerTok/'+' IntegerTok/'-' IntegerTok +=(MINUS #1)/+0)
       /?(IDENTP (CURRENT-SYMBOL)) =(SETQ $1 (FLOATEXPID (CURRENT-SYMBOL)))
       .(ADVANCE-TOKEN) +=$1 ;

Enclosure:      '(' ( Expr{6} ')' / ')' +(Tuple) )
              / '{' ( Expr{6} '}' +(brace (construct #1)) / '}' +(brace)) ;

IntegerTok:     NUMBER ;

FloatTok:       NUMBER +=(IF \$BOOT #1 (BFP- #1)) ;

FormalParameter: FormalParameterTok ;

FormalParameterTok: ARGUMENT-DESIGNATOR ;

Quad:           '$' +\$ / ?\$BOOT GliphTok{"\.} +\. ;

String:         SPADSTRING ;

VarForm:        Name <Scripts +(Scripts #2 #1) > +#1 ;

Scripts:        ?NONBLANK '[' ScriptItem ']' ;

ScriptItem:     Expr{90} <(';' ScriptItem)* +(\; #2 -#1)>
              / ';' ScriptItem +(PrefixSC #1) ;

Name:           IDENTIFIER +#1 ;

Data:           .(SETQ LABLASOC NIL) Sexpr +(QUOTE =(TRANSLABEL #1 LABLASOC)) ;

Sexpr:          .(ADVANCE-TOKEN) Sexpr1 ;

Sexpr1:       AnyId
              < NBGliphTok{"\=} Sexpr1
                 .(SETQ LABLASOC (CONS (CONS #2 ##1) LABLASOC))>
              / '\'' Sexpr1 +(QUOTE #1)
              / IntegerTok
              / '-' IntegerTok +=(MINUS #1)
              / String
              / '<' <Sexpr1*>! '>' +=(LIST2VEC #1)
              / '(' <Sexpr1* <GliphTok{"\.} Sexpr1 +=(NCONC #2 #1)>>! ')' ;
```

```
NBGliphTok{tok}:   ?(AND (MATCH-CURRENT-TOKEN "GLIPH tok) NONBLANK)
                    .(ADVANCE-TOKEN) ;

GliphTok{tok}:     ?(MATCH-CURRENT-TOKEN "GLIPH tok) .(ADVANCE-TOKEN) ;

AnyId:           IDENTIFIER
               / (='$' +=(CURRENT-SYMBOL) .(ADVANCE-TOKEN) / KEYWORD) ;

Sequence:        OpenBracket Sequence1 ']'
               / OpenBrace Sequence1 '}' +(brace #1) ;

Sequence1:     (Expression +(#2 #1) / +(#1)) <IteratorTail +(COLLECT -#1 #1)>  ;

OpenBracket:    =(EQ (getToken (SETQ $1 (CURRENT-SYMBOL))) "\[ )
                      (=(EQCAR $1 "elt) +(elt =(CADR $1) construct)
                        / +construct) .(ADVANCE-TOKEN) ;

OpenBrace:      =(EQ (getToken (SETQ $1 (CURRENT-SYMBOL))) "\{ )
                      (=(EQCAR $1 "elt) +(elt =(CADR $1) brace)
                        / +construct) .(ADVANCE-TOKEN) ;

IteratorTail:   ('repeat' <Iterator*>! / Iterator*) ;

.FIN ;
```

8.2 The PARSE code

8.2.1 defvar $tmptok

— **initvars** —

```
(defvar |tmptok| nil)
```

————-

8.2.2 defvar $tok

— **initvars** —

```
(defvar tok nil)
```

————-

8.2.3 defvar $ParseMode

— **initvars** —

```
(defvar |ParseMode| nil)
```

————-

8.2.4 defvar $definition-name

— **initvars** —

```
(defvar definition-name nil)
```

————-

8.2.5 defvar $lablasoc

— **initvars** —

```
(defvar lablasoc nil)
```

————-

8.2.6 defun PARSE-NewExpr

[match-string p278]
[action p291]
[PARSE-NewExpr processSynonyms (vol5)]
[must p290]
[current-symbol p284]
[PARSE-Statement p246]
[definition-name p241]

— **defun PARSE-NewExpr** —

```
(defun |PARSE-NewExpr| ()
 (or (and (match-string ")") (action (|processSynonyms|))
          (must (|PARSE-Command|)))
```

```
     (and (action (setq definition-name (current-symbol)))
          (|PARSE-Statement|))))
```

———-

8.2.7 defun PARSE-Command

[match-advance-string p279]
[must p290]
[PARSE-SpecialKeyWord p242]
[PARSE-SpecialCommand p243]
[push-reduction p292]

— **defun PARSE-Command** —

```
(defun |PARSE-Command| ()
 (and (match-advance-string ")") (must (|PARSE-SpecialKeyWord|))
      (must (|PARSE-SpecialCommand|))
      (push-reduction '|PARSE-Command| nil)))
```

———-

8.2.8 defun PARSE-SpecialKeyWord

[match-current-token p283]
[action p291]
[token-symbol p**??**]
[current-token p285]
[PARSE-SpecialKeyWord unAbbreviateKeyword (vol5)]
[current-symbol p284]

— **defun PARSE-SpecialKeyWord** —

```
(defun |PARSE-SpecialKeyWord| ()
 (and (match-current-token 'identifier)
      (action (setf (token-symbol (current-token))
                    (|unAbbreviateKeyword| (current-symbol))))))
```

———-

8.2.9 defun PARSE-SpecialCommand

[match-advance-string p279]
[bang p??]
[optional p291]
[PARSE-Expression p249]
[push-reduction p292]
[PARSE-SpecialCommand p243]
[pop-stack-1 p302]
[PARSE-CommandTail p245]
[must p290]
[current-symbol p284]
[action p291]
[PARSE-TokenList p244]
[PARSE-TokenCommandTail p243]
[star p291]
[PARSE-PrimaryOrQM p245]
[PARSE-CommandTail p245]
[$noParseCommands p??]
[$tokenCommands p??]

— **defun PARSE-SpecialCommand** —

```
(defun |PARSE-SpecialCommand| ()
 (declare (special $noParseCommands $tokenCommands))
 (or (and (match-advance-string "show")
          (bang fil_test
                (optional
                    (or (match-advance-string "?")
                        (|PARSE-Expression|))))
          (push-reduction '|PARSE-SpecialCommand|
              (list '|show| (pop-stack-1)))
          (must (|PARSE-CommandTail|)))
     (and (member (current-symbol) |$noParseCommands|)
          (action (funcall (current-symbol))))
     (and (member (current-symbol) |$tokenCommands|)
          (|PARSE-TokenList|) (must (|PARSE-TokenCommandTail|)))
     (and (star repeator (|PARSE-PrimaryOrQM|))
          (must (|PARSE-CommandTail|)))))
```

———-

8.2.10 defun PARSE-TokenCommandTail

[bang p??]
[optional p291]

[star p291]
[PARSE-TokenOption p244]
[atEndOfLine p??]
[push-reduction p292]
[PARSE-TokenCommandTail p243]
[pop-stack-2 p303]
[pop-stack-1 p302]
[action p291]
[PARSE-TokenCommandTail systemCommand (vol5)]

— **defun PARSE-TokenCommandTail** —

```
(defun |PARSE-TokenCommandTail| ()
 (and (bang fil_test (optional (star repeator (|PARSE-TokenOption|))))
      (|atEndOfLine|)
      (push-reduction '|PARSE-TokenCommandTail|
          (cons (pop-stack-2) (append (pop-stack-1) nil)))
      (action (|systemCommand| (pop-stack-1)))))
```

———-

8.2.11 defun PARSE-TokenOption

[match-advance-string p279]
[must p290]
[PARSE-TokenList p244]

— **defun PARSE-TokenOption** —

```
(defun |PARSE-TokenOption| ()
 (and (match-advance-string ")") (must (|PARSE-TokenList|))))
```

———-

8.2.12 defun PARSE-TokenList

[star p291]
[isTokenDelimiter p281]
[push-reduction p292]
[current-symbol p284]
[action p291]
[advance-token p286]

— **defun PARSE-TokenList** —

```
(defun |PARSE-TokenList| ()
 (star repeator
  (and (not (|isTokenDelimiter|))
       (push-reduction '|PARSE-TokenList| (current-symbol))
       (action (advance-token)))))
```

———-

8.2.13 defun PARSE-CommandTail

[bang p??]
[optional p291]
[star p291]
[push-reduction p292]
[PARSE-Option p246]
[PARSE-CommandTail p245]
[pop-stack-2 p303]
[pop-stack-1 p302]
[action p291]
[PARSE-CommandTail systemCommand (vol5)]

— **defun PARSE-CommandTail** —

```
(defun |PARSE-CommandTail| ()
 (and (bang fil_test (optional (star repeator (|PARSE-Option|))))
      (|atEndOfLine|)
      (push-reduction '|PARSE-CommandTail|
          (cons (pop-stack-2) (append (pop-stack-1) nil)))
      (action (|systemCommand| (pop-stack-1)))))
```

———-

8.2.14 defun PARSE-PrimaryOrQM

[match-advance-string p279]
[push-reduction p292]
[PARSE-PrimaryOrQM p245]
[PARSE-Primary p258]

— **defun PARSE-PrimaryOrQM** —

```
(defun |PARSE-PrimaryOrQM| ()
 (or (and (match-advance-string "?")
          (push-reduction '|PARSE-PrimaryOrQM| '?))
```

```
     (|PARSE-Primary|)))
```

———-

8.2.15 defun PARSE-Option

[match-advance-string p279]
[must p290]
[star p291]
[PARSE-PrimaryOrQM p245]

— **defun PARSE-Option** —

```
(defun |PARSE-Option| ()
 (and (match-advance-string ")")
      (must (star repeator (|PARSE-PrimaryOrQM|)))))
```

———-

8.2.16 defun PARSE-Statement

[PARSE-Expr p250]
[optional p291]
[star p291]
[match-advance-string p279]
[must p290]
[push-reduction p292]
[pop-stack-2 p303]
[pop-stack-1 p302]

— **defun PARSE-Statement** —

```
(defun |PARSE-Statement| ()
  (and (|PARSE-Expr| 0)
       (optional
           (and (star repeator
                      (and (match-advance-string ",")
                           (must (|PARSE-Expr| 0))))
                (push-reduction '|PARSE-Statement|
                    (cons '|Series|
                          (cons (pop-stack-2)
                                (append (pop-stack-1) nil))))))))
```

———-

8.2.17 defun PARSE-InfixWith

[PARSE-With p247]
[push-reduction p292]
[pop-stack-2 p303]
[pop-stack-1 p302]

— **defun PARSE-InfixWith** —

```
(defun |PARSE-InfixWith| ()
  (and (|PARSE-With|)
       (push-reduction '|PARSE-InfixWith|
           (list '|Join| (pop-stack-2) (pop-stack-1)))))
```

————-

8.2.18 defun PARSE-With

[match-advance-string p279]
[must p290]
[push-reduction p292]
[pop-stack-1 p302]

— **defun PARSE-With** —

```
(defun |PARSE-With| ()
  (and (match-advance-string "with") (must (|PARSE-Category|))
       (push-reduction '|PARSE-With|
           (cons '|with| (cons (pop-stack-1) nil)))))
```

————-

8.2.19 defun PARSE-Category

[match-advance-string p279]
[must p290]
[bang p**??**]
[optional p291]
[push-reduction p292]
[PARSE-Expression p249]
[PARSE-Category p247]
[pop-stack-3 p303]
[pop-stack-2 p303]
[pop-stack-1 p302]

[star p291]
[line-number p??]
[PARSE-Application p256]
[action p291]
[recordSignatureDocumentation p??]
[nth-stack p304]
[recordAttributeDocumentation p??]
[current-line p92]

— defun PARSE-Category —

```
(defun |PARSE-Category| ()
 (let (g1)
  (or (and (match-advance-string "if") (must (|PARSE-Expression|))
           (must (match-advance-string "then"))
           (must (|PARSE-Category|))
           (bang fil_test
                 (optional
                     (and (match-advance-string "else")
                          (must (|PARSE-Category|)))))
           (push-reduction '|PARSE-Category|
             (list '|if| (pop-stack-3) (pop-stack-2) (pop-stack-1))))
      (and (match-advance-string "(") (must (|PARSE-Category|))
           (bang fil_test
                 (optional
                     (star repeator
                           (and (match-advance-string ";")
                                (must (|PARSE-Category|))))))
           (must (match-advance-string ")"))
           (push-reduction '|PARSE-Category|
               (cons 'category
                     (cons (pop-stack-2)
                           (append (pop-stack-1) nil)))))
      (and (action (setq g1 (line-number current-line)))
           (|PARSE-Application|)
           (must (or (and (match-advance-string ":")
                          (must (|PARSE-Expression|))
                          (push-reduction '|PARSE-Category|
                           (list '|Signature| (pop-stack-2) (pop-stack-1) ))
                          (action (|recordSignatureDocumentation|
                                   (nth-stack 1) g1)))
                     (and (push-reduction '|PARSE-Category|
                            (list '|Attribute| (pop-stack-1) ))
                          (action (|recordAttributeDocumentation|
                                   (nth-stack 1) g1)))))))))
```

———-

8.2.20 defun PARSE-Expression

[PARSE-Expr p250]
[PARSE-rightBindingPowerOf p252]
[make-symbol-of p284]
[push-reduction p292]
[pop-stack-1 p302]
[ParseMode p241]
[prior-token p99]

— **defun PARSE-Expression** —

```
(defun |PARSE-Expression| ()
 (declare (special prior-token))
 (and (|PARSE-Expr|
          (|PARSE-rightBindingPowerOf| (make-symbol-of prior-token)
              |ParseMode|))
      (push-reduction '|PARSE-Expression| (pop-stack-1))))
```

———-

8.2.21 defun PARSE-Import

[match-advance-string p279]
[must p290]
[PARSE-Expr p250]
[bang p??]
[optional p291]
[star p291]
[push-reduction p292]
[pop-stack-2 p303]
[pop-stack-1 p302]

— **defun PARSE-Import** —

```
(defun |PARSE-Import| ()
  (and (match-advance-string "import") (must (|PARSE-Expr| 1000))
       (bang fil_test
             (optional
                 (star repeator
                       (and (match-advance-string ",")
                            (must (|PARSE-Expr| 1000))))))
       (push-reduction '|PARSE-Import|
           (cons '|import|
                 (cons (pop-stack-2) (append (pop-stack-1) nil))))))
```

———-

8.2.22 defun PARSE-Expr

[PARSE-NudPart p250]
[PARSE-LedPart p250]
[optional p291]
[star p291]
[push-reduction p292]
[pop-stack-1 p302]

— **defun PARSE-Expr** —

```
(defun |PARSE-Expr| (rbp)
 (declare (special rbp))
 (and (|PARSE-NudPart| rbp)
      (optional (star opt_expr (|PARSE-LedPart| rbp)))
      (push-reduction '|PARSE-Expr| (pop-stack-1))))
```

———-

8.2.23 defun PARSE-LedPart

[PARSE-Operation p251]
[push-reduction p292]
[pop-stack-1 p302]

— **defun PARSE-LedPart** —

```
(defun |PARSE-LedPart| (rbp)
  (declare (special rbp))
  (and (|PARSE-Operation| '|Led| rbp)
       (push-reduction '|PARSE-LedPart| (pop-stack-1))))
```

———-

8.2.24 defun PARSE-NudPart

[PARSE-Operation p251]
[PARSE-Reduction p255]
[PARSE-Form p255]
[push-reduction p292]
[pop-stack-1 p302]

[rbp p??]

— **defun PARSE-NudPart** —

```
(defun |PARSE-NudPart| (rbp)
  (declare (special rbp))
  (and (or (|PARSE-Operation| '|Nud| rbp) (|PARSE-Reduction|)
           (|PARSE-Form|))
       (push-reduction '|PARSE-NudPart| (pop-stack-1))))
```

———-

8.2.25 defun PARSE-Operation

[match-current-token p283]
[current-symbol p284]
[PARSE-leftBindingPowerOf p251]
[lt p??]
[getl p??]
[action p291]
[PARSE-rightBindingPowerOf p252]
[PARSE-getSemanticForm p252]
[elemn p??]
[ParseMode p241]
[rbp p??]
[tmptok p240]

— **defun PARSE-Operation** —

```
(defun |PARSE-Operation| (|ParseMode| rbp)
 (declare (special |ParseMode| rbp |tmptok|))
 (and (not (match-current-token 'identifier))
      (getl (setq |tmptok| (current-symbol)) |ParseMode|)
      (lt rbp (|PARSE-leftBindingPowerOf| |tmptok| |ParseMode|))
      (action (setq rbp (|PARSE-rightBindingPowerOf| |tmptok| |ParseMode|)))
      (|PARSE-getSemanticForm| |tmptok| |ParseMode|
          (elemn (getl |tmptok| |ParseMode|) 5 nil))))
```

———-

8.2.26 defun PARSE-leftBindingPowerOf

[getl p??]
[elemn p??]

— **defun PARSE-leftBindingPowerOf** —

```
(defun |PARSE-leftBindingPowerOf| (x ind)
 (declare (special x ind))
 (let ((y (getl x ind))) (if y (elemn y 3 0) 0)))
```

———-

8.2.27 defun PARSE-rightBindingPowerOf

[getl p**??**]
[elemn p**??**]

— **defun PARSE-rightBindingPowerOf** —

```
(defun |PARSE-rightBindingPowerOf| (x ind)
  (declare (special x ind))
  (let ((y (getl x ind))) (if y (elemn y 4 105) 105)))
```

———-

8.2.28 defun PARSE-getSemanticForm

[PARSE-Prefix p252]
[PARSE-Infix p253]

— **defun PARSE-getSemanticForm** —

```
(defun |PARSE-getSemanticForm| (x ind y)
  (declare (special x ind y))
  (or (and y (eval y)) (and (eq ind '|Nud|) (|PARSE-Prefix|))
      (and (eq ind '|Led|) (|PARSE-Infix|))))
```

———-

8.2.29 defun PARSE-Prefix

[push-reduction p292]
[current-symbol p284]
[action p291]
[advance-token p286]

[optional p291]
[PARSE-TokTail p254]
[must p290]
[PARSE-Expression p249]
[push-reduction p292]
[pop-stack-2 p303]
[pop-stack-1 p302]

— **defun PARSE-Prefix** —

```
(defun |PARSE-Prefix| ()
  (and (push-reduction '|PARSE-Prefix| (current-symbol))
       (action (advance-token)) (optional (|PARSE-TokTail|))
       (must (|PARSE-Expression|))
       (push-reduction '|PARSE-Prefix|
           (list (pop-stack-2) (pop-stack-1)))))
```

————-

8.2.30 defun PARSE-Infix

[push-reduction p292]
[current-symbol p284]
[action p291]
[advance-token p286]
[optional p291]
[PARSE-TokTail p254]
[must p290]
[PARSE-Expression p249]
[pop-stack-2 p303]
[pop-stack-1 p302]

— **defun PARSE-Infix** —

```
(defun |PARSE-Infix| ()
  (and (push-reduction '|PARSE-Infix| (current-symbol))
       (action (advance-token)) (optional (|PARSE-TokTail|))
       (must (|PARSE-Expression|))
       (push-reduction '|PARSE-Infix|
           (list (pop-stack-2) (pop-stack-2) (pop-stack-1) ))))
```

————-

8.2.31 defun PARSE-TokTail

[current-symbol p284]
[current-char p287]
[char-eq p288]
[copy-token p??]
[action p291]
[PARSE-Qualification p254]
[$boot p??]

— defun PARSE-TokTail —

```
(defun |PARSE-TokTail| ()
 (let (g1)
  (and (null $boot) (eq (current-symbol) '$)
       (or (alpha-char-p (current-char))
           (char-eq (current-char) "$")
           (char-eq (current-char) "%")
           (char-eq (current-char) "("))
       (action (setq g1 (copy-token prior-token)))
       (|PARSE-Qualification|) (action (setq prior-token g1)))))
```

———-

8.2.32 defun PARSE-Qualification

[match-advance-string p279]
[must p290]
[PARSE-Primary1 p258]
[push-reduction p292]
[dollarTran p289]
[pop-stack-1 p302]

— defun PARSE-Qualification —

```
(defun |PARSE-Qualification| ()
  (and (match-advance-string "$") (must (|PARSE-Primary1|))
       (push-reduction '|PARSE-Qualification|
           (|dollarTran| (pop-stack-1) (pop-stack-1)))))
```

———-

8.2.33 defun PARSE-Reduction

[PARSE-ReductionOp p255]
[must p290]
[PARSE-Expr p250]
[push-reduction p292]
[pop-stack-2 p303]
[pop-stack-1 p302]

— **defun PARSE-Reduction** —

```
(defun |PARSE-Reduction| ()
 (and (|PARSE-ReductionOp|) (must (|PARSE-Expr| 1000))
      (push-reduction '|PARSE-Reduction|
          (list '|Reduce| (pop-stack-2) (pop-stack-1) ))))
```

————-

8.2.34 defun PARSE-ReductionOp

[getl p**??**]
[current-symbol p284]
[match-next-token p284]
[action p291]
[advance-token p286]

— **defun PARSE-ReductionOp** —

```
(defun |PARSE-ReductionOp| ()
  (and (getl (current-symbol) '|Led|)
       (match-next-token 'special-char (code-char 47))
       (push-reduction '|PARSE-ReductionOp| (current-symbol))
       (action (advance-token)) (action (advance-token))))
```

————-

8.2.35 defun PARSE-Form

[match-advance-string p279]
[bang p**??**]
[optional p291]
[must p290]
[push-reduction p292]
[pop-stack-1 p302]

[PARSE-Application p256]

— **defun PARSE-Form** —

```
(defun |PARSE-Form| ()
  (or (and (match-advance-string "iterate")
           (bang fil_test
                 (optional
                     (and (match-advance-string "from")
                          (must (|PARSE-Label|))
                          (push-reduction '|PARSE-Form|
                             (list (pop-stack-1))))))
           (push-reduction '|PARSE-Form|
               (cons '|iterate| (append (pop-stack-1) nil))))
      (and (match-advance-string "yield") (must (|PARSE-Application|))
           (push-reduction '|PARSE-Form|
               (list '|yield| (pop-stack-1))))
      (|PARSE-Application|)))
```

———-

8.2.36 defun PARSE-Application

[PARSE-Primary p258]
[optional p291]
[star p291]
[PARSE-Selector p257]
[PARSE-Application p256]
[push-reduction p292]
[pop-stack-2 p303]
[pop-stack-1 p302]

— **defun PARSE-Application** —

```
(defun |PARSE-Application| ()
 (and (|PARSE-Primary|) (optional (star opt_expr (|PARSE-Selector|)))
      (optional
          (and (|PARSE-Application|)
               (push-reduction '|PARSE-Application|
                   (list (pop-stack-2) (pop-stack-1)))))))
```

———-

8.2.37 defun PARSE-Label

[match-advance-string p279]
[must p290]
[PARSE-Name p265]

— **defun PARSE-Label** —

```
(defun |PARSE-Label| ()
  (and (match-advance-string "<<") (must (|PARSE-Name|))
       (must (match-advance-string ">>"))))
```

————-

8.2.38 defun PARSE-Selector

[current-symbol p284]
[char-ne p288]
[current-char p287]
[match-advance-string p279]
[must p290]
[PARSE-PrimaryNoFloat p258]
[push-reduction p292]
[pop-stack-2 p303]
[pop-stack-1 p302]
[PARSE-Float p259]
[PARSE-Primary p258]
[$boot p**??**]

— **defun PARSE-Selector** —

```
(defun |PARSE-Selector| ()
  (declare (special $boot))
  (or (and nonblank (eq (current-symbol) '|.|)
           (char-ne (current-char) '| |) (match-advance-string ".")
           (must (|PARSE-PrimaryNoFloat|))
           (must (or (and $boot
                          (push-reduction '|PARSE-Selector|
                              (list 'elt (pop-stack-2) (pop-stack-1))))
                     (push-reduction '|PARSE-Selector|
                         (list (pop-stack-2) (pop-stack-1))))))
      (and (or (|PARSE-Float|)
               (and (match-advance-string ".")
                    (must (|PARSE-Primary|))))
           (must (or (and $boot
                          (push-reduction '|PARSE-Selector|
```

```
                (list 'elt (pop-stack-2) (pop-stack-1))))
        (push-reduction '|PARSE-Selector|
            (list (pop-stack-2) (pop-stack-1))))))))
```

———-

8.2.39 defun PARSE-PrimaryNoFloat

[PARSE-Primary1 p258]
[optional p291]
[PARSE-TokTail p254]

— **defun PARSE-PrimaryNoFloat** —

```
(defun |PARSE-PrimaryNoFloat| ()
  (and (|PARSE-Primary1|) (optional (|PARSE-TokTail|))))
```

———-

8.2.40 defun PARSE-Primary

[PARSE-Float p259]
[PARSE-PrimaryNoFloat p258]

— **defun PARSE-Primary** —

```
(defun |PARSE-Primary| ()
  (or (|PARSE-Float|) (|PARSE-PrimaryNoFloat|)))
```

———-

8.2.41 defun PARSE-Primary1

[PARSE-VarForm p264]
[optional p291]
[current-symbol p284]
[PARSE-Primary1 p258]
[must p290]
[pop-stack-2 p303]
[pop-stack-1 p302]
[push-reduction p292]
[PARSE-Quad p263]

[PARSE-String p263]
[PARSE-IntegerTok p262]
[PARSE-FormalParameter p263]
[match-string p278]
[PARSE-Data p266]
[match-advance-string p279]
[PARSE-Expr p250]
[PARSE-Sequence p269]
[PARSE-Enclosure p262]
[$boot p??]

— **defun PARSE-Primary1** —

```
(defun |PARSE-Primary1| ()
  (or (and (|PARSE-VarForm|)
           (optional
               (and nonblank (eq (current-symbol) '|(|)
                    (must (|PARSE-Primary1|))
                    (push-reduction '|PARSE-Primary1|
                        (list (pop-stack-2) (pop-stack-1))))))
      (|PARSE-Quad|) (|PARSE-String|) (|PARSE-IntegerTok|)
      (|PARSE-FormalParameter|)
      (and (match-string "'")
           (must (or (and $boot (|PARSE-Data|))
                     (and (match-advance-string "'")
                          (must (|PARSE-Expr| 999))
                          (push-reduction '|PARSE-Primary1|
                              (list 'quote (pop-stack-1)))))))
      (|PARSE-Sequence|) (|PARSE-Enclosure|))))
```

———-

8.2.42 defun PARSE-Float

[PARSE-FloatBase p260]
[must p290]
[PARSE-FloatExponent p261]
[push-reduction p292]
[make-float p??]
[pop-stack-4 p303]
[pop-stack-3 p303]
[pop-stack-2 p303]
[pop-stack-1 p302]

— **defun PARSE-Float** —

```
(defun |PARSE-Float| ()
  (and (|PARSE-FloatBase|)
       (must (or (and nonblank (|PARSE-FloatExponent|))
                 (push-reduction '|PARSE-Float| 0)))
       (push-reduction '|PARSE-Float|
           (make-float (pop-stack-4) (pop-stack-2) (pop-stack-2)
               (pop-stack-1)))))
```

———-

8.2.43 defun PARSE-FloatBase

[current-symbol p284]
[char-eq p288]
[current-char p287]
[char-ne p288]
[next-char p287]
[PARSE-IntegerTok p262]
[must p290]
[PARSE-FloatBasePart p260]
[PARSE-IntegerTok p262]
[push-reduction p292]
[PARSE-FloatBase digitp (vol5)]

— defun PARSE-FloatBase —

```
(defun |PARSE-FloatBase| ()
  (or (and (integerp (current-symbol)) (char-eq (current-char) ".")
           (char-ne (next-char) ".") (|PARSE-IntegerTok|)
           (must (|PARSE-FloatBasePart|)))
      (and (integerp (current-symbol))
           (char-eq (char-upcase (current-char)) 'e)
           (|PARSE-IntegerTok|) (push-reduction '|PARSE-FloatBase| 0)
           (push-reduction '|PARSE-FloatBase| 0))
      (and (digitp (current-char)) (eq (current-symbol) '|.|)
           (push-reduction '|PARSE-FloatBase| 0)
           (|PARSE-FloatBasePart|))))
```

———-

8.2.44 defun PARSE-FloatBasePart

[match-advance-string p279]
[must p290]

[PARSE-FloatBasePart digitp (vol5)]
[current-char p287]
[push-reduction p292]
[token-nonblank p**??**]
[current-token p285]
[PARSE-IntegerTok p262]

— **defun PARSE-FloatBasePart** —

```
(defun |PARSE-FloatBasePart| ()
  (and (match-advance-string ".")
       (must (or (and (digitp (current-char))
                      (push-reduction '|PARSE-FloatBasePart|
                          (token-nonblank (current-token)))
                      (|PARSE-IntegerTok|))
                 (and (push-reduction '|PARSE-FloatBasePart| 0)
                      (push-reduction '|PARSE-FloatBasePart| 0))))))
```

————-

8.2.45 defun PARSE-FloatExponent

[current-symbol p284]
[current-char p287]
[action p291]
[advance-token p286]
[PARSE-IntegerTok p262]
[match-advance-string p279]
[must p290]
[push-reduction p292]
[PARSE-FloatExponent identp (vol5)]
[floatexpid p289]

— **defun PARSE-FloatExponent** —

```
(defun |PARSE-FloatExponent| ()
 (let (g1)
  (or (and (member (current-symbol) '(e |e|))
           (find (current-char) "+-") (action (advance-token))
           (must (or (|PARSE-IntegerTok|)
                     (and (match-advance-string "+")
                          (must (|PARSE-IntegerTok|)))
                     (and (match-advance-string "-")
                          (must (|PARSE-IntegerTok|))
                          (push-reduction '|PARSE-FloatExponent|
                              (- (pop-stack-1))))
```

```
                    (push-reduction '|PARSE-FloatExponent| 0))))
      (and (identp (current-symbol))
           (setq g1 (floatexpid (current-symbol)))
           (action (advance-token))
           (push-reduction '|PARSE-FloatExponent| g1)))))
```

———-

8.2.46 defun PARSE-Enclosure

[match-advance-string p279]
[must p290]
[PARSE-Expr p250]
[push-reduction p292]
[pop-stack-1 p302]

— defun PARSE-Enclosure —

```
(defun |PARSE-Enclosure| ()
  (or (and (match-advance-string "(")
           (must (or (and (|PARSE-Expr| 6)
                          (must (match-advance-string ")")))
                     (and (match-advance-string ")")
                          (push-reduction '|PARSE-Enclosure|
                              (list '|@Tuple|)))))))
      (and (match-advance-string "{")
           (must (or (and (|PARSE-Expr| 6)
                          (must (match-advance-string "}"))
                          (push-reduction '|PARSE-Enclosure|
                              (cons '|brace|
                               (list (list '|construct| (pop-stack-1))))))
                     (and (match-advance-string "}")
                          (push-reduction '|PARSE-Enclosure|
                              (list '|brace|)))))))))
```

———-

8.2.47 defun PARSE-IntegerTok

[parse-number p300]

— defun PARSE-IntegerTok —

```
(defun |PARSE-IntegerTok| () (parse-number))
```

———-

8.2.48 defun PARSE-FormalParameter

[PARSE-FormalParameterTok p263]

— **defun PARSE-FormalParameter** —

```
(defun |PARSE-FormalParameter| () (|PARSE-FormalParameterTok|))
```

———-

8.2.49 defun PARSE-FormalParameterTok

[parse-argument-designator p301]

— **defun PARSE-FormalParameterTok** —

```
(defun |PARSE-FormalParameterTok| () (parse-argument-designator))
```

———-

8.2.50 defun PARSE-Quad

[match-advance-string p279]
[push-reduction p292]
[PARSE-GliphTok p268]
[$boot p**??**]

— **defun PARSE-Quad** —

```
(defun |PARSE-Quad| ()
 (or (and (match-advance-string "$")
          (push-reduction '|PARSE-Quad| '$))
     (and $boot (|PARSE-GliphTok| '|.|)
          (push-reduction '|PARSE-Quad| '|.|))))
```

———-

8.2.51 defun PARSE-String

[parse-spadstring p298]

— **defun PARSE-String** —

```
(defun |PARSE-String| () (parse-spadstring))
```

————-

8.2.52 defun PARSE-VarForm

[PARSE-Name p265]
[optional p291]
[PARSE-Scripts p264]
[push-reduction p292]
[pop-stack-2 p303]
[pop-stack-1 p302]

— **defun PARSE-VarForm** —

```
(defun |PARSE-VarForm| ()
  (and (|PARSE-Name|)
       (optional
           (and (|PARSE-Scripts|)
                (push-reduction '|PARSE-VarForm|
                    (list '|Scripts| (pop-stack-2) (pop-stack-1)))))
       (push-reduction '|PARSE-VarForm| (pop-stack-1))))
```

————-

8.2.53 defun PARSE-Scripts

[match-advance-string p279]
[must p290]
[PARSE-ScriptItem p265]

— **defun PARSE-Scripts** —

```
(defun |PARSE-Scripts| ()
 (and nonblank (match-advance-string "[") (must (|PARSE-ScriptItem|))
      (must (match-advance-string "]"))))
```

————-

8.2.54 defun PARSE-ScriptItem

[PARSE-Expr p250]
[optional p291]
[star p291]
[match-advance-string p279]
[must p290]
[PARSE-ScriptItem p265]
[push-reduction p292]
[pop-stack-2 p303]
[pop-stack-1 p302]

— **defun PARSE-ScriptItem** —

```
(defun |PARSE-ScriptItem| ()
  (or (and (|PARSE-Expr| 90)
           (optional
               (and (star repeator
                          (and (match-advance-string ";")
                               (must (|PARSE-ScriptItem|))))
                    (push-reduction '|PARSE-ScriptItem|
                        (cons '|;|
                              (cons (pop-stack-2)
                                    (append (pop-stack-1) nil)))))))
      (and (match-advance-string ";") (must (|PARSE-ScriptItem|))
           (push-reduction '|PARSE-ScriptItem|
               (list '|PrefixSC| (pop-stack-1))))))
```

———-

8.2.55 defun PARSE-Name

[parse-identifier p299]
[push-reduction p292]
[pop-stack-1 p302]

— **defun PARSE-Name** —

```
(defun |PARSE-Name| ()
  (and (parse-identifier) (push-reduction '|PARSE-Name| (pop-stack-1))))
```

———-

8.2.56 defun PARSE-Data

[action p291]
[PARSE-Sexpr p266]
[push-reduction p292]
[translabel p295]
[pop-stack-1 p302]
[labasoc p**??**]

— **defun PARSE-Data** —

```
(defun |PARSE-Data| ()
  (declare (special lablasoc))
  (and (action (setq lablasoc nil)) (|PARSE-Sexpr|)
       (push-reduction '|PARSE-Data|
           (list 'quote (translabel (pop-stack-1) lablasoc)))))
```

———-

8.2.57 defun PARSE-Sexpr

[PARSE-Sexpr1 p266]

— **defun PARSE-Sexpr** —

```
(defun |PARSE-Sexpr| ()
  (and (action (advance-token)) (|PARSE-Sexpr1|)))
```

———-

8.2.58 defun PARSE-Sexpr1

[PARSE-AnyId p268]
[optional p291]
[PARSE-NBGliphTok p267]
[must p290]
[PARSE-Sexpr1 p266]
[action p291]
[pop-stack-2 p303]
[nth-stack p304]
[match-advance-string p279]
[push-reduction p292]
[PARSE-IntegerTok p262]
[pop-stack-1 p302]

[PARSE-String p263]
[bang p??]
[star p291]
[PARSE-GliphTok p268]

— **defun PARSE-Sexpr1** —

```
(defun |PARSE-Sexpr1| ()
  (or (and (|PARSE-AnyId|)
           (optional
               (and (|PARSE-NBGliphTok| '=) (must (|PARSE-Sexpr1|))
                    (action (setq lablasoc
                                  (cons (cons (pop-stack-2)
                                              (nth-stack 1))
                                        lablasoc))))))
      (and (match-advance-string "'") (must (|PARSE-Sexpr1|))
           (push-reduction '|PARSE-Sexpr1|
               (list 'quote (pop-stack-1))))
      (|PARSE-IntegerTok|)
      (and (match-advance-string "-") (must (|PARSE-IntegerTok|))
           (push-reduction '|PARSE-Sexpr1| (- (pop-stack-1))))
      (|PARSE-String|)
      (and (match-advance-string "<")
           (bang fil_test (optional (star repeator (|PARSE-Sexpr1|))))
           (must (match-advance-string ">"))
           (push-reduction '|PARSE-Sexpr1| (list2vec (pop-stack-1))))
      (and (match-advance-string "(")
           (bang fil_test
                 (optional
                     (and (star repeator (|PARSE-Sexpr1|))
                          (optional
                              (and (|PARSE-GliphTok| '|.|)
                                   (must (|PARSE-Sexpr1|))
                                   (push-reduction '|PARSE-Sexpr1|
                                    (nconc (pop-stack-2) (pop-stack-1))))))))
           (must (match-advance-string ")")))))
```

———-

8.2.59 defun PARSE-NBGliphTok

[match-current-token p283]
[action p291]
[advance-token p286]
[tok p240]

— **defun PARSE-NBGliphTok** —

```
(defun |PARSE-NBGliphTok| (|tok|)
 (declare (special |tok|))
 (and (match-current-token 'gliph |tok|) nonblank (action (advance-token))))
```

———-

8.2.60 defun PARSE-GliphTok

[match-current-token p283]
[action p291]
[advance-token p286]
[tok p240]

— **defun PARSE-GliphTok** —

```
(defun |PARSE-GliphTok| (|tok|)
 (declare (special |tok|))
 (and (match-current-token 'gliph |tok|) (action (advance-token))))
```

———-

8.2.61 defun PARSE-AnyId

[parse-identifier p299]
[match-string p278]
[push-reduction p292]
[current-symbol p284]
[action p291]
[advance-token p286]
[parse-keyword p300]

— **defun PARSE-AnyId** —

```
(defun |PARSE-AnyId| ()
  (or (parse-identifier)
      (or (and (match-string "$")
               (push-reduction '|PARSE-AnyId| (current-symbol))
               (action (advance-token)))
          (parse-keyword))))
```

———-

8.2.62 defun PARSE-Sequence

[PARSE-OpenBracket p270]
[must p290]
[PARSE-Sequence1 p269]
[match-advance-string p279]
[PARSE-OpenBrace p270]
[push-reduction p292]
[pop-stack-1 p302]

— **defun PARSE-Sequence** —

```
(defun |PARSE-Sequence| ()
  (or (and (|PARSE-OpenBracket|) (must (|PARSE-Sequence1|))
           (must (match-advance-string "]")))
      (and (|PARSE-OpenBrace|) (must (|PARSE-Sequence1|))
           (must (match-advance-string "}"))
           (push-reduction '|PARSE-Sequence|
               (list '|brace| (pop-stack-1))))))
```

———-

8.2.63 defun PARSE-Sequence1

[PARSE-Expression p249]
[push-reduction p292]
[pop-stack-2 p303]
[pop-stack-1 p302]
[optional p291]
[PARSE-IteratorTail p271]

— **defun PARSE-Sequence1** —

```
(defun |PARSE-Sequence1| ()
  (and (or (and (|PARSE-Expression|)
                (push-reduction '|PARSE-Sequence1|
                    (list (pop-stack-2) (pop-stack-1))))
           (push-reduction '|PARSE-Sequence1| (list (pop-stack-1))))
       (optional
           (and (|PARSE-IteratorTail|)
                (push-reduction '|PARSE-Sequence1|
                    (cons 'collect
                          (append (pop-stack-1)
                                  (list (pop-stack-1)))))))))
```

———-

8.2.64 defun PARSE-OpenBracket

[getToken p282]
[current-symbol p284]
[eqcar p??]
[push-reduction p292]
[action p291]
[advance-token p286]

— **defun PARSE-OpenBracket** —

```
(defun |PARSE-OpenBracket| ()
 (let (g1)
  (and (eq (|getToken| (setq g1 (current-symbol))) '[)
       (must (or (and (eqcar g1 '|elt|)
                      (push-reduction '|PARSE-OpenBracket|
                          (list '|elt| (second g1) '|construct|)))
                 (push-reduction '|PARSE-OpenBracket| '|construct|)))
       (action (advance-token)))))
```

———-

8.2.65 defun PARSE-OpenBrace

[getToken p282]
[current-symbol p284]
[eqcar p??]
[push-reduction p292]
[action p291]
[advance-token p286]

— **defun PARSE-OpenBrace** —

```
(defun |PARSE-OpenBrace| ()
 (let (g1)
  (and (eq (|getToken| (setq g1 (current-symbol))) '{)
       (must (or (and (eqcar g1 '|elt|)
                      (push-reduction '|PARSE-OpenBrace|
                          (list '|elt| (second g1) '|brace|)))
                 (push-reduction '|PARSE-OpenBrace| '|construct|)))
       (action (advance-token)))))
```

———-

8.2.66 defun PARSE-IteratorTail

[match-advance-string p279]
[bang p??]
[optional p291]
[star p291]
[PARSE-Iterator p271]

— **defun PARSE-IteratorTail** —

```
(defun |PARSE-IteratorTail| ()
 (or (and (match-advance-string "repeat")
          (bang fil_test (optional (star repeator (|PARSE-Iterator|)))))
     (star repeator (|PARSE-Iterator|))))
```

———-

8.2.67 defun PARSE-Iterator

[match-advance-string p279]
[must p290]
[PARSE-Primary p258]
[PARSE-Expression p249]
[PARSE-Expr p250]
[pop-stack-3 p303]
[pop-stack-2 p303]
[pop-stack-1 p302]
[optional p291]

— **defun PARSE-Iterator** —

```
(defun |PARSE-Iterator| ()
  (or (and (match-advance-string "for") (must (|PARSE-Primary|))
           (must (match-advance-string "in"))
           (must (|PARSE-Expression|))
           (must (or (and (match-advance-string "by")
                          (must (|PARSE-Expr| 200))
                          (push-reduction '|PARSE-Iterator|
                           (list 'inby (pop-stack-3)
                                       (pop-stack-2) (pop-stack-1))))
                     (push-reduction '|PARSE-Iterator|
                         (list 'in (pop-stack-2) (pop-stack-1)))))
           (optional
               (and (match-advance-string "|")
                    (must (|PARSE-Expr| 111))
                    (push-reduction '|PARSE-Iterator|
```

```
                    (list '|\|| (pop-stack-1))))))
      (and (match-advance-string "while") (must (|PARSE-Expr| 190))
           (push-reduction '|PARSE-Iterator|
               (list 'while (pop-stack-1))))
      (and (match-advance-string "until") (must (|PARSE-Expr| 190))
           (push-reduction '|PARSE-Iterator|
               (list 'until (pop-stack-1))))))
```

———-

8.2.68 The PARSE implicit routines

These symbols are not explicitly referenced in the source. Nevertheless, they are called during runtime. For example, PARSE-SemiColon is called in the chain:

```
PARSE-Enclosure {loc0=nil,loc1="(V  ==> Vector; "} [ihs=35]
 PARSE-Expr
  PARSE-LedPart
   PARSE-Operation
    PARSE-getSemanticForm
     PARSE-SemiColon
```

so there is a bit of indirection involved in the call.

8.2.69 defun PARSE-Suffix

[push-reduction p292]
[current-symbol p284]
[action p291]
[advance-token p286]
[optional p291]
[PARSE-TokTail p254]
[pop-stack-1 p302]

— defun PARSE-Suffix —

```
(defun |PARSE-Suffix| ()
  (and (push-reduction '|PARSE-Suffix| (current-symbol))
       (action (advance-token)) (optional (|PARSE-TokTail|))
       (push-reduction '|PARSE-Suffix|
           (list (pop-stack-1) (pop-stack-1)))))
```

———-

8.2.70 defun PARSE-SemiColon

[match-advance-string p279]
[must p290]
[PARSE-Expr p250]
[push-reduction p292]
[pop-stack-2 p303]
[pop-stack-1 p302]

— **defun PARSE-SemiColon** —

```
(defun |PARSE-SemiColon| ()
  (and (match-advance-string ";")
       (must (or (|PARSE-Expr| 82)
                 (push-reduction '|PARSE-SemiColon| '|/throwAway|)))
       (push-reduction '|PARSE-SemiColon|
           (list '|;| (pop-stack-2) (pop-stack-1)))))
```

———-

8.2.71 defun PARSE-Return

[match-advance-string p279]
[must p290]
[PARSE-Expression p249]
[push-reduction p292]
[pop-stack-1 p302]

— **defun PARSE-Return** —

```
(defun |PARSE-Return| ()
  (and (match-advance-string "return") (must (|PARSE-Expression|))
       (push-reduction '|PARSE-Return|
           (list '|return| (pop-stack-1)))))
```

———-

8.2.72 defun PARSE-Exit

[match-advance-string p279]
[must p290]
[PARSE-Expression p249]
[push-reduction p292]
[pop-stack-1 p302]

— defun PARSE-Exit —

```
(defun |PARSE-Exit| ()
  (and (match-advance-string "exit")
       (must (or (|PARSE-Expression|)
                 (push-reduction '|PARSE-Exit| '|$NoValue|)))
       (push-reduction '|PARSE-Exit|
           (list '|exit| (pop-stack-1)))))
```

———-

8.2.73 defun PARSE-Leave

[match-advance-string p279]
[PARSE-Expression p249]
[must p290]
[push-reduction p292]
[PARSE-Label p257]
[pop-stack-1 p302]

— defun PARSE-Leave —

```
(defun |PARSE-Leave| ()
  (and (match-advance-string "leave")
       (must (or (|PARSE-Expression|)
                 (push-reduction '|PARSE-Leave| '|$NoValue|)))
       (must (or (and (match-advance-string "from")
                      (must (|PARSE-Label|))
                      (push-reduction '|PARSE-Leave|
                          (list '|leaveFrom| (pop-stack-1) (pop-stack-1))))
                 (push-reduction '|PARSE-Leave|
                     (list '|leave| (pop-stack-1)))))))
```

———-

8.2.74 defun PARSE-Seg

[PARSE-GliphTok p268]
[bang p**??**]
[optional p291]
[PARSE-Expression p249]
[push-reduction p292]
[pop-stack-2 p303]

[pop-stack-1 p302]

— **defun PARSE-Seg** —

```
(defun |PARSE-Seg| ()
  (and (|PARSE-GliphTok| '|..|)
       (bang fil_test (optional (|PARSE-Expression|)))
       (push-reduction '|PARSE-Seg|
           (list 'segment (pop-stack-2) (pop-stack-1)))))
```

————-

8.2.75 defun PARSE-Conditional

[match-advance-string p279]
[must p290]
[PARSE-Expression p249]
[bang p??]
[optional p291]
[PARSE-ElseClause p275]
[push-reduction p292]
[pop-stack-3 p303]
[pop-stack-2 p303]
[pop-stack-1 p302]

— **defun PARSE-Conditional** —

```
(defun |PARSE-Conditional| ()
  (and (match-advance-string "if") (must (|PARSE-Expression|))
       (must (match-advance-string "then")) (must (|PARSE-Expression|))
       (bang fil_test
             (optional
                 (and (match-advance-string "else")
                      (must (|PARSE-ElseClause|)))))
       (push-reduction '|PARSE-Conditional|
           (list '|if| (pop-stack-3) (pop-stack-2) (pop-stack-1)))))
```

————-

8.2.76 defun PARSE-ElseClause

[current-symbol p284]
[PARSE-Conditional p275]
[PARSE-Expression p249]

— **defun PARSE-ElseClause** —

```
(defun |PARSE-ElseClause| ()
  (or (and (eq (current-symbol) '|if|) (|PARSE-Conditional|))
      (|PARSE-Expression|)))
```

———-

8.2.77 defun PARSE-Loop

[star p291]
[PARSE-Iterator p271]
[must p290]
[match-advance-string p279]
[PARSE-Expr p250]
[push-reduction p292]
[pop-stack-2 p303]
[pop-stack-1 p302]

— **defun PARSE-Loop** —

```
(defun |PARSE-Loop| ()
  (or (and (star repeator (|PARSE-Iterator|))
           (must (match-advance-string "repeat"))
           (must (|PARSE-Expr| 110))
           (push-reduction '|PARSE-Loop|
               (cons 'repeat
                     (append (pop-stack-2) (list (pop-stack-1))))))
      (and (match-advance-string "repeat") (must (|PARSE-Expr| 110))
           (push-reduction '|PARSE-Loop|
               (list 'repeat (pop-stack-1))))))
```

———-

8.2.78 defun PARSE-LabelExpr

[PARSE-Label p257]
[must p290]
[PARSE-Expr p250]
[push-reduction p292]
[pop-stack-2 p303]
[pop-stack-1 p302]

— **defun PARSE-LabelExpr** —

```
(defun |PARSE-LabelExpr| ()
  (and (|PARSE-Label|) (must (|PARSE-Expr| 120))
       (push-reduction '|PARSE-LabelExpr|
           (list 'label (pop-stack-2) (pop-stack-1)))))
```

———-

8.2.79 defun PARSE-FloatTok

[parse-number p300]
[push-reduction p292]
[pop-stack-1 p302]
[bfp- p??]
[$boot p??]

— **defun PARSE-FloatTok** —

```
(defun |PARSE-FloatTok| ()
  (and (parse-number)
       (push-reduction '|PARSE-FloatTok|
           (if $boot (pop-stack-1) (bfp- (pop-stack-1))))))
```

———-

8.3 The PARSE support routines

This section is broken up into 3 levels:

- String grabbing: Match String, Match Advance String
- Token handling: Current Token, Next Token, Advance Token
- Character handling: Current Char, Next Char, Advance Char
- Line handling: Next Line, Print Next Line
- Error Handling
- Floating Point Support
- Dollar Translation

8.3.1 String grabbing

String grabbing is the art of matching initial segments of the current line, and removing them from the line before the get tokenized if they match (or removing the corresponding current tokens).

8.3.2 defun match-string

The match-string function returns length of X if X matches initial segment of inputstream. [unget-tokens p282]
[skip-blanks p278]
[line-past-end-p p93]
[current-char p287]
[initial-substring-p p280]
[subseq p**??**]
[$line p92]
[line p92]

— **defun match-string** —

```
(defun match-string (x)
  (unget-tokens) ; So we don't get out of synch with token stream
  (skip-blanks)
  (if (and (not (line-past-end-p current-line)) (current-char) )
    (initial-substring-p x
     (subseq (line-buffer current-line) (line-current-index current-line)))))
```

————-

8.3.3 defun skip-blanks

[current-char p287]
[token-lookahead-type p279]
[advance-char p**??**]

— **defun skip-blanks** —

```
(defun skip-blanks ()
  (loop (let ((cc (current-char)))
          (if (not cc) (return nil))
          (if (eq (token-lookahead-type cc) 'white)
              (if (not (advance-char)) (return nil))
              (return t)))))
```

———-

— **initvars** —

```
(defvar Escape-Character #\\ "Superquoting character.")
```

———-

8.3.4 defun token-lookahead-type

[Escape-Character p??]

— **defun token-lookahead-type** —

```
(defun token-lookahead-type (char)
  "Predicts the kind of token to follow, based on the given initial character."
 (declare (special Escape-Character))
  (cond
   ((not char)                                          'eof)
   ((or (char= char Escape-Character) (alpha-char-p char)) 'id)
   ((digitp char)                                       'num)
   ((char= char #\')                                    'string)
   ((char= char #\[)                                    'bstring)
   ((member char '(#\Space #\Tab #\Return) :test #'char=)  'white)
   (t                                                    'special-char)))
```

———-

8.3.5 defun match-advance-string

The match-string function returns length of X if X matches initial segment of inputstream. If it is successful, advance inputstream past X. [quote-if-string p280]
[current-token p285]
[match-string p278]
[line-current-index p??]
[line-past-end-p p93]
[line-current-char p??]
[$token p99]
[$line p92]

— **defun match-advance-string** —

```
(defun match-advance-string (x)
  (let ((y (if (>= (length (string x))
                   (length (string (quote-if-string (current-token)))))
               (match-string x)
               nil))) ; must match at least the current token
    (when y
      (incf (line-current-index current-line) y)
      (if (not (line-past-end-p current-line))
       (setf (line-current-char current-line)
             (elt (line-buffer current-line)
                  (line-current-index current-line)))
       (setf (line-current-char current-line) #\space))
      (setq prior-token
       (make-token :symbol (intern (string x))
                   :type 'identifier
                   :nonblank nonblank))
      t)))
```

———-

8.3.6 defun initial-substring-p

[string-not-greaterp p??]

— **defun initial-substring-p** —

```
(defun initial-substring-p (part whole)
  "Returns length of part if part matches initial segment of whole."
  (let ((x (string-not-greaterp part whole)))
    (and x (= x (length part)) x)))
```

———-

8.3.7 defun quote-if-string

[token-type p??]
[strconc p??]
[token-symbol p??]
[underscore p282]
[token-nonblank p??]
[pack p??]
[escape-keywords p281]
[$boot p??]
[$spad p329]

— **defun quote-if-string** —

```
(defun quote-if-string (token)
 (declare (special $boot $spad))
 (when token   ;only use token-type on non-null tokens
  (case (token-type token)
   (bstring      (strconc "[" (token-symbol token) "]*"))
   (string       (strconc "'" (token-symbol token) "'"))
   (spadstring   (strconc "\"" (underscore (token-symbol token)) "\""))
   (number       (format nil "~v,'0D" (token-nonblank token)
                              (token-symbol token)))
   (special-char (string (token-symbol token)))
   (identifier   (let ((id (symbol-name (token-symbol token)))
                       (pack (package-name (symbol-package
                                            (token-symbol token)))))
                  (if (or $boot $spad)
                   (if (string= pack "BOOT")
                    (escape-keywords (underscore id) (token-symbol token))
                    (concatenate 'string
                      (underscore pack) "'" (underscore id)))
                   id)))
   (t            (token-symbol token)))))
```

———-

8.3.8 defun escape-keywords

— **defun escape-keywords** —

```
(defun escape-keywords (pname id)
  (if (member id keywords)
   (concatenate 'string "_" pname)
   pname))
```

———-

8.3.9 defun isTokenDelimiter

NIL needed below since END_UNIT is not generated by current parser [current-symbol p284]

— **defun isTokenDelimiter** —

```
(defun |isTokenDelimiter| ()
 (member (current-symbol) '(\) end\_unit nil)))
```

———-

8.3.10 defun underscore

[vector-push p??]

— **defun underscore** —

```
(defun underscore (string)
 (if (every #'alpha-char-p string)
  string
  (let* ((size (length string))
         (out-string (make-array (* 2 size)
                                 :element-type 'string-char
                                 :fill-pointer 0))
         next-char)
   (dotimes (i size)
    (setq next-char (char string i))
    (unless (alpha-char-p next-char) (vector-push #\_ out-string))
    (vector-push next-char out-string))
   out-string)))
```

———-

8.3.11 Token Handling

8.3.12 defun getToken

[eqcar p??]

— **defun getToken** —

```
(defun |getToken| (x)
 (if (eqcar x '|elt|) (third x) x))
```

———-

8.3.13 defun unget-tokens

[quote-if-string p280]
[line-current-segment p94]

[strconc p**??**]
[line-number p**??**]
[token-nonblank p**??**]
[line-new-line p94]
[line-number p**??**]
[valid-tokens p100]

— **defun unget-tokens** —

```
(defun unget-tokens ()
 (case valid-tokens
  (0 t)
  (1 (let* ((cursym (quote-if-string current-token))
            (curline (line-current-segment current-line))
            (revised-line (strconc cursym curline (copy-seq " "))))
         (line-new-line revised-line current-line (line-number current-line))
         (setq nonblank (token-nonblank current-token))
         (setq valid-tokens 0)))
  (2 (let* ((cursym (quote-if-string current-token))
            (nextsym (quote-if-string next-token))
            (curline (line-current-segment Current-Line))
            (revised-line
             (strconc (if (token-nonblank current-token) "" " ")
                      cursym
                      (if (token-nonblank next-token) "" " ")
                      nextsym curline " ")))
      (setq nonblank (token-nonblank current-token))
      (line-new-line revised-line current-line (line-number current-line))
      (setq valid-tokens 0)))
  (t (error "How many tokens do you think you have?"))))
```

———-

8.3.14 defun match-current-token

This returns the current token if it has EQ type and (optionally) equal symbol.
[current-token p285]
[match-token p284]

— **defun match-current-token** —

```
(defun match-current-token (type &optional (symbol nil))
  (match-token (current-token) type symbol))
```

———-

8.3.15 defun match-token

[token-type p??]
[token-symbol p??]

— **defun match-token** —

```
(defun match-token (token type &optional (symbol nil))
  (when (and token (eq (token-type token) type))
   (if symbol
    (when (equal symbol (token-symbol token)) token)
    token)))
```

———-

8.3.16 defun match-next-token

This returns the next token if it has equal type and (optionally) equal symbol.
[next-token p286]
[match-token p284]

— **defun match-next-token** —

```
(defun match-next-token (type &optional (symbol nil))
  (match-token (next-token) type symbol))
```

———-

8.3.17 defun current-symbol

[make-symbol-of p284]
[current-token p285]

— **defun current-symbol** —

```
(defun current-symbol ()
 (make-symbol-of (current-token)))
```

———-

8.3.18 defun make-symbol-of

[$token p99]

— **defun make-symbol-of** —

```
(defun make-symbol-of (token)
 (let ((u (and token (token-symbol token))))
  (cond
   ((not u) nil)
   ((characterp u) (intern (string u)))
   (u))))
```

————-

8.3.19 defun current-token

This returns the current token getting a new one if necessary. [try-get-token p285]
[valid-tokens p100]
[current-token p285]

— **defun current-token** —

```
(defun current-token ()
 (declare (special valid-tokens current-token))
 (if (> valid-tokens 0)
  current-token
  (try-get-token current-token)))
```

————-

8.3.20 defun try-get-token

[get-token p287]
[valid-tokens p100]

— **defun try-get-token** —

```
(defun try-get-token (token)
 (declare (special valid-tokens))
 (let ((tok (get-token token)))
  (when tok
   (incf valid-tokens)
   token)))
```

————-

8.3.21 defun next-token

This returns the token after the current token, or NIL if there is none after.
[try-get-token p285]
[current-token p285]
[valid-tokens p100]
[next-token p286]

— **defun next-token** —

```
(defun next-token ()
 (declare (special valid-tokens next-token))
 (current-token)
 (if (> valid-tokens 1)
  next-token
  (try-get-token next-token)))
```

———-

8.3.22 defun advance-token

This makes the next token be the current token. [current-token p285]
[copy-token p**??**]
[try-get-token p285]
[valid-tokens p100]
[current-token p285]

— **defun advance-token** —

```
(defun advance-token ()
  (current-token)                        ;don't know why this is needed
  (case valid-tokens
    (0 (try-get-token (current-token)))
    (1 (decf valid-tokens)
       (setq prior-token (copy-token current-token))
       (try-get-token current-token))
    (2 (setq prior-token (copy-token current-token))
       (setq current-token (copy-token next-token))
       (decf valid-tokens))))
```

———-

8.3.23 defvar $XTokenReader

— **initvars** —

```
(defvar XTokenReader 'get-meta-token "Name of tokenizing function")
```

———-

8.3.24 defun get-token

[XTokenReader p287]
[XTokenReader p287]

— **defun get-token** —

```
(defun get-token (token)
 (funcall XTokenReader token))
```

———-

8.3.25 Character handling

8.3.26 defun current-char

This returns the current character of the line, initially blank for an unread line.
[$line p92]
[current-line p92]

— **defun current-char** —

```
(defun current-char ()
  (if (line-past-end-p current-line)
   #\return
   (line-current-char current-line)))
```

———-

8.3.27 defun next-char

This returns the character after the current character, blank if at end of line. The blank-at-end-of-line assumption is allowable because we assume that end-of-line is a token separator, which blank is equivalent to. [line-at-end-p p93]

[line-next-char p93]
[current-line p92]

— **defun next-char** —

```
(defun next-char ()
 (if (line-at-end-p current-line)
  #\return
  (line-next-char current-line)))
```

————-

8.3.28 defun char-eq

— **defun char-eq** —

```
(defun char-eq (x y)
 (char= (character x) (character y)))
```

————-

8.3.29 defun char-ne

— **defun char-ne** —

```
(defun char-ne (x y)
 (char/= (character x) (character y)))
```

————-

8.3.30 Error handling

8.3.31 defvar $meta-error-handler

— **initvars** —

```
(defvar meta-error-handler 'meta-meta-error-handler)
```

————-

8.3.32 defun meta-syntax-error

[meta-error-handler p288]
[meta-error-handler p288]

— **defun meta-syntax-error** —

```
(defun meta-syntax-error (&optional (wanted nil) (parsing nil))
  (declare (special meta-error-handler))
  (funcall meta-error-handler wanted parsing))
```

———-

8.3.33 Floating Point Support

8.3.34 defun floatexpid

[floatexpid identp (vol5)]
[floatexpid pname (vol5)]
[spadreduce p**??**]
[collect p204]
[step p**??**]
[maxindex p**??**]
[floatexpid digitp (vol5)]

— **defun floatexpid** —

```
(defun floatexpid (x &aux s)
 (when (and (identp x) (char= (char-upcase (elt (setq s (pname x)) 0)) #\E)
            (> (length s) 1)
            (spadreduce and 0 (collect (step i 1 1 (maxindex s))
                                       (digitp (elt s i)))))
   (read-from-string s t nil :start 1)))
```

———-

8.3.35 Dollar Translation

8.3.36 defun dollarTran

[$InteractiveMode p**??**]

— **defun dollarTran** —

```
(defun |dollarTran| (dom rand)
 (let ((eltWord (if |$InteractiveMode| '|$elt| '|elt|)))
  (declare (special |$InteractiveMode|))
  (if (and (not (atom rand)) (cdr rand))
   (cons (list eltWord dom (car rand)) (cdr rand))
   (list eltWord dom rand))))
```

———-

8.3.37 Applying metagrammatical elements of a production (e.g., Star).

- **must** means that if it is not present in the token stream, it is a syntax error.
- **optional** means that if it is present in the token stream, that is a good thing, otherwise don't worry (like [foo] in BNF notation).
- **action** is something we do as a consequence of successful parsing; it is inserted at the end of the conjunction of requirements for a successful parse, and so should return T.
- **sequence** consists of a head, which if recognized implies that the tail must follow. Following tail are actions, which are performed upon recognizing the head and tail.

8.3.38 defmacro Bang

If the execution of prod does not result in an increase in the size of the stack, then stack a NIL. Return the value of prod.

— **defmacro bang** —

```
(defmacro bang (lab prod)
 `(progn
   (setf (stack-updated reduce-stack) nil)
   (let* ((prodvalue ,prod) (updated (stack-updated reduce-stack)))
    (unless updated (push-reduction ',lab nil))
    prodvalue)))
```

———-

8.3.39 defmacro must

[meta-syntax-error p289]

— **defmacro must** —

```
(defmacro must (dothis &optional (this-is nil) (in-rule nil))
  `(or ,dothis (meta-syntax-error ,this-is ,in-rule)))
```

———-

8.3.40 defun action

— **defun action** —

```
(defun action (dothis) (or dothis t))
```

———-

8.3.41 defun optional

— **defun optional** —

```
(defun optional (dothis) (or dothis t))
```

———-

8.3.42 defmacro star

Succeeds if there are one or more of PROD, stacking as one unit the sub-reductions of PROD and labelling them with LAB. E.G., `(Star IDs (parse-id))` with A B C will stack (3 IDs (A B C)), where (parse-id) would stack (1 ID (A)) when applied once. [stack-size p**??**]
[push-reduction p292]
[pop-stack-1 p302]

— **defmacro star** —

```
(defmacro star (lab prod)
  `(prog ((oldstacksize (stack-size reduce-stack)))
     (if (not ,prod) (return nil))
loop
     (if (not ,prod)
      (let* ((newstacksize (stack-size reduce-stack))
```

```
          (number-of-new-reductions (- newstacksize oldstacksize)))
      (if (> number-of-new-reductions 0)
       (return (do ((i 0 (1+ i)) (accum nil))
                   ((= i number-of-new-reductions)
                     (push-reduction ',lab accum)
                     (return t))
                 (push (pop-stack-1) accum)))
       (return t)))
    (go loop))))
```

———-

8.3.43 Stacking and retrieving reductions of rules.

8.3.44 defvar $reduce-stack

Stack of results of reduced productions. [$stack p97]

— **initvars** —

```
(defvar reduce-stack (make-stack) )
```

———-

8.3.45 defmacro reduce-stack-clear

— **defmacro reduce-stack-clear** —

```
(defmacro reduce-stack-clear () `(stack-load nil reduce-stack))
```

———-

8.3.46 defun push-reduction

[stack-push p98]
[make-reduction p**??**]
[reduce-stack p292]

— **defun push-reduction** —

```
(defun push-reduction (rule redn)
  (stack-push (make-reduction :rule rule :value redn) reduce-stack))
```

Chapter 9

Utility Functions

9.0.47 defun translabel

[translabel1 p295]

— **defun translabel** —

```
(defun translabel (x al)
 (translabel1 x al) x)
```

————-

9.0.48 defun translabel1

[refvecp p??]
[maxindex p??]
[translabel1 p295]
[lassoc p??]

— **defun translabel1** —

```
(defun translabel1 (x al)
 "Transforms X according to AL = ((<label> . Sexpr) ..)."
  (cond
   ((refvecp x)
    (do ((i 0 (1+ i)) (k (maxindex x)))
        ((> i k))
      (if (let ((y (lassoc (elt x i) al))) (setelt x i y))
       (translabel1 (elt x i) al))))
   ((atom x) nil)
   ((let ((y (lassoc (first x) al)))
```

```
          (if y (setf (first x) y) (translabel1 (cdr x) al))))
   ((translabel1 (first x) al) (translabel1 (cdr x) al))))
```

———-

9.0.49 defun displayPreCompilationErrors

[length p??]
[remdup p??]
[sayBrightly p??]
[nequal p??]
[sayMath p??]
[$postStack p??]
[$topOp p??]

— defun displayPreCompilationErrors —

```
(defun |displayPreCompilationErrors| ()
 (let (n errors heading)
  (declare (special |$postStack| |$topOp|))
  (setq n (|#| (setq |$postStack| (remdup (nreverse |$postStack|)))))
  (unless (eql n 0)
    (setq errors (cond ((> n 1) "errors") (t "error")))
    (cond
     (|$InteractiveMode|
      (|sayBrightly| (list "   Semantic " errors " detected: ")))
     (t
      (setq heading
       (if (nequal |$topOp| '|$topOp|)
        (list "   " |$topOp| " has")
        (list "   You have")))
      (|sayBrightly|
       (append heading (list n "precompilation " errors ":" )))))
    (cond
     ((> n 1)
       (let ((i 1))
        (dolist (x |$postStack|)
          (|sayMath| (cons "   " (cons i (cons ") " x)))))))
     (t (|sayMath| (cons "    " (car |$postStack|)))))
    (terpri))))
```

———-

9.0.50 defun bumperrorcount

[$InteractiveMode p??]
[$spad-errors p??]

— **defun bumperrorcount** —

```
(defun bumperrorcount (kind)
 (unless |$InteractiveMode|
  (let ((index (case kind
                (|syntax| 0)
                (|precompilation| 1)
                (|semantic| 2)
                (t (error "BUMPERRORCOUNT")))))
    (setelt $spad_errors index (1+ (elt $spad_errors index))))))
```

———-

9.0.51 defun parseTranCheckForRecord

```
;parseTranCheckForRecord(x,op) ==
;  (x:= parseTran x) is ['Record,:l] =>
;    or/[y for y in l | y isnt [":",.,.]] =>
;      postError ['"   Constructor",:bright x,'"has missing label"]
;    x
;  x
```

[qcar p??]
[qcdr p??]
[postError p196]
[parseTran p103]

— **defun parseTranCheckForRecord** —

```
(defun |parseTranCheckForRecord| (x op)
 (let (tmp3)
  (setq x (|parseTran| x))
  (cond
   ((and (pairp x) (eq (qcar x) '|Record|))
     (cond
      ((do ((z nil tmp3) (tmp4 (qcdr x) (cdr tmp4)) (y nil))
           ((or z (atom tmp4)) tmp3)
          (setq y (car tmp4))
          (cond
           ((null (and (pairp y) (eq (qcar y) '|:|) (pairp (qcdr y))
                  (pairp (qcdr (qcdr y))) (eq (qcdr (qcdr (qcdr y))) nil)))
              (setq tmp3 (or tmp3 y)))))
```

```
          (|postError| (list "   Constructor" x "has missing label" )))
        (t x)))
   (t x))))
```

———-

9.0.52 defun new2OldLisp

[new2OldTran p**??**]
[postTransform p191]

— **defun new2OldLisp** —

```
(defun |new2OldLisp| (x)
 (|new2OldTran| (|postTransform| x)))
```

———-

9.0.53 defun makeSimplePredicateOrNil

[isSimple p**??**]
[isAlmostSimple p**??**]
[wrapSEQExit p**??**]

— **defun makeSimplePredicateOrNil** —

```
(defun |makeSimplePredicateOrNil| (p)
 (let (u g)
  (cond
   ((|isSimple| p) nil)
   ((setq u (|isAlmostSimple| p)) u)
   (t (|wrapSEQExit| (list (list 'let (setq g (gensym)) p) g))))))
```

———-

9.0.54 defun parse-spadstring

[match-current-token p283]
[token-symbol p**??**]
[push-reduction p292]
[advance-token p286]

— **defun parse-spadstring** —

```
(defun parse-spadstring ()
  (let* ((tok (match-current-token 'spadstring))
         (symbol (if tok (token-symbol tok))))
    (when tok
      (push-reduction 'spadstring-token (copy-tree symbol))
      (advance-token)
      t)))
```

———-

9.0.55 defun parse-string

[match-current-token p283]
[token-symbol p??]
[push-reduction p292]
[advance-token p286]

— **defun parse-string** —

```
(defun parse-string ()
  (let* ((tok (match-current-token 'string))
         (symbol (if tok (token-symbol tok))))
    (when tok
      (push-reduction 'string-token (copy-tree symbol))
      (advance-token)
      t)))
```

———-

9.0.56 defun parse-identifier

[match-current-token p283]
[token-symbol p??]
[push-reduction p292]
[advance-token p286]

— **defun parse-identifier** —

```
(defun parse-identifier ()
  (let* ((tok (match-current-token 'identifier))
         (symbol (if tok (token-symbol tok))))
    (when tok
      (push-reduction 'identifier-token (copy-tree symbol))
      (advance-token)
      t)))
```

———-

9.0.57 defun parse-number

[match-current-token p283]
[token-symbol p**??**]
[push-reduction p292]
[advance-token p286]

— **defun parse-number** —

```
(defun parse-number ()
  (let* ((tok (match-current-token 'number))
         (symbol (if tok (token-symbol tok))))
    (when tok
      (push-reduction 'number-token (copy-tree symbol))
      (advance-token)
      t)))
```

———-

9.0.58 defun parse-keyword

[match-current-token p283]
[token-symbol p**??**]
[push-reduction p292]
[advance-token p286]

— **defun parse-keyword** —

```
(defun parse-keyword ()
  (let* ((tok (match-current-token 'keyword))
         (symbol (if tok (token-symbol tok))))
    (when tok
      (push-reduction 'keyword-token (copy-tree symbol))
      (advance-token)
      t)))
```

———-

9.0.59 defun parse-argument-designator

[push-reduction p292]
[match-current-token p283]
[token-symbol p??]
[advance-token p286]

— **defun parse-argument-designator** —

```
(defun parse-argument-designator ()
  (let* ((tok (match-current-token 'argument-designator))
         (symbol (if tok (token-symbol tok))))
    (when tok
      (push-reduction 'argument-designator-token (copy-tree symbol))
      (advance-token)
      t)))
```

———-

9.0.60 defun print-package

— **defun print-package** —

```
(defun print-package (package)
  (format out-stream "~&~%(IN-PACKAGE ~S )~%~%" package))
```

———-

9.0.61 defun checkWarning

[postError p196]
[concat p??]

— **defun checkWarning** —

```
(defun |checkWarning| (msg)
 (|postError| (|concat| "Parsing error: " msg)))
```

———-

9.0.62 defun tuple2List

[tuple2List p302]
[postTranSegment p209]
[postTran p192]
[$boot p??]
[$InteractiveMode p??]

— **defun tuple2List** —

```
(defun |tuple2List| (arg)
 (let (u p q)
 (declare (special |$InteractiveMode| $boot))
  (when (pairp arg)
   (setq u (|tuple2List| (qcdr arg)))
   (cond
    ((and (pairp (qcar arg)) (eq (qcar (qcar arg)) 'segment)
          (pairp (qcdr (qcar arg)))
          (pairp (qcdr (qcdr (qcar arg))))
          (eq (qcdr (qcdr (qcdr (qcar arg)))) nil))
       (setq p (qcar (qcdr (qcar arg))))
       (setq q (qcar (qcdr (qcdr (qcar arg)))))
       (cond
        ((null u) (list '|construct| (|postTranSegment| p q)))
        ((and |$InteractiveMode| (null $boot))
          (cons '|append|
           (cons (list '|construct| (|postTranSegment| p q))
                 (list (|tuple2List| (qcdr arg))))))
         (t
          (cons '|nconc|
           (cons (list '|construct| (|postTranSegment| p q))
                 (list (|tuple2List| (qcdr arg))))))))
    ((null u) (list '|construct|  (|postTran| (qcar arg))))
    (t (list '|cons| (|postTran| (qcar arg)) (|tuple2List| (qcdr arg)))))))
```

———-

9.0.63 defmacro pop-stack-1

[reduction-value p??]
[Pop-Reduction p304]

— **defmacro pop-stack-1** —

```
(defmacro pop-stack-1 () '(reduction-value (Pop-Reduction)))
```

———-

9.0.64 defmacro pop-stack-2

[stack-push p98]
[reduction-value p??]
[Pop-Reduction p304]

— **defmacro pop-stack-2** —

```
(defmacro pop-stack-2 ()
  `(let* ((top (Pop-Reduction)) (next (Pop-Reduction)))
     (stack-push top Reduce-Stack)
     (reduction-value next)))
```

————-

9.0.65 defmacro pop-stack-3

[stack-push p98]
[reduction-value p??]
[Pop-Reduction p304]

— **defmacro pop-stack-3** —

```
(defmacro pop-stack-3 ()
  `(let* ((top (Pop-Reduction)) (next (Pop-Reduction)) (nnext (Pop-Reduction)))
     (stack-push next Reduce-Stack)
     (stack-push top Reduce-Stack)
     (reduction-value nnext)))
```

————-

9.0.66 defmacro pop-stack-4

[stack-push p98]
[reduction-value p??]
[Pop-Reduction p304]

— **defmacro pop-stack-4** —

```
(defmacro pop-stack-4 ()
  `(let* ((top (Pop-Reduction))
          (next (Pop-Reduction))
          (nnext (Pop-Reduction))
          (nnnext (Pop-Reduction)))
```

```
    (stack-push nnext Reduce-Stack)
    (stack-push next Reduce-Stack)
    (stack-push top Reduce-Stack)
    (reduction-value nnnext)))
```

———-

9.0.67 defmacro nth-stack

[stack-store p**??**]
[reduction-value p**??**]

— **defmacro nth-stack** —

```
(defmacro nth-stack (x)
  `(reduction-value (nth (1- ,x) (stack-store Reduce-Stack))))
```

———-

9.0.68 defun Pop-Reduction

[stack-pop p98]

— **defun Pop-Reduction** —

```
(defun Pop-Reduction () (stack-pop Reduce-Stack))
```

———-

9.0.69 defun addclose

[suffix p**??**]

— **defun addclose** —

```
(defun addclose (line char)
 (cond
  ((char= (char line (maxindex line)) #\; )
    (setelt line (maxindex line) char)
    (if (char= char #\;) line (suffix #\; line)))
  ((suffix char line))))
```

———-

9.0.70 defun blankp

— **defun blankp** —

```
(defun blankp (char)
 (or (eq char #\Space) (eq char #\tab)))
```

————-

9.0.71 defun drop

Return a pointer to the Nth cons of X, counting 0 as the first cons. [drop p305]
[take p**??**]
[croak p**??**]

— **defun drop** —

```
(defun drop (n x &aux m)
 (cond
  ((eql n 0) x)
  ((> n 0) (drop (1- n) (cdr x)))
  ((>= (setq m (+ (length x) n)) 0) (take m x))
  ((croak (list "Bad args to DROP" n x)))))
```

————-

9.0.72 defun escaped

— **defun escaped** —

```
(defun escaped (str n)
 (and (> n 0) (eq (char str (1- n)) #\_)))
```

————-

9.0.73 defvar $comblocklist

— **initvars** —

```
(defvar $comblocklist nil "a dynamic lists of comments for this block")
```

———-

9.0.74 defun fincomblock

- NUM is the line number of the current line
- OLDNUMS is the list of line numbers of previous lines
- OLDLOCS is the list of previous indentation locations
- NCBLOCK is the current comment block

[preparse-echo p90]
[$comblocklist p305]
[$EchoLineStack p**??**]

— **defun fincomblock** —

```
(defun fincomblock (num oldnums oldlocs ncblock linelist)
 (declare (special $EchoLineStack $comblocklist))
 (push
  (cond
   ((eql (car ncblock) 0) (cons (1- num) (reverse (cdr ncblock))))
    ;; comment for constructor itself paired with 1st line -1
   (t
    (when $EchoLineStack
     (setq num (pop $EchoLineStack))
     (preparse-echo linelist)
     (setq $EchoLineStack (list num)))
    (cons            ;; scan backwards for line to left of current
     (do ((onums oldnums (cdr onums))
          (olocs oldlocs (cdr olocs))
          (sloc (car ncblock)))
         ((null onums) nil)
       (when (and (numberp (car olocs)) (<= (car olocs) sloc))
         (return (car onums))))
     (reverse (cdr ncblock)))))
   $comblocklist))
```

———-

9.0.75 defun indent-pos

— **defun indent-pos** —

```
(defun indent-pos (str)
  (do ((i 0 (1+ i)) (pos 0))
      ((>= i (length str)) nil)
   (case (char str i)
    (#\space (incf pos))
    (#\tab (setq pos (next-tab-loc pos)))
    (otherwise (return pos)))))
```

———-

9.0.76 defun infixtok

[string2id-n p??]

— **defun infixtok** —

```
(defun infixtok (s)
 (member (string2id-n s 1) '(|then| |else|) :test #'eq))
```

———-

9.0.77 defun is-console

[fp-output-stream p??]
[*terminal-io* p??]

— **defun is-console** —

```
(defun is-console (stream)
  (and (streamp stream) (output-stream-p stream)
       (eq (system:fp-output-stream stream)
           (system:fp-output-stream *terminal-io*))))
```

———-

9.0.78 defun next-tab-loc

— **defun next-tab-loc** —

```
(defun next-tab-loc (i)
 (* (1+ (truncate i 8)) 8))
```

———-

9.0.79 defun nonblankloc

[blankp p305]

— **defun nonblankloc** —

```
(defun nonblankloc (str)
 (position-if-not #'blankp str))
```

————-

9.0.80 defun parseprint

— **defun parseprint** —

```
(defun parseprint (l)
 (when l
  (format t "~&~%       ***       PREPARSE      ***~%~%")
  (dolist (x l) (format t "~5d. ~a~%" (car x) (cdr x)))
  (format t "~%")))
```

————-

9.0.81 defun skip-to-endif

[initial-substring p96]
[preparseReadLine p88]
[preparseReadLine1 p89]
[skip-to-endif p308]

— **defun skip-to-endif** —

```
(defun skip-to-endif (x)
 (let (line ind)
  (dcq (ind . line) (preparseReadLine1))
  (cond
   ((not (stringp line)) (cons ind line))
   ((initial-substring line ")endif") (preparseReadLine x))
   ((initial-substring line ")fin") (cons ind nil))
   (t (skip-to-endif x)))))
```

————-

Chapter 10

The Compiler

10.1 Compiling EQ.spad

Given the top level command:

```
)co EQ
```

The default call chain looks like:

```
1> (|compiler| ...)
 2> (|compileSpad2Cmd| ...)
   Compiling AXIOM source code from file /tmp/A.spad using old system
      compiler.
  3> (|compilerDoit| ...)
   4> (|/RQ,LIB|)
    5> (/RF-1 ...)
     6> (SPAD ...)
   AXSERV abbreviates package AxiomServer
      7> (S-PROCESS ...)
       8> (|compTopLevel| ...)
        9> (|compOrCroak| ...)
         10> (|compOrCroak1| ...)
          11> (|comp| ...)
           12> (|compNoStacking| ...)
            13> (|comp2| ...)
             14> (|comp3| ...)
              15> (|compExpression| ...)
*              16> (|compWhere| ...)
                17> (|comp| ...)
                 18> (|compNoStacking| ...)
                  19> (|comp2| ...)
                   20> (|comp3| ...)
                    21> (|compExpression| ...)
```

```
22> (|compSeq| ...)
 23> (|compSeq1| ...)
  24> (|compSeqItem| ...)
   25> (|comp| ...)
    26> (|compNoStacking| ...)
     27> (|comp2| ...)
      28> (|comp3| ...)
       29> (|compExpression| ...)
       <29 (|compExpression| ...)
      <28 (|comp3| ...)
     <27 (|comp2| ...)
    <26 (|compNoStacking| ...)
   <25 (|comp| ...)
  <24 (|compSeqItem| ...)
  24> (|compSeqItem| ...)
   25> (|comp| ...)
    26> (|compNoStacking| ...)
     27> (|comp2| ...)
      28> (|comp3| ...)
       29> (|compExpression| ...)
        30> (|compExit| ...)
         31> (|comp| ...)
          32> (|compNoStacking| ...)
           33> (|comp2| ...)
            34> (|comp3| ...)
             35> (|compExpression| ...)
             <35 (|compExpression| ...)
            <34 (|comp3| ...)
           <33 (|comp2| ...)
          <32 (|compNoStacking| ...)
         <31 (|comp| ...)
         31> (|modifyModeStack| ...)
         <31 (|modifyModeStack| ...)
        <30 (|compExit| ...)
       <29 (|compExpression| ...)
      <28 (|comp3| ...)
     <27 (|comp2| ...)
    <26 (|compNoStacking| ...)
   <25 (|comp| ...)
  <24 (|compSeqItem| ...)
  24> (|replaceExitEtc| ...)
   25> (|replaceExitEtc,fn| ...)
    26> (|replaceExitEtc| ...)
     27> (|replaceExitEtc,fn| ...)
      28> (|replaceExitEtc| ...)
       29> (|replaceExitEtc,fn| ...)
       <29 (|replaceExitEtc,fn| ...)
      <28 (|replaceExitEtc| ...)
      28> (|replaceExitEtc| ...)
       29> (|replaceExitEtc,fn| ...)
```

```
                <29 (|replaceExitEtc,fn| ...)
               <28 (|replaceExitEtc| ...)
              <27 (|replaceExitEtc,fn| ...)
             <26 (|replaceExitEtc| ...)
             26> (|replaceExitEtc| ...)
              27> (|replaceExitEtc,fn| ...)
               28> (|replaceExitEtc| ...)
                29> (|replaceExitEtc,fn| ...)
                 30> (|replaceExitEtc| ...)
                  31> (|replaceExitEtc,fn| ...)
                   32> (|replaceExitEtc| ...)
                    33> (|replaceExitEtc,fn| ...)
                    <33 (|replaceExitEtc,fn| ...)
                   <32 (|replaceExitEtc| ...)
                   32> (|replaceExitEtc| ...)
                    33> (|replaceExitEtc,fn| ...)
                    <33 (|replaceExitEtc,fn| ...)
                   <32 (|replaceExitEtc| ...)
                  <31 (|replaceExitEtc,fn| ...)
                 <30 (|replaceExitEtc| ...)
                 30> (|convertOrCroak| ...)
                  31> (|convert| ...)
                  <31 (|convert| ...)
                 <30 (|convertOrCroak| ...)
                <29 (|replaceExitEtc,fn| ...)
               <28 (|replaceExitEtc| ...)
               28> (|replaceExitEtc| ...)
                29> (|replaceExitEtc,fn| ...)
                <29 (|replaceExitEtc,fn| ...)
               <28 (|replaceExitEtc| ...)
              <27 (|replaceExitEtc,fn| ...)
             <26 (|replaceExitEtc| ...)
            <25 (|replaceExitEtc,fn| ...)
           <24 (|replaceExitEtc| ...)
          <23 (|compSeq1| ...)
         <22 (|compSeq| ...)
        <21 (|compExpression| ...)
       <20 (|comp3| ...)
      <19 (|comp2| ...)
     <18 (|compNoStacking| ...)
    <17 (|comp| ...)
    17> (|comp| ...)
     18> (|compNoStacking| ...)
      19> (|comp2| ...)
       20> (|comp3| ...)
        21> (|compExpression| ...)
         22> (|comp| ...)
          23> (|compNoStacking| ...)
           24> (|comp2| ...)
            25> (|comp3| ...)
```

```
                    26> (|compColon| ...)
                    <26 (|compColon| ...)
                   <25 (|comp3| ...)
                  <24 (|comp2| ...)
                 <23 (|compNoStacking| ...)
                <22 (|comp| ...)
```

In order to explain the compiler we will walk through the compilation of EQ.spad, which handles equations as mathematical objects. We start the system. Most of the structure in Axiom are circular so we have to the `*print-cycle*` to true.

```
root@spiff:/tmp# axiom -nox

(1) -> )lisp (setq *print-circle* t)

Value = T
```

We trace the function we find interesting:

```
(1) -> )lisp (trace |compiler|)

Value = (|compiler|)
```

10.1.1 The top level compiler command

We compile the spad file. We can see that the **compiler** function gets a list

```
(1) -> )co EQ

  1> (|compiler| (EQ))
```

In order to find this file, the **pathname** and **pathnameType** functions are used to find the location and pathname to the file. They **pathnameType** function eventually returns the fact that this is a spad source file. Once that is known we call the **compileSpad2Cmd** function with a list containing the full pathname as a string.

```
  1> (|compiler| (EQ))
    2> (|pathname| (EQ))
    <2 (|pathname| #p"EQ")
    2> (|pathnameType| #p"EQ")
      3> (|pathname| #p"EQ")
      <3 (|pathname| #p"EQ")
    <2 (|pathnameType| NIL)
    2> (|pathnameType| "/tmp/EQ.spad")
      3> (|pathname| "/tmp/EQ.spad")
      <3 (|pathname| #p"/tmp/EQ.spad")
    <2 (|pathnameType| "spad")
```

```
    2> (|pathnameType| "/tmp/EQ.spad")
      3> (|pathname| "/tmp/EQ.spad")
      <3 (|pathname| #p"/tmp/EQ.spad")
    <2 (|pathnameType| "spad")
    2> (|pathnameType| "/tmp/EQ.spad")
      3> (|pathname| "/tmp/EQ.spad")
      <3 (|pathname| #p"/tmp/EQ.spad")
    <2 (|pathnameType| "spad")
    2> (|compileSpad2Cmd| ("/tmp/EQ.spad"))
```

[compiler helpSpad2Cmd (vol5)]
[compiler selectOptionLC (vol5)]
[compiler pathname (vol5)]
[compiler mergePathnames (vol5)]
[compiler pathnameType (vol5)]
[compiler namestring (vol5)]
[throwKeyedMsg p**??**]
[findfile p**??**]
[compileSpad2Cmd p314]
[compileSpadLispCmd p365]
[$newConlist p**??**]
[$options p**??**]
[/editfile p**??**]

— defun compiler —

```
(defun |compiler| (args)
 "The top level compiler command"
 (let (|$newConlist| optlist optname optargs havenew haveold aft ef af af1)
  (declare (special |$newConlist| |$options| /editfile))
  (setq |$newConlist| nil)
  (cond
   ((and (null args) (null |$options|) (null /editfile))
     (|helpSpad2Cmd| '(|compiler|)))
   (t
    (cond ((null args) (setq args (cons /editfile nil))))
    (setq optlist '(|new| |old| |translate| |constructor|))
    (setq havenew nil)
    (setq haveold nil)
    (do ((t0 |$options| (cdr t0)) (opt nil))
        ((or (atom t0)
             (progn (setq opt (car t0)) nil)
             (null (null (and havenew haveold))))
          nil)
     (setq optname (car opt))
     (setq optargs (cdr opt))
     (case (|selectOptionLC| optname optlist nil)
      (|new|         (setq havenew t))
      (|translate|   (setq haveold t))
```

```
  (|constructor| (setq haveold t))
  (|old|         (setq haveold t))))
(cond
 ((and havenew haveold) (|throwKeyedMsg| 's2iz0081 nil))
 (t
  (setq af (|pathname| args))
  (setq aft (|pathnameType| af))
  (cond
   ((or haveold (string= aft "spad"))
    (if (null (setq af1 ($findfile af '(|spad|))))
       (|throwKeyedMsg| 's2il0003 (cons (namestring af) nil))
       (|compileSpad2Cmd| (cons af1 nil))))
   ((string= aft "nrlib")
    (if (null (setq af1 ($findfile af '(|nrlib|))))
      (|throwKeyedMsg| 'S2IL0003 (cons (namestring af) nil))
      (|compileSpadLispCmd| (cons af1 nil))))
   (t
    (setq af1 ($findfile af '(|spad|)))
    (cond
     ((and af1 (string= (|pathnameType| af1) "spad"))
      (|compileSpad2Cmd| (cons af1 nil)))
     (t
      (setq ef (|pathname| /editfile))
      (setq ef (|mergePathnames| af ef))
      (cond
       ((boot-equal ef af) (|throwKeyedMsg| 's2iz0039 nil))
       (t
        (setq af ef)
        (cond
         ((string= (|pathnameType| af) "spad")
          (|compileSpad2Cmd| args))
         (t
          (setq af1 ($findfile af '(|spad|)))
          (cond
           ((and af1 (string= (|pathnameType| af1) "spad"))
             (|compileSpad2Cmd| (cons af1 nil)))
           (t (|throwKeyedMsg| 's2iz0039 nil)))))))))))))))
```

———-

10.1.2 The Spad compiler top level function

The argument to this function, as noted above, is a list containing the string pathname to the file.

```
2> (|compileSpad2Cmd| ("/tmp/EQ.spad"))
```

There is a fair bit of redundant work to find the full filename and pathname of the file. This needs to be eliminated.

The trace of the functions in this routines is:

```
1> (|selectOptionLC| "compiler" (|abbreviations| |boot| |browse| |cd| |clear| |close| |compiler| |copy
<1 (|selectOptionLC| |compiler|)
1> (|selectOptionLC| |compiler| (|abbreviations| |boot| |browse| |cd| |clear| |close| |compiler| |copy
<1 (|selectOptionLC| |compiler|)
1> (|pathname| (EQ))
<1 (|pathname| #p"EQ")
1> (|pathnameType| #p"EQ")
  2> (|pathname| #p"EQ")
  <2 (|pathname| #p"EQ")
<1 (|pathnameType| NIL)
1> (|pathnameType| "/tmp/EQ.spad")
  2> (|pathname| "/tmp/EQ.spad")
  <2 (|pathname| #p"/tmp/EQ.spad")
<1 (|pathnameType| "spad")
1> (|pathnameType| "/tmp/EQ.spad")
  2> (|pathname| "/tmp/EQ.spad")
  <2 (|pathname| #p"/tmp/EQ.spad")
<1 (|pathnameType| "spad")
1> (|pathnameType| "/tmp/EQ.spad")
  2> (|pathname| "/tmp/EQ.spad")
  <2 (|pathname| #p"/tmp/EQ.spad")
<1 (|pathnameType| "spad")
1> (|compileSpad2Cmd| ("/tmp/EQ.spad"))
  2> (|pathname| ("/tmp/EQ.spad"))
  <2 (|pathname| #p"/tmp/EQ.spad")
  2> (|pathnameType| #p"/tmp/EQ.spad")
    3> (|pathname| #p"/tmp/EQ.spad")
    <3 (|pathname| #p"/tmp/EQ.spad")
  <2 (|pathnameType| "spad")
  2> (|updateSourceFiles| #p"/tmp/EQ.spad")
    3> (|pathname| #p"/tmp/EQ.spad")
    <3 (|pathname| #p"/tmp/EQ.spad")
    3> (|pathname| #p"/tmp/EQ.spad")
    <3 (|pathname| #p"/tmp/EQ.spad")
    3> (|pathnameType| #p"/tmp/EQ.spad")
      4> (|pathname| #p"/tmp/EQ.spad")
      <4 (|pathname| #p"/tmp/EQ.spad")
    <3 (|pathnameType| "spad")
    3> (|pathname| ("EQ" "spad" "*"))
    <3 (|pathname| #p"EQ.spad")
    3> (|pathnameType| #p"EQ.spad")
      4> (|pathname| #p"EQ.spad")
      <4 (|pathname| #p"EQ.spad")
    <3 (|pathnameType| "spad")
  <2 (|updateSourceFiles| #p"EQ.spad")
```

```
    2> (|namestring| ("/tmp/EQ.spad"))
      3> (|pathname| ("/tmp/EQ.spad"))
      <3 (|pathname| #p"/tmp/EQ.spad")
    <2 (|namestring| "/tmp/EQ.spad")
   Compiling AXIOM source code from file /tmp/EQ.spad using old system
      compiler.
```

Again we find a lot of redundant work. We finally end up calling **compilerDoit** with a constructed argument list:

```
    2> (|compilerDoit| NIL (|rq| |lib|))
```

[compileSpad2Cmd pathname (vol5)]
[compileSpad2Cmd pathnameType (vol5)]
[compileSpad2Cmd namestring (vol5)]
[compileSpad2Cmd updateSourceFiles (vol5)]
[compileSpad2Cmd selectOptionLC (vol5)]
[compileSpad2Cmd terminateSystemCommand (vol5)]
[nequal p**??**]
[throwKeyedMsg p**??**]
[compileSpad2Cmd sayKeyedMsg (vol5)]
[error p**??**]
[strconc p**??**]
[object2String p**??**]
[browserAutoloadOnceTrigger p**??**]
[spad2AsTranslatorAutoloadOnceTrigger p**??**]
[convertSpadToAsFile p**??**]
[compilerDoitWithScreenedLisplib p**??**]
[compilerDoit p318]
[extendLocalLibdb p**??**]
[spadPrompt p**??**]
[$newComp p**??**]
[$scanIfTrue p**??**]
[$compileOnlyCertainItems p**??**]
[$f p**??**]
[$m p**??**]
[$QuickLet p**??**]
[$QuickCode p**??**]
[$sourceFileTypes p**??**]
[$InteractiveMode p**??**]
[$options p**??**]
[$newConlist p**??**]
[/editfile p**??**]

— defun compileSpad2Cmd —

```
(defun |compileSpad2Cmd| (args)
```

```
(let (|$newComp| |$scanIfTrue|
      |$compileOnlyCertainItems| |$f| |$m| |$QuickLet| |$QuickCode|
      |$sourceFileTypes| |$InteractiveMode| path optlist fun optname
      optargs fullopt constructor)
 (declare (special |$newComp| |$scanIfTrue|
      |$compileOnlyCertainItems| |$f| |$m| |$QuickLet| |$QuickCode|
      |$sourceFileTypes| |$InteractiveMode| /editfile |$options|
      |$newConlist|))
  (setq path (|pathname| args))
  (cond
   ((nequal (|pathnameType| path) "spad") (|throwKeyedMsg| 's2iz0082 nil))
   ((null (probe-file path))
    (|throwKeyedMsg| 's2il0003 (cons (|namestring| args) nil)))
   (t
    (setq /editfile path)
    (|updateSourceFiles| path)
    (|sayKeyedMsg| 's2iz0038 (list (|namestring| args)))
    (setq optlist '(|break| |constructor| |functions| |library| |lisp|
        |new| |old| |nobreak| |nolibrary| |noquiet| |vartrace| |quiet|
        |translate|))
    (setq |$QuickLet| t)
    (setq |$QuickCode| t)
    (setq fun '(|rq| |lib|))
    (setq |$sourceFileTypes| '("SPAD"))
    (dolist (opt |$options|)
     (setq optname (car opt))
     (setq optargs (cdr opt))
     (setq fullopt (|selectOptionLC| optname optlist nil))
     (case fullopt
      (|old| nil)
      (|library| (setelt fun 1 '|lib|))
      (|nolibrary| (setelt fun 1 '|nolib|))
      (|quiet| (when (nequal (elt fun 0) '|c|) (setelt fun 0 '|rq|)))
      (|noquiet| (when (nequal (elt fun 0) '|c|) (setelt fun 0 '|rf|)))
      (|nobreak| (setq |$scanIfTrue| t))
      (|break| (setq |$scanIfTrue| nil))
      (|vartrace| (setq |$QuickLet| nil))
      (|lisp| (|throwKeyedMsg| 's2iz0036 (list ")lisp")))
      (|functions|
       (if (null optargs)
        (|throwKeyedMsg| 's2iz0037 (list ")functions"))
        (setq |$compileOnlyCertainItems| optargs)))
      (|constructor|
       (if (null optargs)
        (|throwKeyedMsg| 's2iz0037 (list ")constructor"))
        (progn
         (setelt fun 0 '|c|)
         (setq constructor (mapcar #'|unabbrev| optargs)))))
      (t
       (|throwKeyedMsg| 's2iz0036
```

```
        (list (strconc ")" (|object2String| optname)))))))
    (setq |$InteractiveMode| nil)
    (cond
     (|$compileOnlyCertainItems|
      (if (null constructor)
       (|sayKeyedMsg| 's2iz0040 nil)
       (|compilerDoitWithScreenedLisplib| constructor fun)))
     (t (|compilerDoit| constructor fun)))
    (|extendLocalLibdb| |$newConlist|)
    (|terminateSystemCommand|)
    (|spadPrompt|)))))
```

This trivial function cases on the second argument to decide which combination of operations was requested. For this case we see:

```
(1) -> )co EQ
   Compiling AXIOM source code from file /tmp/EQ.spad using old system
      compiler.
  1> (|compilerDoit| NIL (|rq| |lib|))
    2> (|/RQ,LIB|)

... [snip]...

    <2 (|/RQ,LIB| T)
  <1 (|compilerDoit| T)
(1) ->
```

10.1.3 defun compilerDoit

[compilerDoit /rq (vol5)]
[compilerDoit /rf (vol5)]
[compilerDoit member (vol5)]
[sayBrightly p**??**]
[opOf p**??**]
[/RQ,LIB p319]
[$byConstructors p368]
[$constructorsSeen p368]

— defun compilerDoit —

```
(defun |compilerDoit| (constructor fun)
 (let (|$byConstructors| |$constructorsSeen|)
 (declare (special |$byConstructors| |$constructorsSeen|))
  (cond
   ((equal fun '(|rf| |lib|))   (|/RQ,LIB|))   ; Ignore "noquiet"
```

```
   ((equal fun '(|rf| |nolib|)) (/rf))
   ((equal fun '(|rq| |lib|))   (|/RQ,LIB|))
   ((equal fun '(|rq| |nolib|)) (/rq))
   ((equal fun '(|c| |lib|))
    (setq |$byConstructors| (loop for x in constructor collect (|opOf| x)))
    (|/RQ,LIB|)
    (dolist (x |$byConstructors|)
     (unless (|member| x |$constructorsSeen|)
      (|sayBrightly| `(">>> Warning " |%b| ,x |%d| " was not found")))))))
```

———-

This function simply calls `/rf-1`.

```
(2) -> )co EQ
   Compiling AXIOM source code from file /tmp/EQ.spad using old system
      compiler.
  1> (|compilerDoit| NIL (|rq| |lib|))
    2> (|/RQ,LIB|)
      3> (/RF-1 NIL)
...[snip]...
      <3 (/RF-1 T)
    <2 (|/RQ,LIB| T)
  <1 (|compilerDoit| T)
```

10.1.4 defun /RQ,LIB

[/rf-1 p320]
[/RQ,LIB echo-meta (vol5)]
[$lisplib p??]

— **defun /RQ,LIB** —

```
(defun |/RQ,LIB| (&rest foo &aux (echo-meta nil) ($lisplib t))
 (declare (special echo-meta $lisplib) (ignore foo))
  (/rf-1 nil))
```

———-

Since this function is called with nil we fall directly into the call to the function **spad**:

```
(2) -> )co EQ
   Compiling AXIOM source code from file /tmp/EQ.spad using old system
      compiler.
  1> (|compilerDoit| NIL (|rq| |lib|))
```

```
    2> (|/RQ,LIB|)
      3> (/RF-1 NIL)
        4> (SPAD "/tmp/EQ.spad")
...[snip]...
        <4 (SPAD T)
      <3 (/RF-1 T)
    <2 (|/RQ,LIB| T)
  <1 (|compilerDoit| T)
```

10.1.5 defun /rf-1

[/rf-1 makeInputFilename (vol5)]
[ncINTERPFILE p365]
[/rf-1 spad (vol5)]
[/editfile p**??**]
[echo-meta p**??**]

— **defun /rf-1** —

```
(defun /rf-1 (ignore)
 (declare (ignore ignore))
 (let* ((input-file (makeInputFilename /editfile))
        (type (pathname-type input-file)))
 (declare (special echo-meta /editfile))
 (cond
  ((string= type "lisp") (load input-file))
  ((string= type "input") (|ncINTERPFILE| input-file echo-meta))
  (t (spad input-file)))))
```

———-

Here we begin the actual compilation process.

```
  1> (SPAD "/tmp/EQ.spad")
    2> (|makeInitialModemapFrame|)
    <2 (|makeInitialModemapFrame| ((NIL)))
    2> (INIT-BOOT/SPAD-READER)
    <2 (INIT-BOOT/SPAD-READER NIL)
    2> (OPEN "/tmp/EQ.spad" :DIRECTION :INPUT)
    <2 (OPEN #<input stream "/tmp/EQ.spad">)
    2> (INITIALIZE-PREPARSE #<input stream "/tmp/EQ.spad">)
    <2 (INITIALIZE-PREPARSE ")abbrev domain EQ Equation")
    2> (PREPARSE #<input stream "/tmp/EQ.spad">)
   EQ abbreviates domain Equation
    <2 (PREPARSE (# # # # # # # # ...))
    2> (|PARSE-NewExpr|)
    <2 (|PARSE-NewExpr| T)
```

```
    2> (S-PROCESS (|where| # #))
...[snip]...
      3> (OPEN "/tmp/EQ.erlib/info" :DIRECTION :OUTPUT)
      <3 (OPEN #<output stream "/tmp/EQ.erlib/info">)
      3> (OPEN #p"/tmp/EQ.nrlib/EQ.lsp")
      <3 (OPEN #<input stream "/tmp/EQ.nrlib/EQ.lsp">)
      3> (OPEN #p"/tmp/EQ.nrlib/EQ.data" :DIRECTION :OUTPUT)
      <3 (OPEN #<output stream "/tmp/EQ.nrlib/EQ.data">)
      3> (OPEN #p"/tmp/EQ.nrlib/EQ.c" :DIRECTION :OUTPUT)
      <3 (OPEN #<output stream "/tmp/EQ.nrlib/EQ.c">)
      3> (OPEN #p"/tmp/EQ.nrlib/EQ.h" :DIRECTION :OUTPUT)
      <3 (OPEN #<output stream "/tmp/EQ.nrlib/EQ.h">)
      3> (OPEN #p"/tmp/EQ.nrlib/EQ.fn" :DIRECTION :OUTPUT)
      <3 (OPEN #<output stream "/tmp/EQ.nrlib/EQ.fn">)
      3> (OPEN #p"/tmp/EQ.nrlib/EQ.o" :DIRECTION :OUTPUT :IF-EXISTS :APPEND)
      <3 (OPEN #<output stream "/tmp/EQ.nrlib/EQ.o">)
      3> (OPEN #p"/tmp/EQ.nrlib/EQ.data")
      <3 (OPEN #<input stream "/tmp/EQ.nrlib/EQ.data">)
      3> (OPEN "/tmp/EQ.nrlib/index.kaf")
      <3 (OPEN #<input stream "/tmp/EQ.nrlib/index.kaf">)
    <2 (S-PROCESS NIL)
  <1 (SPAD T)
  1> (OPEN "temp.text" :DIRECTION :OUTPUT)
  <1 (OPEN #<output stream "temp.text">)
  1> (OPEN "libdb.text")
  <1 (OPEN #<input stream "libdb.text">)
  1> (OPEN "temp.text")
  <1 (OPEN #<input stream "temp.text">)
  1> (OPEN "libdb.text" :DIRECTION :OUTPUT)
  <1 (OPEN #<output stream "libdb.text">)
```

The major steps in this process involve the **preparse** function. (See book volume 5 for more details). The **preparse** function returns a list of pairs of the form: ((linenumber . linestring) .... (linenumber . linestring)) For instance, for the file `EQ.spad`, we get:

```
    <2 (PREPARSE (
 (19 . "Equation(S: Type): public == private where")
 (20 . " (Ex ==> OutputForm;")
 (21 . "  public ==> Type with")
 (22 . "   (\"=\": (S, S) -> $;")
...[skip]...
 (202 . "        inv eq == [inv lhs eq, inv rhs eq]);")
 (203 . "    if S has ExpressionSpace then")
 (204 . "        subst(eq1,eq2) ==")
 (205 . "           (eq3 := eq2 pretend Equation S;")
 (206 . "            [subst(lhs eq1,eq3),subst(rhs eq1,eq3)])))")))
```

And the **s-process** function which returns a parsed version of the input.

```
    2> (S-PROCESS
(|where|
 (== (|:| (|Equation| (|:| S |Type|)) |public|) |private|)
 (|;|
  (|;|
   (==> |Ex| |OutputForm|)
   (==> |public|
    (|Join| |Type|
     (|with|
      (CATEGORY
       (|Signature| "=" (-> (|,| S S) $))
       (|Signature| |equation| (-> (|,| S S) $))
       (|Signature| |swap| (-> $ $))
       (|Signature| |lhs| (-> $ S))
       (|Signature| |rhs| (-> $ S))
       (|Signature| |map| (-> (|,| (-> S S) $) $))
       (|if| (|has| S (|InnerEvalable| (|,| |Symbol| S)))
        (|Attribute| (|InnerEvalable| (|,| |Symbol| S)))
        NIL)
       (|if| (|has| S |SetCategory|)
        (CATEGORY
         (|Attribute| |SetCategory|)
         (|Attribute| (|CoercibleTo| |Boolean|))
         (|if| (|has| S (|Evalable| S))
          (CATEGORY
           (|Signature| |eval| (-> (|,| $ $) $))
           (|Signature| |eval| (-> (|,| $ (|List| $)) $)))
          NIL))
        NIL)
       (|if| (|has| S |AbelianSemiGroup|)
        (CATEGORY
         (|Attribute| |AbelianSemiGroup|)
         (|Signature| "+" (-> (|,| S $) $))
         (|Signature| "+" (-> (|,| $ S) $)))
         NIL)
       (|if| (|has| S |AbelianGroup|)
        (CATEGORY
         (|Attribute| |AbelianGroup|)
         (|Signature| |leftZero| (-> $ $))
         (|Signature| |rightZero| (-> $ $))
         (|Signature| "-" (-> (|,| S $) $))
         (|Signature| "-" (-> (|,| $ S) $))) NIL)
       (|if| (|has| S |SemiGroup|)
        (CATEGORY
         (|Attribute| |SemiGroup|)
         (|Signature| "*" (-> (|,| S $) $))
         (|Signature| "*" (-> (|,| $ S) $)))
        NIL)
       (|if| (|has| S |Monoid|)
        (CATEGORY
```

```
        (|Attribute| |Monoid|)
        (|Signature| |leftOne| (-> $ (|Union| (|,| $ "failed"))))
        (|Signature| |rightOne| (-> $ (|Union| (|,| $ "failed")))))
       NIL)
      (|if| (|has| S |Group|)
       (CATEGORY
        (|Attribute| |Group|)
        (|Signature| |leftOne| (-> $ (|Union| (|,| $ "failed"))))
        (|Signature| |rightOne| (-> $ (|Union| (|,| $ "failed")))))
        NIL)
      (|if| (|has| S |Ring|)
       (CATEGORY
        (|Attribute| |Ring|)
        (|Attribute| (|BiModule| (|,| S S))))
       NIL)
      (|if| (|has| S |CommutativeRing|)
       (|Attribute| (|Module| S))
       NIL)
      (|if| (|has| S |IntegralDomain|)
       (|Signature| |factorAndSplit| (-> $ (|List| $)))
       NIL)
      (|if| (|has| S (|PartialDifferentialRing| |Symbol|))
       (|Attribute| (|PartialDifferentialRing| |Symbol|))
       NIL)
      (|if| (|has| S |Field|)
       (CATEGORY
        (|Attribute| (|VectorSpace| S))
        (|Signature| "/" (-> (|,| $ $) $))
        (|Signature| |inv| (-> $ $)))
       NIL)
      (|if| (|has| S |ExpressionSpace|)
       (|Signature| |subst| (-> (|,| $ $) $))
       NIL)
  )))))
 (==> |private|
  (|add|
   (|;|
    (|;|
     (|;|
      (|;|
       (|;|
        (|;|
         (|;|
          (|;|
           (|;|
            (|;|
             (|;|
              (|;|
               (|;|
```

```
(|;|
 (|;|
  (|;|
   (|;|
    (|;|
     (|;|
      (|;|
       (|;|
        (|:=| |Rep|
         (|Record| (|,| (|:| |lhs| S) (|:| |rhs| S))))
        (|,| |eq1| (|:| |eq2| $)))
       (|:| |s| S))
      (|if| (|has| S |IntegralDomain|)
       (==
        (|factorAndSplit| |eq|)
        (|;|
         (=> (|has| S (|:| |factor| (-> S (|Factored| S))))
          (|;|
           (|:=| |eq0| (|rightZero| |eq|))
           (COLLECT
            (IN |rcf| (|factors| (|factor| (|lhs| |eq0|))))
            (|construct|
             (|equation| (|,| (|rcf| |factor|) 0))))))
         (|construct| |eq|)))
        NIL))
     (==
      (= (|:| |l| S) (|:| |r| S))
      (|construct| (|,| |l| |r|))))
    (==
     (|equation| (|,| |l| |r|))
     (|construct| (|,| |l| |r|))))
   (== (|lhs| |eqn|) (|eqn| |lhs|)))
  (== (|rhs| |eqn|) (|eqn| |rhs|)))
 (==
  (|swap| |eqn|)
  (|construct| (|,| (|rhs| |eqn|) (|lhs| |eqn|)))))
(==
 (|map| (|,| |fn| |eqn|))
 (|equation|
  (|,| (|fn| (|eqn| |lhs|)) (|fn| (|eqn| |rhs|))))))
(|if| (|has| S (|InnerEvalable| (|,| |Symbol| S)))
 (|;|
  (|;|
   (|;|
    (|;|
     (|;| (|:| |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|)))))
      NIL))
    (|if| (|has| S (|Evalable| S))
     (|;|
      (==
       (|:| (|eval| (|,| (|:| |eqn1| $) (|:| |eqn2| $))) $)
       (=
        (|eval|
         (|,| (|eqn1| |lhs|) (|pretend| |eqn2| (|Equation| S))))
        (|eval|
         (|,| (|eqn1| |rhs|) (|pretend| |eqn2| (|Equation| S))))))
      (==
       (|:|
        (|eval| (|,| (|:| |eqn1| $) (|:| |leqn2| (|List| $)))) $)
       (=
        (|eval|
         (|,|
          (|eqn1| |lhs|)
          (|pretend| |leqn2| (|List| (|Equation| S)))))
        (|eval|
         (|,|
          (|eqn1| |rhs|)
          (|pretend| |leqn2| (|List| (|Equation| S))))))))
      NIL))
    (|if| (|has| S |SetCategory|)
     (|;|
      (|;|
       (==
        (= |eq1| |eq2|)
        (|and|
         (@ (= (|eq1| |lhs|) (|eq2| |lhs|)) |Boolean|)
         (@ (= (|eq1| |rhs|) (|eq2| |rhs|)) |Boolean|)))
       (==
        (|:| (|coerce| (|:| |eqn| $)) |Ex|)
        (= (|::| (|eqn| |lhs|) |Ex|) (|::| (|eqn| |rhs|) |Ex|))))
       (==
        (|:| (|coerce| (|:| |eqn| $)) |Boolean|)
        (= (|eqn| |lhs|) (|eqn| |rhs|))))
      NIL))
    (|if| (|has| S |AbelianSemiGroup|)
     (|;|
      (|;|
       (==
```

```
        (+ |eq1| |eq2|)
        (=
         (+ (|eq1| |lhs|) (|eq2| |lhs|))
         (+ (|eq1| |rhs|) (|eq2| |rhs|))))
       (== (+ |s| |eq2|) (+ (|construct| (|,| |s| |s|)) |eq2|)))
      (== (+ |eq1| |s|) (+ |eq1| (|construct| (|,| |s| |s|)))))
     NIL))
   (|if| (|has| S |AbelianGroup|)
    (|;|
     (|;|
      (|;|
       (|;|
        (|;|
         (|;|
          (== (- |eq|) (= (- (|lhs| |eq|)) (- (|rhs| |eq|))))
          (== (- |s| |eq2|) (- (|construct| (|,| |s| |s|)) |eq2|)))
         (== (- |eq1| |s|) (- |eq1| (|construct| (|,| |s| |s|)))))
        (== (|leftZero| |eq|) (= 0 (- (|rhs| |eq|) (|lhs| |eq|)))))
       (== (|rightZero| |eq|) (= (- (|lhs| |eq|) (|rhs| |eq|)) 0)))
      (== 0 (|equation| (|,| (|elt| S 0) (|elt| S 0)))))
     (==
      (- |eq1| |eq2|)
      (=
       (- (|eq1| |lhs|) (|eq2| |lhs|))
       (- (|eq1| |rhs|) (|eq2| |rhs|)))))
     NIL))
   (|if| (|has| S |SemiGroup|)
    (|;|
     (|;|
      (|;|
       (==
        (* (|:| |eq1| $) (|:| |eq2| $))
        (=
         (* (|eq1| |lhs|) (|eq2| |lhs|))
         (* (|eq1| |rhs|) (|eq2| |rhs|))))
       (==
        (* (|:| |l| S) (|:| |eqn| $))
        (= (* |l| (|eqn| |lhs|)) (* |l| (|eqn| |rhs|)))))
      (==
       (* (|:| |l| S) (|:| |eqn| $))
       (= (* |l| (|eqn| |lhs|)) (* |l| (|eqn| |rhs|)))))
     (==
      (* (|:| |eqn| $) (|:| |l| S))
      (= (* (|eqn| |lhs|) |l|) (* (|eqn| |rhs|) |l|))))
    NIL))
  (|if| (|has| S |Monoid|)
   (|;|
    (|;|
     (|;|
      (== 1 (|equation| (|,| (|elt| S 1) (|elt| S 1))))
```

```
          (==
           (|recip| |eq|)
           (|;|
            (|;|
             (=> (|case| (|:=| |lh| (|recip| (|lhs| |eq|))) "failed")
                 "failed")
             (=> (|case| (|:=| |rh| (|recip| (|rhs| |eq|))) "failed")
                 "failed"))
            (|construct| (|,| (|::| |lh| S) (|::| |rh| S))))))
         (==
          (|leftOne| |eq|)
          (|;|
           (=> (|case| (|:=| |re| (|recip| (|lhs| |eq|))) "failed")
               "failed")
           (= 1 (* (|rhs| |eq|) |re|)))))
        (==
         (|rightOne| |eq|)
         (|;|
          (=> (|case| (|:=| |re| (|recip| (|rhs| |eq|))) "failed")
              "failed")
          (= (* (|lhs| |eq|) |re|) 1))))
       NIL))
     (|if| (|has| S |Group|)
      (|;|
       (|;|
        (==
         (|inv| |eq|)
         (|construct| (|,| (|inv| (|lhs| |eq|)) (|inv| (|rhs| |eq|)))))
        (== (|leftOne| |eq|) (= 1 (* (|rhs| |eq|) (|inv| (|rhs| |eq|)))))
       (== (|rightOne| |eq|) (= (* (|lhs| |eq|) (|inv| (|rhs| |eq|))) 1)))
      NIL))
    (|if| (|has| S |Ring|)
     (|;|
      (==
       (|characteristic| (|@Tuple|))
       ((|elt| S |characteristic|) (|@Tuple|)))
      (== (* (|:| |i| |Integer|) (|:| |eq| $)) (* (|::| |i| S) |eq|)))
     NIL))
   (|if| (|has| S |IntegralDomain|)
    (==
     (|factorAndSplit| |eq|)
     (|;|
      (|;|
       (=>
        (|has| S (|:| |factor| (-> S (|Factored| S))))
        (|;|
         (|:=| |eq0| (|rightZero| |eq|))
         (COLLECT
          (IN |rcf| (|factors| (|factor| (|lhs| |eq0|))))
          (|construct| (|equation| (|,| (|rcf| |factor|) 0))))))
```

```
        (=>
         (|has| S (|Polynomial| |Integer|))
         (|;|
          (|;|
           (|;|
            (|:=| |eq0| (|rightZero| |eq|))
            (==> MF
             (|MultivariateFactorize|
              (|,|
               (|,| (|,| |Symbol| (|IndexedExponents| |Symbol|)) |Integer|)
              (|Polynomial| |Integer|)))))
           (|:=|
            (|:| |p| (|Polynomial| |Integer|))
            (|pretend| (|lhs| |eq0|) (|Polynomial| |Integer|))))
          (COLLECT
           (IN |rcf| (|factors| ((|elt| MF |factor|) |p|)))
           (|construct|
            (|equation| (|,| (|pretend| (|rcf| |factor|) S) 0)))))))
       (|construct| |eq|)))
     NIL))
   (|if| (|has| S (|PartialDifferentialRing| |Symbol|))
    (==
     (|:| (|differentiate| (|,| (|:| |eq| $) (|:| |sym| |Symbol|))) $)
     (|construct|
      (|,|
       (|differentiate| (|,| (|lhs| |eq|) |sym|))
       (|differentiate| (|,| (|rhs| |eq|) |sym|)))))
    NIL))
  (|if| (|has| S |Field|)
   (|;|
    (|;|
     (== (|dimension| (|@Tuple|)) (|::| 2 |CardinalNumber|))
     (==
      (/ (|:| |eq1| $) (|:| |eq2| $))
      (= (/ (|eq1| |lhs|) (|eq2| |lhs|)) (/ (|eq1| |rhs|) (|eq2| |rhs|)))))
    (==
     (|inv| |eq|)
     (|construct| (|,| (|inv| (|lhs| |eq|)) (|inv| (|rhs| |eq|))))))
   NIL))
 (|if| (|has| S |ExpressionSpace|)
  (==
   (|subst| (|,| |eq1| |eq2|))
   (|;|
    (|:=| |eq3| (|pretend| |eq2| (|Equation| S)))
    (|construct|
     (|,|
      (|subst| (|,| (|lhs| |eq1|) |eq3|))
      (|subst| (|,| (|rhs| |eq1|) |eq3|))))))
  NIL)))))))
```

10.1.6 defun spad

[spad-reader p??]
[spad addBinding (vol5)]
[spad makeInitialModemapFrame (vol5)]
[spad init-boot/spad-reader (vol5)]
[initialize-preparse p73]
[preparse p76]
[PARSE-NewExpr p241]
[pop-stack-1 p302]
[s-process p330]
[ioclear p??]
[spad shut (vol5)]
[$noSubsumption p??]
[$InteractiveFrame p??]
[$InitialDomainsInScope p??]
[$InteractiveMode p??]
[line p92]
[echo-meta p??]
[/editfile p??]
[*comp370-apply* p??]
[*eof* p??]
[file-closed p??]
[spad-reader p??]

— **defun spad** —

```
(defun spad (&optional (*spad-input-file* nil) (*spad-output-file* nil)
             &aux (*comp370-apply* #'print-defun)
                  (*fileactq-apply* #'print-defun)
                 ($spad t) ($boot nil) (optionlist nil) (*eof* nil)
                 (file-closed nil) (/editfile *spad-input-file*)
                (|$noSubsumption| |$noSubsumption|) in-stream out-stream)
  (declare (special echo-meta /editfile *comp370-apply* *eof*
                    file-closed |$noSubsumption| |$InteractiveFrame|
                    |$InteractiveMode| |$InitialDomainsInScope|))
  ;; only rebind |$InteractiveFrame| if compiling
  (progv (if (not |$InteractiveMode|) '(|$InteractiveFrame|))
         (if (not |$InteractiveMode|)
             (list (|addBinding| '|$DomainsInScope|
                    `((fluid . |true|)
                      (special . ,(copy-tree |$InitialDomainsInScope|)))
                   (|addBinding| '|$Information| nil
                      (|makeInitialModemapFrame|)))))
  (init-boot/spad-reader)
  (unwind-protect
    (progn
      (setq in-stream (if *spad-input-file*
```

```
                    (open *spad-input-file* :direction :input)
                     *standard-input*))
     (initialize-preparse in-stream)
     (setq out-stream (if *spad-output-file*
                    (open *spad-output-file* :direction :output)
                     *standard-output*))
     (when *spad-output-file*
        (format out-stream "~&;;; -*- Mode:Lisp; Package:Boot  -*-~%~%")
        (print-package "BOOT"))
     (setq curoutstream out-stream)
     (loop
      (if (or *eof* file-closed) (return nil))
      (catch 'spad_reader
        (if (setq boot-line-stack (preparse in-stream))
            (let ((line (cdar boot-line-stack)))
              (declare (special line))
              (|PARSE-NewExpr|)
              (let ((parseout (pop-stack-1)) )
                (when parseout
                      (let ((*standard-output* out-stream))
                        (s-process parseout))
                      (format out-stream "~&")))
              )))
     (ioclear in-stream out-stream)))
   (if *spad-input-file* (shut in-stream))
   (if *spad-output-file* (shut out-stream)))
 t))
```

———-

10.1.7 defun Interpreter interface to the compiler

[curstrm p??]
[def-rename p333]
[new2OldLisp p298]
[parseTransform p103]
[postTransform p191]
[displayPreCompilationErrors p296]
[prettyprint p??]
[s-process processInteractive (vol5)]
[compTopLevel p334]
[def-process p??]
[displaySemanticErrors p??]
[terpri p??]
[get-internal-run-time p??]
[$Index p??]
[$macroassoc p??]

[$newspad p??]
[$PolyMode p??]
[$EmptyMode p??]
[$compUniquelyIfTrue p??]
[$currentFunction p??]
[$postStack p??]
[$topOp p??]
[$semanticErrorStack p??]
[$warningStack p??]
[$exitMode p??]
[$exitModeStack p??]
[$returnMode p??]
[$leaveMode p??]
[$leaveLevelStack p??]
[$top-level p??]
[$insideFunctorIfTrue p??]
[$insideExpressionIfTrue p??]
[$insideCoerceInteractiveHardIfTrue p??]
[$insideWhereIfTrue p??]
[$insideCategoryIfTrue p??]
[$insideCapsuleFunctionIfTrue p??]
[$form p??]
[$DomainFrame p??]
[$e p??]
[$EmptyEnvironment p??]
[$genFVar p??]
[$genSDVar p??]
[$VariableCount p??]
[$previousTime p??]
[$LocalFrame p??]
[$Translation p??]
[curoutstream p??]

— defun s-process —

```
(defun s-process (x)
 (prog ((|$Index| 0)
        ($macroassoc ())
        ($newspad t)
        (|$PolyMode| |$EmptyMode|)
        (|$compUniquelyIfTrue| nil)
        |$currentFunction|
        (|$postStack| nil)
        |$topOp|
        (|$semanticErrorStack| ())
        (|$warningStack| ())
        (|$exitMode| |$EmptyMode|)
```

```
      (|$exitModeStack| ())
      (|$returnMode| |$EmptyMode|)
      (|$leaveMode| |$EmptyMode|)
      (|$leaveLevelStack| ())
      $top_level |$insideFunctorIfTrue| |$insideExpressionIfTrue|
      |$insideCoerceInteractiveHardIfTrue| |$insideWhereIfTrue|
      |$insideCategoryIfTrue| |$insideCapsuleFunctionIfTrue| |$form|
      (|$DomainFrame| '((NIL)))
      (|$e| |$EmptyEnvironment|)
      (|$genFVar| 0)
      (|$genSDVar| 0)
      (|$VariableCount| 0)
      (|$previousTime| (get-internal-run-time))
      (|$LocalFrame| '((NIL)))
      (curstrm curoutstream) |$s| |$x| |$m| u)
 (declare (special |$Index| $macroassoc $newspad |$PolyMode| |$EmptyMode|
          |$compUniquelyIfTrue| |$currentFunction| |$postStack| |$topOp|
          |$semanticErrorStack| |$warningStack| |$exitMode| |$exitModeStack|
          |$returnMode| |$leaveMode| |$leaveLevelStack| $top_level
          |$insideFunctorIfTrue| |$insideExpressionIfTrue|
          |$insideCoerceInteractiveHardIfTrue| |$insideWhereIfTrue|
          |$insideCategoryIfTrue| |$insideCapsuleFunctionIfTrue| |$form|
          |$DomainFrame| |$e| |$EmptyEnvironment| |$genFVar| |$genSDVar|
          |$VariableCount| |$previousTime| |$LocalFrame|
          curstrm |$s| |$x| |$m| curoutstream $traceflag |$Translation|))
 (setq $traceflag t)
 (if (not x) (return nil))
 (if $boot
   (setq x (def-rename (|new2OldLisp| x)))
   (setq x (|parseTransform| (|postTransform| x))))
 (when |$TranslateOnly| (return (setq |$Translation| x)))
 (when |$postStack| (|displayPreCompilationErrors|) (return nil))
 (when |$PrintOnly|
      (format t "~S   =====>~%" |$currentLine|)
      (return (prettyprint x)))
 (if (not $boot)
  (if |$InteractiveMode|
    (|processInteractive| x nil)
    (when (setq u (|compTopLevel| x |$EmptyMode| |$InteractiveFrame|))
      (setq |$InteractiveFrame| (third u))))
  (def-process x))
 (when |$semanticErrorStack| (|displaySemanticErrors|))
 (terpri)))
```

———-

10.1.8 defun print-defun

[is-console p307]
[print-full p??]
[vmlisp::optionlist p??]
[$PrettyPrint p??]

— **defun print-defun** —

```
(defun print-defun (name body)
 (let* ((sp (assoc 'vmlisp::compiler-output-stream vmlisp::optionlist))
        (st (if sp (cdr sp) *standard-output*)))
 (declare (special vmlisp::optionlist |$PrettyPrint|))
  (when (and (is-console st) (symbolp name) (fboundp name)
             (not (compiled-function-p (symbol-function name))))
    (compile name))
  (when (or |$PrettyPrint| (not (is-console st)))
        (print-full body st) (force-output st))))
```

———-

10.1.9 defun def-rename

[def-rename1 p333]

— **defun def-rename** —

```
(defun def-rename (x)
 (def-rename1 x))
```

———-

10.1.10 defun def-rename1

[def-rename1 p333]

— **defun def-rename1** —

```
(defun def-rename1 (x)
 (cond
  ((symbolp x)
   (let ((y (get x 'rename))) (if y (first y) x)))
  ((and (listp x) x)
   (if (eqcar x 'quote)
```

```
      x
      (cons (def-rename1 (first x)) (def-rename1 (cdr x)))))
  (x)))
```

10.1.11 defun compTopLevel

[newComp p??]
[compOrCroak p335]
[$NRTderivedTargetIfTrue p??]
[$killOptimizeIfTrue p??]
[$forceAdd p??]
[$compTimeSum p??]
[$resolveTimeSum p??]
[$packagesUsed p??]
[$envHashTable p??]

— defun compTopLevel —

```
(defun |compTopLevel| (x m e)
 (let (|$NRTderivedTargetIfTrue| |$killOptimizeIfTrue| |$forceAdd|
       |$compTimeSum| |$resolveTimeSum| |$packagesUsed| |$envHashTable|
        t1 t2 t3 val mode)
 (declare (special |$NRTderivedTargetIfTrue| |$killOptimizeIfTrue|
                   |$forceAdd| |$compTimeSum| |$resolveTimeSum|
                   |$packagesUsed| |$envHashTable| ))
   (setq |$NRTderivedTargetIfTrue| nil)
   (setq |$killOptimizeIfTrue| nil)
   (setq |$forceAdd| nil)
   (setq |$compTimeSum| 0)
   (setq |$resolveTimeSum| 0)
   (setq |$packagesUsed| NIL)
   (setq |$envHashTable| (make-hashtable 'equal))
   (dolist (u (car (car e)))
    (dolist (v (cdr u))
     (hput |$envHashTable| (cons (car u) (cons (car v) nil)) t)))
   (cond
    ((or (and (pairp x) (eq (qcar x) 'def))
         (and (pairp x) (eq (qcar x) '|where|)
              (progn
                (setq t1 (qcdr x))
                (and (pairp t1)
                (progn
                 (setq t2 (qcar t1))
                 (and (pairp t2) (eq (qcar t2) 'def)))))))
      (setq t3 (|compOrCroak| x m e))
```

```
      (setq val (car t3))
      (setq mode (second t3))
      (cons val (cons mode (cons e nil))))
    (t (|compOrCroak| x m e)))))
```

Given:

```
CohenCategory(): Category == SetCategory with

  kind:(CExpr)->Boolean
  operand:(CExpr,Integer)->CExpr
  numberOfOperand:(CExpr)->Integer
  construct:(CExpr,CExpr)->CExpr
```

the resulting call looks like:

```
 (|compOrCroak|
       (DEF (|CohenCategory|)
        ((|Category|))
        (NIL)
        (|Join|
         (|SetCategory|)
         (CATEGORY |package|
          (SIGNATURE |kind| ((|Boolean|) |CExpr|))
          (SIGNATURE |operand| (|CExpr| |CExpr| (|Integer|)))
          (SIGNATURE |numberOfOperand| ((|Integer|) |CExpr|))
          (SIGNATURE |construct| (|CExpr| |CExpr| |CExpr|)))))
        |$EmptyMode|
        (((
           (|$DomainsInScope|
            (FLUID . |true|)
            (special |$EmptyMode| |$NoValueMode|))))))
```

This compiler call expects the first argument `x` to be a `DEF` form to compile, The second argument, `m`, is the mode. The third argument, `e`, is the environment.

10.1.12 defun compOrCroak

[compOrCroak1 p336]

— defun compOrCroak —

```
(defun |compOrCroak| (x m e)
  (|compOrCroak1| x m e nil nil))
```

———-

This results in a call to the inner function with

```
(|compOrCroak1|
          (DEF (|CohenCategory|)
           ((|Category|))
           (NIL)
           (|Join|
            (|SetCategory|)
            (CATEGORY |package|
             (SIGNATURE |kind| ((|Boolean|) |CExpr|))
             (SIGNATURE |operand| (|CExpr| |CExpr| (|Integer|)))
             (SIGNATURE |numberOfOperand| ((|Integer|) |CExpr|))
             (SIGNATURE |construct| (|CExpr| |CExpr| |CExpr|)))))
          |$EmptyMode|
          ((((
              |$DomainsInScope|
              (FLUID . |true|)
              (special |$EmptyMode| |$NoValueMode|)))))
          NIL
          NIL
          |comp|)
```

The inner function augments the environment with information from the compiler stack `$compStack` and `$compErrorMessageStack`. Note that these variables are passed in the argument list so they get preserved on the call stack. The calling function gets called for every inner form so we use this implicit stacking to retain the information.

10.1.13 defun compOrCroak1

[comp p337]
[compOrCroak1,compactify p364]
[stackSemanticError p**??**]
[mkErrorExpr p**??**]
[displaySemanticErrors p**??**]
[say p**??**]
[displayComp p**??**]
[userError p**??**]
[$compStack p**??**]
[$compErrorMessageStack p**??**]
[$level p**??**]
[$s p**??**]
[$scanIfTrue p**??**]
[$exitModeStack p**??**]
[compOrCroak p335]

— **defun compOrCroak1** —

```
(defun |compOrCroak1| (x m e |$compStack| |$compErrorMessageStack|)
 (declare (special |$compStack| |$compErrorMessageStack|))
 (let (td errorMessage)
  (declare (special |$level| |$s| |$scanIfTrue| |$exitModeStack|))
  (cond
   ((setq td (catch '|compOrCroak| (|comp| x m e))) td)
   (t
     (setq |$compStack| (cons (list x m e |$exitModeStack|) |$compStack|))
     (setq |$s| (|compOrCroak1,compactify| |$compStack|))
     (setq |$level| (|#| |$s|))
     (setq errorMessage
       (if |$compErrorMessageStack|
         (car |$compErrorMessageStack|)
         '|unspecified error|))
     (cond
      (|$scanIfTrue|
        (|stackSemanticError| errorMessage (|mkErrorExpr| |$level|))
        (list '|failedCompilation| m e ))
      (t
       (|displaySemanticErrors|)
       (say "****** comp fails at level " |$level| " with expression: ******")
       (|displayComp| |$level|)
       (|userError| errorMessage)))))))
```

———-

10.1.14 defun comp

[compNoStacking p338]
[$compStack p**??**]
[$exitModeStack p**??**]

— **defun comp** —

```
(defun |comp| (x m e)
 (let (td)
 (declare (special |$compStack| |$exitModeStack|))
  (if (setq td (|compNoStacking| x m e))
    (setq |$compStack| nil)
    (push (list x m e |$exitModeStack|) |$compStack|))
  td))
```

———-

10.1.15 defun compNoStacking

`$Representation` is bound in compDefineFunctor, set by doIt. This hack says that when something is undeclared, $ is preferred to the underlying representation – RDJ 9/12/83 [comp2 p339]
[compNoStacking1 p338]
[$compStack p**??**]
[$Representation p**??**]
[$EmptyMode p**??**]

— **defun compNoStacking** —

```
(defun |compNoStacking| (x m e)
 (let (td)
 (declare (special |$compStack| |$Representation| |$EmptyMode|))
  (if (setq td (|comp2| x m e))
    (if (and (equal m |$EmptyMode|) (equal (second td) |$Representation|))
      (list (car td) '$ (third td))
      td)
   (|compNoStacking1| x m e |$compStack|))))
```

———-

10.1.16 defun compNoStacking1

[get p**??**]
[comp2 p339]
[$compStack p**??**]

— **defun compNoStacking1** —

```
(defun |compNoStacking1| (x m e |$compStack|)
 (declare (special |$compStack|))
 (let (u td)
  (if (setq u (|get| (if (eq m '$) '|Rep| m) '|value| e))
    (if (setq td (|comp2| x (car u) e))
      (list (car td) m (third td))
      nil)
    nil)))
```

———-

10.1.17 defun comp2

[comp3 p339]
[isDomainForm p??]
[isFunctor p??]
[insert p??]
[opOf p??]
[nequal p??]
[addDomain p??]
[$bootStrapMode p??]
[$packagesUsed p??]
[$lisplib p??]

— **defun comp2** —

```
(defun |comp2| (x m e)
 (let (tmp1)
  (declare (special |$bootStrapMode| |$packagesUsed| $lisplib))
   (when (setq tmp1 (|comp3| x m e))
    (destructuring-bind (y mprime e) tmp1
     (when (and $lisplib (|isDomainForm| x e) (|isFunctor| x))
       (setq |$packagesUsed| (|insert| (list (|opOf| x)) |$packagesUsed|)))
     ; isDomainForm test needed to prevent error while compiling Ring
     ; $bootStrapMode-test necessary for compiling Ring in $bootStrapMode
     (if (and (nequal m mprime)
              (or |$bootStrapMode| (|isDomainForm| mprime e)))
       (list y mprime (|addDomain| mprime e))
       (list y mprime e))))))
```

———-

10.1.18 defun comp3

```
;comp3(x,m,$e) ==
;  --returns a Triple or %else nil to signalcan't do'
;  $e:= addDomain(m,$e)
;  e:= $e --for debugging purposes
;  m is ["Mapping",:.] => compWithMappingMode(x,m,e)
;  m is ["QUOTE",a] => (x=a => [x,m,$e]; nil)
;  STRINGP m => (atom x => (m=x or m=STRINGIMAGE x => [m,m,e]; nil); nil)
;  ^x or atom x => compAtom(x,m,e)
;  op:= first x
;  getmode(op,e) is ["Mapping",:ml] and (u:= applyMapping(x,m,e,ml)) => u
;  op is ["KAPPA",sig,varlist,body] => compApply(sig,varlist,body,rest x,m,e)
;  op=":" => compColon(x,m,e)
;  op="::" => compCoerce(x,m,e)
;  not ($insideCompTypeOf=true) and stringPrefix?('"TypeOf",PNAME op) =>
```

```
;    compTypeOf(x,m,e)
;  t:= compExpression(x,m,e)
;  t is [x',m',e'] and not MEMBER(m',getDomainsInScope e') =>
;    [x',m',addDomain(m',e')]
;  t
```

[addDomain p**??**]
[compWithMappingMode p353]
[compAtom p343]
[getmode p**??**]
[applyMapping p**??**]
[compApply p**??**]
[compColon p150]
[compCoerce p148]
[stringPrefix? p**??**]
[comp3 pname (vol5)]
[compTypeOf p342]
[compExpression p347]
[comp3 member (vol5)]
[getDomainsInScope p**??**]
[$e p**??**]
[$insideCompTypeOf p**??**]

— defun comp3 —

```
(defun |comp3| (x m |$e|)
 (declare (special |$e|))
 (let (e a op ml u sig varlist tmp3 body tt xprime tmp1 mprime tmp2 eprime)
 (declare (special |$insideCompTypeOf|))
  (setq |$e| (|addDomain| m |$e|))
  (setq e |$e|)
  (cond
   ((and (pairp m) (eq (qcar m) '|Mapping|)) (|compWithMappingMode| x m e))
   ((and (pairp m) (eq (qcar m) 'quote)
         (progn
          (setq tmp1 (qcdr m))
          (and (pairp tmp1) (eq (qcdr tmp1) nil)
          (progn (setq a (qcar tmp1)) t))))
    (when (equal x a) (list x m |$e|)))
   ((stringp m)
    (when (and (atom x) (or (equal m x) (equal m (princ-to-string x))))
     (list m m e )))
   ((or (null x) (atom x)) (|compAtom| x m e))
   (t
    (setq op (car x))
    (cond
     ((and (progn
            (setq tmp1 (|getmode| op e))
```

```
        (and (pairp tmp1)
             (eq (qcar tmp1) '|Mapping|)
             (progn (setq ml (qcdr tmp1)) t)))
        (setq u (|applyMapping| x m e ml)))
   u)
((and (pairp op) (eq (qcar op) 'kappa)
      (progn
       (setq tmp1 (qcdr op))
       (and (pairp tmp1)
            (progn
             (setq sig (qcar tmp1))
              (setq tmp2 (qcdr tmp1))
              (and (pairp tmp2)
                   (progn
                    (setq varlist (qcar tmp2))
                    (setq tmp3 (qcdr tmp2))
                    (and (pairp tmp3)
                         (eq (qcdr tmp3) nil)
                         (progn
                           (setq body (qcar tmp3))
                           t))))))))
  (|compApply| sig varlist body (cdr x) m e))
((eq op '|:|)  (|compColon| x m e))
((eq op '|::|) (|compCoerce| x m e))
((and (null (eq |$insideCompTypeOf| t))
      (|stringPrefix?| "TypeOf" (pname op)))
  (|compTypeOf| x m e))
 (t
  (setq tt (|compExpression| x m e))
  (cond
   ((and (pairp tt)
         (progn
          (setq xprime (qcar tt))
          (setq tmp1 (qcdr tt))
          (and (pairp tmp1)
               (progn
                (setq mprime (qcar tmp1))
                (setq tmp2 (qcdr tmp1))
                (and (pairp tmp2)
                     (eq (qcdr tmp2) nil)
                     (progn
                       (setq eprime (qcar tmp2))
                       t)))))
          (null (|member| mprime (|getDomainsInScope| eprime))))
    (list xprime mprime (|addDomain| mprime eprime)))
   (t tt))))))))
```

10.1.19 defun compTypeOf

[eqsubstlist p??]
[get p??]
[put p??]
[comp3 p339]
[$insideCompTypeOf p??]
[$FormalMapVariableList p??]

— **defun compTypeOf** —

```
(defun |compTypeOf| (x m e)
 (let (|$insideCompTypeOf| op argl newModemap)
 (declare (special |$insideCompTypeOf| |$FormalMapVariableList|))
  (setq op (car x))
  (setq argl (cdr x))
  (setq |$insideCompTypeOf| t)
  (setq newModemap
    (eqsubstlist argl |$FormalMapVariableList| (|get| op '|modemap| e)))
  (setq e (|put| op '|modemap| newModemap e))
  (|comp3| x m e)))
```

———-

10.1.20 defun compColonInside

[addDomain p??]
[comp p337]
[coerce p??]
[stackWarning p??]
[opOf p??]
[stackSemanticError p??]
[$newCompilerUnionFlag p??]
[$EmptyMode p??]

— **defun compColonInside** —

```
(defun |compColonInside| (x m e mprime)
 (let (mpp warningMessage td tprime)
 (declare (special |$newCompilerUnionFlag| |$EmptyMode|))
    (setq e (|addDomain| mprime e))
    (when (setq td (|comp| x |$EmptyMode| e))
     (cond
      ((equal (setq mpp (second td)) mprime)
       (setq warningMessage
        (list '|:| mprime '| -- should replace by @|))))
```

```
(setq td (list (car td) mprime (third td)))
(when (setq tprime (|coerce| td m))
 (cond
  (warningMessage (|stackWarning| warningMessage))
  ((and |$newCompilerUnionFlag| (eq (|opOf| mpp) '|Union|))
   (setq tprime
    (|stackSemanticError|
     (list '|cannot pretend | x '| of mode | mpp '| to mode | mprime )
     nil)))
  (t
   (|stackWarning|
    (list '|:| mprime '| -- should replace by pretend|))))
 tprime))))
```

———-

10.1.21 defun compAtom

```
;compAtom(x,m,e) ==
;  T:= compAtomWithModemap(x,m,e,get(x,"modemap",e)) => T
;  x="nil" =>
;    T:=
;      modeIsAggregateOf('List,m,e) is [.,R]=> compList(x,['List,R],e)
;      modeIsAggregateOf('Vector,m,e) is [.,R]=> compVector(x,['Vector,R],e)
;    T => convert(T,m)
;  t:=
;    isSymbol x =>
;      compSymbol(x,m,e) or return nil
;    m = $Expression and primitiveType x => [x,m,e]
;    STRINGP x => [x,x,e]
;    [x,primitiveType x or return nil,e]
;  convert(t,m)
```

[compAtomWithModemap p**??**]
[get p**??**]
[modeIsAggregateOf p**??**]
[compList p347]
[compVector p187]
[convert p344]
[isSymbol p**??**]
[compSymbol p345]
[primitiveType p345]
[primitiveType p345]
[$Expression p**??**]

— **defun compAtom** —

```
(defun |compAtom| (x m e)
 (prog (tmp1 tmp2 r td tt)
  (declare (special |$Expression|))
   (return
   (cond
    ((setq td (|compAtomWithModemap| x m e (|get| x '|modemap| e))) td)
    ((eq x '|nil|)
     (setq td
      (cond
       ((progn
         (setq tmp1 (|modeIsAggregateOf| '|List| m e))
         (and (pairp tmp1)
              (progn
               (setq tmp2 (qcdr tmp1))
               (and (pairp tmp2)
                    (eq (qcdr tmp2) nil)
                    (progn
                      (setq r (qcar tmp2)) t)))))
         (|compList| x (list '|List| r) e))
       ((progn
         (setq tmp1 (|modeIsAggregateOf| '|Vector| m e))
         (and (pairp tmp1)
              (progn
               (setq tmp2 (qcdr tmp1))
               (and (pairp tmp2) (eq (qcdr tmp2) nil)
               (progn
                 (setq r (qcar tmp2)) t)))))
         (|compVector| x (list '|Vector| r) e))))
      (when td (|convert| td m)))
    (t
     (setq tt
      (cond
       ((|isSymbol| x) (or (|compSymbol| x m e) (return nil)))
       ((and (equal m |$Expression|) (|primitiveType| x)) (list x m e ))
       ((stringp x) (list x x e ))
       (t (list x (or (|primitiveType| x) (return nil)) e ))))
     (|convert| tt m))))))
```

———-

10.1.22 defun convert

[resolve p**??**]
[coerce p**??**]

— defun convert —

```
(defun |convert| (td m)
```

```
 (let (res)
  (when (setq res (|resolve| (second td) m))
   (|coerce| td res))))
```

———-

10.1.23 defun primitiveType

[$DoubleFloat p??]
[$NegativeInteger p??]
[$PositiveInteger p??]
[$NonNegativeInteger p??]
[$String p183]
[$EmptyMode p??]

— **defun primitiveType** —

```
(defun |primitiveType| (x)
  (declare (special |$DoubleFloat| |$NegativeInteger| |$PositiveInteger|
                    |$NonNegativeInteger| |$String| |$EmptyMode|))
  (cond
    ((null x) |$EmptyMode|)
    ((stringp x) |$String|)
    ((integerp x)
     (cond
       ((eql x 0) |$NonNegativeInteger|)
       ((> x 0) |$PositiveInteger|)
       (t |$NegativeInteger|)))
    ((floatp x) |$DoubleFloat|)
    (t nil)))
```

———-

10.1.24 defun compSymbol

[getmode p??]
[get p??]
[NRTgetLocalIndex p??]
[compSymbol member (vol5)]
[isFunction p??]
[errorRef p??]
[stackMessage p??]
[$Symbol p??]
[$Expression p??]

[$FormalMapVariableList p??]
[$compForModeIfTrue p??]
[$formalArgList p??]
[$NoValueMode p??]
[$functorLocalParameters p??]
[$Boolean p??]
[$NoValue p??]

— defun compSymbol —

```
(defun |compSymbol| (s m e)
 (let (v mprime mode)
  (declare (special |$Symbol| |$Expression| |$FormalMapVariableList|
                    |$compForModeIfTrue| |$formalArgList| |$NoValueMode|
                    |$functorLocalParameters| |$Boolean| |$NoValue|))
   (cond
     ((eq s '|$NoValue|) (list '|$NoValue| |$NoValueMode| e ))
     ((|isFluid| s)
      (setq mode (|getmode| s e))
      (when mode (list s (|getmode| s e) e)))
     ((eq s '|true|) (list '(quote t) |$Boolean| e ))
     ((eq s '|false|) (list nil |$Boolean| e ))
     ((or (equal s m) (|get| s '|isLiteral| e)) (list (list 'quote s) s e))
     ((setq v (|get| s '|value| e))
      (cond
        ((member s |$functorLocalParameters|)
         ; s will be replaced by an ELT form in beforeCompile
         (|NRTgetLocalIndex| s)
         (list s (second v) e))
        (t
         ; s has been SETQd
         (list s (second v) e))))
     ((setq mprime (|getmode| s e))
      (cond
        ((and (null (|member| s |$formalArgList|))
              (null (member s |$FormalMapVariableList|))
              (null (|isFunction| s e))
              (null (eq |$compForModeIfTrue| t)))
         (|errorRef| s)))
      (list s mprime e ))
     ((member s |$FormalMapVariableList|)
      (|stackMessage| (list '|no mode found for| s )))
     ((or (equal m |$Expression|) (equal m |$Symbol|))
      (list (list 'quote s) m e ))
     ((null (|isFunction| s e)) (|errorRef| s)))))
```

———-

10.1.25 defun compList

```
;compList(l,m is ["List",mUnder],e) ==
;  null l => [NIL,m,e]
;  Tl:= [[.,mUnder,e]:= comp(x,mUnder,e) or return "failed" for x in l]
;  Tl="failed" => nil
;  T:= [["LIST",:[T.expr for T in Tl]],["List",mUnder],e]
```

[comp p337]

— **defun compList** —

```
(defun |compList| (l m e)
 (let (tmp1 tmp2 t0 failed (mUnder (second m)))
  (if (null l)
   (list nil m e)
   (progn
    (setq t0
     (do ((t3 l (cdr t3)) (x nil))
         ((or (atom t3) failed) (unless failed (nreverse0 tmp2)))
       (setq x (car t3))
       (if (setq tmp1 (|comp| x mUnder e))
        (progn
         (setq mUnder (second tmp1))
         (setq e (third tmp1))
         (push tmp1 tmp2))
        (setq failed t))))
    (unless failed
     (cons
       (cons 'list (loop for texpr in t0 collect (car texpr)))
       (list (list '|List| mUnder) e)))))))
```

———-

10.1.26 defun compExpression

[getl p**??**]
[compForm p348]
[$insideExpressionIfTrue p**??**]

— **defun compExpression** —

```
(defun |compExpression| (x m e)
 (let (|$insideExpressionIfTrue| fn)
 (declare (special |$insideExpressionIfTrue|))
  (setq |$insideExpressionIfTrue| t)
  (if (and (atom (car x)) (setq fn (getl (car x) 'special)))
```

```
    (funcall fn x m e)
    (|compForm| x m e))))
```

———-

10.1.27 defun compForm

[compForm1 p348]
[compArgumentsAndTryAgain p352]
[stackMessageIfNone p??]

— **defun compForm** —

```
(defun |compForm| (form m e)
 (cond
  ((|compForm1| form m e))
  ((|compArgumentsAndTryAgain| form m e))
  (t (|stackMessageIfNone| (list '|cannot compile| '|%b| form '|%d| )))))
```

———-

10.1.28 defun compForm1

[length p??]
[outputComp p??]
[compOrCroak p335]
[compExpressionList p??]
[coerceable p??]
[comp p337]
[coerce p??]
[compForm2 p350]
[augModemapsFromDomain1 p??]
[getFormModemaps p??]
[nreverse0 p??]
[addDomain p??]
[compToApply p??]
[$NumberOfArgsIfInteger p??]
[$Expression p??]
[$EmptyMode p??]

— **defun compForm1** —

```
(defun |compForm1| (form m e)
 (let (|$NumberOfArgsIfInteger| op argl domain tmp1 opprime ans mmList td
```

```
      tmp2 tmp3 tmp4 tmp5 tmp6 tmp7)
 (declare (special |$NumberOfArgsIfInteger| |$Expression| |$EmptyMode|))
   (setq op (car form))
   (setq argl (cdr form))
   (setq |$NumberOfArgsIfInteger| (|#| argl))
   (cond
    ((eq op '|error|)
     (list
      (cons op
       (dolist (x argl (nreverse0 tmp4))
         (setq tmp2 (|outputComp| x e))
         (setq e (third tmp2))
         (push (car tmp2) tmp4)))
      m e))
    ((and (pairp op) (eq (qcar op) '|elt|)
          (progn
           (setq tmp3 (qcdr op))
           (and (pairp tmp3)
                (progn
                 (setq domain (qcar tmp3))
                 (setq tmp1 (qcdr tmp3))
                 (and (pairp tmp1)
                      (eq (qcdr tmp1) nil)
                      (progn
                       (setq opprime (qcar tmp1))
                       t))))))
      (cond
       ((eq domain '|Lisp|)
         (list
          (cons opprime
           (dolist (x argl (nreverse tmp7))
            (setq tmp2 (|compOrCroak| x |$EmptyMode| e))
            (setq e (third tmp2))
            (push (car tmp2) tmp7)))
          m e))
       ((and (equal domain |$Expression|) (eq opprime '|construct|))
         (|compExpressionList| argl m e))
       ((and (eq opprime 'collect) (|coerceable| domain m e))
         (when (setq td (|comp| (cons opprime argl) domain e))
          (|coerce| td m)))
       ((and (pairp domain) (eq (qcar domain) '|Mapping|)
             (setq ans
              (|compForm2| (cons opprime argl) m
               (setq e (|augModemapsFromDomain1| domain domain e))
               (dolist (x (|getFormModemaps| (cons opprime argl) e)
                                    (nreverse0 tmp6))
                (when
                 (and (pairp x)
                      (and (pairp (qcar x)) (equal (qcar (qcar x)) domain)))
                   (push x tmp6))))))
```

```
          ans)
         ((setq ans
           (|compForm2| (cons opprime argl) m
            (setq e (|addDomain| domain e))
            (dolist (x (|getFormModemaps| (cons opprime argl) e)
                     (nreverse0 tmp5))
              (when
                (and (pairp x)
                     (and (pairp (qcar x)) (equal (qcar (qcar x)) domain)))
                 (push x tmp5)))))
           ans)
         ((and (eq opprime '|construct|) (|coerceable| domain m e))
           (when (setq td (|comp| (cons opprime argl) domain e))
            (|coerce| td m)))
         (t nil)))
      (t
       (setq e (|addDomain| m e))
       (cond
        ((and (setq mmList (|getFormModemaps| form e))
              (setq td (|compForm2| form m e mmList)))
          td)
        (t
          (|compToApply| op argl m e)))))))
```

———-

10.1.29 defun compForm2

[take p??]
[length p??]
[nreverse0 p??]
[sublis p??]
[assoc p??]
[PredImplies p??]
[isSimple p??]
[compUniquely p??]
[compFormPartiallyBottomUp p??]
[compForm3 p??]
[$EmptyMode p??]
[$TriangleVariableList p??]

— defun compForm2 —

```
(defun |compForm2| (form m e modemapList)
 (let (op argl sargl aList dc cond nsig v ncond deleteList newList td tl
       partialModeList tmp1 tmp2 tmp3 tmp4 tmp5 tmp6 tmp7)
 (declare (special |$EmptyMode| |$TriangleVariableList|))
```

```
(setq op (car form))
(setq argl (cdr form))
(setq sargl (take (|#| argl) |$TriangleVariableList|))
(setq aList (mapcar #'(lambda (x y) (cons x y)) sargl argl))
(setq modemaplist (sublis aList modemapList))
; now delete any modemaps that are subsumed by something else, provided
; the conditions are right (i.e. subsumer true whenever subsumee true)
(dolist (u modemapList)
 (cond
  ((and (pairp u)
        (progn
         (setq tmp6 (qcar u))
         (and (pairp tmp6) (progn (setq dc (qcar tmp6)) t)))
        (progn
         (setq tmp7 (qcdr u))
         (and (pairp tmp7) (eq (qcdr tmp7) nil)
              (progn
               (setq tmp1 (qcar tmp7))
               (and (pairp tmp1)
                    (progn
                     (setq cond (qcar tmp1))
                     (setq tmp2 (qcdr tmp1))
                     (and (pairp tmp2) (eq (qcdr tmp2) nil)
                          (progn
                           (setq tmp3 (qcar tmp2))
                           (and (pairp tmp3) (eq (qcar tmp3) '|Subsumed|)
                                (progn
                                 (setq tmp4 (qcdr tmp3))
                                 (and (pairp tmp4)
                                      (progn
                                       (setq tmp5 (qcdr tmp4))
                                       (and (pairp tmp5)
                                            (eq (qcdr tmp5) nil)
                                            (progn
                                             (setq nsig (qcar tmp5))
                                             t)))))))))))))
        (setq v (|assoc| (cons dc nsig) modemapList))
        (pairp v)
        (progn
         (setq tmp6 (qcdr v))
         (and (pairp tmp6) (eq (qcdr tmp6) nil)
              (progn
               (setq tmp7 (qcar tmp6))
               (and (pairp tmp7)
                    (progn
                     (setq ncond (qcar tmp7))
                     t))))))
    (setq deleteList (cons u deleteList))
    (unless  (|PredImplies| ncond cond)
       (setq newList (push `(,(car u) (,cond (elt ,dc nil))) newList))))))
```

```
  (when deleteList
   (setq modemapList
    (remove-if #'(lambda (x) (member x deletelist)) modemapList)))
  ; it is important that subsumed ops (newList) be considered last
  (when newList (setq modemapList (append modemapList newList)))
  (setq tl
   (loop for x in argl
         while (and (|isSimple| x)
                    (setq td (|compUniquely| x |$EmptyMode| e)))
         collect td
         do (setq e (third td))))
  (cond
   ((some #'identity tl)
     (setq partialModeList (loop for x in tl collect (when x (second x))))
     (or (|compFormPartiallyBottomUp| form m e modemapList partialModeList)
         (|compForm3| form m e modemapList)))
   (t (|compForm3| form m e modemapList)))))
```

———-

10.1.30 defun compArgumentsAndTryAgain

[comp p337]
[compForm1 p348]
[$EmptyMode p??]

— **defun compArgumentsAndTryAgain** —

```
(defun |compArgumentsAndTryAgain| (form m e)
 (let (argl tmp1 a tmp2 tmp3 u)
 (declare (special |$EmptyMode|))
  (setq argl (cdr form))
  (cond
   ((and (pairp form) (eq (qcar form) '|elt|)
         (progn
           (setq tmp1 (qcdr form))
           (and (pairp tmp1)
                (progn
                 (setq a (qcar tmp1))
                 (setq tmp2 (qcdr tmp1))
                 (and (pairp tmp2) (eq (qcdr tmp2) nil))))))
    (when (setq tmp3 (|comp| a |$EmptyMode| e))
      (setq e (third tmp3))
      (|compForm1| form m e)))
   (t
     (setq u
      (dolist (x argl)
```

```
      (setq tmp3 (or (|comp| x |$EmptyMode| e) (return '|failed|)))
      (setq e (third tmp3))
      tmp3))
    (unless (eq u '|failed|)
      (|compForm1| form m e))))))
```

———-

10.1.31 defun compWithMappingMode

[compWithMappingMode1 p353]
[$formalArgList p??]

— **defun compWithMappingMode** —

```
(defun |compWithMappingMode| (x m oldE)
  (declare (special |$formalArgList|))
  (|compWithMappingMode1| x m oldE |$formalArgList|))
```

———-

10.1.32 defun compWithMappingMode1

```
;compWithMappingMode1(x,m is ["Mapping",m',:sl],oldE,$formalArgList) ==
;  $killOptimizeIfTrue: local:= true
;  e:= oldE
;  isFunctor x =>
;    if get(x,"modemap",$CategoryFrame) is [[[.,target,:argModeList],.],:.] and
;     (and/[extendsCategoryForm("$",s,mode) for mode in argModeList for s in sl]
;       ) and extendsCategoryForm("$",target,m') then return [x,m,e]
;  if STRINGP x then x:= INTERN x
;  ress:=nil
;  old_style:=true
;  if x is ["+->",vl,nx] then
;    old_style:=false
;    vl is [":",:.] =>
;      ress:=compLambda(x,m,oldE)
;      ress
;    vl:=
;      vl is ["Tuple",:vl1] => vl1
;      vl
;    vl:=
;      SYMBOLP(vl) => [vl]
;      LISTP(vl) and (and/[SYMBOLP(v) for v in vl]) => vl
;      stackAndThrow ["bad +-> arguments:",vl]
```

```
;    $formatArgList:=[:vl,:$formalArgList]
;    x:=nx
;  else
;    vl:=take(#sl,$FormalMapVariableList)
;  ress => ress
;  for m in sl for v in vl repeat
;    [.,.,e]:= compMakeDeclaration([":",v,m],$EmptyMode,e)
;  old_style and not null vl and not hasFormalMapVariable(x, vl) => return
;    [u,.,.] := comp([x,:vl],m',e) or return nil
;    extractCodeAndConstructTriple(u, m, oldE)
;  null vl and (t := comp([x], m', e)) => return
;    [u,.,.] := t
;    extractCodeAndConstructTriple(u, m, oldE)
;  [u,.,.]:= comp(x,m',e) or return nil
;  uu:=optimizeFunctionDef [nil,['LAMBDA,vl,u]]
;  --  At this point, we have a function that we would like to pass.
;  --  Unfortunately, it makes various free variable references outside
;  --  itself.  So we build a mini-vector that contains them all, and
;  --  pass this as the environment to our inner function.
;  $FUNNAME :local := nil
;  $FUNNAME__TAIL :local := [nil]
;  expandedFunction:=COMP_-TRAN CADR uu
;  frees:=freelist(expandedFunction,vl,nil,e)
;    where freelist(u,bound,free,e) ==
;      atom u =>
;        not IDENTP u => free
;        MEMQ(u,bound) => free
;        v:=ASSQ(u,free) =>
;          RPLACD(v,1+CDR v)
;          free
;        not getmode(u, e) => free
;        [[u,:1],:free]
;      op:=CAR u
;      MEMQ(op, '(QUOTE GO function)) => free
;      EQ(op,'LAMBDA) =>
;        bound:=UNIONQ(bound,CADR u)
;        for v in CDDR u repeat
;          free:=freelist(v,bound,free,e)
;        free
;      EQ(op,'PROG) =>
;        bound:=UNIONQ(bound,CADR u)
;        for v in CDDR u | NOT ATOM v repeat
;          free:=freelist(v,bound,free,e)
;        free
;      EQ(op,'SEQ) =>
;        for v in CDR u | NOT ATOM v repeat
;          free:=freelist(v,bound,free,e)
;        free
;      EQ(op,'COND) =>
;        for v in CDR u repeat
```

```
;          for vv in v repeat
;            free:=freelist(vv,bound,free,e)
;        free
;      if ATOM op then u:=CDR u  --Atomic functions aren't descended
;      for v in u repeat
;        free:=freelist(v,bound,free,e)
;      free
;  expandedFunction :=
;            --One free can go by itself, more than one needs a vector
;         --An A-list name . number of times used
;    #frees = 0 => ['LAMBDA,[:vl,"$$"], :CDDR expandedFunction]
;    #frees = 1 =>
;      vec:=first first frees
;      ['LAMBDA,[:vl,vec], :CDDR expandedFunction]
;    scode:=nil
;    vec:=nil
;    locals:=nil
;    i:=-1
;    for v in frees repeat
;      i:=i+1
;      vec:=[first v,:vec]
;      scode:=[['SETQ,first v,[($QuickCode => 'QREFELT;'ELT),"$$",i]],:scode]
;      locals:=[first v,:locals]
;    body:=CDDR expandedFunction
;    if locals then
;      if body is [['DECLARE,:.],:.] then
;        body:=[CAR body,['PROG,locals,:scode,['RETURN,['PROGN,:CDR body]]]]
;      else body:=[['PROG,locals,:scode,['RETURN,['PROGN,:body]]]]
;    vec:=['VECTOR,:NREVERSE vec]
;    ['LAMBDA,[:vl,"$$"],:body]
;  fname:=['CLOSEDFN,expandedFunction]
;         --Like QUOTE, but gets compiled
;  uu:=
;    frees => ['CONS,fname,vec]
;    ['LIST,fname]
;  [uu,m,oldE]
```

[isFunctor p??]
[get p??]
[qcar p??]
[qcdr p??]
[extendsCategoryForm p??]
[compLambda p168]
[stackAndThrow p??]
[take p??]
[compMakeDeclaration p362]
[hasFormalMapVariable p361]
[comp p337]
[extractCodeAndConstructTriple p360]

[optimizeFunctionDef p??]
[comp-tran p??]
[freelist p363]
[$formalArgList p??]
[$killOptimizeIfTrue p??]
[$funname p??]
[$funnameTail p??]
[$QuickCode p??]
[$EmptyMode p??]
[$FormalMapVariableList p??]
[$CategoryFrame p??]
[$formatArgList p??]

— defun compWithMappingMode1 —

```
(defun |compWithMappingMode1| (x m oldE |$formalArgList|)
 (declare (special |$formalArgList|))
  (prog (|$killOptimizeIfTrue| $funname $funnameTail mprime sl tmp1 tmp2
         tmp3 tmp4 tmp5 tmp6 target argModeList nx oldstyle ress vl1 vl e tt
             u frees i scode locals body vec expandedFunction fname uu)
  (declare (special |$killOptimizeIfTrue| $funname $funnameTail
                    |$QuickCode| |$EmptyMode| |$FormalMapVariableList|
                    |$CategoryFrame| |$formatArgList|))
   (return
    (seq
     (progn
      (setq mprime (second m))
      (setq sl (cddr m))
      (setq |$killOptimizeIfTrue| t)
      (setq e oldE)
      (cond
       ((|isFunctor| x)
        (cond
         ((and (progn
                (setq tmp1 (|get| x '|modemap| |$CategoryFrame|))
                (and (pairp tmp1)
                     (progn
                      (setq tmp2 (qcar tmp1))
                      (and (pairp tmp2)
                           (progn
                            (setq tmp3 (qcar tmp2))
                            (and (pairp tmp3)
                                 (progn
                                  (setq tmp4 (qcdr tmp3))
                                  (and (pairp tmp4)
                                       (progn
                                        (setq target (qcar tmp4))
                                        (setq argModeList (qcdr tmp4))
                                        t)))))))))
```

```
                    (progn
                     (setq tmp5 (qcdr tmp2))
                     (and (pairp tmp5) (eq (qcdr tmp5) nil)))))))
          (prog (t1)
           (setq t1 t)
           (return
            (do ((t2 nil (null t1))
                 (t3 argModeList (cdr t3))
                 (mode nil)
                 (t4 sl (cdr t4))
                 (s nil))
                ((or t2 (atom t3)
                     (progn (setq mode (car t3)) nil)
                     (atom t4)
                     (progn (setq s (car t4)) nil))
                 t1)
              (seq (exit
                (setq t1
                 (and t1 (|extendsCategoryForm| '$ s mode))))))))
                   (|extendsCategoryForm| '$ target mprime))
          (return (list x m e )))
    (t nil)))
  (t
   (when (stringp x) (setq x (intern x)))
   (setq ress nil)
   (setq oldstyle t)
   (cond
    ((and (pairp x)
          (eq (qcar x) '+->)
          (progn
           (setq tmp1 (qcdr x))
           (and (pairp tmp1)
                (progn
                 (setq vl (qcar tmp1))
                 (setq tmp2 (qcdr tmp1))
                 (and (pairp tmp2)
                      (eq (qcdr tmp2) nil)
                      (progn (setq nx (qcar tmp2)) t))))))
      (setq oldstyle nil)
      (cond
       ((and (pairp vl) (eq (qcar vl) '|:|))
         (setq ress (|compLambda| x m oldE))
         ress)
       (t
        (setq vl
         (cond
          ((and (pairp vl)
                (eq (qcar vl) '|@Tuple|)
                (progn (setq vl1 (qcdr vl)) t))
            vl1)
```

```
        (t vl)))
       (setq vl
        (cond
         ((symbolp vl) (cons vl nil))
         ((and
           (listp vl)
           (prog (t5)
            (setq t5 t)
            (return
             (do ((t7 nil (null t5))
                  (t6 vl (cdr t6))
                  (v nil))
                 ((or t7 (atom t6) (progn (setq v (car t6)) nil)) t5)
                (seq
                 (exit
                  (setq t5 (and t5 (symbolp v))))))))
           vl)
         (t
          (|stackAndThrow| (cons '|bad +-> arguments:| (list vl ))))))
       (setq |$formatArgList| (append vl |$formalArgList|))
       (setq x nx))))
 (t
  (setq vl (take (|#| sl) |$FormalMapVariableList|))))
(cond
 (ress ress)
 (t
  (do ((t8 sl (cdr t8)) (m nil) (t9 vl (cdr t9)) (v nil))
      ((or (atom t8)
           (progn (setq m (car t8)) nil)
           (atom t9)
           (progn (setq v (car t9)) nil))
       nil)
      (seq (exit (progn
       (setq tmp6
        (|compMakeDeclaration| (list '|:| v m ) |$EmptyMode| e))
       (setq e (third tmp6))
       tmp6))))
  (cond
   ((and oldstyle
         (null (null vl))
         (null (|hasFormalMapVariable| x vl)))
    (return
     (progn
      (setq tmp6 (or (|comp| (cons x vl) mprime e) (return nil)))
      (setq u (car tmp6))
      (|extractCodeAndConstructTriple| u m oldE))))
   ((and (null vl) (setq tt (|comp| (cons x nil) mprime e)))
    (return
     (progn
      (setq u (car tt))
```

```
              (|extractCodeAndConstructTriple| u m oldE))))
           (t
            (setq tmp6 (or (|comp| x mprime e) (return nil)))
            (setq u (car tmp6))
            (setq uu (|optimizeFunctionDef| `(nil (lambda ,vl ,u))))
;  --  At this point, we have a function that we would like to pass.
;  --  Unfortunately, it makes various free variable references outside
;  --  itself.  So we build a mini-vector that contains them all, and
;  --  pass this as the environment to our inner function.
            (setq $funname nil)
            (setq $funnameTail (list nil))
            (setq expandedFunction (comp-tran (second uu)))
            (setq frees (freelist expandedFunction vl nil e))
            (setq expandedFunction
             (cond
              ((eql (|#| frees) 0)
               (cons 'lambda (cons (append vl (list '$$))
                                   (cddr expandedFunction))))
              ((eql (|#| frees) 1)
               (setq vec (caar frees))
                (cons 'lambda (cons (append vl (list vec))
                                   (cddr expandedFunction))))
              (t
               (setq scode nil)
               (setq vec nil)
               (setq locals nil)
               (setq i -1)
               (do ((t0 frees (cdr t0)) (v nil))
                   ((or (atom t0) (progn (setq v (car t0)) nil)) nil)
                 (seq
                  (exit
                   (progn
                    (setq i (plus i 1))
                    (setq vec (cons (car v) vec))
                    (setq scode
                     (cons
                      (cons 'setq
                       (cons (car v)
                        (cons
                         (cons
                          (cond
                           (|$QuickCode| 'qrefelt)
                           (t 'elt))
                          (cons '$$ (cons i nil)))
                         nil)))
                       scode))
                    (setq locals (cons (car v) locals))))))
               (setq body (cddr expandedFunction))
               (cond
                (locals
```

```
      (cond
       ((and (pairp body)
             (progn
              (setq tmp1 (qcar body))
              (and (pairp tmp1)
                   (eq (qcar tmp1) 'declare))))
        (setq body
         (cons (car body)
          (cons
           (cons 'prog
            (cons locals
             (append scode
              (cons
               (cons 'return
                (cons
                 (cons 'progn
                  (cdr body))
                 nil))
               nil))))
            nil))))
       (t
        (setq body
         (cons
          (cons 'prog
           (cons locals
            (append scode
             (cons
              (cons 'return
               (cons
                (cons 'progn body)
                nil))
              nil))))
           nil))))))
    (setq vec (cons 'vector (nreverse vec)))
    (cons 'lambda (cons (append vl (list '$$)) body)))))
(setq fname (list 'closedfn expandedFunction))
(setq uu
 (cond
  (frees (list 'cons fname vec))
  (t (list 'list fname))))
(list uu m oldE)))))))))))
```

———-

10.1.33 defun extractCodeAndConstructTriple

— defun extractCodeAndConstructTriple —

```
(defun |extractCodeAndConstructTriple| (u m oldE)
 (let (tmp1 a fn op env)
  (cond
   ((and (pairp u) (eq (qcar u) '|call|)
         (progn
          (setq tmp1 (qcdr u))
          (and (pairp tmp1)
               (progn (setq fn (qcar tmp1)) t))))
    (cond
     ((and (pairp fn) (eq (qcar fn) '|applyFun|)
           (progn
            (setq tmp1 (qcdr fn))
            (and (pairp tmp1) (eq (qcdr tmp1) nil)
                 (progn (setq a (qcar tmp1)) t))))
            (setq fn a)))
    (list fn m oldE))
   (t
    (setq op (car u))
    (setq env (car (reverse (cdr u))))
    (list (list 'cons  (list '|function| op) env) m oldE)))))
```

———-

10.1.34 defun hasFormalMapVariable

[hasFormalMapVariable ScanOrPairVec (vol5)]
[$formalMapVariables p**??**]

— **defun hasFormalMapVariable** —

```
(defun |hasFormalMapVariable| (x vl)
 (let (|$formalMapVariables|)
 (declare (special |$formalMapVariables|))
 (when (setq |$formalMapVariables| vl)
  (|ScanOrPairVec| #'(lambda (y) (member y |$formalMapVariables|)) x))))
```

———-

10.1.35 defun argsToSig

— **defun argsToSig** —

```
(defun |argsToSig| (args)
 (let (tmp1 v tmp2 tt sig1 arg1 bad)
```

```
  (cond
   ((and (pairp args) (eq (qcar args) '|:|)
         (progn
          (setq tmp1 (qcdr args))
          (and (pairp tmp1)
               (progn
                (setq v (qcar tmp1))
                (setq tmp2 (qcdr tmp1))
                (and (pairp tmp2)
                     (eq (qcdr tmp2) nil)
                     (progn
                      (setq tt (qcar tmp2))
                      t))))))
    (list (list v) (list tt)))
   (t
    (setq sig1 nil)
    (setq arg1 nil)
    (setq bad nil)
    (dolist (arg args)
      (cond
       ((and (pairp arg) (eq (qcar arg) '|:|)
             (progn
              (setq tmp1 (qcdr arg))
              (and (pairp tmp1)
                   (progn
                    (setq v (qcar tmp1))
                    (setq tmp2 (qcdr tmp1))
                    (and (pairp tmp2) (eq (qcdr tmp2) nil)
                         (progn
                          (setq tt (qcar tmp2))
                          t))))))
        (setq sig1 (cons tt sig1))
        (setq arg1 (cons v arg1)))
       (t (setq bad t))))
    (cond
     (bad (list nil nil ))
     (t (list (reverse arg1) (reverse sig1))))))))
```

———-

10.1.36 defun compMakeDeclaration

[compColon p150]
[$insideExpressionIfTrue p**??**]

— defun compMakeDeclaration —

```
(defun |compMakeDeclaration| (x m e)
```

```
(let (|$insideExpressionIfTrue|)
(declare (special |$insideExpressionIfTrue|))
 (setq |$insideExpressionIfTrue| nil)
 (|compColon| x m e)))
```

———-

10.1.37 defun modifyModeStack

[say p??]
[copy p??]
[setelt p??]
[resolve p??]
[$reportExitModeStack p??]
[$exitModeStack p??]

— **defun modifyModeStack** —

```
(defun |modifyModeStack| (|m| |index|)
 (declare (special |$exitModeStack| |$reportExitModeStack|))
 (if |$reportExitModeStack|
   (say "exitModeStack: " (copy |$exitModeStack|)
    " ====> "
    (progn
     (setelt |$exitModeStack| |index|
      (|resolve| |m| (elt |$exitModeStack| |index|)))
     |$exitModeStack|))
   (setelt |$exitModeStack| |index|
      (|resolve| |m| (elt |$exitModeStack| |index|)))))
```

———-

10.1.38 defun Create a list of unbound symbols

We walk argument u looking for symbols that are unbound. If we find a symbol we add it to the free list. If it occurs in a prog then it is bound and we remove it from the free list. Multiple instances of a single symbol in the free list are represented by the alist (symbol . count) [freelist p363]
[freelist assq (vol5)]
[freelist identp (vol5)]
[getmode p??]
[unionq p??]

— **defun freelist** —

```
(defun freelist (u bound free e)
 (let (v op)
  (if (atom u)
   (cond
    ((null (identp u)) free)
    ((member u bound) free)
    ; more than 1 free becomes alist (name . number)
    ((setq v (assq u free)) (rplacd v (+ 1 (cdr v))) free)
    ((null (|getmode| u e)) free)
    (t (cons (cons u 1) free)))
   (progn
    (setq op (car u))
    (cond
     ((member op '(quote go |function|)) free)
     ((eq op 'lambda) ; lambdas bind symbols
      (setq bound (unionq bound (second u)))
      (dolist (v (cddr u))
       (setq free (freelist v bound free e))))
     ((eq op 'prog) ; progs bind symbols
      (setq bound (unionq bound (second u)))
      (dolist (v (cddr u))
       (unless (atom v)
        (setq free (freelist v bound free e)))))
     ((eq op 'seq)
      (dolist (v (cdr u))
       (unless (atom v)
        (setq free (freelist v bound free e)))))
     ((eq op 'cond)
      (dolist (v (cdr u))
       (dolist (vv v)
        (setq free (freelist vv bound free e)))))
     (t
      (when (atom op) (setq u (cdr u))) ; atomic functions aren't descended
      (dolist (v u)
       (setq free (freelist v bound free e)))))
    free))))
```

———-

10.1.39 defun compOrCroak1,compactify

[compOrCroak1,compactify p364]
[lassoc p??]

— defun compOrCroak1,compactify —

```
(defun |compOrCroak1,compactify| (al)
```

```
(cond
 ((null al) nil)
 ((lassoc (caar al) (cdr al)) (|compOrCroak1,compactify| (cdr al)))
 (t (cons (car al) (|compOrCroak1,compactify| (cdr al))))))
```

———-

10.1.40 defun Compiler/Interpreter interface

[ncINTERPFILE SpadInterpretStream (vol5)]
[$EchoLines p??]
[$ReadingFile p??]

— **defun ncINTERPFILE** —

```
(defun |ncINTERPFILE| (file echo)
 (let ((|$EchoLines| echo) (|$ReadingFile| t))
 (declare (special |$EchoLines| |$ReadingFile|))
  (|SpadInterpretStream| 1 file nil)))
```

———-

10.1.41 defun compileSpadLispCmd

[compileSpadLispCmd pathname (vol5)]
[compileSpadLispCmd pathnameType (vol5)]
[compileSpadLispCmd selectOptionLC (vol5)]
[compileSpadLispCmd namestring (vol5)]
[compileSpadLispCmd terminateSystemCommand (vol5)]
[compileSpadLispCmd fnameMake (vol5)]
[compileSpadLispCmd pathnameDirectory (vol5)]
[compileSpadLispCmd pathnameName (vol5)]
[compileSpadLispCmd fnameReadable? (vol5)]
[compileSpadLispCmd localdatabase (vol5)]
[throwKeyedMsg p??]
[object2String p??]
[compileSpadLispCmd sayKeyedMsg (vol5)]
[recompile-lib-file-if-necessary p366]
[spadPrompt p??]
[$options p??]

— **defun compileSpadLispCmd** —

```
(defun |compileSpadLispCmd| (args)
```

```
(let (path optlist optname optargs beQuiet dolibrary lsp)
 (declare (special |$options|))
 (setq path (|pathname| (|fnameMake| (car args) "code" "lsp")))
 (cond
  ((null (probe-file path))
    (|throwKeyedMsg| 's2il0003 (cons (|namestring| args) nil)))
  (t
   (setq optlist '(|quiet| |noquiet| |library| |nolibrary|))
   (setq beQuiet nil)
   (setq dolibrary t)
   (dolist (opt |$options|)
     (setq optname (car opt))
     (setq optargs (cdr opt))
     (case (|selectOptionLC| optname optlist nil)
        (|quiet|     (setq beQuiet t))
        (|noquiet|   (setq beQuiet nil))
        (|library|   (setq dolibrary t))
        (|nolibrary| (setq dolibrary nil))
        (t
         (|throwKeyedMsg| 's2iz0036
          (list (strconc ")" (|object2String| optname)))))))
   (setq lsp
    (|fnameMake|
     (|pathnameDirectory| path)
     (|pathnameName| path)
     (|pathnameType| path)))
   (cond
    ((|fnameReadable?| lsp)
     (unless beQuiet (|sayKeyedMsg| 's2iz0089 (list (|namestring| lsp))))
      (recompile-lib-file-if-necessary lsp))
    (t
     (|sayKeyedMsg| 's2il0003 (list (|namestring| lsp)))))
   (cond
    (dolibrary
     (unless beQuiet (|sayKeyedMsg| 's2iz0090 (list (|pathnameName| path))))
     (localdatabase (list (|pathnameName| (car args))) nil))
    ((null beQuiet) (|sayKeyedMsg| 's2iz0084 nil))
    (t nil))
   (|terminateSystemCommand|)
   (|spadPrompt|)))))
```

———-

10.1.42 defun recompile-lib-file-if-necessary

[compile-lib-file p367]
[*lisp-bin-filetype* p??]

— defun recompile-lib-file-if-necessary —

```
(defun recompile-lib-file-if-necessary (lfile)
 (let* ((bfile (make-pathname :type *lisp-bin-filetype* :defaults lfile))
        (bdate (and (probe-file bfile) (file-write-date bfile)))
        (ldate (and (probe-file lfile) (file-write-date lfile))))
  (unless (and ldate bdate (> bdate ldate))
   (compile-lib-file lfile)
   (list bfile))))
```

———-

10.1.43 defun spad-fixed-arg

— defun spad-fixed-arg —

```
(defun spad-fixed-arg (fname )
 (and (equal (symbol-package fname) (find-package "BOOT"))
      (not (get fname 'compiler::spad-var-arg))
      (search ";" (symbol-name fname))
      (or (get fname 'compiler::fixed-args)
          (setf (get fname 'compiler::fixed-args) t)))
   nil)
```

———-

10.1.44 defun compile-lib-file

— defun compile-lib-file —

```
(defun compile-lib-file (fn &rest opts)
 (unwind-protect
  (progn
   (trace (compiler::fast-link-proclaimed-type-p
           :exitcond nil
           :entrycond (spad-fixed-arg (car system::arglist))))
   (trace (compiler::t1defun
           :exitcond nil
           :entrycond (spad-fixed-arg (caar system::arglist))))
   (apply #'compile-file fn opts))
  (untrace compiler::fast-link-proclaimed-type-p compiler::t1defun)))
```

———-

10.1.45 defun compileFileQuietly

if `$InteractiveMode` then use a null outputstream [$InteractiveMode p**??**] [*standard-output* p**??**]

— **defun compileFileQuietly** —

```
(defun |compileFileQuietly| (fn)
  (let (
        (*standard-output*
        (if |$InteractiveMode| (make-broadcast-stream)
          *standard-output*)))
  (declare (special *standard-output* |$InteractiveMode|))
  (compile-file fn)))
```

————-

10.1.46 defvar $byConstructors

— **initvars** —

```
(defvar |$byConstructors| () "list of constructors to be compiled")
```

————-

10.1.47 defvar $constructorsSeen

— **initvars** —

```
(defvar |$constructorsSeen| () "list of constructors found")
```

————-

— **Compiler** —

```
(in-package "BOOT")

\getchunk{initvars}

\getchunk{LEDNUDTables}
\getchunk{GLIPHTable}
```

```
\getchunk{RENAMETOKTable}
\getchunk{GENERICTable}

\getchunk{defmacro bang}
\getchunk{defmacro line-clear}
\getchunk{defmacro must}
\getchunk{defmacro nth-stack}
\getchunk{defmacro pop-stack-1}
\getchunk{defmacro pop-stack-2}
\getchunk{defmacro pop-stack-3}
\getchunk{defmacro pop-stack-4}
\getchunk{defmacro reduce-stack-clear}
\getchunk{defmacro stack-/-empty}
\getchunk{defmacro star}

\getchunk{defun action}
\getchunk{defun addclose}
\getchunk{defun add-parens-and-semis-to-line}
\getchunk{defun Advance-Char}
\getchunk{defun advance-token}
\getchunk{defun aplTran}
\getchunk{defun aplTran1}
\getchunk{defun aplTranList}
\getchunk{defun argsToSig}

\getchunk{defun blankp}
\getchunk{defun bumperrorcount}

\getchunk{defun char-eq}
\getchunk{defun char-ne}
\getchunk{defun checkWarning}
\getchunk{defun comma2Tuple}
\getchunk{defun comp}
\getchunk{defun comp2}
\getchunk{defun comp3}
\getchunk{defun compAdd}
\getchunk{defun compArgumentsAndTryAgain}
\getchunk{defun compAtom}
\getchunk{defun compAtSign}
\getchunk{defun compCapsule}
\getchunk{defun compCapsuleInner}
\getchunk{defun compCase}
\getchunk{defun compCase1}
\getchunk{defun compCat}
\getchunk{defun compCategory}
\getchunk{defun compCoerce}
\getchunk{defun compCoerce1}
\getchunk{defun compColon}
\getchunk{defun compColonInside}
\getchunk{defun compCons}
```

```
\getchunk{defun compCons1}
\getchunk{defun compConstruct}
\getchunk{defun compConstructorCategory}
\getchunk{defun compDefine}
\getchunk{defun compDefine1}
\getchunk{defun compElt}
\getchunk{defun compExit}
\getchunk{defun compExpression}
\getchunk{defun compForm}
\getchunk{defun compForm1}
\getchunk{defun compForm2}
\getchunk{defun compHas}
\getchunk{defun compIf}
\getchunk{defun compileFileQuietly}
\getchunk{defun compile-lib-file}
\getchunk{defun compiler}
\getchunk{defun compilerDoit}
\getchunk{defun compileSpad2Cmd}
\getchunk{defun compileSpadLispCmd}
\getchunk{defun compImport}
\getchunk{defun compIs}
\getchunk{defun compJoin}
\getchunk{defun compLambda}
\getchunk{defun compLeave}
\getchunk{defun compList}
\getchunk{defun compMacro}
\getchunk{defun compMakeDeclaration}
\getchunk{defun compNoStacking}
\getchunk{defun compNoStacking1}
\getchunk{defun compOrCroak}
\getchunk{defun compOrCroak1}
\getchunk{defun compOrCroak1,compactify}
\getchunk{defun compPretend}
\getchunk{defun compQuote}
\getchunk{defun compRepeatOrCollect}
\getchunk{defun compReduce}
\getchunk{defun compReduce1}
\getchunk{defun compReturn}
\getchunk{defun compSeq}
\getchunk{defun compSeqItem}
\getchunk{defun compSeq1}
\getchunk{defun setqSetelt}
\getchunk{defun setqSingle}
\getchunk{defun compSetq}
\getchunk{defun compSetq1}
\getchunk{defun compString}
\getchunk{defun compSubDomain}
\getchunk{defun compSubDomain1}
\getchunk{defun compSymbol}
\getchunk{defun compSubsetCategory}
```

```
\getchunk{defun compSuchthat}
\getchunk{defun compTopLevel}
\getchunk{defun compTypeOf}
\getchunk{defun compVector}
\getchunk{defun compWhere}
\getchunk{defun compWithMappingMode}
\getchunk{defun compWithMappingMode1}
\getchunk{defun containsBang}
\getchunk{defun convert}
\getchunk{defun current-char}
\getchunk{defun current-symbol}
\getchunk{defun current-token}

\getchunk{defun decodeScripts}
\getchunk{defun deepestExpression}
\getchunk{defun def-rename}
\getchunk{defun def-rename1}
\getchunk{defun displayPreCompilationErrors}
\getchunk{defun dollarTran}
\getchunk{defun drop}

\getchunk{defun errhuh}
\getchunk{defun escape-keywords}
\getchunk{defun escaped}
\getchunk{defun extractCodeAndConstructTriple}

\getchunk{defun fincomblock}
\getchunk{defun floatexpid}
\getchunk{defun freelist}

\getchunk{defun get-a-line}
\getchunk{defun getScriptName}
\getchunk{defun get-token}
\getchunk{defun getToken}

\getchunk{defun hackforis}
\getchunk{defun hackforis1}
\getchunk{defun hasAplExtension}
\getchunk{defun hasFormalMapVariable}

\getchunk{defun indent-pos}
\getchunk{defun infixtok}
\getchunk{defun initialize-preparse}
\getchunk{defun initial-substring}
\getchunk{defun initial-substring-p}
\getchunk{defun is-console}
\getchunk{defun isListConstructor}
\getchunk{defun isTokenDelimiter}

\getchunk{defun killColons}
```

```
\getchunk{defun line-advance-char}
\getchunk{defun line-at-end-p}
\getchunk{defun line-current-segment}
\getchunk{defun line-next-char}
\getchunk{defun line-past-end-p}
\getchunk{defun line-print}
\getchunk{defun line-new-line}

\getchunk{defun postMakeCons}
\getchunk{defun makeSimplePredicateOrNil}
\getchunk{defun make-string-adjustable}
\getchunk{defun make-symbol-of}
\getchunk{defun match-advance-string}
\getchunk{defun match-current-token}
\getchunk{defun match-next-token}
\getchunk{defun match-string}
\getchunk{defun match-token}
\getchunk{defun meta-syntax-error}
\getchunk{defun modifyModeStack}

\getchunk{defun ncINTERPFILE}
\getchunk{defun next-char}
\getchunk{defun next-line}
\getchunk{defun next-tab-loc}
\getchunk{defun next-token}
\getchunk{defun new2OldLisp}
\getchunk{defun nonblankloc}

\getchunk{defun optional}

\getchunk{defun PARSE-AnyId}
\getchunk{defun PARSE-Application}
\getchunk{defun parse-argument-designator}
\getchunk{defun parse-identifier}
\getchunk{defun parse-keyword}
\getchunk{defun parse-number}
\getchunk{defun parse-spadstring}
\getchunk{defun parse-string}
\getchunk{defun PARSE-Category}
\getchunk{defun PARSE-Command}
\getchunk{defun PARSE-CommandTail}
\getchunk{defun PARSE-Conditional}
\getchunk{defun PARSE-Data}
\getchunk{defun PARSE-ElseClause}
\getchunk{defun PARSE-Enclosure}
\getchunk{defun PARSE-Exit}
\getchunk{defun PARSE-Expr}
\getchunk{defun PARSE-Expression}
\getchunk{defun PARSE-Float}
```

```
\getchunk{defun PARSE-FloatBase}
\getchunk{defun PARSE-FloatBasePart}
\getchunk{defun PARSE-FloatExponent}
\getchunk{defun PARSE-FloatTok}
\getchunk{defun PARSE-Form}
\getchunk{defun PARSE-FormalParameter}
\getchunk{defun PARSE-FormalParameterTok}
\getchunk{defun PARSE-getSemanticForm}
\getchunk{defun PARSE-GliphTok}
\getchunk{defun PARSE-Import}
\getchunk{defun PARSE-Infix}
\getchunk{defun PARSE-InfixWith}
\getchunk{defun PARSE-IntegerTok}
\getchunk{defun PARSE-Iterator}
\getchunk{defun PARSE-IteratorTail}
\getchunk{defun PARSE-Label}
\getchunk{defun PARSE-LabelExpr}
\getchunk{defun PARSE-Leave}
\getchunk{defun PARSE-LedPart}
\getchunk{defun PARSE-leftBindingPowerOf}
\getchunk{defun PARSE-Loop}
\getchunk{defun PARSE-Name}
\getchunk{defun PARSE-NBGliphTok}
\getchunk{defun PARSE-NewExpr}
\getchunk{defun PARSE-NudPart}
\getchunk{defun PARSE-OpenBrace}
\getchunk{defun PARSE-OpenBracket}
\getchunk{defun PARSE-Operation}
\getchunk{defun PARSE-Option}
\getchunk{defun PARSE-Prefix}
\getchunk{defun PARSE-Primary}
\getchunk{defun PARSE-Primary1}
\getchunk{defun PARSE-PrimaryNoFloat}
\getchunk{defun PARSE-PrimaryOrQM}
\getchunk{defun PARSE-Qualification}
\getchunk{defun PARSE-Quad}
\getchunk{defun PARSE-Reduction}
\getchunk{defun PARSE-ReductionOp}
\getchunk{defun PARSE-Return}
\getchunk{defun PARSE-rightBindingPowerOf}
\getchunk{defun PARSE-ScriptItem}
\getchunk{defun PARSE-Scripts}
\getchunk{defun PARSE-Seg}
\getchunk{defun PARSE-Selector}
\getchunk{defun PARSE-SemiColon}
\getchunk{defun PARSE-Sequence}
\getchunk{defun PARSE-Sequence1}
\getchunk{defun PARSE-Sexpr}
\getchunk{defun PARSE-Sexpr1}
\getchunk{defun PARSE-SpecialCommand}
```

```
\getchunk{defun PARSE-SpecialKeyWord}
\getchunk{defun PARSE-Statement}
\getchunk{defun PARSE-String}
\getchunk{defun PARSE-Suffix}
\getchunk{defun PARSE-TokenCommandTail}
\getchunk{defun PARSE-TokenList}
\getchunk{defun PARSE-TokenOption}
\getchunk{defun PARSE-TokTail}
\getchunk{defun PARSE-VarForm}
\getchunk{defun PARSE-With}
\getchunk{defun parsepiles}
\getchunk{defun parseAnd}
\getchunk{defun parseAtom}
\getchunk{defun parseAtSign}
\getchunk{defun parseCategory}
\getchunk{defun parseCoerce}
\getchunk{defun parseColon}
\getchunk{defun parseConstruct}
\getchunk{defun parseDEF}
\getchunk{defun parseDollarGreaterEqual}
\getchunk{defun parseDollarGreaterThan}
\getchunk{defun parseDollarLessEqual}
\getchunk{defun parseDollarNotEqual}
\getchunk{defun parseDropAssertions}
\getchunk{defun parseEquivalence}
\getchunk{defun parseExit}
\getchunk{defun postFlatten}
\getchunk{defun postFlattenLeft}
\getchunk{defun postForm}
\getchunk{defun parseGreaterEqual}
\getchunk{defun parseGreaterThan}
\getchunk{defun parseHas}
\getchunk{defun parseHasRhs}
\getchunk{defun parseIf}
\getchunk{defun parseIf,ifTran}
\getchunk{defun parseImplies}
\getchunk{defun parseIn}
\getchunk{defun parseInBy}
\getchunk{defun parseIs}
\getchunk{defun parseIsnt}
\getchunk{defun parseJoin}
\getchunk{defun parseLeave}
\getchunk{defun parseLessEqual}
\getchunk{defun parseLET}
\getchunk{defun parseLETD}
\getchunk{defun parseLhs}
\getchunk{defun parseMDEF}
\getchunk{defun parseNot}
\getchunk{defun parseNotEqual}
\getchunk{defun parseOr}
```

```
\getchunk{defun parsePretend}
\getchunk{defun parseprint}
\getchunk{defun parseReturn}
\getchunk{defun parseSegment}
\getchunk{defun parseSeq}
\getchunk{defun parseTran}
\getchunk{defun parseTranCheckForRecord}
\getchunk{defun parseTranList}
\getchunk{defun parseTransform}
\getchunk{defun parseType}
\getchunk{defun parseVCONS}
\getchunk{defun parseWhere}
\getchunk{defun Pop-Reduction}
\getchunk{defun postAdd}
\getchunk{defun postAtom}
\getchunk{defun postAtSign}
\getchunk{defun postBigFloat}
\getchunk{defun postBlock}
\getchunk{defun postBlockItemList}
\getchunk{defun postCapsule}
\getchunk{defun postCategory}
\getchunk{defun postcheck}
\getchunk{defun postCollect}
\getchunk{defun postCollect,finish}
\getchunk{defun postColon}
\getchunk{defun postColonColon}
\getchunk{defun postComma}
\getchunk{defun postConstruct}
\getchunk{defun postDef}
\getchunk{defun postDefArgs}
\getchunk{defun postError}
\getchunk{defun postExit}
\getchunk{defun postIf}
\getchunk{defun postin}
\getchunk{defun postIn}
\getchunk{defun postInSeq}
\getchunk{defun postIteratorList}
\getchunk{defun postJoin}
\getchunk{defun postMapping}
\getchunk{defun postMDef}
\getchunk{defun postOp}
\getchunk{defun postPretend}
\getchunk{defun postQUOTE}
\getchunk{defun postReduce}
\getchunk{defun postRepeat}
\getchunk{defun postScripts}
\getchunk{defun postScriptsForm}
\getchunk{defun postSemiColon}
\getchunk{defun postSignature}
\getchunk{defun postSlash}
```

```
\getchunk{defun postTran}
\getchunk{defun postTranList}
\getchunk{defun postTranScripts}
\getchunk{defun postTranSegment}
\getchunk{defun postTransform}
\getchunk{defun postTransformCheck}
\getchunk{defun postTuple}
\getchunk{defun postTupleCollect}
\getchunk{defun postType}
\getchunk{defun postWhere}
\getchunk{defun postWith}
\getchunk{defun print-package}
\getchunk{defun preparse}
\getchunk{defun preparse1}
\getchunk{defun preparse-echo}
\getchunk{defun preparseReadLine}
\getchunk{defun preparseReadLine1}
\getchunk{defun primitiveType}
\getchunk{defun print-defun}
\getchunk{defun push-reduction}

\getchunk{defun quote-if-string}

\getchunk{defun read-a-line}
\getchunk{defun recompile-lib-file-if-necessary}
\getchunk{defun /rf-1}
\getchunk{defun /RQ,LIB}

\getchunk{defun setDefOp}
\getchunk{defun skip-blanks}
\getchunk{defun skip-ifblock}
\getchunk{defun skip-to-endif}
\getchunk{defun spad}
\getchunk{defun spad-fixed-arg}
\getchunk{defun stack-clear}
\getchunk{defun stack-load}
\getchunk{defun stack-pop}
\getchunk{defun stack-push}
\getchunk{defun storeblanks}
\getchunk{defun s-process}

\getchunk{defun token-install}
\getchunk{defun token-lookahead-type}
\getchunk{defun token-print}
\getchunk{defun transIs}
\getchunk{defun transIs1}
\getchunk{defun translabel}
\getchunk{defun translabel1}
\getchunk{defun try-get-token}
\getchunk{defun tuple2List}
```

```
\getchunk{defun underscore}
\getchunk{defun unget-tokens}
\getchunk{defun unTuple}

\getchunk{postvars}
```

———-

Bibliography

[1] Jenks, R.J. and Sutor, R.S. "Axiom – The Scientific Computation System" Springer-Verlag New York (1992) ISBN 0-387-97855-0

[2] Knuth, Donald E., "Literate Programming" Center for the Study of Language and Information ISBN 0-937073-81-4 Stanford CA (1992)

[3] Daly, Timothy, "The Axiom Wiki Website"
http://axiom.axiom-developer.org

[4] Watt, Stephen, "Aldor",
http://www.aldor.org

[5] Lamport, Leslie, "Latex – A Document Preparation System", Addison-Wesley, New York ISBN 0-201-52983-1

[6] Ramsey, Norman "Noweb – A Simple, Extensible Tool for Literate Programming"
http://www.eecs.harvard.edu/ ~nr/noweb

[7] Daly, Timothy, "The Axiom Literate Documentation"
http://axiom.axiom-developer.org/axiom-website/documentation.html

[8] Pratt, Vaughn "Top down operator precedence" POPL '73 Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages
`hall.org.ua/halls/wizzard/pdf/Vaughan.Pratt.TDOP.pdf`

[9] Floyd, R. W. "Semantic Analysis and Operator Precedence" JACM 10, 3, 316-333 (1963)

Chapter 11

Index

Index

+− >, 168
  defplist, 168
− >, 215
  defplist, 215
<=, 128
  defplist, 128
==>, 216
  defplist, 216
=>, 212
  defplist, 212
>, 118
  defplist, 118
>=, 116, 117
  defplist, 116, 117
comp370-apply
  usedby spad, 329
eof
  usedby read-a-line, 91
  usedby spad, 329
lisp-bin-filetype
  usedby recompile-lib-file-if-necessary, 367
standard-output
  usedby compileFileQuietly, 368
terminal-io
  usedby is-console, 307
,, 207
  defplist, 207
/, 222
  defplist, 222
/RQ,LIB, 319
  calledby compilerDoit, 318
  calls /rf-1, 319
  calls echo-meta[5], 319
  uses $lisplib, 319
  defun, 319
/editfile
  usedby /rf-1, 320
  usedby compAdd, 141
  usedby compileSpad2Cmd, 316
  usedby compiler, 313
  usedby spad, 329
/rf-1, 320
  calledby /RQ,LIB, 319
  calls makeInputFilename[5], 320
  calls ncINTERPFILE, 320
  calls spad[5], 320
  uses /editfile, 320
  uses echo-meta, 320
  defun, 320
/rf[5]
  called by compilerDoit, 318
/rq[5]
  called by compilerDoit, 318
:, 110, 150, 206
  defplist, 110, 150, 206
::, 109, 148, 206
  defplist, 109, 148, 206
:BF:, 201
  defplist, 201
;, 220
  defplist, 220
==, 210
  defplist, 210
$Boolean
  usedby compCase1, 145
  usedby compIf, 164
  usedby compIs, 165
  usedby compReduce1, 176
  usedby compRepeatOrCollect, 173
  usedby compSubDomain1, 185
  usedby compSuchthat, 187
  usedby compSymbol, 346
$CategoryFrame

usedby compSubDomain1, 185
usedby compWithMappingMode1, 356
usedby parseHasRhs, 120
usedby parseHas, 119
$Category
usedby compConstructorCategory, 156
usedby compDefine1, 159
usedby compJoin, 166
$ConstructorNames
usedby compDefine1, 159
$DomainFrame
usedby s-process, 331
$DoubleFloat
usedby primitiveType, 345
$EchoLineStack
usedby fincomblock, 306
usedby preparse-echo, 90
usedby preparseReadLine1, 89
$EchoLines
usedby ncINTERPFILE, 365
$EmptyEnvironment
usedby s-process, 331
$EmptyMode
usedby compAdd, 141
usedby compArgumentsAndTryAgain, 352
usedby compCase1, 145
usedby compColonInside, 342
usedby compCons1, 154
usedby compDefine1, 159
usedby compForm1, 348
usedby compForm2, 350
usedby compIs, 165
usedby compMacro, 170
usedby compNoStacking, 338
usedby compPretend, 171
usedby compSetq1, 181
usedby compSubDomain1, 185
usedby compWhere, 189
usedby compWithMappingMode1, 356
usedby primitiveType, 345
usedby s-process, 331
usedby setqSingle, 182
$EmptyVector
usedby compVector, 188
$Expression
usedby compAtom, 343
usedby compForm1, 348
usedby compSymbol, 346
$FormalMapVariableList
usedby compColon, 151
usedby compSymbol, 346
usedby compTypeOf, 342
usedby compWithMappingMode1, 356
$Index
usedby s-process, 330
$InitialDomainsInScope
usedby spad, 329
$InteractiveFrame
usedby spad, 329
$InteractiveMode
usedby bumperrorcount, 297
usedby compileFileQuietly, 368
usedby compileSpad2Cmd, 316
usedby dollarTran, 289
usedby parseAnd, 107
usedby parseAtSign, 108
usedby parseCoerce, 110
usedby parseColon, 110
usedby parseHas, 119
usedby parseIf,ifTran, 121
usedby parseNot, 132
usedby postBigFloat, 201
usedby postDef, 210
usedby postError, 196
usedby postMDef, 216
usedby postReduce, 219
usedby spad, 329
usedby tuple2List, 302
$LocalFrame
usedby s-process, 331
$NRTaddForm
usedby compAdd, 141
usedby compSubDomain, 184
$NRTderivedTargetIfTrue
usedby compTopLevel, 334
$NegativeInteger
usedby primitiveType, 345

$NoValueMode
usedby compDefine1, 159
usedby compImport, 165
usedby compMacro, 170
usedby compRepeatOrCollect, 173
usedby compSeq1, 179
usedby compSymbol, 346
usedby setqSingle, 182
$NoValue
usedby compSymbol, 346
usedby parseAtom, 104
$NonNegativeInteger
usedby primitiveType, 345
$NumberOfArgsIfInteger
usedby compForm1, 348
$One
usedby compElt, 161
$PolyMode
usedby s-process, 331
$PositiveInteger
usedby primitiveType, 345
$PrettyPrint
usedby print-defun, 333
$QuickCode
usedby compWithMappingMode1, 356
usedby compileSpad2Cmd, 316
$QuickLet
usedby compileSpad2Cmd, 316
usedby setqSingle, 182
$ReadingFile
usedby ncINTERPFILE, 365
$Representation
usedby compNoStacking, 338
$StringCategory
usedby compString, 184
$String
usedby primitiveType, 345
$Symbol
usedby compSymbol, 345
$Translation
usedby s-process, 331
$TriangleVariableList
usedby compForm2, 350
$VariableCount
usedby s-process, 331
$Zero
usedby compElt, 161
$addFormLhs
usedby compAdd, 141
usedby compSubDomain, 184
$addForm
usedby compAdd, 141
usedby compCapsuleInner, 144
usedby compSubDomain, 184
$bootStrapMode
usedby comp2, 339
usedby compAdd, 141
usedby compCapsule, 143
usedby compColon, 151
$boot
usedby PARSE-FloatTok, 277
usedby PARSE-Primary1, 259
usedby PARSE-Quad, 263
usedby PARSE-Selector, 257
usedby PARSE-TokTail, 254
usedby aplTran1, 226
usedby aplTran, 226
usedby postAtom, 193
usedby postBigFloat, 201
usedby postColonColon, 207
usedby postDef, 210
usedby postForm, 196
usedby postIf, 213
usedby postMDef, 216
usedby quote-if-string, 280
usedby tuple2List, 302
$byConstructors, 368
usedby compilerDoit, 318
usedby preparse1, 81
defvar, 368
$comblocklist, 305
usedby fincomblock, 306
usedby preparse, 77
defvar, 305
$compErrorMessageStack
usedby compOrCroak1, 336
$compForModeIfTrue
usedby compSymbol, 346
$compStack
usedby compNoStacking1, 338
usedby compNoStacking, 338

usedby compOrCroak1, 336
usedby comp, 337
$compTimeSum
usedby compTopLevel, 334
$compUniquelyIfTrue
usedby s-process, 331
$compileOnlyCertainItems
usedby compileSpad2Cmd, 316
$constructorLineNumber
usedby preparse, 77
$constructorsSeen, 368
usedby compilerDoit, 318
usedby preparse1, 81
defvar, 368
$currentFunction
usedby s-process, 331
$defOp
usedby parseTransform, 103
usedby postError, 196
usedby postTransformCheck, 195
usedby setDefOp, 225
$defstack, 231
defvar, 231
$docList
usedby postDef, 210
usedby preparse, 77
$echolinestack, 72
usedby initialize-preparse, 73
usedby preparse1, 81
defvar, 72
$endTestList
usedby compReduce1, 176
$envHashTable
usedby compTopLevel, 334
$exitModeStack
usedby compExit, 162
usedby compLeave, 169
usedby compOrCroak1, 336
usedby compRepeatOrCollect, 173
usedby compReturn, 178
usedby compSeq1, 179
usedby compSeq, 179
usedby comp, 337
usedby modifyModeStack, 363
usedby s-process, 331
$exitMode
usedby s-process, 331
$e
usedby comp3, 340
usedby compHas, 163
usedby compReduce1, 176
usedby s-process, 331
$finalEnv
usedby compSeq1, 179
$forceAdd
usedby compTopLevel, 334
$formalArgList
usedby compDefine1, 159
usedby compReduce, 175
usedby compRepeatOrCollect, 173
usedby compSymbol, 346
usedby compWithMappingMode1, 356
usedby compWithMappingMode, 353
$formalMapVariables
usedby hasFormalMapVariable, 361
$formatArgList
usedby compWithMappingMode1, 356
$form
usedby compCapsuleInner, 144
usedby compDefine1, 159
usedby s-process, 331
usedby setqSingle, 182
$functorForm
usedby compAdd, 141
usedby compCapsule, 143
$functorLocalParameters
usedby compCapsuleInner, 144
usedby compSymbol, 346
$funnameTail
usedby compWithMappingMode1, 356
$funname
usedby compWithMappingMode1, 356
$f
usedby compileSpad2Cmd, 316
$genFVar
usedby s-process, 331
$genSDVar
usedby s-process, 331

$genno
usedby aplTran, 226
$getDomainCode
usedby compCapsuleInner, 144
$headerDocumentation
usedby postDef, 210
usedby preparse, 77
$index, 72
usedby initialize-preparse, 73
usedby preparseReadLine1, 89
usedby preparse, 77
defvar, 72
$initList
usedby compReduce1, 176
$insideCapsuleFunctionIfTrue
usedby compDefine1, 159
usedby s-process, 331
$insideCategoryIfTrue
usedby compCapsuleInner, 144
usedby compColon, 151
usedby compDefine1, 159
usedby s-process, 331
$insideCategoryPackageIfTrue
usedby compCapsuleInner, 144
$insideCoerceInteractiveHardIfTrue
usedby s-process, 331
$insideCompTypeOf
usedby comp3, 340
usedby compTypeOf, 342
$insideConstructIfTrue
usedby parseColon, 110
usedby parseConstruct, 105
$insideExpressionIfTrue
usedby compCapsule, 143
usedby compColon, 151
usedby compDefine1, 159
usedby compExpression, 347
usedby compMakeDeclaration, 362
usedby compSeq1, 179
usedby compWhere, 189
usedby s-process, 331
$insideFunctorIfTrue
usedby compColon, 151
usedby compDefine1, 159
usedby s-process, 331
$insidePostCategoryIfTrue
usedby postCategory, 202
usedby postWith, 225
$insideSetqSingleIfTrue
usedby setqSingle, 182
$insideWhereIfTrue
usedby compDefine1, 159
usedby compWhere, 189
usedby s-process, 331
$is-eqlist, 232
defvar, 232
$is-gensymlist, 232
defvar, 232
$is-spill-list, 231
defvar, 231
$is-spill, 231
defvar, 231
$killOptimizeIfTrue
usedby compTopLevel, 334
usedby compWithMappingMode1, 356
$leaveLevelStack
usedby compLeave, 169
usedby compRepeatOrCollect, 173
usedby s-process, 331
$leaveMode
usedby s-process, 331
$level
usedby compOrCroak1, 336
$lhsOfColon
usedby compColon, 151
usedby compSubsetCategory, 186
$lhs
usedby parseDEF, 111
usedby parseMDEF, 131
$linelist, 72
usedby initialize-preparse, 73
usedby preparse1, 81
usedby preparseReadLine1, 89
defvar, 72
$line
usedby Advance-Char, 95
usedby current-char, 287
usedby line-advance-char, 94
usedby line-at-end-p, 93
usedby line-clear, 92
usedby line-new-line, 94

usedby line-next-char, 93
usedby line-past-end-p, 93
usedby line-print, 93, 94
usedby match-advance-string, 279
usedby match-string, 278
$lisplibSuperDomain
usedby compSubDomain1, 185
$lisplib
usedby /RQ,LIB, 319
usedby comp2, 339
$macroIfTrue
usedby compDefine, 158
usedby compMacro, 170
$macroassoc
usedby s-process, 331
$maxSignatureLineNumber
usedby postDef, 210
usedby preparse, 77
$m
usedby compileSpad2Cmd, 316
$newCompilerUnionFlag
usedby compColonInside, 342
usedby compPretend, 171
$newComp
usedby compileSpad2Cmd, 316
$newConlist
usedby compileSpad2Cmd, 316
usedby compiler, 313
$newspad
usedby s-process, 331
$noEnv
usedby compColon, 151
$noParseCommands
usedby PARSE-SpecialCommand, 243
$noSubsumption
usedby spad, 329
$options
usedby compileSpad2Cmd, 316
usedby compileSpadLispCmd, 365
usedby compiler, 313
$op
usedby compDefine1, 159
usedby compSubDomain1, 185
usedby parseDollarGreaterEqual, 114
usedby parseDollarGreaterThan, 114
usedby parseDollarLessEqual, 115
usedby parseDollarNotEqual, 116
usedby parseGreaterEqual, 117
usedby parseGreaterThan, 118
usedby parseLessEqual, 129
usedby parseNotEqual, 132
usedby parseTran, 103
$packagesUsed
usedby comp2, 339
usedby compAdd, 141
usedby compDefine, 158
usedby compTopLevel, 334
$postStack
usedby displayPreCompilationErrors, 296
usedby postError, 196
usedby s-process, 331
$prefix
usedby compDefine1, 159
$preparse-last-line, 72
usedby initialize-preparse, 73
usedby preparse1, 81
usedby preparseReadLine1, 89
usedby preparse, 77
defvar, 72
$preparseReportIfTrue
usedby preparse, 77
$previousTime
usedby s-process, 331
$profileCompiler
usedby setqSingle, 182
$reportExitModeStack
usedby modifyModeStack, 363
$resolveTimeSum
usedby compTopLevel, 334
$returnMode
usedby compReturn, 178
usedby s-process, 331
$scanIfTrue
usedby compOrCroak1, 336
usedby compileSpad2Cmd, 316
$semanticErrorStack
usedby s-process, 331
$sideEffectsList
usedby compReduce1, 176
$signature

usedby compCapsuleInner, 144
$skipme
usedby preparse1, 81
usedby preparse, 77
$sourceFileTypes
usedby compileSpad2Cmd, 316
$spad-errors
usedby bumperrorcount, 297
$spad
usedby quote-if-string, 281
$stack
usedby reduce-stack, 292
usedby stack-/-empty, 98
usedby stack-clear, 97
usedby stack-load, 97
usedby stack-pop, 98
usedby stack-push, 98
$s
usedby compOrCroak1, 336
$tokenCommands
usedby PARSE-SpecialCommand, 243
$token
usedby current-token, 100
usedby make-symbol-of, 284
usedby match-advance-string, 279
usedby next-token, 100
usedby prior-token, 99
usedby token-install, 100
usedby token-print, 101
usedby valid-tokens, 100
$top-level
usedby s-process, 331
$topOp
usedby displayPreCompilationErrors, 296
usedby s-process, 331
usedby setDefOp, 225
$tripleCache
usedby compDefine, 158
$tripleHits
usedby compDefine, 158
$until
usedby compReduce1, 176
usedby compRepeatOrCollect, 173
$vl, 232
defvar, 232
$warningStack
usedby s-process, 331

, 32

abbreviation?
calledby parseHasRhs, 120
action, 291
calledby PARSE-AnyId, 268
calledby PARSE-Category, 248
calledby PARSE-CommandTail, 245
calledby PARSE-Data, 266
calledby PARSE-FloatExponent, 261
calledby PARSE-GliphTok, 268
calledby PARSE-Infix, 253
calledby PARSE-NBGliphTok, 267
calledby PARSE-NewExpr, 241
calledby PARSE-OpenBrace, 270
calledby PARSE-OpenBracket, 270
calledby PARSE-Operation, 251
calledby PARSE-Prefix, 252
calledby PARSE-ReductionOp, 255
calledby PARSE-Sexpr1, 266
calledby PARSE-SpecialCommand, 243
calledby PARSE-SpecialKeyWord, 242
calledby PARSE-Suffix, 272
calledby PARSE-TokTail, 254
calledby PARSE-TokenCommandTail, 244
calledby PARSE-TokenList, 244
defun, 291
add, 198
defplist, 198
add-parens-and-semis-to-line, 87
calledby parsepiles, 86
calls addclose, 87
calls drop, 87
calls infixtok, 87
calls nonblankloc, 87
defun, 87
addBinding[5]
called by setqSingle, 182
called by spad, 329

addclose, 304
calledby add-parens-and-semis-to-line, 87
calls suffix, 304
defun, 304
addContour
calledby compWhere, 189
addDomain
calledby comp2, 339
calledby comp3, 340
calledby compAtSign, 142
calledby compCapsule, 143
calledby compCase, 145
calledby compCoerce, 148
calledby compColonInside, 342
calledby compColon, 151
calledby compElt, 161
calledby compForm1, 348
calledby compImport, 165
calledby compPretend, 171
calledby compSubDomain1, 185
addEmptyCapsuleIfNecessary
calledby compDefine1, 159
addInformation
calledby compCapsuleInner, 144
Advance-Char, 95
calls Line-Advance-Char, 95
calls Line-At-End-P, 95
calls current-char, 95
calls next-line, 95
uses $line, 95
defun, 95
advance-char
calledby skip-blanks, 278
advance-token, 286
calledby PARSE-AnyId, 268
calledby PARSE-FloatExponent, 261
calledby PARSE-GliphTok, 268
calledby PARSE-Infix, 253
calledby PARSE-NBGliphTok, 267
calledby PARSE-OpenBrace, 270
calledby PARSE-OpenBracket, 270
calledby PARSE-Prefix, 253
calledby PARSE-ReductionOp, 255
calledby PARSE-Suffix, 272
calledby PARSE-TokenList, 244
calledby parse-argument-designator, 301
calledby parse-identifier, 299
calledby parse-keyword, 300
calledby parse-number, 300
calledby parse-spadstring, 298
calledby parse-string, 299
calls copy-token, 286
calls current-token, 286
calls try-get-token, 286
uses current-token, 286
uses valid-tokens, 286
defun, 286
and, 107
defplist, 107
aplTran, 226
calledby postTransform, 191
calls aplTran1, 226
calls containsBang, 226
uses $boot, 226
uses $genno, 226
defun, 226
aplTran1, 226
calledby aplTran1, 226
calledby aplTranList, 228
calledby aplTran, 226
calledby hasAplExtension, 228
calls aplTran1, 226
calls aplTranList, 226
calls hasAplExtension, 226
calls nreverse0, 226
calls , 226
uses $boot, 226
defun, 226
aplTranList, 228
calledby aplTran1, 226
calledby aplTranList, 228
calls aplTran1, 228
calls aplTranList, 228
defun, 228
applyMapping
calledby comp3, 340
argsToSig, 361
calledby compLambda, 168
defun, 361
assignError

calledby setqSingle, 182
assoc
calledby compColon, 151
calledby compForm2, 350
assq[5]
called by freelist, 363
atEndOfLine
calledby PARSE-TokenCommandTail, 244
augModemapsFromDomain1
calledby compForm1, 348
calledby setqSingle, 182

Bang, 290
defmacro, 290
bang
calledby PARSE-Category, 247
calledby PARSE-CommandTail, 245
calledby PARSE-Conditional, 275
calledby PARSE-Form, 255
calledby PARSE-Import, 249
calledby PARSE-IteratorTail, 271
calledby PARSE-Seg, 274
calledby PARSE-Sexpr1, 267
calledby PARSE-SpecialCommand, 243
calledby PARSE-TokenCommandTail, 243
bfp-
calledby PARSE-FloatTok, 277
blankp, 305
calledby nonblankloc, 308
defun, 305
Block, 201
defplist, 201
bootStrapError
calledby compCapsule, 143
bright
calledby parseInBy, 125
calledby postForm, 196
browserAutoloadOnceTrigger
calledby compileSpad2Cmd, 316
bumperrorcount, 297
calledby postError, 196
uses $InteractiveMode, 297
uses $spad-errors, 297
defun, 297

canReturn
calledby compIf, 164
capsule, 143
defplist, 143
case, 144
defplist, 144
catches
compOrCroak1, 337
preparse1, 81
spad, 329
category, 108, 147, 202
defplist, 108, 147, 202
char-eq, 288
calledby PARSE-FloatBase, 260
calledby PARSE-TokTail, 254
defun, 288
char-ne, 288
calledby PARSE-FloatBase, 260
calledby PARSE-Selector, 257
defun, 288
chaseInferences
calledby compHas, 163
checkWarning, 301
calledby postCapsule, 199
calls concat, 301
calls postError, 301
defun, 301
coerce
calledby compAtSign, 142
calledby compCase, 145
calledby compCoerce1, 149
calledby compCoerce, 148
calledby compColonInside, 342
calledby compForm1, 348
calledby compHas, 163
calledby compIf, 164
calledby compIs, 165
calledby convert, 344
coerceable
calledby compForm1, 348
coerceByModemap
calledby compCoerce1, 149
coerceExit
calledby compRepeatOrCollect, 173

collect, 172, 204
  calledby floatexpid, 289
  defplist, 172, 204
comma2Tuple, 207
  calledby postComma, 207
  calledby postConstruct, 208
  calls postFlatten, 207
  defun, 207
comp, 337
  calledby compAdd, 140
  calledby compArgumentsAndTryAgain, 352
  calledby compAtSign, 142
  calledby compCase1, 145
  calledby compCoerce1, 149
  calledby compColonInside, 342
  calledby compCons1, 154
  calledby compExit, 162
  calledby compForm1, 348
  calledby compIs, 165
  calledby compLeave, 169
  calledby compList, 347
  calledby compOrCroak1, 336
  calledby compPretend, 171
  calledby compReduce1, 175
  calledby compRepeatOrCollect, 173
  calledby compReturn, 178
  calledby compSeqItem, 180
  calledby compSubsetCategory, 186
  calledby compSuchthat, 187
  calledby compVector, 188
  calledby compWhere, 188
  calledby compWithMappingMode1, 355
  calledby setqSetelt, 181
  calledby setqSingle, 182
  calls compNoStacking, 337
  uses $compStack, 337
  uses $exitModeStack, 337
  defun, 337
comp-tran
  calledby compWithMappingMode1, 356
comp2, 339
  calledby compNoStacking1, 338
  calledby compNoStacking, 338
  calls addDomain, 339
  calls comp3, 339
  calls insert, 339
  calls isDomainForm, 339
  calls isFunctor, 339
  calls nequal, 339
  calls opOf, 339
  uses $bootStrapMode, 339
  uses $lisplib, 339
  uses $packagesUsed, 339
  defun, 339
comp3, 339
  calledby comp2, 339
  calledby compTypeOf, 342
  calls addDomain, 340
  calls applyMapping, 340
  calls compApply, 340
  calls compAtom, 340
  calls compCoerce, 340
  calls compColon, 340
  calls compExpression, 340
  calls compTypeOf, 340
  calls compWithMappingMode, 340
  calls getDomainsInScope, 340
  calls getmode, 340
  calls member[5], 340
  calls pname[5], 340
  calls stringPrefix?, 340
  uses $e, 340
  uses $insideCompTypeOf, 340
  defun, 339
compAdd, 140
  calls NRTgetLocalIndex, 141
  calls compCapsule, 141
  calls compOrCroak, 141
  calls compSubDomain1, 140
  calls compTuple2Record, 141
  calls comp, 140
  calls nreverse0, 140
  calls pairp, 140
  calls qcar, 140
  calls qcdr, 140
  uses /editfile, 141
  uses $EmptyMode, 141
  uses $NRTaddForm, 141
  uses $addFormLhs, 141

uses $addForm, 141
uses $bootStrapMode, 141
uses $functorForm, 141
uses $packagesUsed, 141
defun, 140
compApply
calledby comp3, 340
compArgumentsAndTryAgain, 352
calledby compForm, 348
calls compForm1, 352
calls comp, 352
uses $EmptyMode, 352
defun, 352
compAtom, 343
calledby comp3, 340
calls compAtomWithModemap, 343
calls compList, 343
calls compSymbol, 343
calls compVector, 343
calls convert, 343
calls get, 343
calls isSymbol, 343
calls modeIsAggregateOf, 343
calls primitiveType, 343
uses $Expression, 343
defun, 343
compAtomWithModemap
calledby compAtom, 343
compAtSign, 142
calledby compLambda, 168
calls addDomain, 142
calls coerce, 142
calls comp, 142
defun, 142
compBoolean
calledby compIf, 164
compCapsule, 143
calledby compAdd, 141
calledby compSubDomain, 184
calls addDomain, 143
calls bootStrapError, 143
calls compCapsuleInner, 143
uses $bootStrapMode, 143
uses $functorForm, 143
uses $insideExpressionIfTrue, 143
uses editfile, 143
defun, 143
compCapsuleInner, 144
calledby compCapsule, 143
calls addInformation, 144
calls compCapsuleItems, 144
calls mkpf, 144
calls processFunctorOrPackage, 144
uses $addForm, 144
uses $form, 144
uses $functorLocalParameters, 144
uses $getDomainCode, 144
uses $insideCategoryIfTrue, 144
uses $insideCategoryPackageIfTrue, 144
uses $signature, 144
defun, 144
compCapsuleItems
calledby compCapsuleInner, 144
compCase, 145
calls addDomain, 145
calls coerce, 145
calls compCase1, 145
defun, 145
compCase1, 145
calledby compCase, 145
calls comp, 145
calls getModemapList, 145
calls modeEqual, 145
calls nreverse0, 145
uses $Boolean, 145
uses $EmptyMode, 145
defun, 145
compCat, 147
calls getl, 147
defun, 147
compCategory, 147
calls compCategoryItem, 147
calls mkExplicitCategoryFunction, 147
calls qcar, 147
calls qcdr, 147
calls resolve, 147
calls systemErrorHere, 148
defun, 147
compCategoryItem
calledby compCategory, 147

compCoerce, 148
calledby comp3, 340
calls addDomain, 148
calls coerce, 148
calls compCoerce1, 148
calls getmode, 148
defun, 148
compCoerce1, 149
calledby compCoerce, 148
calls coerceByModemap, 149
calls coerce, 149
calls comp, 149
calls mkq, 149
calls msubst, 149
calls resolve, 149
defun, 149
compColon, 150
calledby comp3, 340
calledby compColon, 151
calledby compMakeDeclaration, 362
calls addDomain, 151
calls assoc, 151
calls compColonInside, 151
calls compColon, 151
calls eqsubstlist, 151
calls genSomeVariable, 151
calls getDomainsInScope, 151
calls getmode, 151
calls isCategoryForm, 151
calls isDomainForm, 151
calls length, 151
calls makeCategoryForm, 151
calls member[5], 151
calls nreverse0, 151
calls put, 151
calls systemErrorHere, 151
calls take, 151
calls unknownTypeError, 151
uses $FormalMapVariableList, 151
uses $bootStrapMode, 151
uses $insideCategoryIfTrue, 151
uses $insideExpressionIfTrue, 151
uses $insideFunctorIfTrue, 151
uses $lhsOfColon, 151
uses $noEnv, 151
defun, 150
compColonInside, 342
calledby compColon, 151
calls addDomain, 342
calls coerce, 342
calls comp, 342
calls opOf, 342
calls stackSemanticError, 342
calls stackWarning, 342
uses $EmptyMode, 342
uses $newCompilerUnionFlag, 342
defun, 342
compCons, 154
calls compCons1, 154
calls compForm, 154
defun, 154
compCons1, 154
calledby compCons, 154
calls comp, 154
calls convert, 154
calls pairp, 154
calls qcar, 154
calls qcdr, 154
uses $EmptyMode, 154
defun, 154
compConstruct, 157
calls compForm, 157
calls compList, 157
calls compVector, 157
calls convert, 157
calls getDomainsInScope, 157
calls modeIsAggregateOf, 157
defun, 157
compConstructorCategory, 156
calls resolve, 156
uses $Category, 156
defun, 156
compDefine, 158
calls compDefine1, 158
uses $macroIfTrue, 158
uses $packagesUsed, 158
uses $tripleCache, 158
uses $tripleHits, 158
defun, 158
compDefine1, 158
calledby compDefine1, 158
calledby compDefine, 158

calls addEmptyCapsuleIfNecessary, 159
calls compDefWhereClause, 158
calls compDefine1, 158
calls compDefineAddSignature, 158
calls compDefineCapsuleFunction, 159
calls compDefineCategory, 158
calls compDefineFunctor, 159
calls compInternalFunction, 158
calls getAbbreviation, 159
calls getSignatureFromMode, 158
calls getTargetFromRhs, 159
calls giveFormalParametersValues, 159
calls isDomainForm, 158
calls isMacro, 158
calls length, 159
calls macroExpand, 158
calls stackAndThrow, 159
calls strconc, 159
uses $Category, 159
uses $ConstructorNames, 159
uses $EmptyMode, 159
uses $NoValueMode, 159
uses $formalArgList, 159
uses $form, 159
uses $insideCapsuleFunctionIfTrue, 159
uses $insideCategoryIfTrue, 159
uses $insideExpressionIfTrue, 159
uses $insideFunctorIfTrue, 159
uses $insideWhereIfTrue, 159
uses $op, 159
uses $prefix, 159
defun, 158
compDefineAddSignature
calledby compDefine1, 158
compDefineCapsuleFunction
calledby compDefine1, 159
compDefineCategory
calledby compDefine1, 158
compDefineFunctor
calledby compDefine1, 159
compDefWhereClause
calledby compDefine1, 158
compElt, 160
calls addDomain, 161
calls compForm, 160
calls convert, 161
calls getDeltaEntry, 161
calls getModemapListFromDomain, 161
calls isDomainForm, 160
calls length, 161
calls nequal, 161
calls opOf, 161
calls stackMessage, 161
calls stackWarning, 161
uses $One, 161
uses $Zero, 161
defun, 160
compExit, 162
calls comp, 162
calls modifyModeStack, 162
calls stackMessageIfNone, 162
uses $exitModeStack, 162
defun, 162
compExpression, 347
calledby comp3, 340
calls compForm, 347
calls getl, 347
uses $insideExpressionIfTrue, 347
defun, 347
compExpressionList
calledby compForm1, 348
compForm, 348
calledby compConstruct, 157
calledby compCons, 154
calledby compElt, 160
calledby compExpression, 347
calls compArgumentsAndTryAgain, 348
calls compForm1, 348
calls stackMessageIfNone, 348
defun, 348
compForm1, 348
calledby compArgumentsAndTryAgain, 352
calledby compForm, 348
calls addDomain, 348

calls augModemapsFromDomain1, 348
calls coerceable, 348
calls coerce, 348
calls compExpressionList, 348
calls compForm2, 348
calls compOrCroak, 348
calls compToApply, 348
calls comp, 348
calls getFormModemaps, 348
calls length, 348
calls nreverse0, 348
calls outputComp, 348
uses $EmptyMode, 348
uses $Expression, 348
uses $NumberOfArgsIfInteger, 348
defun, 348
compForm2, 350
calledby compForm1, 348
calls PredImplies, 350
calls assoc, 350
calls compForm3, 350
calls compFormPartiallyBottomUp, 350
calls compUniquely, 350
calls isSimple, 350
calls length, 350
calls nreverse0, 350
calls sublis, 350
calls take, 350
uses $EmptyMode, 350
uses $TriangleVariableList, 350
defun, 350
compForm3
calledby compForm2, 350
compForMode
calledby compJoin, 166
compFormPartiallyBottomUp
calledby compForm2, 350
compFromIf
calledby compIf, 164
compHas, 163
calls chaseInferences, 163
calls coerce, 163
calls compHasFormat, 163
uses $e, 163
defun, 163
compHasFormat
calledby compHas, 163
compIf, 164
calls canReturn, 164
calls coerce, 164
calls compBoolean, 164
calls compFromIf, 164
calls intersectionEnvironment, 164
calls quotify, 164
calls resolve, 164
uses $Boolean, 164
defun, 164
compile-lib-file, 367
calledby recompile-lib-file-if-necessary, 366
defun, 367
compileFileQuietly, 368
uses *standard-output*, 368
uses $InteractiveMode, 368
defun, 368
compiler, 312
calls compileSpad2Cmd, 313
calls compileSpadLispCmd, 313
calls findfile, 313
calls helpSpad2Cmd[5], 313
calls mergePathnames[5], 313
calls namestring[5], 313
calls pathnameType[5], 313
calls pathname[5], 313
calls selectOptionLC[5], 313
calls throwKeyedMsg, 313
uses /editfile, 313
uses $newConlist, 313
uses $options, 313
defun, 312
compilerDoit, 318
calledby compileSpad2Cmd, 316
calls /RQ,LIB, 318
calls /rf[5], 318
calls /rq[5], 318
calls member[5], 318
calls opOf, 318
calls sayBrightly, 318
uses $byConstructors, 318
uses $constructorsSeen, 318

defun, 318
compilerDoitWithScreenedLisplib
calledby compileSpad2Cmd, 316
compileSpad2Cmd, 314
calledby compiler, 313
calls browserAutoloadOnceTrigger, 316
calls compilerDoitWithScreenedLisplib, 316
calls compilerDoit, 316
calls convertSpadToAsFile, 316
calls error, 316
calls extendLocalLibdb, 316
calls namestring[5], 316
calls nequal, 316
calls object2String, 316
calls pathnameType[5], 316
calls pathname[5], 316
calls sayKeyedMsg[5], 316
calls selectOptionLC[5], 316
calls spad2AsTranslatorAutoloadOnceTrigger, 316
calls spadPrompt, 316
calls strconc, 316
calls terminateSystemCommand[5], 316
calls throwKeyedMsg, 316
calls updateSourceFiles[5], 316
uses /editfile, 316
uses $InteractiveMode, 316
uses $QuickCode, 316
uses $QuickLet, 316
uses $compileOnlyCertainItems, 316
uses $f, 316
uses $m, 316
uses $newComp, 316
uses $newConlist, 316
uses $options, 316
uses $scanIfTrue, 316
uses $sourceFileTypes, 316
defun, 314
compileSpadLispCmd, 365
calledby compiler, 313
calls fnameMake[5], 365
calls fnameReadable?[5], 365
calls localdatabase[5], 365
calls namestring[5], 365
calls object2String, 365
calls pathnameDirectory[5], 365
calls pathnameName[5], 365
calls pathnameType[5], 365
calls pathname[5], 365
calls recompile-lib-file-if-necessary, 365
calls sayKeyedMsg[5], 365
calls selectOptionLC[5], 365
calls spadPrompt, 365
calls terminateSystemCommand[5], 365
calls throwKeyedMsg, 365
uses $options, 365
defun, 365
compImport, 165
calls addDomain, 165
uses $NoValueMode, 165
defun, 165
compInternalFunction
calledby compDefine1, 158
compIs, 165
calls coerce, 165
calls comp, 165
uses $Boolean, 165
uses $EmptyMode, 165
defun, 165
compIterator
calledby compReduce1, 175
calledby compRepeatOrCollect, 173
compJoin, 166
calls compForMode, 166
calls compJoin,getParms, 166
calls convert, 166
calls isCategoryForm, 166
calls nreverse0, 166
calls pairp, 166
calls qcar, 166
calls qcdr, 166
calls stackSemanticError, 166
calls union, 166
calls wrapDomainSub, 166
uses $Category, 166
defun, 166
compJoin,getParms

calledby compJoin, 166
compLambda, 168
calledby compWithMappingMode1, 355
calls argsToSig, 168
calls compAtSign, 168
calls qcar, 168
calls qcdr, 168
calls stackAndThrow, 168
defun, 168
compLeave, 169
calls comp, 169
calls modifyModeStack, 169
uses $exitModeStack, 169
uses $leaveLevelStack, 169
defun, 169
compList, 347
calledby compAtom, 343
calledby compConstruct, 157
calls comp, 347
defun, 347
compMacro, 170
calls formatUnabbreviated, 170
calls macroExpand, 170
calls put, 170
calls qcar, 170
calls sayBrightly, 170
uses $EmptyMode, 170
uses $NoValueMode, 170
uses $macroIfTrue, 170
defun, 170
compMakeDeclaration, 362
calledby compSetq1, 181
calledby compSubDomain1, 185
calledby compWithMappingMode1, 355
calls compColon, 362
uses $insideExpressionIfTrue, 362
defun, 362
compNoStacking, 338
calledby comp, 337
calls comp2, 338
calls compNoStacking1, 338
uses $EmptyMode, 338
uses $Representation, 338
uses $compStack, 338
defun, 338
compNoStacking1, 338
calledby compNoStacking, 338
calls comp2, 338
calls get, 338
uses $compStack, 338
defun, 338
compOrCroak, 335
calledby compAdd, 141
calledby compForm1, 348
calledby compRepeatOrCollect, 173
calledby compSubDomain1, 185
calledby compTopLevel, 334
calls compOrCroak1, 335
defun, 335
compOrCroak1, 336
calledby compOrCroak, 335
calls compOrCroak1,compactify, 336
calls comp, 336
calls displayComp, 336
calls displaySemanticErrors, 336
calls mkErrorExpr, 336
calls say, 336
calls stackSemanticError, 336
calls userError, 336
uses $compErrorMessageStack, 336
uses $compStack, 336
uses $exitModeStack, 336
uses $level, 336
uses $scanIfTrue, 336
uses $s, 336
catches, 337
defun, 336
compOrCroak1,compactify, 364
calledby compOrCroak1,compactify, 364
calledby compOrCroak1, 336
calls compOrCroak1,compactify, 364
calls lassoc, 364
defun, 364
compPretend, 171
calls addDomain, 171
calls comp, 171
calls nequal, 171
calls opOf, 171
calls stackSemanticError, 171

calls stackWarning, 171
uses $EmptyMode, 171
uses $newCompilerUnionFlag, 171
defun, 171
compQuote, 172
defun, 172
compReduce, 175
calls compReduce1, 175
uses $formalArgList, 175
defun, 175
compReduce1, 175
calledby compReduce, 175
calls compIterator, 175
calls comp, 175
calls getIdentity, 175
calls msubst, 175
calls nreverse0, 175
calls parseTran, 175
calls systemError, 175
uses $Boolean, 176
uses $endTestList, 176
uses $e, 176
uses $initList, 176
uses $sideEffectsList, 176
uses $until, 176
defun, 175
compRepeatOrCollect, 173
calls coerceExit, 173
calls compIterator, 173
calls compOrCroak, 173
calls comp, 173
calls length, 173
calls modeIsAggregateOf, 173
calls msubst, 173
calls stackMessage, 173
calls , 173
uses $Boolean, 173
uses $NoValueMode, 173
uses $exitModeStack, 173
uses $formalArgList, 173
uses $leaveLevelStack, 173
uses $until, 173
defun, 173
compReturn, 177
calls comp, 178
calls modifyModeStack, 178
calls nequal, 177
calls resolve, 177
calls stackSemanticError, 177
calls userError, 177
uses $exitModeStack, 178
uses $returnMode, 178
defun, 177
compSeq, 179
calls compSeq1, 179
uses $exitModeStack, 179
defun, 179
compSeq1, 179
calledby compSeq, 179
calls compSeqItem, 179
calls mkq, 179
calls nreverse0, 179
calls replaceExitEtc, 179
uses $NoValueMode, 179
uses $exitModeStack, 179
uses $finalEnv, 179
uses $insideExpressionIfTrue, 179
defun, 179
compSeqItem, 180
calledby compSeq1, 179
calls comp, 180
calls macroExpand, 180
defun, 180
compSetq, 180
calledby compSetq1, 181
calls compSetq1, 180
defun, 180
compSetq1, 181
calledby compSetq, 180
calls compMakeDeclaration, 181
calls compSetq, 181
calls identp[5], 181
calls qcar, 181
calls qcdr, 181
calls setqMultiple, 181
calls setqSetelt, 181
calls setqSingle, 181
uses $EmptyMode, 181
defun, 181
compString, 184
calls resolve, 184
uses $StringCategory, 184

defun, 184
compSubDomain, 184
calls compCapsule, 184
calls compSubDomain1, 184
uses $NRTaddForm, 184
uses $addFormLhs, 184
uses $addForm, 184
defun, 184
compSubDomain1, 185
calledby compAdd, 140
calledby compSubDomain, 184
calls addDomain, 185
calls compMakeDeclaration, 185
calls compOrCroak, 185
calls evalAndRwriteLispForm, 185
calls lispize, 185
calls stackSemanticError, 185
uses $Boolean, 185
uses $CategoryFrame, 185
uses $EmptyMode, 185
uses $lisplibSuperDomain, 185
uses $op, 185
defun, 185
compSubsetCategory, 186
calls comp, 186
calls msubst, 186
calls put, 186
uses $lhsOfColon, 186
defun, 186
compSuchthat, 187
calls comp, 187
calls put, 187
uses $Boolean, 187
defun, 187
compSymbol, 345
calledby compAtom, 343
calls NRTgetLocalIndex, 345
calls errorRef, 345
calls getmode, 345
calls get, 345
calls isFunction, 345
calls member[5], 345
calls stackMessage, 345
uses $Boolean, 346
uses $Expression, 346
uses $FormalMapVariableList, 346
uses $NoValueMode, 346
uses $NoValue, 346
uses $Symbol, 345
uses $compForModeIfTrue, 346
uses $formalArgList, 346
uses $functorLocalParameters, 346
defun, 345
compToApply
calledby compForm1, 348
compTopLevel, 334
calledby s-process, 330
calls compOrCroak, 334
calls newComp, 334
uses $NRTderivedTargetIfTrue, 334
uses $compTimeSum, 334
uses $envHashTable, 334
uses $forceAdd, 334
uses $killOptimizeIfTrue, 334
uses $packagesUsed, 334
uses $resolveTimeSum, 334
defun, 334
compTuple2Record
calledby compAdd, 141
compTypeOf, 342
calledby comp3, 340
calls comp3, 342
calls eqsubstlist, 342
calls get, 342
calls put, 342
uses $FormalMapVariableList, 342
uses $insideCompTypeOf, 342
defun, 342
compUniquely
calledby compForm2, 350
compVector, 187
calledby compAtom, 343
calledby compConstruct, 157
calls comp, 188
uses $EmptyVector, 188
defun, 187
compWhere, 188
calls addContour, 189
calls comp, 188
calls deltaContour, 189
calls macroExpand, 188
uses $EmptyMode, 189

uses $insideExpressionIfTrue, 189
uses $insideWhereIfTrue, 189
defun, 188
compWithMappingMode, 353
calledby comp3, 340
calls compWithMappingMode1, 353
uses $formalArgList, 353
defun, 353
compWithMappingMode1, 353
calledby compWithMappingMode, 353
calls comp-tran, 356
calls compLambda, 355
calls compMakeDeclaration, 355
calls comp, 355
calls extendsCategoryForm, 355
calls extractCodeAndConstructTriple, 356
calls freelist, 356
calls get, 355
calls hasFormalMapVariable, 355
calls isFunctor, 355
calls optimizeFunctionDef, 356
calls qcar, 355
calls qcdr, 355
calls stackAndThrow, 355
calls take, 355
uses $CategoryFrame, 356
uses $EmptyMode, 356
uses $FormalMapVariableList, 356
uses $QuickCode, 356
uses $formalArgList, 356
uses $formatArgList, 356
uses $funnameTail, 356
uses $funname, 356
uses $killOptimizeIfTrue, 356
defun, 353
concat
calledby checkWarning, 301
cons, 154
defplist, 154
consProplistOf
calledby setqSingle, 182
construct, 105, 156, 208
defplist, 105, 156, 208
contained
calledby parseCategory, 109
containsBang, 229
calledby aplTran, 226
calledby containsBang, 229
calls containsBang, 229
defun, 229
convert, 344
calledby compAtom, 343
calledby compCons1, 154
calledby compConstruct, 157
calledby compElt, 161
calledby compJoin, 166
calledby setqSingle, 182
calls coerce, 344
calls resolve, 344
defun, 344
convertSpadToAsFile
calledby compileSpad2Cmd, 316
copy
calledby modifyModeStack, 363
copy-token
calledby PARSE-TokTail, 254
calledby advance-token, 286
croak
calledby drop, 305
curoutstream
usedby s-process, 331
current-char, 287
calledby Advance-Char, 95
calledby PARSE-FloatBasePart, 261
calledby PARSE-FloatBase, 260
calledby PARSE-FloatExponent, 261
calledby PARSE-Selector, 257
calledby PARSE-TokTail, 254
calledby match-string, 278
calledby skip-blanks, 278
uses $line, 287
uses current-line, 287
defun, 287
current-fragment, 90
defvar, 90
current-line, 92
usedby PARSE-Category, 248
usedby current-char, 287
usedby next-char, 288
defvar, 92

current-symbol, 284
calledby PARSE-AnyId, 268
calledby PARSE-ElseClause, 275
calledby PARSE-FloatBase, 260
calledby PARSE-FloatExponent, 261
calledby PARSE-Infix, 253
calledby PARSE-NewExpr, 241
calledby PARSE-OpenBrace, 270
calledby PARSE-OpenBracket, 270
calledby PARSE-Operation, 251
calledby PARSE-Prefix, 252
calledby PARSE-Primary1, 258
calledby PARSE-ReductionOp, 255
calledby PARSE-Selector, 257
calledby PARSE-SpecialCommand, 243
calledby PARSE-SpecialKeyWord, 242
calledby PARSE-Suffix, 272
calledby PARSE-TokTail, 254
calledby PARSE-TokenList, 244
calledby isTokenDelimiter, 281
calls current-token, 284
calls make-symbol-of, 284
defun, 284
current-token, 100, 285
calledby PARSE-FloatBasePart, 261
calledby PARSE-SpecialKeyWord, 242
calledby advance-token, 286
calledby current-symbol, 284
calledby match-advance-string, 279
calledby match-current-token, 283
calledby next-token, 286
calls try-get-token, 285
usedby advance-token, 286
usedby current-token, 285
uses $token, 100
uses current-token, 285
uses valid-tokens, 285
defun, 285
defvar, 100
curstrm
calledby s-process, 330

dcq
calledby preparseReadLine, 88
decodeScripts, 230
calledby decodeScripts, 230
calledby getScriptName, 230
calls decodeScripts, 230
calls qcar, 230
calls qcdr, 230
calls strconc, 230
defun, 230
deepestExpression, 229
calledby deepestExpression, 229
calledby hasAplExtension, 228
calls deepestExpression, 229
defun, 229
def, 111, 158
defplist, 111, 158
def-process
calledby s-process, 330
def-rename, 333
calledby s-process, 330
calls def-rename1, 333
defun, 333
def-rename1, 333
calledby def-rename1, 333
calledby def-rename, 333
calls def-rename1, 333
defun, 333
definition-name, 241
usedby PARSE-NewExpr, 241
defvar, 241
defmacro
Bang, 290
line-clear, 92
must, 290
nth-stack, 304
pop-stack-1, 302
pop-stack-2, 303
pop-stack-3, 303
pop-stack-4, 303
reduce-stack-clear, 292
stack-/-empty, 98
star, 291
defplist, 107, 140, 142, 186, 200
+− >, 168
− >, 215
<=, 128

==>, 216
=>, 212
>, 118
>=, 116, 117
,, 207
/, 222
:, 110, 150, 206
::, 109, 148, 206
:BF:, 201
;, 220
==, 210
add, 198
and, 107
Block, 201
capsule, 143
case, 144
category, 108, 147, 202
collect, 172, 204
cons, 154
construct, 105, 156, 208
def, 111, 158
dollargreaterequal, 114
dollargreaterthan, 114
dollarnotequal, 115
elt, 160
eqv, 116
exit, 162
has, 118, 163
if, 123, 163, 212
implies, 123
import, 165
In, 214
in, 124, 213
inby, 125
is, 126, 165
isnt, 126
Join, 127, 166, 215
leave, 128, 169
let, 129, 180
letd, 130
ListCategory, 155
Mapping, 146
mdef, 130, 170
not, 131
notequal, 132
or, 132
pretend, 133, 171, 217
quote, 172, 218
Record, 146
RecordCategory, 155
reduce, 175, 218
repeat, 173, 219
return, 134, 177
Scripts, 220
segment, 135
seq, 178
setq, 180
Signature, 221
String, 183
SubDomain, 184
SubsetCategory, 186
TupleCollect, 223
Union, 146
UnionCategory, 156
vcons, 136
vector, 187
VectorCategory, 156
where, 136, 188, 224
with, 225

defstruct
line, 92
reduction, 101
stack, 97
token, 99

defun
/RQ,LIB, 319
/rf-1, 320
action, 291
add-parens-and-semis-to-line, 87
addclose, 304
Advance-Char, 95
advance-token, 286
aplTran, 226
aplTran1, 226
aplTranList, 228
argsToSig, 361
blankp, 305
bumperrorcount, 297
char-eq, 288
char-ne, 288
checkWarning, 301
comma2Tuple, 207

comp, 337
comp2, 339
comp3, 339
compAdd, 140
compArgumentsAndTryAgain, 352
compAtom, 343
compAtSign, 142
compCapsule, 143
compCapsuleInner, 144
compCase, 145
compCase1, 145
compCat, 147
compCategory, 147
compCoerce, 148
compCoerce1, 149
compColon, 150
compColonInside, 342
compCons, 154
compCons1, 154
compConstruct, 157
compConstructorCategory, 156
compDefine, 158
compDefine1, 158
compElt, 160
compExit, 162
compExpression, 347
compForm, 348
compForm1, 348
compForm2, 350
compHas, 163
compIf, 164
compile-lib-file, 367
compileFileQuietly, 368
compiler, 312
compilerDoit, 318
compileSpad2Cmd, 314
compileSpadLispCmd, 365
compImport, 165
compIs, 165
compJoin, 166
compLambda, 168
compLeave, 169
compList, 347
compMacro, 170
compMakeDeclaration, 362
compNoStacking, 338
compNoStacking1, 338
compOrCroak, 335
compOrCroak1, 336
compOrCroak1,compactify, 364
compPretend, 171
compQuote, 172
compReduce, 175
compReduce1, 175
compRepeatOrCollect, 173
compReturn, 177
compSeq, 179
compSeq1, 179
compSeqItem, 180
compSetq, 180
compSetq1, 181
compString, 184
compSubDomain, 184
compSubDomain1, 185
compSubsetCategory, 186
compSuchthat, 187
compSymbol, 345
compTopLevel, 334
compTypeOf, 342
compVector, 187
compWhere, 188
compWithMappingMode, 353
compWithMappingMode1, 353
containsBang, 229
convert, 344
current-char, 287
current-symbol, 284
current-token, 285
decodeScripts, 230
deepestExpression, 229
def-rename, 333
def-rename1, 333
displayPreCompilationErrors, 296
dollarTran, 289
drop, 305
errhuh, 233
escape-keywords, 281
escaped, 305
extractCodeAndConstructTriple, 360
fincomblock, 306
floatexpid, 289
freelist, 363

get-a-line, 96
get-token, 287
getScriptName, 230
getToken, 282
hackforis, 232
hackforis1, 233
hasAplExtension, 228
hasFormalMapVariable, 361
indent-pos, 306
infixtok, 307
initial-substring, 96
initial-substring-p, 280
initialize-preparse, 73
is-console, 307
isListConstructor, 113
isTokenDelimiter, 281
killColons, 222
line-advance-char, 94
line-at-end-p, 93
line-current-segment, 94
line-new-line, 94
line-next-char, 93
line-past-end-p, 93
line-print, 93
make-string-adjustable, 96
make-symbol-of, 284
makeSimplePredicateOrNil, 298
match-advance-string, 279
match-current-token, 283
match-next-token, 284
match-string, 278
match-token, 284
meta-syntax-error, 289
modifyModeStack, 363
ncINTERPFILE, 365
new2OldLisp, 298
next-char, 287
next-line, 95
next-tab-loc, 307
next-token, 286
nonblankloc, 308
optional, 291
PARSE-AnyId, 268
PARSE-Application, 256
parse-argument-designator, 301
PARSE-Category, 247
PARSE-Command, 242
PARSE-CommandTail, 245
PARSE-Conditional, 275
PARSE-Data, 266
PARSE-ElseClause, 275
PARSE-Enclosure, 262
PARSE-Exit, 273
PARSE-Expr, 250
PARSE-Expression, 249
PARSE-Float, 259
PARSE-FloatBase, 260
PARSE-FloatBasePart, 260
PARSE-FloatExponent, 261
PARSE-FloatTok, 277
PARSE-Form, 255
PARSE-FormalParameter, 263
PARSE-FormalParameterTok, 263
PARSE-getSemanticForm, 252
PARSE-GliphTok, 268
parse-identifier, 299
PARSE-Import, 249
PARSE-Infix, 253
PARSE-InfixWith, 247
PARSE-IntegerTok, 262
PARSE-Iterator, 271
PARSE-IteratorTail, 271
parse-keyword, 300
PARSE-Label, 257
PARSE-LabelExpr, 276
PARSE-Leave, 274
PARSE-LedPart, 250
PARSE-leftBindingPowerOf, 251
PARSE-Loop, 276
PARSE-Name, 265
PARSE-NBGliphTok, 267
PARSE-NewExpr, 241
PARSE-NudPart, 250
parse-number, 300
PARSE-OpenBrace, 270
PARSE-OpenBracket, 270
PARSE-Operation, 251
PARSE-Option, 246
PARSE-Prefix, 252
PARSE-Primary, 258
PARSE-Primary1, 258
PARSE-PrimaryNoFloat, 258

PARSE-PrimaryOrQM, 245
PARSE-Quad, 263
PARSE-Qualification, 254
PARSE-Reduction, 255
PARSE-ReductionOp, 255
PARSE-Return, 273
PARSE-rightBindingPowerOf, 252
PARSE-ScriptItem, 265
PARSE-Scripts, 264
PARSE-Seg, 274
PARSE-Selector, 257
PARSE-SemiColon, 273
PARSE-Sequence, 269
PARSE-Sequence1, 269
PARSE-Sexpr, 266
PARSE-Sexpr1, 266
parse-spadstring, 298
PARSE-SpecialCommand, 243
PARSE-SpecialKeyWord, 242
PARSE-Statement, 246
PARSE-String, 263
parse-string, 299
PARSE-Suffix, 272
PARSE-TokenCommandTail, 243
PARSE-TokenList, 244
PARSE-TokenOption, 244
PARSE-TokTail, 254
PARSE-VarForm, 264
PARSE-With, 247
parseAnd, 107
parseAtom, 104
parseAtSign, 108
parseCategory, 109
parseCoerce, 110
parseColon, 110
parseConstruct, 105
parseDEF, 111
parseDollarGreaterEqual, 114
parseDollarGreaterThan, 114
parseDollarLessEqual, 115
parseDollarNotEqual, 115
parseDropAssertions, 109
parseEquivalence, 116
parseExit, 117
parseGreaterEqual, 117
parseGreaterThan, 118
parseHas, 118
parseHasRhs, 120
parseIf, 123
parseIf,ifTran, 121
parseImplies, 124
parseIn, 124
parseInBy, 125
parseIs, 126
parseIsnt, 127
parseJoin, 127
parseLeave, 128
parseLessEqual, 129
parseLET, 129
parseLETD, 130
parseLhs, 112
parseMDEF, 131
parseNot, 132
parseNotEqual, 132
parseOr, 133
parsepiles, 86
parsePretend, 133
parseprint, 308
parseReturn, 134
parseSegment, 135
parseSeq, 135
parseTran, 103
parseTranCheckForRecord, 297
parseTranList, 105
parseTransform, 103
parseType, 108
parseVCONS, 136
parseWhere, 137
Pop-Reduction, 304
postAdd, 198
postAtom, 193
postAtSign, 200
postBigFloat, 201
postBlock, 202
postBlockItemList, 199
postCapsule, 199
postCategory, 202
postcheck, 195
postCollect, 204
postCollect,finish, 203
postColon, 206
postColonColon, 207

postComma, 207
postConstruct, 208
postDef, 210
postDefArgs, 211
postError, 196
postExit, 212
postFlatten, 208
postFlattenLeft, 221
postForm, 196
postIf, 213
postIn, 214
postin, 213
postInSeq, 214
postIteratorList, 205
postJoin, 215
postMakeCons, 204
postMapping, 216
postMDef, 216
postOp, 193
postPretend, 218
postQUOTE, 218
postReduce, 219
postRepeat, 219
postScripts, 220
postScriptsForm, 194
postSemiColon, 220
postSignature, 221
postSlash, 222
postTran, 192
postTranList, 194
postTranScripts, 194
postTranSegment, 209
postTransform, 191
postTransformCheck, 195
postTuple, 223
postTupleCollect, 224
postType, 200
postWhere, 224
postWith, 225
preparse, 76
preparse-echo, 90
preparse1, 81
preparseReadLine, 88
preparseReadLine1, 89
primitiveType, 345
print-defun, 333
print-package, 301
push-reduction, 292
quote-if-string, 280
read-a-line, 91
recompile-lib-file-if-necessary, 366
s-process, 330
setDefOp, 225
setqSetelt, 181
setqSingle, 182
skip-blanks, 278
skip-ifblock, 88
skip-to-endif, 308
spad, 329
spad-fixed-arg, 367
stack-clear, 97
stack-load, 97
stack-pop, 98
stack-push, 98
storeblanks, 95
token-install, 100
token-lookahead-type, 279
token-print, 101
transIs, 112
transIs1, 112
translabel, 295
translabel1, 295
try-get-token, 285
tuple2List, 302
underscore, 282
unget-tokens, 282
unTuple, 233
defvar
$byConstructors, 368
$comblocklist, 305
$constructorsSeen, 368
$defstack, 231
$echolinestack, 72
$index, 72
$is-eqlist, 232
$is-gensymlist, 232
$is-spill-list, 231
$is-spill, 231
$linelist, 72
$preparse-last-line, 72
$vl, 232
current-fragment, 90

current-line, 92
current-token, 100
definition-name, 241
initial-gensym, 232
lablasoc, 241
meta-error-handler, 288
next-token, 100
nonblank, 99
ParseMode, 241
prior-token, 99
reduce-stack, 292
tmptok, 240
tok, 240
valid-tokens, 100
XTokenReader, 287
deltaContour
calledby compWhere, 189
digitp[5]
called by PARSE-FloatBasePart, 261
called by PARSE-FloatBase, 260
called by floatexpid, 289
displayComp
calledby compOrCroak1, 336
displayPreCompilationErrors, 296
calledby s-process, 330
calls length, 296
calls nequal, 296
calls remdup, 296
calls sayBrightly, 296
calls sayMath, 296
uses $postStack, 296
uses $topOp, 296
defun, 296
displaySemanticErrors
calledby compOrCroak1, 336
calledby s-process, 330
dollargreaterequal, 114
defplist, 114
dollargreaterthan, 114
defplist, 114
dollarnotequal, 115
defplist, 115
dollarTran, 289
calledby PARSE-Qualification, 254
uses $InteractiveMode, 289
defun, 289
doSystemCommand[5]
called by preparse1, 81
drop, 305
calledby add-parens-and-semis-to-line, 87
calledby drop, 305
calls croak, 305
calls drop, 305
calls take, 305
defun, 305

Echo-Meta
usedby preparse-echo, 90
echo-meta
usedby /rf-1, 320
usedby spad, 329
echo-meta[5]
called by /RQ,LIB, 319
editfile
usedby compCapsule, 143
elemn
calledby PARSE-Operation, 251
calledby PARSE-leftBindingPowerOf, 252
calledby PARSE-rightBindingPowerOf, 252
elt, 160
defplist, 160
eqcar
calledby PARSE-OpenBrace, 270
calledby PARSE-OpenBracket, 270
calledby getToken, 282
calledby hackforis1, 233
eqsubstlist
calledby compColon, 151
calledby compTypeOf, 342
eqv, 116
defplist, 116
errhuh, 233
calls systemError, 233
defun, 233
error
calledby compileSpad2Cmd, 316
errorRef
calledby compSymbol, 345

Escape-Character
usedby token-lookahead-type, 279
escape-keywords, 281
calledby quote-if-string, 280
defun, 281
escaped, 305
calledby preparse1, 81
defun, 305
evalAndRwriteLispForm
calledby compSubDomain1, 185
exit, 162
defplist, 162
expand-tabs
calledby preparseReadLine1, 89
extendLocalLibdb
calledby compileSpad2Cmd, 316
extendsCategoryForm
calledby compWithMappingMode1, 355
extractCodeAndConstructTriple, 360
calledby compWithMappingMode1, 356
defun, 360

file-closed
usedby spad, 329
fincomblock, 306
calledby preparse1, 81
calls preparse-echo, 306
uses $EchoLineStack, 306
uses $comblocklist, 306
defun, 306
findfile
calledby compiler, 313
floatexpid, 289
calledby PARSE-FloatExponent, 261
calls collect, 289
calls digitp[5], 289
calls identp[5], 289
calls maxindex, 289
calls pname[5], 289
calls spadreduce, 289
calls step, 289
defun, 289
fnameMake[5]
called by compileSpadLispCmd, 365
fnameReadable?[5]
called by compileSpadLispCmd, 365
formatUnabbreviated
calledby compMacro, 170
fp-output-stream
calledby is-console, 307
freelist, 363
calledby compWithMappingMode1, 356
calledby freelist, 363
calls assq[5], 363
calls freelist, 363
calls getmode, 363
calls identp[5], 363
calls unionq, 363
defun, 363

genSomeVariable
calledby compColon, 151
genvar
calledby hasAplExtension, 228
get
calledby compAtom, 343
calledby compNoStacking1, 338
calledby compSymbol, 345
calledby compTypeOf, 342
calledby compWithMappingMode1, 355
calledby parseHasRhs, 120
calledby setqSingle, 182
get-a-line, 96
calledby initialize-preparse, 73
calledby preparseReadLine1, 89
calls is-console, 96
calls make-string-adjustable, 96
calls mkprompt[5], 96
calls read-a-line, 96
defun, 96
get-internal-run-time
calledby s-process, 330
get-token, 287
calledby try-get-token, 285
calls XTokenReader, 287
uses XTokenReader, 287
defun, 287
getAbbreviation

calledby compDefine1, 159
getdatabase
calledby parseHas, 118
getDeltaEntry
calledby compElt, 161
getDomainsInScope
calledby comp3, 340
calledby compColon, 151
calledby compConstruct, 157
getFormModemaps
calledby compForm1, 348
getIdentity
calledby compReduce1, 175
getl
calledby PARSE-Operation, 251
calledby PARSE-ReductionOp, 255
calledby PARSE-leftBindingPowerOf, 251
calledby PARSE-rightBindingPowerOf, 252
calledby compCat, 147
calledby compExpression, 347
calledby parseTran, 103
getmode
calledby comp3, 340
calledby compCoerce, 148
calledby compColon, 151
calledby compSymbol, 345
calledby freelist, 363
calledby setqSingle, 182
getModemapList
calledby compCase1, 145
getModemapListFromDomain
calledby compElt, 161
getProplist[5]
called by setqSingle, 182
getScriptName, 230
calledby postScriptsForm, 194
calledby postScripts, 220
calls decodeScripts, 230
calls identp[5], 230
calls internl, 230
calls pname[5], 230
calls postError, 230
defun, 230
getSignatureFromMode
calledby compDefine1, 158
getTargetFromRhs
calledby compDefine1, 159
getToken, 282
calledby PARSE-OpenBrace, 270
calledby PARSE-OpenBracket, 270
calls eqcar, 282
defun, 282
giveFormalParametersValues
calledby compDefine1, 159

hackforis, 232
calls hackforis1, 232
defun, 232
hackforis1, 233
calledby hackforis, 232
calls eqcar, 233
calls kar, 233
defun, 233
has, 118, 163
defplist, 118, 163
hasAplExtension, 228
calledby aplTran1, 226
calls aplTran1, 228
calls deepestExpression, 228
calls genvar, 228
calls msubst, 228
calls nreverse0, 228
defun, 228
hasFormalMapVariable, 361
calledby compWithMappingMode1, 355
calls ScanOrPairVec[5], 361
uses $formalMapVariables, 361
defun, 361
helpSpad2Cmd[5]
called by compiler, 313

identp[5]
called by PARSE-FloatExponent, 261
called by compSetq1, 181
called by floatexpid, 289
called by freelist, 363
called by getScriptName, 230
called by postTransform, 191

called by setqSingle, 182
if, 123, 163, 212
defplist, 123, 163, 212
ifcar
calledby preparse, 77
implies, 123
defplist, 123
import, 165
defplist, 165
In, 214
defplist, 214
in, 124, 213
defplist, 124, 213
inby, 125
defplist, 125
incExitLevel
calledby parseIf,ifTran, 121
indent-pos, 306
calledby preparse1, 81
defun, 306
infixtok, 307
calledby add-parens-and-semis-to-line, 87
calls string2id-n, 307
defun, 307
init-boot/spad-reader[5]
called by spad, 329
initial-gensym, 232
defvar, 232
initial-substring, 96
calledby preparseReadLine, 88
calledby skip-ifblock, 88
calledby skip-to-endif, 308
calls mismatch, 96
defun, 96
initial-substring-p, 280
calledby match-string, 278
calls string-not-greaterp, 280
defun, 280
initialize-preparse, 73
calledby spad, 329
calls get-a-line, 73
uses $echolinestack, 73
uses $index, 73
uses $linelist, 73
uses $preparse-last-line, 73
defun, 73
insert
calledby comp2, 339
internl
calledby getScriptName, 230
calledby postForm, 196
intersectionEnvironment
calledby compIf, 164
ioclear
calledby spad, 329
is, 126, 165
defplist, 126, 165
is-console, 307
calledby get-a-line, 96
calledby preparse1, 81
calledby print-defun, 333
calls fp-output-stream, 307
uses *terminal-io*, 307
defun, 307
isAlmostSimple
calledby makeSimplePredicateOrNil, 298
isCategoryForm
calledby compColon, 151
calledby compJoin, 166
isDomainForm
calledby comp2, 339
calledby compColon, 151
calledby compDefine1, 158
calledby compElt, 160
calledby setqSingle, 182
isDomainInScope
calledby setqSingle, 182
isFunction
calledby compSymbol, 345
isFunctor
calledby comp2, 339
calledby compWithMappingMode1, 355
isListConstructor, 113
calledby transIs, 112
calls member, 113
defun, 113
isMacro
calledby compDefine1, 158
isnt, 126

defplist, 126
isSimple
calledby compForm2, 350
calledby makeSimplePredicateOrNil, 298
isSymbol
calledby compAtom, 343
isTokenDelimiter, 281
calledby PARSE-TokenList, 244
calls current-symbol, 281
defun, 281

Join, 127, 166, 215
defplist, 127, 166, 215

kar
calledby hackforis1, 233
killColons, 222
calledby killColons, 222
calledby postSignature, 221
calls killColons, 222
defun, 222

labasoc
usedby PARSE-Data, 266
lablasoc, 241
defvar, 241
lassoc
calledby compOrCroak1,compactify, 364
calledby translabel1, 295
last
calledby parseSeq, 135
leave, 128, 169
defplist, 128, 169
length
calledby compColon, 151
calledby compDefine1, 159
calledby compElt, 161
calledby compForm1, 348
calledby compForm2, 350
calledby compRepeatOrCollect, 173
calledby displayPreCompilationErrors, 296
calledby postScriptsForm, 194
let, 129, 180
defplist, 129, 180
letd, 130
defplist, 130
line, 92
usedby match-string, 278
usedby spad, 329
defstruct, 92
Line-Advance-Char
calledby Advance-Char, 95
line-advance-char, 94
uses $line, 94
defun, 94
Line-At-End-P
calledby Advance-Char, 95
line-at-end-p, 93
calledby next-char, 288
uses $line, 93
defun, 93
line-clear, 92
uses $line, 92
defmacro, 92
line-current-char
calledby match-advance-string, 279
line-current-index
calledby match-advance-string, 279
line-current-segment, 94
calledby unget-tokens, 283
defun, 94
Line-New-Line
calledby read-a-line, 91
line-new-line, 94
calledby unget-tokens, 283
uses $line, 94
defun, 94
line-next-char, 93
calledby next-char, 288
uses $line, 93
defun, 93
line-number
calledby PARSE-Category, 248
calledby unget-tokens, 283
line-past-end-p, 93
calledby match-advance-string, 279
calledby match-string, 278
uses $line, 93
defun, 93

line-print, 93
  uses $line, 93, 94
  defun, 93
lispize
  calledby compSubDomain1, 185
ListCategory, 155
  defplist, 155
loadIfNecessary
  calledby parseHasRhs, 120
localdatabase[5]
  called by compileSpadLispCmd, 365
lt
  calledby PARSE-Operation, 251

macroExpand
  calledby compDefine1, 158
  calledby compMacro, 170
  calledby compSeqItem, 180
  calledby compWhere, 188
make-float
  calledby PARSE-Float, 259
make-full-cvec
  calledby preparse1, 81
make-reduction
  calledby push-reduction, 292
make-string-adjustable, 96
  calledby get-a-line, 96
  defun, 96
make-symbol-of, 284
  calledby PARSE-Expression, 249
  calledby current-symbol, 284
  uses $token, 284
  defun, 284
makeCategoryForm
  calledby compColon, 151
makeInitialModemapFrame[5]
  called by spad, 329
makeInputFilename[5]
  called by /rf-1, 320
makeNonAtomic
  calledby parseHas, 119
makeSimplePredicateOrNil, 298
  calledby parseIf,ifTran, 121
  calls isAlmostSimple, 298
  calls isSimple, 298
  calls wrapSEQExit, 298
  defun, 298
mapInto
  calledby parseSeq, 135
  calledby parseWhere, 137
Mapping, 146
  defplist, 146
match-advance-string, 279
  calledby PARSE-Category, 247
  calledby PARSE-Command, 242
  calledby PARSE-Conditional, 275
  calledby PARSE-Enclosure, 262
  calledby PARSE-Exit, 273
  calledby PARSE-FloatBasePart, 260
  calledby PARSE-FloatExponent, 261
  calledby PARSE-Form, 255
  calledby PARSE-Import, 249
  calledby PARSE-IteratorTail, 271
  calledby PARSE-Iterator, 271
  calledby PARSE-Label, 257
  calledby PARSE-Leave, 274
  calledby PARSE-Loop, 276
  calledby PARSE-Option, 246
  calledby PARSE-Primary1, 259
  calledby PARSE-PrimaryOrQM, 245
  calledby PARSE-Quad, 263
  calledby PARSE-Qualification, 254
  calledby PARSE-Return, 273
  calledby PARSE-ScriptItem, 265
  calledby PARSE-Scripts, 264
  calledby PARSE-Selector, 257
  calledby PARSE-SemiColon, 273
  calledby PARSE-Sequence, 269
  calledby PARSE-Sexpr1, 266
  calledby PARSE-SpecialCommand, 243
  calledby PARSE-Statement, 246
  calledby PARSE-TokenOption, 244
  calledby PARSE-With, 247
  calls current-token, 279
  calls line-current-char, 279
  calls line-current-index, 279
  calls line-past-end-p, 279
  calls match-string, 279
  calls quote-if-string, 279
  uses $line, 279
  uses $token, 279

defun, 279
match-current-token, 283
calledby PARSE-GliphTok, 268
calledby PARSE-NBGliphTok, 267
calledby PARSE-Operation, 251
calledby PARSE-SpecialKeyWord, 242
calledby parse-argument-designator, 301
calledby parse-identifier, 299
calledby parse-keyword, 300
calledby parse-number, 300
calledby parse-spadstring, 298
calledby parse-string, 299
calls current-token, 283
calls match-token, 283
defun, 283
match-next-token, 284
calledby PARSE-ReductionOp, 255
calls match-token, 284
calls next-token, 284
defun, 284
match-string, 278
calledby PARSE-AnyId, 268
calledby PARSE-NewExpr, 241
calledby PARSE-Primary1, 259
calledby match-advance-string, 279
calls current-char, 278
calls initial-substring-p, 278
calls line-past-end-p, 278
calls skip-blanks, 278
calls subseq, 278
calls unget-tokens, 278
uses $line, 278
uses line, 278
defun, 278
match-token, 284
calledby match-current-token, 283
calledby match-next-token, 284
calls token-symbol, 284
calls token-type, 284
defun, 284
maxindex
calledby floatexpid, 289
calledby preparse1, 81
calledby preparseReadLine1, 89
calledby translabel1, 295
maxSuperType
calledby setqSingle, 182
mdef, 130, 170
defplist, 130, 170
member
calledby isListConstructor, 113
calledby parseHasRhs, 120
calledby parseHas, 119
member[5]
called by comp3, 340
called by compColon, 151
called by compSymbol, 345
called by compilerDoit, 318
mergePathnames[5]
called by compiler, 313
meta-error-handler, 288
calledby meta-syntax-error, 289
usedby meta-syntax-error, 289
defvar, 288
meta-syntax-error, 289
calledby must, 290
calls meta-error-handler, 289
uses meta-error-handler, 289
defun, 289
mismatch
calledby initial-substring, 96
mkErrorExpr
calledby compOrCroak1, 336
mkExplicitCategoryFunction
calledby compCategory, 147
mkpf
calledby compCapsuleInner, 144
mkprompt[5]
called by get-a-line, 96
mkq
calledby compCoerce1, 149
calledby compSeq1, 179
moan
calledby parseExit, 117
calledby parseReturn, 134
modeEqual
calledby compCase1, 145
modeIsAggregateOf
calledby compAtom, 343
calledby compConstruct, 157

calledby compRepeatOrCollect, 173
modifyModeStack, 363
calledby compExit, 162
calledby compLeave, 169
calledby compReturn, 178
calls copy, 363
calls resolve, 363
calls say, 363
calls setelt, 363
uses $exitModeStack, 363
uses $reportExitModeStack, 363
defun, 363
msubst
calledby compCoerce1, 149
calledby compReduce1, 175
calledby compRepeatOrCollect, 173
calledby compSubsetCategory, 186
calledby hasAplExtension, 228
calledby parseDollarGreaterEqual, 114
calledby parseDollarGreaterThan, 114
calledby parseDollarLessEqual, 115
calledby parseDollarNotEqual, 116
calledby parseNotEqual, 132
calledby parseTransform, 103
calledby parseType, 108
must, 290
calledby PARSE-Category, 247
calledby PARSE-Command, 242
calledby PARSE-Conditional, 275
calledby PARSE-Enclosure, 262
calledby PARSE-Exit, 273
calledby PARSE-FloatBasePart, 261
calledby PARSE-FloatBase, 260
calledby PARSE-FloatExponent, 261
calledby PARSE-Float, 259
calledby PARSE-Form, 255
calledby PARSE-Import, 249
calledby PARSE-Infix, 253
calledby PARSE-Iterator, 271
calledby PARSE-LabelExpr, 276
calledby PARSE-Label, 257
calledby PARSE-Leave, 274
calledby PARSE-Loop, 276
calledby PARSE-NewExpr, 241
calledby PARSE-Option, 246
calledby PARSE-Prefix, 253
calledby PARSE-Primary1, 258
calledby PARSE-Qualification, 254
calledby PARSE-Reduction, 255
calledby PARSE-Return, 273
calledby PARSE-ScriptItem, 265
calledby PARSE-Scripts, 264
calledby PARSE-Selector, 257
calledby PARSE-SemiColon, 273
calledby PARSE-Sequence, 269
calledby PARSE-Sexpr1, 266
calledby PARSE-SpecialCommand, 243
calledby PARSE-Statement, 246
calledby PARSE-TokenOption, 244
calledby PARSE-With, 247
calls meta-syntax-error, 290
defmacro, 290

namestring[5]
called by compileSpad2Cmd, 316
called by compileSpadLispCmd, 365
called by compiler, 313
ncINTERPFILE, 365
calledby /rf-1, 320
calls SpadInterpretStream[5], 365
uses $EchoLines, 365
uses $ReadingFile, 365
defun, 365
nequal
calledby comp2, 339
calledby compElt, 161
calledby compPretend, 171
calledby compReturn, 177
calledby compileSpad2Cmd, 316
calledby displayPreCompilationErrors, 296
calledby postDef, 210
calledby postError, 196
calledby setqSingle, 182
new2OldLisp, 298
calledby s-process, 330
calls new2OldTran, 298
calls postTransform, 298
defun, 298

new2OldTran
calledby new2OldLisp, 298
newComp
calledby compTopLevel, 334
next-char, 287
calledby PARSE-FloatBase, 260
calls line-at-end-p, 288
calls line-next-char, 288
uses current-line, 288
defun, 287
next-line, 95
calledby Advance-Char, 95
defun, 95
next-tab-loc, 307
defun, 307
next-token, 100, 286
calledby match-next-token, 284
calls current-token, 286
calls try-get-token, 286
usedby next-token, 286
uses $token, 100
uses next-token, 286
uses valid-tokens, 286
defun, 286
defvar, 100
nonblank, 99
defvar, 99
nonblankloc, 308
calledby add-parens-and-semis-to-line, 87
calls blankp, 308
defun, 308
not, 131
defplist, 131
notequal, 132
defplist, 132
nreverse0
calledby aplTran1, 226
calledby compAdd, 140
calledby compCase1, 145
calledby compColon, 151
calledby compForm1, 348
calledby compForm2, 350
calledby compJoin, 166
calledby compReduce1, 175
calledby compSeq1, 179
calledby hasAplExtension, 228
calledby parseHas, 119
calledby postCategory, 202
calledby postDef, 210
calledby postIf, 213
calledby postMDef, 216
calledby transIs1, 112
NRTassocIndex
calledby setqSingle, 182
NRTgetLocalIndex
calledby compAdd, 141
calledby compSymbol, 345
nth-stack, 304
calledby PARSE-Category, 248
calledby PARSE-Sexpr1, 266
calls reduction-value, 304
calls stack-store, 304
defmacro, 304

object2String
calledby compileSpad2Cmd, 316
calledby compileSpadLispCmd, 365
opFf
calledby parseDEF, 111
opOf
calledby comp2, 339
calledby compColonInside, 342
calledby compElt, 161
calledby compPretend, 171
calledby compilerDoit, 318
calledby parseHas, 118
calledby parseLET, 129
calledby parseMDEF, 131
optimizeFunctionDef
calledby compWithMappingMode1, 356
optional, 291
calledby PARSE-Application, 256
calledby PARSE-Category, 247
calledby PARSE-CommandTail, 245
calledby PARSE-Conditional, 275
calledby PARSE-Expr, 250
calledby PARSE-Form, 255
calledby PARSE-Import, 249
calledby PARSE-Infix, 253
calledby PARSE-IteratorTail, 271

calledby PARSE-Iterator, 271
calledby PARSE-Prefix, 253
calledby PARSE-Primary1, 258
calledby PARSE-PrimaryNoFloat, 258
calledby PARSE-ScriptItem, 265
calledby PARSE-Seg, 274
calledby PARSE-Sequence1, 269
calledby PARSE-Sexpr1, 266
calledby PARSE-SpecialCommand, 243
calledby PARSE-Statement, 246
calledby PARSE-Suffix, 272
calledby PARSE-TokenCommandTail, 244
calledby PARSE-VarForm, 264
defun, 291
or, 132
defplist, 132
outputComp
calledby compForm1, 348
calledby setqSingle, 182

pack
calledby quote-if-string, 280
pairp
calledby compAdd, 140
calledby compCons1, 154
calledby compJoin, 166
calledby postSignature, 221
calledby postTran, 192
calledby transIs1, 112
PARSE-AnyId, 268
calledby PARSE-Sexpr1, 266
calls action, 268
calls advance-token, 268
calls current-symbol, 268
calls match-string, 268
calls parse-identifier, 268
calls parse-keyword, 268
calls push-reduction, 268
defun, 268
PARSE-Application, 256
calledby PARSE-Application, 256
calledby PARSE-Category, 248
calledby PARSE-Form, 256
calls PARSE-Application, 256
calls PARSE-Primary, 256
calls PARSE-Selector, 256
calls optional, 256
calls pop-stack-1, 256
calls pop-stack-2, 256
calls push-reduction, 256
calls star, 256
defun, 256
parse-argument-designator, 301
calledby PARSE-FormalParameterTok, 263
calls advance-token, 301
calls match-current-token, 301
calls push-reduction, 301
calls token-symbol, 301
defun, 301
PARSE-Category, 247
calledby PARSE-Category, 247
calls PARSE-Application, 248
calls PARSE-Category, 247
calls PARSE-Expression, 247
calls action, 248
calls bang, 247
calls line-number, 248
calls match-advance-string, 247
calls must, 247
calls nth-stack, 248
calls optional, 247
calls pop-stack-1, 248
calls pop-stack-2, 247
calls pop-stack-3, 247
calls push-reduction, 247
calls recordAttributeDocumentation, 248
calls recordSignatureDocumentation, 248
calls star, 248
uses current-line, 248
defun, 247
PARSE-Command, 242
calls PARSE-SpecialCommand, 242
calls PARSE-SpecialKeyWord, 242
calls match-advance-string, 242
calls must, 242
calls push-reduction, 242

defun, 242
PARSE-CommandTail, 245
calledby PARSE-CommandTail, 245
calledby PARSE-SpecialCommand, 243
calls PARSE-CommandTail, 245
calls PARSE-Option, 245
calls action, 245
calls bang, 245
calls optional, 245
calls pop-stack-1, 245
calls pop-stack-2, 245
calls push-reduction, 245
calls star, 245
calls systemCommand[5], 245
defun, 245
PARSE-Conditional, 275
calledby PARSE-ElseClause, 275
calls PARSE-ElseClause, 275
calls PARSE-Expression, 275
calls bang, 275
calls match-advance-string, 275
calls must, 275
calls optional, 275
calls pop-stack-1, 275
calls pop-stack-2, 275
calls pop-stack-3, 275
calls push-reduction, 275
defun, 275
PARSE-Data, 266
calledby PARSE-Primary1, 259
calls PARSE-Sexpr, 266
calls action, 266
calls pop-stack-1, 266
calls push-reduction, 266
calls translabel, 266
uses labasoc, 266
defun, 266
PARSE-ElseClause, 275
calledby PARSE-Conditional, 275
calls PARSE-Conditional, 275
calls PARSE-Expression, 276
calls current-symbol, 275
defun, 275
PARSE-Enclosure, 262
calledby PARSE-Primary1, 259
calls PARSE-Expr, 262
calls match-advance-string, 262
calls must, 262
calls pop-stack-1, 262
calls push-reduction, 262
defun, 262
PARSE-Exit, 273
calls PARSE-Expression, 273
calls match-advance-string, 273
calls must, 273
calls pop-stack-1, 274
calls push-reduction, 273
defun, 273
PARSE-Expr, 250
calledby PARSE-Enclosure, 262
calledby PARSE-Expression, 249
calledby PARSE-Import, 249
calledby PARSE-Iterator, 271
calledby PARSE-LabelExpr, 276
calledby PARSE-Loop, 276
calledby PARSE-Primary1, 259
calledby PARSE-Reduction, 255
calledby PARSE-ScriptItem, 265
calledby PARSE-SemiColon, 273
calledby PARSE-Statement, 246
calls PARSE-LedPart, 250
calls PARSE-NudPart, 250
calls optional, 250
calls pop-stack-1, 250
calls push-reduction, 250
calls star, 250
defun, 250
PARSE-Expression, 249
calledby PARSE-Category, 247
calledby PARSE-Conditional, 275
calledby PARSE-ElseClause, 276
calledby PARSE-Exit, 273
calledby PARSE-Infix, 253
calledby PARSE-Iterator, 271
calledby PARSE-Leave, 274
calledby PARSE-Prefix, 253
calledby PARSE-Return, 273
calledby PARSE-Seg, 274
calledby PARSE-Sequence1, 269
calledby PARSE-SpecialCommand, 243

calls PARSE-Expr, 249
calls PARSE-rightBindingPowerOf, 249
calls make-symbol-of, 249
calls pop-stack-1, 249
calls push-reduction, 249
uses ParseMode, 249
uses prior-token, 249
defun, 249
PARSE-Float, 259
calledby PARSE-Primary, 258
calledby PARSE-Selector, 257
calls PARSE-FloatBase, 259
calls PARSE-FloatExponent, 259
calls make-float, 259
calls must, 259
calls pop-stack-1, 259
calls pop-stack-2, 259
calls pop-stack-3, 259
calls pop-stack-4, 259
calls push-reduction, 259
defun, 259
PARSE-FloatBase, 260
calledby PARSE-Float, 259
calls PARSE-FloatBasePart, 260
calls PARSE-IntegerTok, 260
calls char-eq, 260
calls char-ne, 260
calls current-char, 260
calls current-symbol, 260
calls digitp[5], 260
calls must, 260
calls next-char, 260
calls push-reduction, 260
defun, 260
PARSE-FloatBasePart, 260
calledby PARSE-FloatBase, 260
calls PARSE-IntegerTok, 261
calls current-char, 261
calls current-token, 261
calls digitp[5], 261
calls match-advance-string, 260
calls must, 261
calls push-reduction, 261
calls token-nonblank, 261
defun, 260
PARSE-FloatExponent, 261
calledby PARSE-Float, 259
calls PARSE-IntegerTok, 261
calls action, 261
calls advance-token, 261
calls current-char, 261
calls current-symbol, 261
calls floatexpid, 261
calls identp[5], 261
calls match-advance-string, 261
calls must, 261
calls push-reduction, 261
defun, 261
PARSE-FloatTok, 277
calls bfp-, 277
calls parse-number, 277
calls pop-stack-1, 277
calls push-reduction, 277
uses $boot, 277
defun, 277
PARSE-Form, 255
calledby PARSE-NudPart, 250
calls PARSE-Application, 256
calls bang, 255
calls match-advance-string, 255
calls must, 255
calls optional, 255
calls pop-stack-1, 256
calls push-reduction, 255
defun, 255
PARSE-FormalParameter, 263
calledby PARSE-Primary1, 259
calls PARSE-FormalParameterTok, 263
defun, 263
PARSE-FormalParameterTok, 263
calledby PARSE-FormalParameter, 263
calls parse-argument-designator, 263
defun, 263
PARSE-getSemanticForm, 252
calledby PARSE-Operation, 251
calls PARSE-Infix, 252
calls PARSE-Prefix, 252
defun, 252
PARSE-GliphTok, 268

calledby PARSE-Quad, 263
calledby PARSE-Seg, 274
calledby PARSE-Sexpr1, 267
calls action, 268
calls advance-token, 268
calls match-current-token, 268
uses tok, 268
defun, 268
parse-identifier, 299
calledby PARSE-AnyId, 268
calledby PARSE-Name, 265
calls advance-token, 299
calls match-current-token, 299
calls push-reduction, 299
calls token-symbol, 299
defun, 299
PARSE-Import, 249
calls PARSE-Expr, 249
calls bang, 249
calls match-advance-string, 249
calls must, 249
calls optional, 249
calls pop-stack-1, 249
calls pop-stack-2, 249
calls push-reduction, 249
calls star, 249
defun, 249
PARSE-Infix, 253
calledby PARSE-getSemanticForm, 252
calls PARSE-Expression, 253
calls PARSE-TokTail, 253
calls action, 253
calls advance-token, 253
calls current-symbol, 253
calls must, 253
calls optional, 253
calls pop-stack-1, 253
calls pop-stack-2, 253
calls push-reduction, 253
defun, 253
PARSE-InfixWith, 247
calls PARSE-With, 247
calls pop-stack-1, 247
calls pop-stack-2, 247
calls push-reduction, 247
defun, 247
PARSE-IntegerTok, 262
calledby PARSE-FloatBasePart, 261
calledby PARSE-FloatBase, 260
calledby PARSE-FloatExponent, 261
calledby PARSE-Primary1, 259
calledby PARSE-Sexpr1, 266
calls parse-number, 262
defun, 262
PARSE-Iterator, 271
calledby PARSE-IteratorTail, 271
calledby PARSE-Loop, 276
calls PARSE-Expression, 271
calls PARSE-Expr, 271
calls PARSE-Primary, 271
calls match-advance-string, 271
calls must, 271
calls optional, 271
calls pop-stack-1, 271
calls pop-stack-2, 271
calls pop-stack-3, 271
defun, 271
PARSE-IteratorTail, 271
calledby PARSE-Sequence1, 269
calls PARSE-Iterator, 271
calls bang, 271
calls match-advance-string, 271
calls optional, 271
calls star, 271
defun, 271
parse-keyword, 300
calledby PARSE-AnyId, 268
calls advance-token, 300
calls match-current-token, 300
calls push-reduction, 300
calls token-symbol, 300
defun, 300
PARSE-Label, 257
calledby PARSE-LabelExpr, 276
calledby PARSE-Leave, 274
calls PARSE-Name, 257
calls match-advance-string, 257
calls must, 257
defun, 257
PARSE-LabelExpr, 276
calls PARSE-Expr, 276

calls PARSE-Label, 276
calls must, 276
calls pop-stack-1, 276
calls pop-stack-2, 276
calls push-reduction, 276
defun, 276
PARSE-Leave, 274
calls PARSE-Expression, 274
calls PARSE-Label, 274
calls match-advance-string, 274
calls must, 274
calls pop-stack-1, 274
calls push-reduction, 274
defun, 274
PARSE-LedPart, 250
calledby PARSE-Expr, 250
calls PARSE-Operation, 250
calls pop-stack-1, 250
calls push-reduction, 250
defun, 250
PARSE-leftBindingPowerOf, 251
calledby PARSE-Operation, 251
calls elemn, 252
calls getl, 251
defun, 251
PARSE-Loop, 276
calls PARSE-Expr, 276
calls PARSE-Iterator, 276
calls match-advance-string, 276
calls must, 276
calls pop-stack-1, 276
calls pop-stack-2, 276
calls push-reduction, 276
calls star, 276
defun, 276
PARSE-Name, 265
calledby PARSE-Label, 257
calledby PARSE-VarForm, 264
calls parse-identifier, 265
calls pop-stack-1, 265
calls push-reduction, 265
defun, 265
PARSE-NBGliphTok, 267
calledby PARSE-Sexpr1, 266
calls action, 267
calls advance-token, 267
calls match-current-token, 267
uses tok, 267
defun, 267
PARSE-NewExpr, 241
calledby spad, 329
calls PARSE-Statement, 241
calls action, 241
calls current-symbol, 241
calls match-string, 241
calls must, 241
calls processSynonyms[5], 241
uses definition-name, 241
defun, 241
PARSE-NudPart, 250
calledby PARSE-Expr, 250
calls PARSE-Form, 250
calls PARSE-Operation, 250
calls PARSE-Reduction, 250
calls pop-stack-1, 251
calls push-reduction, 250
uses rbp, 251
defun, 250
parse-number, 300
calledby PARSE-FloatTok, 277
calledby PARSE-IntegerTok, 262
calls advance-token, 300
calls match-current-token, 300
calls push-reduction, 300
calls token-symbol, 300
defun, 300
PARSE-OpenBrace, 270
calledby PARSE-Sequence, 269
calls action, 270
calls advance-token, 270
calls current-symbol, 270
calls eqcar, 270
calls getToken, 270
calls push-reduction, 270
defun, 270
PARSE-OpenBracket, 270
calledby PARSE-Sequence, 269
calls action, 270
calls advance-token, 270
calls current-symbol, 270
calls eqcar, 270
calls getToken, 270

calls push-reduction, 270
defun, 270
PARSE-Operation, 251
calledby PARSE-LedPart, 250
calledby PARSE-NudPart, 250
calls PARSE-getSemanticForm, 251
calls PARSE-leftBindingPowerOf, 251
calls PARSE-rightBindingPowerOf, 251
calls action, 251
calls current-symbol, 251
calls elemn, 251
calls getl, 251
calls lt, 251
calls match-current-token, 251
uses ParseMode, 251
uses rbp, 251
uses tmptok, 251
defun, 251
PARSE-Option, 246
calledby PARSE-CommandTail, 245
calls PARSE-PrimaryOrQM, 246
calls match-advance-string, 246
calls must, 246
calls star, 246
defun, 246
PARSE-Prefix, 252
calledby PARSE-getSemanticForm, 252
calls PARSE-Expression, 253
calls PARSE-TokTail, 253
calls action, 252
calls advance-token, 253
calls current-symbol, 252
calls must, 253
calls optional, 253
calls pop-stack-1, 253
calls pop-stack-2, 253
calls push-reduction, 252, 253
defun, 252
PARSE-Primary, 258
calledby PARSE-Application, 256
calledby PARSE-Iterator, 271
calledby PARSE-PrimaryOrQM, 245
calledby PARSE-Selector, 257
calls PARSE-Float, 258
calls PARSE-PrimaryNoFloat, 258
defun, 258
PARSE-Primary1, 258
calledby PARSE-Primary1, 258
calledby PARSE-PrimaryNoFloat, 258
calledby PARSE-Qualification, 254
calls PARSE-Data, 259
calls PARSE-Enclosure, 259
calls PARSE-Expr, 259
calls PARSE-FormalParameter, 259
calls PARSE-IntegerTok, 259
calls PARSE-Primary1, 258
calls PARSE-Quad, 259
calls PARSE-Sequence, 259
calls PARSE-String, 259
calls PARSE-VarForm, 258
calls current-symbol, 258
calls match-advance-string, 259
calls match-string, 259
calls must, 258
calls optional, 258
calls pop-stack-1, 258
calls pop-stack-2, 258
calls push-reduction, 258
uses $boot, 259
defun, 258
PARSE-PrimaryNoFloat, 258
calledby PARSE-Primary, 258
calledby PARSE-Selector, 257
calls PARSE-Primary1, 258
calls PARSE-TokTail, 258
calls optional, 258
defun, 258
PARSE-PrimaryOrQM, 245
calledby PARSE-Option, 246
calledby PARSE-PrimaryOrQM, 245
calledby PARSE-SpecialCommand, 243
calls PARSE-PrimaryOrQM, 245
calls PARSE-Primary, 245
calls match-advance-string, 245
calls push-reduction, 245
defun, 245
PARSE-Quad, 263

calledby PARSE-Primary1, 259
calls PARSE-GliphTok, 263
calls match-advance-string, 263
calls push-reduction, 263
uses $boot, 263
defun, 263
PARSE-Qualification, 254
calledby PARSE-TokTail, 254
calls PARSE-Primary1, 254
calls dollarTran, 254
calls match-advance-string, 254
calls must, 254
calls pop-stack-1, 254
calls push-reduction, 254
defun, 254
PARSE-Reduction, 255
calledby PARSE-NudPart, 250
calls PARSE-Expr, 255
calls PARSE-ReductionOp, 255
calls must, 255
calls pop-stack-1, 255
calls pop-stack-2, 255
calls push-reduction, 255
defun, 255
PARSE-ReductionOp, 255
calledby PARSE-Reduction, 255
calls action, 255
calls advance-token, 255
calls current-symbol, 255
calls getl, 255
calls match-next-token, 255
defun, 255
PARSE-Return, 273
calls PARSE-Expression, 273
calls match-advance-string, 273
calls must, 273
calls pop-stack-1, 273
calls push-reduction, 273
defun, 273
PARSE-rightBindingPowerOf, 252
calledby PARSE-Expression, 249
calledby PARSE-Operation, 251
calls elemn, 252
calls getl, 252
defun, 252
PARSE-ScriptItem, 265
calledby PARSE-ScriptItem, 265
calledby PARSE-Scripts, 264
calls PARSE-Expr, 265
calls PARSE-ScriptItem, 265
calls match-advance-string, 265
calls must, 265
calls optional, 265
calls pop-stack-1, 265
calls pop-stack-2, 265
calls push-reduction, 265
calls star, 265
defun, 265
PARSE-Scripts, 264
calledby PARSE-VarForm, 264
calls PARSE-ScriptItem, 264
calls match-advance-string, 264
calls must, 264
defun, 264
PARSE-Seg, 274
calls PARSE-Expression, 274
calls PARSE-GliphTok, 274
calls bang, 274
calls optional, 274
calls pop-stack-1, 275
calls pop-stack-2, 275
calls push-reduction, 274
defun, 274
PARSE-Selector, 257
calledby PARSE-Application, 256
calls PARSE-Float, 257
calls PARSE-PrimaryNoFloat, 257
calls PARSE-Primary, 257
calls char-ne, 257
calls current-char, 257
calls current-symbol, 257
calls match-advance-string, 257
calls must, 257
calls pop-stack-1, 257
calls pop-stack-2, 257
calls push-reduction, 257
uses $boot, 257
defun, 257
PARSE-SemiColon, 273
calls PARSE-Expr, 273
calls match-advance-string, 273
calls must, 273

calls pop-stack-1, 273
calls pop-stack-2, 273
calls push-reduction, 273
defun, 273
PARSE-Sequence, 269
calledby PARSE-Primary1, 259
calls PARSE-OpenBrace, 269
calls PARSE-OpenBracket, 269
calls PARSE-Sequence1, 269
calls match-advance-string, 269
calls must, 269
calls pop-stack-1, 269
calls push-reduction, 269
defun, 269
PARSE-Sequence1, 269
calledby PARSE-Sequence, 269
calls PARSE-Expression, 269
calls PARSE-IteratorTail, 269
calls optional, 269
calls pop-stack-1, 269
calls pop-stack-2, 269
calls push-reduction, 269
defun, 269
PARSE-Sexpr, 266
calledby PARSE-Data, 266
calls PARSE-Sexpr1, 266
defun, 266
PARSE-Sexpr1, 266
calledby PARSE-Sexpr1, 266
calledby PARSE-Sexpr, 266
calls PARSE-AnyId, 266
calls PARSE-GliphTok, 267
calls PARSE-IntegerTok, 266
calls PARSE-NBGliphTok, 266
calls PARSE-Sexpr1, 266
calls PARSE-String, 267
calls action, 266
calls bang, 267
calls match-advance-string, 266
calls must, 266
calls nth-stack, 266
calls optional, 266
calls pop-stack-1, 267
calls pop-stack-2, 266
calls push-reduction, 266
calls star, 267
defun, 266
parse-spadstring, 298
calledby PARSE-String, 263
calls advance-token, 298
calls match-current-token, 298
calls push-reduction, 298
calls token-symbol, 298
defun, 298
PARSE-SpecialCommand, 243
calledby PARSE-Command, 242
calledby PARSE-SpecialCommand, 243
calls PARSE-CommandTail, 243
calls PARSE-Expression, 243
calls PARSE-PrimaryOrQM, 243
calls PARSE-SpecialCommand, 243
calls PARSE-TokenCommandTail, 243
calls PARSE-TokenList, 243
calls action, 243
calls bang, 243
calls current-symbol, 243
calls match-advance-string, 243
calls must, 243
calls optional, 243
calls pop-stack-1, 243
calls push-reduction, 243
calls star, 243
uses $noParseCommands, 243
uses $tokenCommands, 243
defun, 243
PARSE-SpecialKeyWord, 242
calledby PARSE-Command, 242
calls action, 242
calls current-symbol, 242
calls current-token, 242
calls match-current-token, 242
calls token-symbol, 242
calls unAbbreviateKeyword[5], 242
defun, 242
PARSE-Statement, 246
calledby PARSE-NewExpr, 241
calls PARSE-Expr, 246
calls match-advance-string, 246
calls must, 246
calls optional, 246

calls pop-stack-1, 246
calls pop-stack-2, 246
calls push-reduction, 246
calls star, 246
defun, 246
PARSE-String, 263
calledby PARSE-Primary1, 259
calledby PARSE-Sexpr1, 267
calls parse-spadstring, 263
defun, 263
parse-string, 299
calls advance-token, 299
calls match-current-token, 299
calls push-reduction, 299
calls token-symbol, 299
defun, 299
PARSE-Suffix, 272
calls PARSE-TokTail, 272
calls action, 272
calls advance-token, 272
calls current-symbol, 272
calls optional, 272
calls pop-stack-1, 272
calls push-reduction, 272
defun, 272
PARSE-TokenCommandTail, 243
calledby PARSE-SpecialCommand, 243
calledby PARSE-TokenCommandTail, 244
calls PARSE-TokenCommandTail, 244
calls PARSE-TokenOption, 244
calls action, 244
calls atEndOfLine, 244
calls bang, 243
calls optional, 244
calls pop-stack-1, 244
calls pop-stack-2, 244
calls push-reduction, 244
calls star, 244
calls systemCommand[5], 244
defun, 243
PARSE-TokenList, 244
calledby PARSE-SpecialCommand, 243
calledby PARSE-TokenOption, 244
calls action, 244
calls advance-token, 244
calls current-symbol, 244
calls isTokenDelimiter, 244
calls push-reduction, 244
calls star, 244
defun, 244
PARSE-TokenOption, 244
calledby PARSE-TokenCommandTail, 244
calls PARSE-TokenList, 244
calls match-advance-string, 244
calls must, 244
defun, 244
PARSE-TokTail, 254
calledby PARSE-Infix, 253
calledby PARSE-Prefix, 253
calledby PARSE-PrimaryNoFloat, 258
calledby PARSE-Suffix, 272
calls PARSE-Qualification, 254
calls action, 254
calls char-eq, 254
calls copy-token, 254
calls current-char, 254
calls current-symbol, 254
uses $boot, 254
defun, 254
PARSE-VarForm, 264
calledby PARSE-Primary1, 258
calls PARSE-Name, 264
calls PARSE-Scripts, 264
calls optional, 264
calls pop-stack-1, 264
calls pop-stack-2, 264
calls push-reduction, 264
defun, 264
PARSE-With, 247
calledby PARSE-InfixWith, 247
calls match-advance-string, 247
calls must, 247
calls pop-stack-1, 247
calls push-reduction, 247
defun, 247
parseAnd, 107

calledby parseAnd, 107
calls parseAnd, 107
calls parseIf, 107
calls parseTranList, 107
calls parseTran, 107
uses $InteractiveMode, 107
defun, 107
parseAtom, 104
calledby parseTran, 103
calls parseLeave, 104
uses $NoValue, 104
defun, 104
parseAtSign, 108
calls parseTran, 108
calls parseType, 108
uses $InteractiveMode, 108
defun, 108
parseCategory, 109
calls contained, 109
calls parseDropAssertions, 109
calls parseTranList, 109
defun, 109
parseCoerce, 110
calls parseTran, 110
calls parseType, 110
uses $InteractiveMode, 110
defun, 110
parseColon, 110
calls parseTran, 110
calls parseType, 110
uses $InteractiveMode, 110
uses $insideConstructIfTrue, 110
defun, 110
parseConstruct, 105
calledby parseTran, 103
calls parseTranList, 105
uses $insideConstructIfTrue, 105
defun, 105
parseDEF, 111
calls opFf, 111
calls parseLhs, 111
calls parseTranCheckForRecord, 111
calls parseTranList, 111
calls setDefOp, 111
uses $lhs, 111
defun, 111
parseDollarGreaterEqual, 114
calls msubst, 114
calls parseTran, 114
uses $op, 114
defun, 114
parseDollarGreaterThan, 114
calls msubst, 114
calls parseTran, 114
uses $op, 114
defun, 114
parseDollarLessEqual, 115
calls msubst, 115
calls parseTran, 115
uses $op, 115
defun, 115
parseDollarNotEqual, 115
calls msubst, 116
calls parseTran, 115
uses $op, 116
defun, 115
parseDropAssertions, 109
calledby parseCategory, 109
calledby parseDropAssertions, 109
calls parseDropAssertions, 109
defun, 109
parseEquivalence, 116
calls parseIf, 116
defun, 116
parseExit, 117
calls moan, 117
calls parseTran, 117
defun, 117
parseGreaterEqual, 117
calls parseTran, 117
uses $op, 117
defun, 117
parseGreaterThan, 118
calls parseTran, 118
uses $op, 118
defun, 118
parseHas, 118
calls getdatabase, 118
calls makeNonAtomic, 119
calls member, 119
calls nreverse0, 119
calls opOf, 118

calls parseHasRhs, 119
calls parseType, 119
calls qcar, 118
calls qcdr, 118
calls unabbrevAndLoad, 118
uses $CategoryFrame, 119
uses $InteractiveMode, 119
defun, 118
parseHasRhs, 120
calledby parseHas, 119
calls abbreviation?, 120
calls get, 120
calls loadIfNecessary, 120
calls member, 120
calls qcar, 120
calls qcdr, 120
calls unabbrevAndLoad, 120
uses $CategoryFrame, 120
defun, 120
parseIf, 123
calledby parseAnd, 107
calledby parseEquivalence, 116
calledby parseImplies, 124
calledby parseOr, 133
calls parseIf,ifTran, 123
calls parseTran, 123
defun, 123
parseIf,ifTran, 121
calledby parseIf,ifTran, 121
calledby parseIf, 123
calls incExitLevel, 121
calls makeSimplePredicateOrNil, 121
calls parseIf,ifTran, 121
calls parseTran, 121
uses $InteractiveMode, 121
defun, 121
parseImplies, 124
calls parseIf, 124
defun, 124
parseIn, 124
calledby parseInBy, 125
calls parseTran, 124
calls postError, 124
defun, 124
parseInBy, 125
calls bright, 125
calls parseIn, 125
calls parseTran, 125
calls postError, 125
defun, 125
parseIs, 126
calls parseTran, 126
calls transIs, 126
defun, 126
parseIsnt, 127
calls parseTran, 127
calls transIs, 127
defun, 127
parseJoin, 127
calls parseTranList, 127
defun, 127
parseLeave, 128
calledby parseAtom, 104
calls parseTran, 128
defun, 128
parseLessEqual, 129
calls parseTran, 129
uses $op, 129
defun, 129
parseLET, 129
calls opOf, 129
calls parseTranCheckForRecord, 129
calls parseTran, 129
calls transIs, 129
defun, 129
parseLETD, 130
calls parseTran, 130
calls parseType, 130
defun, 130
parseLhs, 112
calledby parseDEF, 111
calls parseTran, 112
calls transIs, 112
defun, 112
parseMDEF, 131
calls opOf, 131
calls parseTranCheckForRecord, 131
calls parseTranList, 131
calls parseTran, 131
uses $lhs, 131
defun, 131
ParseMode, 241

usedby PARSE-Expression, 249
usedby PARSE-Operation, 251
defvar, 241
parseNot, 132
calls parseTran, 132
uses $InteractiveMode, 132
defun, 132
parseNotEqual, 132
calls msubst, 132
calls parseTran, 132
uses $op, 132
defun, 132
parseOr, 133
calledby parseOr, 133
calls parseIf, 133
calls parseOr, 133
calls parseTranList, 133
calls parseTran, 133
defun, 133
parsepiles, 86
calledby preparse1, 81
calls add-parens-and-semis-to-line, 86
defun, 86
parsePretend, 133
calls parseTran, 133
calls parseType, 134
defun, 133
parseprint, 308
calledby preparse, 76
defun, 308
parseReturn, 134
calls moan, 134
calls parseTran, 134
defun, 134
parseSegment, 135
calls parseTran, 135
defun, 135
parseSeq, 135
calls last, 135
calls mapInto, 135
calls postError, 135
calls transSeq, 135
defun, 135
parseTran, 103
calledby compReduce1, 175
calledby parseAnd, 107
calledby parseAtSign, 108
calledby parseCoerce, 110
calledby parseColon, 110
calledby parseDollarGreaterEqual, 114
calledby parseDollarGreaterThan, 114
calledby parseDollarLessEqual, 115
calledby parseDollarNotEqual, 115
calledby parseExit, 117
calledby parseGreaterEqual, 117
calledby parseGreaterThan, 118
calledby parseIf,ifTran, 121
calledby parseIf, 123
calledby parseInBy, 125
calledby parseIn, 124
calledby parseIsnt, 127
calledby parseIs, 126
calledby parseLETD, 130
calledby parseLET, 129
calledby parseLeave, 128
calledby parseLessEqual, 129
calledby parseLhs, 112
calledby parseMDEF, 131
calledby parseNotEqual, 132
calledby parseNot, 132
calledby parseOr, 133
calledby parsePretend, 133
calledby parseReturn, 134
calledby parseSegment, 135
calledby parseTranCheckForRecord, 297
calledby parseTranList, 105
calledby parseTransform, 103
calledby parseTran, 103
calledby parseType, 108
calls getl, 103
calls parseAtom, 103
calls parseConstruct, 103
calls parseTranList, 103
calls parseTran, 103
uses $op, 103
defun, 103
parseTranCheckForRecord, 297
calledby parseDEF, 111

calledby parseLET, 129
calledby parseMDEF, 131
calls parseTran, 297
calls postError, 297
calls qcar, 297
calls qcdr, 297
defun, 297
parseTranList, 105
calledby parseAnd, 107
calledby parseCategory, 109
calledby parseConstruct, 105
calledby parseDEF, 111
calledby parseJoin, 127
calledby parseMDEF, 131
calledby parseOr, 133
calledby parseTranList, 105
calledby parseTran, 103
calledby parseVCONS, 136
calls parseTranList, 105
calls parseTran, 105
defun, 105
parseTransform, 103
calledby s-process, 330
calls msubst, 103
calls parseTran, 103
uses $defOp, 103
defun, 103
parseType, 108
calledby parseAtSign, 108
calledby parseCoerce, 110
calledby parseColon, 110
calledby parseHas, 119
calledby parseLETD, 130
calledby parsePretend, 134
calls msubst, 108
calls parseTran, 108
defun, 108
parseVCONS, 136
calls parseTranList, 136
defun, 136
parseWhere, 137
calls mapInto, 137
defun, 137
pathname[5]
called by compileSpad2Cmd, 316
called by compileSpadLispCmd, 365
called by compiler, 313
pathnameDirectory[5]
called by compileSpadLispCmd, 365
pathnameName[5]
called by compileSpadLispCmd, 365
pathnameType[5]
called by compileSpad2Cmd, 316
called by compileSpadLispCmd, 365
called by compiler, 313
pname[5]
called by comp3, 340
called by floatexpid, 289
called by getScriptName, 230
Pop-Reduction, 304
calledby pop-stack-1, 302
calledby pop-stack-2, 303
calledby pop-stack-3, 303
calledby pop-stack-4, 303
calls stack-pop, 304
defun, 304
pop-stack-1, 302
calledby PARSE-Application, 256
calledby PARSE-Category, 248
calledby PARSE-CommandTail, 245
calledby PARSE-Conditional, 275
calledby PARSE-Data, 266
calledby PARSE-Enclosure, 262
calledby PARSE-Exit, 274
calledby PARSE-Expression, 249
calledby PARSE-Expr, 250
calledby PARSE-FloatTok, 277
calledby PARSE-Float, 259
calledby PARSE-Form, 256
calledby PARSE-Import, 249
calledby PARSE-InfixWith, 247
calledby PARSE-Infix, 253
calledby PARSE-Iterator, 271
calledby PARSE-LabelExpr, 276
calledby PARSE-Leave, 274
calledby PARSE-LedPart, 250
calledby PARSE-Loop, 276
calledby PARSE-Name, 265
calledby PARSE-NudPart, 251
calledby PARSE-Prefix, 253
calledby PARSE-Primary1, 258
calledby PARSE-Qualification, 254

calledby PARSE-Reduction, 255
calledby PARSE-Return, 273
calledby PARSE-ScriptItem, 265
calledby PARSE-Seg, 275
calledby PARSE-Selector, 257
calledby PARSE-SemiColon, 273
calledby PARSE-Sequence1, 269
calledby PARSE-Sequence, 269
calledby PARSE-Sexpr1, 267
calledby PARSE-SpecialCommand, 243
calledby PARSE-Statement, 246
calledby PARSE-Suffix, 272
calledby PARSE-TokenCommandTail, 244
calledby PARSE-VarForm, 264
calledby PARSE-With, 247
calledby spad, 329
calledby star, 291
calls Pop-Reduction, 302
calls reduction-value, 302
defmacro, 302
pop-stack-2, 303
calledby PARSE-Application, 256
calledby PARSE-Category, 247
calledby PARSE-CommandTail, 245
calledby PARSE-Conditional, 275
calledby PARSE-Float, 259
calledby PARSE-Import, 249
calledby PARSE-InfixWith, 247
calledby PARSE-Infix, 253
calledby PARSE-Iterator, 271
calledby PARSE-LabelExpr, 276
calledby PARSE-Loop, 276
calledby PARSE-Prefix, 253
calledby PARSE-Primary1, 258
calledby PARSE-Reduction, 255
calledby PARSE-ScriptItem, 265
calledby PARSE-Seg, 275
calledby PARSE-Selector, 257
calledby PARSE-SemiColon, 273
calledby PARSE-Sequence1, 269
calledby PARSE-Sexpr1, 266
calledby PARSE-Statement, 246
calledby PARSE-TokenCommandTail, 244
calledby PARSE-VarForm, 264
calls Pop-Reduction, 303
calls reduction-value, 303
calls stack-push, 303
defmacro, 303
pop-stack-3, 303
calledby PARSE-Category, 247
calledby PARSE-Conditional, 275
calledby PARSE-Float, 259
calledby PARSE-Iterator, 271
calls Pop-Reduction, 303
calls reduction-value, 303
calls stack-push, 303
defmacro, 303
pop-stack-4, 303
calledby PARSE-Float, 259
calls Pop-Reduction, 303
calls reduction-value, 303
calls stack-push, 303
defmacro, 303
postAdd, 198
calls postCapsule, 198
calls postTran, 198
defun, 198
postAtom, 193
calledby postTran, 192
uses $boot, 193
defun, 193
postAtSign, 200
calls postTran, 200
calls postType, 200
defun, 200
postBigFloat, 201
calls postTran, 201
uses $InteractiveMode, 201
uses $boot, 201
defun, 201
postBlock, 202
calledby postSemiColon, 220
calls postBlockItemList, 202
calls postTran, 202
defun, 202
postBlockItem
calledby postBlockItemList, 199
calledby postCapsule, 199
postBlockItemList, 199

calledby postBlock, 202
calledby postCapsule, 199
calls postBlockItem, 199
defun, 199
postCapsule, 199
calledby postAdd, 198
calls checkWarning, 199
calls postBlockItemList, 199
calls postBlockItem, 199
calls postFlatten, 199
defun, 199
postCategory, 202
calls nreverse0, 202
calls postTran, 202
uses $insidePostCategoryIfTrue, 202
defun, 202
postcheck, 195
calledby postTransformCheck, 195
calledby postcheck, 195
calls postcheck, 195
calls setDefOp, 195
defun, 195
postCollect, 204
calledby postCollect, 204
calledby postTupleCollect, 224
calls postCollect,finish, 204
calls postCollect, 204
calls postIteratorList, 204
calls postTran, 204
defun, 204
postCollect,finish, 203
calledby postCollect, 204
calls postMakeCons, 203
calls postTranList, 203
calls qcar, 203
calls qcdr, 203
calls tuple2List, 203
defun, 203
postColon, 206
calls postTran, 206
calls postType, 206
defun, 206
postColonColon, 207
calls postForm, 207
uses $boot, 207
defun, 207
postComma, 207
calls comma2Tuple, 207
calls postTuple, 207
defun, 207
postConstruct, 208
calls comma2Tuple, 208
calls postMakeCons, 208
calls postTranList, 209
calls postTranSegment, 208
calls postTran, 209
calls tuple2List, 208
defun, 208
postDef, 210
calls nequal, 210
calls nreverse0, 210
calls postDefArgs, 210
calls postMDef, 210
calls postTran, 210
calls recordHeaderDocumentation, 210
uses $InteractiveMode, 210
uses $boot, 210
uses $docList, 210
uses $headerDocumentation, 210
uses $maxSignatureLineNumber, 210
defun, 210
postDefArgs, 211
calledby postDefArgs, 211
calledby postDef, 210
calls postDefArgs, 211
calls postError, 211
defun, 211
postError, 196
calledby checkWarning, 301
calledby getScriptName, 230
calledby parseInBy, 125
calledby parseIn, 124
calledby parseSeq, 135
calledby parseTranCheckForRecord, 297
calledby postDefArgs, 211
calledby postForm, 196
calls bumperrorcount, 196
calls nequal, 196
uses $InteractiveMode, 196
uses $defOp, 196

uses $postStack, 196
defun, 196
postExit, 212
calls postTran, 212
defun, 212
postFlatten, 208
calledby comma2Tuple, 207
calledby postCapsule, 199
calledby postFlatten, 208
calls postFlatten, 208
defun, 208
postFlattenLeft, 221
calledby postFlattenLeft, 221
calledby postSemiColon, 220
calls postFlattenLeft, 221
defun, 221
postForm, 196
calledby postColonColon, 207
calledby postTran, 192
calls bright, 196
calls internl, 196
calls postError, 196
calls postTranList, 196
calls postTran, 196
uses $boot, 196
defun, 196
postIf, 213
calls nreverse0, 213
calls postTran, 213
uses $boot, 213
defun, 213
postIn, 214
calls postInSeq, 214
calls postTran, 214
calls systemErrorHere, 214
defun, 214
postin, 213
calls postInSeq, 213
calls postTran, 213
calls systemErrorHere, 213
defun, 213
postInSeq, 214
calledby postIn, 214
calledby postIteratorList, 205
calledby postin, 213
calls postTranSegment, 214
calls postTran, 214
calls tuple2List, 214
defun, 214
postIteratorList, 205
calledby postCollect, 204
calledby postIteratorList, 205
calledby postRepeat, 219
calls postInSeq, 205
calls postIteratorList, 205
calls postTran, 205
defun, 205
postJoin, 215
calls postTranList, 215
calls postTran, 215
defun, 215
postMakeCons, 204
calledby postCollect,finish, 203
calledby postConstruct, 208
calledby postMakeCons, 204
calls postMakeCons, 204
calls postTran, 204
defun, 204
postMapping, 216
calls postTran, 216
calls unTuple, 216
defun, 216
postMDef, 216
calledby postDef, 210
calls nreverse0, 216
calls postTran, 216
calls throwkeyedmsg, 216
uses $InteractiveMode, 216
uses $boot, 216
defun, 216
postOp, 193
calledby postTran, 192
defun, 193
postPretend, 218
calls postTran, 218
calls postType, 218
defun, 218
postQUOTE, 218
defun, 218
postReduce, 219
calledby postReduce, 219
calls postReduce, 219

calls postTran, 219
uses $InteractiveMode, 219
defun, 219
postRepeat, 219
calls postIteratorList, 219
calls postTran, 219
defun, 219
postScripts, 220
calls getScriptName, 220
calls postTranScripts, 220
defun, 220
postScriptsForm, 194
calledby postTran, 192
calls getScriptName, 194
calls length, 194
calls postTranScripts, 194
defun, 194
postSemiColon, 220
calls postBlock, 220
calls postFlattenLeft, 220
defun, 220
postSignature, 221
calls killColons, 221
calls pairp, 221
calls postType, 221
calls removeSuperfluousMapping, 221
defun, 221
postSlash, 222
calls postTran, 222
defun, 222
postTran, 192
calledby postAdd, 198
calledby postAtSign, 200
calledby postBigFloat, 201
calledby postBlock, 202
calledby postCategory, 202
calledby postCollect, 204
calledby postColon, 206
calledby postConstruct, 209
calledby postDef, 210
calledby postExit, 212
calledby postForm, 196
calledby postIf, 213
calledby postInSeq, 214
calledby postIn, 214
calledby postIteratorList, 205
calledby postJoin, 215
calledby postMDef, 216
calledby postMakeCons, 204
calledby postMapping, 216
calledby postPretend, 218
calledby postReduce, 219
calledby postRepeat, 219
calledby postSlash, 222
calledby postTranList, 194
calledby postTranScripts, 194
calledby postTranSegment, 209
calledby postTransform, 191
calledby postTran, 192
calledby postType, 200
calledby postWhere, 224
calledby postWith, 225
calledby postin, 213
calledby tuple2List, 302
calls pairp, 192
calls postAtom, 192
calls postForm, 192
calls postOp, 192
calls postScriptsForm, 192
calls postTranList, 192
calls postTran, 192
calls qcar, 192
calls qcdr, 192
calls unTuple, 192
defun, 192
postTranList, 194
calledby postCollect,finish, 203
calledby postConstruct, 209
calledby postForm, 196
calledby postJoin, 215
calledby postTran, 192
calledby postTuple, 223
calledby postWhere, 224
calls postTran, 194
defun, 194
postTranScripts, 194
calledby postScriptsForm, 194
calledby postScripts, 220
calledby postTranScripts, 194
calls postTranScripts, 194
calls postTran, 194

defun, 194
postTranSegment, 209
calledby postConstruct, 208
calledby postInSeq, 214
calledby tuple2List, 302
calls postTran, 209
defun, 209
postTransform, 191
calledby new2OldLisp, 298
calledby s-process, 330
calls aplTran, 191
calls identp[5], 191
calls postTransformCheck, 191
calls postTran, 191
defun, 191
postTransformCheck, 195
calledby postTransform, 191
calls postcheck, 195
uses $defOp, 195
defun, 195
postTuple, 223
calledby postComma, 207
calls postTranList, 223
defun, 223
postTupleCollect, 224
calls postCollect, 224
defun, 224
postType, 200
calledby postAtSign, 200
calledby postColon, 206
calledby postPretend, 218
calledby postSignature, 221
calls postTran, 200
calls unTuple, 200
defun, 200
postWhere, 224
calls postTranList, 224
calls postTran, 224
defun, 224
postWith, 225
calls postTran, 225
uses $insidePostCategoryIfTrue, 225
defun, 225
PredImplies
calledby compForm2, 350
preparse, 71, 76
calledby preparse, 76
calledby spad, 329
calls ifcar, 77
calls parseprint, 76
calls preparse1, 76
calls preparse, 76
uses $comblocklist, 77
uses $constructorLineNumber, 77
uses $docList, 77
uses $headerDocumentation, 77
uses $index, 77
uses $maxSignatureLineNumber, 77
uses $preparse-last-line, 77
uses $preparseReportIfTrue, 77
uses $skipme, 77
defun, 76
preparse-echo, 90
calledby fincomblock, 306
calledby preparse1, 81
uses Echo-Meta, 90
uses $EchoLineStack, 90
defun, 90
preparse1, 81
calledby preparse, 76
calls doSystemCommand[5], 81
calls escaped, 81
calls fincomblock, 81
calls indent-pos, 81
calls is-console, 81
calls make-full-cvec, 81
calls maxindex, 81
calls parsepiles, 81
calls preparse-echo, 81
calls preparseReadLine, 81
calls strposl[5], 81
uses $byConstructors, 81
uses $constructorsSeen, 81
uses $echolinestack, 81
uses $linelist, 81
uses $preparse-last-line, 81
uses $skipme, 81
catches, 81
defun, 81
preparseReadLine, 88
calledby preparse1, 81
calledby preparseReadLine, 88

calledby skip-to-endif, 308
calls dcq, 88
calls initial-substring, 88
calls preparseReadLine1, 88
calls preparseReadLine, 88
calls skip-to-endif, 88
calls storeblanks, 88
calls string2BootTree, 88
defun, 88
preparseReadLine1, 89
calledby preparseReadLine1, 89
calledby preparseReadLine, 88
calledby skip-ifblock, 88
calledby skip-to-endif, 308
calls expand-tabs, 89
calls get-a-line, 89
calls maxindex, 89
calls preparseReadLine1, 89
calls strconc, 89
uses $EchoLineStack, 89
uses $index, 89
uses $linelist, 89
uses $preparse-last-line, 89
defun, 89
pretend, 133, 171, 217
defplist, 133, 171, 217
prettyprint
calledby s-process, 330
primitiveType, 345
calledby compAtom, 343
uses $DoubleFloat, 345
uses $EmptyMode, 345
uses $NegativeInteger, 345
uses $NonNegativeInteger, 345
uses $PositiveInteger, 345
uses $String, 345
defun, 345
print-defun, 333
calls is-console, 333
calls print-full, 333
uses $PrettyPrint, 333
uses vmlisp::optionlist, 333
defun, 333
print-full
calledby print-defun, 333
print-package, 301
defun, 301
prior-token, 99
usedby PARSE-Expression, 249
uses $token, 99
defvar, 99
processFunctorOrPackage
calledby compCapsuleInner, 144
processInteractive[5]
called by s-process, 330
processSynonyms[5]
called by PARSE-NewExpr, 241
profileRecord
calledby setqSingle, 182
push-reduction, 292
calledby PARSE-AnyId, 268
calledby PARSE-Application, 256
calledby PARSE-Category, 247
calledby PARSE-CommandTail, 245
calledby PARSE-Command, 242
calledby PARSE-Conditional, 275
calledby PARSE-Data, 266
calledby PARSE-Enclosure, 262
calledby PARSE-Exit, 273
calledby PARSE-Expression, 249
calledby PARSE-Expr, 250
calledby PARSE-FloatBasePart, 261
calledby PARSE-FloatBase, 260
calledby PARSE-FloatExponent, 261
calledby PARSE-FloatTok, 277
calledby PARSE-Float, 259
calledby PARSE-Form, 255
calledby PARSE-Import, 249
calledby PARSE-InfixWith, 247
calledby PARSE-Infix, 253
calledby PARSE-LabelExpr, 276
calledby PARSE-Leave, 274
calledby PARSE-LedPart, 250
calledby PARSE-Loop, 276
calledby PARSE-Name, 265
calledby PARSE-NudPart, 250
calledby PARSE-OpenBrace, 270
calledby PARSE-OpenBracket, 270
calledby PARSE-Prefix, 252, 253
calledby PARSE-Primary1, 258
calledby PARSE-PrimaryOrQM, 245
calledby PARSE-Quad, 263

calledby PARSE-Qualification, 254
calledby PARSE-Reduction, 255
calledby PARSE-Return, 273
calledby PARSE-ScriptItem, 265
calledby PARSE-Seg, 274
calledby PARSE-Selector, 257
calledby PARSE-SemiColon, 273
calledby PARSE-Sequence1, 269
calledby PARSE-Sequence, 269
calledby PARSE-Sexpr1, 266
calledby PARSE-SpecialCommand, 243
calledby PARSE-Statement, 246
calledby PARSE-Suffix, 272
calledby PARSE-TokenCommandTail, 244
calledby PARSE-TokenList, 244
calledby PARSE-VarForm, 264
calledby PARSE-With, 247
calledby parse-argument-designator, 301
calledby parse-identifier, 299
calledby parse-keyword, 300
calledby parse-number, 300
calledby parse-spadstring, 298
calledby parse-string, 299
calledby star, 291
calls make-reduction, 292
calls stack-push, 292
uses reduce-stack, 292
defun, 292

put
calledby compColon, 151
calledby compMacro, 170
calledby compSubsetCategory, 186
calledby compSuchthat, 187
calledby compTypeOf, 342

qcar
calledby compAdd, 140
calledby compCategory, 147
calledby compCons1, 154
calledby compJoin, 166
calledby compLambda, 168
calledby compMacro, 170
calledby compSetq1, 181
calledby compWithMappingMode1, 355
calledby decodeScripts, 230
calledby parseHasRhs, 120
calledby parseHas, 118
calledby parseTranCheckForRecord, 297
calledby postCollect,finish, 203
calledby postTran, 192
calledby transIs1, 112

qcdr
calledby compAdd, 140
calledby compCategory, 147
calledby compCons1, 154
calledby compJoin, 166
calledby compLambda, 168
calledby compSetq1, 181
calledby compWithMappingMode1, 355
calledby decodeScripts, 230
calledby parseHasRhs, 120
calledby parseHas, 118
calledby parseTranCheckForRecord, 297
calledby postCollect,finish, 203
calledby postTran, 192
calledby transIs1, 112

quote, 172, 218
defplist, 172, 218

quote-if-string, 280
calledby match-advance-string, 279
calledby unget-tokens, 282
calls escape-keywords, 280
calls pack, 280
calls strconc, 280
calls token-nonblank, 280
calls token-symbol, 280
calls token-type, 280
calls underscore, 280
uses $boot, 280
uses $spad, 281
defun, 280

quotify
calledby compIf, 164

rbp

usedby PARSE-NudPart, 251
usedby PARSE-Operation, 251
read-a-line, 91
calledby get-a-line, 96
calledby read-a-line, 91
calls Line-New-Line, 91
calls read-a-line, 91
calls subseq, 91
uses *eof*, 91
defun, 91
recompile-lib-file-if-necessary, 366
calledby compileSpadLispCmd, 365
calls compile-lib-file, 366
uses *lisp-bin-filetype*, 367
defun, 366
Record, 146
defplist, 146
recordAttributeDocumentation
calledby PARSE-Category, 248
RecordCategory, 155
defplist, 155
recordHeaderDocumentation
calledby postDef, 210
recordSignatureDocumentation
calledby PARSE-Category, 248
reduce, 175, 218
defplist, 175, 218
reduce-stack, 292
usedby push-reduction, 292
uses $stack, 292
defvar, 292
reduce-stack-clear, 292
defmacro, 292
reduction, 101
defstruct, 101
reduction-value
calledby nth-stack, 304
calledby pop-stack-1, 302
calledby pop-stack-2, 303
calledby pop-stack-3, 303
calledby pop-stack-4, 303
refvecp
calledby translabel1, 295
remdup
calledby displayPreCompilationErrors, 296
removeEnv
calledby setqSingle, 182
removeSuperfluousMapping
calledby postSignature, 221
repeat, 173, 219
defplist, 173, 219
replaceExitEtc
calledby compSeq1, 179
resolve
calledby compCategory, 147
calledby compCoerce1, 149
calledby compConstructorCategory, 156
calledby compIf, 164
calledby compReturn, 177
calledby compString, 184
calledby convert, 344
calledby modifyModeStack, 363
return, 134, 177
defplist, 134, 177

s-process, 330
calledby spad, 329
calls compTopLevel, 330
calls curstrm, 330
calls def-process, 330
calls def-rename, 330
calls displayPreCompilationErrors, 330
calls displaySemanticErrors, 330
calls get-internal-run-time, 330
calls new2OldLisp, 330
calls parseTransform, 330
calls postTransform, 330
calls prettyprint, 330
calls processInteractive[5], 330
calls terpri, 330
uses $DomainFrame, 331
uses $EmptyEnvironment, 331
uses $EmptyMode, 331
uses $Index, 330
uses $LocalFrame, 331
uses $PolyMode, 331
uses $Translation, 331
uses $VariableCount, 331
uses $compUniquelyIfTrue, 331

uses $currentFunction, 331
uses $exitModeStack, 331
uses $exitMode, 331
uses $e, 331
uses $form, 331
uses $genFVar, 331
uses $genSDVar, 331
uses $insideCapsuleFunctionIfTrue, 331
uses $insideCategoryIfTrue, 331
uses $insideCoerceInteractiveHardIfTrue, 331
uses $insideExpressionIfTrue, 331
uses $insideFunctorIfTrue, 331
uses $insideWhereIfTrue, 331
uses $leaveLevelStack, 331
uses $leaveMode, 331
uses $macroassoc, 331
uses $newspad, 331
uses $postStack, 331
uses $previousTime, 331
uses $returnMode, 331
uses $semanticErrorStack, 331
uses $top-level, 331
uses $topOp, 331
uses $warningStack, 331
uses curoutstream, 331
defun, 330
say
calledby compOrCroak1, 336
calledby modifyModeStack, 363
sayBrightly
calledby compMacro, 170
calledby compilerDoit, 318
calledby displayPreCompilationErrors, 296
sayKeyedMsg[5]
called by compileSpad2Cmd, 316
called by compileSpadLispCmd, 365
sayMath
calledby displayPreCompilationErrors, 296
ScanOrPairVec[5]
called by hasFormalMapVariable, 361
Scripts, 220
defplist, 220
segment, 135
defplist, 135
selectOptionLC[5]
called by compileSpad2Cmd, 316
called by compileSpadLispCmd, 365
called by compiler, 313
seq, 178
defplist, 178
setDefOp, 225
calledby parseDEF, 111
calledby postcheck, 195
uses $defOp, 225
uses $topOp, 225
defun, 225
setelt
calledby modifyModeStack, 363
setq, 180
defplist, 180
setqMultiple
calledby compSetq1, 181
setqSetelt, 181
calledby compSetq1, 181
calls comp, 181
defun, 181
setqSingle, 182
calledby compSetq1, 181
calls NRTassocIndex, 182
calls addBinding[5], 182
calls assignError, 182
calls augModemapsFromDomain1, 182
calls comp, 182
calls consProplistOf, 182
calls convert, 182
calls getProplist[5], 182
calls getmode, 182
calls get, 182
calls identp[5], 182
calls isDomainForm, 182
calls isDomainInScope, 182
calls maxSuperType, 182
calls nequal, 182
calls outputComp, 182
calls profileRecord, 182
calls removeEnv, 182

calls stackWarning, 182
uses $EmptyMode, 182
uses $NoValueMode, 182
uses $QuickLet, 182
uses $form, 182
uses $insideSetqSingleIfTrue, 182
uses $profileCompiler, 182
defun, 182
shut[5]
called by spad, 329
Signature, 221
defplist, 221
skip-blanks, 278
calledby match-string, 278
calls advance-char, 278
calls current-char, 278
calls token-lookahead-type, 278
defun, 278
skip-ifblock, 88
calledby skip-ifblock, 88
calls initial-substring, 88
calls preparseReadLine1, 88
calls skip-ifblock, 88
calls storeblanks, 88
calls string2BootTree, 88
defun, 88
skip-to-endif, 308
calledby preparseReadLine, 88
calledby skip-to-endif, 308
calls initial-substring, 308
calls preparseReadLine1, 308
calls preparseReadLine, 308
calls skip-to-endif, 308
defun, 308
spad, 329
calls PARSE-NewExpr, 329
calls addBinding[5], 329
calls init-boot/spad-reader[5], 329
calls initialize-preparse, 329
calls ioclear, 329
calls makeInitialModemapFrame[5], 329
calls pop-stack-1, 329
calls preparse, 329
calls s-process, 329
calls shut[5], 329
uses *comp370-apply*, 329
uses *eof*, 329
uses /editfile, 329
uses $InitialDomainsInScope, 329
uses $InteractiveFrame, 329
uses $InteractiveMode, 329
uses $noSubsumption, 329
uses echo-meta, 329
uses file-closed, 329
uses line, 329
catches, 329
defun, 329
spad-fixed-arg, 367
defun, 367
spad2AsTranslatorAutoloadOnceTrigger
calledby compileSpad2Cmd, 316
spad[5]
called by /rf-1, 320
SpadInterpretStream[5]
called by ncINTERPFILE, 365
spadPrompt
calledby compileSpad2Cmd, 316
calledby compileSpadLispCmd, 365
spadreduce
calledby floatexpid, 289
stack, 97
defstruct, 97
stack-/-empty, 98
uses $stack, 98
defmacro, 98
stack-clear, 97
uses $stack, 97
defun, 97
stack-load, 97
uses $stack, 97
defun, 97
stack-pop, 98
calledby Pop-Reduction, 304
uses $stack, 98
defun, 98
stack-push, 98
calledby pop-stack-2, 303
calledby pop-stack-3, 303
calledby pop-stack-4, 303
calledby push-reduction, 292
uses $stack, 98

defun, 98
stack-size
calledby star, 291
stack-store
calledby nth-stack, 304
stackAndThrow
calledby compDefine1, 159
calledby compLambda, 168
calledby compWithMappingMode1, 355
stackMessage
calledby compElt, 161
calledby compRepeatOrCollect, 173
calledby compSymbol, 345
stackMessageIfNone
calledby compExit, 162
calledby compForm, 348
stackSemanticError
calledby compColonInside, 342
calledby compJoin, 166
calledby compOrCroak1, 336
calledby compPretend, 171
calledby compReturn, 177
calledby compSubDomain1, 185
stackWarning
calledby compColonInside, 342
calledby compElt, 161
calledby compPretend, 171
calledby setqSingle, 182
star, 291
calledby PARSE-Application, 256
calledby PARSE-Category, 248
calledby PARSE-CommandTail, 245
calledby PARSE-Expr, 250
calledby PARSE-Import, 249
calledby PARSE-IteratorTail, 271
calledby PARSE-Loop, 276
calledby PARSE-Option, 246
calledby PARSE-ScriptItem, 265
calledby PARSE-Sexpr1, 267
calledby PARSE-SpecialCommand, 243
calledby PARSE-Statement, 246
calledby PARSE-TokenCommandTail, 244
calledby PARSE-TokenList, 244
calls pop-stack-1, 291
calls push-reduction, 291
calls stack-size, 291
defmacro, 291
step
calledby floatexpid, 289
storeblanks, 95
calledby preparseReadLine, 88
calledby skip-ifblock, 88
defun, 95
strconc
calledby compDefine1, 159
calledby compileSpad2Cmd, 316
calledby decodeScripts, 230
calledby preparseReadLine1, 89
calledby quote-if-string, 280
calledby unget-tokens, 283
String, 183
defplist, 183
string-not-greaterp
calledby initial-substring-p, 280
string2BootTree
calledby preparseReadLine, 88
calledby skip-ifblock, 88
string2id-n
calledby infixtok, 307
stringPrefix?
calledby comp3, 340
strposl[5]
called by preparse1, 81
SubDomain, 184
defplist, 184
sublis
calledby compForm2, 350
subseq
calledby match-string, 278
calledby read-a-line, 91
SubsetCategory, 186
defplist, 186
suffix
calledby addclose, 304
systemCommand[5]
called by PARSE-CommandTail, 245
called by PARSE-TokenCommandTail, 244

systemError
calledby compReduce1, 175
calledby errhuh, 233
systemErrorHere
calledby compCategory, 148
calledby compColon, 151
calledby postIn, 214
calledby postin, 213

take
calledby compColon, 151
calledby compForm2, 350
calledby compWithMappingMode1, 355
calledby drop, 305
terminateSystemCommand[5]
called by compileSpad2Cmd, 316
called by compileSpadLispCmd, 365
terpri
calledby s-process, 330
throwKeyedMsg
calledby compileSpad2Cmd, 316
calledby compileSpadLispCmd, 365
calledby compiler, 313
throwkeyedmsg
calledby postMDef, 216
tmptok, 240
usedby PARSE-Operation, 251
defvar, 240
tok, 240
usedby PARSE-GliphTok, 268
usedby PARSE-NBGliphTok, 267
defvar, 240
token, 99
defstruct, 99
token-install, 100
uses $token, 100
defun, 100
token-lookahead-type, 279
calledby skip-blanks, 278
uses Escape-Character, 279
defun, 279
token-nonblank
calledby PARSE-FloatBasePart, 261
calledby quote-if-string, 280
calledby unget-tokens, 283
token-print, 101
uses $token, 101
defun, 101
token-symbol
calledby PARSE-SpecialKeyWord, 242
calledby match-token, 284
calledby parse-argument-designator, 301
calledby parse-identifier, 299
calledby parse-keyword, 300
calledby parse-number, 300
calledby parse-spadstring, 298
calledby parse-string, 299
calledby quote-if-string, 280
token-type
calledby match-token, 284
calledby quote-if-string, 280
transIs, 112
calledby parseIsnt, 127
calledby parseIs, 126
calledby parseLET, 129
calledby parseLhs, 112
calledby transIs1, 112
calls isListConstructor, 112
calls transIs1, 112
defun, 112
transIs1, 112
calledby transIs1, 112
calledby transIs, 112
calls nreverse0, 112
calls pairp, 112
calls qcar, 112
calls qcdr, 112
calls transIs1, 112
calls transIs, 112
defun, 112
translabel, 295
calledby PARSE-Data, 266
calls translabel1, 295
defun, 295
translabel1, 295
calledby translabel1, 295
calledby translabel, 295
calls lassoc, 295
calls maxindex, 295

calls refvecp, 295
calls translabel1, 295
defun, 295
transSeq
calledby parseSeq, 135
try-get-token, 285
calledby advance-token, 286
calledby current-token, 285
calledby next-token, 286
calls get-token, 285
uses valid-tokens, 285
defun, 285
tuple2List, 302
calledby postCollect,finish, 203
calledby postConstruct, 208
calledby postInSeq, 214
calledby tuple2List, 302
calls postTranSegment, 302
calls postTran, 302
calls tuple2List, 302
uses $InteractiveMode, 302
uses $boot, 302
defun, 302
TupleCollect, 223
defplist, 223

unabbrevAndLoad
calledby parseHasRhs, 120
calledby parseHas, 118
unAbbreviateKeyword[5]
called by PARSE-SpecialKeyWord, 242
underscore, 282
calledby quote-if-string, 280
calls vector-push, 282
defun, 282
unget-tokens, 282
calledby match-string, 278
calls line-current-segment, 283
calls line-new-line, 283
calls line-number, 283
calls quote-if-string, 282
calls strconc, 283
calls token-nonblank, 283
uses valid-tokens, 283
defun, 282
Union, 146
defplist, 146
union
calledby compJoin, 166
UnionCategory, 156
defplist, 156
unionq
calledby freelist, 363
unknownTypeError
calledby compColon, 151
unTuple, 233
calledby postMapping, 216
calledby postTran, 192
calledby postType, 200
defun, 233
updateSourceFiles[5]
called by compileSpad2Cmd, 316
userError
calledby compOrCroak1, 336
calledby compReturn, 177

valid-tokens, 100
usedby advance-token, 286
usedby current-token, 285
usedby next-token, 286
usedby try-get-token, 285
usedby unget-tokens, 283
uses $token, 100
defvar, 100
vcons, 136
defplist, 136
vector, 187
defplist, 187
vector-push
calledby underscore, 282
VectorCategory, 156
defplist, 156
vmlisp::optionlist
usedby print-defun, 333

where, 136, 188, 224
defplist, 136, 188, 224
with, 225
defplist, 225
wrapDomainSub
calledby compJoin, 166

wrapSEQExit
calledby makeSimplePredicateOrNil, 298

XTokenReader, 287
calledby get-token, 287
usedby get-token, 287
defvar, 287