

axiomTM



The 30 Year Horizon

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

Volume 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 Baumsлаг	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 Lehouby	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	Compiler top level	3
1.1)compile	3
1.1.1	Spad compiler	6
2	The Parser	9
2.1	EQ.spad	9
2.2	Parsing routines	14
2.2.1	defun initialize-prepare	14
2.2.2	defun prepare	18
2.2.3	defun Build the lines from the input for piles	23
2.2.4	defun parsepiles	26
2.2.5	defun add-parens-and-semis-to-line	27
2.2.6	defun prepareReadLine	28
2.2.7	defun prepareReadLine1	29
2.3	I/O Handling	30
2.3.1	defun prepare-echo	30
2.3.2	defun read-a-line	31
2.4	Line Handling	32
2.4.1	Line Buffer	32
2.4.2	defstruct \$Line	32
2.4.3	defun Line-New-Line	32
2.4.4	defun next-line	33
2.4.5	defun storeblanks	33
2.4.6	defun initial-substring	33
2.4.7	defun get-a-line	33
2.4.8	defun make-string-adjustable	34
3	Parse Transformers	35
3.1	Direct called parse routines	35
3.1.1	defun parseTransform	35
3.1.2	defun parseTran	36
3.1.3	defun parseAtom	37
3.1.4	defun parseTranList	37

3.1.5	defun parseConstruct	37
3.1.6	defun parseConstruct	38
3.2	Indirect called parse routines	39
3.2.1	defun parseAnd	40
3.2.2	defun parseAnd	40
3.2.3	defun parseAtSign	40
3.2.4	defun parseAtSign	41
3.2.5	defun parseCategory	41
3.2.6	defun parseCategory	41
3.2.7	defun parseCoerce	41
3.2.8	defun parseCoerce	42
3.2.9	defun parseColon	42
3.2.10	defun parseColon	42
3.2.11	defun parseDEF	43
3.2.12	defun parseDEF	43
3.2.13	defun parseDollarGreaterthan	43
3.2.14	defun parseDollarGreaterThan	44
3.2.15	defun parseDollarGreaterEqual	44
3.2.16	defun parseDollarGreaterEqual	44
3.2.17	defun parseDollarLessEqual	44
3.2.18	defun parseDollarLessEqual	45
3.2.19	defun parseDollarNotEqual	45
3.2.20	defun parseDollarNotEqual	45
3.2.21	defun parseEquivalence	45
3.2.22	defun parseEquivalence	46
3.2.23	defun parseExit	46
3.2.24	defun parseExit	46
3.2.25	defun parseGreaterEqual	47
3.2.26	defun parseGreaterEqual	47
3.2.27	defun parseGreaterThan	47
3.2.28	defun parseGreaterThan	47
3.2.29	defun parseHas	48
3.2.30	defun parseIf,ifTran	51
3.2.31	defun parseIf	53
3.2.32	defun parseIf	53
3.2.33	defun parseImplies	53
3.2.34	defun parseImplies	54
3.2.35	defun parseIn	54
3.2.36	defun parseIn	55
3.2.37	defun parseInBy	56
3.2.38	defun parseInBy	56
3.2.39	defun parseIs	56
3.2.40	defun parseIs	57
3.2.41	defun parseIsnt	57
3.2.42	defun parseIsnt	57
3.2.43	defun parseJoin	57

3.2.44	defun parseJoin	58
3.2.45	defun parseLeave	58
3.2.46	defun parseLeave	59
3.2.47	defun parseLessEqual	59
3.2.48	defun parseLessEqual	59
3.2.49	defun parseLET	59
3.2.50	defun parseLET	60
3.2.51	defun parseLETD	60
3.2.52	defun parseLETD	60
3.2.53	defun parseMDEF	60
3.2.54	defun parseMDEF	61
3.2.55	defun parseNot	61
3.2.56	defun parseNot	61
3.2.57	defun parseNot	62
3.2.58	defun parseNotEqual	62
3.2.59	defun parseNotEqual	62
3.2.60	defun parseOr	62
3.2.61	defun parseOr	63
3.2.62	defun parsePretend	63
3.2.63	defun parsePretend	64
3.2.64	defun parseReturn	64
3.2.65	defun parseReturn	64
3.2.66	defun parseSegment	65
3.2.67	defun parseSegment	65
3.2.68	defun parseSeq	65
3.2.69	defun parseSeq	65
3.2.70	defun parseVCONS	66
3.2.71	defun parseVCONS	66
3.2.72	defun parseWhere	66
3.2.73	defun parseWhere	66
4	Compile Transformers	67
4.1	Direct called comp routines	67
4.2	Indirect called comp routines	67
4.2.1	defun compAtSign	68
4.2.2	defun compAdd	69
4.2.3	defun compAtSign	70
4.2.4	defun compAtSign	71
4.2.5	defun compCapsule	71
4.2.6	defun compCapsule	71
4.2.7	defun compCapsuleInner	72
4.2.8	defun compCase	72
4.2.9	defun compCase	73
4.2.10	defun compCase1	74
4.2.11	defun compCat	74
4.2.12	defun compCat	75

4.2.13	defun compCat	75
4.2.14	defun compCat	75
4.2.15	defun compCategory	76
4.2.16	defun compCategory	76
4.2.17	defun compCoerce	77
4.2.18	defun compCoerce	77
4.2.19	defun compCoerce1	78
4.2.20	defun compColon	78
4.2.21	defun compColon	79
4.2.22	defun compCons	82
4.2.23	defun compCons	83
4.2.24	defun compCons1	84
4.2.25	defun compConstructorCategory	85
4.2.26	defun compConstructorCategory	85
4.2.27	defun compConstructorCategory	85
4.2.28	defun compConstructorCategory	85
4.2.29	defun compConstructorCategory	85
4.2.30	defun compConstruct	86
4.2.31	defun compConstruct	86
4.2.32	defun compDefine	87
4.2.33	defun compDefine	87
4.2.34	defun compDefine1	88
4.2.35	defun compElt	90
4.2.36	defun compElt	91
4.2.37	defun compExit	92
4.2.38	defun compExit	93
4.2.39	defun compHas	93
4.2.40	defun compHas	94
4.2.41	defun compIf	94
4.2.42	defun compIf	95
4.2.43	defun compImport	96
4.2.44	defun compImport	96
4.2.45	defun compIs	96
4.2.46	defun compIs	97
4.2.47	defun compJoin	97
4.2.48	defun compJoin	98
4.2.49	defun compLambda	99
4.2.50	defun compLambda	100
4.2.51	defun compLeave	101
4.2.52	defun compLeave	101
4.2.53	defun compMacro	101
4.2.54	defun compMacro	102
4.2.55	defun compPretend	103
4.2.56	defun compPretend	103
4.2.57	defun compQuote	104
4.2.58	defun compQuote	104

4.2.59	defun compReduce	104
4.2.60	defun compReduce	104
4.2.61	defun compReduce1	105
4.2.62	defun compSeq	107
4.2.63	defun compSeq	107
4.2.64	defun compSeq1	108
4.2.65	defun compSeqItem	108
4.2.66	defun compVector	109
4.2.67	defun compVector	109
4.2.68	defun compWhere	109
4.2.69	defun compWhere	110
5	Post Transformers	111
5.1	Direct called postparse routines	111
5.1.1	defun postTransform	111
5.1.2	defun postTran	113
5.1.3	defun postOp	114
5.1.4	defun postAtom	114
5.1.5	defun postTranList	114
5.1.6	defun postScriptsForm	115
5.1.7	defun postTranScripts	115
5.1.8	defun postTransformCheck	116
5.1.9	defun postcheck	116
5.1.10	defun postError	117
5.1.11	defun postForm	118
5.2	Indirect called postparse routines	120
5.2.1	defun postAdd	120
5.2.2	defun postAdd	121
5.2.3	defun postAtSign	121
5.2.4	defun postAtSign	121
5.2.5	defun postBigFloat	121
5.2.6	defun postBigFloat	122
5.2.7	defun postBlock	122
5.2.8	defun postBlock	122
5.2.9	defun postCategory	123
5.2.10	defun postCategory	123
5.2.11	defun postCollect,finish	124
5.2.12	defun postCollect	124
5.2.13	defun postCollect	125
5.2.14	defun postColon	125
5.2.15	defun postColon	126
5.2.16	defun postColonColon	126
5.2.17	defun postColonColon	126
5.2.18	defun postComma	126
5.2.19	defun postComma	127
5.2.20	defun comma2Tuple	127

5.2.21	defun postConstruct	127
5.2.22	defun postConstruct	128
5.2.23	defun postDef	129
5.2.24	defun postDef	130
5.2.25	defun postExit	131
5.2.26	defun postExit	131
5.2.27	defun postIf	132
5.2.28	defun postIf	132
5.2.29	defun postin	132
5.2.30	defun postin	133
5.2.31	defun postIn	133
5.2.32	defun postIn	133
5.2.33	defun postJoin	133
5.2.34	defun postJoin	134
5.2.35	defun postMapping	134
5.2.36	defun postMapping	134
5.2.37	defun postMDef	135
5.2.38	defun postMDef	136
5.2.39	defun postPretend	137
5.2.40	defun postPretend	137
5.2.41	defun postQUOTE	137
5.2.42	defun postQUOTE	137
5.2.43	defun postReduce	137
5.2.44	defun postReduce	138
5.2.45	defun postRepeat	138
5.2.46	defun postRepeat	138
5.2.47	defun postScripts	139
5.2.48	defun postScripts	139
5.2.49	defun postSemiColon	139
5.2.50	defun postSemiColon	139
5.2.51	defun postSignature	139
5.2.52	defun postSignature	140
5.2.53	defun postSlash	140
5.2.54	defun postSlash	140
5.2.55	defun postTuple	140
5.2.56	defun postTuple	141
5.2.57	defun postTupleCollect	141
5.2.58	defun postTupleCollect	141
5.2.59	defun postWhere	141
5.2.60	defun postWhere	142
5.2.61	defun postWith	142
5.2.62	defun postWith	142
5.3	Support routines	143
5.3.1	defun setDefOp	143
5.3.2	defun aplTran	143
5.3.3	defun aplTran1	144

5.3.4	defun aplTranList	145
5.3.5	defun hasAplExtension	146
5.3.6	defun deepestExpression	146
5.3.7	defun containsBang	147
5.3.8	defun getScriptName	147
5.3.9	defun decodeScripts	148
6	DEF forms	149
6.0.10	defun def	149
6.0.11	defun deftran	150
6.0.12	defun def-process	151
6.0.13	defun def-rename	151
6.0.14	defun def-rename1	152
6.0.15	defun def-insert-let	152
6.0.16	defun def-let	153
6.0.17	defun defLET	153
6.0.18	defun defLET1	154
6.0.19	defun defLET2	156
6.0.20	defun defLetForm	158
6.0.21	defvar \$defstack	158
6.0.22	defun def-where	158
6.0.23	defun def-whereclauselist	159
6.0.24	defun def-whereclause	159
6.0.25	defun def-message	159
6.0.26	defun def-message1	160
6.0.27	defun def-in2on	160
6.0.28	defun def-cond	161
6.0.29	defvar \$is-spill	161
6.0.30	defvar \$is-spill-list	161
6.0.31	defun def-is-eqlist	162
6.0.32	defvar \$vl	163
6.0.33	defun def-is-remdup	163
6.0.34	defun def-is-remdup1	164
6.0.35	defun addCARorCDR	165
6.0.36	IS	166
6.0.37	defun def-is	166
6.0.38	defvar \$is-eqlist	166
6.0.39	defun def-is2	167
6.0.40	defun defIS	168
6.0.41	defun defIS1	169
6.0.42	defun def-is-rev	171
6.0.43	defun defISReverse	172
6.0.44	defun def-collect	172
6.0.45	defun def-it	173
6.0.46	defun def-repeat	173
6.0.47	defun def-string	174

6.0.48	defun def-stringtoquote	174
6.0.49	defun def-addlet	174
6.0.50	defun def-inner	175
6.0.51	defun hackforis	175
6.0.52	defun hackforis1	175
6.0.53	defun unTuple	176
6.0.54	defun errhuh	176
7	PARSE forms	177
7.1	The original meta specification	177
7.2	The PARSE code	182
7.2.1	defvar \$tmptok	182
7.2.2	defvar \$tok	182
7.2.3	defvar \$ParseMode	182
7.2.4	defvar \$definition-name	183
7.2.5	defvar \$lablasoc	183
7.2.6	defun PARSE-NewExpr	183
7.2.7	defun PARSE-Command	184
7.2.8	defun PARSE-SpecialKeyWord	184
7.2.9	defun PARSE-SpecialCommand	185
7.2.10	defun PARSE-TokenCommandTail	186
7.2.11	defun PARSE-TokenOption	186
7.2.12	defun PARSE-TokenList	187
7.2.13	defun PARSE-CommandTail	187
7.2.14	defun PARSE-PrimaryOrQM	188
7.2.15	defun PARSE-Option	188
7.2.16	defun PARSE-Statement	189
7.2.17	defun PARSE-InfixWith	189
7.2.18	defun PARSE-With	190
7.2.19	defun PARSE-Category	191
7.2.20	defun PARSE-Expression	192
7.2.21	defun PARSE-Import	193
7.2.22	defun PARSE-Expr	193
7.2.23	defun PARSE-LedPart	194
7.2.24	defun PARSE-NudPart	194
7.2.25	defun PARSE-Operation	195
7.2.26	defun PARSE-leftBindingPowerOf	195
7.2.27	defun PARSE-rightBindingPowerOf	196
7.2.28	defun PARSE-getSemanticForm	196
7.2.29	defun PARSE-Prefix	197
7.2.30	defun PARSE-Infix	197
7.2.31	defun PARSE-TokTail	198
7.2.32	defun PARSE-Qualification	198
7.2.33	defun PARSE-Reduction	199
7.2.34	defun PARSE-ReductionOp	199
7.2.35	defun PARSE-Form	200

7.2.36	defun PARSE-Application	201
7.2.37	defun PARSE-Label	201
7.2.38	defun PARSE-Selector	202
7.2.39	defun PARSE-PrimaryNoFloat	203
7.2.40	defun PARSE-Primary	203
7.2.41	defun PARSE-Primary1	204
7.2.42	defun PARSE-Float	205
7.2.43	defun PARSE-FloatBase	206
7.2.44	defun PARSE-FloatBasePart	207
7.2.45	defun PARSE-FloatExponent	208
7.2.46	defun PARSE-Enclosure	209
7.2.47	defun PARSE-IntegerTok	209
7.2.48	defun PARSE-FormalParameter	209
7.2.49	defun PARSE-FormalParameterTok	210
7.2.50	defun PARSE-Quad	210
7.2.51	defun PARSE-String	210
7.2.52	defun PARSE-VarForm	211
7.2.53	defun PARSE-Scripts	211
7.2.54	defun PARSE-ScriptItem	212
7.2.55	defun PARSE-Name	212
7.2.56	defun PARSE-Data	213
7.2.57	defun PARSE-Sexpr	213
7.2.58	defun PARSE-Sexpr1	214
7.2.59	defun PARSE-NBGlyphTok	215
7.2.60	defun PARSE-GlyphTok	215
7.2.61	defun PARSE-AnyId	216
7.2.62	defun PARSE-Sequence	216
7.2.63	defun PARSE-Sequence1	217
7.2.64	defun PARSE-OpenBracket	217
7.2.65	defun PARSE-OpenBrace	218
7.2.66	defun PARSE-IteratorTail	218
7.2.67	defun PARSE-Iterator	219
7.2.68	The PARSE implicit routines	220
7.2.69	defun PARSE-Suffix	220
7.2.70	defun PARSE-SemiColon	221
7.2.71	defun PARSE-Return	221
7.2.72	defun PARSE-Exit	222
7.2.73	defun PARSE-Leave	222
7.2.74	defun PARSE-Seg	223
7.2.75	defun PARSE-Conditional	223
7.2.76	defun PARSE-ElseClause	224
7.2.77	defun PARSE-Loop	224
7.2.78	defun PARSE-LabelExpr	225
7.2.79	defun PARSE-FloatTok	225
7.3	The PARSE support routines	226
7.3.1	String grabbing	226

7.3.2	defun match-string	226
7.3.3	defun match-advance-string	227
7.3.4	defun initial-substring-p	228
7.3.5	defun quote-if-string	229
7.3.6	defun escape-keywords	230
7.3.7	defun underscore	230
7.3.8	Token Handling	230
7.3.9	defun getToken	230
7.3.10	defun unget-tokens	231
7.3.11	defun match-current-token	232
7.3.12	defun match-token	232
7.3.13	defun match-next-token	232
7.3.14	defun current-symbol	233
7.3.15	defun make-symbol-of	233
7.3.16	defun current-token	233
7.3.17	defun try-get-token	234
7.3.18	defun next-token	234
7.3.19	defun advance-token	235
7.3.20	defvar \$XTokenReader	235
7.3.21	defun get-token	235
7.3.22	Character handling	236
7.3.23	defun current-char	236
7.3.24	defun next-char	236
7.3.25	defun char-eq	236
7.3.26	defun char-ne	237
7.3.27	Error handling	237
7.3.28	defvar \$meta-error-handler	237
7.3.29	defun meta-syntax-error	237
7.3.30	Floating Point Support	238
7.3.31	defun floatexpid	238
7.3.32	Dollar Translation	238
7.3.33	defun dollarTran	238
7.3.34	Applying metagrammatical elements of a production (e.g., Star).	239
7.3.35	defmacro Bang	239
7.3.36	defmacro must	239
7.3.37	defun action	239
7.3.38	defun optional	240
7.3.39	defmacro star	240
7.3.40	Stacking and retrieving reductions of rules.	241
7.3.41	defun push-reduction	241

8 The Compiler	243
8.1 Compiling EQ.spad	243
8.1.1 The top level compiler command	246
8.1.2 The Spad compiler top level function	249
8.1.3 defun compilerDoit	253
8.1.4 defun /RQ,LIB	254
8.1.5 defun /rf-1	255
8.1.6 defun spad	264
8.1.7 defun Interpreter interface to the compiler	267
8.1.8 defun compTopLevel	270
8.1.9 defun compOrCroak	271
8.1.10 defun compOrCroak1	272
8.1.11 defun comp	273
8.1.12 defun compNoStacking	274
8.1.13 defun compNoStacking1	274
8.1.14 defun comp2	275
8.1.15 defun comp3	276
8.1.16 defun compTypeOf	278
8.1.17 defun compColonInside	279
8.1.18 defun compAtom	280
8.1.19 defun convert	281
8.1.20 defun primitiveType	282
8.1.21 defun compSymbol	283
8.1.22 defun compList	284
8.1.23 defun compExpression	285
8.1.24 defun compForm	285
8.1.25 defun compForm1	286
8.1.26 defun compForm2	289
8.1.27 defun compArgumentsAndTryAgain	291
8.1.28 defun compWithMappingMode	291
8.1.29 defun compWithMappingModel	292
8.1.30 defun extractCodeAndConstructTriple	300
8.1.31 defun hasFormalMapVariable	300
8.1.32 defun argsToSig	301
8.1.33 defun compMakeDeclaration	302
8.1.34 defun modifyModeStack	302
8.1.35 defun Create a list of unbound symbols	303
8.1.36 defun compOrCroak1,compactify	304
8.1.37 defun Compiler/Interpreter interface	304
8.1.38 defun compileSpadLispCmd	305
8.1.39 defun recompile-lib-file-if-necessary	306
8.1.40 defun spad-fixed-arg	306
8.1.41 defun compile-lib-file	307
8.1.42 defun compileFileQuietly	307
8.1.43 defvar \$byConstructors	307
8.1.44 defvar \$constructorsSeen	307

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 can contain 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

Compiler top level

1.1)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 interpreter 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 .

```

1.1.1 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. (see ?? on page ??). 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.

Chapter 2

The Parser

2.1 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
  1:S * eqn:$ == 1 * eqn.lhs = 1 * eqn.rhs
  1:S * eqn:$ == 1 * eqn.lhs = 1 * eqn.rhs
  eqn:$ * 1:S == eqn.lhs * 1 = eqn.rhs * 1
  -- We have to be a bit careful here: raising to a +ve integer is OK
  -- (since it's the equivalent of repeated multiplication)
  -- but other powers may cause contradictions
  -- Watch what else you add here! JHD 2/Aug 1990
if S has Monoid then
  1 == equation(1$S,1$S)
  recip eq ==
    (lh := recip lhs eq) case "failed" => "failed"
    (rh := recip rhs eq) case "failed" => "failed"
    [lh :: S, rh :: S]
  leftOne eq ==

```

```

      (re := recip lhs eq) case "failed" => "failed"
      1 = rhs eq * re
    rightOne eq ==
      (re := recip rhs eq) case "failed" => "failed"
      lhs eq * re = 1
  if S has Group then
    inv eq == [inv lhs eq, inv rhs eq]
    leftOne eq == 1 = rhs eq * inv rhs eq
    rightOne eq == lhs eq * inv rhs eq = 1
  if S has Ring then
    characteristic() == characteristic()$S
    i:Integer * eq:$ == (i::S) * eq
  if S has IntegralDomain then
    factorAndSplit eq ==
      (S has factor : S -> Factored S) =>
        eq0 := rightZero eq
        [equation(rcf.factor,0) for rcf in factors factor lhs eq0]
      (S has Polynomial Integer) =>
        eq0 := rightZero eq
        MF ==> MultivariateFactorize(Symbol, IndexedExponents Symbol, _
          Integer, Polynomial Integer)
        p : Polynomial Integer := (lhs eq0) pretend Polynomial Integer
        [equation((rcf.factor) pretend S,0) for rcf in factors factor(p)$MF]
        [eq]
  if S has PartialDifferentialRing(Symbol) then
    differentiate(eq:$, sym:Symbol):$ ==
      [differentiate(lhs eq, sym), differentiate(rhs eq, sym)]
  if S has Field then
    dimension() == 2 :: CardinalNumber
    eq1:$ / eq2:$ == eq1.lhs / eq2.lhs = eq1.rhs / eq2.rhs
    inv eq == [inv lhs eq, inv rhs eq]
  if S has ExpressionSpace then
    subst(eq1,eq2) ==
      eq3 := eq2 pretend Equation S
      [subst(lhs eq1,eq3),subst(rhs eq1,eq3)]

<initvars>≡
  (defvar $index 0 "File line number of most recently read line")

<initvars>+≡
  (defvar $linelist nil "Stack of parsed lines")

<initvars>+≡
  (defvar $echolinestack nil "Stack of lines to list")

```

```

<initvars>+≡
  (defvar $preparse-last-line nil "Most recently read line")

```

2.2 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")

```

2.2.1 defun initialize-preparse

```

[get-a-line p33]
[$index p??]
[$linelist p??]
[$echolinestack p??]
[$preparse-last-line p??]

<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)])))))

```

2.2.2 defun preparse

```

[preparse p18]
[preparse1 p23]
[parseprint p??]
[ifcar p??]
[$comblocklist p??]
[$skipme p??]
[$preparse-last-line p??]
[$index p??]
[$docList p??]
[$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 **prepare** 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)])))"))

```

2.2.3 defun Build the lines from the input for piles

```

[preparseReadLine p28]
[preparse-echo p30]
[fincomblock p??]
[parsepiles p26]
[doSystemCommand p??]
[escaped p??]
[instring p??]
[indent-pos p??]
[getfullstr p??]
[maxindex p??]
[strposl p??]
[is-console p??]
[spad-reader p??]
[$linelist p??]
[$echolinestack p??]
[$byConstructors p307]
[$skipme p??]
[$constructorsSeen p307]
[$preparse-last-line p??]

⟨defun preparse1⟩≡
  (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 ) 1))
(setq comsym (or (search "--" a :start2 i ) 1))
(setq ncomsym (or (search "++" a :start2 i ) 1))
(setq oparsym (or (position #\" a :start i ) 1))
(setq cparsym (or (position #\" a :start i ) 1))
(setq n (min strsym comsym ncomsym oparsym cparsym))
(cond
  ; nope, we found no comment, quoting, or grouping
  ((= n 1) (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 (getfullstr n " ") (substring a n ())) $linelist)
    (setq $index (1- $index))
    (setq a (subseq a 0 n)))
  (go NOCOMS))
; know how deep we are into parens
((= n oparsym) (setq pcount (1+ pcount)))
((= n cparsym) (setq pcount (1- pcount)))
(setq i (1+ n))
(go STRLOOP)
NOCOMS
; remember the indentation level
(setq sloc (indent-pos a))
(setq a (string-right-trim " " a))
(when (null sloc)
  (setq sloc psloc)
  (go READLOOP))
; handle line that ends in a continuation character
(cond
  ((eq (elt a (maxindex a)) xcaped)
    (setq continue t)
    (setq a (subseq a (maxindex a)))
    ((setq continue nil)))
  ; test for skipping constructors
  (when (and (null lines) (= sloc 0))
    (if (and |$byConstructors|
      (null (search "==" a))
      (not
        (member
          (setq functor
            (intern (substring a 0 (strpos1 ": (" 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)))

```

2.2.4 defun parsepiles

Add parens and semis to lines to aid parsing. [add-parens-and-semis-to-line p27]

```

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

```

2.2.5 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 p??]

[drop p??]

[addclose p??]

[nonblankloc p??]

<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)))) #\ ( )
                (setf slines (drop (1- i) slines))
                (rplaca slines (addclose (car slines) #\ ) ))))))))
```

2.2.6 defun preparseReadLine

```

[dcq p??]
[preparseReadLine1 p29]
[initial-substring p33]
[string2BootTree p??]
[storeblanks p33]
[skip-to-endif p??]
[preparseReadLine p28]

⟨defun preparseReadLine⟩≡
  (defun preparseReadLine (x)
    (let (line ind)
      (dcq (ind . line) (preparseReadLine1 x))
      (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)))

```

2.2.7 defun preparseReadLine1

```

[get-a-line p33]
[expand-tabs p??]
[maxindex p??]
[strconc p??]
[preparseReadLine1 p29]
[$linelist p??]
[$preparse-last-line p??]
[$index p??]
[$EchoLineStack p??]

⟨defun preparseReadLine1⟩≡
  (defun preparseReadLine1 (x)
    (let (line ind)
      (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)
          (setq line (string-right-trim " " line))
          (push (copy-seq line) $EchoLineStack)
          (cons $index
            (if (and (> (setq ind (maxindex line)) -1) (char= (elt line ind) #\_))
              (setq $preparse-last-line
                (strconc (substring line 0 ind) (cdr (preparseReadLine1 x))))
              line)))
          (cons $index line))))

```

2.3 I/O Handling

2.3.1 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 ())))
```

```
<initvars>+≡
  (defparameter Current-Fragment nil
    "A string containing remaining chars from readline; needed because
    Symbolics read-line returns embedded newlines in a c-m-Y.")
```

2.3.2 defun read-a-line

```

[subseq p??]
[Line-New-Line p32]
[read-a-line p31]
[*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)))))))

```

2.4 Line Handling

2.4.1 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.

2.4.2 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))

```

2.4.3 defun Line-New-Line

[\$Line p32]

```

<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)))))

```


2.4.4 defun next-line

```

⟨defun next-line⟩≡
  (defun next-line (&optional (in-stream t))
    (funcall Line-Handler in-stream))

```

2.4.5 defun storeblanks

```

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

```

2.4.6 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)))))

```

2.4.7 defun get-a-line

```

[is-console p??]
[mkprompt p??]
[read-a-line p31]
[make-string-adjustable p34]

```

```

⟨defun get-a-line⟩≡
  (defun get-a-line (stream)
    (when (is-console stream) (princ (mkprompt)))
    (let ((l1 (read-a-line stream)))
      (if (stringp l1)
          (make-string-adjustable l1)
          l1)))

```

2.4.8 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)))
```

Chapter 3

Parse Transformers

3.1 Direct called parse routines

3.1.1 defun parseTransform

```
[msubst p??]  
[parseTran p36]  
[$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)))
```

3.1.2 defun parseTran

```

[parseAtom p37]
[parseConstruct p38]
[parseTran p36]
[parseTranList p37]
[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))))))))))

```

3.1.3 defun parseAtom

```
[parseLeave p59]
[$NoValue p??]
```

```
<defun parseAtom>≡
  (defun |parseAtom| (x)
    (declare (special |$NoValue|))
    (if (eq x '|break|)
        (|parseLeave| (list '|$NoValue|))
        x))
```

3.1.4 defun parseTranList

```
[parseTran p36]
[parseTranList p37]
```

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

3.1.5 defun parseConstruct

```
<postvars>≡
  (eval-when (eval load)
    (setf (get '|construct| '|parseTran|) '|parseConstruct|))
```

3.1.6 defun parseConstruct

```
[parseTranList p37]  
[$insideConstructIfTrue p??]
```

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

3.2 Indirect called parse routines

In the `parseTran` function there is the code:

```
((and (atom u) (setq fn (get1 u '|parseTran|)))
  (funcall fn arg1))
```

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

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

```
VCONS      parseVCONS
where      parseWhere
```

3.2.1 defun parseAnd

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '|and| '|parseTran|) '|parseAnd|))
```

3.2.2 defun parseAnd

```
[parseTran p36]
[parseAnd p40]
[parseTranList p37]
[parseIf p53]
[$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| )))))
```

3.2.3 defun parseAtSign

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '@ '|parseTran|) '|parseAtSign|))
```


3.2.4 defun parseAtSign

```
[parseTran p36]
[parseType p??]
[$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)))))
```

3.2.5 defun parseCategory

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get 'category '|parseTran|) '|parseCategory|))
```

3.2.6 defun parseCategory

```
[parseTranList p37]
[parseDropAssertions p??]
[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)))))
```

3.2.7 defun parseCoerce

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '|::| '|parseTran|) '|parseCoerce|))
```

3.2.8 defun parseCoerce

```
[parseType p??]
[parseTran p36]
[$InteractiveMode p??]

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

3.2.9 defun parseColon

```
⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '|:|' '|parseTran|') '|parseColon|))
```

3.2.10 defun parseColon

```
[parseTran p36]
[parseType p??]
[$InteractiveMode p??]
[$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)))))
```

3.2.11 defun parseDEF

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get 'def ' |parseTran|) ' |parseDEF|))

```

3.2.12 defun parseDEF

```

[setDefOp p143]
[parseLhs p??]
[parseTranList p37]
[parseTranCheckForRecord p??]
[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|)))))

```

3.2.13 defun parseDollarGreaterthan

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|$>| ' |parseTran|) ' |parseDollarGreaterthan|))

```

3.2.14 defun parseDollarGreaterThan

```
[msubst p??]
[parseTran p36]
[$op p??]
```

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

3.2.15 defun parseDollarGreaterEqual

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

3.2.16 defun parseDollarGreaterEqual

```
[msubst p??]
[parseTran p36]
[$op p??]
```

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

3.2.17 defun parseDollarLessEqual

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

3.2.18 defun parseDollarLessEqual

```
[msubst p??]
[parseTran p36]
[$op p??]

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

3.2.19 defun parseDollarNotEqual

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

3.2.20 defun parseDollarNotEqual

```
[parseTran p36]
[msubst p??]
[$op p??]

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

3.2.21 defun parseEquivalence

```
⟨postvars⟩+=
  (eval-when (eval load)
    (setf (get '|eqv| '|parseTran|) '|parseEquivalence|))
```

3.2.22 defun parseEquivalence

[parseIf p53]

```

⟨defun parseEquivalence⟩≡
  (defun |parseEquivalence| (arg)
    (|parseIf|
     (list (first arg) (second arg)
           (|parseIf| (cons (second arg) '(|false| |true|))))))

```

3.2.23 defun parseExit

```

⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '|exit| '|parseTran|) '|parseExit|))

```

3.2.24 defun parseExit

[parseTran p36]

[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 ))))

```

3.2.25 defun parseGreaterEqual

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|>=| '|parseTran|) '|parseGreaterEqual|))

```

3.2.26 defun parseGreaterEqual

```

[parseTran p36]
[$op p??]

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

```

3.2.27 defun parseGreaterThan

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|>| '|parseTran|) '|parseGreaterThan|))

```

3.2.28 defun parseGreaterThan

```

[parseTran p36]
[$op p??]

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

```

3.2.29 defun parseHas

```
<postvars>+≡  
(eval-when (eval load)  
  (setf (get '|has| '|parseTran|) '|parseHas|))
```



```

⟨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))))))
```

3.2.30 defun parseIf,ifTran

```
[parseIf,ifTran p51]
[incExitLevel p??]
[makeSimplePredicateOrNil p??]
[incExitLevel p??]
[parseTran p36]
[$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))))))))
              (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 ))))

```

3.2.31 defun parseIf

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get 'if '|parseTran|) '|parseIf|))

```

3.2.32 defun parseIf

```

[parseIf,ifTran p51]
[parseTran p36]

```

```

<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)))))

```

3.2.33 defun parseImplies

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|implies| '|parseTran|) '|parseImplies|))

```

3.2.34 defun parseImplies

[parseIf p53]

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

3.2.35 defun parseIn

```
<postvars>+≡  
  (eval-when (eval load)  
    (setf (get 'in '|parseTran|) '|parseIn|))
```

3.2.36 defun parseIn

[parseTran p36]
 [postError p117]

```

<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))))))

```

3.2.37 defun parseInBy

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get 'inby '|parseTran|) '|parseInBy|))

```

3.2.38 defun parseInBy

```

[postError p117]
[parseTran p36]
[bright p??]
[parseIn p55]

<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))))))))))

```

3.2.39 defun parseIs

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|is| '|parseTran|) '|parseIs|))

```


3.2.40 defun parseIs

```
[parseTran p36]
[transIs p??]
```

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

3.2.41 defun parseIsnt

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '|isnt| '|parseTran|) '|parseIsnt|))
```

3.2.42 defun parseIsnt

```
[parseTran p36]
[transIs p??]
```

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

3.2.43 defun parseJoin

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '|Join| '|parseTran|) '|parseJoin|))
```

3.2.44 defun parseJoin

[parseTranList p37]

```

⟨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)))))

```

3.2.45 defun parseLeave

```

⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '|leave| '|parseTran|) '|parseLeave|))

```

3.2.46 defun parseLeave

[parseTran p36]

```

⟨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)))))

```

3.2.47 defun parseLessEqual

```

⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '|<=|' |parseTran|) '|parseLessEqual|))

```

3.2.48 defun parseLessEqual

[parseTran p36]

[\$op p??]

```

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

```

3.2.49 defun parseLET

```

⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '|let|' |parseTran|) '|parseLET|))

```

3.2.50 defun parseLET

```
[parseTran p36]
[parseTranCheckForRecord p??]
[opOf p??]
[transIs p??]

⟨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)))
```

3.2.51 defun parseLETD

```
⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get 'letd '|parseTran|) '|parseLETD|))
```

3.2.52 defun parseLETD

```
[parseTran p36]
[parseType p??]

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

3.2.53 defun parseMDEF

```
⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get 'mdef '|parseTran|) '|parseMDEF|))
```

3.2.54 defun parseMDEF

```

[parseTran p36]
[parseTranList p37]
[parseTranCheckForRecord p??]
[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|)))))

```

3.2.55 defun parseNot

```

⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '|not| '|parseTran|) '|parseNot|))

```

3.2.56 defun parseNot

```

⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '|^| '|parseTran|) '|parseNot|))

```

3.2.57 defun parseNot

```
[parseTran p36]
[$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|))))))
```

3.2.58 defun parseNotEqual

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '|^=' '|parseTran|) '|parseNotEqual|))
```

3.2.59 defun parseNotEqual

```
[parseTran p36]
[msubst p??]
[$op p??]
```

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

3.2.60 defun parseOr

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '|or| '|parseTran|) '|parseOr|))
```

3.2.61 defun parseOr

```

[parseTran p36]
[parseTranList p37]
[parseIf p53]
[parseOr p63]

⟨defun parseOr⟩≡
  (defun |parseOr| (arg)
    (let (x)
      (setq x (|parseTran| (car arg)))
      (cond
        (|$InteractiveModel| (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))))))))

```

3.2.62 defun parsePretend

```

⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '|pretend| '|parseTran|) '|parsePretend|))

```

3.2.63 defun parsePretend

```
[parseTran p36]
[parseType p??]

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

3.2.64 defun parseReturn

```
⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '|return| '|parseTran|) '|parseReturn|))
```

3.2.65 defun parseReturn

```
[parseTran p36]
[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)))))
```


3.2.66 defun parseSegment

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get 'segment '|parseTran|) '|parseSegment|))

```

3.2.67 defun parseSegment

```

[parseTran p36]

```

```

<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)))

```

3.2.68 defun parseSeq

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get 'seq '|parseTran|) '|parseSeq|))

```

3.2.69 defun parseSeq

```

[postError p117]
[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|)))))

```

3.2.70 defun parseVCONS

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get 'vcons '|parseTran|) '|parseVCONS|))

```

3.2.71 defun parseVCONS

```

[parseTranList p37]

```

```

<defun parseVCONS>≡
  (defun |parseVCONS| (arg)
    (cons 'vector (|parseTranList| arg)))

```

3.2.72 defun parseWhere

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|where| '|parseTran|) '|parseWhere|))

```

3.2.73 defun parseWhere

```

[mapInto p??]

```

```

<defun parseWhere>≡
  (defun |parseWhere| (arg)
    (cons '|where| (|mapInto| arg '|parseTran|)))

```

Chapter 4

Compile Transformers

4.1 Direct called comp routines

4.2 Indirect called comp routines

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

```
(if (and (atom (car x)) (setq fn (get1 (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|)

```

4.2.1 defun compAtSign

```

⟨postvars⟩ +=
  (eval-when (eval load)
    (setf (get '|add| 'special) '|compAdd|))

```



```

      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))))))

```

4.2.3 defun compAtSign

```

<postvars> +=
  (eval-when (eval load)
    (setf (get '|@| 'special) '|compAtSign|))

```

4.2.4 defun compAtSign

```
[addDomain p??]
[comp p273]
[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))))
```

4.2.5 defun compCapsule

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get 'capsule 'special) '|compCapsule|))
```

4.2.6 defun compCapsule

```
[bootStrapError p??]
[compCapsuleInner p72]
[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))))))
```

4.2.7 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))))

```

4.2.8 defun compCase

```

⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '|case| 'special) '|compCase|))

```


4.2.9 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 p74]

[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))))

4.2.10 defun compCase1

```
[comp p273]
[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)))))))
```

4.2.11 defun compCat

```
⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '|Record| 'special) '|compCat|))
```

4.2.12 defun compCat

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|Mapping| 'special) '|compCat|))

```

4.2.13 defun compCat

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|Union| 'special) '|compCat|))

```

4.2.14 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))))

```

4.2.15 defun compCategory

```

<postvars> +=
  (eval-when (eval load)
    (setf (get 'category 'special) '|compCategory|))

```

4.2.16 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")))))

```

4.2.17 defun compCoerce

```

⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '|::| 'special) '|compCoerce|))

```

4.2.18 defun compCoerce

```

[addDomain p??]
[getmode p??]
[compCoerce1 p78]
[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))))))

```

4.2.19 defun compCoerce1

```
[comp p273]
[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 |$String| |$EmptyModel|))
      (when (setq td (or (|comp| x mp e) (|comp| x |$EmptyModel| e)))
        (setq m1 (if (stringp (second td)) |$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 'progn
              (list 'let gg (first td))
              (cons '|check-subtype| (cons pred (list (mkq mp) gg))))))
            (list code mp (third td)))))))
```

4.2.20 defun compColon

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '|| 'special) '|compColon|))
```

4.2.21 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 p279]
[assoc p??]
[getDomainsInScope p??]
[isDomainForm p??]
[member p??]
[addDomain p??]
[isDomainForm p??]
[isCategoryForm p??]
[unknownTypeError p??]
[compColon p79]
[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 ))))))))

```

4.2.22 defun compCons

```

<postvars> +=
  (eval-when (eval load)
    (setf (get 'cons 'special) '|compCons|))

```

4.2.23 defun compCons

[compCons1 p84]
[compForm p285]

$\langle \text{defun compCons} \rangle \equiv$
 (defun |compCons| (form m e)
 (or (|compCons1| form m e) (|compForm| form m e)))

4.2.24 defun compCons1

```
[comp p273]
[convert p281]
[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))))))
                     (list (list 'cons x y) (list '|Pair| mx my) e ))))
                   (|convert| td m))))))
```

4.2.25 defun compConstructorCategory

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|ListCategory| 'special) '|compConstructorCategory|))

```

4.2.26 defun compConstructorCategory

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|RecordCategory| 'special) '|compConstructorCategory|))

```

4.2.27 defun compConstructorCategory

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|UnionCategory| 'special) '|compConstructorCategory|))

```

4.2.28 defun compConstructorCategory

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|VectorCategory| 'special) '|compConstructorCategory|))

```

4.2.29 defun compConstructorCategory

```

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

<defun compConstructorCategory>≡
  (defun |compConstructorCategory| (x m e)
    (declare (special |$Category|))
    (list x (|resolve| |$Category| m) e))

```

4.2.30 defun compConstruct

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|construct| 'special) '|compConstruct|))

```

4.2.31 defun compConstruct

```

[modeIsAggregateOf p??]
[compList p284]
[convert p281]
[compForm p285]
[compVector p109]
[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)))))))

```

4.2.32 defun compDefine

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get 'def 'special) '|compDefine|))

```

4.2.33 defun compDefine

```

[compDefine1 p88]
[$tripleCache p??]
[$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)))

```

4.2.34 defun compDefine1

```

[macroExpand p??]
[isMacro p??]
[getSignatureFromMode p??]
[compDefine1 p88]
[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|))))))

```

4.2.35 defun compElt

```
 $\langle postvars \rangle + \equiv$   
  (eval-when (eval load)  
    (setf (get '|elt| 'special) '|compElt|))
```

4.2.36 defun compElt

```
[compForm p285]
[isDomainForm p??]
[addDomain p??]
[getModemapListFromDomain p??]
[length p??]
[stackMessage p??]
[stackWarning p??]
[convert p281]
[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))))))

```

4.2.37 defun compExit

```

<postvars>+≡
(eval-when (eval load)
  (setf (get '|exit| 'special) '|compExit|))

```

4.2.38 defun compExit

```

[comp p273]
[modifyModeStack p302]
[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))))))))))

```

4.2.39 defun compHas

```

⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get ' |has| 'special) ' |compHas|))

```

4.2.40 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))))
```

4.2.41 defun compIf

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get 'if 'special) '|compIf|))
```

4.2.42 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))))))))))

```

4.2.43 defun compImport

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|import| 'special) '|compImport|))

```

4.2.44 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))

```

4.2.45 defun compIs

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|is| 'special) '|compIs|))

```


4.2.46 defun compIs

```
[comp p273]
[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))))))
```

4.2.47 defun compJoin

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '|Join| 'special) '|compJoin|))
```

4.2.48 `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 p281]
[$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 (arg1 catList p1 tmp2 tmp3 tmp4 tmp5 body parameters catListp td)
        (declare (special |$Category|))
        (setq arg1 (cdr arg))
        (setq catList
          (dolist (x arg1 (nreverse0 tmp3))
            (push (car (or (|compForMode| x |$Category| e) (return '|failed|)))
              tmp3)))
        (cond
          ((eq catList '|failed|)
            (|stackSemanticError| (list '|cannot form Join of: | arg1) 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))))))

```

4.2.49 defun compLambda

```

<postvars>+≡
(eval-when (eval load)
  (setf (get '|+>| 'special) '|compLambda|))

```

4.2.50 defun compLambda

```
[qcar p??]
[qcdr p??]
[argsToSig p301]
[compAtSign p71]
[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 ))))))
```

4.2.51 defun compLeave

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|leave| 'special) '|compLeave|))

```

4.2.52 defun compLeave

```

[comp p273]
[modifyModeStack p302]
[$exitModeStack p??]
[$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 )))))

```

4.2.53 defun compMacro

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get 'mdef 'special) '|compMacro|))

```

4.2.54 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))))))

```

4.2.55 defun compPretend

```

⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '|pretend| 'special) '|compPretend|))

```

4.2.56 defun compPretend

```

[addDomain p??]
[comp p273]
[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))))))

```

4.2.57 defun compQuote

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get 'quote 'special) '|compQuote|))

```

4.2.58 defun compQuote

```

<defun compQuote>≡
  (defun |compQuote| (expr m e)
    (list expr m e))

```

4.2.59 defun compReduce

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get 'reduce 'special) '|compReduce|))

```

4.2.60 defun compReduce

```

[compReduce1 p105]
[$formalArgList p??]

<defun compReduce>≡
  (defun |compReduce| (form m e)
    (declare (special |$formalArgList|))
    (|compReduce1| form m e |$formalArgList|))

```


4.2.61 defun compReduce1

```
[systemError p??]
[nreverse0 p??]
[compIterator p??]
[comp p273]
[parseTran p36]
[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 ))))))))))))

```

4.2.62 defun compSeq

```

⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get 'seq 'special) '|compSeq|))

```

4.2.63 defun compSeq

```

[compSeq1 p108]
[$exitModeStack p??]

⟨defun compSeq⟩≡
  (defun |compSeq| (arg0 m e)
    (declare (special |$exitModeStack|))
    (|compSeq1| (cdr arg0) (cons m |$exitModeStack|) e))

```

4.2.64 defun compSeq1

```

[nreverse0 p??]
[compSeqItem p108]
[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|))))

```

4.2.65 defun compSeqItem

```

[comp p273]
[macroExpand p??]

⟨defun compSeqItem⟩≡
  (defun |compSeqItem| (x m e)
    (|comp| (|macroExpand| x e) m e))

```

4.2.66 defun compVector

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get 'vector 'special) '|compVector|))

```

4.2.67 defun compVector

```

; null l => [$EmptyVector,m,e]
; Tl:= {\tt{}.,mUnder,e] := \ comp(x,mUnder,e)\ or\ return\ "failed"\ for\ x\ in\ l]\nwnewline
;\ \ Tl="failed"\ =>\ nil\nwnewline
;\ \ [{"VECTOR",:[T.expr\ for\ T\ in\ Tl},m,e]

```

```

[comp p273]
[$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))))))

```

4.2.68 defun compWhere

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|where| 'special) '|compWhere|))

```

4.2.69 defun compWhere

```
[comp p273]
[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 5

Post Transformers

5.1 Direct called postparse routines

5.1.1 defun postTransform

```
[postTran p113]  
[identp p??]  
[postTransformCheck p116]  
[aplTran p143]
```

```
<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))

```


5.1.2 defun postTran

```
[postAtom p114]
[postTran p113]
[paip p??]
[qcar p??]
[qcdr p??]
[unTuple p176]
[postTranList p114]
[postForm p118]
[postOp p114]
[postScriptsForm p115]
```

```
(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 (get1 op '|postTran|)))
              (funcall f x))
            ((and (paip op) (eq (qcar op) '|elt|))
              (progn
                (setq tmp1 (qcdr op))
                (and (paip tmp1)
                  (progn
                    (setq a (qcar tmp1))
                    (setq tmp2 (qcdr tmp1))
                    (and (paip tmp2)
                      (eq (qcdr tmp2) nil)
                      (progn (setq b (qcar tmp2)) t))))))
              (cons (|postTran| op) (cdr (|postTran| (cons b (cdr x))))))
            ((and (paip 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)))))))
```

5.1.3 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)))

```

5.1.4 defun postAtom

[\$boot p??]

```

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

```

5.1.5 defun postTranList

[postTran p113]

```

⟨defun postTranList⟩≡
  (defun |postTranList| (x)
    (loop for y in x collect (|postTran| y)))

```

5.1.6 defun postScriptsForm

```
[getScriptName p147]
[length p??]
[postTranScripts p115]
```

```
<defun postScriptsForm>≡
  (defun |postScriptsForm| (arg0 arg1)
    (let ((op (second arg0)) (a (third arg0)))
      (cons (|getScriptName| op a (|#| arg1))
            (append (|postTranScripts| a) arg1))))
```

5.1.7 defun postTranScripts

```
[postTranScripts p115]
[postTran p113]
```

```
<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)))))))
```

5.1.8 defun postTransformCheck

```
[postcheck p116]
[$defOp p??]
```

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

5.1.9 defun postcheck

```
[setDefOp p143]
[postcheck p116]
```

```
<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)))))
```

5.1.10 defun postError

```
[nequal p??]  
[bumperrorcount p??]  
[$defOp p??]  
[$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))
```

5.1.11 defun postForm

```

[postTranList p114]
[internal p??]
[postTran p113]
[postError p117]
[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)))
                (internal '* (stringimage 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!" '|%1|
                      " Did you misuse infix dot?"))))))
              (t
               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))))
```

5.2 Indirect called postparse routines

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

```
((and (atom op) (setq f (get1 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	

5.2.1 defun postAdd

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


5.2.2 defun postAdd

```
[postTran p113]
[postCapsule p??]
```

```
<defun postAdd>≡
  (defun |postAdd| (arg)
    (if (null (cddr arg))
        (|postCapsule| (second arg))
        (list '|add| (|postTran| (second arg)) (|postCapsule| (third arg)))))
```

5.2.3 defun postAtSign

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '|@| '|postTran|) '|postAtSign|))
```

5.2.4 defun postAtSign

```
[postTran p113]
[postType p??]
```

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

5.2.5 defun postBigFloat

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '|:BF:| '|postTran|) '|postBigFloat|))
```

5.2.6 defun postBigFloat

```
[postTran p113]
[$boot p??]
[$InteractiveMode p??]

⟨defun postBigFloat⟩≡
  (defun |postBigFloat| (arg)
    (let (mant expon eltword)
      (declare (special $boot |$InteractiveMode|))
      (setq mant (second arg))
      (setq expon (caddr 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)))))))
```

5.2.7 defun postBlock

```
⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '|Block| '|postTran|) '|postBlock|))
```

5.2.8 defun postBlock

```
[postBlockItemList p??]
[postTran p113]

⟨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)))))))
```

5.2.9 defun postCategory

```

(postvars) +=
  (eval-when (eval load)
    (setf (get 'category '|postTran|) '|postCategory|))

```

5.2.10 defun postCategory

```

[postTran p113]
[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))))))))

```

5.2.11 defun postCollect,finish

```

<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)))))))

```

5.2.12 defun postCollect

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get 'collect '|postTran|) '|postCollect|))

```

5.2.13 defun postCollect

```
[postCollect,finish p124]
[postCollect p125]
[postIteratorList p??]
[postTran p113]
```

```
<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))))))
```

5.2.14 defun postColon

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '|:| '|postTran|) '|postColon|))
```

5.2.15 defun postColon

```
[postTran p113]
[postType p??]
```

```
<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)))))))
```

5.2.16 defun postColonColon

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '|::| '|postTran|) '|postColonColon|))
```

5.2.17 defun postColonColon

```
[stringimage p??]
[postForm p118]
[$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 (stringimage (third u)) (second u))
      (|postForm| u)))
```

5.2.18 defun postComma

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '|,| '|postTran|) '|postComma|))
```

5.2.19 defun postComma

```
[postTuple p141]
[comma2Tuple p127]
```

```
<defun postComma>≡
  (defun |postComma| (u)
    (|postTuple| (|comma2Tuple| u)))
```

5.2.20 defun comma2Tuple

```
[postFlatten p??]
```

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

5.2.21 defun postConstruct

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '|construct| '|postTran|) '|postConstruct|))
```

5.2.22 defun postConstruct

```

[comma2Tuple p127]
[postTranSegment p??]
[postMakeCons p??]
[tuple2List p??]
[postTranList p114]
[postTran p113]

⟨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))))

```


5.2.23 defun postDef

```
<postvars>+≡  
(eval-when (eval load)  
  (setf (get '|==| 'postTran|) '|postDef|))
```

5.2.24 defun postDef

```
[postMDef p136]
[recordHeaderDocumentation p??]
[nequal p??]
[postTran p113]
[postDefArgs p??]
[nreverse0 p??]
[$boot p??]
[$maxSignatureLineNumber p??]
[$headerDocumentation p??]
[$docList p??]
[$InteractiveMode p??]

⟨defun postDef⟩≡
  (defun |postDef| (arg)
    (let (defOp rhs lhs targetType tmp1 op arg1 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 arg1 (cdr tmp1))
  (cons op (|postDefArgs| arg1))))
(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))))))

```

5.2.25 defun postExit

```

<postvars> +=
  (eval-when (eval load)
    (setf (get '|=>| '|postTran|) '|postExit|))

```

5.2.26 defun postExit

[postTran p113]

```

<defun postExit> ≡
  (defun |postExit| (arg)
    (list 'if (|postTran| (second arg))
          (list '|exit| (|postTran| (third arg)))
          '|noBranch|))

```

5.2.27 defun postIf

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|if| '|postTran|) '|postIf|))

```

5.2.28 defun postIf

```

[nreverse0 p??]
[postTran p113]
[$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)))))))

```

5.2.29 defun postin

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|in| '|postTran|) '|postin|))

```

5.2.30 defun postin

```
[systemErrorHere p??]
[postTran p113]
[postInSeq p??]
```

```
<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)))))
```

5.2.31 defun postIn

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get 'in '|postTran|) '|postIn|))
```

5.2.32 defun postIn

```
[systemErrorHere p??]
[postTran p113]
[postInSeq p??]
```

```
<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)))))
```

5.2.33 defun postJoin

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '|Join| '|postTran|) '|postJoin|))
```

5.2.34 defun postJoin

```
[postTran p113]
[postTranList p114]
```

```
<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))))
```

5.2.35 defun postMapping

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '|->' '|postTran|) '|postMapping|))
```

5.2.36 defun postMapping

```
[postTran p113]
[unTuple p176]
```

```
<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)))))))
```

5.2.37 defun postMDef

```
<postvars>+≡  
(eval-when (eval load)  
  (setf (get '|==>' '|postTran|) '|postMDef|))
```

5.2.38 defun postMDef

```
[postTran p113]
[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))))))
```


5.2.39 defun postPretend

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|pretend| '|postTran|) '|postPretend|))

```

5.2.40 defun postPretend

```

[postTran p113]
[postType p??]

<defun postPretend>≡
  (defun |postPretend| (arg)
    (cons '|pretend| (cons (|postTran| (second arg)) (|postType| (third arg))))))

```

5.2.41 defun postQUOTE

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|quote| '|postTran|) '|postQUOTE|))

```

5.2.42 defun postQUOTE

```

<defun postQUOTE>≡
  (defun |postQUOTE| (arg) arg)

```

5.2.43 defun postReduce

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|Reduce| '|postTran|) '|postReduce|))

```

5.2.44 defun postReduce

```
[postTran p113]
[postReduce p138]
[$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))))))))))
```

5.2.45 defun postRepeat

```
⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get 'repeat '|postTran|) '|postRepeat|))
```

5.2.46 defun postRepeat

```
[postIteratorList p??]
[postTran p113]

⟨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))))))
```

5.2.47 defun postScripts

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|Scripts| '|postTran|) '|postScripts|))

```

5.2.48 defun postScripts

```

[getScriptName p147]
[postTranScripts p115]

<defun postScripts>≡
  (defun |postScripts| (arg)
    (cons (|getScriptName| (second arg) (third arg) 0)
          (|postTranScripts| (third arg))))

```

5.2.49 defun postSemiColon

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|;| '|postTran|) '|postSemiColon|))

```

5.2.50 defun postSemiColon

```

[postBlock p122]
[postFlattenLeft p??]

<defun postSemiColon>≡
  (defun |postSemiColon| (u)
    (|postBlock| (cons '|Block| (|postFlattenLeft| u '|;|))))

```

5.2.51 defun postSignature

```

<postvars>+≡
  (eval-when (eval load)
    (setf (get '|Signature| '|postTran|) '|postSignature|))

```

5.2.52 defun postSignature

```

[pairst p??]
[postType p??]
[removeSuperfluousMapping p??]
[killColons p??]

⟨defun postSignature⟩≡
  (defun |postSignature| (arg)
    (let (sig sig1 op)
      (setq op (second arg))
      (setq sig (third arg))
      (when (and (pairst 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)))))))

```

5.2.53 defun postSlash

```

⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '/' '|postTran|) '|postSlash|))

```

5.2.54 defun postSlash

```

[postTran p113]

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

```

5.2.55 defun postTuple

```

⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '@Tuple' '|postTran|) '|postTuple|))

```

5.2.56 defun postTuple

[postTranList p114]

```

⟨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))))))

```

5.2.57 defun postTupleCollect

```

⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '|TupleCollect| '|postTran|) '|postTupleCollect|))

```

5.2.58 defun postTupleCollect

[postCollect p125]

```

⟨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)))))))

```

5.2.59 defun postWhere

```

⟨postvars⟩+≡
  (eval-when (eval load)
    (setf (get '|where| '|postTran|) '|postWhere|))

```

5.2.60 defun postWhere

```
[postTran p113]
[postTranList p114]
```

```
<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)))))
```

5.2.61 defun postWith

```
<postvars>+≡
  (eval-when (eval load)
    (setf (get '|with| '|postTran|) '|postWith|))
```

5.2.62 defun postWith

```
[postTran p113]
[$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))))
```

5.3 Support routines

5.3.1 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)))))
```

5.3.2 defun aplTran

```
[aplTran1 p144]
[containsBang p147]
[$genno p??]
[$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)))))))
```

5.3.3 defun aplTran1

```

[aplTranList p145]
[aplTran1 p144]
[hasAplExtension p146]
[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| arg1))
    (and (pairp tmp1)
      (progn
        (setq arg1Assoc (qcar tmp1))
        (setq futureArg1 (qcdr tmp1))
        t)))
    (cons '|reshape|
      (cons
        (cons 'collect
          (append
            (do ((tmp3 arg1Assoc (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 futureArg1))))))
        (list (cdar arg1Assoc))))
    (t (cons op arg1))))))

```

5.3.4 defun aplTranList

[aplTran1 p144]

[aplTranList p145]

```

⟨defun aplTranList⟩≡
  (defun |aplTranList| (x)
    (if (atom x)
      x
      (cons (|aplTran1| (car x)) (|aplTranList| (cdr x)))))

```

5.3.5 defun hasAplExtension

```

[nreverse0 p??]
[deepestExpression p146]
[genvar p??]
[aplTran1 p144]
[msubst p??]

⟨defun hasAplExtension⟩≡
  (defun |hasAplExtension| (arg1)
    (let (tmp2 tmp3 y z g arg1Assoc u)
      (when
        (dolist (x arg1 tmp2)
          (setq tmp2 (or tmp2 (and (pairp x) (eq (qcar x) '!))))))
      (setq u
        (dolist (x arg1 (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 arg1Assoc
                  (cons (cons (setq g (genvar)) (|aplTran1| z)) arg1Assoc))
                (msubst g z y))
              x)
            tmp3)))
      (cons arg1Assoc u))))

```

5.3.6 defun deepestExpression

```

[deepestExpression p146]

⟨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))

```

5.3.7 defun containsBang

[containsBang p147]

```

⟨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)))))))

```

5.3.8 defun getScriptName

```

[identp p??]
[postError p117]
[internl p??]
[stringimage p??]
[decodeScripts p148]
[pname p??]

```

```

⟨defun getScriptName⟩≡
  (defun |getScriptName| (op a numberOfFunctionalArgs)
    (when (null (identp op))
      (|postError| (list " " op " cannot have scripts" )))
    (internl '* (stringimage numberOfFunctionalArgs)
              (|decodeScripts| a) (pname op)))

```

5.3.9 defun decodeScripts

```
[qcar p??]
[qcdr p??]
[strconc p??]
[stringimage p??]
[decodeScripts p148]
```

```
<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 (stringimage 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) '|,|))
         (stringimage (fn a)))
        (t
         (stringimage 1))))))
```

Chapter 6

DEF forms

6.0.10 defun def

```
[deftran p150]  
[def-insert-let p152]  
[def-stringtoquote p174]  
[bootTransform p??]  
[comp p273]  
[sublis p??]  
[$body p??]  
[$opassoc p??]  
[$op p??]
```

```
<defun def>≡  
(defun def (form signature $body)  
  (declare (ignore signature))  
  (let* ($opassoc  
        ($op (first form))  
        (argl (rest form))  
        ($body (deftran $body))  
        (argl (def-insert-let argl))  
        (arglp (def-stringtoquote argl))  
        ($body (|bootTransform| $body)))  
    (declare (special $body $opassoc $op))  
    (comp (sublis $opassoc (list (list $op (list 'lam arglp $body)))))))
```

6.0.11 defun deftran

This two-level call allows DEF-RENAME to be locally bound to do nothing (see boot2Lisp) yet still allow function call (lisp2BootAndComp). [p??]
 [\$macroassoc p??]

```

⟨defun deftran⟩≡
  (defun deftran (x)
    (let (op y)
      (cond
        ((stringp x) (def-string x))
        ((identp x) (cond ((lassoc x $macroassoc)) (x)))
        ((atom x) x)
        ((eq (setq op (first x)) 'where) (def-where (cdr x)))
        ((eq op 'repeat) (def-repeat (cdr x)))
        ((eq op 'collect) (def-collect (cdr x)))
        ((eq op 'makestring)
         (cond ((stringp (second x)) x)
               ((eqcar (second x) 'quote)
                (list 'makestring (stringimage (cadadr x))))
               ((list 'makestring (deftran (second x))))))
        ((eq op 'quote)
         (if (stringp (setq y (second x))) (list 'makestring y)
             (if (and (identp y) (char= (elt (pname y) 0) #\.)
                 '(intern ,(pname y) ,(package-name *package*) x)))
             ((eq op 'is) (|defIS| (second x) (third x)))
             ((eq op 'spadlet) (def-let (second x) (third x)))
             ((eq op 'dcq) (list 'dcq (second x) (deftran (third x))))
             ((eq op 'cond) (cons 'cond (def-cond (cdr x))))
             ((member (first x) '(|sayBrightly| say moan croak) :test #'eq)
              (def-message x))
             ((setq y (getl (first x) 'def-tran))
              (funcall y (mapcar #'deftran (cdr x))))
             ((mapcar #'deftran x))))))

```

6.0.12 defun def-process

```
[def p149]
[b-mdef p??]
[eqcar p??]
[def-process p151]
[is-console p??]
[say p??]
[deftran p150]
[print-full p??]
[deftran p150]
[$macroassoc p??]
```

```
<defun def-process>≡
  (defun def-process (x &aux $macroassoc)
    (cond
      ((eqcar x 'def)
       (def (second x) (third x) (first (cddddr x))))
      ((eqcar x 'mdef)
       (b-mdef (second x) (third x) (first (cddddr x))))
      ((and (eqcar x 'where) (eqcar (second x) 'def))
       (let* ((u (second x)) (y (cdr u)))
         (def-process
          (list 'def
                (car y)
                (car (setq y (cdr y)))
                (car (setq y (cdr y)))
                (cons 'where (cons (car (setq y (cdr y))) (cddr x)))))))
      ((is-console *standard-output*)
       (say "  VALUE = " (eval (deftran x))))
      ((print-full (deftran x)))))
```

6.0.13 defun def-rename

```
[def-rename1 p152]
```

```
<defun def-rename>≡
  (defun def-rename (x)
    (def-rename1 x))
```

6.0.14 defun def-rename1

[def-rename1 p152]

```

⟨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)))

```

6.0.15 defun def-insert-let

[def-insert-let p152]

[deftran p150]

[def-let p153]

[errhuh p176]

```

⟨defun def-insert-let⟩≡
  (defun def-insert-let (x)
    (labels (
      (insert-let1 (y)
        (declare (special $body))
        (if (and (consp y) (eq (qcar y) 'spadlet))
          (cond
            ((identp (second y))
              (setq $body (cons 'progn (list (def-let (third y) (second y)) $body)))
              (setq y (second y)))
            ((identp (third y))
              (setq $body (cons 'progn (list (deftran y) $body))) (setq y (third y)))
            ((errhuh)))
          y)))
      (if (atom x)
        x
        (cons (insert-let1 (first x)) (def-insert-let (cdr x))))))

```


6.0.16 defun def-let

[deftran p150]
[defLET p153]

```

⟨defun def-let⟩≡
  (defun def-let (form rhs)
    (let (f1 f2)
      (unless (and (consp form) (eq (qcar form) '\:))
        (setq form (macroexpand form)))
      (cond
        ((and (consp form) (eq (qcar form) '\:))
         (setq f1 (deftran form))
         (setq f2 (deftran (list 'spadlet (second form) rhs))))
        (if (and (eq (car f2) 'spadlet) (equal (second f2) (second form)))
            (list 'spadlet (second f1) (third f2))
            (list 'progn f1 f2)))
      ((and (consp form) (eq (qcar form) 'elt))
       (deftran (list 'setelt (second form) (third form) rhs)))
      (t
       (|defLET| form (deftran rhs))))))

```

6.0.17 defun defLET

[defLET1 p154]
[\$letGenVarCounter p??]
[\$inDefLET p??]

```

⟨defun defLET⟩≡
  (defun |defLET| (lhs rhs)
    (let (|$letGenVarCounter| |$inDefLET|)
      (declare (special |$letGenVarCounter| |$inDefLET|))
      (setq |$letGenVarCounter| 1)
      (setq |$inDefLET| t)
      (|defLET1| lhs rhs)))

```

6.0.18 defun defLET1

```
[identp p??]
[defLetForm p158]
[contained p??]
[defLET2 p156]
[mkprogn p??]
[defLET1 p154]
[strconc p??]
[stringimage p??]
[$let p59]
[$letGenVarCounter p??]
```

```
(defun defLET1)≡
  (defun |defLET1| (lhs rhs)
    (let (name l1 l2 g rhsprime letprime)
      (declare (special $let |$letGenVarCounter|))
      (cond
        ((identp lhs) (|defLetForm| lhs rhs))
        ((and (pairp lhs) (eq (qcar lhs) 'fluid)
              (pairp (qcdr lhs)) (eq (qcdr (qcdr lhs)) nil))
         (|defLetForm| lhs rhs))
        ((and (identp rhs) (null (contained rhs lhs)))
         (setq rhsprime (|defLET2| lhs rhs))
         (cond
            ((and (consp rhsprime) (eql (qcar rhsprime) $let))
             (mkprogn (list rhsprime rhs)))
            ((and (consp rhsprime) (eq (qcar rhsprime) 'progn))
             (append rhsprime (list rhs)))
            (t
             (when (identp (car rhsprime)) (setq rhsprime (list rhsprime)))
             (mkprogn (append rhsprime (list rhs))))))
        ((and (pairp rhs) (eqcar rhs $let) (identp (setq name (cadr rhs))))
         (setq l1 (|defLET1| name (third rhs)))
         (setq l2 (|defLET1| lhs name))
         (if (and (consp l2) (eq (qcar l2) 'progn))
             (mkprogn (cons l1 (cdr l2)))
             (progn
              (when (identp (car l2)) (setq l2 (list l2)))
              (mkprogn (cons l1 (append l2 (list name)))))))
        (t
         (setq g (intern (strconc "LETTMP#" (stringimage |$letGenVarCounter|))))
         (setq |$letGenVarCounter| (1+ |$letGenVarCounter|))
         (setq rhsprime (list $let g rhs))
         (setq letprime (|defLET1| lhs g))
```

```
(if (and (consp letprime) (eq (qcar letprime) 'progn))
    (mkprogn (cons rhsprime (cdr letprime)))
    (progn
      (when (identp (car letprime)) (setq letprime (list letprime)))
      (mkprogn (cons rhsprime (append letprime (list g)))))))))
```

6.0.19 defun defLET2

```
[identp p??]
[defLetForm p158]
[qcar p??]
[qcdr p??]
[defLET2 p156]
[addCARorCDR p165]
[defISReverse p172]
[strconc p??]
[stringimage p??]
[defIS1 p169]
[defIS p168]
[$inDefIS p??]
[$let p59]
[$letGenVarCounter p??]
```

```
<defun defLET2>≡
  (defun |defLET2| (lhs rhs)
    (let (a b l1 var2 patrev rev g l2 val1 var1 isPred)
      (declare (special |$inDefIS| $let |$letGenVarCounter|))
      (cond
        ((identp lhs) (|defLetForm| lhs rhs))
        ((null lhs) nil)
        ((and (pairp lhs) (eq (qcar lhs) 'fluid)
              (pairp (qcdr lhs)) (eq (qcdr (qcdr lhs)) nil))
         (|defLetForm| lhs rhs))
        ((and (pairp lhs) (equal (qcar lhs) $let)
              (pairp (qcdr lhs)) (pairp (qcdr (qcdr lhs)))
              (eq (qcdr (qcdr (qcdr lhs))) nil))
         (setq a (|defLET2| (qcar (qcdr lhs)) rhs))
         (setq b (qcar (qcdr (qcdr lhs))))
         (cond
           ((null (setq b (|defLET2| b rhs))) a)
           ((atom b) (list a b))
           ((pairp (qcar b)) (cons a b))
           (t (list a b))))
        ((and (pairp lhs) (eq (qcar lhs) 'cons)
              (pairp (qcdr lhs)) (pairp (qcdr (qcdr lhs)))
              (eq (qcdr (qcdr (qcdr lhs))) nil))
         (setq var1 (qcar (qcdr lhs)))
         (setq var2 (qcar (qcdr (qcdr lhs))))
         (if (or (eq var1 (intern "." "BOOT"))
                 (and (pairp var1) (eqcar var1 'quote)))
             (|defLET2| var2 (|addCARorCDR| 'cdr rhs))
```

```

(progn
  (setq l1 (|defLET2| var1 (|addCARorCDR| 'car rhs)))
  (if (member var2 '(nil |.|))
      l1
      (progn
        (when (and (pairp l1) (atom (car l1))) (setq l1 (cons l1 nil)))
        (if (identp var2)
            (append l1 (cons (|defLetForm| var2 (|addCARorCDR| 'cdr rhs)) nil))
            (progn
              (setq l2 (|defLET2| var2 (|addCARorCDR| 'cdr rhs)))
              (when (and (pairp l2) (atom (car l2))) (setq l2 (cons l2 nil)))
              (append l1 l2))))))
  ((and (pairp lhs) (eq (qcar lhs) 'append)
        (pairp (qcdr lhs)) (pairp (qcdr (qcdr lhs)))
        (eq (qcdr (qcdr (qcdr lhs))) nil))
   (setq var1 (qcar (qcdr lhs)))
   (setq var2 (qcar (qcdr (qcdr lhs))))
   (setq patrev (|defISReverse| var2 var1))
   (setq rev (list 'reverse rhs))
   (setq g (intern (strconc "LETMP#" (stringimage |$letGenVarCounter|))))
   (setq |$letGenVarCounter| (1+ |$letGenVarCounter|))
   (setq l2 (|defLET2| patrev g))
   (when (and (pairp l2) (atom (car l2))) (setq l2 (cons l2 nil)))
   (cond
    ((eq var1 (intern "." "BOOT"))
     (cons (list $LET g rev) l2))
    ((and (pairp (|last| l2)) (equal (qcar (|last| l2)) $let)
          (pairp (qcdr (|last| l2)))
          (equal (qcar (qcdr (|last| l2))) var1)
          (pairp (qcdr (qcdr (|last| l2))))
          (eq (qcdr (qcdr (qcdr (|last| l2)))) nil))
     (setq val1 (qcar (qcdr (qcdr (|last| l2)))))
     (cons
      (list $let g rev)
      (append
       (reverse (cdr (reverse l2)))
       (list (|defLetForm| var1 (list 'nreverse val1))))))
    (t
     (cons
      (list $let g rev)
      (append l2 (list (|defLetForm| var1 (list 'nreverse var1)))))))
  ((and (pairp lhs) (eq (qcar lhs) 'equal)
        (pairp (qcdr lhs)) (eq (qcdr (qcdr lhs)) nil))
   (setq var1 (qcar (qcdr lhs)))
   (list 'cond (list (list 'equal var1 rhs) var1)))
  (t

```

```
(setq isPred
  (if |$inDefIS|
    (|defIS1| rhs lhs)
    (|defIS| rhs lhs)))
(list 'cond (list isPred rhs))))))
```

6.0.20 defun defLetForm

[*\$let* p59]

```
<defun defLetForm>≡
  (defun |defLetForm| (lhs rhs)
    (declare (special $let))
    (list $let lhs rhs))
```

6.0.21 defvar \$defstack

```
<initvars>+≡
  (defparameter $defstack nil)
```

6.0.22 defun def-where

[def-whereclauselist p159]
 [def-inner p175]
 [sublis p??]
 [mkprogn p??]
 [deftran p150]
 [\$defstack p158]
 [\$opassoc p??]

```
<defun def-where>≡
  (defun def-where (args)
    (let ((x (car args)) (y (cdr args)) $defstack)
      (declare (special $defstack $opassoc))
      (let ((u (def-whereclauselist y)))
        (mapc #'(lambda (X) (def-inner (first x) nil (sublis $opassoc (second x))))
              $defstack)
        (mkprogn (nconc u (list (deftran x)))))))
```

6.0.23 defun def-whereclauselist

[def-whereclause p159]
[deftran p150]

```
⟨defun def-whereclauselist⟩≡
  (defun def-whereclauselist (l)
    (if (not (cdr l))
        (def-whereclause (deftran (first l)))
        (reduce #'append (mapcar #'(lambda (u) (def-whereclause (deftran u))) l))))
```

6.0.24 defun def-whereclause

[eqcar p??]
[def-whereclause p159]
[whdef p??]

```
⟨defun def-whereclause⟩≡
  (defun def-whereclause (x)
    (cond
      ((or (eqcar x 'seq) (eqcar x 'progn))
       (reduce #'append (mapcar #'def-whereclause (cdr x))))
      ((eqcar x 'def)
       (whdef (second x) (first (cddddr x))) nil)
      ((and (eqcar x '|exit|) (eqcar (second x) 'def))
       (whdef (cadadr x) (first (cddddr (second x))) ) nil)
      ((list x))))
```

6.0.25 defun def-message

[def-message1 p160]

```
⟨defun def-message⟩≡
  (defun def-message (u)
    (cons (first u) (mapcar #'def-message1 (cdr u))))
```

6.0.26 defun def-message1

```
[eqcar p??]
[def-message1 p160]
[deftran p150]

⟨defun def-message1⟩≡
  (defun def-message1 (v)
    (cond
      ((and (stringp v) (> (size v) 0) (not (eq (elt v 0) '\%)))
        (list 'makestring v))
      ((eqcar v 'cons)
        (list 'cons (def-message1 (second v)) (def-message1 (third v))))
      ((deftran v))))
```

6.0.27 defun def-in2on

```
[eqcar p??]

⟨defun def-in2on⟩≡
  (defun def-in2on (it)
    (mapcar
      #'(lambda (x) (let (u)
        (cond
          ((and (eqcar x 'in) (eqcar (third x) '|tails|))
            (list 'on (second x) (second (third x))))
          ((and (eqcar x 'in) (eqcar (setq u (third x)) 'segment))
            (cond
              ((third u) (list 'step (second x) (second u) 1 (third u)))
              ((list 'step (second x) (second u) 1))))
          ((and (eqcar x 'inby) (eqcar (setq u (third x)) 'segment))
            (cond
              ((third u) (list 'step (second x) (second u) (|last| x) (third u)))
              ((list 'step (second x) (second u) (|last| x))))
          (x))))
      it))
```


6.0.28 defun def-cond

[deftran p150]
[def-cond p161]

```
⟨defun def-cond⟩≡
  (defun def-cond (l)
    (cond
      ((not l) nil)
      ((cons (mapcar #'deftran (first l)) (def-cond (cdr l)))))))
```

6.0.29 defvar \$is-spill

```
⟨initvars⟩+≡
  (defvar $is-spill nil)
```

6.0.30 defvar \$is-spill-list

```
⟨initvars⟩+≡
  (defvar $is-spill-list nil)
```

6.0.31 defun def-is-eqlist

```

[ p??]
[$is-eqlist p166]
[$is-spill-list p161]

⟨defun def-is-eqlist⟩≡
  (defun def-is-eqlist (str)
    (let (g e)
      (declare (special $is-eqlist $is-spill-list))
      (cond
        ((not str) (push '(eq ,(setq g (is-gensym)) nil) $is-eqlist) g)
        ((eq str '\.) (is-gensym))
        ((identp str) str)
        ((stringp str)
         (setq e (def-string str))
         (push (list (if (atom (second e)) 'eq 'equal)
                     (setq g (is-gensym)) e)
                $is-eqlist)
         g)
        ((or (numberp str) (member str '(|Zero| |One|))))
         (push (list 'eq (setq g (is-gensym)) str) $is-eqlist)
         g)
        ((atom str) (errhuh))
        ((eqcar str 'spadlet)
         (cond
           ((identp (second str))
            (push (def-is2 (second str) (third str)) $is-spill-list)
            (second str))
           ((identp (third str))
            (push (deftran str) $is-spill-list) (third str))
           ((errhuh))))
        ((eqcar str 'quote)
         (push (list (cond ((atom (second str)) 'eq) ('equal))
                     (setq g (is-gensym)) str)
                $is-eqlist)
         g)
        ((eqcar str 'list) (def-is-eqlist (list2cons str)))
        ((or (eqcar str 'cons) (eqcar str 'vcons))
         (cons (def-is-eqlist (second str)) (def-is-eqlist (third str))))
        ((eqcar str 'append)
         (unless (identp (second str)) (error "CANT!"))
         (push (def-is2 (list 'reverse (setq g (is-gensym)))
                        (def-is-rev (third str) (second str)))
                $is-eqlist)
         g)
        t)
    $is-eqlist)

```

```

(cond ((eq (second str) '\.) ''t)
      ((push (subst (second str) 'l '(or (setq l (nreverse l)) t))
             $is-spill-list)))
g)
((errhuh))))

```

6.0.32 defvar \$vl

$\langle \textit{initvars} \rangle + \equiv$
 (defparameter \$vl nil)

6.0.33 defun def-is-remdup

[def-is-remdup1 p164]
 [\$vl p163]

$\langle \textit{defun def-is-remdup} \rangle \equiv$
 (defun def-is-remdup (x)
 (let (\$vl)
 (def-is-remdup1 x)))

6.0.34 defun def-is-remdup1

```

[is-gensym p??]
[eqcar p??]
[def-is-remdup1 p164]
[errhuh p176]
[$v1 p163]
[$is-eqlist p166]

⟨defun def-is-remdup1⟩≡
  (defun def-is-remdup1 (x)
    (let (rhs lhs g)
      (declare (special $v1 $is-eqlist))
      (cond
        ((not x) nil)
        ((eq x '\.) x)
        ((identp x)
         (cond
           ((member x $v1)
            (push (list 'equal (setq g (is-gensym)) x) $is-eqlist)
            g)
           ((push x $v1)
            x)))
        ((member x '(|Zero|) (|One|))) x)
      ((atom x) x)
      ((eqcar x 'spadlet)
       (setq rhs (def-is-remdup1 (third x)))
       (setq lhs (def-is-remdup1 (second x)))
       (list 'spadlet lhs rhs))
      ((eqcar x 'let)
       (setq rhs (def-is-remdup1 (third x)))
       (setq lhs (def-is-remdup1 (second x)))
       (list 'let lhs rhs))
      ((eqcar x 'quote) x)
      ((and (eqcar x 'equal) (not (cddr x)))
       (push (list 'equal (setq g (is-gensym)) (second x)) $is-eqlist)
       g)
      ((member (first x) '(list append cons vcons))
       (cons
        (cond ((eq (first x) 'vcons) 'cons) ( (first x)))
        (mapcar #'def-is-remdup1 (cdr x))))
      ((errhuh))))

```

6.0.35 defun addCARorCDR

```
[eqcar p??]
[qcdr p??]
[qcar p??]
```

```
<defun addCARorCDR>≡
  (defun |addCARorCDR| (acc expr)
    (let (funs p funsA funsR)
      (cond
        ((null (pairp expr)) (list acc expr))
        ((and (eq acc 'car) (eqcar expr 'reverse)) (cons '|last| (qcdr expr)))
        (t
         (setq funs
                '(car cdr caar cdar cadr cddr caaar cadar caadr caddr
                  cdaar cddar cdadr cdddr))
         (setq p (position (qcar expr) funs))
         (if (null p)
             (list acc expr)
             (progn
              (setq funsA
                     '(caar cadr caaar cadar caadr caddr caaaar caadar caaadr caaddr
                       cadaar caddar cadadr cadddr))
              (setq funsR
                     '(cdar cddr cdaar cddar cdadr cdddr cdaaar cdadar cdaadr cdaddr
                       cddaar cdddar cddadr cddddr))
              (if (eq acc 'car)
                  (cons (elt funsA p) (qcdr expr))
                  (cons (elt funsR p) (qcdr expr))))))))))
```

```
<initvars>+≡
  (defparameter $IS-GENSYMLIST nil)
```

```
<initvars>+≡
  (defparameter Initial-Gensym (list (gensym)))
```

6.0.36 IS**6.0.37 defun def-is**

```
[def-is2 p167]  
[$is-gensymlist p??]  
[Initial-Gensym p??]
```

```
<defun def-is>≡  
  (defun def-is (x)  
    (let (($is-gensymlist Initial-Gensym))  
      (declare (special is-gensymlist Initial-Gensym))  
      (def-is2 (first X) (second x))))
```

6.0.38 defvar \$is-eqlist

```
<initvars>+≡  
  (defparameter $is-eqlist nil)
```

6.0.39 defun def-is2

```

[eqcar p??]
[moan p??]
[def-is-eqlist p162]
[def-is-remdup p163]
[mkpf p??]
[subst p??]
[dcq p??]
[listofatoms p??]
[/tracelet-print p??]
[$is-eqlist p166]
[$is-spill-list p161]

<defun def-is2>≡
  (defun def-is2 (form struct)
    (let ($is-eqlist $is-spill-list (form (deftran form)))
      (when (eqcar struct '|@Tuple|)
        (moan "you must use square brackets around right arg. to" '%b "is" '%d))
      (let* ((x (def-is-eqlist (def-is-remdup struct)))
             (code (if (identp x)
                        (mkpf (subst form x $is-eqlist) 'and)
                        (mkpf '((dcq ,x ,form) . , $is-eqlist) 'and))))
        (let ((code (mkpf '(',code . , $is-spill-list) 'and)))
          (if $traceletflag
              (let ((l (remove-if #'gensymp (listofatoms x))))
                '(prog1 ,code ,@(mapcar #'(lambda (y) '(/tracelet-print ,y ,y)) L)))
              code))))))

```

6.0.40 defun defIS

```
[deftran p150]
[defIS1 p169]
[$isGenVarCounter p??]
[$inDefIS p??]
```

```
<defun defIS>≡
  (defun |defIS| (lhs rhs)
    (let (|$isGenVarCounter| |$inDefIS|)
      (declare (special |$isGenVarCounter| |$inDefIS|))
      (setq |$isGenVarCounter| 1)
      (setq |$inDefIS| t)
      (|defIS1| (deftran lhs) rhs)))
```


6.0.41 defun defIS1

```
[defLetForm p158]
[defLET1 p154]
[defLET p153]
[defIS1 p169]
[mkprogn p??]
[strconc p??]
[stringimage p??]
[qcar p??]
[qcdr p??]
[defISReverse p172]
[say p??]
[def-is p166]
[$let p59]
[$isGenVarCounter p??]
[$inDefLET p??]
```

```
(defun defIS1)≡
  (defun |defIS1| (lhs rhs)
    (let (d l a1 b1 c cls a b patrev g rev l2)
      (declare (special $let |$isGenVarCounter| |$inDefLET|))
      (cond
        ((null rhs) (list 'null lhs))
        ((stringp rhs) (list 'eq lhs (list 'quote (intern rhs))))
        ((numberp rhs) (list 'equal lhs rhs))
        ((atom rhs) (list 'progn (|defLetForm| rhs lhs) 't))
        ((and (pairp rhs) (eq (qcar rhs) 'quote)
              (pairp (qcdr rhs)) (eq (qcdr (qcdr rhs)) nil))
          (if (identp (qcar (qcdr rhs)))
              (list 'eq lhs rhs)
              (list 'equal lhs rhs)))
        ((and (pairp rhs) (equal (qcar rhs) $let)
              (pairp (qcdr rhs)) (pairp (qcdr (qcdr rhs)))
              (eq (qcdr (qcdr (qcdr rhs))) nil))
          (setq c (qcar (qcdr rhs)))
          (setq d (qcar (qcdr (qcdr rhs))))
          (setq l
            (if |$inDefLET|
              (|defLET1| c lhs)
              (|defLET| c lhs)))
          (list 'and (|defIS1| lhs d) (mkprogn (list l t))))
        ((and (pairp rhs) (eq (qcar rhs) 'equal)
              (pairp (qcdr rhs)) (eq (qcdr (qcdr rhs)) nil))
          (setq a (qcar (qcdr rhs)))
```

```

(list 'equal lhs a ))
(pairp lhs)
(setq g (intern (strconc "ISTMP#" (stringimage |$isGenVarCounter|))))
(setq |$isGenVarCounter| (1+ |$isGenVarCounter|))
(mkprogn (list (list $let g lhs) (|defIS1| g rhs))))
((and (pairp rhs) (eq (qcar rhs) 'cons) (pairp (qcdr rhs))
      (pairp (qcdr (qcdr rhs))) (eq (qcdr (qcdr (qcdr rhs))) nil))
  (setq a (qcar (qcdr rhs)))
  (setq b (qcar (qcdr (qcdr rhs))))
  (cond
   ((eq a (intern "." "BOOT"))
    (if (null b)
        (list 'and (list 'pairp lhs) (list 'eq (list 'qcdr lhs) nil))
        (list 'and (list 'pairp lhs) (|defIS1| (list 'qcdr lhs) b))))
   (null b)
   (list 'and (list 'pairp lhs)
             (list 'eq (list 'qcdr lhs) nil)
             (|defIS1| (list 'qcar lhs) a)))
  ((eq b (intern "." "BOOT"))
   (list 'and (list 'pairp lhs) (|defIS1| (list 'qcar lhs) a)))
  (t
   (setq a1 (|defIS1| (list 'qcar lhs) a))
   (setq b1 (|defIS1| (list 'qcdr lhs) b))
   (cond
    ((and (pairp a1) (eq (qcar a1) 'progn)
          (pairp (qcdr a1)) (pairp (qcdr (qcdr a1)))
          (eq (qcdr (qcdr (qcdr a1))) nil)
          (equal (qcar (qcdr (qcdr a1))) t)
          (pairp b1) (eq (qcar b1) 'progn))
     (setq c (qcar (qcdr a1)))
     (setq cls (qcdr b1))
     (list 'and (list 'pairp lhs) (mkprogn (cons c cls))))
    (t
     (list 'and (list 'pairp lhs) a1 b1))))))
((and (pairp rhs) (eq (qcar rhs) 'append) (pairp (qcdr rhs))
      (pairp (qcdr (qcdr rhs))) (eq (qcdr (qcdr (qcdr rhs))) nil))
  (setq a (qcar (qcdr rhs)))
  (setq b (qcar (qcdr (qcdr rhs))))
  (setq patrev (|defISReverse| b a))
  (setq g (intern (strconc "ISTMP#" (stringimage |$isGenVarCounter|))))
  (setq |$isGenVarCounter| (1+ |$isGenVarCounter|))
  (setq rev
   (list 'and
    (list 'pairp lhs)
    (list 'progn (list $let g (list 'reverse lhs)) t)))
  (setq l2 (|defIS1| g patrev))

```

```

(when (and (pairp l2) (atom (car l2))) (setq l2 (list l2)))
(cond
  ((eq a (intern "." "BOOT"))
   (cons 'and (cons rev l2)))
  (t
   (cons 'and
        (cons rev
              (append l2
                      (list
                       (list 'progn (list (|defLetForm| a (list 'nreverse a )) t))))))))))
(t
 (say "WARNING (defIS1): possibly bad IS code being generated")
 (def-is (list lhs rhs))))))

```

6.0.42 defun def-is-rev

[def-is-rev p171]
 [errhuh p176]

```

⟨defun def-is-rev⟩≡
(defun def-is-rev (x a)
  (let (y)
    (if (eq (first x) 'cons)
        (cond
          ((not (third x)) (list 'cons (second x) a))
          ((setq y (def-is-rev (third x) nil))
           (setf (third y) (list 'cons (second x) a))
           y))
        (errhuh))))

```

6.0.43 defun defISReverse

This reverses forms coming from APPENDs in patterns. It is pretty much just a translation of DEF-IS-REV [defISReverse p172] [errhuh p176]

```

⟨defun defISReverse⟩≡
  (defun |defISReverse| (x a)
    (let (y)
      (if (and (pairp x) (eq (qcar x) 'cons))
        (if (null (caddr x))
          (list 'cons (cadr x) a)
          (progn
            (setq y (|defISReverse| (caddr x) nil))
            (rplac (caddr y) (list 'cons (cadr x) a))
            y))
        (errhuh))))

```

6.0.44 defun def-collect

[def-it p173]
 [deftran p150]
 [hackforis p175]

```

⟨defun def-collect⟩≡
  (defun def-collect (l)
    (def-it 'collect (mapcar #'deftran (hackforis l))))

```

6.0.45 defun def-it

```
[def-in2on p160]
[deftran p150]
[reset p??]
[def-let p153]
[errhuh p176]
```

```
<defun def-it>≡
  (defun def-it (fn l)
    (setq l (reverse l))
    (let ((b (first l)))
      (let ((it (def-in2on (nreverse (rest l)))))
        (let ((itp
              (apply #'append
                    (mapcar
                     #'(lambda (x &aux op y g)
                        (if (and (member (setq op (first x)) '(in on))
                            (not (atom (second x))))
                        (if (eqcar (setq y (second x)) 'spadlet)
                            (if (atom (setq g (second y)))
                                (list
                                 '(',op ,g ,(deftran (third x)))
                                 ' (reset ,(def-let (deftran (third y)) g)))
                                (errhuh))
                        (list
                         '(',op ,(setq g (gensym)) ,(deftran (third x)))
                         ' (reset ,(def-let (deftran (second x)) g)))
                         '(',x)))
              it))))
          (cons fn (nconc itp (list b)))))))
```

6.0.46 defun def-repeat

```
[def-it p173]
[deftran p150]
[hackforis p175]
```

```
<defun def-repeat>≡
  (defun def-repeat (l)
    (def-it 'repeat (mapcar #'deftran (hackforis l))))
```

6.0.47 defun def-string

```
[deftran p150]
[*package* p??]
```

```
<defun def-string>≡
  (defun def-string (x)
    ;; following patches needed to fix reader bug in Lucid Common Lisp
    (if (and (> (size x) 0) (or (char= (elt x 0) #\.) (char= (elt x 0) #\Page)))
        '(intern ,X ,(package-name *package*))
        '(quote ,(deftran (intern x)))))
```

6.0.48 defun def-stringtoquote

```
[def-addlet p174]
[def-stringtoquote p174]
```

```
<defun def-stringtoquote>≡
  (defun def-stringtoquote (x)
    (cond
      ((stringp x) (list 'quote (intern x)))
      ((atom x) x)
      ((cons (def-addlet (first x)) (def-stringtoquote (cdr x))))))
```

6.0.49 defun def-addlet

```
[mkprogn p??]
[def-let p153]
[compfluidize p??]
[$body p??]
```

```
<defun def-addlet>≡
  (defun def-addlet (x)
    (declare (special $body))
    (if (atom x)
        (if (stringp x) '(quote ,(intern x)) x)
        (let ((g (gensym)))
          (setq $body (mkprogn (list (def-let (compfluidize x) g) $body)))
          g)))
```

6.0.50 defun def-inner

```
[def-insert-let p152]
[def-stringtoquote p174]
[sublis p??]
[comp p273]
[$body p??]
[$OpAssoc p??]
[$op p??]
```

```
<defun def-inner>≡
  (defun def-inner (form signature $body)
    "Same as DEF but assumes body has already been DEFTRANned"
    (declare (special $body) (ignore signature))
    (let ($OpAssoc ($op (first form)) (argl (rest form)))
      (declare (special $OpAssoc $op))
      (let* ((argl (def-insert-let argl))
             (arglp (def-stringtoquote argl)))
        (comp (sublis $opassoc '((,$op (lam ,arglp , $body))))))))
```

6.0.51 defun hackforis

```
[hackforis1 p175]
```

```
<defun hackforis>≡
  (defun hackforis (l) (mapcar #'hackforis1 L))
```

6.0.52 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))
```

6.0.53 defun unTuple

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

6.0.54 defun errhuh

```
[systemError p??]
```

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


Chapter 7

PARSE forms

7.1 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: GlyphTok{"\.\.} <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 GlyphTok{"\."} + \. ;

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
                < NBGlyphTok{"\="} Sexpr1
                . (SETQ LABLASOC (CONS (CONS #2 ##1) LABLASOC))>
                / '\'' Sexpr1 + (QUOTE #1)
                / IntegerTok
                / '- ' IntegerTok += (MINUS #1)
                / String
                / '<' <Sexpr1*>!' '>' += (LIST2VEC #1)
                / '(' <Sexpr1* <GlyphTok{"\."} 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 ;

```

7.2 The PARSE code

7.2.1 defvar \$tmptok

```

<initvars>+≡
  (defvar |tmptok| nil)

```

7.2.2 defvar \$tok

```

<initvars>+≡
  (defvar tok nil)

```

7.2.3 defvar \$ParseMode

```

<initvars>+≡
  (defvar |ParseMode| nil)

```

7.2.4 defvar \$definition-name

$\langle initvars \rangle + \equiv$
 (defvar definition-name nil)

7.2.5 defvar \$lablasoc

$\langle initvars \rangle + \equiv$
 (defvar lablasoc nil)

7.2.6 defun PARSE-NewExpr

[match-string p226]
 [action p239]
 [processSynonyms p??]
 [must p239]
 [current-symbol p233]
 [PARSE-Statement p189]
 [definition-name p183]

$\langle defun PARSE-NewExpr \rangle \equiv$
 (defun |PARSE-NewExpr| ()
 (or (and (match-string "") (action (|processSynonyms|))
 (must (|PARSE-Command|)))
 (and (action (setq definition-name (current-symbol)))
 (|PARSE-Statement|))))

7.2.7 defun PARSE-Command

```
[match-advance-string p227]
[must p239]
[PARSE-SpecialKeyWord p184]
[PARSE-SpecialCommand p185]
[push-reduction p241]

⟨defun PARSE-Command⟩≡
  (defun |PARSE-Command| ()
    (and (match-advance-string "") (must (|PARSE-SpecialKeyWord|))
         (must (|PARSE-SpecialCommand|))
         (push-reduction '|PARSE-Command| nil))))
```

7.2.8 defun PARSE-SpecialKeyWord

```
[match-current-token p232]
[action p239]
[token-symbol p??]
[current-token p233]
[unAbbreviateKeyword p??]
[current-symbol p233]

⟨defun PARSE-SpecialKeyWord⟩≡
  (defun |PARSE-SpecialKeyWord| ()
    (and (match-current-token 'identifier)
         (action (setf (token-symbol (current-token))
                       (|unAbbreviateKeyword| (current-symbol))))))
```


7.2.9 defun PARSE-SpecialCommand

```

[match-advance-string p227]
[bang p??]
[optional p240]
[PARSE-Expression p192]
[push-reduction p241]
[PARSE-SpecialCommand p185]
[pop-stack-1 p??]
[PARSE-CommandTail p187]
[must p239]
[current-symbol p233]
[action p239]
[PARSE-TokenList p187]
[PARSE-TokenCommandTail p186]
[star p240]
[PARSE-PrimaryOrQM p188]
[PARSE-CommandTail p187]
[$noParseCommands p??]
[$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 repeater (|PARSE-PrimaryOrQM|))
              (must (|PARSE-CommandTail|))))))

```

7.2.10 defun PARSE-TokenCommandTail

```

[bang p??]
[optional p240]
[star p240]
[PARSE-TokenOption p186]
[atEndOfLine p??]
[push-reduction p241]
[PARSE-TokenCommandTail p186]
[pop-stack-2 p??]
[pop-stack-1 p??]
[action p239]
[systemCommand p??]

```

```

⟨defun PARSE-TokenCommandTail⟩≡
  (defun |PARSE-TokenCommandTail| ()
    (and (bang fil_test (optional (star repeater (|PARSE-TokenOption|))))
         (|atEndOfLine|)
         (push-reduction ' |PARSE-TokenCommandTail|
                          (cons (pop-stack-2) (append (pop-stack-1) nil)))
         (action (|systemCommand| (pop-stack-1)))))

```

7.2.11 defun PARSE-TokenOption

```

[match-advance-string p227]
[must p239]
[PARSE-TokenList p187]

```

```

⟨defun PARSE-TokenOption⟩≡
  (defun |PARSE-TokenOption| ()
    (and (match-advance-string "") (must (|PARSE-TokenList|))))

```

7.2.12 defun PARSE-TokenList

```

[star p240]
[isTokenDelimiter p??]
[push-reduction p241]
[current-symbol p233]
[action p239]
[advance-token p235]

⟨defun PARSE-TokenList⟩≡
  (defun |PARSE-TokenList| ()
    (star repeater
      (and (not (|isTokenDelimiter|))
            (push-reduction '|PARSE-TokenList| (current-symbol))
            (action (advance-token))))))

```

7.2.13 defun PARSE-CommandTail

```

[bang p??]
[optional p240]
[star p240]
[push-reduction p241]
[PARSE-Option p188]
[PARSE-CommandTail p187]
[pop-stack-2 p??]
[pop-stack-1 p??]
[action p239]
[systemCommand p??]

⟨defun PARSE-CommandTail⟩≡
  (defun |PARSE-CommandTail| ()
    (and (bang fil_test (optional (star repeater (|PARSE-Option|))))
          (|atEndOfLine|)
          (push-reduction '|PARSE-CommandTail|
                          (cons (pop-stack-2) (append (pop-stack-1) nil)))
          (action (|systemCommand| (pop-stack-1)))))

```

7.2.14 defun PARSE-PrimaryOrQM

```

[match-advance-string p227]
[push-reduction p241]
[PARSE-PrimaryOrQM p188]
[PARSE-Primary p203]

⟨defun PARSE-PrimaryOrQM⟩≡
  (defun |PARSE-PrimaryOrQM| ()
    (or (and (match-advance-string "?")
              (push-reduction '|PARSE-PrimaryOrQM| '??))
        (|PARSE-Primary|)))

```

7.2.15 defun PARSE-Option

```

[match-advance-string p227]
[must p239]
[star p240]
[PARSE-PrimaryOrQM p188]

⟨defun PARSE-Option⟩≡
  (defun |PARSE-Option| ()
    (and (match-advance-string "")
          (must (star repeater (|PARSE-PrimaryOrQM|)))))

```

7.2.16 defun PARSE-Statement

[PARSE-Expr p193]
 [optional p240]
 [star p240]
 [match-advance-string p227]
 [must p239]
 [push-reduction p241]
 [pop-stack-2 p??]
 [pop-stack-1 p??]

```

<defun PARSE-Statement>≡
  (defun |PARSE-Statement| ()
    (and (|PARSE-Expr| 0)
      (optional
        (and (star repeater
              (and (match-advance-string ",")
                    (must (|PARSE-Expr| 0))))
              (push-reduction '|PARSE-Statement|
                              (cons '|Series|
                                    (cons (pop-stack-2)
                                          (append (pop-stack-1) nil))))))))))

```

7.2.17 defun PARSE-InfixWith

[PARSE-With p190]
 [push-reduction p241]
 [pop-stack-2 p??]
 [pop-stack-1 p??]

```

<defun PARSE-InfixWith>≡
  (defun |PARSE-InfixWith| ()
    (and (|PARSE-With|)
      (push-reduction '|PARSE-InfixWith|
                      (list '|Join| (pop-stack-2) (pop-stack-1)))))

```

7.2.18 defun PARSE-With

```
[match-advance-string p227]  
[must p239]  
[push-reduction p241]  
[pop-stack-1 p??]
```

```
<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))))))
```

7.2.19 defun PARSE-Category

```

[match-advance-string p227]
[must p239]
[bang p??]
[optional p240]
[push-reduction p241]
[PARSE-Expression p192]
[PARSE-Category p191]
[pop-stack-3 p??]
[pop-stack-2 p??]
[pop-stack-1 p??]
[star p240]
[line-number p??]
[PARSE-Application p201]
[action p239]
[recordSignatureDocumentation p??]
[nth-stack p??]
[recordAttributeDocumentation p??]
[current-line p??]

⟨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 repeater
                    (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)))))))

```

7.2.20 defun PARSE-Expression

```

[PARSE-Expr p193]
[PARSE-rightBindingPowerOf p196]
[make-symbol-of p233]
[push-reduction p241]
[pop-stack-1 p??]
[ParseMode p182]
[prior-token p??]

```

```

⟨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))))

```


7.2.21 defun PARSE-Import

```
[match-advance-string p227]
[must p239]
[PARSE-Expr p193]
[bang p??]
[optional p240]
[star p240]
[push-reduction p241]
[pop-stack-2 p??]
[pop-stack-1 p??]
```

```
<defun PARSE-Import>≡
  (defun |PARSE-Import| ()
    (and (match-advance-string "import") (must (|PARSE-Expr| 1000))
      (bang fil_test
        (optional
          (star repeater
            (and (match-advance-string ",")
              (must (|PARSE-Expr| 1000))))))
        (push-reduction '|PARSE-Import|
          (cons '|import|
            (cons (pop-stack-2) (append (pop-stack-1) nil)))))))
```

7.2.22 defun PARSE-Expr

```
[PARSE-NudPart p194]
[PARSE-LedPart p194]
[optional p240]
[star p240]
[push-reduction p241]
[pop-stack-1 p??]
```

```
<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))))
```

7.2.23 defun PARSE-LedPart

[PARSE-Operation p195]
 [push-reduction p241]
 [pop-stack-1 p??]

```

<defun PARSE-LedPart>≡
  (defun |PARSE-LedPart| (rbp)
    (declare (special rbp))
    (and (|PARSE-Operation| '|Led| rbp)
          (push-reduction '|PARSE-LedPart| (pop-stack-1))))

```

7.2.24 defun PARSE-NudPart

[PARSE-Operation p195]
 [PARSE-Reduction p199]
 [PARSE-Form p200]
 [push-reduction p241]
 [pop-stack-1 p??]
 [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))))

```

7.2.25 defun PARSE-Operation

```

[match-current-token p232]
[current-symbol p233]
[PARSE-leftBindingPowerOf p195]
[lt p??]
[getl p??]
[action p239]
[PARSE-rightBindingPowerOf p196]
[PARSE-getSemanticForm p196]
[elemn p??]
[ParseMode p182]
[rbp p??]
[tmptok p182]

```

```

⟨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))))

```

7.2.26 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)))

```

7.2.27 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)))
```

7.2.28 defun PARSE-getSemanticForm

```
[PARSE-Prefix p197]
[PARSE-Infix p197]
```

```
<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|))))
```

7.2.29 defun PARSE-Prefix

```
[push-reduction p241]
[current-symbol p233]
[action p239]
[advance-token p235]
[optional p240]
[PARSE-TokTail p198]
[must p239]
[PARSE-Expression p192]
[push-reduction p241]
[pop-stack-2 p??]
[pop-stack-1 p??]
```

```
<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))))))
```

7.2.30 defun PARSE-Infix

```
[push-reduction p241]
[current-symbol p233]
[action p239]
[advance-token p235]
[optional p240]
[PARSE-TokTail p198]
[must p239]
[PARSE-Expression p192]
[pop-stack-2 p??]
[pop-stack-1 p??]
```

```
<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) )))))
```

7.2.31 defun PARSE-TokTail

[current-symbol p233]
 [current-char p236]
 [char-eq p236]
 [copy-token p??]
 [action p239]
 [PARSE-Qualification p198]
 [\$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))))))

```

7.2.32 defun PARSE-Qualification

[match-advance-string p227]
 [must p239]
 [PARSE-Primary1 p204]
 [push-reduction p241]
 [dollarTran p238]
 [pop-stack-1 p??]

```

<defun PARSE-Qualification>≡
  (defun |PARSE-Qualification| ()
    (and (match-advance-string "$") (must (|PARSE-Primary1|))
      (push-reduction '|PARSE-Qualification|
        (|dollarTran| (pop-stack-1) (pop-stack-1)))))

```

7.2.33 defun PARSE-Reduction

```
[PARSE-ReductionOp p199]
[must p239]
[PARSE-Expr p193]
[push-reduction p241]
[pop-stack-2 p??]
[pop-stack-1 p??]
```

```
<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) )))))
```

7.2.34 defun PARSE-ReductionOp

```
[getl p??]
[current-symbol p233]
[match-next-token p232]
[action p239]
[advance-token p235]
```

```
<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)))))
```

7.2.35 defun PARSE-Form

[match-advance-string p227]
 [bang p??]
 [optional p240]
 [must p239]
 [push-reduction p241]
 [pop-stack-1 p??]
 [PARSE-Application p201]

```

<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|)))

```


7.2.36 defun PARSE-Application

[PARSE-Primary p203]
 [optional p240]
 [star p240]
 [PARSE-Selector p202]
 [PARSE-Application p201]
 [push-reduction p241]
 [pop-stack-2 p??]
 [pop-stack-1 p??]

```

⟨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)))))))

```

7.2.37 defun PARSE-Label

[match-advance-string p227]
 [must p239]
 [PARSE-Name p212]

```

⟨defun PARSE-Label⟩≡
  (defun |PARSE-Label| ()
    (and (match-advance-string "<<") (must (|PARSE-Name|))
      (must (match-advance-string ">>"))))

```

7.2.38 defun PARSE-Selector

```

[current-symbol p233]
[char-ne p237]
[current-char p236]
[match-advance-string p227]
[must p239]
[PARSE-PrimaryNoFloat p203]
[push-reduction p241]
[pop-stack-2 p??]
[pop-stack-1 p??]
[PARSE-Float p205]
[PARSE-Primary p203]
[$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))))))))))

```

7.2.39 defun PARSE-PrimaryNoFloat

[PARSE-Primary1 p204]
 [optional p240]
 [PARSE-TokTail p198]

```
⟨defun PARSE-PrimaryNoFloat⟩≡
  (defun |PARSE-PrimaryNoFloat| ()
    (and (|PARSE-Primary1|) (optional (|PARSE-TokTail|)))))
```

7.2.40 defun PARSE-Primary

[p??]
 [p??]

```
⟨defun PARSE-Primary⟩≡
  (defun |PARSE-Primary| ()
    (or (|PARSE-Float|) (|PARSE-PrimaryNoFloat|)))
```

7.2.41 defun PARSE-Primary1

[PARSE-VarForm p211]
 [optional p240]
 [current-symbol p233]
 [PARSE-Primary1 p204]
 [must p239]
 [pop-stack-2 p??]
 [pop-stack-1 p??]
 [push-reduction p241]
 [PARSE-Quad p210]
 [PARSE-String p210]
 [PARSE-IntegerTok p209]
 [PARSE-FormalParameter p209]
 [match-string p226]
 [PARSE-Data p213]
 [match-advance-string p227]
 [PARSE-Expr p193]
 [PARSE-Sequence p216]
 [PARSE-Enclosure p209]
 [\$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|)))

```

7.2.42 defun PARSE-Float

[PARSE-FloatBase p206]

[must p239]

[PARSE-FloatExponent p208]

[push-reduction p241]

[make-float p??]

[pop-stack-4 p??]

[pop-stack-3 p??]

[pop-stack-2 p??]

[pop-stack-1 p??]

 \langle *defun PARSE-Float* $\rangle \equiv$

(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))))))

7.2.43 defun PARSE-FloatBase

```

[current-symbol p233]
[char-eq p236]
[current-char p236]
[char-ne p237]
[next-char p236]
[PARSE-IntegerTok p209]
[must p239]
[PARSE-FloatBasePart p207]
[PARSE-IntegerTok p209]
[push-reduction p241]
[digitp p??]

```

```

⟨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|))))

```

7.2.44 defun PARSE-FloatBasePart

```

[match-advance-string p227]
[must p239]
[digitp p??]
[current-char p236]
[push-reduction p241]
[token-nonblank p??]
[current-token p233]
[PARSE-IntegerTok p209]

```

```

⟨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))))))

```

7.2.45 defun PARSE-FloatExponent

[current-symbol p233]
 [current-char p236]
 [action p239]
 [advance-token p235]
 [PARSE-IntegerTok p209]
 [match-advance-string p227]
 [must p239]
 [push-reduction p241]
 [identp p??]
 [floatexpid p238]

```

<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))))))

```


7.2.46 defun PARSE-Enclosure

[match-advance-string p227]
 [must p239]
 [PARSE-Expr p193]
 [push-reduction p241]
 [pop-stack-1 p??]

```

<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|))))))))))

```

7.2.47 defun PARSE-IntegerTok

[parse-number p??]

```

<defun PARSE-IntegerTok>≡
  (defun |PARSE-IntegerTok| () (parse-number))

```

7.2.48 defun PARSE-FormalParameter

[PARSE-FormalParameterTok p210]

```

<defun PARSE-FormalParameter>≡
  (defun |PARSE-FormalParameter| () (|PARSE-FormalParameterTok|))

```

7.2.49 defun PARSE-FormalParameterTok

[parse-argument-designator p??]

```
⟨defun PARSE-FormalParameterTok⟩≡
  (defun |PARSE-FormalParameterTok| () (parse-argument-designator))
```

7.2.50 defun PARSE-Quad

[match-advance-string p227]
 [push-reduction p241]
 [PARSE-GlyphTok p215]
 [\$boot p??]

```
⟨defun PARSE-Quad⟩≡
  (defun |PARSE-Quad| ()
    (or (and (match-advance-string "$")
              (push-reduction '|PARSE-Quad| '$))
        (and $boot (|PARSE-GlyphTok| '|.|)
              (push-reduction '|PARSE-Quad| '|.|))))
```

7.2.51 defun PARSE-String

[parse-spadstring p??]

```
⟨defun PARSE-String⟩≡
  (defun |PARSE-String| () (parse-spadstring))
```

7.2.52 defun PARSE-VarForm

[PARSE-Name p212]
 [optional p240]
 [PARSE-Scripts p211]
 [push-reduction p241]
 [pop-stack-2 p??]
 [pop-stack-1 p??]

```

⟨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))))))

```

7.2.53 defun PARSE-Scripts

[match-advance-string p227]
 [must p239]
 [PARSE-ScriptItem p212]

```

⟨defun PARSE-Scripts⟩≡
  (defun |PARSE-Scripts| ()
    (and nonblank (match-advance-string "[" (must (|PARSE-ScriptItem|))
      (must (match-advance-string "]")))))

```

7.2.54 defun PARSE-ScriptItem

[PARSE-Expr p193]
 [optional p240]
 [star p240]
 [match-advance-string p227]
 [must p239]
 [PARSE-ScriptItem p212]
 [push-reduction p241]
 [pop-stack-2 p??]
 [pop-stack-1 p??]

```

<defun PARSE-ScriptItem>≡
  (defun |PARSE-ScriptItem| ()
    (or (and (|PARSE-Expr| 90)
              (optional
                (and (star repeater
                      (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)))))))

```

7.2.55 defun PARSE-Name

[parse-identifier p??]
 [push-reduction p241]
 [pop-stack-1 p??]

```

<defun PARSE-Name>≡
  (defun |PARSE-Name| ()
    (and (parse-identifier) (push-reduction '|PARSE-Name| (pop-stack-1))))

```

7.2.56 defun PARSE-Data

[action p239]
 [PARSE-Sexpr p213]
 [push-reduction p241]
 [translabel p??]
 [pop-stack-1 p??]
 [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))))))

```

7.2.57 defun PARSE-Sexpr

[PARSE-Sexpr1 p214]

```

⟨defun PARSE-Sexpr⟩≡
  (defun |PARSE-Sexpr| ()
    (and (action (advance-token)) (|PARSE-Sexpr1|)))

```

7.2.58 defun PARSE-Sexpr1

```

[PARSE-AnyId p216]
[optional p240]
[PARSE-NBGlyphTok p215]
[must p239]
[PARSE-Sexpr1 p214]
[action p239]
[pop-stack-2 p??]
[nth-stack p??]
[match-advance-string p227]
[push-reduction p241]
[PARSE-IntegerTok p209]
[pop-stack-1 p??]
[PARSE-String p210]
[bang p??]
[star p240]
[PARSE-GlyphTok p215]

```

```

⟨defun PARSE-Sexpr1⟩≡
  (defun |PARSE-Sexpr1| ()
    (or (and (|PARSE-AnyId|)
              (optional
                (and (|PARSE-NBGlyphTok| '=) (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 repeater (|PARSE-Sexpr1|))))
              (must (match-advance-string ">"))
              (push-reduction '|PARSE-Sexpr1| (list2vec (pop-stack-1))))
        (and (match-advance-string "(")
              (bang fil_test
                (optional
                  (and (star repeater (|PARSE-Sexpr1|))
                      (optional
                        (and (|PARSE-GlyphTok| '|.|)

```

```

                                (must (|PARSE-Sexpr1|))
                                (push-reduction '|PARSE-Sexpr1|
                                (nconc (pop-stack-2) (pop-stack-1)))))))))
    (must (match-advance-string "))))))

```

7.2.59 defun PARSE-NBGlyphTok

```

[match-current-token p232]
[action p239]
[advance-token p235]
[tok p182]

```

```

⟨defun PARSE-NBGlyphTok⟩≡
  (defun |PARSE-NBGlyphTok| (|tok|)
    (declare (special |tok|))
    (and (match-current-token 'glyph |tok|) nonblank (action (advance-token)))))

```

7.2.60 defun PARSE-GlyphTok

```

[match-current-token p232]
[action p239]
[advance-token p235]
[tok p182]

```

```

⟨defun PARSE-GlyphTok⟩≡
  (defun |PARSE-GlyphTok| (|tok|)
    (declare (special |tok|))
    (and (match-current-token 'glyph |tok|) (action (advance-token)))))

```

[parse-identifier p??]
[match-string p226]
[push-reduction p241]
[current-symbol p233]
[action p239]
[advance-token p235]
[parse-keyword p??]

7.2.62 defun PARSE-Sequence

[PARSE-OpenBracket p217]
 [must p239]
 [PARSE-Sequence1 p217]
 [match-advance-string p227]
 [PARSE-OpenBrace p218]
 [push-reduction p241]
 [pop-stack-1 p??]

[illegible]

7.2.63 defun PARSE-Sequence1

[PARSE-Expression p192]
 [push-reduction p241]
 [pop-stack-2 p??]
 [pop-stack-1 p??]
 [optional p240]
 [PARSE-IteratorTail p218]

```

<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))))))))))

```

7.2.64 defun PARSE-OpenBracket

[getToken p230]
 [current-symbol p233]
 [eqcar p??]
 [push-reduction p241]
 [action p239]
 [advance-token p235]

```

<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))))))

```

7.2.65 defun PARSE-OpenBrace

```
[getToken p230]
[current-symbol p233]
[eqcar p??]
[push-reduction p241]
[action p239]
[advance-token p235]
```

```
<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))))))
```

7.2.66 defun PARSE-IteratorTail

```
[match-advance-string p227]
[bang p??]
[optional p240]
[star p240]
[PARSE-Iterator p219]
```

```
<defun PARSE-IteratorTail>≡
  (defun |PARSE-IteratorTail| ()
    (or (and (match-advance-string "repeat")
              (bang fil_test (optional (star repeater (|PARSE-Iterator|))))
          (star repeater (|PARSE-Iterator|))))
```

[match-advance-string p227]
[must p239]
[PARSE-Primary p203]
[PARSE-Expression p192]
[PARSE-Expr p193]
[pop-stack-3 p??]
[pop-stack-2 p??]
[pop-stack-1 p??]
[optional p240]

```
(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))))))
```

7.2.68 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.

7.2.69 defun PARSE-Suffix

```

[push-reduction p241]
[current-symbol p233]
[action p239]
[advance-token p235]
[optional p240]
[PARSE-TokTail p198]
[pop-stack-1 p??]

```

```

⟨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))))))

```

```
[match-advance-string p227]
[must p239]
[PARSE-Expr p193]
[push-reduction p241]
[pop-stack-2 p??]
[pop-stack-1 p??]
```

7.2.71 defun PARSE-Return

[match-advance-string p227]
 [must p239]
 [PARSE-Expression p192]
 [push-reduction p241]
 [pop-stack-1 p??]

```

⟨defun PARSE-Return⟩≡
  (defun |PARSE-Return| ()
    (and (match-advance-string "return") (must (|PARSE-Expression|))
      (push-reduction ' |PARSE-Return|
        (list ' |return| (pop-stack-1))))))

```

7.2.72 defun PARSE-Exit

[match-advance-string p227]
 [must p239]
 [PARSE-Expression p192]
 [push-reduction p241]
 [pop-stack-1 p??]

```

<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)))))

```

7.2.73 defun PARSE-Leave

[match-advance-string p227]
 [PARSE-Expression p192]
 [must p239]
 [push-reduction p241]
 [PARSE-Label p201]
 [pop-stack-1 p??]

```

<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)))))
    (list '|leave| (pop-stack-1)))))

```

7.2.74 defun PARSE-Seg

```
[PARSE-GlyphTok p215]
[bang p??]
[optional p240]
[PARSE-Expression p192]
[push-reduction p241]
[pop-stack-2 p??]
[pop-stack-1 p??]
```

```
<defun PARSE-Seg>≡
  (defun |PARSE-Seg| ()
    (and (|PARSE-GlyphTok| '|.|. )
          (bang fil_test (optional (|PARSE-Expression|)))
          (push-reduction '|PARSE-Seg|
                           (list 'segment (pop-stack-2) (pop-stack-1)))))
```

7.2.75 defun PARSE-Conditional

```
[match-advance-string p227]
[must p239]
[PARSE-Expression p192]
[bang p??]
[optional p240]
[PARSE-ElseClause p224]
[push-reduction p241]
[pop-stack-3 p??]
[pop-stack-2 p??]
[pop-stack-1 p??]
```

```
<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)))))
```


7.2.78 defun PARSE-LabelExpr

```
[PARSE-Label p201]
[must p239]
[PARSE-Expr p193]
[push-reduction p241]
[pop-stack-2 p??]
[pop-stack-1 p??]
```

```
<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))))))
```

7.2.79 defun PARSE-FloatTok

```
[parse-number p??]
[push-reduction p241]
[pop-stack-1 p??]
[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))))))
```

7.3 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

7.3.1 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).

7.3.2 defun match-string

The match-string function returns length of X if X matches initial segment of inputstream. [unget-tokens p231]

[skip-blanks p??]
 [line-past-end-p p??]
 [current-char p236]
 [initial-substring-p p228]
 [subseq p??]
 [line-buffer p??]
 [line-current-index p??]
 [line p??]

```

<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))))))

```

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 p229]

[illegible]

7.3.4 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)))
```

7.3.5 defun quote-if-string

```

[token-type p??]
[strconc p??]
[token-symbol p??]
[underscore p230]
[token-nonblank p??]
[pack p??]
[escape-keywords p230]
[$boot p??]
[$spad p264]

<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,'OD" (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)))
                        (token-symbol token))))))
  (t (token-symbol token))))

```

7.3.6 defun escape-keywords

[p??]

```
⟨defun escape-keywords⟩≡
  (defun escape-keywords (pname id)
    (if (member id keywords)
        (concatenate 'string "_" pname)
        pname))
```

7.3.7 defun underscore

[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)))
```

7.3.8 Token Handling

7.3.9 defun getToken

[eqcar p??]

```
⟨defun getToken⟩≡
  (defun |getToken| (x)
    (if (eqcar x '|elt|) (third x) x))
```

7.3.10 defun unget-tokens

```

[quote-if-string p229]
[line-current-segment p??]
[strconc p??]
[line-number p??]
[token-nonblank p??]
[line-new-line p??]
[line-number p??]
[valid-tokens p??]

⟨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?"))))

```

7.3.11 defun match-current-token

This returns the current token if it has EQ type and (optionally) equal symbol.

[current-token p233]

[match-token p232]

```
⟨defun match-current-token⟩≡
  (defun match-current-token (type &optional (symbol nil))
    (match-token (current-token) type symbol))
```

7.3.12 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)))
```

7.3.13 defun match-next-token

This returns the next token if it has equal type and (optionally) equal symbol.

[next-token p234]

[match-token p232]

```
⟨defun match-next-token⟩≡
  (defun match-next-token (type &optional (symbol nil))
    (match-token (next-token) type symbol))
```


7.3.14 defun current-symbol

[make-symbol-of p233]
[current-token p233]

```
<defun current-symbol>≡  
  (defun current-symbol ()  
    (make-symbol-of (current-token)))
```

7.3.15 defun make-symbol-of

[token-symbol p??]

```
<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))))
```

7.3.16 defun current-token

This returns the current token getting a new one if necessary. [try-get-token p234]

[valid-tokens p??]
[current-token p233]

```
<defun current-token>≡  
  (defun current-token ()  
    (declare (special valid-tokens current-token))  
    (if (> valid-tokens 0)  
        current-token  
        (try-get-token current-token)))
```

7.3.17 defun try-get-token

```
[get-token p235]
[valid-tokens p??]
```

```
<defun try-get-token>≡
  (defun try-get-token (token)
    (declare (special valid-tokens))
    (let ((tok (get-token token)))
      (when tok
        (incf valid-tokens)
        token)))
```

7.3.18 defun next-token

This returns the token after the current token, or NIL if there is none after.

```
[try-get-token p234]
[current-token p233]
[valid-tokens p??]
[next-token p234]
```

```
<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)))
```

7.3.19 defun advance-token

This makes the next token be the current token. [current-token p233]

```
[copy-token p??]
[try-get-token p234]
[valid-tokens p??]
[current-token p233]
```

```
<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))))
```

7.3.20 defvar \$XTokenReader

```
<initvars>+≡
  (defvar XTokenReader 'get-meta-token "Name of tokenizing function")
```

7.3.21 defun get-token

```
[XTokenReader p235]
[XTokenReader p235]
```

```
<defun get-token>≡
  (defun get-token (token)
    (funcall XTokenReader token))
```

7.3.22 Character handling

7.3.23 `defun current-char`

This returns the current character of the line, initially blank for an unread line.

[line-past-end-p p??]
 [line-current-char p??]
 [current-line p??]

```
⟨defun current-char⟩≡
  (defun current-char ()
    (if (line-past-end-p current-line)
        #\return
        (line-current-char current-line)))
```

7.3.24 `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 p??]

[line-next-char p??]
 [current-line p??]

```
⟨defun next-char⟩≡
  (defun next-char ()
    (if (line-at-end-p current-line)
        #\return
        (line-next-char current-line)))
```

7.3.25 `defun char-eq`

```
⟨defun char-eq⟩≡
  (defun char-eq (x y)
    (char= (character x) (character y)))
```

7.3.26 defun char-ne

```
<defun char-ne>≡  
  (defun char-ne (x y)  
    (char/= (character x) (character y)))
```

7.3.27 Error handling**7.3.28 defvar \$meta-error-handler**

```
<initvars>+≡  
  (defvar meta-error-handler 'meta-meta-error-handler)
```

7.3.29 defun meta-syntax-error

```
[meta-error-handler p237]  
[meta-error-handler p237]
```

```
<defun meta-syntax-error>≡  
  (defun meta-syntax-error (&optional (wanted nil) (parsing nil))  
    (funcall meta-error-handler wanted parsing))
```

7.3.30 Floating Point Support

7.3.31 defun floatexpid

```
[identp p??]
[pname p??]
[spadreduce p??]
[collect p124]
[step p??]
[maxindex p??]
[digitp p??]

⟨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)))
```

7.3.32 Dollar Translation

7.3.33 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))))
```

7.3.34 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.

7.3.35 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)))
```

7.3.36 defmacro must

```
<defmacro must>≡
  (defmacro must (dothis &optional (this-is nil) (in-rule nil))
    '(or ,dothis (meta-syntax-error ,this-is ,in-rule)))
```

7.3.37 defun action

```
<defun action>≡
  (defun action (dothis) (or dothis t))
```

7.3.38 defun optional

$\langle \text{defun optional} \rangle \equiv$
 (defun optional (dothis) (or dothis t))

7.3.39 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 p241]
 [push p??]
 [pop-stack-1 p??]

$\langle \text{defmacro star} \rangle \equiv$
 (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))))

7.3.40 Stacking and retrieving reductions of rules.**7.3.41** `defun push-reduction`

```
[stack-push p??]  
[make-reduction p??]  
[reduce-stack p??]
```

```
<defun push-reduction>≡  
  (defun push-reduction (rule redn)  
    (stack-push (make-reduction :rule rule :value redn) reduce-stack))
```


Chapter 8

The Compiler

8.1 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> (|comp0rCroak| ...)
10> (|comp0rCroak1| ...)
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|)
```

8.1.1 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"))

```

```

[helpSpad2Cmd(5) p??]
[selectOptionLC(5) p??]
[pathname(5) p??]
[mergePathnames(5) p??]
[pathnameType(5) p??]
[namestring(5) p??]
[throwKeyedMsg p??]
[findfile p??]
[compileSpad2Cmd p249]
[compileSpadLispCmd p305]
[$newConlist p??]
[$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))))))))))))))

```


8.1.2 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|))

[pathname(5) p??]
[pathnameType(5) p??]
[namestring(5) p??]
[updateSourceFiles(5) p??]
[selectOptionLC(5) p??]
[terminateSystemCommand(5) p??]
[nequal p??]
[throwKeyedMsg p??]
[sayKeyedMsg p??]
[error p??]
[strconc p??]
[object2String p??]
[oldParserAutoloadOnceTrigger p??]
[browserAutoloadOnceTrigger p??]
[spad2AsTranslatorAutoloadOnceTrigger p??]
[convertSpadToAsFile p??]
[compilerDoitWithScreenedLisplib p??]
[compilerDoit p253]
[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| |$InteractiveModel| path optlist fun optname
          optargs fullopt constructor)
      (declare (special |$newComp| |$scanIfTrue|
                        |$compileOnlyCertainItems| |$f| |$m| |$QuickLet| |$QuickCode|
                        |$sourceFileTypes| |$InteractiveModel| /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) ->
```

8.1.3 defun compilerDoit

```
[/rq(5) p??]
[/rf(5) p??]
[member(5) p??]
[sayBrightly p??]
[$byConstructors p307]
[$constructorsSeen p307]

<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)
```

8.1.4 defun `/RQ,LIB`

```
[/rf-1(5) p??]
[echo-meta(5) p??]
[$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)
```

8.1.5 defun /rf-1

```
[makeInputFilename(5) p??]
[ncINTERPFILE p304]
[spad p264]
[/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.nrllib/EQ.lsp")
<3 (OPEN #<input stream "/tmp/EQ.nrllib/EQ.lsp">)
3> (OPEN #p"/tmp/EQ.nrllib/EQ.data" :DIRECTION :OUTPUT)
<3 (OPEN #<output stream "/tmp/EQ.nrllib/EQ.data">)
3> (OPEN #p"/tmp/EQ.nrllib/EQ.c" :DIRECTION :OUTPUT)
<3 (OPEN #<output stream "/tmp/EQ.nrllib/EQ.c">)
3> (OPEN #p"/tmp/EQ.nrllib/EQ.h" :DIRECTION :OUTPUT)
<3 (OPEN #<output stream "/tmp/EQ.nrllib/EQ.h">)
3> (OPEN #p"/tmp/EQ.nrllib/EQ.fn" :DIRECTION :OUTPUT)
<3 (OPEN #<output stream "/tmp/EQ.nrllib/EQ.fn">)
3> (OPEN #p"/tmp/EQ.nrllib/EQ.o" :DIRECTION :OUTPUT :IF-EXISTS :APPEND)
<3 (OPEN #<output stream "/tmp/EQ.nrllib/EQ.o">)
3> (OPEN #p"/tmp/EQ.nrllib/EQ.data")
<3 (OPEN #<input stream "/tmp/EQ.nrllib/EQ.data">)
3> (OPEN "/tmp/EQ.nrllib/index.kaf")
<3 (OPEN #<input stream "/tmp/EQ.nrllib/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| $) (|:| |eqn2| (|List| $)))) $)
(=
(|eval|
(|,|
(|eqn1| |lhs|)
(|pretend| |eqn2| (|List| (|Equation| S))))
(|eval|
(|,|
(|eqn1| |rhs|)
(|pretend| |eqn2| (|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))))))

```

8.1.6 defun spad

```

[spad-reader p??]
[addBinding p??]
[makeInitialModemapFrame p??]
[init-boot/spad-reader p??]
[initialize-prepare p14]
[prepare p18]
[PARSE-NewExpr p183]
[pop-stack-1 p??]
[s-process p267]
[ioclear p??]
[shut p??]
[$noSubsumption p??]
[$InteractiveFrame p??]
[$InitialDomainsInScope p??]
[$InteractiveMode p??]
[line p??]
[echo-meta p??]
[/editfile p??]
[*comp370-apply* p??]
[*eof* p??]
[file-closed p??]
[xcape 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) (xcape #\_) (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 xcape |$noSubsumption| |$InteractiveFrame|
          |$InteractiveModel| |$InitialDomainsInScope|))
;; only rebind |$InteractiveFrame| if compiling
(progv (if (not |$InteractiveModel|) '(|$InteractiveFrame|))
      (if (not |$InteractiveModel|)
          (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-prepare 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 (prepare 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))

```


8.1.7 defun Interpreter interface to the compiler

```

[curstrm p??]
[def-rename p151]
[new2OldLisp p??]
[parseTransform p35]
[postTransform p111]
[displayPreCompilationErrors p??]
[prettyprint p??]
[processInteractive p??]
[compTopLevel p270]
[def-process p151]
[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??]
[$stopOp p??]
[$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)
           |$stopOp|
           (|$semanticErrorStack| ())
           (|$warningStack| ())
           (|$exitMode| |$EmptyMode|)
           (|$exitModeStack| ())
           (|$returnMode| |$EmptyMode|)
           (|$leaveMode| |$EmptyMode|)
           (|$leaveLevelStack| ())
           $stop_level |$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| |$stopOp|
                        |$semanticErrorStack| |$warningStack| |$exitMode| |$exitModeStack|
                        |$returnMode| |$leaveMode| |$leaveLevelStack| $stop_level
                        |$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)))

```

8.1.8 defun compTopLevel

```

[newComp p??]
[compOrCroak p271]
[$NRTderivedTargetIfTrue p??]
[$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.

8.1.9 defun compOrCroak

[compOrCroak1 p272]

```
(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|) |CEpr|))
        (SIGNATURE |operand| (|CEpr| |CEpr| (|Integer|)))
        (SIGNATURE |numberOfOperand| ((|Integer|) |CEpr|))
        (SIGNATURE |construct| (|CEpr| |CEpr| |CEpr|))))
    |$EmptyMode|
    (((
      |$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.

8.1.10 defun compOrCroak1

```
[comp p273]
[compOrCroak1,compactify p304]
[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 p271]
```

$\langle \text{defun compOrCroak1} \rangle \equiv$


```

(defun |compOrCroak1| (x m e |$compStack| |$compErrorMessageStack|)
  (declare (special |$compStack| |$compErrorMessageStack|))
  (let (td errorMessage)
    (declare (special |$level| |$s| |$scanIfTrue| |$exitModeStack|))
    (cond
      ((setq td (catch '|compOrCroak1| (|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)))))))

```

8.1.11 defun comp

```

[compNoStacking p274]
[$compStack p??]
[$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))

```

8.1.12 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 p275]

```
[compNoStacking1 p274]
[$compStack p??]
[$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|))))
```

8.1.13 defun compNoStacking1

```
[get p??]
[comp2 p275]
[$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)))
```

8.1.14 defun comp2

```

[comp3 p276]
[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))))))

```

8.1.15 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 p291]
[stringimage p??]
[compAtom p280]
[getmode p??]
[applyMapping p??]
[compApply p??]
[compColon p79]
[compCoerce p77]
[stringPrefix? p??]
[pname p??]
[compTypeOf p278]
[compExpression p285]
[member p??]
[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 |$e|)))
  ((stringp m)
    (when (and (atom x) (or (equal m x) (equal m (stringimage x))))
      (list m m e )))
  ((or (null x) (atom x)) (|compAtom| x m e))
  (t
    (setq op (car x))
    (cond
      ((and (progn
        (setq tmp1 (|getmode| op e))
        (and (pairp tmp1)
          (eq (qcar tmp1) '|Mapping|)
          (progn (setq ml (qcdr tmp1)) t)))
        (setq u (|applyMapping| x m e ml)))
        u)
      ((and (pairp op) (eq (qcar op) 'kappa)
        (progn
          (setq tmp1 (qcdr op))
          (and (pairp tmp1)
            (progn
              (setq sig (qcar tmp1))
              (setq tmp2 (qcdr tmp1))
              (and (pairp tmp2)
                (progn
                  (setq varlist (qcar tmp2))
                  (setq tmp3 (qcdr tmp2))
                  (and (pairp tmp3)
                    (eq (qcdr tmp3) nil)
                    (progn
                      (setq body (qcar tmp3))
                      t))))))))))
            (|compApply| sig varlist body (cdr x) m e))
        ((eq op '|:|) (|compColon| x m e))
        ((eq op '|::|) (|compCoerce| x m e))
        ((and (null (eq |$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))))))

```

8.1.16 defun compTypeOf

```

[eqsubstlist p??]
[get p??]
[put p??]
[comp3 p276]
[$insideCompTypeOf p??]
[$FormalMapVariableList p??]

⟨defun compTypeOf⟩≡
  (defun |compTypeOf| (x m e)
    (let (|$insideCompTypeOf| op arg1 newModemap)
      (declare (special |$insideCompTypeOf| |$FormalMapVariableList|))
      (setq op (car x))
      (setq arg1 (cdr x))
      (setq |$insideCompTypeOf| t)
      (setq newModemap
        (eqsubstlist arg1 |$FormalMapVariableList| (|get| op '|modemap| e)))
      (setq e (|put| op '|modemap| newModemap e))
      (|comp3| x m e)))

```

8.1.17 defun compColonInside

```

[addDomain p??]
[comp p273]
[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))))))

```

8.1.18 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 p284]
[compVector p109]
[convert p281]
[isSymbol p??]
[compSymbol p283]
[primitiveType p282]
[primitiveType p282]
[$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))))))

```

8.1.19 defun convert

[resolve p??]
[coerce p??]

```

⟨defun convert⟩≡
  (defun |convert| (td m)
    (let (res)
      (when (setq res (|resolve| (second td) m))
        (|coerce| td res))))

```

8.1.20 defun primitiveType

```

[$DoubleFloat p??]
[$NegativeInteger p??]
[$PositiveInteger p??]
[$NonNegativeInteger p??]
[$String p??]
[$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
         ((eq x 0) |$NonNegativeInteger|)
         ((> x 0) |$PositiveInteger|)
         (t |$NegativeInteger|)))
      ((floatp x) |$DoubleFloat|)
      (t nil)))

```

8.1.21 defun compSymbol

```

[getmode p??]
[get p??]
[NRTgetLocalIndex p??]
[member p??]
[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))))))

```

8.1.22 defun compList

```

;compList(l,m is ["List",mUnder],e) ==
;  null l => [NIL,m,e]
;  Tl:= {\tt{.},mUnder,e]:= \ comp(x,mUnder,e)\ or\ return\ "failed"\ for\ x\ in\ l\ \nwnewline
;\ \ Tl="failed"\ =>\ nil\ \nwnewline
;\ \ T:= \ ["LIST",:[T.expr\ for\ T\ in\ Tl],["List",mUnder],e]

```

[comp p273]

```

⟨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)))))))

```

8.1.23 defun compExpression

```
[get1 p??]
[compForm p285]
[$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 (get1 (car x) 'special)))
          (funcall fn x m e)
          (|compForm| x m e))))
```

8.1.24 defun compForm

```
[compForm1 p286]
[compArgumentsAndTryAgain p291]
[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| )))))
```

8.1.25 defun compForm1

```

[length p??]
[outputComp p??]
[compOrCroak p271]
[compExpressionList p??]
[coerceable p??]
[comp p273]
[coerce p??]
[compForm2 p289]
[augModemapsFromDomain1 p??]
[getFormModemaps p??]
[nreverse0 p??]
[addDomain p??]
[compToApply p??]
[$NumberOfArgsIfInteger p??]
[$Expression p??]
[$EmptyMode p??]

⟨defun compForm1⟩≡
  (defun |compForm1| (form m e)
    (let (|$NumberOfArgsIfInteger| op arg1 domain tmp1 opprime ans mmList td
          tmp2 tmp3 tmp4 tmp5 tmp6 tmp7)
      (declare (special |$NumberOfArgsIfInteger| |$Expression| |$EmptyMode|))
      (setq op (car form))
      (setq arg1 (cdr form))
      (setq |$NumberOfArgsIfInteger| (|#| arg1))
      (cond
        ((eq op '|error|)
         (list
          (cons op
                (dolist (x arg1 (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 arg1 (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| arg1 m e))
    ((and (eq opprime 'collect) (|coerceable| domain m e))
      (when (setq td (|comp| (cons opprime arg1) domain e))
        (|coerce| td m)))
    ((and (pairp domain) (eq (qcar domain) '|Mapping|)
      (setq ans
        (|compForm2| (cons opprime arg1) m
          (setq e (|augModemapsFromDomain1| domain domain e))
          (dolist (x (|getFormModemaps| (cons opprime arg1) 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 arg1) m
          (setq e (|addDomain| domain e))
          (dolist (x (|getFormModemaps| (cons opprime arg1) 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 arg1) 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 arg1 m e))))))
```


8.1.26 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 arg1 sarg1 aList dc cond nsig v ncond deleteList newList td t1
          partialModeList tmp1 tmp2 tmp3 tmp4 tmp5 tmp6 tmp7)
      (declare (special |$EmptyMode| |$TriangleVariableList|))
      (setq op (car form))
      (setq arg1 (cdr form))
      (setq sarg1 (take (|#| arg1) |$TriangleVariableList|))
      (setq aList (mapcar #'(lambda (x y) (cons x y)) sarg1 arg1))
      (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 t1
  (loop for x in arg1
    while (and (|isSimple| x)
      (setq td (|compUniquely| x |$EmptyMode| e)))
    collect td
    do (setq e (third td))))
(cond
  ((some #'identity t1)
    (setq partialModeList (loop for x in t1 collect (when x (second x))))
    (or (|compFormPartiallyBottomUp| form m e modemapList partialModeList)
      (|compForm3| form m e modemapList)))
  (t (|compForm3| form m e modemapList))))

```

8.1.27 defun compArgumentsAndTryAgain

```
[comp p273]
[compForm1 p286]
[$EmptyMode p??]

⟨defun compArgumentsAndTryAgain⟩≡
  (defun |compArgumentsAndTryAgain| (form m e)
    (let (arg1 tmp1 a tmp2 tmp3 u)
      (declare (special |$EmptyMode|))
      (setq arg1 (cdr form))
      (cond
        ((and (pairp form) (eq (qcar form) '|elt|))
          (progn
            (setq tmp1 (qcdr form))
            (and (pairp tmp1)
              (progn
                (setq a (qcar tmp1))
                (setq tmp2 (qcdr tmp1))
                (and (pairp tmp2) (eq (qcdr tmp2) nil)))))))
        (t
          (when (setq tmp3 (|comp| a |$EmptyMode| e))
            (setq e (third tmp3))
            (|compForm1| form m e)))
          (setq u
            (dolist (x arg1)
              (setq tmp3 (or (|comp| x |$EmptyMode| e) (return '|failed|)))
              (setq e (third tmp3))
              tmp3))
            (unless (eq u '|failed|)
              (|compForm1| form m e)))))))
```

8.1.28 defun compWithMappingMode

```
[compWithMappingMode1 p292]
[$formalArgList p??]

⟨defun compWithMappingMode⟩≡
  (defun |compWithMappingMode| (x m oldE)
    (declare (special |$formalArgList|))
    (|compWithMappingMode1| x m oldE |$formalArgList|))
```

8.1.29 defun compWithMappingMode1

```
;compWithMapping(x,m is ["Mapping",m',:sl],oldE,$formalArgList) ==
; $skillOptimizeIfTrue: local:= true
; e:= oldE
; isFunctor x =>
;   if get(x,"modemap",$CategoryFrame) is {\tt{[.,target,:argModeList],[.,.:]}\ and\nwnewline
;\ \ \ \ \ (and/[extendsCategoryForm("{\char36}",s,mode)\ for\ mode\ in\ argModeList\ for\ s\
;\ \ \ \ \ \ \ )\ and\ extendsCategoryForm("{\char36}",target,m')\ then\ return\ [x,m,e]\nwnewline
;\ \ if\ STRINGP\ x\ then\ x:=\ INTERN\ x\nwnewline
;\ \ ress:=nil\nwnewline
;\ \ old{\char95}style:=true\nwnewline
;\ \ if\ x\ is\ ["+>",vl,nx]\ then\nwnewline
;\ \ \ \ old{\char95}style:=false\nwnewline
;\ \ \ \ vl\ is\ [":",.:]\ =>\nwnewline
;\ \ \ \ \ ress:=compLambda(x,m,oldE)\nwnewline
;\ \ \ \ \ ress\nwnewline
;\ \ \ \ \ vl:=\nwnewline
;\ \ \ \ \ vl\ is\ ["Tuple",:vl1]\ =>\ vl1\nwnewline
;\ \ \ \ \ vl\nwnewline
;\ \ \ \ \ vl:=\nwnewline
;\ \ \ \ \ SYMBOLP(vl)\ =>\ [vl]\nwnewline
;\ \ \ \ \ LISTP(vl)\ and\ (and/[SYMBOLP(v)\ for\ v\ in\ vl])\ =>\ vl\nwnewline
;\ \ \ \ \ stackAndThrow\ ["bad\ +->\ arguments:",vl]\nwnewline
;\ \ \ \ {\char36}formatArgList:=[:vl,{:\char36}formalArgList]\nwnewline
;\ \ \ \ x=nx\nwnewline
;\ \ else\nwnewline
;\ \ \ \ vl:=take({:\char35}sl,{:\char36}FormalMapVariableList)\nwnewline
;\ \ \ ress\ =>\ ress\nwnewline
;\ \ for\ m\ in\ sl\ for\ v\ in\ vl\ repeat\nwnewline
;\ \ \ \ [,.,e]:=\ compMakeDeclaration([":",v,m],{\char36}EmptyMode,e)\nwnewline
;\ \ \ old{\char95}style\ and\ not\ null\ vl\ and\ not\ hasFormalMapVariable(x,\ vl)\ =>\ return
;\ \ \ \ [u,.,.] :=\ comp([x,:vl],m',e)\ or\ return\ nil\nwnewline
;\ \ \ \ extractCodeAndConstructTriple(u,\ m,\ oldE)\nwnewline
;\ \ \ null\ vl\ and\ (t:=\ comp([x],\ m',\ e))\ =>\ return\nwnewline
;\ \ \ \ [u,.,.] :=\ t\nwnewline
;\ \ \ \ extractCodeAndConstructTriple(u,\ m,\ oldE)\nwnewline
;\ \ \ [u,.,.] :=\ comp(x,m',e)\ or\ return\ nil\nwnewline
;\ \ 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
;      {\tt{}u,:1],:free]\nwnewline
;\ \ \ \ \ op:=CAR\ u\nwnewline
;\ \ \ \ \ MEMQ(op,\ '(QUOTE\ GO\ function))\ =>\ free\nwnewline
;\ \ \ \ \ EQ(op,'LAMBDA)\ =>\nwnewline
;\ \ \ \ \ bound:=UNIONQ(bound,CADR\ u)\nwnewline
;\ \ \ \ \ for\ v\ in\ CDDR\ u\ repeat\nwnewline
;\ \ \ \ \ free:=freelist(v,bound,free,e)\nwnewline
;\ \ \ \ \ free\nwnewline
;\ \ \ \ \ EQ(op,'PROG)\ =>\nwnewline
;\ \ \ \ \ bound:=UNIONQ(bound,CADR\ u)\nwnewline
;\ \ \ \ \ for\ v\ in\ CDDR\ u\ |\ NOT\ ATOM\ v\ repeat\nwnewline
;\ \ \ \ \ free:=freelist(v,bound,free,e)\nwnewline
;\ \ \ \ \ free\nwnewline
;\ \ \ \ \ EQ(op,'SEQ)\ =>\nwnewline
;\ \ \ \ \ for\ v\ in\ CDR\ u\ |\ NOT\ ATOM\ v\ repeat\nwnewline
;\ \ \ \ \ free:=freelist(v,bound,free,e)\nwnewline
;\ \ \ \ \ free\nwnewline
;\ \ \ \ \ EQ(op,'COND)\ =>\nwnewline
;\ \ \ \ \ for\ v\ in\ CDR\ u\ repeat\nwnewline
;\ \ \ \ \ for\ vv\ in\ v\ repeat\nwnewline
;\ \ \ \ \ free:=freelist(vv,bound,free,e)\nwnewline
;\ \ \ \ \ free\nwnewline
;\ \ \ \ \ if\ ATOM\ op\ then\ u:=CDR\ u\ \ --Atomic\ functions\ aren't\ descended\nwnewline
;\ \ \ \ \ for\ v\ in\ u\ repeat\nwnewline
;\ \ \ \ \ free:=freelist(v,bound,free,e)\nwnewline
;\ \ \ \ \ free\nwnewline
;\ \ expandedFunction\ :=\nwnewline
;\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ --One\ free\ can\ go\ by\ itself,\ more\ than\ one\ needs\ a\ vector\nwnewline
;\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ --An\ A-list\ name\ .\ number\ of\ times\ used\nwnewline
;\ \ \ \ \ {\char35}frees\ =\ 0\ =>\ ['LAMBDA,[:vl,"{\char36}{\char36}",\ :CDDR\ expandedFunction]\nwnewline
;\ \ \ \ \ {\char35}frees\ =\ 1\ =>\nwnewline
;\ \ \ \ \ vec:=first\ first\ frees\nwnewline
;\ \ \ \ \ ['LAMBDA,[:vl,vec],\ :CDDR\ expandedFunction]\nwnewline
;\ \ \ \ scode:=nil\nwnewline
;\ \ \ \ vec:=nil\nwnewline
;\ \ \ \ locals:=nil\nwnewline
;\ \ \ \ i:=-1\nwnewline
;\ \ \ \ for\ v\ in\ frees\ repeat\nwnewline
;\ \ \ \ \ i:=i+1\nwnewline
;\ \ \ \ \ vec:=[first\ v,:vec]\nwnewline
;\ \ \ \ \ scode:=[['SETQ,first\ v,[(\{\char36}QuickCode\ =>\ 'QREFELT;'ELT),"\{\char36}{\char36}",i],:s
;      locals:=[first v,:locals]
;      body:=CDDR expandedFunction
;      if locals then
;      if body is {\tt{}'DECLARE,..],:.]\ then\nwnewline

```

```

;\ \ \ \ \ \ \ body:=[CAR\ body,['PROG,locals,:scode,['RETURN,['PROGN,:CDR\ body]]]
;      else body:={\tt{'PROG,locals,:scode,['RETURN,['PROGN,:body]]}
;      vec:=['VECTOR,:NREVERSE vec]
;      ['LAMBDA,[:v1,"$$"],: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 p100]
[stackAndThrow p??]
[take p??]
[compMakeDeclaration p302]
[hasFormalMapVariable p300]
[comp p273]
[extractCodeAndConstructTriple p300]
[optimizeFunctionDef p??]
[comp-tran p??]
[freelist p303]
[$formalArgList p??]
[$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 $funnameTail mprime s1 tmp1 tmp2
      tmp3 tmp4 tmp5 tmp6 target argModeList nx oldstyle ress v11 v1 e tt
      u frees i scode locals body vec expandedFunction fname uu)
      (declare (special |$killOptimizeIfTrue| $funname $funnameTail
        |$QuickCode| |$EmptyMode| |$FormalMapVariableList|
        |$CategoryFrame| |$formatArgList|))
      (return

```

```

(seq
  (progn
    (setq mprime (second m))
    (setq s1 (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))))))
                    (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))))))))
            (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 s1 (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 v1 (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 v1) (eq (qcar v1) '|:|))
    (setq ress (|compLambda| x m oldE))
    ress)
   (t
    (setq v1
      (cond
       ((and (pairp v1)
              (eq (qcar v1) '|@Tuple|)
              (progn (setq v11 (qcdr v1)) t))
        v11)
       (t v1)))
    (setq v1
      (cond
       ((symbolp v1) (cons v1 nil))
       ((and
         (listp v1)
         (prog (t5)
              (setq t5 t)
              (return
               (do ((t7 nil (null t5))
                   (t6 v1 (cdr t6))
                   (v nil))
                   ((or t7 (atom t6) (progn (setq v (car t6)) nil)) t5)
               (seq
                (exit
                 (setq t5 (and t5 (symbolp v))))))))))
        v1)

```



```

      (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)))))))))

```

8.1.30 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)))))

```

8.1.31 defun hasFormalMapVariable

```

[ScanOrPairVec p??]
[$formalMapVariables p??]

```

```

<defun hasFormalMapVariable>≡
  (defun |hasFormalMapVariable| (x vl)
    (let (|$formalMapVariables|)
      (declare (special |$formalMapVariables|))
      (when (setq |$formalMapVariables| vl)
        (|ScanOrPairVec| #'(lambda (y) (member y |$formalMapVariables|)) x))))

```

8.1.32 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))))
              (t
                (cond
                  (bad (list nil nil ))
                  (t (list (reverse arg1) (reverse sig1)))))))
          (t (list (reverse arg1) (reverse sig1))))))

```

8.1.33 defun compMakeDeclaration

```
[compColon p79]
[$insideExpressionIfTrue p??]
```

```
<defun compMakeDeclaration>≡
  (defun |compMakeDeclaration| (x m e)
    (let (|$insideExpressionIfTrue|)
      (declare (special |$insideExpressionIfTrue|))
      (setq |$insideExpressionIfTrue| nil)
      (|compColon| x m e)))
```

8.1.34 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|)))))
```

8.1.35 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 p303]

```
[assq p??]
[identp p??]
[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)))

```

8.1.36 defun compOrCroak1,compactify

```

[compOrCroak1,compactify p304]
[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))))))

```

8.1.37 defun Compiler/Interpreter interface

```

[SpadInterpretStream(5) p??]
[$EchoLines p??]
[$ReadingFile p??]

```

```

⟨defun ncINTERPFILE⟩≡
  (defun |ncINTERPFILE| (file echo)
    (let ((|$EchoLines| echo) (|$ReadingFile| t))
      (declare (special |$EchoLines| |$ReadingFile|))
      (|SpadInterpretStream| 1 file nil)))

```


8.1.38 defun compileSpadLispCmd

```

[pathname(5) p??]
[pathnameType(5) p??]
[selectOptionLC(5) p??]
[namestring(5) p??]
[terminateSystemCommand(5) p??]
[fnameMake(5) p??]
[pathnameDirectory(5) p??]
[pathnameName(5) p??]
[fnameReadable?(5) p??]
[localdatabase(5) p??]
[throwKeyedMsg p??]
[object2String p??]
[sayKeyedMsg p??]
[recompile-lib-file-if-necessary p306]
[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|))))))

```

8.1.39 defun recompile-lib-file-if-necessary

[compile-lib-file p307]
 [*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))))

```

8.1.40 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)

```

8.1.41 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)))

```

8.1.42 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)))

```

8.1.43 defvar \$byConstructors

```

⟨initvars⟩+≡
  (defvar |$byConstructors| () "list of constructors to be compiled")

```

8.1.44 defvar \$constructorsSeen

```

⟨initvars⟩+≡
  (defvar |$constructorsSeen| () "list of constructors found")

```

```

⟨Compiler⟩≡
  (in-package "BOOT")

  ⟨initvars⟩

  ⟨defmacro bang⟩
  ⟨defmacro must⟩
  ⟨defmacro star⟩

  ⟨defun action⟩
  ⟨defun addCARorCDR⟩
  ⟨defun add-parens-and-semis-to-line⟩
  ⟨defun advance-token⟩
  ⟨defun aplTran⟩
  ⟨defun aplTran1⟩
  ⟨defun aplTranList⟩
  ⟨defun argsToSig⟩

  ⟨defun char-eq⟩
  ⟨defun char-ne⟩
  ⟨defun comma2Tuple⟩
  ⟨defun comp⟩
  ⟨defun comp2⟩
  ⟨defun comp3⟩
  ⟨defun compAdd⟩
  ⟨defun compArgumentsAndTryAgain⟩
  ⟨defun compAtom⟩
  ⟨defun compAtSign⟩
  ⟨defun compCapsule⟩
  ⟨defun compCapsuleInner⟩
  ⟨defun compCase⟩
  ⟨defun compCase1⟩
  ⟨defun compCat⟩
  ⟨defun compCategory⟩
  ⟨defun compCoerce⟩
  ⟨defun compCoerce1⟩
  ⟨defun compColon⟩
  ⟨defun compColonInside⟩
  ⟨defun compCons⟩
  ⟨defun compCons1⟩
  ⟨defun compConstruct⟩
  ⟨defun compConstructorCategory⟩
  ⟨defun compDefine⟩
  ⟨defun compDefine1⟩
  ⟨defun compElt⟩
  ⟨defun compExit⟩

```

```

⟨defun compExpression⟩
⟨defun compForm⟩
⟨defun compForm1⟩
⟨defun compForm2⟩
⟨defun compHas⟩
⟨defun compIf⟩
⟨defun compileFileQuietly⟩
⟨defun compile-lib-file⟩
⟨defun compiler⟩
⟨defun compilerDoit⟩
⟨defun compileSpad2Cmd⟩
⟨defun compileSpadLispCmd⟩
⟨defun compImport⟩
⟨defun compIs⟩
⟨defun compJoin⟩
⟨defun compLambda⟩
⟨defun compLeave⟩
⟨defun compList⟩
⟨defun compMacro⟩
⟨defun compMakeDeclaration⟩
⟨defun compNoStacking⟩
⟨defun compNoStacking1⟩
⟨defun compOrCroak⟩
⟨defun compOrCroak1⟩
⟨defun compOrCroak1,compactify⟩
⟨defun compPretend⟩
⟨defun compQuote⟩
⟨defun compReduce⟩
⟨defun compReduce1⟩
⟨defun compSeq⟩
⟨defun compSeqItem⟩
⟨defun compSeq1⟩
⟨defun compSymbol⟩
⟨defun compTopLevel⟩
⟨defun compTypeOf⟩
⟨defun compVector⟩
⟨defun compWhere⟩
⟨defun compWithMappingMode⟩
⟨defun compWithMappingMode1⟩
⟨defun containsBang⟩
⟨defun convert⟩
⟨defun current-char⟩
⟨defun current-symbol⟩
⟨defun current-token⟩

⟨defun decodeScripts⟩

```

```
<defun deepestExpression>
<defun def>
<defun def-addlet>
<defun def-collect>
<defun def-cond>
<defun def-inner>
<defun def-insert-let>
<defun def-in2on>
<defun def-is>
<defun def-is2>
<defun def-is-eqlist>
<defun defIS>
<defun defIS1>
<defun defISReverse>
<defun def-is-remdup>
<defun def-is-remdup1>
<defun def-is-rev>
<defun def-it>
<defun def-let>
<defun defLET>
<defun defLET1>
<defun defLET2>
<defun defLetForm>
<defun def-message>
<defun def-message1>
<defun def-process>
<defun def-rename>
<defun def-rename1>
<defun def-repeat>
<defun def-string>
<defun def-stringtoquote>
<defun deftran>
<defun def-where>
<defun def-whereclause>
<defun def-whereclauselist>
<defun dollarTran>

<defun errhuh>
<defun escape-keywords>
<defun extractCodeAndConstructTriple>

<defun floatexid>
<defun freelist>

<defun get-a-line>
<defun getScriptName>
```

```

⟨defun get-token⟩
⟨defun getToken⟩

⟨defun hackforis⟩
⟨defun hackforis1⟩
⟨defun hasAplExtension⟩
⟨defun hasFormalMapVariable⟩

⟨defun initialize-prepare⟩
⟨defun initial-substring⟩
⟨defun initial-substring-p⟩

⟨defun Line-New-Line⟩

⟨defun make-string-adjustable⟩
⟨defun make-symbol-of⟩
⟨defun match-advance-string⟩
⟨defun match-current-token⟩
⟨defun match-next-token⟩
⟨defun match-string⟩
⟨defun match-token⟩
⟨defun meta-syntax-error⟩
⟨defun modifyModeStack⟩

⟨defun next-char⟩
⟨defun next-line⟩
⟨defun next-token⟩
⟨defun ncINTERPFILE⟩

⟨defun optional⟩

⟨defun PARSE-AnyId⟩
⟨defun PARSE-Application⟩
⟨defun PARSE-Category⟩
⟨defun PARSE-Command⟩
⟨defun PARSE-CommandTail⟩
⟨defun PARSE-Conditional⟩
⟨defun PARSE-Data⟩
⟨defun PARSE-ElseClause⟩
⟨defun PARSE-Enclosure⟩
⟨defun PARSE-Exit⟩
⟨defun PARSE-Expr⟩
⟨defun PARSE-Expression⟩
⟨defun PARSE-Float⟩
⟨defun PARSE-FloatBase⟩
⟨defun PARSE-FloatBasePart⟩

```

```

⟨defun PARSE-FloatExponent⟩
⟨defun PARSE-FloatTok⟩
⟨defun PARSE-Form⟩
⟨defun PARSE-FormalParameter⟩
⟨defun PARSE-FormalParameterTok⟩
⟨defun PARSE-getSemanticForm⟩
⟨defun PARSE-GlyphTok⟩
⟨defun PARSE-Import⟩
⟨defun PARSE-Infix⟩
⟨defun PARSE-InfixWith⟩
⟨defun PARSE-IntegerTok⟩
⟨defun PARSE-Iterator⟩
⟨defun PARSE-IteratorTail⟩
⟨defun PARSE-Label⟩
⟨defun PARSE-LabelExpr⟩
⟨defun PARSE-Leave⟩
⟨defun PARSE-LedPart⟩
⟨defun PARSE-leftBindingPowerOf⟩
⟨defun PARSE-Loop⟩
⟨defun PARSE-Name⟩
⟨defun PARSE-NBGlyphTok⟩
⟨defun PARSE-NewExpr⟩
⟨defun PARSE-NudPart⟩
⟨defun PARSE-OpenBrace⟩
⟨defun PARSE-OpenBracket⟩
⟨defun PARSE-Operation⟩
⟨defun PARSE-Option⟩
⟨defun PARSE-Prefix⟩
⟨defun PARSE-Primary⟩
⟨defun PARSE-Primary1⟩
⟨defun PARSE-PrimaryNoFloat⟩
⟨defun PARSE-PrimaryOrQM⟩
⟨defun PARSE-Qualification⟩
⟨defun PARSE-Quad⟩
⟨defun PARSE-Reduction⟩
⟨defun PARSE-ReductionOp⟩
⟨defun PARSE-Return⟩
⟨defun PARSE-rightBindingPowerOf⟩
⟨defun PARSE-ScriptItem⟩
⟨defun PARSE-Scripts⟩
⟨defun PARSE-Seg⟩
⟨defun PARSE-Selector⟩
⟨defun PARSE-SemiColon⟩
⟨defun PARSE-Sequence⟩
⟨defun PARSE-Sequence1⟩
⟨defun PARSE-Sexpr⟩

```



```

⟨defun PARSE-Sexpr1⟩
⟨defun PARSE-SpecialCommand⟩
⟨defun PARSE-SpecialKeyWord⟩
⟨defun PARSE-Statement⟩
⟨defun PARSE-String⟩
⟨defun PARSE-Suffix⟩
⟨defun PARSE-TokenCommandTail⟩
⟨defun PARSE-TokenList⟩
⟨defun PARSE-TokenOption⟩
⟨defun PARSE-TokTail⟩
⟨defun PARSE-VarForm⟩
⟨defun PARSE-With⟩
⟨defun parsepiles⟩
⟨defun parseAnd⟩
⟨defun parseAtom⟩
⟨defun parseAtSign⟩
⟨defun parseCategory⟩
⟨defun parseCoerce⟩
⟨defun parseColon⟩
⟨defun parseConstruct⟩
⟨defun parseDEF⟩
⟨defun parseDollarGreaterEqual⟩
⟨defun parseDollarGreaterThan⟩
⟨defun parseDollarLessEqual⟩
⟨defun parseDollarNotEqual⟩
⟨defun parseEquivalence⟩
⟨defun parseExit⟩
⟨defun postForm⟩
⟨defun parseGreaterEqual⟩
⟨defun parseGreaterThan⟩
⟨defun parseHas⟩
⟨defun parseIf⟩
⟨defun parseIf,ifTran⟩
⟨defun parseImplies⟩
⟨defun parseIn⟩
⟨defun parseInBy⟩
⟨defun parseIs⟩
⟨defun parseIsnt⟩
⟨defun parseJoin⟩
⟨defun parseLeave⟩
⟨defun parseLessEqual⟩
⟨defun parseLET⟩
⟨defun parseLETD⟩
⟨defun parseMDEF⟩
⟨defun parseNot⟩
⟨defun parseNotEqual⟩

```

```

⟨defun parseOr⟩
⟨defun parsePretend⟩
⟨defun parseReturn⟩
⟨defun parseSegment⟩
⟨defun parseSeq⟩
⟨defun parseTran⟩
⟨defun parseTranList⟩
⟨defun parseTransform⟩
⟨defun parseVCONS⟩
⟨defun parseWhere⟩
⟨defun postAdd⟩
⟨defun postAtom⟩
⟨defun postAtSign⟩
⟨defun postBigFloat⟩
⟨defun postBlock⟩
⟨defun postCategory⟩
⟨defun postcheck⟩
⟨defun postCollect⟩
⟨defun postCollect,finish⟩
⟨defun postColon⟩
⟨defun postColonColon⟩
⟨defun postComma⟩
⟨defun postConstruct⟩
⟨defun postDef⟩
⟨defun postError⟩
⟨defun postExit⟩
⟨defun postIf⟩
⟨defun postin⟩
⟨defun postIn⟩
⟨defun postJoin⟩
⟨defun postMapping⟩
⟨defun postMDef⟩
⟨defun postOp⟩
⟨defun postPretend⟩
⟨defun postQUOTE⟩
⟨defun postReduce⟩
⟨defun postRepeat⟩
⟨defun postScripts⟩
⟨defun postScriptsForm⟩
⟨defun postSemiColon⟩
⟨defun postSignature⟩
⟨defun postSlash⟩
⟨defun postTran⟩
⟨defun postTranList⟩
⟨defun postTranScripts⟩
⟨defun postTransform⟩

```

```
<defun postTransformCheck>
<defun postTuple>
<defun postTupleCollect>
<defun postWhere>
<defun postWith>
<defun preparse>
<defun preparse1>
<defun preparse-echo>
<defun preparseReadLine>
<defun preparseReadLine1>
<defun primitiveType>
<defun push-reduction>

<defun quote-if-string>

<defun read-a-line>
<defun recompile-lib-file-if-necessary>
<defun /rf-1>
<defun /RQ,LIB>

<defun setDefOp>
<defun spad>
<defun spad-fixed-arg>
<defun storeblanks>
<defun s-process>

<defun try-get-token>

<defun underscore>
<defun unget-tokens>
<defun unTuple>

<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>

Chapter 9

Index

Index

- *comp370-apply*
 - usedby spad, 264
- *eof*
 - usedby read-a-line, 31
 - usedby spad, 264
- *lisp-bin-filetype*
 - usedby recompile-lib-file-if-necessary, 306
- *package*
 - usedby def-string, 174
- *standard-output*
 - usedby compileFileQuietly, 307
- +ι, 99
 - defplist, 99
- ., 126
 - defplist, 126
- ι, 134
 - defplist, 134
- /, 140
 - defplist, 140
- /RQ,LIB, 254
 - calls /rf-1(5), 254
 - uses \$lisplib, 254
 - uses echo-meta(5), 254
 - defun, 254
- /editfile
 - usedby /rf-1, 255
 - usedby compAdd, 69
 - usedby compileSpad2Cmd, 251
 - usedby compiler, 247
 - usedby spad, 264
- /rf(5)
 - calledby compilerDoit, 253
- /rf-1, 255
 - calls makeInputFilename(5), 255
 - calls ncINTERPFILE, 255
 - calls spad, 255
 - uses /editfile, 255
 - uses echo-meta, 255
 - defun, 255
 - /rf-1(5)
 - calledby /RQ,LIB, 254
 - /rq(5)
 - calledby compilerDoit, 253
 - /tracelet-print
 - calledby def-is2, 167
- :, 42, 78, 125
 - defplist, 42, 78, 125
- ::, 41, 77, 126
 - defplist, 41, 77, 126
- :BF:, 121
 - defplist, 121
- ;, 139
 - defplist, 139
- ι=, 59
 - defplist, 59
- ==, 129
 - defplist, 129
- ==ι, 135
 - defplist, 135
- =ι, 131
 - defplist, 131
- ι, 47
 - defplist, 47
- ι=, 46, 47
 - defplist, 46, 47
- \$Boolean
 - usedby compCase1, 74
 - usedby compIf, 95
 - usedby compIs, 97
 - usedby compReduce1, 105
 - usedby compSymbol, 283
- \$CategoryFrame

- usedby compWithMappingModel1, 294
- \$Category
 - usedby compConstructorCategory, 85
 - usedby compDefine1, 88
 - usedby compJoin, 98
- \$ConstructorNames
 - usedby compDefine1, 88
- \$DomainFrame
 - usedby s-process, 267
- \$DoubleFloat
 - usedby primitiveType, 282
- \$EchoLineStack
 - usedby prepare-echo, 30
 - usedby prepareReadLine1, 29
- \$EchoLines
 - usedby ncINTERPFILE, 304
- \$EmptyEnvironment
 - usedby s-process, 267
- \$EmptyMode
 - usedby compAdd, 69
 - usedby compArgumentsAndTryA-
gain, 291
 - usedby compCase1, 74
 - usedby compColonInside, 279
 - usedby compCons1, 84
 - usedby compDefine1, 88
 - usedby compForm1, 286
 - usedby compForm2, 289
 - usedby compIs, 97
 - usedby compMacro, 102
 - usedby compNoStacking, 274
 - usedby compPretend, 103
 - usedby compWhere, 110
 - usedby compWithMappingModel1, 294
 - usedby primitiveType, 282
 - usedby s-process, 267
- \$EmptyVector
 - usedby compVector, 109
- \$Expression
 - usedby compAtom, 280
 - usedby compForm1, 286
 - usedby compSymbol, 283
- \$FormalMapVariableList
 - usedby compColon, 80
 - usedby compSymbol, 283
 - usedby compTypeOf, 278
 - usedby compWithMappingModel1, 294
- \$Index
 - usedby s-process, 267
- \$InitialDomainsInScope
 - usedby spad, 264
- \$InteractiveFrame
 - usedby spad, 264
- \$InteractiveMode
 - usedby compileFileQuietly, 307
 - usedby compileSpad2Cmd, 251
 - usedby dollarTran, 238
 - usedby parseAnd, 40
 - usedby parseAtSign, 41
 - usedby parseCoerce, 42
 - usedby parseColon, 42
 - usedby parseIf,ifTran, 51
 - usedby parseNot, 62
 - usedby postBigFloat, 122
 - usedby postDef, 130
 - usedby postError, 117
 - usedby postMDef, 136
 - usedby postReduce, 138
 - usedby spad, 264
- \$Line
 - usedby Line-New-Line, 32
- \$LocalFrame
 - usedby s-process, 268
- \$NRTaddForm
 - usedby compAdd, 69
- \$NRTderivedTargetIfTrue
 - usedby compTopLevel, 270
- \$NegativeInteger
 - usedby primitiveType, 282
- \$NoValueMode
 - usedby compDefine1, 88
 - usedby compImport, 96
 - usedby compMacro, 102
 - usedby compSeq1, 108
 - usedby compSymbol, 283
- \$NoValue
 - usedby compSymbol, 283
 - usedby parseAtom, 37

- \$NonNegativeInteger
 - usedby primitiveType, 282
- \$NumberOfArgsIfInteger
 - usedby compForm1, 286
- \$One
 - usedby compElt, 91
- \$OpAssoc
 - usedby def-inner, 175
- \$PolyMode
 - usedby s-process, 267
- \$PositiveInteger
 - usedby primitiveType, 282
- \$QuickCode
 - usedby compWithMappingModel, 294
 - usedby compileSpad2Cmd, 250
- \$QuickLet
 - usedby compileSpad2Cmd, 250
- \$ReadingFile
 - usedby ncINTERPFILE, 304
- \$Representation
 - usedby compNoStacking, 274
- \$String
 - usedby primitiveType, 282
- \$Symbol
 - usedby compSymbol, 283
- \$Translation
 - usedby s-process, 268
- \$TriangleVariableList
 - usedby compForm2, 289
- \$VariableCount
 - usedby s-process, 267
- \$Zero
 - usedby compElt, 91
- \$addFormLhs
 - usedby compAdd, 69
- \$addForm
 - usedby compAdd, 69
 - usedby compCapsuleInner, 72
- \$body
 - usedby def-addlet, 174
 - usedby def-inner, 175
 - usedby def, 149
- \$bootStrapMode
 - usedby comp2, 275
 - usedby compAdd, 69
- usedby compCapsule, 71
 - usedby compColon, 80
- \$boot
 - usedby PARSE-FloatTok, 225
 - usedby PARSE-Primary1, 204
 - usedby PARSE-Quad, 210
 - usedby PARSE-Selector, 202
 - usedby PARSE-TokTail, 198
 - usedby aplTran1, 144
 - usedby aplTran, 143
 - usedby postAtom, 114
 - usedby postBigFloat, 122
 - usedby postColonColon, 126
 - usedby postDef, 130
 - usedby postForm, 118
 - usedby postIf, 132
 - usedby postMDef, 136
 - usedby quote-if-string, 229
- \$byConstructors, 307
 - usedby compilerDoit, 253
 - usedby prepare1, 23
 - defvar, 307
- \$comblocklist
 - usedby prepare, 18
- \$compErrorMessageStack
 - usedby compOrCroak1, 272
- \$compForModelIfTrue
 - usedby compSymbol, 283
- \$compStack
 - usedby compNoStacking1, 274
 - usedby compNoStacking, 274
 - usedby compOrCroak1, 272
 - usedby comp, 273
- \$compTimeSum
 - usedby compTopLevel, 270
- \$compUniquelyIfTrue
 - usedby s-process, 267
- \$compileOnlyCertainItems
 - usedby compileSpad2Cmd, 250
- \$constructorLineNumber
 - usedby prepare, 18
- \$constructorsSeen, 307
 - usedby compilerDoit, 253
 - usedby prepare1, 23
 - defvar, 307
- \$currentFunction

- usedby s-process, 267
- `$defOp`
 - usedby parseTransform, 35
 - usedby postError, 117
 - usedby postTransformCheck, 116
 - usedby setDefOp, 143
- `$defstack`, 158
 - usedby def-where, 158
- `$docList`
 - usedby postDef, 130
 - usedby preparse, 18
- `$echolinestack`
 - usedby initialize-preparse, 14
 - usedby preparse1, 23
- `$endTestList`
 - usedby compReduce1, 105
- `$envHashTable`
 - usedby compTopLevel, 270
- `$exitModeStack`
 - usedby compExit, 93
 - usedby compLeave, 101
 - usedby compOrCroak1, 272
 - usedby compSeq1, 108
 - usedby compSeq, 107
 - usedby comp, 273
 - usedby modifyModeStack, 302
 - usedby s-process, 267
- `$exitMode`
 - usedby s-process, 267
- `$e`
 - usedby comp3, 276
 - usedby compHas, 94
 - usedby compReduce1, 105
 - usedby s-process, 267
- `$finalEnv`
 - usedby compSeq1, 108
- `$forceAdd`
 - usedby compTopLevel, 270
- `$formalArgList`
 - usedby compDefine1, 88
 - usedby compReduce, 104
 - usedby compSymbol, 283
 - usedby compWithMappingMode1, 294
 - usedby compWithMappingMode, 291
- `$formalMapVariables`
 - usedby hasFormalMapVariable, 300
- `$formatArgList`
 - usedby compWithMappingMode1, 294
- `$form`
 - usedby compCapsuleInner, 72
 - usedby compDefine1, 88
 - usedby s-process, 267
- `$functorForm`
 - usedby compAdd, 69
 - usedby compCapsule, 71
- `$functorLocalParameters`
 - usedby compCapsuleInner, 72
 - usedby compSymbol, 283
- `$funnameTail`
 - usedby compWithMappingMode1, 294
- `$funname`
 - usedby compWithMappingMode1, 294
- `$f`
 - usedby compileSpad2Cmd, 250
- `$genFVar`
 - usedby s-process, 267
- `$genSDVar`
 - usedby s-process, 267
- `$genno`
 - usedby aplTran, 143
- `$getDomainCode`
 - usedby compCapsuleInner, 72
- `$headerDocumentation`
 - usedby postDef, 130
 - usedby preparse, 18
- `$inDefIS`
 - usedby defIS, 168
 - usedby defLET2, 156
- `$inDefLET`
 - usedby defIS1, 169
 - usedby defLET, 153
- `$index`
 - usedby initialize-preparse, 14
 - usedby preparseReadLine1, 29
 - usedby preparse, 18
- `$initList`
 - usedby compReduce1, 105

- `$insideCapsuleFunctionIfTrue`
 - usedby `compDefine1`, 88
 - usedby `s-process`, 267
- `$insideCategoryIfTrue`
 - usedby `compCapsuleInner`, 72
 - usedby `compColon`, 80
 - usedby `compDefine1`, 88
 - usedby `s-process`, 267
- `$insideCategoryPackageIfTrue`
 - usedby `compCapsuleInner`, 72
- `$insideCoerceInteractiveHardIfTrue`
 - usedby `s-process`, 267
- `$insideCompTypeOf`
 - usedby `comp3`, 276
 - usedby `compTypeOf`, 278
- `$insideConstructIfTrue`
 - usedby `parseColon`, 42
 - usedby `parseConstruct`, 38
- `$insideExpressionIfTrue`
 - usedby `compCapsule`, 71
 - usedby `compColon`, 80
 - usedby `compDefine1`, 88
 - usedby `compExpression`, 285
 - usedby `compMakeDeclaration`, 302
 - usedby `compSeq1`, 108
 - usedby `compWhere`, 110
 - usedby `s-process`, 267
- `$insideFunctorIfTrue`
 - usedby `compColon`, 80
 - usedby `compDefine1`, 88
 - usedby `s-process`, 267
- `$insidePostCategoryIfTrue`
 - usedby `postCategory`, 123
 - usedby `postWith`, 142
- `$insideWhereIfTrue`
 - usedby `compDefine1`, 88
 - usedby `compWhere`, 110
 - usedby `s-process`, 267
- `$is-eqlist`, 166
 - usedby `def-is-eqlist`, 162
 - usedby `def-is-remdup1`, 164
 - usedby `def-is2`, 167
 - defvar, 166
- `$is-gensymlist`
 - usedby `def-is`, 166
- `$is-spill-list`, 161
 - usedby `def-is-eqlist`, 162
 - usedby `def-is2`, 167
 - defvar, 161
- `$is-spill`, 161
 - defvar, 161
- `$isGenVarCounter`
 - usedby `defIS1`, 169
 - usedby `defIS`, 168
- `$killOptimizeIfTrue`
 - usedby `compTopLevel`, 270
 - usedby `compWithMappingMode1`, 294
- `$leaveLevelStack`
 - usedby `compLeave`, 101
 - usedby `s-process`, 267
- `$leaveMode`
 - usedby `s-process`, 267
- `$letGenVarCounter`
 - usedby `defLET1`, 154
 - usedby `defLET2`, 156
 - usedby `defLET`, 153
- `$let`
 - usedby `defIS1`, 169
 - usedby `defLET1`, 154
 - usedby `defLET2`, 156
 - usedby `defLetForm`, 158
- `$level`
 - usedby `compOrCroak1`, 272
- `$lhsOfColon`
 - usedby `compColon`, 80
- `$lhs`
 - usedby `parseDEF`, 43
 - usedby `parseMDEF`, 61
- `$linelist`
 - usedby `initialize-preparse`, 14
 - usedby `preparse1`, 23
 - usedby `preparseReadLine1`, 29
- `$line`
 - usedby `match-advance-string`, 227
- `$lisplib`
 - usedby `/RQ,LIB`, 254
 - usedby `comp2`, 275
- `$macroIfTrue`
 - usedby `compDefine`, 87
 - usedby `compMacro`, 102
- `$macroassoc`

- usedby def-process, 151
- usedby deftran, 150
- usedby s-process, 267
- \$maxSignatureLineNumber
 - usedby postDef, 130
 - usedby preparse, 18
- \$m
 - usedby compileSpad2Cmd, 250
- \$newCompilerUnionFlag
 - usedby compColonInside, 279
 - usedby compPretend, 103
- \$newComp
 - usedby compileSpad2Cmd, 250
- \$newConlist
 - usedby compileSpad2Cmd, 251
 - usedby compiler, 247
- \$newspad
 - usedby s-process, 267
- \$noEnv
 - usedby compColon, 80
- \$noParseCommands
 - usedby PARSE-SpecialCommand, 185
- \$noSubsumption
 - usedby spad, 264
- \$opassoc
 - usedby def-where, 158
 - usedby def, 149
- \$options
 - usedby compileSpad2Cmd, 251
 - usedby compileSpadLispCmd, 305
 - usedby compiler, 247
- \$op
 - usedby compDefine1, 88
 - usedby def-inner, 175
 - usedby def, 149
 - usedby parseDollarGreaterEqual, 44
 - usedby parseDollarGreaterThan, 44
 - usedby parseDollarLessEqual, 45
 - usedby parseDollarNotEqual, 45
 - usedby parseGreaterEqual, 47
 - usedby parseGreaterThan, 47
 - usedby parseLessEqual, 59
 - usedby parseNotEqual, 62
 - usedby parseTran, 36
- \$packagesUsed
 - usedby comp2, 275
 - usedby compAdd, 69
 - usedby compDefine, 87
 - usedby compTopLevel, 270
- \$postStack
 - usedby postError, 117
 - usedby s-process, 267
- \$prefix
 - usedby compDefine1, 88
- \$preparse-last-line
 - usedby initialize-preparse, 14
 - usedby preparse1, 23
 - usedby preparseReadLine1, 29
 - usedby preparse, 18
- \$preparseReportIfTrue
 - usedby preparse, 18
- \$previousTime
 - usedby s-process, 268
- \$reportExitModeStack
 - usedby modifyModeStack, 302
- \$resolveTimeSum
 - usedby compTopLevel, 270
- \$returnMode
 - usedby s-process, 267
- \$scanIfTrue
 - usedby compOrCroak1, 272
 - usedby compileSpad2Cmd, 250
- \$semanticErrorStack
 - usedby s-process, 267
- \$sideEffectsList
 - usedby compReduce1, 105
- \$signature
 - usedby compCapsuleInner, 72
- \$skipme
 - usedby preparse1, 23
 - usedby preparse, 18
- \$sourceFileTypes
 - usedby compileSpad2Cmd, 251
- \$spad
 - usedby quote-if-string, 229
- \$s
 - usedby compOrCroak1, 272
- \$tokenCommands
 - usedby PARSE-SpecialCommand, 185
- \$token

- usedby match-advance-string, 227
- \$top-level
 - usedby s-process, 267
- \$topOp
 - usedby s-process, 267
 - usedby setDefOp, 143
- \$tripleCache
 - usedby compDefine, 87
- \$tripleHits
 - usedby compDefine, 87
- \$until
 - usedby compReduce1, 105
- \$v1, 163
 - usedby def-is-remdup1, 164
 - usedby def-is-remdup, 163
 - defvar, 163
- \$warningStack
 - usedby s-process, 267
- action, 239
 - calledby PARSE-AnyId, 216
 - calledby PARSE-Category, 191
 - calledby PARSE-CommandTail, 187
 - calledby PARSE-Data, 213
 - calledby PARSE-FloatExponent, 208
 - calledby PARSE-GlyphTok, 215
 - calledby PARSE-Infix, 197
 - calledby PARSE-NBGlyphTok, 215
 - calledby PARSE-NewExpr, 183
 - calledby PARSE-OpenBrace, 218
 - calledby PARSE-OpenBracket, 217
 - calledby PARSE-Operation, 195
 - calledby PARSE-Prefix, 197
 - calledby PARSE-ReductionOp, 199
 - calledby PARSE-Sexpr1, 214
 - calledby PARSE-SpecialCommand, 185
 - calledby PARSE-SpecialKeyWord, 184
 - calledby PARSE-Suffix, 220
 - calledby PARSE-TokTail, 198
 - calledby PARSE-TokenCommandTail, 186
 - calledby PARSE-TokenList, 187
 - defun, 239
- add, 120
 - defplist, 120
- add-parens-and-semis-to-line, 27
 - calledby parsepiles, 26
 - calls addclose, 27
 - calls drop, 27
 - calls infxtok, 27
 - calls nonblankloc, 27
 - defun, 27
- addBinding
 - calledby spad, 264
- addCARorCDR, 165
 - calledby defLET2, 156
 - calls eqcar, 165
 - calls qcar, 165
 - calls qcdr, 165
 - defun, 165
- addclose
 - calledby add-parens-and-semis-to-line, 27
- addContour
 - calledby compWhere, 110
- addDomain
 - calledby comp2, 275
 - calledby comp3, 276
 - calledby compAtSign, 71
 - calledby compCapsule, 71
 - calledby compCase, 73
 - calledby compCoerce, 77
 - calledby compColonInside, 279
 - calledby compColon, 79
 - calledby compElt, 91
 - calledby compForm1, 286
 - calledby compImport, 96
 - calledby compPretend, 103
- addEmptyCapsuleIfNecessary
 - calledby compDefine1, 88
- addInformation
 - calledby compCapsuleInner, 72
- advance-token, 235
 - calledby PARSE-AnyId, 216
 - calledby PARSE-FloatExponent, 208
 - calledby PARSE-GlyphTok, 215
 - calledby PARSE-Infix, 197
 - calledby PARSE-NBGlyphTok, 215
 - calledby PARSE-OpenBrace, 218
 - calledby PARSE-OpenBracket, 217

- calledby PARSE-Prefix, 197
- calledby PARSE-ReductionOp, 199
- calledby PARSE-Suffix, 220
- calledby PARSE-TokenList, 187
- calls copy-token, 235
- calls current-token, 235
- calls try-get-token, 235
- uses current-token, 235
- uses valid-tokens, 235
- defun, 235
- and, 40
- defplist, 40
- aplTran, 143
 - calledby postTransform, 111
 - calls aplTran1, 143
 - calls containsBang, 143
 - uses \$boot, 143
 - uses \$genno, 143
 - defun, 143
- aplTran1, 144
 - calledby aplTran1, 144
 - calledby aplTranList, 145
 - calledby aplTran, 143
 - calledby hasAplExtension, 146
 - calls aplTran1, 144
 - calls aplTranList, 144
 - calls hasAplExtension, 144
 - calls nreverse0, 144
 - calls , 144
 - uses \$boot, 144
 - defun, 144
- aplTranList, 145
 - calledby aplTran1, 144
 - calledby aplTranList, 145
 - calls aplTran1, 145
 - calls aplTranList, 145
 - defun, 145
- applyMapping
 - calledby comp3, 276
- argsToSig, 301
 - calledby compLambda, 100
 - defun, 301
- assoc
 - calledby compColon, 79
 - calledby compForm2, 289
- assq
 - calledby freelist, 303
- atEndOfLine
 - calledby PARSE-TokenCommandTail, 186
- augModemapsFromDomain1
 - calledby compForm1, 286
- b-mdef
 - calledby def-process, 151
- Bang, 239
 - defmacro, 239
- bang
 - calledby PARSE-Category, 191
 - calledby PARSE-CommandTail, 187
 - calledby PARSE-Conditional, 223
 - calledby PARSE-Form, 200
 - calledby PARSE-Import, 193
 - calledby PARSE-IteratorTail, 218
 - calledby PARSE-Seg, 223
 - calledby PARSE-Sexpr1, 214
 - calledby PARSE-SpecialCommand, 185
 - calledby PARSE-TokenCommandTail, 186
- bfp-
 - calledby PARSE-FloatTok, 225
- Block, 122
 - defplist, 122
- bootStrapError
 - calledby compCapsule, 71
- bootTransform
 - calledby def, 149
- bright
 - calledby parseInBy, 56
 - calledby postForm, 118
- browserAutoloadOnceTrigger
 - calledby compileSpad2Cmd, 250
- bumperrorcount
 - calledby postError, 117
- canReturn
 - calledby compIf, 95
- capsule, 71
 - defplist, 71
- case, 72
 - defplist, 72

- catches
 - compOrCroak1, 272
 - preparse1, 23
 - spad, 264
- category, 41, 76, 123
 - defplist, 41, 76, 123
- char-eq, 236
 - calledby PARSE-FloatBase, 206
 - calledby PARSE-TokTail, 198
 - defun, 236
- char-ne, 237
 - calledby PARSE-FloatBase, 206
 - calledby PARSE-Selector, 202
 - defun, 237
- chaseInferences
 - calledby compHas, 94
- coerce
 - calledby compAtSign, 71
 - calledby compCase, 73
 - calledby compCoerce1, 78
 - calledby compCoerce, 77
 - calledby compColonInside, 279
 - calledby compForm1, 286
 - calledby compHas, 94
 - calledby compIf, 95
 - calledby compIs, 97
 - calledby convert, 281
- coerceable
 - calledby compForm1, 286
- coerceByModemap
 - calledby compCoerce1, 78
- collect, 124
 - calledby floatexpid, 238
 - defplist, 124
- comma2Tuple, 127
 - calledby postComma, 127
 - calledby postConstruct, 128
 - calls postFlatten, 127
 - defun, 127
- comp, 273
 - calledby compAdd, 69
 - calledby compArgumentsAndTryA-
gain, 291
 - calledby compAtSign, 71
 - calledby compCase1, 74
 - calledby compCoerce1, 78
 - calledby compColonInside, 279
 - calledby compCons1, 84
 - calledby compExit, 93
 - calledby compForm1, 286
 - calledby compIs, 97
 - calledby compLeave, 101
 - calledby compList, 284
 - calledby compOrCroak1, 272
 - calledby compPretend, 103
 - calledby compReduce1, 105
 - calledby compSeqItem, 108
 - calledby compVector, 109
 - calledby compWhere, 110
 - calledby compWithMappingModel1,
294
 - calledby def-inner, 175
 - calledby def, 149
 - calls compNoStacking, 273
 - uses \$compStack, 273
 - uses \$exitModeStack, 273
 - defun, 273
- comp-tran
 - calledby compWithMappingModel1,
294
- comp2, 275
 - calledby compNoStacking1, 274
 - calledby compNoStacking, 274
 - calls addDomain, 275
 - calls comp3, 275
 - calls insert, 275
 - calls isDomainForm, 275
 - calls isFunctor, 275
 - calls nequal, 275
 - calls opOf, 275
 - uses \$bootStrapMode, 275
 - uses \$lisplib, 275
 - uses \$packagesUsed, 275
 - defun, 275
- comp3, 276
 - calledby comp2, 275
 - calledby compTypeOf, 278
 - calls addDomain, 276
 - calls applyMapping, 276
 - calls compApply, 276
 - calls compAtom, 276
 - calls compCoerce, 276

- calls compColon, 276
- calls compExpression, 276
- calls compTypeOf, 276
- calls compWithMappingMode, 276
- calls getDomainsInScope, 276
- calls getmode, 276
- calls member, 276
- calls pname, 276
- calls stringPrefix?, 276
- calls stringimage, 276
- uses \$e, 276
- uses \$insideCompTypeOf, 276
- defun, 276
- compAdd, 69
 - calls NRTgetLocalIndex, 69
 - calls compCapsule, 69
 - calls compOrCroak, 69
 - calls compSubDomain1, 69
 - calls compTuple2Record, 69
 - calls comp, 69
 - calls nreverse0, 69
 - calls pairp, 69
 - calls qcar, 69
 - calls qcdr, 69
 - uses /editfile, 69
 - uses \$EmptyMode, 69
 - uses \$NRTaddForm, 69
 - uses \$addFormLhs, 69
 - uses \$addForm, 69
 - uses \$bootStrapMode, 69
 - uses \$functorForm, 69
 - uses \$packagesUsed, 69
 - defun, 69
- compApply
 - calledby comp3, 276
- compArgumentsAndTryAgain, 291
 - calledby compForm, 285
 - calls compForm1, 291
 - calls comp, 291
 - uses \$EmptyMode, 291
 - defun, 291
- compAtom, 280
 - calledby comp3, 276
 - calls compAtomWithModemap, 280
 - calls compList, 280
 - calls compSymbol, 280
 - calls compVector, 280
 - calls convert, 280
 - calls get, 280
 - calls isSymbol, 280
 - calls modelsAggregateOf, 280
 - calls primitiveType, 280
 - uses \$Expression, 280
 - defun, 280
- compAtomWithModemap
 - calledby compAtom, 280
- compAtSign, 71
 - calledby compLambda, 100
 - calls addDomain, 71
 - calls coerce, 71
 - calls comp, 71
 - defun, 71
- compBoolean
 - calledby compIf, 95
- compCapsule, 71
 - calledby compAdd, 69
 - calls addDomain, 71
 - calls bootStrapError, 71
 - calls compCapsuleInner, 71
 - uses \$bootStrapMode, 71
 - uses \$functorForm, 71
 - uses \$insideExpressionIfTrue, 71
 - uses editfile, 71
 - defun, 71
- compCapsuleInner, 72
 - calledby compCapsule, 71
 - calls addInformation, 72
 - calls compCapsuleItems, 72
 - calls mkpf, 72
 - calls processFunctorOrPackage, 72
 - uses \$addForm, 72
 - uses \$form, 72
 - uses \$functorLocalParameters, 72
 - uses \$getDomainCode, 72
 - uses \$insideCategoryIfTrue, 72
 - uses \$insideCategoryPackageIfTrue, 72
 - uses \$signature, 72
 - defun, 72
- compCapsuleItems
 - calledby compCapsuleInner, 72
- compCase, 73

- calls addDomain, 73
- calls coerce, 73
- calls compCase1, 73
- defun, 73
- compCase1, 74
 - calledby compCase, 73
 - calls comp, 74
 - calls getModemapList, 74
 - calls modeEqual, 74
 - calls nreverse0, 74
 - uses \$Boolean, 74
 - uses \$EmptyMode, 74
 - defun, 74
- compCat, 75
 - calls getl, 75
 - defun, 75
- compCategory, 76
 - calls compCategoryItem, 76
 - calls mkExplicitCategoryFunction, 76
 - calls qcar, 76
 - calls qcdr, 76
 - calls resolve, 76
 - calls systemErrorHere, 76
 - defun, 76
- compCategoryItem
 - calledby compCategory, 76
- compCoerce, 77
 - calledby comp3, 276
 - calls addDomain, 77
 - calls coerce, 77
 - calls compCoerce1, 77
 - calls getmode, 77
 - defun, 77
- compCoerce1, 78
 - calledby compCoerce, 77
 - calls coerceByModemap, 78
 - calls coerce, 78
 - calls comp, 78
 - calls mkq, 78
 - calls msubst, 78
 - calls resolve, 78
 - defun, 78
- compColon, 79
 - calledby comp3, 276
 - calledby compColon, 79
 - calledby compMakeDeclaration, 302
 - calls addDomain, 79
 - calls assoc, 79
 - calls compColonInside, 79
 - calls compColon, 79
 - calls eqsubstlist, 79
 - calls genSomeVariable, 80
 - calls getDomainsInScope, 79
 - calls getmode, 79
 - calls isCategoryForm, 79
 - calls isDomainForm, 79
 - calls length, 79
 - calls makeCategoryForm, 80
 - calls member, 79
 - calls nreverse0, 79
 - calls put, 80
 - calls systemErrorHere, 80
 - calls take, 79
 - calls unknownTypeError, 79
 - uses \$FormalMapVariableList, 80
 - uses \$bootStrapMode, 80
 - uses \$insideCategoryIfTrue, 80
 - uses \$insideExpressionIfTrue, 80
 - uses \$insideFunctorIfTrue, 80
 - uses \$lhsOfColon, 80
 - uses \$noEnv, 80
 - defun, 79
- compColonInside, 279
 - calledby compColon, 79
 - calls addDomain, 279
 - calls coerce, 279
 - calls comp, 279
 - calls opOf, 279
 - calls stackSemanticError, 279
 - calls stackWarning, 279
 - uses \$EmptyMode, 279
 - uses \$newCompilerUnionFlag, 279
 - defun, 279
- compCons, 83
 - calls compCons1, 83
 - calls compForm, 83
 - defun, 83
- compCons1, 84
 - calledby compCons, 83
 - calls comp, 84
 - calls convert, 84

- calls pairp, 84
- calls qcar, 84
- calls qcdr, 84
- uses \$EmptyMode, 84
- defun, 84
- compConstruct, 86
 - calls compForm, 86
 - calls compList, 86
 - calls compVector, 86
 - calls convert, 86
 - calls getDomainsInScope, 86
 - calls modeIsAggregateOf, 86
 - defun, 86
- compConstructorCategory, 85
 - calls resolve, 85
 - uses \$Category, 85
 - defun, 85
- compDefine, 87
 - calls compDefine1, 87
 - uses \$macroIfTrue, 87
 - uses \$packagesUsed, 87
 - uses \$tripleCache, 87
 - uses \$tripleHits, 87
 - defun, 87
- compDefine1, 88
 - calledby compDefine1, 88
 - calledby compDefine, 87
 - calls addEmptyCapsuleIfNecessary, 88
 - calls compDefWhereClause, 88
 - calls compDefine1, 88
 - calls compDefineAddSignature, 88
 - calls compDefineCapsuleFunction, 88
 - calls compDefineCategory, 88
 - calls compDefineFunctor, 88
 - calls compInternalFunction, 88
 - calls getAbbreviation, 88
 - calls getSignatureFromMode, 88
 - calls getTargetFromRhs, 88
 - calls giveFormalParametersValues, 88
 - calls isDomainForm, 88
 - calls isMacro, 88
 - calls length, 88
 - calls macroExpand, 88
 - calls stackAndThrow, 88
 - calls strconc, 88
 - uses \$Category, 88
 - uses \$ConstructorNames, 88
 - uses \$EmptyMode, 88
 - uses \$NoValueMode, 88
 - uses \$formalArgList, 88
 - uses \$form, 88
 - uses \$insideCapsuleFunctionIfTrue, 88
 - uses \$insideCategoryIfTrue, 88
 - uses \$insideExpressionIfTrue, 88
 - uses \$insideFunctorIfTrue, 88
 - uses \$insideWhereIfTrue, 88
 - uses \$op, 88
 - uses \$prefix, 88
 - defun, 88
- compDefineAddSignature
 - calledby compDefine1, 88
- compDefineCapsuleFunction
 - calledby compDefine1, 88
- compDefineCategory
 - calledby compDefine1, 88
- compDefineFunctor
 - calledby compDefine1, 88
- compDefWhereClause
 - calledby compDefine1, 88
- compElt, 91
 - calls addDomain, 91
 - calls compForm, 91
 - calls convert, 91
 - calls getDeltaEntry, 91
 - calls getModemapListFromDomain, 91
 - calls isDomainForm, 91
 - calls length, 91
 - calls nequal, 91
 - calls opOf, 91
 - calls stackMessage, 91
 - calls stackWarning, 91
 - uses \$One, 91
 - uses \$Zero, 91
 - defun, 91
- compExit, 93
 - calls comp, 93
 - calls modifyModeStack, 93

- calls `stackMessageIfNone`, 93
- uses `$exitModeStack`, 93
- defun, 93
- `compExpression`, 285
 - calledby `comp3`, 276
 - calls `compForm`, 285
 - calls `getl`, 285
 - uses `$insideExpressionIfTrue`, 285
 - defun, 285
- `compExpressionList`
 - calledby `compForm1`, 286
- `compfluidize`
 - calledby `def-addlet`, 174
- `compForm`, 285
 - calledby `compConstruct`, 86
 - calledby `compCons`, 83
 - calledby `compElt`, 91
 - calledby `compExpression`, 285
 - calls `compArgumentsAndTryAgain`, 285
 - calls `compForm1`, 285
 - calls `stackMessageIfNone`, 285
 - defun, 285
- `compForm1`, 286
 - calledby `compArgumentsAndTryAgain`, 291
 - calledby `compForm`, 285
 - calls `addDomain`, 286
 - calls `augModemapsFromDomain1`, 286
 - calls `coerceable`, 286
 - calls `coerce`, 286
 - calls `compExpressionList`, 286
 - calls `compForm2`, 286
 - calls `compOrCroak`, 286
 - calls `compToApply`, 286
 - calls `comp`, 286
 - calls `getFormModemaps`, 286
 - calls `length`, 286
 - calls `nreverse0`, 286
 - calls `outputComp`, 286
 - uses `$EmptyMode`, 286
 - uses `$Expression`, 286
 - uses `$NumberOfArgsIfInteger`, 286
 - defun, 286
- `compForm2`, 289
 - calledby `compForm1`, 286
 - calls `PredImplies`, 289
 - calls `assoc`, 289
 - calls `compForm3`, 289
 - calls `compFormPartiallyBottomUp`, 289
 - calls `compUniquely`, 289
 - calls `isSimple`, 289
 - calls `length`, 289
 - calls `nreverse0`, 289
 - calls `sublis`, 289
 - calls `take`, 289
 - uses `$EmptyMode`, 289
 - uses `$TriangleVariableList`, 289
 - defun, 289
- `compForm3`
 - calledby `compForm2`, 289
- `compForMode`
 - calledby `compJoin`, 98
- `compFormPartiallyBottomUp`
 - calledby `compForm2`, 289
- `compFromIf`
 - calledby `compIf`, 95
- `compHas`, 94
 - calls `chaseInferences`, 94
 - calls `coerce`, 94
 - calls `compHasFormat`, 94
 - uses `$e`, 94
 - defun, 94
- `compHasFormat`
 - calledby `compHas`, 94
- `compIf`, 95
 - calls `canReturn`, 95
 - calls `coerce`, 95
 - calls `compBoolean`, 95
 - calls `compFromIf`, 95
 - calls `intersectionEnvironment`, 95
 - calls `quotify`, 95
 - calls `resolve`, 95
 - uses `$Boolean`, 95
 - defun, 95
- `compile-lib-file`, 307
 - calledby `recompile-lib-file-if-necessary`, 306
 - defun, 307
- `compileFileQuietly`, 307

- uses `*standard-output*`, 307
 - uses `$InteractiveMode`, 307
 - defun, 307
- compiler, 246
 - calls `compileSpad2Cmd`, 247
 - calls `compileSpadLispCmd`, 247
 - calls `findfile`, 247
 - calls `helpSpad2Cmd(5)`, 247
 - calls `mergePathnames(5)`, 247
 - calls `namestring(5)`, 247
 - calls `pathname(5)`, 247
 - calls `pathnameType(5)`, 247
 - calls `selectOptionLC(5)`, 247
 - calls `throwKeyedMsg`, 247
 - uses `/editfile`, 247
 - uses `$newConlist`, 247
 - uses `$options`, 247
 - defun, 246
- compilerDoit, 253
 - calledby `compileSpad2Cmd`, 250
 - calls `/rf(5)`, 253
 - calls `/rq(5)`, 253
 - calls `member(5)`, 253
 - calls `sayBrightly`, 253
 - uses `$byConstructors`, 253
 - uses `$constructorsSeen`, 253
 - defun, 253
- compilerDoitWithScreenedLisplib
 - calledby `compileSpad2Cmd`, 250
- compileSpad2Cmd, 249
 - calledby compiler, 247
 - calls `browserAutoloadOnceTrigger`, 250
 - calls `compilerDoitWithScreenedLisplib`, 250
 - calls `compilerDoit`, 250
 - calls `convertSpadToAsFile`, 250
 - calls `error`, 250
 - calls `extendLocalLibdb`, 250
 - calls `namestring(5)`, 250
 - calls `nequal`, 250
 - calls `object2String`, 250
 - calls `oldParserAutoloadOnceTrigger`, 250
 - calls `pathname(5)`, 250
 - calls `pathnameType(5)`, 250
 - calls `sayKeyedMsg`, 250
 - calls `selectOptionLC(5)`, 250
 - calls `spad2AsTranslatorAutoloadOnceTrigger`, 250
 - calls `spadPrompt`, 250
 - calls `strconc`, 250
 - calls `terminateSystemCommand(5)`, 250
 - calls `throwKeyedMsg`, 250
 - calls `updateSourceFiles(5)`, 250
 - uses `/editfile`, 251
 - uses `$InteractiveMode`, 251
 - uses `$QuickCode`, 250
 - uses `$QuickLet`, 250
 - uses `$compileOnlyCertainItems`, 250
 - uses `$f`, 250
 - uses `$m`, 250
 - uses `$newComp`, 250
 - uses `$newConlist`, 251
 - uses `$options`, 251
 - uses `$scanIfTrue`, 250
 - uses `$sourceFileTypes`, 251
 - defun, 249
- compileSpadLispCmd, 305
 - calledby compiler, 247
 - calls `fnameMake(5)`, 305
 - calls `fnameReadable?(5)`, 305
 - calls `localdatabase(5)`, 305
 - calls `namestring(5)`, 305
 - calls `object2String`, 305
 - calls `pathname(5)`, 305
 - calls `pathnameDirectory(5)`, 305
 - calls `pathnameName(5)`, 305
 - calls `pathnameType(5)`, 305
 - calls `recompile-lib-file-if-necessary`, 305
 - calls `sayKeyedMsg`, 305
 - calls `selectOptionLC(5)`, 305
 - calls `spadPrompt`, 305
 - calls `terminateSystemCommand(5)`, 305
 - calls `throwKeyedMsg`, 305
 - uses `$options`, 305
 - defun, 305
- compImport, 96
 - calls `addDomain`, 96

- uses \$NoValueMode, 96
 - defun, 96
- compInternalFunction
 - calledby compDefine1, 88
- compIs, 97
 - calls coerce, 97
 - calls comp, 97
 - uses \$Boolean, 97
 - uses \$EmptyMode, 97
 - defun, 97
- compIterator
 - calledby compReduce1, 105
- compJoin, 98
 - calls compForMode, 98
 - calls compJoin, getParms, 98
 - calls convert, 98
 - calls isCategoryForm, 98
 - calls nreverse0, 98
 - calls pairp, 98
 - calls qcar, 98
 - calls qcdr, 98
 - calls stackSemanticError, 98
 - calls union, 98
 - calls wrapDomainSub, 98
 - uses \$Category, 98
 - defun, 98
- compJoin, getParms
 - calledby compJoin, 98
- compLambda, 100
 - calledby compWithMappingModel1, 294
 - calls argsToSig, 100
 - calls compAtSign, 100
 - calls qcar, 100
 - calls qcdr, 100
 - calls stackAndThrow, 100
 - defun, 100
- compLeave, 101
 - calls comp, 101
 - calls modifyModeStack, 101
 - uses \$exitModeStack, 101
 - uses \$leaveLevelStack, 101
 - defun, 101
- compList, 284
 - calledby compAtom, 280
 - calledby compConstruct, 86
 - calls comp, 284
 - defun, 284
- compMacro, 102
 - calls formatUnabbreviated, 102
 - calls macroExpand, 102
 - calls put, 102
 - calls qcar, 102
 - calls sayBrightly, 102
 - uses \$EmptyMode, 102
 - uses \$NoValueMode, 102
 - uses \$macroIfTrue, 102
 - defun, 102
- compMakeDeclaration, 302
 - calledby compWithMappingModel1, 294
 - calls compColon, 302
 - uses \$insideExpressionIfTrue, 302
 - defun, 302
- compNoStacking, 274
 - calledby comp, 273
 - calls comp2, 274
 - calls compNoStacking1, 274
 - uses \$EmptyMode, 274
 - uses \$Representation, 274
 - uses \$compStack, 274
 - defun, 274
- compNoStacking1, 274
 - calledby compNoStacking, 274
 - calls comp2, 274
 - calls get, 274
 - uses \$compStack, 274
 - defun, 274
- compOrCroak, 271
 - calledby compAdd, 69
 - calledby compForm1, 286
 - calledby compTopLevel, 270
 - calls compOrCroak1, 271
 - defun, 271
- compOrCroak1, 272
 - calledby compOrCroak, 271
 - calls compOrCroak1, compactify, 272
 - calls comp, 272
 - calls displayComp, 272
 - calls displaySemanticErrors, 272
 - calls mkErrorExpr, 272
 - calls say, 272

- calls stackSemanticError, 272
- calls userError, 272
- uses \$compErrorMessageStack, 272
- uses \$compStack, 272
- uses \$exitModeStack, 272
- uses \$level, 272
- uses \$scanIfTrue, 272
- uses \$s, 272
- catches, 272
- defun, 272
- compOrCroak1,compactify, 304
 - calledby compOrCroak1,compactify, 304
 - calledby compOrCroak1, 272
 - calls compOrCroak1,compactify, 304
 - calls lassoc, 304
 - defun, 304
- compPretend, 103
 - calls addDomain, 103
 - calls comp, 103
 - calls nequal, 103
 - calls opOf, 103
 - calls stackSemanticError, 103
 - calls stackWarning, 103
 - uses \$EmptyMode, 103
 - uses \$newCompilerUnionFlag, 103
 - defun, 103
- compQuote, 104
 - defun, 104
- compReduce, 104
 - calls compReduce1, 104
 - uses \$formalArgList, 104
 - defun, 104
- compReduce1, 105
 - calledby compReduce, 104
 - calls compIterator, 105
 - calls comp, 105
 - calls getIdentity, 105
 - calls msubst, 105
 - calls nreverse0, 105
 - calls parseTran, 105
 - calls systemError, 105
 - uses \$Boolean, 105
 - uses \$endTestList, 105
 - uses \$e, 105
 - uses \$initList, 105
 - uses \$sideEffectsList, 105
 - uses \$until, 105
 - defun, 105
- compSeq, 107
 - calls compSeq1, 107
 - uses \$exitModeStack, 107
 - defun, 107
- compSeq1, 108
 - calledby compSeq, 107
 - calls compSeqItem, 108
 - calls mkq, 108
 - calls nreverse0, 108
 - calls replaceExitEtc, 108
 - uses \$NoValueMode, 108
 - uses \$exitModeStack, 108
 - uses \$finalEnv, 108
 - uses \$insideExpressionIfTrue, 108
 - defun, 108
- compSeqItem, 108
 - calledby compSeq1, 108
 - calls comp, 108
 - calls macroExpand, 108
 - defun, 108
- compSubDomain1
 - calledby compAdd, 69
- compSymbol, 283
 - calledby compAtom, 280
 - calls NRTgetLocalIndex, 283
 - calls errorRef, 283
 - calls getmode, 283
 - calls get, 283
 - calls isFunction, 283
 - calls member, 283
 - calls stackMessage, 283
 - uses \$Boolean, 283
 - uses \$Expression, 283
 - uses \$FormalMapVariableList, 283
 - uses \$NoValueMode, 283
 - uses \$NoValue, 283
 - uses \$Symbol, 283
 - uses \$compForModeIfTrue, 283
 - uses \$formalArgList, 283
 - uses \$functorLocalParameters, 283
 - defun, 283
- compToApply
 - calledby compForm1, 286

- compTopLevel, 270
 - calledby s-process, 267
 - calls compOrCroak, 270
 - calls newComp, 270
 - uses \$NRTderivedTargetIfTrue, 270
 - uses \$compTimeSum, 270
 - uses \$envHashTable, 270
 - uses \$forceAdd, 270
 - uses \$killOptimizeIfTrue, 270
 - uses \$packagesUsed, 270
 - uses \$resolveTimeSum, 270
 - defun, 270
- compTuple2Record
 - calledby compAdd, 69
- compTypeOf, 278
 - calledby comp3, 276
 - calls comp3, 278
 - calls eqsubstlist, 278
 - calls get, 278
 - calls put, 278
 - uses \$FormalMapVariableList, 278
 - uses \$insideCompTypeOf, 278
 - defun, 278
- compUniquely
 - calledby compForm2, 289
- compVector, 109
 - calledby compAtom, 280
 - calledby compConstruct, 86
 - calls comp, 109
 - uses \$EmptyVector, 109
 - defun, 109
- compWhere, 110
 - calls addContour, 110
 - calls comp, 110
 - calls deltaContour, 110
 - calls macroExpand, 110
 - uses \$EmptyMode, 110
 - uses \$insideExpressionIfTrue, 110
 - uses \$insideWhereIfTrue, 110
 - defun, 110
- compWithMappingMode, 291
 - calledby comp3, 276
 - calls compWithMappingMode1, 291
 - uses \$formalArgList, 291
 - defun, 291
- compWithMappingMode1, 292
 - calledby compWithMappingMode, 291
 - calls comp-tran, 294
 - calls compLambda, 294
 - calls compMakeDeclaration, 294
 - calls comp, 294
 - calls extendsCategoryForm, 294
 - calls extractCodeAndConstructTriple, 294
 - calls freelist, 294
 - calls get, 294
 - calls hasFormalMapVariable, 294
 - calls isFunctor, 294
 - calls optimizeFunctionDef, 294
 - calls qcar, 294
 - calls qcdr, 294
 - calls stackAndThrow, 294
 - calls take, 294
 - uses \$CategoryFrame, 294
 - uses \$EmptyMode, 294
 - uses \$FormalMapVariableList, 294
 - uses \$QuickCode, 294
 - uses \$formalArgList, 294
 - uses \$formatArgList, 294
 - uses \$funnameTail, 294
 - uses \$funname, 294
 - uses \$killOptimizeIfTrue, 294
 - defun, 292
- cons, 82
 - defplist, 82
- construct, 37, 86, 127
 - defplist, 37, 86, 127
- contained
 - calledby defLET1, 154
 - calledby parseCategory, 41
- containsBang, 147
 - calledby aplTran, 143
 - calledby containsBang, 147
 - calls containsBang, 147
 - defun, 147
- convert, 281
 - calledby compAtom, 280
 - calledby compCons1, 84
 - calledby compConstruct, 86
 - calledby compElt, 91
 - calledby compJoin, 98

- calls coerce, 281
- calls resolve, 281
- defun, 281
- convertSpadToAsFile
 - calledby compileSpad2Cmd, 250
- copy
 - calledby modifyModeStack, 302
- copy-token
 - calledby PARSE-TokTail, 198
 - calledby advance-token, 235
- curoutstream
 - usedby s-process, 268
- current-char, 236
 - calledby PARSE-FloatBasePart, 207
 - calledby PARSE-FloatBase, 206
 - calledby PARSE-FloatExponent, 208
 - calledby PARSE-Selector, 202
 - calledby PARSE-TokTail, 198
 - calledby match-string, 226
 - calls line-current-char, 236
 - calls line-past-end-p, 236
 - uses current-line, 236
 - defun, 236
- current-line
 - usedby PARSE-Category, 191
 - usedby current-char, 236
 - usedby next-char, 236
- current-symbol, 233
 - calledby PARSE-AnyId, 216
 - calledby PARSE-ElseClause, 224
 - calledby PARSE-FloatBase, 206
 - calledby PARSE-FloatExponent, 208
 - calledby PARSE-Infix, 197
 - calledby PARSE-NewExpr, 183
 - calledby PARSE-OpenBrace, 218
 - calledby PARSE-OpenBracket, 217
 - calledby PARSE-Operation, 195
 - calledby PARSE-Prefix, 197
 - calledby PARSE-Primary1, 204
 - calledby PARSE-ReductionOp, 199
 - calledby PARSE-Selector, 202
 - calledby PARSE-SpecialCommand, 185
 - calledby PARSE-SpecialKeyWord, 184
 - calledby PARSE-Suffix, 220
 - calledby PARSE-TokTail, 198
 - calledby PARSE-TokenList, 187
 - calls current-token, 233
 - calls make-symbol-of, 233
 - defun, 233
- current-token, 233
 - calledby PARSE-FloatBasePart, 207
 - calledby PARSE-SpecialKeyWord, 184
 - calledby advance-token, 235
 - calledby current-symbol, 233
 - calledby match-advance-string, 227
 - calledby match-current-token, 232
 - calledby next-token, 234
 - calls try-get-token, 233
 - usedby advance-token, 235
 - usedby current-token, 233
 - uses current-token, 233
 - uses valid-tokens, 233
 - defun, 233
- curstrm
 - calledby s-process, 267
- dcq
 - calledby def-is2, 167
 - calledby preparseReadLine, 28
- decodeScripts, 148
 - calledby decodeScripts, 148
 - calledby getScriptName, 147
 - calls decodeScripts, 148
 - calls qcar, 148
 - calls qcdr, 148
 - calls strconc, 148
 - calls stringimage, 148
 - defun, 148
- deepestExpression, 146
 - calledby deepestExpression, 146
 - calledby hasAplExtension, 146
 - calls deepestExpression, 146
 - defun, 146
- def, 43, 87, 149
 - calledby def-process, 151
 - calls bootTransform, 149
 - calls comp, 149
 - calls def-insert-let, 149
 - calls def-stringtoquote, 149

- calls deftran, 149
- calls sublis, 149
- uses \$body, 149
- uses \$opassoc, 149
- uses \$op, 149
- defplist, 43, 87
- defun, 149
- def-addlet, 174
 - calledby def-stringtoquote, 174
 - calls compfluidize, 174
 - calls def-let, 174
 - calls mkprogn, 174
 - uses \$body, 174
 - defun, 174
- def-collect, 172
 - calls def-it, 172
 - calls deftran, 172
 - calls hackforis, 172
 - defun, 172
- def-cond, 161
 - calledby def-cond, 161
 - calls def-cond, 161
 - calls deftran, 161
 - defun, 161
- def-in2on, 160
 - calledby def-it, 173
 - calls eqcar, 160
 - defun, 160
- def-inner, 175
 - calledby def-where, 158
 - calls comp, 175
 - calls def-insert-let, 175
 - calls def-stringtoquote, 175
 - calls sublis, 175
 - uses \$OpAssoc, 175
 - uses \$body, 175
 - uses \$op, 175
 - defun, 175
- def-insert-let, 152
 - calledby def-inner, 175
 - calledby def-insert-let, 152
 - calledby def, 149
 - calls def-insert-let, 152
 - calls def-let, 152
 - calls deftran, 152
 - calls errhuh, 152
- defun, 152
- def-is, 166
 - calledby defIS1, 169
 - calls def-is2, 166
 - uses Initial-Gensym, 166
 - uses \$is-gensymlist, 166
 - defun, 166
- def-is-eqlist, 162
 - calledby def-is2, 167
 - calls , 162
 - uses \$is-eqlist, 162
 - uses \$is-spill-list, 162
 - defun, 162
- def-is-remdup, 163
 - calledby def-is2, 167
 - calls def-is-remdup1, 163
 - uses \$vl, 163
 - defun, 163
- def-is-remdup1, 164
 - calledby def-is-remdup1, 164
 - calledby def-is-remdup, 163
 - calls def-is-remdup1, 164
 - calls eqcar, 164
 - calls errhuh, 164
 - calls is-gensym, 164
 - uses \$is-eqlist, 164
 - uses \$vl, 164
 - defun, 164
- def-is-rev, 171
 - calledby def-is-rev, 171
 - calls def-is-rev, 171
 - calls errhuh, 171
 - defun, 171
- def-is2, 167
 - calledby def-is, 166
 - calls /tracelet-print, 167
 - calls dcq, 167
 - calls def-is-eqlist, 167
 - calls def-is-remdup, 167
 - calls eqcar, 167
 - calls listofatoms, 167
 - calls mkpf, 167
 - calls moan, 167
 - calls subst, 167
 - uses \$is-eqlist, 167
 - uses \$is-spill-list, 167

- defun, 167
- def-it, 173
 - calledby def-collect, 172
 - calledby def-repeat, 173
 - calls def-in2on, 173
 - calls def-let, 173
 - calls deftran, 173
 - calls errhuh, 173
 - calls reset, 173
 - defun, 173
- def-let, 153
 - calledby def-addlet, 174
 - calledby def-insert-let, 152
 - calledby def-it, 173
 - calls defLET, 153
 - calls deftran, 153
 - defun, 153
- def-message, 159
 - calls def-message1, 159
 - defun, 159
- def-message1, 160
 - calledby def-message1, 160
 - calledby def-message, 159
 - calls def-message1, 160
 - calls deftran, 160
 - calls eqcar, 160
 - defun, 160
- def-process, 151
 - calledby def-process, 151
 - calledby s-process, 267
 - calls b-mdef, 151
 - calls def-process, 151
 - calls deftran, 151
 - calls def, 151
 - calls eqcar, 151
 - calls is-console, 151
 - calls print-full, 151
 - calls say, 151
 - uses \$macroassoc, 151
 - defun, 151
- def-rename, 151
 - calledby s-process, 267
 - calls def-rename1, 151
 - defun, 151
- def-rename1, 152
 - calledby def-rename1, 152
 - calledby def-rename, 151
 - calls def-rename1, 152
 - defun, 152
- def-repeat, 173
 - calls def-it, 173
 - calls deftran, 173
 - calls hackforis, 173
 - defun, 173
- def-string, 174
 - calls deftran, 174
 - uses *package*, 174
 - defun, 174
- def-stringtoquote, 174
 - calledby def-inner, 175
 - calledby def-stringtoquote, 174
 - calledby def, 149
 - calls def-addlet, 174
 - calls def-stringtoquote, 174
 - defun, 174
- def-where, 158
 - calls def-inner, 158
 - calls def-whereclauselist, 158
 - calls deftran, 158
 - calls mkprogn, 158
 - calls sublis, 158
 - uses \$defstack, 158
 - uses \$opassoc, 158
 - defun, 158
- def-whereclause, 159
 - calledby def-whereclauselist, 159
 - calledby def-whereclause, 159
 - calls def-whereclause, 159
 - calls eqcar, 159
 - calls whdef, 159
 - defun, 159
- def-whereclauselist, 159
 - calledby def-where, 158
 - calls def-whereclause, 159
 - calls deftran, 159
 - defun, 159
- definition-name, 183
 - usedby PARSE-NewExpr, 183
 - defvar, 183
- defIS, 168
 - calledby defLET2, 156
 - calls defIS1, 168

- calls deftran, 168
- uses \$inDefIS, 168
- uses \$isGenVarCounter, 168
- defun, 168
- defIS1, 169
 - calledby defIS1, 169
 - calledby defIS, 168
 - calledby defLET2, 156
 - calls def-is, 169
 - calls defIS1, 169
 - calls defISReverse, 169
 - calls defLET1, 169
 - calls defLET, 169
 - calls defLetForm, 169
 - calls mkprogn, 169
 - calls qcar, 169
 - calls qcdr, 169
 - calls say, 169
 - calls strconc, 169
 - calls stringimage, 169
 - uses \$inDefLET, 169
 - uses \$isGenVarCounter, 169
 - uses \$let, 169
 - defun, 169
- defISReverse, 172
 - calledby defIS1, 169
 - calledby defISReverse, 172
 - calledby defLET2, 156
 - calls defISReverse, 172
 - calls errhuh, 172
 - defun, 172
- defLET, 153
 - calledby def-let, 153
 - calledby defIS1, 169
 - calls defLET1, 153
 - uses \$inDefLET, 153
 - uses \$letGenVarCounter, 153
 - defun, 153
- defLET1, 154
 - calledby defIS1, 169
 - calledby defLET1, 154
 - calledby defLET, 153
 - calls contained, 154
 - calls defLET1, 154
 - calls defLET2, 154
 - calls defLetForm, 154
- calls identp, 154
- calls mkprogn, 154
- calls strconc, 154
- calls stringimage, 154
- uses \$letGenVarCounter, 154
- uses \$let, 154
- defun, 154
- defLET2, 156
 - calledby defLET1, 154
 - calledby defLET2, 156
 - calls addCARorCDR, 156
 - calls defIS1, 156
 - calls defISReverse, 156
 - calls defIS, 156
 - calls defLET2, 156
 - calls defLetForm, 156
 - calls identp, 156
 - calls qcar, 156
 - calls qcdr, 156
 - calls strconc, 156
 - calls stringimage, 156
 - uses \$inDefIS, 156
 - uses \$letGenVarCounter, 156
 - uses \$let, 156
 - defun, 156
- defLetForm, 158
 - calledby defIS1, 169
 - calledby defLET1, 154
 - calledby defLET2, 156
 - uses \$let, 158
 - defun, 158
- defmacro
 - Bang, 239
 - must, 239
 - star, 240
- defplist, 40, 68, 70, 121
 - +i, 99
 - ., 126
 - i, 134
 - /, 140
 - ., 42, 78, 125
 - ::, 41, 77, 126
 - :BF:, 121
 - ;, 139
 - j=, 59
 - ==, 129

- `==i`, 135
- `=i`, 131
- `i`, 47
- `i=`, 46, 47
- `add`, 120
- `and`, 40
- `Block`, 122
- `capsule`, 71
- `case`, 72
- `category`, 41, 76, 123
- `collect`, 124
- `cons`, 82
- `construct`, 37, 86, 127
- `def`, 43, 87
- `dollargreaterequal`, 44
- `dollargreaterthan`, 43
- `dollarlessequal`, 44
- `dollarnotequal`, 45
- `elt`, 90
- `eqv`, 45
- `exit`, 92
- `has`, 48, 93
- `if`, 53, 94, 132
- `implies`, 53
- `import`, 96
- `In`, 133
- `in`, 54, 132
- `inby`, 56
- `is`, 56, 96
- `isnt`, 57
- `Join`, 57, 97, 133
- `leave`, 58, 101
- `let`, 59
- `letd`, 60
- `ListCategory`, 85
- `Mapping`, 75
- `mdef`, 60, 101
- `not`, 61
- `notequal`, 62
- `or`, 62
- `pretend`, 63, 103, 137
- `quote`, 104, 137
- `Record`, 74
- `RecordCategory`, 85
- `reduce`, 104, 137
- `repeat`, 138
- `return`, 64
- `Scripts`, 139
- `segment`, 65
- `seq`, 107
- `Signature`, 139
- `TupleCollect`, 141
- `Union`, 75
- `UnionCategory`, 85
- `vcons`, 66
- `vector`, 109
- `VectorCategory`, 85
- `where`, 66, 109, 141
- `with`, 142
- `defstruct`
 - `Line`, 32
- `deftran`, 150
 - `calledby def-collect`, 172
 - `calledby def-cond`, 161
 - `calledby def-insert-let`, 152
 - `calledby def-it`, 173
 - `calledby def-let`, 153
 - `calledby def-message1`, 160
 - `calledby def-process`, 151
 - `calledby def-repeat`, 173
 - `calledby def-string`, 174
 - `calledby def-whereclauselist`, 159
 - `calledby def-where`, 158
 - `calledby defIS`, 168
 - `calledby def`, 149
 - `calls`, 150
 - `uses $macroassoc`, 150
 - `defun`, 150
- `defun`
 - `/RQ,LIB`, 254
 - `/rf-1`, 255
 - `action`, 239
 - `add-parens-and-semis-to-line`, 27
 - `addCARorCDR`, 165
 - `advance-token`, 235
 - `aplTran`, 143
 - `aplTran1`, 144
 - `aplTranList`, 145
 - `argsToSig`, 301
 - `char-eq`, 236
 - `char-ne`, 237
 - `comma2Tuple`, 127

- comp, 273
- comp2, 275
- comp3, 276
- compAdd, 69
- compArgumentsAndTryAgain, 291
- compAtom, 280
- compAtSign, 71
- compCapsule, 71
- compCapsuleInner, 72
- compCase, 73
- compCase1, 74
- compCat, 75
- compCategory, 76
- compCoerce, 77
- compCoerce1, 78
- compColon, 79
- compColonInside, 279
- compCons, 83
- compCons1, 84
- compConstruct, 86
- compConstructorCategory, 85
- compDefine, 87
- compDefine1, 88
- compElt, 91
- compExit, 93
- compExpression, 285
- compForm, 285
- compForm1, 286
- compForm2, 289
- compHas, 94
- compIf, 95
- compile-lib-file, 307
- compileFileQuietly, 307
- compiler, 246
- compilerDoit, 253
- compileSpad2Cmd, 249
- compileSpadLispCmd, 305
- compImport, 96
- compIs, 97
- compJoin, 98
- compLambda, 100
- compLeave, 101
- compList, 284
- compMacro, 102
- compMakeDeclaration, 302
- compNoStacking, 274
- compNoStacking1, 274
- compOrCroak, 271
- compOrCroak1, 272
- compOrCroak1,compactify, 304
- compPretend, 103
- compQuote, 104
- compReduce, 104
- compReduce1, 105
- compSeq, 107
- compSeq1, 108
- compSeqItem, 108
- compSymbol, 283
- compTopLevel, 270
- compTypeOf, 278
- compVector, 109
- compWhere, 110
- compWithMappingMode, 291
- compWithMappingMode1, 292
- containsBang, 147
- convert, 281
- current-char, 236
- current-symbol, 233
- current-token, 233
- decodeScripts, 148
- deepestExpression, 146
- def, 149
- def-addlet, 174
- def-collect, 172
- def-cond, 161
- def-in2on, 160
- def-inner, 175
- def-insert-let, 152
- def-is, 166
- def-is-eqlist, 162
- def-is-remdup, 163
- def-is-remdup1, 164
- def-is-rev, 171
- def-is2, 167
- def-it, 173
- def-let, 153
- def-message, 159
- def-message1, 160
- def-process, 151
- def-rename, 151
- def-rename1, 152
- def-repeat, 173

- def-string, 174
- def-stringtoquote, 174
- def-where, 158
- def-whereclause, 159
- def-whereclauselist, 159
- defIS, 168
- defIS1, 169
- defISReverse, 172
- defLET, 153
- defLET1, 154
- defLET2, 156
- defLetForm, 158
- deftran, 150
- dollarTran, 238
- errhuh, 176
- escape-keywords, 230
- extractCodeAndConstructTriple, 300
- floatexpid, 238
- freelist, 303
- get-a-line, 33
- get-token, 235
- getScriptName, 147
- getToken, 230
- hackforis, 175
- hackforis1, 175
- hasAplExtension, 146
- hasFormalMapVariable, 300
- initial-substring, 33
- initial-substring-p, 228
- initialize-prepare, 14
- Line-New-Line, 32
- make-string-adjustable, 34
- make-symbol-of, 233
- match-advance-string, 227
- match-current-token, 232
- match-next-token, 232
- match-string, 226
- match-token, 232
- meta-syntax-error, 237
- modifyModeStack, 302
- ncINTERPFILE, 304
- next-char, 236
- next-line, 33
- next-token, 234
- optional, 240
- PARSE-AnyId, 216
- PARSE-Application, 201
- PARSE-Category, 191
- PARSE-Command, 184
- PARSE-CommandTail, 187
- PARSE-Conditional, 223
- PARSE-Data, 213
- PARSE-ElseClause, 224
- PARSE-Enclosure, 209
- PARSE-Exit, 222
- PARSE-Expr, 193
- PARSE-Expression, 192
- PARSE-Float, 205
- PARSE-FloatBase, 206
- PARSE-FloatBasePart, 207
- PARSE-FloatExponent, 208
- PARSE-FloatTok, 225
- PARSE-Form, 200
- PARSE-FormalParameter, 209
- PARSE-FormalParameterTok, 210
- PARSE-getSemanticForm, 196
- PARSE-GlyphTok, 215
- PARSE-Import, 193
- PARSE-Infix, 197
- PARSE-InfixWith, 189
- PARSE-IntegerTok, 209
- PARSE-Iterator, 219
- PARSE-IteratorTail, 218
- PARSE-Label, 201
- PARSE-LabelExpr, 225
- PARSE-Leave, 222
- PARSE-LedPart, 194
- PARSE-leftBindingPowerOf, 195
- PARSE-Loop, 224
- PARSE-Name, 212
- PARSE-NBGlyphTok, 215
- PARSE-NewExpr, 183
- PARSE-NudPart, 194
- PARSE-OpenBrace, 218
- PARSE-OpenBracket, 217
- PARSE-Operation, 195
- PARSE-Option, 188
- PARSE-Prefix, 197
- PARSE-Primary, 203
- PARSE-Primary1, 204
- PARSE-PrimaryNoFloat, 203
- PARSE-PrimaryOrQM, 188

- PARSE-Quad, 210
- PARSE-Qualification, 198
- PARSE-Reduction, 199
- PARSE-ReductionOp, 199
- PARSE-Return, 221
- PARSE-rightBindingPowerOf, 196
- PARSE-ScriptItem, 212
- PARSE-Scripts, 211
- PARSE-Seg, 223
- PARSE-Selector, 202
- PARSE-SemiColon, 221
- PARSE-Sequence, 216
- PARSE-Sequence1, 217
- PARSE-Sexpr, 213
- PARSE-Sexpr1, 214
- PARSE-SpecialCommand, 185
- PARSE-SpecialKeyword, 184
- PARSE-Statement, 189
- PARSE-String, 210
- PARSE-Suffix, 220
- PARSE-TokenCommandTail, 186
- PARSE-TokenList, 187
- PARSE-TokenOption, 186
- PARSE-TokTail, 198
- PARSE-VarForm, 211
- PARSE-With, 190
- parseAnd, 40
- parseAtom, 37
- parseAtSign, 41
- parseCategory, 41
- parseCoerce, 42
- parseColon, 42
- parseConstruct, 38
- parseDEF, 43
- parseDollarGreaterEqual, 44
- parseDollarGreaterThan, 44
- parseDollarLessEqual, 45
- parseDollarNotEqual, 45
- parseEquivalence, 46
- parseExit, 46
- parseGreaterEqual, 47
- parseGreaterThan, 47
- parseIf, 53
- parseIf,ifTran, 51
- parseImplies, 54
- parseIn, 55
- parseInBy, 56
- parseIs, 57
- parseIsnt, 57
- parseJoin, 58
- parseLeave, 59
- parseLessEqual, 59
- parseLET, 60
- parseLETD, 60
- parseMDEF, 61
- parseNot, 62
- parseNotEqual, 62
- parseOr, 63
- parsepiles, 26
- parsePretend, 64
- parseReturn, 64
- parseSegment, 65
- parseSeq, 65
- parseTran, 36
- parseTranList, 37
- parseTransform, 35
- parseVCONS, 66
- parseWhere, 66
- postAdd, 121
- postAtom, 114
- postAtSign, 121
- postBigFloat, 122
- postBlock, 122
- postCategory, 123
- postcheck, 116
- postCollect, 125
- postCollect,finish, 124
- postColon, 126
- postColonColon, 126
- postComma, 127
- postConstruct, 128
- postDef, 130
- postError, 117
- postExit, 131
- postForm, 118
- postIf, 132
- postIn, 133
- postin, 133
- postJoin, 134
- postMapping, 134
- postMDef, 136
- postOp, 114

- postPretend, 137
- postQUOTE, 137
- postReduce, 138
- postRepeat, 138
- postScripts, 139
- postScriptsForm, 115
- postSemiColon, 139
- postSignature, 140
- postSlash, 140
- postTran, 113
- postTranList, 114
- postTranScripts, 115
- postTransform, 111
- postTransformCheck, 116
- postTuple, 141
- postTupleCollect, 141
- postWhere, 142
- postWith, 142
- preparse, 18
- preparse-echo, 30
- preparse1, 23
- preparseReadLine, 28
- preparseReadLine1, 29
- primitiveType, 282
- push-reduction, 241
- quote-if-string, 229
- read-a-line, 31
- recompile-lib-file-if-necessary, 306
- s-process, 267
- setDefOp, 143
- spad, 264
- spad-fixed-arg, 306
- storeblanks, 33
- try-get-token, 234
- underscore, 230
- unget-tokens, 231
- unTuple, 176
- defvar
 - \$byConstructors, 307
 - \$constructorsSeen, 307
 - \$defstack, 158
 - \$is-eqlist, 166
 - \$is-spill-list, 161
 - \$is-spill, 161
 - \$vl, 163
 - definition-name, 183
 - lablasoc, 183
 - meta-error-handler, 237
 - ParseMode, 182
 - tmptok, 182
 - tok, 182
 - XTokenReader, 235
- deltaContour
 - calledby compWhere, 110
- digitp
 - calledby PARSE-FloatBasePart, 207
 - calledby PARSE-FloatBase, 206
 - calledby floatexpid, 238
- displayComp
 - calledby compOrCroak1, 272
- displayPreCompilationErrors
 - calledby s-process, 267
- displaySemanticErrors
 - calledby compOrCroak1, 272
 - calledby s-process, 267
- dollargreaterequal, 44
 - defplist, 44
- dollargreaterthan, 43
 - defplist, 43
- dollarlessequal, 44
 - defplist, 44
- dollarnotequal, 45
 - defplist, 45
- dollarTran, 238
 - calledby PARSE-Qualification, 198
 - uses \$InteractiveMode, 238
 - defun, 238
- doSystemCommand
 - calledby preparse1, 23
- drop
 - calledby add-parens-and-semis-to-line, 27
- Echo-Meta
 - usedby preparse-echo, 30
- echo-meta
 - usedby /rf-1, 255
 - usedby spad, 264
- echo-meta(5)
 - usedby /RQ,LIB, 254
- editfile
 - usedby compCapsule, 71

- elemn
 - calledby PARSE-Operation, 195
 - calledby PARSE-leftBindingPowerOf, 195
 - calledby PARSE-rightBindingPowerOf, 196
- elt, 90
 - defplist, 90
- eqcar
 - calledby PARSE-OpenBrace, 218
 - calledby PARSE-OpenBracket, 217
 - calledby addCARorCDR, 165
 - calledby def-in2on, 160
 - calledby def-is-remdup1, 164
 - calledby def-is2, 167
 - calledby def-message1, 160
 - calledby def-process, 151
 - calledby def-whereclause, 159
 - calledby getToken, 230
 - calledby hackforis1, 175
- eqsubstlist
 - calledby compColon, 79
 - calledby compTypeOf, 278
- eqv, 45
 - defplist, 45
- errhuh, 176
 - calledby def-insert-let, 152
 - calledby def-is-remdup1, 164
 - calledby def-is-rev, 171
 - calledby def-it, 173
 - calledby defISReverse, 172
 - calls systemError, 176
 - defun, 176
- error
 - calledby compileSpad2Cmd, 250
- errorRef
 - calledby compSymbol, 283
- escape-keywords, 230
 - calledby quote-if-string, 229
 - calls , 230
 - defun, 230
- escaped
 - calledby preparse1, 23
- exit, 92
 - defplist, 92
- expand-tabs
 - calledby preparseReadLine1, 29
 - extendLocalLibdb
 - calledby compileSpad2Cmd, 250
 - extendsCategoryForm
 - calledby compWithMappingMode1, 294
 - extractCodeAndConstructTriple, 300
 - calledby compWithMappingMode1, 294
 - defun, 300
 - file-closed
 - usedby spad, 264
 - fincomblock
 - calledby preparse1, 23
 - findfile
 - calledby compiler, 247
 - floatexpid, 238
 - calledby PARSE-FloatExponent, 208
 - calls collect, 238
 - calls digitp, 238
 - calls identp, 238
 - calls maxindex, 238
 - calls pname, 238
 - calls spadreduce, 238
 - calls step, 238
 - defun, 238
 - fnameMake(5)
 - calledby compileSpadLispCmd, 305
 - fnameReadable?(5)
 - calledby compileSpadLispCmd, 305
 - formatUnabbreviated
 - calledby compMacro, 102
 - freelist, 303
 - calledby compWithMappingMode1, 294
 - calledby freelist, 303
 - calls assq, 303
 - calls freelist, 303
 - calls getmode, 303
 - calls identp, 303
 - calls unionq, 303
 - defun, 303
 - genSomeVariable
 - calledby compColon, 80

- genvar
 - calledby hasAplExtension, 146
- get
 - calledby compAtom, 280
 - calledby compNoStacking1, 274
 - calledby compSymbol, 283
 - calledby compTypeOf, 278
 - calledby compWithMappingModel, 294
- get-a-line, 33
 - calledby initialize-prepare, 14
 - calledby prepareReadLine1, 29
 - calls is-console, 33
 - calls make-string-adjustable, 33
 - calls mkprompt, 33
 - calls read-a-line, 33
 - defun, 33
- get-internal-run-time
 - calledby s-process, 267
- get-token, 235
 - calledby try-get-token, 234
 - calls XTokenReader, 235
 - uses XTokenReader, 235
 - defun, 235
- getAbbreviation
 - calledby compDefine1, 88
- getDeltaEntry
 - calledby compElt, 91
- getDomainsInScope
 - calledby comp3, 276
 - calledby compColon, 79
 - calledby compConstruct, 86
- getFormModemaps
 - calledby compForm1, 286
- getfullstr
 - calledby prepare1, 23
- getIdentity
 - calledby compReduce1, 105
- getl
 - calledby PARSE-Operation, 195
 - calledby PARSE-ReductionOp, 199
 - calledby PARSE-leftBindingPowerOf, 195
 - calledby PARSE-rightBindingPowerOf, 196
 - calledby compCat, 75
 - calledby compExpression, 285
 - calledby parseTran, 36
- getmode
 - calledby comp3, 276
 - calledby compCoerce, 77
 - calledby compColon, 79
 - calledby compSymbol, 283
 - calledby freelist, 303
- getModemapList
 - calledby compCase1, 74
- getModemapListFromDomain
 - calledby compElt, 91
- getScriptName, 147
 - calledby postScriptsForm, 115
 - calledby postScripts, 139
 - calls decodeScripts, 147
 - calls identp, 147
 - calls internl, 147
 - calls pname, 147
 - calls postError, 147
 - calls stringimage, 147
 - defun, 147
- getSignatureFromMode
 - calledby compDefine1, 88
- getTargetFromRhs
 - calledby compDefine1, 88
- getToken, 230
 - calledby PARSE-OpenBrace, 218
 - calledby PARSE-OpenBracket, 217
 - calls eqcar, 230
 - defun, 230
- giveFormalParametersValues
 - calledby compDefine1, 88
- hackforis, 175
 - calledby def-collect, 172
 - calledby def-repeat, 173
 - calls hackforis1, 175
 - defun, 175
- hackforis1, 175
 - calledby hackforis, 175
 - calls eqcar, 175
 - calls kar, 175
 - defun, 175
 - has, 48, 93
 - defplist, 48, 93

- hasAplExtension, 146
 - calledby aplTran1, 144
 - calls aplTran1, 146
 - calls deepestExpression, 146
 - calls genvar, 146
 - calls msubst, 146
 - calls nreverse0, 146
 - defun, 146
- hasFormalMapVariable, 300
 - calledby compWithMappingModel1, 294
 - calls ScanOrPairVec, 300
 - uses \$formalMapVariables, 300
 - defun, 300
- helpSpad2Cmd(5)
 - calledby compiler, 247
- identp
 - calledby PARSE-FloatExponent, 208
 - calledby defLET1, 154
 - calledby defLET2, 156
 - calledby floatexpid, 238
 - calledby freelist, 303
 - calledby getScriptName, 147
 - calledby postTransform, 111
- if, 53, 94, 132
 - defplist, 53, 94, 132
- ifcar
 - calledby preparse, 18
- implies, 53
 - defplist, 53
- import, 96
 - defplist, 96
- In, 133
 - defplist, 133
- in, 54, 132
 - defplist, 54, 132
- inby, 56
 - defplist, 56
- incExitLevel
 - calledby parseIf,ifTran, 51
- indent-pos
 - calledby preparse1, 23
- infixtok
 - calledby add-parens-and-semis-to-line, 27
- init-boot/spad-reader
 - calledby spad, 264
- Initial-Gensym
 - usedby def-is, 166
- initial-substring, 33
 - calledby preparseReadLine, 28
 - calls mismatch, 33
 - defun, 33
- initial-substring-p, 228
 - calledby match-string, 226
 - calls string-not-greaterp, 228
 - defun, 228
- initialize-preparse, 14
 - calledby spad, 264
 - calls get-a-line, 14
 - uses \$echolinestack, 14
 - uses \$index, 14
 - uses \$linelist, 14
 - uses \$preparse-last-line, 14
 - defun, 14
- insert
 - calledby comp2, 275
- instring
 - calledby preparse1, 23
- internl
 - calledby getScriptName, 147
 - calledby postForm, 118
- intersectionEnvironment
 - calledby compIf, 95
- ioclear
 - calledby spad, 264
- is, 56, 96
 - defplist, 56, 96
- is-console
 - calledby def-process, 151
 - calledby get-a-line, 33
 - calledby preparse1, 23
- is-gensym
 - calledby def-is-remdup1, 164
- isCategoryForm
 - calledby compColon, 79
 - calledby compJoin, 98
- isDomainForm
 - calledby comp2, 275
 - calledby compColon, 79
 - calledby compDefine1, 88

- calledby compElt, 91
- isFunction
 - calledby compSymbol, 283
- isFunctor
 - calledby comp2, 275
 - calledby compWithMappingModel, 294
- isMacro
 - calledby compDefine1, 88
- isnt, 57
 - defplist, 57
- isSimple
 - calledby compForm2, 289
- isSymbol
 - calledby compAtom, 280
- isTokenDelimiter
 - calledby PARSE-TokenList, 187
- Join, 57, 97, 133
 - defplist, 57, 97, 133
- kar
 - calledby hackforis1, 175
- killColons
 - calledby postSignature, 140
- labasoc
 - usedby PARSE-Data, 213
- lablasoc, 183
 - defvar, 183
- lassoc
 - calledby compOrCroak1, compactify, 304
- last
 - calledby parseSeq, 65
- leave, 58, 101
 - defplist, 58, 101
- length
 - calledby compColon, 79
 - calledby compDefine1, 88
 - calledby compElt, 91
 - calledby compForm1, 286
 - calledby compForm2, 289
 - calledby postScriptsForm, 115
- let, 59
 - defplist, 59
- letd, 60
 - defplist, 60
- Line, 32
 - defstruct, 32
- line
 - usedby match-string, 226
 - usedby spad, 264
- line-at-end-p
 - calledby next-char, 236
- line-buffer
 - calledby match-advance-string, 227
 - calledby match-string, 226
- line-current-char
 - calledby current-char, 236
 - calledby match-advance-string, 227
- line-current-index
 - calledby match-advance-string, 227
 - calledby match-string, 226
- line-current-segment
 - calledby unget-tokens, 231
- Line-New-Line, 32
 - calledby read-a-line, 31
 - uses \$Line, 32
 - defun, 32
- line-new-line
 - calledby unget-tokens, 231
- line-next-char
 - calledby next-char, 236
- line-number
 - calledby PARSE-Category, 191
 - calledby unget-tokens, 231
- line-past-end-p
 - calledby current-char, 236
 - calledby match-advance-string, 227
 - calledby match-string, 226
- ListCategory, 85
 - defplist, 85
- listofatoms
 - calledby def-is2, 167
- localdatabase(5)
 - calledby compileSpadLispCmd, 305
- lt
 - calledby PARSE-Operation, 195
- macroExpand
 - calledby compDefine1, 88

- calledby compMacro, 102
- calledby compSeqItem, 108
- calledby compWhere, 110
- make-float
 - calledby PARSE-Float, 205
- make-reduction
 - calledby push-reduction, 241
- make-string-adjustable, 34
 - calledby get-a-line, 33
 - defun, 34
- make-symbol-of, 233
 - calledby PARSE-Expression, 192
 - calledby current-symbol, 233
 - calls token-symbol, 233
 - defun, 233
- make-token
 - calledby match-advance-string, 227
- makeCategoryForm
 - calledby compColon, 80
- makeInitialModemapFrame
 - calledby spad, 264
- makeInputFilename(5)
 - calledby /rf-1, 255
- makeSimplePredicateOrNil
 - calledby parseIf,ifTran, 51
- mapInto
 - calledby parseSeq, 65
 - calledby parseWhere, 66
- Mapping, 75
 - defplist, 75
- match-advance-string, 227
 - calledby PARSE-Category, 191
 - calledby PARSE-Command, 184
 - calledby PARSE-Conditional, 223
 - calledby PARSE-Enclosure, 209
 - calledby PARSE-Exit, 222
 - calledby PARSE-FloatBasePart, 207
 - calledby PARSE-FloatExponent, 208
 - calledby PARSE-Form, 200
 - calledby PARSE-Import, 193
 - calledby PARSE-IteratorTail, 218
 - calledby PARSE-Iterator, 219
 - calledby PARSE-Label, 201
 - calledby PARSE-Leave, 222
 - calledby PARSE-Loop, 224
 - calledby PARSE-Option, 188
 - calledby PARSE-Primary1, 204
 - calledby PARSE-PrimaryOrQM, 188
 - calledby PARSE-Quad, 210
 - calledby PARSE-Qualification, 198
 - calledby PARSE-Return, 221
 - calledby PARSE-ScriptItem, 212
 - calledby PARSE-Scripts, 211
 - calledby PARSE-Selector, 202
 - calledby PARSE-SemiColon, 221
 - calledby PARSE-Sequence, 216
 - calledby PARSE-Sexpr1, 214
 - calledby PARSE-SpecialCommand, 185
 - calledby PARSE-Statement, 189
 - calledby PARSE-TokenOption, 186
 - calledby PARSE-With, 190
 - calls current-token, 227
 - calls line-buffer, 227
 - calls line-current-char, 227
 - calls line-current-index, 227
 - calls line-past-end-p, 227
 - calls make-token, 227
 - calls match-string, 227
 - calls quote-if-string, 227
 - calls , 227
 - uses \$line, 227
 - uses \$token, 227
 - defun, 227
- match-current-token, 232
 - calledby PARSE-GlyphTok, 215
 - calledby PARSE-NBGlyphTok, 215
 - calledby PARSE-Operation, 195
 - calledby PARSE-SpecialKeyword, 184
 - calls current-token, 232
 - calls match-token, 232
 - defun, 232
- match-next-token, 232
 - calledby PARSE-ReductionOp, 199
 - calls match-token, 232
 - calls next-token, 232
 - defun, 232
- match-string, 226
 - calledby PARSE-AnyId, 216
 - calledby PARSE-NewExpr, 183
 - calledby PARSE-Primary1, 204

- calledby match-advance-string, 227
- calls current-char, 226
- calls initial-substring-p, 226
- calls line-buffer, 226
- calls line-current-index, 226
- calls line-past-end-p, 226
- calls skip-blanks, 226
- calls subseq, 226
- calls unget-tokens, 226
- uses line, 226
- defun, 226
- match-token, 232
 - calledby match-current-token, 232
 - calledby match-next-token, 232
 - calls token-symbol, 232
 - calls token-type, 232
 - defun, 232
- maxindex
 - calledby floatexpid, 238
 - calledby preparse1, 23
 - calledby preparseReadLine1, 29
- mdef, 60, 101
 - defplist, 60, 101
- member
 - calledby comp3, 276
 - calledby compColon, 79
 - calledby compSymbol, 283
- member(5)
 - calledby compilerDoit, 253
- mergePathnames(5)
 - calledby compiler, 247
- meta-error-handler, 237
 - calledby meta-syntax-error, 237
 - usedby meta-syntax-error, 237
 - defvar, 237
- meta-syntax-error, 237
 - calls meta-error-handler, 237
 - uses meta-error-handler, 237
 - defun, 237
- mismatch
 - calledby initial-substring, 33
- mkErrorExpr
 - calledby compOrCroak1, 272
- mkExplicitCategoryFunction
 - calledby compCategory, 76
- mkpf
 - calledby compCapsuleInner, 72
 - calledby def-is2, 167
- mkprogn
 - calledby def-addlet, 174
 - calledby def-where, 158
 - calledby defIS1, 169
 - calledby defLET1, 154
- mkprompt
 - calledby get-a-line, 33
- mkq
 - calledby compCoerce1, 78
 - calledby compSeq1, 108
- moan
 - calledby def-is2, 167
 - calledby parseExit, 46
 - calledby parseReturn, 64
- modeEqual
 - calledby compCase1, 74
- modeIsAggregateOf
 - calledby compAtom, 280
 - calledby compConstruct, 86
- modifyModeStack, 302
 - calledby compExit, 93
 - calledby compLeave, 101
 - calls copy, 302
 - calls resolve, 302
 - calls say, 302
 - calls setelt, 302
 - uses \$exitModeStack, 302
 - uses \$reportExitModeStack, 302
 - defun, 302
- msubst
 - calledby compCoerce1, 78
 - calledby compReduce1, 105
 - calledby hasAplExtension, 146
 - calledby parseDollarGreaterEqual, 44
 - calledby parseDollarGreaterThan, 44
 - calledby parseDollarLessEqual, 45
 - calledby parseDollarNotEqual, 45
 - calledby parseNotEqual, 62
 - calledby parseTransform, 35
- must, 239
 - calledby PARSE-Category, 191
 - calledby PARSE-Command, 184

- calledby PARSE-Conditional, 223
- calledby PARSE-Enclosure, 209
- calledby PARSE-Exit, 222
- calledby PARSE-FloatBasePart, 207
- calledby PARSE-FloatBase, 206
- calledby PARSE-FloatExponent, 208
- calledby PARSE-Float, 205
- calledby PARSE-Form, 200
- calledby PARSE-Import, 193
- calledby PARSE-Infix, 197
- calledby PARSE-Iterator, 219
- calledby PARSE-LabelExpr, 225
- calledby PARSE-Label, 201
- calledby PARSE-Leave, 222
- calledby PARSE-Loop, 224
- calledby PARSE-NewExpr, 183
- calledby PARSE-Option, 188
- calledby PARSE-Prefix, 197
- calledby PARSE-Primary1, 204
- calledby PARSE-Qualification, 198
- calledby PARSE-Reduction, 199
- calledby PARSE-Return, 221
- calledby PARSE-ScriptItem, 212
- calledby PARSE-Scripts, 211
- calledby PARSE-Selector, 202
- calledby PARSE-SemiColon, 221
- calledby PARSE-Sequence, 216
- calledby PARSE-Sexpr1, 214
- calledby PARSE-SpecialCommand, 185
- calledby PARSE-Statement, 189
- calledby PARSE-TokenOption, 186
- calledby PARSE-With, 190
- defmacro, 239
- namestring(5)
 - calledby compileSpad2Cmd, 250
 - calledby compileSpadLispCmd, 305
 - calledby compiler, 247
- ncINTERPFILE, 304
 - calledby /rf-1, 255
 - calls SpadInterpretStream(5), 304
 - uses \$EchoLines, 304
 - uses \$ReadingFile, 304
 - defun, 304
- nequal
 - calledby comp2, 275
 - calledby compElt, 91
 - calledby compPretend, 103
 - calledby compileSpad2Cmd, 250
 - calledby postDef, 130
 - calledby postError, 117
- new2OldLisp
 - calledby s-process, 267
- newComp
 - calledby compTopLevel, 270
- next-char, 236
 - calledby PARSE-FloatBase, 206
 - calls line-at-end-p, 236
 - calls line-next-char, 236
 - uses current-line, 236
 - defun, 236
- next-line, 33
 - defun, 33
- next-token, 234
 - calledby match-next-token, 232
 - calls current-token, 234
 - calls try-get-token, 234
 - usedby next-token, 234
 - uses next-token, 234
 - uses valid-tokens, 234
 - defun, 234
- nonblankloc
 - calledby add-parens-and-semis-to-line, 27
- not, 61
 - defplist, 61
- notequal, 62
 - defplist, 62
- nreverse0
 - calledby aplTran1, 144
 - calledby compAdd, 69
 - calledby compCase1, 74
 - calledby compColon, 79
 - calledby compForm1, 286
 - calledby compForm2, 289
 - calledby compJoin, 98
 - calledby compReduce1, 105
 - calledby compSeq1, 108
 - calledby hasAplExtension, 146
 - calledby postCategory, 123
 - calledby postDef, 130

- calledby postIf, 132
- calledby postMDef, 136
- NRTgetLocalIndex
 - calledby compAdd, 69
 - calledby compSymbol, 283
- nth-stack
 - calledby PARSE-Category, 191
 - calledby PARSE-Sexpr1, 214
- object2String
 - calledby compileSpad2Cmd, 250
 - calledby compileSpadLispCmd, 305
- oldParserAutoloadOnceTrigger
 - calledby compileSpad2Cmd, 250
- opFf
 - calledby parseDEF, 43
- opOf
 - calledby comp2, 275
 - calledby compColonInside, 279
 - calledby compElt, 91
 - calledby compPretend, 103
 - calledby parseLET, 60
 - calledby parseMDEF, 61
- optimizeFunctionDef
 - calledby compWithMappingMode1, 294
- optional, 240
 - calledby PARSE-Application, 201
 - calledby PARSE-Category, 191
 - calledby PARSE-CommandTail, 187
 - calledby PARSE-Conditional, 223
 - calledby PARSE-Expr, 193
 - calledby PARSE-Form, 200
 - calledby PARSE-Import, 193
 - calledby PARSE-Infix, 197
 - calledby PARSE-IteratorTail, 218
 - calledby PARSE-Iterator, 219
 - calledby PARSE-Prefix, 197
 - calledby PARSE-Primary1, 204
 - calledby PARSE-PrimaryNoFloat, 203
 - calledby PARSE-ScriptItem, 212
 - calledby PARSE-Seg, 223
 - calledby PARSE-Sequence1, 217
 - calledby PARSE-Sexpr1, 214
 - calledby PARSE-SpecialCommand, 185
 - calledby PARSE-Statement, 189
 - calledby PARSE-Suffix, 220
 - calledby PARSE-TokenCommandTail, 186
 - calledby PARSE-VarForm, 211
 - defun, 240
- or, 62
 - defplist, 62
- outputComp
 - calledby compForm1, 286
- pack
 - calledby quote-if-string, 229
- pairp
 - calledby compAdd, 69
 - calledby compCons1, 84
 - calledby compJoin, 98
 - calledby postSignature, 140
 - calledby postTran, 113
- PARSE-AnyId, 216
 - calledby PARSE-Sexpr1, 214
- calls action, 216
- calls advance-token, 216
- calls current-symbol, 216
- calls match-string, 216
- calls parse-identifier, 216
- calls parse-keyword, 216
- calls push-reduction, 216
- defun, 216
- PARSE-Application, 201
 - calledby PARSE-Application, 201
 - calledby PARSE-Category, 191
 - calledby PARSE-Form, 200
 - calls PARSE-Application, 201
 - calls PARSE-Primary, 201
 - calls PARSE-Selector, 201
 - calls optional, 201
 - calls pop-stack-1, 201
 - calls pop-stack-2, 201
 - calls push-reduction, 201
 - calls star, 201
 - defun, 201
- parse-argument-designator

- calledby PARSE-FormalParameterTok, defun, 187
 - 210
- PARSE-Category, 191
 - calledby PARSE-Category, 191
 - calls PARSE-Application, 191
 - calls PARSE-Category, 191
 - calls PARSE-Expression, 191
 - calls action, 191
 - calls bang, 191
 - calls line-number, 191
 - calls match-advance-string, 191
 - calls must, 191
 - calls nth-stack, 191
 - calls optional, 191
 - calls pop-stack-1, 191
 - calls pop-stack-2, 191
 - calls pop-stack-3, 191
 - calls push-reduction, 191
 - calls recordAttributeDocumentation, 191
 - calls recordSignatureDocumentation, 191
 - calls star, 191
 - uses current-line, 191
 - defun, 191
- PARSE-Command, 184
 - calls PARSE-SpecialCommand, 184
 - calls PARSE-SpecialKeyWord, 184
 - calls match-advance-string, 184
 - calls must, 184
 - calls push-reduction, 184
 - defun, 184
- PARSE-CommandTail, 187
 - calledby PARSE-CommandTail, 187
 - calledby PARSE-SpecialCommand, 185
 - calls PARSE-CommandTail, 187
 - calls PARSE-Option, 187
 - calls action, 187
 - calls bang, 187
 - calls optional, 187
 - calls pop-stack-1, 187
 - calls pop-stack-2, 187
 - calls push-reduction, 187
 - calls star, 187
 - calls systemCommand, 187
- PARSE-Conditional, 223
 - calledby PARSE-ElseClause, 224
 - calls PARSE-ElseClause, 223
 - calls PARSE-Expression, 223
 - calls bang, 223
 - calls match-advance-string, 223
 - calls must, 223
 - calls optional, 223
 - calls pop-stack-1, 223
 - calls pop-stack-2, 223
 - calls pop-stack-3, 223
 - calls push-reduction, 223
 - defun, 223
- PARSE-Data, 213
 - calledby PARSE-Primary1, 204
 - calls PARSE-Sexpr, 213
 - calls action, 213
 - calls pop-stack-1, 213
 - calls push-reduction, 213
 - calls translabel, 213
 - uses labasoc, 213
 - defun, 213
- PARSE-ElseClause, 224
 - calledby PARSE-Conditional, 223
 - calls PARSE-Conditional, 224
 - calls PARSE-Expression, 224
 - calls current-symbol, 224
 - defun, 224
- PARSE-Enclosure, 209
 - calledby PARSE-Primary1, 204
 - calls PARSE-Expr, 209
 - calls match-advance-string, 209
 - calls must, 209
 - calls pop-stack-1, 209
 - calls push-reduction, 209
 - defun, 209
- PARSE-Exit, 222
 - calls PARSE-Expression, 222
 - calls match-advance-string, 222
 - calls must, 222
 - calls pop-stack-1, 222
 - calls push-reduction, 222
 - defun, 222
- PARSE-Expr, 193
 - calledby PARSE-Enclosure, 209

- calledby PARSE-Expression, 192
- calledby PARSE-Import, 193
- calledby PARSE-Iterator, 219
- calledby PARSE-LabelExpr, 225
- calledby PARSE-Loop, 224
- calledby PARSE-Primary1, 204
- calledby PARSE-Reduction, 199
- calledby PARSE-ScriptItem, 212
- calledby PARSE-SemiColon, 221
- calledby PARSE-Statement, 189
- calls PARSE-LedPart, 193
- calls PARSE-NudPart, 193
- calls optional, 193
- calls pop-stack-1, 193
- calls push-reduction, 193
- calls star, 193
- defun, 193
- PARSE-Expression, 192
 - calledby PARSE-Category, 191
 - calledby PARSE-Conditional, 223
 - calledby PARSE-ElseClause, 224
 - calledby PARSE-Exit, 222
 - calledby PARSE-Infix, 197
 - calledby PARSE-Iterator, 219
 - calledby PARSE-Leave, 222
 - calledby PARSE-Prefix, 197
 - calledby PARSE-Return, 221
 - calledby PARSE-Seg, 223
 - calledby PARSE-Sequence1, 217
 - calledby PARSE-SpecialCommand, 185
 - calls PARSE-Expr, 192
 - calls PARSE-rightBindingPowerOf, 192
 - calls make-symbol-of, 192
 - calls pop-stack-1, 192
 - calls push-reduction, 192
 - uses ParseMode, 192
 - uses prior-token, 192
 - defun, 192
- PARSE-Float, 205
 - calledby PARSE-Selector, 202
 - calls PARSE-FloatBase, 205
 - calls PARSE-FloatExponent, 205
 - calls make-float, 205
 - calls must, 205
- calls pop-stack-1, 205
- calls pop-stack-2, 205
- calls pop-stack-3, 205
- calls pop-stack-4, 205
- calls push-reduction, 205
- defun, 205
- PARSE-FloatBase, 206
 - calledby PARSE-Float, 205
 - calls PARSE-FloatBasePart, 206
 - calls PARSE-IntegerTok, 206
 - calls char-eq, 206
 - calls char-ne, 206
 - calls current-char, 206
 - calls current-symbol, 206
 - calls digitp, 206
 - calls must, 206
 - calls next-char, 206
 - calls push-reduction, 206
 - defun, 206
- PARSE-FloatBasePart, 207
 - calledby PARSE-FloatBase, 206
 - calls PARSE-IntegerTok, 207
 - calls current-char, 207
 - calls current-token, 207
 - calls digitp, 207
 - calls match-advance-string, 207
 - calls must, 207
 - calls push-reduction, 207
 - calls token-nonblank, 207
 - defun, 207
- PARSE-FloatExponent, 208
 - calledby PARSE-Float, 205
 - calls PARSE-IntegerTok, 208
 - calls action, 208
 - calls advance-token, 208
 - calls current-char, 208
 - calls current-symbol, 208
 - calls floatexpid, 208
 - calls identp, 208
 - calls match-advance-string, 208
 - calls must, 208
 - calls push-reduction, 208
 - defun, 208
- PARSE-FloatTok, 225
 - calls bfp-, 225
 - calls parse-number, 225

- calls pop-stack-1, 225
- calls push-reduction, 225
- uses \$boot, 225
- defun, 225
- PARSE-Form, 200
 - calledby PARSE-NudPart, 194
 - calls PARSE-Application, 200
 - calls bang, 200
 - calls match-advance-string, 200
 - calls must, 200
 - calls optional, 200
 - calls pop-stack-1, 200
 - calls push-reduction, 200
 - defun, 200
- PARSE-FormalParameter, 209
 - calledby PARSE-Primary1, 204
 - calls PARSE-FormalParameterTok, 209
 - defun, 209
- PARSE-FormalParameterTok, 210
 - calledby PARSE-FormalParameter, 209
 - calls parse-argument-designator, 210
 - defun, 210
- PARSE-getSemanticForm, 196
 - calledby PARSE-Operation, 195
 - calls PARSE-Infix, 196
 - calls PARSE-Prefix, 196
 - defun, 196
- PARSE-GlyphTok, 215
 - calledby PARSE-Quad, 210
 - calledby PARSE-Seg, 223
 - calledby PARSE-Sexpr1, 214
 - calls action, 215
 - calls advance-token, 215
 - calls match-current-token, 215
 - uses tok, 215
 - defun, 215
- parse-identifier
 - calledby PARSE-AnyId, 216
 - calledby PARSE-Name, 212
- PARSE-Import, 193
 - calls PARSE-Expr, 193
 - calls bang, 193
 - calls match-advance-string, 193
 - calls must, 193
 - calls optional, 193
 - calls pop-stack-1, 193
 - calls pop-stack-2, 193
 - calls push-reduction, 193
 - calls star, 193
 - defun, 193
- PARSE-Infix, 197
 - calledby PARSE-getSemanticForm, 196
 - calls PARSE-Expression, 197
 - calls PARSE-TokTail, 197
 - calls action, 197
 - calls advance-token, 197
 - calls current-symbol, 197
 - calls must, 197
 - calls optional, 197
 - calls pop-stack-1, 197
 - calls pop-stack-2, 197
 - calls push-reduction, 197
 - defun, 197
- PARSE-InfixWith, 189
 - calls PARSE-With, 189
 - calls pop-stack-1, 189
 - calls pop-stack-2, 189
 - calls push-reduction, 189
 - defun, 189
- PARSE-IntegerTok, 209
 - calledby PARSE-FloatBasePart, 207
 - calledby PARSE-FloatBase, 206
 - calledby PARSE-FloatExponent, 208
 - calledby PARSE-Primary1, 204
 - calledby PARSE-Sexpr1, 214
 - calls parse-number, 209
 - defun, 209
- PARSE-Iterator, 219
 - calledby PARSE-IteratorTail, 218
 - calledby PARSE-Loop, 224
 - calls PARSE-Expression, 219
 - calls PARSE-Expr, 219
 - calls PARSE-Primary, 219
 - calls match-advance-string, 219
 - calls must, 219
 - calls optional, 219
 - calls pop-stack-1, 219
 - calls pop-stack-2, 219
 - calls pop-stack-3, 219

- defun, 219
- PARSE-IteratorTail, 218
 - calledby PARSE-Sequence1, 217
 - calls PARSE-Iterator, 218
 - calls bang, 218
 - calls match-advance-string, 218
 - calls optional, 218
 - calls star, 218
 - defun, 218
- parse-keyword
 - calledby PARSE-AnyId, 216
- PARSE-Label, 201
 - calledby PARSE-LabelExpr, 225
 - calledby PARSE-Leave, 222
 - calls PARSE-Name, 201
 - calls match-advance-string, 201
 - calls must, 201
 - defun, 201
- PARSE-LabelExpr, 225
 - calls PARSE-Expr, 225
 - calls PARSE-Label, 225
 - calls must, 225
 - calls pop-stack-1, 225
 - calls pop-stack-2, 225
 - calls push-reduction, 225
 - defun, 225
- PARSE-Leave, 222
 - calls PARSE-Expression, 222
 - calls PARSE-Label, 222
 - calls match-advance-string, 222
 - calls must, 222
 - calls pop-stack-1, 222
 - calls push-reduction, 222
 - defun, 222
- PARSE-LedPart, 194
 - calledby PARSE-Expr, 193
 - calls PARSE-Operation, 194
 - calls pop-stack-1, 194
 - calls push-reduction, 194
 - defun, 194
- PARSE-leftBindingPowerOf, 195
 - calledby PARSE-Operation, 195
 - calls elemn, 195
 - calls getl, 195
 - defun, 195
- PARSE-Loop, 224
 - calls PARSE-Expr, 224
 - calls PARSE-Iterator, 224
 - calls match-advance-string, 224
 - calls must, 224
 - calls pop-stack-1, 224
 - calls pop-stack-2, 224
 - calls push-reduction, 224
 - calls star, 224
 - defun, 224
- PARSE-Name, 212
 - calledby PARSE-Label, 201
 - calledby PARSE-VarForm, 211
 - calls parse-identifier, 212
 - calls pop-stack-1, 212
 - calls push-reduction, 212
 - defun, 212
- PARSE-NBGlyphTok, 215
 - calledby PARSE-Sexpr1, 214
 - calls action, 215
 - calls advance-token, 215
 - calls match-current-token, 215
 - uses tok, 215
 - defun, 215
- PARSE-NewExpr, 183
 - calledby spad, 264
 - calls PARSE-Statement, 183
 - calls action, 183
 - calls current-symbol, 183
 - calls match-string, 183
 - calls must, 183
 - calls processSynonyms, 183
 - uses definition-name, 183
 - defun, 183
- PARSE-NudPart, 194
 - calledby PARSE-Expr, 193
 - calls PARSE-Form, 194
 - calls PARSE-Operation, 194
 - calls PARSE-Reduction, 194
 - calls pop-stack-1, 194
 - calls push-reduction, 194
 - uses rbp, 194
 - defun, 194
- parse-number
 - calledby PARSE-FloatTok, 225
 - calledby PARSE-IntegerTok, 209
- PARSE-OpenBrace, 218

- calledby PARSE-Sequence, 216
- calls action, 218
- calls advance-token, 218
- calls current-symbol, 218
- calls eqcar, 218
- calls getToken, 218
- calls push-reduction, 218
- defun, 218
- PARSE-OpenBracket, 217
 - calledby PARSE-Sequence, 216
 - calls action, 217
 - calls advance-token, 217
 - calls current-symbol, 217
 - calls eqcar, 217
 - calls getToken, 217
 - calls push-reduction, 217
 - defun, 217
- PARSE-Operation, 195
 - calledby PARSE-LedPart, 194
 - calledby PARSE-NudPart, 194
 - calls PARSE-getSemanticForm, 195
 - calls PARSE-leftBindingPowerOf, 195
 - calls PARSE-rightBindingPowerOf, 195
 - calls action, 195
 - calls current-symbol, 195
 - calls elemn, 195
 - calls getl, 195
 - calls lt, 195
 - calls match-current-token, 195
 - uses ParseMode, 195
 - uses rbp, 195
 - uses tmptok, 195
 - defun, 195
- PARSE-Option, 188
 - calledby PARSE-CommandTail, 187
 - calls PARSE-PrimaryOrQM, 188
 - calls match-advance-string, 188
 - calls must, 188
 - calls star, 188
 - defun, 188
- PARSE-Prefix, 197
 - calledby PARSE-getSemanticForm, 196
 - calls PARSE-Expression, 197
- calls PARSE-TokTail, 197
- calls action, 197
- calls advance-token, 197
- calls current-symbol, 197
- calls must, 197
- calls optional, 197
- calls pop-stack-1, 197
- calls pop-stack-2, 197
- calls push-reduction, 197
- defun, 197
- PARSE-Primary, 203
 - calledby PARSE-Application, 201
 - calledby PARSE-Iterator, 219
 - calledby PARSE-PrimaryOrQM, 188
 - calledby PARSE-Selector, 202
 - calls , 203
 - defun, 203
- PARSE-Primary1, 204
 - calledby PARSE-Primary1, 204
 - calledby PARSE-PrimaryNoFloat, 203
 - calledby PARSE-Qualification, 198
 - calls PARSE-Data, 204
 - calls PARSE-Enclosure, 204
 - calls PARSE-Expr, 204
 - calls PARSE-FormalParameter, 204
 - calls PARSE-IntegerTok, 204
 - calls PARSE-Primary1, 204
 - calls PARSE-Quad, 204
 - calls PARSE-Sequence, 204
 - calls PARSE-String, 204
 - calls PARSE-VarForm, 204
 - calls current-symbol, 204
 - calls match-advance-string, 204
 - calls match-string, 204
 - calls must, 204
 - calls optional, 204
 - calls pop-stack-1, 204
 - calls pop-stack-2, 204
 - calls push-reduction, 204
 - uses \$boot, 204
 - defun, 204
- PARSE-PrimaryNoFloat, 203
 - calledby PARSE-Selector, 202
 - calls PARSE-Primary1, 203
 - calls PARSE-TokTail, 203

- calls optional, 203
- defun, 203
- PARSE-PrimaryOrQM, 188
 - calledby PARSE-Option, 188
 - calledby PARSE-PrimaryOrQM, 188
 - calledby PARSE-SpecialCommand, 185
 - calls PARSE-PrimaryOrQM, 188
 - calls PARSE-Primary, 188
 - calls match-advance-string, 188
 - calls push-reduction, 188
 - defun, 188
- PARSE-Quad, 210
 - calledby PARSE-Primary1, 204
 - calls PARSE-GlyphTok, 210
 - calls match-advance-string, 210
 - calls push-reduction, 210
 - uses \$boot, 210
 - defun, 210
- PARSE-Qualification, 198
 - calledby PARSE-TokTail, 198
 - calls PARSE-Primary1, 198
 - calls dollarTran, 198
 - calls match-advance-string, 198
 - calls must, 198
 - calls pop-stack-1, 198
 - calls push-reduction, 198
 - defun, 198
- PARSE-Reduction, 199
 - calledby PARSE-NudPart, 194
 - calls PARSE-Expr, 199
 - calls PARSE-ReductionOp, 199
 - calls must, 199
 - calls pop-stack-1, 199
 - calls pop-stack-2, 199
 - calls push-reduction, 199
 - defun, 199
- PARSE-ReductionOp, 199
 - calledby PARSE-Reduction, 199
 - calls action, 199
 - calls advance-token, 199
 - calls current-symbol, 199
 - calls getl, 199
 - calls match-next-token, 199
 - defun, 199
- PARSE-Return, 221
 - calls PARSE-Expression, 221
 - calls match-advance-string, 221
 - calls must, 221
 - calls pop-stack-1, 221
 - calls push-reduction, 221
 - defun, 221
- PARSE-rightBindingPowerOf, 196
 - calledby PARSE-Expression, 192
 - calledby PARSE-Operation, 195
 - calls elemn, 196
 - calls getl, 196
 - defun, 196
- PARSE-ScriptItem, 212
 - calledby PARSE-ScriptItem, 212
 - calledby PARSE-Scripts, 211
 - calls PARSE-Expr, 212
 - calls PARSE-ScriptItem, 212
 - calls match-advance-string, 212
 - calls must, 212
 - calls optional, 212
 - calls pop-stack-1, 212
 - calls pop-stack-2, 212
 - calls push-reduction, 212
 - calls star, 212
 - defun, 212
- PARSE-Scripts, 211
 - calledby PARSE-VarForm, 211
 - calls PARSE-ScriptItem, 211
 - calls match-advance-string, 211
 - calls must, 211
 - defun, 211
- PARSE-Seg, 223
 - calls PARSE-Expression, 223
 - calls PARSE-GlyphTok, 223
 - calls bang, 223
 - calls optional, 223
 - calls pop-stack-1, 223
 - calls pop-stack-2, 223
 - calls push-reduction, 223
 - defun, 223
- PARSE-Selector, 202
 - calledby PARSE-Application, 201
 - calls PARSE-Float, 202
 - calls PARSE-PrimaryNoFloat, 202
 - calls PARSE-Primary, 202
 - calls char-ne, 202

- calls current-char, 202
- calls current-symbol, 202
- calls match-advance-string, 202
- calls must, 202
- calls pop-stack-1, 202
- calls pop-stack-2, 202
- calls push-reduction, 202
- uses \$boot, 202
- defun, 202
- PARSE-SemiColon, 221
 - calls PARSE-Expr, 221
 - calls match-advance-string, 221
 - calls must, 221
 - calls pop-stack-1, 221
 - calls pop-stack-2, 221
 - calls push-reduction, 221
 - defun, 221
- PARSE-Sequence, 216
 - calledby PARSE-Primary1, 204
 - calls PARSE-OpenBrace, 216
 - calls PARSE-OpenBracket, 216
 - calls PARSE-Sequence1, 216
 - calls match-advance-string, 216
 - calls must, 216
 - calls pop-stack-1, 216
 - calls push-reduction, 216
 - defun, 216
- PARSE-Sequence1, 217
 - calledby PARSE-Sequence, 216
 - calls PARSE-Expression, 217
 - calls PARSE-IteratorTail, 217
 - calls optional, 217
 - calls pop-stack-1, 217
 - calls pop-stack-2, 217
 - calls push-reduction, 217
 - defun, 217
- PARSE-Sexpr, 213
 - calledby PARSE-Data, 213
 - calls PARSE-Sexpr1, 213
 - defun, 213
- PARSE-Sexpr1, 214
 - calledby PARSE-Sexpr1, 214
 - calledby PARSE-Sexpr, 213
 - calls PARSE-AnyId, 214
 - calls PARSE-GlyphTok, 214
 - calls PARSE-IntegerTok, 214
 - calls PARSE-NBGlyphTok, 214
 - calls PARSE-Sexpr1, 214
 - calls PARSE-String, 214
 - calls action, 214
 - calls bang, 214
 - calls match-advance-string, 214
 - calls must, 214
 - calls nth-stack, 214
 - calls optional, 214
 - calls pop-stack-1, 214
 - calls pop-stack-2, 214
 - calls push-reduction, 214
 - calls star, 214
 - defun, 214
- parse-spadstring
 - calledby PARSE-String, 210
- PARSE-SpecialCommand, 185
 - calledby PARSE-Command, 184
 - calledby PARSE-SpecialCommand, 185
 - calls PARSE-CommandTail, 185
 - calls PARSE-Expression, 185
 - calls PARSE-PrimaryOrQM, 185
 - calls PARSE-SpecialCommand, 185
 - calls PARSE-TokenCommandTail, 185
 - calls PARSE-TokenList, 185
 - calls action, 185
 - calls bang, 185
 - calls current-symbol, 185
 - calls match-advance-string, 185
 - calls must, 185
 - calls optional, 185
 - calls pop-stack-1, 185
 - calls push-reduction, 185
 - calls star, 185
 - uses \$noParseCommands, 185
 - uses \$tokenCommands, 185
 - defun, 185
- PARSE-SpecialKeyWord, 184
 - calledby PARSE-Command, 184
 - calls action, 184
 - calls current-symbol, 184
 - calls current-token, 184
 - calls match-current-token, 184
 - calls token-symbol, 184

- calls unAbbreviateKeyword, 184
- defun, 184
- PARSE-Statement, 189
 - calledby PARSE-NewExpr, 183
 - calls PARSE-Expr, 189
 - calls match-advance-string, 189
 - calls must, 189
 - calls optional, 189
 - calls pop-stack-1, 189
 - calls pop-stack-2, 189
 - calls push-reduction, 189
 - calls star, 189
 - defun, 189
- PARSE-String, 210
 - calledby PARSE-Primary1, 204
 - calledby PARSE-Sexpr1, 214
 - calls parse-spadstring, 210
 - defun, 210
- PARSE-Suffix, 220
 - calls PARSE-TokTail, 220
 - calls action, 220
 - calls advance-token, 220
 - calls current-symbol, 220
 - calls optional, 220
 - calls pop-stack-1, 220
 - calls push-reduction, 220
 - defun, 220
- PARSE-TokenCommandTail, 186
 - calledby PARSE-SpecialCommand, 185
 - calledby PARSE-TokenCommandTail, 186
 - calls PARSE-TokenCommandTail, 186
 - calls PARSE-TokenOption, 186
 - calls action, 186
 - calls atEndOfLine, 186
 - calls bang, 186
 - calls optional, 186
 - calls pop-stack-1, 186
 - calls pop-stack-2, 186
 - calls push-reduction, 186
 - calls star, 186
 - calls systemCommand, 186
 - defun, 186
- PARSE-TokenList, 187
 - calledby PARSE-SpecialCommand, 185
 - calledby PARSE-TokenOption, 186
 - calls action, 187
 - calls advance-token, 187
 - calls current-symbol, 187
 - calls isTokenDelimiter, 187
 - calls push-reduction, 187
 - calls star, 187
 - defun, 187
- PARSE-TokenOption, 186
 - calledby PARSE-TokenCommandTail, 186
 - calls PARSE-TokenList, 186
 - calls match-advance-string, 186
 - calls must, 186
 - defun, 186
- PARSE-TokTail, 198
 - calledby PARSE-Infix, 197
 - calledby PARSE-Prefix, 197
 - calledby PARSE-PrimaryNoFloat, 203
 - calledby PARSE-Suffix, 220
 - calls PARSE-Qualification, 198
 - calls action, 198
 - calls char-eq, 198
 - calls copy-token, 198
 - calls current-char, 198
 - calls current-symbol, 198
 - uses \$boot, 198
 - defun, 198
- PARSE-VarForm, 211
 - calledby PARSE-Primary1, 204
 - calls PARSE-Name, 211
 - calls PARSE-Scripts, 211
 - calls optional, 211
 - calls pop-stack-1, 211
 - calls pop-stack-2, 211
 - calls push-reduction, 211
 - defun, 211
- PARSE-With, 190
 - calledby PARSE-InfixWith, 189
 - calls match-advance-string, 190
 - calls must, 190
 - calls pop-stack-1, 190
 - calls push-reduction, 190

- defun, 190
- parseAnd, 40
 - calledby parseAnd, 40
 - calls parseAnd, 40
 - calls parseIf, 40
 - calls parseTranList, 40
 - calls parseTran, 40
 - uses \$InteractiveMode, 40
 - defun, 40
- parseAtom, 37
 - calledby parseTran, 36
 - calls parseLeave, 37
 - uses \$NoValue, 37
 - defun, 37
- parseAtSign, 41
 - calls parseTran, 41
 - calls parseType, 41
 - uses \$InteractiveMode, 41
 - defun, 41
- parseCategory, 41
 - calls contained, 41
 - calls parseDropAssertions, 41
 - calls parseTranList, 41
 - defun, 41
- parseCoerce, 42
 - calls parseTran, 42
 - calls parseType, 42
 - uses \$InteractiveMode, 42
 - defun, 42
- parseColon, 42
 - calls parseTran, 42
 - calls parseType, 42
 - uses \$InteractiveMode, 42
 - uses \$insideConstructIfTrue, 42
 - defun, 42
- parseConstruct, 38
 - calledby parseTran, 36
 - calls parseTranList, 38
 - uses \$insideConstructIfTrue, 38
 - defun, 38
- parseDEF, 43
 - calls opFf, 43
 - calls parseLhs, 43
 - calls parseTranCheckForRecord, 43
 - calls parseTranList, 43
 - calls setDefOp, 43
 - uses \$lhs, 43
 - defun, 43
- parseDollarGreaterEqual, 44
 - calls msubst, 44
 - calls parseTran, 44
 - uses \$op, 44
 - defun, 44
- parseDollarGreaterThan, 44
 - calls msubst, 44
 - calls parseTran, 44
 - uses \$op, 44
 - defun, 44
- parseDollarLessEqual, 45
 - calls msubst, 45
 - calls parseTran, 45
 - uses \$op, 45
 - defun, 45
- parseDollarNotEqual, 45
 - calls msubst, 45
 - calls parseTran, 45
 - uses \$op, 45
 - defun, 45
- parseDropAssertions
 - calledby parseCategory, 41
- parseEquivalence, 46
 - calls parseIf, 46
 - defun, 46
- parseExit, 46
 - calls moan, 46
 - calls parseTran, 46
 - defun, 46
- parseGreaterEqual, 47
 - calls parseTran, 47
 - uses \$op, 47
 - defun, 47
- parseGreaterThan, 47
 - calls parseTran, 47
 - uses \$op, 47
 - defun, 47
- parseIf, 53
 - calledby parseAnd, 40
 - calledby parseEquivalence, 46
 - calledby parseImplies, 54
 - calledby parseOr, 63
 - calls parseIf, ifTran, 53
 - calls parseTran, 53

- defun, 53
- parseIf,ifTran, 51
 - calledby parseIf,ifTran, 51
 - calledby parseIf, 53
 - calls incExitLevel, 51
 - calls makeSimplePredicateOrNil, 51
 - calls parseIf,ifTran, 51
 - calls parseTran, 51
 - uses \$InteractiveMode, 51
- defun, 51
- parseImplies, 54
 - calls parseIf, 54
- defun, 54
- parseIn, 55
 - calledby parseInBy, 56
 - calls parseTran, 55
 - calls postError, 55
- defun, 55
- parseInBy, 56
 - calls bright, 56
 - calls parseIn, 56
 - calls parseTran, 56
 - calls postError, 56
- defun, 56
- parseIs, 57
 - calls parseTran, 57
 - calls transIs, 57
- defun, 57
- parseIsnt, 57
 - calls parseTran, 57
 - calls transIs, 57
- defun, 57
- parseJoin, 58
 - calls parseTranList, 58
- defun, 58
- parseLeave, 59
 - calledby parseAtom, 37
 - calls parseTran, 59
- defun, 59
- parseLessEqual, 59
 - calls parseTran, 59
 - uses \$op, 59
- defun, 59
- parseLET, 60
 - calls opOf, 60
 - calls parseTranCheckForRecord, 60
- calls parseTran, 60
- calls transIs, 60
- defun, 60
- parseLETD, 60
 - calls parseTran, 60
 - calls parseType, 60
- defun, 60
- parseLhs
 - calledby parseDEF, 43
- parseMDEF, 61
 - calls opOf, 61
 - calls parseTranCheckForRecord, 61
 - calls parseTranList, 61
 - calls parseTran, 61
 - uses \$lhs, 61
- defun, 61
- ParseMode, 182
 - usedby PARSE-Expression, 192
 - usedby PARSE-Operation, 195
- defvar, 182
- parseNot, 62
 - calls parseTran, 62
 - uses \$InteractiveMode, 62
- defun, 62
- parseNotEqual, 62
 - calls msubst, 62
 - calls parseTran, 62
 - uses \$op, 62
- defun, 62
- parseOr, 63
 - calledby parseOr, 63
 - calls parseIf, 63
 - calls parseOr, 63
 - calls parseTranList, 63
 - calls parseTran, 63
- defun, 63
- parsepiles, 26
 - calledby preparse1, 23
 - calls add-parens-and-semis-to-line, 26
- defun, 26
- parsePretend, 64
 - calls parseTran, 64
 - calls parseType, 64
- defun, 64
- parseprint

- calledby preparse, 18
- parseReturn, 64
 - calls moan, 64
 - calls parseTran, 64
 - defun, 64
- parseSegment, 65
 - calls parseTran, 65
 - defun, 65
- parseSeq, 65
 - calls last, 65
 - calls mapInto, 65
 - calls postError, 65
 - calls transSeq, 65
 - defun, 65
- parseTran, 36
 - calledby compReduce1, 105
 - calledby parseAnd, 40
 - calledby parseAtSign, 41
 - calledby parseCoerce, 42
 - calledby parseColon, 42
 - calledby parseDollarGreaterEqual, 44
 - calledby parseDollarGreaterThan, 44
 - calledby parseDollarLessEqual, 45
 - calledby parseDollarNotEqual, 45
 - calledby parseExit, 46
 - calledby parseGreaterEqual, 47
 - calledby parseGreaterThan, 47
 - calledby parseIf, ifTran, 51
 - calledby parseIf, 53
 - calledby parseInBy, 56
 - calledby parseIn, 55
 - calledby parseIsnt, 57
 - calledby parseIs, 57
 - calledby parseLETD, 60
 - calledby parseLET, 60
 - calledby parseLeave, 59
 - calledby parseLessEqual, 59
 - calledby parseMDEF, 61
 - calledby parseNotEqual, 62
 - calledby parseNot, 62
 - calledby parseOr, 63
 - calledby parsePretend, 64
 - calledby parseReturn, 64
 - calledby parseSegment, 65
 - calledby parseTranList, 37
 - calledby parseTransform, 35
 - calledby parseTran, 36
 - calls getl, 36
 - calls parseAtom, 36
 - calls parseConstruct, 36
 - calls parseTranList, 36
 - calls parseTran, 36
 - uses \$op, 36
 - defun, 36
- parseTranCheckForRecord
 - calledby parseDEF, 43
 - calledby parseLET, 60
 - calledby parseMDEF, 61
- parseTranList, 37
 - calledby parseAnd, 40
 - calledby parseCategory, 41
 - calledby parseConstruct, 38
 - calledby parseDEF, 43
 - calledby parseJoin, 58
 - calledby parseMDEF, 61
 - calledby parseOr, 63
 - calledby parseTranList, 37
 - calledby parseTran, 36
 - calledby parseVCONS, 66
 - calls parseTranList, 37
 - calls parseTran, 37
 - defun, 37
- parseTransform, 35
 - calledby s-process, 267
 - calls msubst, 35
 - calls parseTran, 35
 - uses \$defOp, 35
 - defun, 35
- parseType
 - calledby parseAtSign, 41
 - calledby parseCoerce, 42
 - calledby parseColon, 42
 - calledby parseLETD, 60
 - calledby parsePretend, 64
- parseVCONS, 66
 - calls parseTranList, 66
 - defun, 66
- parseWhere, 66
 - calls mapInto, 66
 - defun, 66

- pathname(5)
 - calledby compileSpad2Cmd, 250
 - calledby compileSpadLispCmd, 305
 - calledby compiler, 247
- pathnameDirectory(5)
 - calledby compileSpadLispCmd, 305
- pathnameName(5)
 - calledby compileSpadLispCmd, 305
- pathnameType(5)
 - calledby compileSpad2Cmd, 250
 - calledby compileSpadLispCmd, 305
 - calledby compiler, 247
- pname
 - calledby comp3, 276
 - calledby floatexpid, 238
 - calledby getScriptName, 147
- pop-stack-1
 - calledby PARSE-Application, 201
 - calledby PARSE-Category, 191
 - calledby PARSE-CommandTail, 187
 - calledby PARSE-Conditional, 223
 - calledby PARSE-Data, 213
 - calledby PARSE-Enclosure, 209
 - calledby PARSE-Exit, 222
 - calledby PARSE-Expression, 192
 - calledby PARSE-Expr, 193
 - calledby PARSE-FloatTok, 225
 - calledby PARSE-Float, 205
 - calledby PARSE-Form, 200
 - calledby PARSE-Import, 193
 - calledby PARSE-InfixWith, 189
 - calledby PARSE-Infix, 197
 - calledby PARSE-Iterator, 219
 - calledby PARSE-LabelExpr, 225
 - calledby PARSE-Leave, 222
 - calledby PARSE-LedPart, 194
 - calledby PARSE-Loop, 224
 - calledby PARSE-Name, 212
 - calledby PARSE-NudPart, 194
 - calledby PARSE-Prefix, 197
 - calledby PARSE-Primary1, 204
 - calledby PARSE-Qualification, 198
 - calledby PARSE-Reduction, 199
 - calledby PARSE-Return, 221
 - calledby PARSE-ScriptItem, 212
 - calledby PARSE-Seg, 223
- calledby PARSE-Selector, 202
- calledby PARSE-SemiColon, 221
- calledby PARSE-Sequence1, 217
- calledby PARSE-Sequence, 216
- calledby PARSE-Sexpr1, 214
- calledby PARSE-SpecialCommand, 185
- calledby PARSE-Statement, 189
- calledby PARSE-Suffix, 220
- calledby PARSE-TokenCommandTail, 186
- calledby PARSE-VarForm, 211
- calledby PARSE-With, 190
- calledby spad, 264
- calledby star, 240
- pop-stack-2
 - calledby PARSE-Application, 201
 - calledby PARSE-Category, 191
 - calledby PARSE-CommandTail, 187
 - calledby PARSE-Conditional, 223
 - calledby PARSE-Float, 205
 - calledby PARSE-Import, 193
 - calledby PARSE-InfixWith, 189
 - calledby PARSE-Infix, 197
 - calledby PARSE-Iterator, 219
 - calledby PARSE-LabelExpr, 225
 - calledby PARSE-Loop, 224
 - calledby PARSE-Prefix, 197
 - calledby PARSE-Primary1, 204
 - calledby PARSE-Reduction, 199
 - calledby PARSE-ScriptItem, 212
 - calledby PARSE-Seg, 223
 - calledby PARSE-Selector, 202
 - calledby PARSE-SemiColon, 221
 - calledby PARSE-Sequence1, 217
 - calledby PARSE-Sexpr1, 214
 - calledby PARSE-Statement, 189
 - calledby PARSE-TokenCommandTail, 186
 - calledby PARSE-VarForm, 211
- pop-stack-3
 - calledby PARSE-Category, 191
 - calledby PARSE-Conditional, 223
 - calledby PARSE-Float, 205
 - calledby PARSE-Iterator, 219
- pop-stack-4

- calledby PARSE-Float, 205
- postAdd, 121
 - calls postCapsule, 121
 - calls postTran, 121
 - defun, 121
- postAtom, 114
 - calledby postTran, 113
 - uses \$boot, 114
 - defun, 114
- postAtSign, 121
 - calls postTran, 121
 - calls postType, 121
 - defun, 121
- postBigFloat, 122
 - calls postTran, 122
 - uses \$InteractiveMode, 122
 - uses \$boot, 122
 - defun, 122
- postBlock, 122
 - calledby postSemiColon, 139
 - calls postBlockItemList, 122
 - calls postTran, 122
 - defun, 122
- postBlockItemList
 - calledby postBlock, 122
- postCapsule
 - calledby postAdd, 121
- postCategory, 123
 - calls nreverse0, 123
 - calls postTran, 123
 - uses \$insidePostCategoryIfTrue, 123
 - defun, 123
- postcheck, 116
 - calledby postTransformCheck, 116
 - calledby postcheck, 116
 - calls postcheck, 116
 - calls setDefOp, 116
 - defun, 116
- postCollect, 125
 - calledby postCollect, 125
 - calledby postTupleCollect, 141
 - calls postCollect,finish, 125
 - calls postCollect, 125
 - calls postIteratorList, 125
 - calls postTran, 125
 - defun, 125
- postCollect,finish, 124
 - calledby postCollect, 125
 - defun, 124
- postColon, 126
 - calls postTran, 126
 - calls postType, 126
 - defun, 126
- postColonColon, 126
 - calls postForm, 126
 - calls stringimage, 126
 - uses \$boot, 126
 - defun, 126
- postComma, 127
 - calls comma2Tuple, 127
 - calls postTuple, 127
 - defun, 127
- postConstruct, 128
 - calls comma2Tuple, 128
 - calls postMakeCons, 128
 - calls postTranList, 128
 - calls postTranSegment, 128
 - calls postTran, 128
 - calls tuple2List, 128
 - defun, 128
- postDef, 130
 - calls nequal, 130
 - calls nreverse0, 130
 - calls postDefArgs, 130
 - calls postMDef, 130
 - calls postTran, 130
 - calls recordHeaderDocumentation, 130
 - uses \$InteractiveMode, 130
 - uses \$boot, 130
 - uses \$docList, 130
 - uses \$headerDocumentation, 130
 - uses \$maxSignatureLineNumber, 130
 - defun, 130
- postDefArgs
 - calledby postDef, 130
- postError, 117
 - calledby getScriptName, 147
 - calledby parseInBy, 56
 - calledby parseIn, 55
 - calledby parseSeq, 65
 - calledby postForm, 118

- calls bumperrorcount, 117
 - calls nequal, 117
 - uses \$InteractiveMode, 117
 - uses \$defOp, 117
 - uses \$postStack, 117
 - defun, 117
- postExit, 131
 - calls postTran, 131
 - defun, 131
- postFlatten
 - calledby comma2Tuple, 127
- postFlattenLeft
 - calledby postSemiColon, 139
- postForm, 118
 - calledby postColonColon, 126
 - calledby postTran, 113
 - calls bright, 118
 - calls internl, 118
 - calls postError, 118
 - calls postTranList, 118
 - calls postTran, 118
 - uses \$boot, 118
 - defun, 118
- postIf, 132
 - calls nreverse0, 132
 - calls postTran, 132
 - uses \$boot, 132
 - defun, 132
- postIn, 133
 - calls postInSeq, 133
 - calls postTran, 133
 - calls systemErrorHere, 133
 - defun, 133
- postin, 133
 - calls postInSeq, 133
 - calls postTran, 133
 - calls systemErrorHere, 133
 - defun, 133
- postInSeq
 - calledby postIn, 133
 - calledby postin, 133
- postIteratorList
 - calledby postCollect, 125
 - calledby postRepeat, 138
- postJoin, 134
 - calls postTranList, 134
 - calls postTran, 134
 - defun, 134
- postMakeCons
 - calledby postConstruct, 128
- postMapping, 134
 - calls postTran, 134
 - calls unTuple, 134
 - defun, 134
- postMDef, 136
 - calledby postDef, 130
 - calls nreverse0, 136
 - calls postTran, 136
 - calls throwkeyedmsg, 136
 - uses \$InteractiveMode, 136
 - uses \$boot, 136
 - defun, 136
- postOp, 114
 - calledby postTran, 113
 - defun, 114
- postPretend, 137
 - calls postTran, 137
 - calls postType, 137
 - defun, 137
- postQUOTE, 137
 - defun, 137
- postReduce, 138
 - calledby postReduce, 138
 - calls postReduce, 138
 - calls postTran, 138
 - uses \$InteractiveMode, 138
 - defun, 138
- postRepeat, 138
 - calls postIteratorList, 138
 - calls postTran, 138
 - defun, 138
- postScripts, 139
 - calls getScriptName, 139
 - calls postTranScripts, 139
 - defun, 139
- postScriptsForm, 115
 - calledby postTran, 113
 - calls getScriptName, 115
 - calls length, 115
 - calls postTranScripts, 115
 - defun, 115
- postSemiColon, 139
 - calls postTran, 134
 - defun, 134

- calls postBlock, 139
- calls postFlattenLeft, 139
- defun, 139
- postSignature, 140
 - calls killColons, 140
 - calls pairp, 140
 - calls postType, 140
 - calls removeSuperfluousMapping, 140
 - defun, 140
- postSlash, 140
 - calls postTran, 140
 - defun, 140
- postTran, 113
 - calledby postAdd, 121
 - calledby postAtSign, 121
 - calledby postBigFloat, 122
 - calledby postBlock, 122
 - calledby postCategory, 123
 - calledby postCollect, 125
 - calledby postColon, 126
 - calledby postConstruct, 128
 - calledby postDef, 130
 - calledby postExit, 131
 - calledby postForm, 118
 - calledby postIf, 132
 - calledby postIn, 133
 - calledby postJoin, 134
 - calledby postMDef, 136
 - calledby postMapping, 134
 - calledby postPretend, 137
 - calledby postReduce, 138
 - calledby postRepeat, 138
 - calledby postSlash, 140
 - calledby postTranList, 114
 - calledby postTranScripts, 115
 - calledby postTransform, 111
 - calledby postTran, 113
 - calledby postWhere, 142
 - calledby postWith, 142
 - calledby postin, 133
 - calls pairp, 113
 - calls postAtom, 113
 - calls postForm, 113
 - calls postOp, 113
 - calls postScriptsForm, 113
 - calls postTranList, 113
 - calls postTran, 113
 - calls qcar, 113
 - calls qcdr, 113
 - calls unTuple, 113
 - defun, 113
- postTranList, 114
 - calledby postConstruct, 128
 - calledby postForm, 118
 - calledby postJoin, 134
 - calledby postTran, 113
 - calledby postTuple, 141
 - calledby postWhere, 142
 - calls postTran, 114
 - defun, 114
- postTranScripts, 115
 - calledby postScriptsForm, 115
 - calledby postScripts, 139
 - calledby postTranScripts, 115
 - calls postTranScripts, 115
 - calls postTran, 115
 - defun, 115
- postTranSegment
 - calledby postConstruct, 128
- postTransform, 111
 - calledby s-process, 267
 - calls aplTran, 111
 - calls identp, 111
 - calls postTransformCheck, 111
 - calls postTran, 111
 - defun, 111
- postTransformCheck, 116
 - calledby postTransform, 111
 - calls postcheck, 116
 - uses \$defOp, 116
 - defun, 116
- postTuple, 141
 - calledby postComma, 127
 - calls postTranList, 141
 - defun, 141
- postTupleCollect, 141
 - calls postCollect, 141
 - defun, 141
- postType
 - calledby postAtSign, 121
 - calledby postColon, 126

- calledby postPretend, 137
- calledby postSignature, 140
- postWhere, 142
 - calls postTranList, 142
 - calls postTran, 142
 - defun, 142
- postWith, 142
 - calls postTran, 142
 - uses \$insidePostCategoryIfTrue, 142
 - defun, 142
- PredImplies
 - calledby compForm2, 289
- prepare, 18
 - calledby prepare, 18
 - calledby spad, 264
 - calls ifcar, 18
 - calls parseprint, 18
 - calls prepare1, 18
 - calls prepare, 18
 - uses \$comblocklist, 18
 - uses \$constructorLineNumber, 18
 - uses \$docList, 18
 - uses \$headerDocumentation, 18
 - uses \$index, 18
 - uses \$maxSignatureLineNumber, 18
 - uses \$prepare-last-line, 18
 - uses \$prepareReportIfTrue, 18
 - uses \$skipme, 18
 - defun, 18
- prepare-echo, 30
 - calledby prepare1, 23
 - uses Echo-Meta, 30
 - uses \$EchoLineStack, 30
 - defun, 30
- prepare1, 23
 - calledby prepare, 18
 - calls doSystemCommand, 23
 - calls escaped, 23
 - calls fincomblock, 23
 - calls getfullstr, 23
 - calls indent-pos, 23
 - calls instring, 23
 - calls is-console, 23
 - calls maxindex, 23
 - calls parsepiles, 23
 - calls prepare-echo, 23
 - calls prepareReadLine, 23
 - calls strposl, 23
 - uses \$byConstructors, 23
 - uses \$constructorsSeen, 23
 - uses \$echolinestack, 23
 - uses \$linelist, 23
 - uses \$prepare-last-line, 23
 - uses \$skipme, 23
 - catches, 23
 - defun, 23
- prepareReadLine, 28
 - calledby prepare1, 23
 - calledby prepareReadLine, 28
 - calls dcq, 28
 - calls initial-substring, 28
 - calls prepareReadLine1, 28
 - calls prepareReadLine, 28
 - calls skip-to-endif, 28
 - calls storeblanks, 28
 - calls string2BootTree, 28
 - defun, 28
- prepareReadLine1, 29
 - calledby prepareReadLine1, 29
 - calledby prepareReadLine, 28
 - calls expand-tabs, 29
 - calls get-a-line, 29
 - calls maxindex, 29
 - calls prepareReadLine1, 29
 - calls strconc, 29
 - uses \$EchoLineStack, 29
 - uses \$index, 29
 - uses \$linelist, 29
 - uses \$prepare-last-line, 29
 - defun, 29
- pretend, 63, 103, 137
 - defplist, 63, 103, 137
- prettyprint
 - calledby s-process, 267
- primitiveType, 282
 - calledby compAtom, 280
 - uses \$DoubleFloat, 282
 - uses \$EmptyMode, 282
 - uses \$NegativeInteger, 282
 - uses \$NonNegativeInteger, 282
 - uses \$PositiveInteger, 282
 - uses \$String, 282

- defun, 282
- print-full
 - calledby def-process, 151
- prior-token
 - usedby PARSE-Expression, 192
- processFunctorOrPackage
 - calledby compCapsuleInner, 72
- processInteractive
 - calledby s-process, 267
- processSynonyms
 - calledby PARSE-NewExpr, 183
- push
 - calledby star, 240
- push-reduction, 241
 - calledby PARSE-AnyId, 216
 - calledby PARSE-Application, 201
 - calledby PARSE-Category, 191
 - calledby PARSE-CommandTail, 187
 - calledby PARSE-Command, 184
 - calledby PARSE-Conditional, 223
 - calledby PARSE-Data, 213
 - calledby PARSE-Enclosure, 209
 - calledby PARSE-Exit, 222
 - calledby PARSE-Expression, 192
 - calledby PARSE-Expr, 193
 - calledby PARSE-FloatBasePart, 207
 - calledby PARSE-FloatBase, 206
 - calledby PARSE-FloatExponent, 208
 - calledby PARSE-FloatTok, 225
 - calledby PARSE-Float, 205
 - calledby PARSE-Form, 200
 - calledby PARSE-Import, 193
 - calledby PARSE-InfixWith, 189
 - calledby PARSE-Infix, 197
 - calledby PARSE-LabelExpr, 225
 - calledby PARSE-Leave, 222
 - calledby PARSE-LedPart, 194
 - calledby PARSE-Loop, 224
 - calledby PARSE-Name, 212
 - calledby PARSE-NudPart, 194
 - calledby PARSE-OpenBrace, 218
 - calledby PARSE-OpenBracket, 217
 - calledby PARSE-Prefix, 197
 - calledby PARSE-Primary1, 204
 - calledby PARSE-PrimaryOrQM, 188
 - calledby PARSE-Quad, 210
 - calledby PARSE-Qualification, 198
 - calledby PARSE-Reduction, 199
 - calledby PARSE-Return, 221
 - calledby PARSE-ScriptItem, 212
 - calledby PARSE-Seg, 223
 - calledby PARSE-Selector, 202
 - calledby PARSE-SemiColon, 221
 - calledby PARSE-Sequence1, 217
 - calledby PARSE-Sequence, 216
 - calledby PARSE-Sexpr1, 214
 - calledby PARSE-SpecialCommand, 185
 - calledby PARSE-Statement, 189
 - calledby PARSE-Suffix, 220
 - calledby PARSE-TokenCommandTail, 186
 - calledby PARSE-TokenList, 187
 - calledby PARSE-VarForm, 211
 - calledby PARSE-With, 190
 - calledby star, 240
 - calls make-reduction, 241
 - calls stack-push, 241
 - uses reduce-stack, 241
 - defun, 241
- put
 - calledby compColon, 80
 - calledby compMacro, 102
 - calledby compTypeOf, 278
- qcar
 - calledby addCARorCDR, 165
 - calledby compAdd, 69
 - calledby compCategory, 76
 - calledby compCons1, 84
 - calledby compJoin, 98
 - calledby compLambda, 100
 - calledby compMacro, 102
 - calledby compWithMappingModel, 294
 - calledby decodeScripts, 148
 - calledby defIS1, 169
 - calledby defLET2, 156
 - calledby postTran, 113
- qcdr
 - calledby addCARorCDR, 165
 - calledby compAdd, 69

- calledby compCategory, 76
- calledby compCons1, 84
- calledby compJoin, 98
- calledby compLambda, 100
- calledby compWithMappingModel, 294
- calledby decodeScripts, 148
- calledby defIS1, 169
- calledby defLET2, 156
- calledby postTran, 113
- quote, 104, 137
 - defplist, 104, 137
- quote-if-string, 229
 - calledby match-advance-string, 227
 - calledby unget-tokens, 231
- calls escape-keywords, 229
- calls pack, 229
- calls strconc, 229
- calls token-nonblank, 229
- calls token-symbol, 229
- calls token-type, 229
- calls underscore, 229
- uses \$boot, 229
- uses \$spad, 229
- defun, 229
- quotify
 - calledby compIf, 95
- rbp
 - usedby PARSE-NudPart, 194
 - usedby PARSE-Operation, 195
- read-a-line, 31
 - calledby get-a-line, 33
 - calledby read-a-line, 31
 - calls Line-New-Line, 31
 - calls read-a-line, 31
 - calls subseq, 31
 - uses *eof*, 31
 - defun, 31
- recompile-lib-file-if-necessary, 306
 - calledby compileSpadLispCmd, 305
 - calls compile-lib-file, 306
 - uses *lisp-bin-filetype*, 306
 - defun, 306
- Record, 74
 - defplist, 74
- recordAttributeDocumentation
 - calledby PARSE-Category, 191
- RecordCategory, 85
 - defplist, 85
- recordHeaderDocumentation
 - calledby postDef, 130
- recordSignatureDocumentation
 - calledby PARSE-Category, 191
- reduce, 104, 137
 - defplist, 104, 137
- reduce-stack
 - usedby push-reduction, 241
- removeSuperfluousMapping
 - calledby postSignature, 140
- repeat, 138
 - defplist, 138
- replaceExitEtc
 - calledby compSeq1, 108
- reset
 - calledby def-it, 173
- resolve
 - calledby compCategory, 76
 - calledby compCoerce1, 78
 - calledby compConstructorCategory, 85
 - calledby compIf, 95
 - calledby convert, 281
 - calledby modifyModeStack, 302
- return, 64
 - defplist, 64
- s-process, 267
 - calledby spad, 264
 - calls compTopLevel, 267
 - calls curstrm, 267
 - calls def-process, 267
 - calls def-rename, 267
 - calls displayPreCompilationErrors, 267
 - calls displaySemanticErrors, 267
 - calls get-internal-run-time, 267
 - calls new2OldLisp, 267
 - calls parseTransform, 267
 - calls postTransform, 267
 - calls prettyprint, 267
 - calls processInteractive, 267

- calls `terpri`, 267
- uses `$DomainFrame`, 267
- uses `$EmptyEnvironment`, 267
- uses `$EmptyMode`, 267
- uses `$Index`, 267
- uses `$LocalFrame`, 268
- uses `$PolyMode`, 267
- uses `$Translation`, 268
- uses `$VariableCount`, 267
- uses `$compUniquelyIfTrue`, 267
- uses `$currentFunction`, 267
- uses `$exitModeStack`, 267
- uses `$exitMode`, 267
- uses `$e`, 267
- uses `$form`, 267
- uses `$genFVar`, 267
- uses `$genSDVar`, 267
- uses `$insideCapsuleFunctionIfTrue`, 267
- uses `$insideCategoryIfTrue`, 267
- uses `$insideCoerceInteractiveHardIfTrue`, 267
- uses `$insideExpressionIfTrue`, 267
- uses `$insideFunctorIfTrue`, 267
- uses `$insideWhereIfTrue`, 267
- uses `$leaveLevelStack`, 267
- uses `$leaveMode`, 267
- uses `$macroassoc`, 267
- uses `$newspad`, 267
- uses `$postStack`, 267
- uses `$previousTime`, 268
- uses `$returnMode`, 267
- uses `$semanticErrorStack`, 267
- uses `$top-level`, 267
- uses `$stopOp`, 267
- uses `$warningStack`, 267
- uses `curoutstream`, 268
- defun, 267
- say
 - calledby `compOrCroak1`, 272
 - calledby `def-process`, 151
 - calledby `defIS1`, 169
 - calledby `modifyModeStack`, 302
- sayBrightly
 - calledby `compMacro`, 102
 - calledby `compilerDoit`, 253
- sayKeyedMsg
 - calledby `compileSpad2Cmd`, 250
 - calledby `compileSpadLispCmd`, 305
- ScanOrPairVec
 - calledby `hasFormalMapVariable`, 300
- Scripts, 139
 - defplist, 139
- segment, 65
 - defplist, 65
- selectOptionLC(5)
 - calledby `compileSpad2Cmd`, 250
 - calledby `compileSpadLispCmd`, 305
 - calledby `compiler`, 247
- seq, 107
 - defplist, 107
- setDefOp, 143
 - calledby `parseDEF`, 43
 - calledby `postcheck`, 116
 - uses `$defOp`, 143
 - uses `$stopOp`, 143
- setelt
 - calledby `modifyModeStack`, 302
- shut
 - calledby `spad`, 264
- Signature, 139
 - defplist, 139
- skip-blanks
 - calledby `match-string`, 226
- skip-to-endif
 - calledby `preparseReadLine`, 28
- spad, 264
 - calledby `/rf-1`, 255
 - calls `PARSE-NewExpr`, 264
 - calls `addBinding`, 264
 - calls `init-boot/spad-reader`, 264
 - calls `initialize-preparse`, 264
 - calls `ioclear`, 264
 - calls `makeInitialModemapFrame`, 264
 - calls `pop-stack-1`, 264
 - calls `preparse`, 264
 - calls `s-process`, 264
 - calls `shut`, 264
 - uses `*comp370-apply*`, 264
 - uses `*eof*`, 264
 - uses `/editfile`, 264

- uses \$InitialDomainsInScope, 264
 - uses \$InteractiveFrame, 264
 - uses \$InteractiveMode, 264
 - uses \$noSubsumption, 264
 - uses echo-meta, 264
 - uses file-closed, 264
 - uses line, 264
 - uses xcape, 264
- catches, 264
- defun, 264
- spad-fixed-arg, 306
 - defun, 306
- spad2AsTranslatorAutoloadOnceTrigger
 - calledby compileSpad2Cmd, 250
- SpadInterpretStream(5)
 - calledby ncINTERPFILE, 304
- spadPrompt
 - calledby compileSpad2Cmd, 250
 - calledby compileSpadLispCmd, 305
- spadreduce
 - calledby floatexpid, 238
- stack-push
 - calledby push-reduction, 241
- stack-size
 - calledby star, 240
- stackAndThrow
 - calledby compDefine1, 88
 - calledby compLambda, 100
 - calledby compWithMappingModel1, 294
- stackMessage
 - calledby compElt, 91
 - calledby compSymbol, 283
- stackMessageIfNone
 - calledby compExit, 93
 - calledby compForm, 285
- stackSemanticError
 - calledby compColonInside, 279
 - calledby compJoin, 98
 - calledby compOrCroak1, 272
 - calledby compPretend, 103
- stackWarning
 - calledby compColonInside, 279
 - calledby compElt, 91
 - calledby compPretend, 103
- star, 240
 - calledby PARSE-Application, 201
 - calledby PARSE-Category, 191
 - calledby PARSE-CommandTail, 187
 - calledby PARSE-Expr, 193
 - calledby PARSE-Import, 193
 - calledby PARSE-IteratorTail, 218
 - calledby PARSE-Loop, 224
 - calledby PARSE-Option, 188
 - calledby PARSE-ScriptItem, 212
 - calledby PARSE-Sexpr1, 214
 - calledby PARSE-SpecialCommand, 185
 - calledby PARSE-Statement, 189
 - calledby PARSE-TokenCommandTail, 186
 - calledby PARSE-TokenList, 187
 - calls pop-stack-1, 240
 - calls push-reduction, 240
 - calls push, 240
 - calls stack-size, 240
 - defmacro, 240
- step
 - calledby floatexpid, 238
- storeblanks, 33
 - calledby preparseReadLine, 28
- defun, 33
- strconc
 - calledby compDefine1, 88
 - calledby compileSpad2Cmd, 250
 - calledby decodeScripts, 148
 - calledby defIS1, 169
 - calledby defLET1, 154
 - calledby defLET2, 156
 - calledby preparseReadLine1, 29
 - calledby quote-if-string, 229
 - calledby unget-tokens, 231
- string-not-greaterp
 - calledby initial-substring-p, 228
- string2BootTree
 - calledby preparseReadLine, 28
- stringimage
 - calledby comp3, 276
 - calledby decodeScripts, 148
 - calledby defIS1, 169
 - calledby defLET1, 154
 - calledby defLET2, 156

- calledby getScriptName, 147
- calledby postColonColon, 126
- stringPrefix?
 - calledby comp3, 276
- strposl
 - calledby preparse1, 23
- sublis
 - calledby compForm2, 289
 - calledby def-inner, 175
 - calledby def-where, 158
 - calledby def, 149
- subseq
 - calledby match-string, 226
 - calledby read-a-line, 31
- subst
 - calledby def-is2, 167
- systemCommand
 - calledby PARSE-CommandTail, 187
 - calledby PARSE-TokenCommandTail, 186
- systemError
 - calledby compReduce1, 105
 - calledby errhuh, 176
- systemErrorHere
 - calledby compCategory, 76
 - calledby compColon, 80
 - calledby postIn, 133
 - calledby postin, 133
- take
 - calledby compColon, 79
 - calledby compForm2, 289
 - calledby compWithMappingMode1, 294
- terminateSystemCommand(5)
 - calledby compileSpad2Cmd, 250
 - calledby compileSpadLispCmd, 305
- terpri
 - calledby s-process, 267
- throwKeyedMsg
 - calledby compileSpad2Cmd, 250
 - calledby compileSpadLispCmd, 305
 - calledby compiler, 247
- throwkeyedmsg
 - calledby postMDef, 136
- tmptok, 182
 - usedby PARSE-Operation, 195
 - defvar, 182
- tok, 182
 - usedby PARSE-GlyphTok, 215
 - usedby PARSE-NBGlyphTok, 215
 - defvar, 182
- token-nonblank
 - calledby PARSE-FloatBasePart, 207
 - calledby quote-if-string, 229
 - calledby unget-tokens, 231
- token-symbol
 - calledby PARSE-SpecialKeyword, 184
 - calledby make-symbol-of, 233
 - calledby match-token, 232
 - calledby quote-if-string, 229
- token-type
 - calledby match-token, 232
 - calledby quote-if-string, 229
- transIs
 - calledby parseIsnt, 57
 - calledby parseIs, 57
 - calledby parseLET, 60
- translabel
 - calledby PARSE-Data, 213
- transSeq
 - calledby parseSeq, 65
- try-get-token, 234
 - calledby advance-token, 235
 - calledby current-token, 233
 - calledby next-token, 234
 - calls get-token, 234
 - uses valid-tokens, 234
 - defun, 234
- tuple2List
 - calledby postConstruct, 128
- TupleCollect, 141
 - defplist, 141
- unAbbreviateKeyword
 - calledby PARSE-SpecialKeyword, 184
- underscore, 230
 - calledby quote-if-string, 229
 - calls , 230
 - defun, 230

- unget-tokens, 231
 - calledby match-string, 226
 - calls line-current-segment, 231
 - calls line-new-line, 231
 - calls line-number, 231
 - calls quote-if-string, 231
 - calls strconc, 231
 - calls token-nonblank, 231
 - uses valid-tokens, 231
 - defun, 231
- Union, 75
 - defplist, 75
- union
 - calledby compJoin, 98
- UnionCategory, 85
 - defplist, 85
- unionq
 - calledby freelist, 303
- unknownTypeError
 - calledby compColon, 79
- unTuple, 176
 - calledby postMapping, 134
 - calledby postTran, 113
 - defun, 176
- updateSourceFiles(5)
 - calledby compileSpad2Cmd, 250
- userError
 - calledby compOrCroak1, 272
- valid-tokens
 - usedby advance-token, 235
 - usedby current-token, 233
 - usedby next-token, 234
 - usedby try-get-token, 234
 - usedby unget-tokens, 231
- vcons, 66
 - defplist, 66
- vector, 109
 - defplist, 109
- VectorCategory, 85
 - defplist, 85
- whdef
 - calledby def-whereclause, 159
- where, 66, 109, 141
 - defplist, 66, 109, 141
- with, 142
 - defplist, 142
- wrapDomainSub
 - calledby compJoin, 98
- xcapex
 - usedby spad, 264
- XTokenReader, 235
 - calledby get-token, 235
 - usedby get-token, 235
 - defvar, 235