

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) 2007 Alfredo Portes

Portions Copyright (c) 2007 Arthur Ralfs

Portions Copyright (c) 2005 Timothy Daly

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

This book and the Axiom software is licensed as follows:

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

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

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

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

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

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

Contents

1	The Axiom Compiler	1
1.1	Makefile	1
2	Overview	3
2.1	The Input	4
2.2	The Output, the EQ.nrlib directory	8
2.3	The code.lsp and EQ.lsp files	9
2.4	The code.o file	23
2.5	The info file	23
2.6	The EQ.fn file	26
2.7	The index.kaf file	31
	The index offset byte	33
	The “loadTimeStuff”	33
	The “compilerInfo”	35
	The “constructorForm”	42
	The “constructorKind”	42
	The “constructorModemap”	42
	The “constructorCategory”	44
	The “sourceFile”	45
	The “modemaps”	45
	The “operationAlist”	47
	The “superDomain”	49
	The “signaturesAndLocals”	49
	The “attributes”	49
	The “predicates”	49
	The “abbreviation”	50
	The “parents”	50
	The “ancestors”	51
	The “documentation”	51
	The “slotInfo”	53
	The “index”	55

3	Compiler top level	57
3.1	Global Data Structures	57
3.2	Pratt Parsing	57
3.3)compile	58
	Spad compiler	61
3.4	Operator Precedence Table Initialization	62
	LED and NUD Tables	62
3.5	Gliph Table	65
	Rename Token Table	65
	Generic function table	66
3.6	Giant steps, Baby steps	66
4	The Parser	67
4.1	EQ.spad	67
4.2	boot transformations	71
	defun string2BootTree	71
	defun new2OldLisp	72
	defun new2OldTran	72
	defun newIf2Cond	73
	defun newDef2Def	74
	defun new2OldDefForm	74
	defun newConstruct	74
4.3	preparse	75
	defvar \$index	75
	defvar \$linelist	75
	defvar \$echolinestack	75
	defvar \$preparse-last-line	76
4.4	Parsing routines	76
	defun initialize-preparse	76
	defun preparse	80
	defun Build the lines from the input for piles	84
	defun parsepiles	87
	defun add-parens-and-semis-to-line	88
	defun preparseReadLine	89
	defun skip-ifblock	89
	defun preparseReadLine1	90
	defun expand-tabs	91
4.5	I/O Handling	92
	defun preparse-echo	92
	Parsing stack	92
	defstruct \$stack	92
	defun stack-load	92
	defun stack-clear	93
	defmacro stack-/empty	93
	defun stack-push	93
	defun stack-pop	94

Parsing token	94
deconstruct \$token	94
defvar \$prior-token	94
defvar \$nonblank	95
defvar \$current-token	95
defvar \$next-token	95
defvar \$valid-tokens	95
defun token-install	96
defun token-print	96
Parsing reduction	96
deconstruct \$reduction	96
5 Parse Transformers	97
5.1 Direct called parse routines	97
defun parseTransform	97
defun parseTran	97
defun parseAtom	98
defun parseTranList	99
defplist parseConstruct	99
defun parseConstruct	99
5.2 Indirect called parse routines	100
defplist parseAnd	101
defun parseAnd	101
defplist parseAtSign	101
defun parseAtSign	102
defun parseType	102
defplist parseCategory	102
defun parseCategory	103
defun parseDropAssertions	103
defplist parseCoerce	103
defun parseCoerce	104
defplist parseColon	104
defun parseColon	104
defplist parseDEF	105
defun parseDEF	105
defun parseLhs	106
defun transIs	106
defun transIs1	106
defun isListConstructor	107
defplist parseDollarGreaterthan	107
defun parseDollarGreaterthan	108
defplist parseDollarGreaterEqual	108
defun parseDollarGreaterEqual	108
defun parseDollarLessEqual	109
defplist parseDollarNotEqual	109
defun parseDollarNotEqual	109

defplist parseEquivalence	110
defun parseEquivalence	110
defplist parseExit	110
defun parseExit	110
defplist parseGreaterEqual	111
defun parseGreaterEqual	111
defplist parseGreaterThan	111
defun parseGreaterThan	112
defplist parseHas	112
defun parseHas	112
defun parseHasRhs	114
defun loadIfNecessary	114
defun loadLibIfNecessary	115
defun updateCategoryFrameForConstructor	116
defun convertOpAlist2compilerInfo	116
defun updateCategoryFrameForCategory	117
defplist parseIf	117
defun parseIf	118
defun parseIf,ifTran	118
defplist parseImplies	120
defun parseImplies	120
defplist parseIn	121
defun parseIn	121
defplist parseInBy	122
defun parseInBy	122
defplist parseIs	123
defun parseIs	123
defplist parseIsnt	123
defun parseIsnt	123
defplist parseJoin	124
defun parseJoin	124
defplist parseLeave	124
defun parseLeave	125
defplist parseLessEqual	125
defun parseLessEqual	125
defplist parseLET	126
defun parseLET	126
defplist parseLETD	126
defun parseLETD	127
defplist parseMDEF	127
defun parseMDEF	127
defplist parseNot	128
defplist parseNot	128
defun parseNot	128
defplist parseNotEqual	129
defun parseNotEqual	129

defplist parseOr	129
defun parseOr	129
defplist parsePretend	130
defun parsePretend	130
defplist parseReturn	131
defun parseReturn	131
defplist parseSegment	131
defun parseSegment	131
defplist parseSeq	132
defun parseSeq	132
defplist parseVCONS	132
defun parseVCONS	133
defplist parseWhere	133
defun parseWhere	133
6 Compile Transformers	135
defun compExpression	135
6.1 Handline Category DEF forms	138
defplist compDefine plist	140
defun compDefine	140
defun compDefine1	141
defun compDefineAddSignature	143
defun compDefineFunctor	144
defun compDefineFunctor1	144
defun compDefineCapsuleFunction	151
defun compInternalFunction	155
defun compDefWhereClause	155
defun compDefineCategory	158
defun compDefineCategory1	158
defun compDefineCategory2	159
defun compDefineLisplib	163
defun compileDocumentation	165
defun compArgumentConditions	166
defun compileCases	167
defun compFunctorBody	168
defun compile	169
defvar \$NoValueMode	172
defvar \$EmptyMode	172
defun hasFullSignature	172
defun addEmptyCapsuleIfNecessary	173
defun getTargetFromRhs	173
defun giveFormalParametersValues	174
defun macroExpandInPlace	174
defun macroExpand	174
defun macroExpandList	175
defun makeCategoryPredicates	175

defun mkCategoryPackage	176
defun mkEvaluableCategoryForm	178
defun encodeFunctionName	179
defun mkRepetitionAssoc	180
defun splitEncodedFunctionName	180
defun encodeItem	181
defun getCaps	181
defun constructMacro	182
defun spadCompileOrSetq	182
defun compileConstructor	183
defun compileConstructor1	184
defun compAndDefine	185
defun putInLocalDomainReferences	185
defun NRTputInTail	185
defun NRTputInHead	186
defun getArgumentModeOrMoan	187
defun augLisplibModemapsFromCategory	187
defun mkAlistOfExplicitCategoryOps	189
defun flattenSignatureList	190
defun interactiveModemapForm	191
defun replaceVars	192
defun fixUpPredicate	192
defun orderPredicateItems	193
defun signatureTran	193
defun orderPredTran	194
defun isDomainSubst	196
defun moveORsOutside	197
defun substVars	198
defun modemapPattern	199
defun evalAndRwriteLispForm	200
defun rwriteLispForm	200
defun mkConstructor	201
defun unloadOneConstructor	201
defun lisplibDoRename	201
defun initializeLisplib	202
defun writeLib1	203
defun finalizeLisplib	203
defun getConstructorOpsAndAtts	205
defun getCategoryOpsAndAtts	205
defun getSlotFromCategoryForm	206
defun transformOperationAlist	206
defun getFunctorOpsAndAtts	208
defun getSlotFromFunctor	208
defun compMakeCategoryObject	208
defun mergeSignatureAndLocalVarAlists	209
defun lisplibWrite	209

defun isCategoryPackageName	210
defun NRTgetLookupFunction	210
defun NRTgetLocalIndex	211
defun augmentLisplibModemapsFromFunctor	212
defun allLASSOCs	213
defun formal2Pattern	214
defun mkDatabasePred	214
defun disallowNilAttribute	214
defun bootStrapError	215
defun reportOnFunctorCompilation	215
defun displayMissingFunctions	216
defun makeFunctorArgumentParameters	217
defun genDomainViewList0	219
defun genDomainViewList	219
defun genDomainView	219
defun genDomainOps	220
defun mkOpVec	221
defun AssocBarGensym	222
defun orderByDependency	222
6.2 Code optimization routines	223
defun optimizeFunctionDef	223
defun optimize	225
defun optXLAMCond	226
defun optCONDtail	226
defvar \$BasicPredicates	227
defun optPredicateIfTrue	227
defun optIF2COND	227
defun subrname	228
Special case optimizers	228
defplist optCall	229
defun Optimize “call” expressions	229
defun optPackageCall	230
defun optCallSpecially	231
defun optSpecialCall	232
defun compileTimeBindingOf	233
defun optCallEval	233
defplist optSEQ	234
defun optSEQ	234
defplist optEQ	235
defun optEQ	236
defplist optMINUS	236
defun optMINUS	236
defplist optQSMINUS	237
defun optQSMINUS	237
defplist opt-	237
defun opt-	238

defplist optLESSP	238
defun optLESSP	238
defplist optSPADCALL	239
defun optSPADCALL	239
defplist optSuchthat	240
defun optSuchthat	240
defplist optCatch	240
defun optCatch	240
defplist optCond	242
defun optCond	242
defun EqualBarGensym	244
defplist optMkRecord	245
defun optMkRecord	245
defplist optRECORDELT	245
defun optRECORDELT	245
defplist optSETRECORDELT	246
defun optSETRECORDELT	246
defplist optRECORDCOPY	247
defun optRECORDCOPY	247
6.3 Functions to manipulate modemaps	248
defun addDomain	248
defun unknownTypeError	249
defun isFunctor	249
defun getDomainsInScope	250
defun putDomainsInScope	250
defun isSuperDomain	251
defun addNewDomain	251
defun augModemapsFromDomain	252
defun augModemapsFromDomain1	252
defun substituteCategoryArguments	253
defun addConstructorModemaps	254
defun getModemap	254
defun compApplyModemap	255
defun compMapCond	256
defun compMapCond'	257
defun compMapCond"	257
defun compMapCondFun	258
defun getUniqueSignature	259
defun getUniqueModemap	259
defun getModemapList	259
defun getModemapListFromDomain	260
defun domainMember	260
defun augModemapsFromCategory	260
defun addEltModemap	261
defun mkNewModemapList	262
defun insertModemap	263

	defun mergeModemap	263
	defun TruthP	264
	defun evalAndSub	265
	defun getOperationAlist	265
	defvar \$FormalMapVariableList	266
	defun substNames	266
	defun augModemapsFromCategoryRep	267
6.4	Maintaining Modemaps	268
	defun addModemapKnown	268
	defun addModemap	269
	defun addModemap0	269
	defun addModemap1	270
6.5	Indirect called comp routines	270
	defplist compAdd plist	271
	defun compAdd	271
	defun compTuple2Record	273
	defplist compCapsule plist	273
	defun compCapsule	274
	defun compCapsuleInner	274
	defun processFunctor	275
	defun compCapsuleItems	275
	defun compSingleCapsuleItem	276
	defun doIt	276
	defun doItIf	281
	defun isMacro	282
	defplist compCase plist	283
	defun compCase	283
	defun compCase1	284
	defplist compCat plist	284
	defplist compCat plist	285
	defplist compCat plist	285
	defun compCat	285
	defplist compCategory plist	286
	defun compCategory	286
	defun compCategoryItem	287
	defun mkExplicitCategoryFunction	288
	defun mustInstantiate	289
	defun wrapDomainSub	290
	defplist compColon plist	290
	defun compColon	290
	defun makeCategoryForm	293
	defplist compCons plist	294
	defun compCons	294
	defun compCons1	294
	defplist compConstruct plist	295
	defun compConstruct	295

defplist compConstructorCategory plist	296
defplist compConstructorCategory plist	296
defplist compConstructorCategory plist	297
defplist compConstructorCategory plist	297
defun compConstructorCategory	297
defun getAbbreviation	298
defun mkAbbrev	298
defun addSuffix	299
defun alistSize	299
defun getSignatureFromMode	299
defun getSpecialCaseAssoc	300
defun addArgumentConditions	300
defun stripOffSubdomainConditions	301
defun stripOffArgumentConditions	302
defun getSignature	302
defun checkAndDeclare	304
defun hasSigInTargetCategory	304
defun getArgumentMode	305
defplist compElt plist	306
defun compElt	306
defplist compExit plist	307
defun compExit	308
defplist compHas plist	308
defun compHas	309
defun compHasFormat	309
defun mkList	310
defplist compIf plist	310
defun compIf	311
defun compFromIf	312
defun canReturn	312
defun compBoolean	314
defun getSuccessEnvironment	314
defun getInverseEnvironment	316
defun getUnionMode	317
defun isUnionMode	317
defplist compImport plist	318
defun compImport	318
defplist compIs plist	318
defun compIs	318
defplist compJoin plist	319
defun compJoin	319
defun compForMode	321
defplist compLambda plist	321
defun compLambda	321
defplist compLeave plist	322
defun compLeave	323

defplist compMacro plist	323
defun compMacro	323
defplist compPretend plist	324
defun compPretend	324
defplist compQuote plist	325
defun compQuote	326
defplist compReduce plist	326
defun compReduce	326
defun compReduce1	326
defplist compRepeatOrCollect plist	328
defun compRepeatOrCollect	328
defun compRepeatOrCollect	329
defplist compReturn plist	331
defun compReturn	331
defplist compSeq plist	332
defun compSeq	332
defun compSeq1	332
defun replaceExitEtc	333
defun convertOrCroak	334
defun compSeqItem	334
defplist compSetq plist	335
defplist compSetq plist	335
defun compSetq	335
defun compSetq1	335
defun uncons	336
defun setqMultiple	337
defun setqMultipleExplicit	339
defun setqSetelt	340
defun setqSingle	340
defun NRTassocIndex	342
defun assignError	342
defun outputComp	343
defun maxSuperType	344
defun isDomainForm	344
defun isDomainConstructorForm	344
defplist compString plist	345
defun compString	345
defplist compSubDomain plist	346
defun compSubDomain	346
defun compSubDomain1	346
defun lispize	347
defplist compSubsetCategory plist	348
defun compSubsetCategory	348
defplist compSuchthat plist	348
defun compSuchthat	349
defplist compVector plist	349

	defun compVector	349
	defplist compWhere plist	350
	defun compWhere	350
6.6	Functions for coercion	351
	defun coerce	351
	defun coerceEasy	352
	defun coerceSubset	353
	defun coerceHard	354
	defun coerceExtraHard	355
	defun hasType	356
	defun coerceable	356
	defun coerceExit	357
	defplist compAtSign plist	357
	defun compAtSign	357
	defplist compCoerce plist	358
	defun compCoerce	358
	defun compCoerce1	359
	defun coerceByModemap	359
	defun autoCoerceByModemap	360
	defun resolve	361
	defun mkUnion	362
	defun This orders Unions	362
	defun modeEqualSubst	363
7	Post Transformers	365
7.1	Direct called postparse routines	365
	defun postTransform	365
	defun postTran	366
	defun postOp	367
	defun postAtom	367
	defun postTranList	368
	defun postScriptsForm	368
	defun postTranScripts	368
	defun postTransformCheck	369
	defun postcheck	369
	defun postError	370
	defun postForm	370
7.2	Indirect called postparse routines	371
	defplist postAdd plist	372
	defun postAdd	372
	defun postCapsule	373
	defun postBlockItemList	373
	defun postBlockItem	373
	defplist postAtSign plist	374
	defun postAtSign	374
	defun postType	375

defplist postBigFloat plist	375
defun postBigFloat	376
defplist postBlock plist	376
defun postBlock	376
defplist postCategory plist	377
defun postCategory	377
defun postCollect,finish	377
defun postMakeCons	378
defplist postCollect plist	379
defun postCollect	379
defun postIteratorList	380
defplist postColon plist	380
defun postColon	381
defplist postColonColon plist	381
defun postColonColon	381
defplist postComma plist	382
defun postComma	382
defun comma2Tuple	382
defun postFlatten	382
defplist postConstruct plist	383
defun postConstruct	383
defun postTranSegment	384
defplist postDef plist	384
defun postDef	384
defun postDefArgs	386
defplist postExit plist	386
defun postExit	387
defplist postIf plist	387
defun postIf	387
defplist postin plist	388
defun postin	388
defun postInSeq	388
defplist postIn plist	389
defun postIn	389
defplist postJoin plist	389
defun postJoin	390
defplist postMapping plist	390
defun postMapping	390
defplist postMDef plist	391
defun postMDef	391
defplist postPretend plist	392
defun postPretend	392
defplist postQUOTE plist	392
defun postQUOTE	393
defplist postReduce plist	393
defun postReduce	393

	defplist postRepeat plist	394
	defun postRepeat	394
	defplist postScripts plist	394
	defun postScripts	394
	defplist postSemiColon plist	395
	defun postSemiColon	395
	defun postFlattenLeft	395
	defplist postSignature plist	396
	defun postSignature	396
	defun removeSuperfluousMapping	396
	defun killColons	397
	defplist postSlash plist	397
	defun postSlash	397
	defplist postTuple plist	398
	defun postTuple	398
	defplist postTupleCollect plist	398
	defun postTupleCollect	398
	defplist postWhere plist	399
	defun postWhere	399
	defplist postWith plist	399
	defun postWith	400
7.3	Support routines	400
	defun setDefOp	400
	defun aplTran	401
	defun aplTran1	401
	defun aplTranList	403
	defun hasAplExtension	403
	defun deepestExpression	404
	defun containsBang	404
	defun getScriptName	404
	defun decodeScripts	405
8	DEF forms	407
	defvar \$defstack	407
	defvar \$is-spill	407
	defvar \$is-spill-list	407
	defvar \$v1	408
	defvar \$is-gensymlist	408
	defvar \$initial-gensym	408
	defvar \$is-eqlist	408
	defun hackforis	408
	defun hackforis1	409
	defun unTuple	409
	defun errhuh	409

9	PARSE forms	411
9.1	The original meta specification	411
9.2	The PARSE code	416
	defvar \$tmptok	416
	defvar \$tok	416
	defvar \$ParseMode	417
	defvar \$definition-name	417
	defvar \$lablasoc	417
	defun PARSE-NewExpr	417
	defun PARSE-Command	418
	defun PARSE-SpecialKeyWord	418
	defun PARSE-SpecialCommand	419
	defun PARSE-TokenCommandTail	419
	defun PARSE-TokenOption	420
	defun PARSE-TokenList	420
	defun PARSE-CommandTail	421
	defun PARSE-PrimaryOrQM	421
	defun PARSE-Option	422
	defun PARSE-Statement	422
	defun PARSE-InfixWith	423
	defun PARSE-With	423
	defun PARSE-Category	423
	defun PARSE-Expression	425
	defun PARSE-Import	425
	defun PARSE-Expr	426
	defun PARSE-LedPart	426
	defun PARSE-NudPart	426
	defun PARSE-Operation	427
	defun PARSE-leftBindingPowerOf	427
	defun PARSE-rightBindingPowerOf	428
	defun PARSE-getSemanticForm	428
	defun PARSE-Prefix	428
	defun PARSE-Infix	429
	defun PARSE-TokTail	430
	defun PARSE-Qualification	430
	defun PARSE-Reduction	431
	defun PARSE-ReductionOp	431
	defun PARSE-Form	431
	defun PARSE-Application	432
	defun PARSE-Label	433
	defun PARSE-Selector	433
	defun PARSE-PrimaryNoFloat	434
	defun PARSE-Primary	434
	defun PARSE-Primary1	434
	defun PARSE-Float	435
	defun PARSE-FloatBase	436

defun PARSE-FloatBasePart	436
defun PARSE-FloatExponent	437
defun PARSE-Enclosure	438
defun PARSE-IntegerTok	438
defun PARSE-FormalParameter	439
defun PARSE-FormalParameterTok	439
defun PARSE-Quad	439
defun PARSE-String	439
defun PARSE-VarForm	440
defun PARSE-Scripts	440
defun PARSE-ScriptItem	441
defun PARSE-Name	441
defun PARSE-Data	442
defun PARSE-Sexpr	442
defun PARSE-Sexpr1	442
defun PARSE-NBGlyphTok	443
defun PARSE-GlyphTok	444
defun PARSE-AnyId	444
defun PARSE-Sequence	445
defun PARSE-Sequence1	445
defun PARSE-OpenBracket	446
defun PARSE-OpenBrace	446
defun PARSE-IteratorTail	447
defun PARSE-Iterator	447
The PARSE implicit routines	448
defun PARSE-Suffix	448
defun PARSE-SemiColon	449
defun PARSE-Return	449
defun PARSE-Exit	449
defun PARSE-Leave	450
defun PARSE-Seg	450
defun PARSE-Conditional	451
defun PARSE-ElseClause	451
defun PARSE-Loop	452
defun PARSE-LabelExpr	452
defun PARSE-FloatTok	453
9.3 The PARSE support routines	453
String grabbing	454
defun match-string	454
defun skip-blanks	454
defun token-lookahead-type	455
defun match-advance-string	455
defun initial-substring-p	456
defun quote-if-string	456
defun escape-keywords	457
defun isTokenDelimiter	457

defun underscore	458
Token Handling	458
defun getToken	458
defun unget-tokens	458
defun match-current-token	459
defun match-token	460
defun match-next-token	460
defun current-symbol	460
defun make-symbol-of	460
defun current-token	461
defun try-get-token	461
defun next-token	462
defun advance-token	462
defvar \$XTokenReader	463
defun get-token	463
Character handling	463
defun current-char	463
defun next-char	463
defun char-eq	464
defun char-ne	464
Error handling	464
defvar \$meta-error-handler	464
defun meta-syntax-error	465
Floating Point Support	465
defun floatexpid	465
Dollar Translation	465
defun dollarTran	465
Applying metagrammatical elements of a production (e.g., Star).	466
defmacro Bang	466
defmacro must	466
defun action	467
defun optional	467
defmacro star	467
Stacking and retrieving reductions of rules.	468
defvar \$reduce-stack	468
defmacro reduce-stack-clear	468
defun push-reduction	468
10 Comment Recording	469
10.1 Comment Recording Layer 0 – API	470
defun recordSignatureDocumentation	470
defun recordAttributeDocumentation	470
10.2 Comment Recording Layer 1	471
defun recordDocumentation	471
10.3 Comment Recording Layer 2	471
defun collectComBlock	471

10.4 Comment Recording Layer 3	472
defun recordHeaderDocumentation	472
defun collectAndDeleteAssoc	472
11 Category handling	475
defun getConstructorExports	475
12 Building libdb.text	477
defun extendLocalLibdb	477
defun buildLibdb	478
defun buildLibdbString	480
defun dbReadLines	481
defun purgeNewConstructorLines	481
defun dbWriteLines	481
defun buildLibdbConEntry	482
defun buildLibOps	484
defun buildLibOp	484
defun buildLibAttrs	485
defun buildLibAttr	485
defun screenLocalLine	486
13 Comment Syntax Checking	487
13.1 Comment Checking Layer 0 – API	492
defun finalizeDocumentation	492
13.2 Comment Checking Layer 1	495
defun transDocList	495
13.3 Comment Checking Layer 2	496
defun transDoc	496
13.4 Comment Checking Layer 3	497
defun transformAndRecheckComments	497
13.5 Comment Checking Layer 4	498
defun checkComments	498
defun checkRewrite	499
13.6 Comment Checking Layer 5	501
defun checkArguments	501
defun checkBalance	501
13.7 Comment Checking Layer 6	502
defun checkBeginEnd	502
defun checkDecorate	504
defun checkDecorateForHt	506
defun checkDocError1	507
defun checkFixCommonProblem	508
defun checkGetLispFunctionName	508
defun checkHTargs	509
defun checkRecordHash	509
defun checkTexht	512

defun checkTransformFirsts	513
defun checkTrim	516
13.8 Comment Checking Layer 7	517
defun checkDocError	517
defun checkRemoveComments	518
defun checkSkipToken	518
defun checkSplit2Words	518
13.9 Comment Checking Layer 8	519
defun checkAddIndented	519
defun checkDocMessage	519
defun checkExtract	520
defun checkGetArgs	521
defun checkGetMargin	522
defun checkGetParse	522
defun checkGetStringBeforeRightBrace	523
defun checkLeEg	523
defun checkIndentedLines	524
defun checkSkipIdentifierToken	525
defun checkSkipOpToken	525
defun checkSplitBrace	525
defun checkTrimCommented	526
defun newString2Words	527
13.10 Comment Checking Layer 9	527
defun checkAddBackSlashes	527
defun checkAddMacros	528
defun checkAddPeriod	529
defun checkAddSpaceSegments	529
defun checkAddSpaces	530
defun checkAlphabetic	531
defun checkLeEgfun	531
defun checkIsValidType	532
defun checkLookForLeftBrace	533
defun checkLookForRightBrace	533
defun checkNumOfArgs	534
defun checkSayBracket	534
defun checkSkipBlanks	534
defun checkSplitBackslash	535
defun checkSplitOn	536
defun checkSplitPunctuation	537
defun firstNonBlankPosition	538
defun getMatchingRightPren	538
defun hasNoVowels	539
defun htcharPosition	539
defun newWordFrom	540
defun removeBackslashes	541
defun whoOwns	541

14 Utility Functions	543
defun translablel	543
defun translable1	543
defun displayPreCompilationErrors	544
defun bumperrorcount	545
defun parseTranCheckForRecord	545
defun makeSimplePredicateOrNil	546
defun parse-spadstring	546
defun parse-string	546
defun parse-identifier	547
defun parse-number	547
defun parse-keyword	548
defun parse-argument-designator	548
defun print-package	549
defun checkWarning	549
defun tuple2List	549
defmacro pop-stack-1	550
defmacro pop-stack-2	550
defmacro pop-stack-3	551
defmacro pop-stack-4	551
defmacro nth-stack	551
defun Pop-Reduction	552
defun addclose	552
defun blankp	552
defun drop	553
defun escaped	553
defvar \$comblocklist	553
defun fincomblock	553
defun indent-pos	554
defun infixtok	555
defun is-console	555
defun next-tab-loc	555
defun nonblankloc	555
defun parseprint	556
defun skip-to-endif	556
15 The Compiler	557
defvar \$newConlist	557
15.1 Compiling EQ.spad	557
15.2 The top level compiler command	560
defun compiler	562
defun compileSpad2Cmd	565
defun compileSpadLispCmd	568
compilerDoitWithScreenedLisplib	569
defun compilerDoit	570
defun /rq	571

defun /rf	571
defun /RQ,LIB	572
defun /rf-1	572
defun spad	573
defun Interpreter interface to the compiler	576
defun compTopLevel	586
defun print-defun	587
defun def-rename	587
defun compOrCroak	588
defun compOrCroak1	589
defun comp	590
defun compNoStacking	590
defun compNoStacking1	591
defun comp2	591
defun comp3	592
defun applyMapping	593
defun compApply	595
defun compTypeOf	596
defun compColonInside	596
defun compAtom	597
defun compAtomWithModemap	598
defun transImplementation	599
defun convert	600
defun primitiveType	600
defun compSymbol	600
defun compList	602
defun compForm	602
defun compForm1	603
defun compToApply	605
defun compApplication	605
defun getFormModemaps	607
defun eltModemapFilter	608
defun seteltModemapFilter	609
defun compExpressionList	609
defun compForm2	610
defun compForm3	612
defun compFocompFormWithModemap	613
defun substituteIntoFunctorModemap	614
defun compFormPartiallyBottomUp	615
defun compFormMatch	615
defun compUniquely	616
defun compArgumentsAndTryAgain	616
defun compWithMappingMode	617
defun compWithMappingModel	617
defun extractCodeAndConstructTriple	622
defun hasFormalMapVariable	623

defun argsToSig	623
defun compMakeDeclaration	624
defun modifyModeStack	625
defun Create a list of unbound symbols	625
defun compOrCroak1,compactify	626
defun Compiler/Interpreter interface	627
defun recompile-lib-file-if-necessary	627
defun spad-fixed-arg	627
defun compile-lib-file	628
defun compileFileQuietly	628
defvar \$byConstructors	629
defvar \$constructorsSeen	629
16 Level 1	631
defvar \$current-fragment	631
defun read-a-line	631
17 Level 0	633
17.1 Line Handling	633
Line Buffer	633
defstruct \$line	633
defvar \$current-line	634
defmacro line-clear	634
defun line-print	634
defun line-at-end-p	634
defun line-past-end-p	635
defun line-next-char	635
defun line-advance-char	635
defun line-current-segment	636
defun line-new-line	636
defun next-line	636
defun Advance-Char	637
defun storeblanks	637
defun initial-substring	637
defun get-a-line	638
18 The Chunks	639
19 Index	657

New Foreword

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

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

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

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

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

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

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

Chapter 1

The Axiom Compiler

1.1 Makefile

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

Chapter 2

Overview

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

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

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

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

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

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

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

Next we will look at the top level data structures used by the compiler. Unfortunately, the compiler uses a large number of "global variables" to pass information and alter control flow. Some of these are used by many routines and some of these are very local to a small subset or a recursion. We will cover the minor ones as they arise.

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

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

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

2.1 The Input

```
)abbrev domain EQ Equation
--FOR THE BENEFIT OF LIBAXO GENERATION
++ Author: Stephen M. Watt, enhancements by Johannes Grabmeier
++ Date Created: April 1985
++ Date Last Updated: June 3, 1991; September 2, 1992
++ Basic Operations: =
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: equation
++ Examples:
++ References:
++ Description:
++ Equations as mathematical objects. All properties of the basis domain,
++ e.g. being an abelian group are carried over the equation domain, by
++ performing the structural operations on the left and on the
++ right hand side.
-- The interpreter translates "=" to "equation". Otherwise, it will
-- find a modemap for "=" in the domain of the arguments.

Equation(S: Type): public == private where
  Ex ==> OutputForm
  public ==> Type with
    "=": (S, S) -> $
      ++ a=b creates an equation.
    equation: (S, S) -> $
      ++ equation(a,b) creates an equation.
    swap: $ -> $
      ++ swap(eq) interchanges left and right hand side of equation eq.
    lhs: $ -> S
      ++ lhs(eqn) returns the left hand side of equation eqn.
    rhs: $ -> S
      ++ rhs(eqn) returns the right hand side of equation eqn.
```



```

map: (S -> S, $) -> $
  ++ map(f,eqn) constructs a new equation by applying f to both
  ++ sides of eqn.
if S has InnerEvalable(Symbol,S) then
  InnerEvalable(Symbol,S)
if S has SetCategory then
  SetCategory
  CoercibleTo Boolean
  if S has Evalable(S) then
    eval: ($, $) -> $
      ++ eval(eqn, x=f) replaces x by f in equation eqn.
    eval: ($, List $) -> $
      ++ eval(eqn, [x1=v1, ... xn=vn]) replaces xi by vi in equation eqn.
if S has AbelianSemiGroup then
  AbelianSemiGroup
  "+": (S, $) -> $
    ++ x+eqn produces a new equation by adding x to both sides of
    ++ equation eqn.
  "+": ($, S) -> $
    ++ eqn+x produces a new equation by adding x to both sides of
    ++ equation eqn.
if S has AbelianGroup then
  AbelianGroup
  leftZero : $ -> $
    ++ leftZero(eq) subtracts the left hand side.
  rightZero : $ -> $
    ++ rightZero(eq) subtracts the right hand side.
  "-": (S, $) -> $
    ++ x-eqn produces a new equation by subtracting both sides of
    ++ equation eqn from x.
  "-": ($, S) -> $
    ++ eqn-x produces a new equation by subtracting x from
    ++ both sides of equation eqn.
if S has SemiGroup then
  SemiGroup
  "*": (S, $) -> $
    ++ x*eqn produces a new equation by multiplying both sides of
    ++ equation eqn by x.
  "*": ($, S) -> $
    ++ eqn*x produces a new equation by multiplying both sides of
    ++ equation eqn by x.
if S has Monoid then
  Monoid
  leftOne : $ -> Union($,"failed")
    ++ leftOne(eq) divides by the left hand side, if possible.
  rightOne : $ -> Union($,"failed")
    ++ rightOne(eq) divides by the right hand side, if possible.
if S has Group then
  Group
  leftOne : $ -> Union($,"failed")

```

```

    ++ leftOne(eq) divides by the left hand side.
    rightOne : $ -> Union($,"failed")
    ++ rightOne(eq) divides by the right hand side.
if S has Ring then
  Ring
  BiModule(S,S)
if S has CommutativeRing then
  Module(S)
  --Algebra(S)
if S has IntegralDomain then
  factorAndSplit : $ -> List $
    ++ factorAndSplit(eq) make the right hand side 0 and
    ++ factors the new left hand side. Each factor is equated
    ++ to 0 and put into the resulting list without repetitions.
if S has PartialDifferentialRing(Symbol) then
  PartialDifferentialRing(Symbol)
if S has Field then
  VectorSpace(S)
  "/" : ($, $) -> $
    ++ e1/e2 produces a new equation by dividing the left and right
    ++ hand sides of equations e1 and e2.
  inv : $ -> $
    ++ inv(x) returns the multiplicative inverse of x.
if S has ExpressionSpace then
  subst : ($, $) -> $
    ++ subst(eq1,eq2) substitutes eq2 into both sides of eq1
    ++ the lhs of eq2 should be a kernel

private ==> add
  Rep := Record(lhs: S, rhs: S)
  eq1,eq2: $
  s : S
  if S has IntegralDomain then
    factorAndSplit eq ==
      (S has factor : S -> Factored S) =>
        eq0 := rightZero eq
        [equation(rcf.factor,0) for rcf in factors factor lhs eq0]
        [eq]
  l:S = r:S      == [l, r]
  equation(l, r) == [l, r]  -- hack! See comment above.
  lhs eqn        == eqn.lhs
  rhs eqn        == eqn.rhs
  swap eqn       == [rhs eqn, lhs eqn]
  map(fn, eqn)   == equation(fn(eqn.lhs), fn(eqn.rhs))

if S has InnerEvalable(Symbol,S) then
  s:Symbol
  ls:List Symbol
  x:S
  lx:List S

```

```

    eval(eqn,s,x) == eval(eqn.lhs,s,x) = eval(eqn.rhs,s,x)
    eval(eqn,ls,lx) == eval(eqn.lhs,ls,lx) = eval(eqn.rhs,ls,lx)
if S has Evalable(S) then
    eval(eqn1:$, eqn2:$):$ ==
        eval(eqn1.lhs, eqn2 pretend Equation S) =
            eval(eqn1.rhs, eqn2 pretend Equation S)
    eval(eqn1:$, leqn2:List $):$ ==
        eval(eqn1.lhs, leqn2 pretend List Equation S) =
            eval(eqn1.rhs, leqn2 pretend List Equation S)
if S has SetCategory then
    eq1 = eq2 == (eq1.lhs = eq2.lhs)@Boolean and
                (eq1.rhs = eq2.rhs)@Boolean
    coerce(eqn:$):Ex == eqn.lhs::Ex = eqn.rhs::Ex
    coerce(eqn:$):Boolean == eqn.lhs = eqn.rhs
if S has AbelianSemiGroup then
    eq1 + eq2 == eq1.lhs + eq2.lhs = eq1.rhs + eq2.rhs
    s + eq2 == [s,s] + eq2
    eq1 + s == eq1 + [s,s]
if S has AbelianGroup then
    - eq == (- lhs eq) = (-rhs eq)
    s - eq2 == [s,s] - eq2
    eq1 - s == eq1 - [s,s]
    leftZero eq == 0 = rhs eq - lhs eq
    rightZero eq == lhs eq - rhs eq = 0
    0 == equation(0$S,0$S)
    eq1 - eq2 == eq1.lhs - eq2.lhs = eq1.rhs - eq2.rhs
if S has SemiGroup then
    eq1:$ * eq2:$ == eq1.lhs * eq2.lhs = eq1.rhs * eq2.rhs
    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)]

```

2.2 The Output, the EQ.nrlib directory

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

```
)abbrev domain EQ Equation
```

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

- one of “category”, “domain”, or “package”
- the abbreviation for this domain (8 Uppercase Characters maximum)
- the name of this domain

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

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

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

2.3 The code.lsp and EQ.lsp files

```
(/VERSIONCHECK 2)
```

```
(DEFUN |EQ;factorAndSplit;$L;1| (|eq| $)
  (PROG (|eq0| #:G1403 |rcf| #:G1404)
    (RETURN
      (SEQ (COND
        ((|HasSignature| (QREFELT $ 6)
          (LIST ' |factor|
            (LIST (LIST ' |Factored|
              (|devaluate| (QREFELT $ 6)))
              (|devaluate| (QREFELT $ 6))))))
          (SEQ (LETT |eq0| (SPADCALL |eq| (QREFELT $ 8))
            |EQ;factorAndSplit;$L;1|)
            (EXIT (PROGN
              (LETT #:G1403 NIL |EQ;factorAndSplit;$L;1|)
              (SEQ (LETT |rcf| NIL
                |EQ;factorAndSplit;$L;1|)
                (LETT #:G1404
                  (SPADCALL
                    (SPADCALL
                      (SPADCALL |eq0| (QREFELT $ 9))
                      (QREFELT $ 11))
                      (QREFELT $ 15))
                    |EQ;factorAndSplit;$L;1|)
                  G190
                  (COND
                    ((OR (ATOM #:G1404)
                      (PROGN
                        (LETT |rcf| (CAR #:G1404)
                          |EQ;factorAndSplit;$L;1|)
                          NIL))
                      (GO G191)))
```

```

(SEQ (EXIT
      (LETT #:G1403
            (CONS
              (SPADCALL (QCAR |rcf|)
                        (|spadConstant| $ 16)
                        (QREFELT $ 17))
              #:G1403)
      |EQ;factorAndSplit;$L;1|)))
(LETT #:G1404 (CDR #:G1404)
      |EQ;factorAndSplit;$L;1|)
(GO G190) G191
(EXIT (NREVERSEO #:G1403))))))
('T (LIST |eq|))))))

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

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

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

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

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

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

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

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

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

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

(DEFUN |EQ;eval;$SS$;8| (|eqn| |s| |x| $)
  (SPADCALL
    (SPADCALL (QCAR |eqn|) |s| |x| (QREFELT $ 26))
    (SPADCALL (QCDR |eqn|) |s| |x| (QREFELT $ 26))
    (QREFELT $ 20)))

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

```

```

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

(DEFUN |EQ;eval;$L$;11| (|eqn1| |eqn2| $)
  (SPADCALL
    (SPADCALL (QCAR |eqn1|) |eqn2| (QREFELT $ 36))
    (SPADCALL (QCDR |eqn1|) |eqn2| (QREFELT $ 36))
    (QREFELT $ 20)))

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

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

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

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

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

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

(DEFUN |EQ;-;2$;18| (|eq| $)
  (SPADCALL
    (SPADCALL (SPADCALL |eq| (QREFELT $ 9)) (QREFELT $ 50))
    (SPADCALL (SPADCALL |eq| (QREFELT $ 21)) (QREFELT $ 50))
    (QREFELT $ 20)))

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

(DEFUN |EQ;-;$S$;20| (|eq1| |s| $)

```

```

(SPADCALL |eq1| (CONS |s| |s|) (QREFELT $ 52)))

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

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

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

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

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

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

(DEFUN |EQ;*;S2$;27| (|l| |eqn| $)
  (SPADCALL
    (SPADCALL |l| (QCAR |eqn|) (QREFELT $ 58))
    (SPADCALL |l| (QCDR |eqn|) (QREFELT $ 58))
    (QREFELT $ 20)))

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

```



```

(SPADCALL (QCDR |eqn|) |1| (QREFELT $ 58))
(QREFELT $ 20)))

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

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

(DEFUN |EQ;leftOne;$U;31| (|eq| $)
  (PROG (|re|)
    (RETURN
      (SEQ
        (LETT |re|
          (SPADCALL (SPADCALL |eq| (QREFELT $ 9)) (QREFELT $ 65))
          |EQ;leftOne;$U;31|)
        (EXIT
          (COND
            ((QEQCAR |re| 1) (CONS 1 "failed"))
            ('T
              (CONS 0
                (SPADCALL
                  (|spadConstant| $ 62)
                  (SPADCALL (SPADCALL |eq| (QREFELT $ 21)) (QCDR |re|) (QREFELT $ 58))
                  (QREFELT $ 20))))))))))

(DEFUN |EQ;rightOne;$U;32| (|eq| $)
  (PROG (|re|)
    (RETURN

```

```

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

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

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

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

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

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

(DEFUN |EQ;factorAndSplit;$L;38| (|eq| $)
  (PROG (#:G1488 #:G1489 |eq0| |p| #:G1490 |rcf| #:G1491)
    (RETURN
      (SEQ (COND
        ((|HasSignature| (QREFELT $ 6)
          (LIST 'factor|
            (LIST (LIST 'Factored|
              (|devaluate| (QREFELT $ 6))))

```

```

(|devaluate| (QREFELT $ 6))))))
(SEQ (LETT |eq0| (SPADCALL |eq| (QREFELT $ 8))
      |EQ;factorAndSplit;$L;38|)
(EXIT (PROGN
      (LETT #:G1488 NIL |EQ;factorAndSplit;$L;38|)
      (SEQ (LETT |rcf| NIL
                |EQ;factorAndSplit;$L;38|)
            (LETT #:G1489
                  (SPADCALL
                    (SPADCALL
                      (SPADCALL |eq0| (QREFELT $ 9))
                      (QREFELT $ 11))
                      (QREFELT $ 15))
                    |EQ;factorAndSplit;$L;38|)
            G190
            (COND
              ((OR (ATOM #:G1489)
                   (PROGN
                     (LETT |rcf| (CAR #:G1489)
                           |EQ;factorAndSplit;$L;38|)
                     NIL))
               (GO G191)))
            (SEQ (EXIT
                  (LETT #:G1488
                        (CONS
                          (SPADCALL (QCAR |rcf|)
                                    (|spadConstant| $ 16)
                                    (QREFELT $ 17))
                          #:G1488)
                  |EQ;factorAndSplit;$L;38|)))
            (LETT #:G1489 (CDR #:G1489)
                  |EQ;factorAndSplit;$L;38|)
            (GO G190) G191
            (EXIT (NREVERSEO #:G1488))))))
((EQUAL (QREFELT $ 6) (|Polynomial| (|Integer|)))
 (SEQ (LETT |eq0| (SPADCALL |eq| (QREFELT $ 8))
      |EQ;factorAndSplit;$L;38|)
      (LETT |p| (SPADCALL |eq0| (QREFELT $ 9))
            |EQ;factorAndSplit;$L;38|)
      (EXIT (PROGN
            (LETT #:G1490 NIL |EQ;factorAndSplit;$L;38|)
            (SEQ (LETT |rcf| NIL
                      |EQ;factorAndSplit;$L;38|)
                  (LETT #:G1491
                        (SPADCALL
                          (SPADCALL |p| (QREFELT $ 80))
                          (QREFELT $ 83))
                        |EQ;factorAndSplit;$L;38|)
                  G190
                  (COND

```

```

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

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

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

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

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

(DEFUN |EQ;subst;3$;43| (|eq1| |eq2| $)
  (PROG (|eq3|)
    (RETURN
      (SEQ (LETT |eq3| |eq2| |EQ;subst;3$;43|)
        (EXIT (CONS (SPADCALL (SPADCALL |eq1| (QREFELT $ 9)) |eq3|
          (QREFELT $ 92))
          (SPADCALL (SPADCALL |eq1| (QREFELT $ 21)) |eq3|
            (QREFELT $ 92)))))))

(DEFUN |Equation| (#:G1503)
  (PROG ()
    (RETURN
      (PROG (#:G1504)
        (RETURN

```

```

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

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

```

```

(|HasCategory| |#1|
  '(|CommutativeRing|))
|Equation|)
(OR #:G1502 (|HasCategory| |#1| '(|Field|))
  (|HasCategory| |#1| '(|Ring|)))
(OR #:G1502
  (|HasCategory| |#1| '(|Field|)))
(LETT #:G1501
  (|HasCategory| |#1| '(|Monoid|))
  |Equation|)
(OR (|HasCategory| |#1| '(|Group|))
  #:G1501)
(LETT #:G1500
  (|HasCategory| |#1| '(|SemiGroup|))
  |Equation|)
(OR (|HasCategory| |#1| '(|Group|)) #:G1501
  #:G1500)
(LETT #:G1499
  (|HasCategory| |#1|
    '(|AbelianGroup|))
  |Equation|)
(OR (|HasCategory| |#1|
  '(|PartialDifferentialRing|
    (|Symbol|)))
  #:G1499 #:G1502
  (|HasCategory| |#1| '(|Field|))
  (|HasCategory| |#1| '(|Ring|)))
(OR #:G1499 #:G1501)
(LETT #:G1498
  (|HasCategory| |#1|
    '(|AbelianSemiGroup|))
  |Equation|)
(OR (|HasCategory| |#1|
  '(|PartialDifferentialRing|
    (|Symbol|)))
  #:G1499 #:G1498 #:G1502
  (|HasCategory| |#1| '(|Field|))
  (|HasCategory| |#1| '(|Ring|)))
(OR (|HasCategory| |#1|
  '(|PartialDifferentialRing|
    (|Symbol|)))
  #:G1499 #:G1498 #:G1502
  (|HasCategory| |#1| '(|Field|))
  (|HasCategory| |#1| '(|Group|)) #:G1501
  (|HasCategory| |#1| '(|Ring|)) #:G1500
  (|HasCategory| |#1| '(|SetCategory|))))
|Equation|))
(|haddProp| |$ConstructorCache| ' |Equation| (LIST DV$1)
  (CONS 1 $))
(|stuffDomainSlots| $)

```

```

(QSETREFV $ 6 |#1|)
(QSETREFV $ 7 (|Record| (|:| |lhs| |#1|) (|:| |rhs| |#1|)))
(COND
  ((|testBitVector| |pv$| 9)
    (QSETREFV $ 19
      (CONS (|dispatchFunction| |EQ;factorAndSplit;$L;1|) $))))
(COND
  ((|testBitVector| |pv$| 7)
    (PROGN
      (QSETREFV $ 27
        (CONS (|dispatchFunction| |EQ;eval;$SS$;8|) $))
      (QSETREFV $ 31
        (CONS (|dispatchFunction| |EQ;eval;$LL$;9|) $))))))
(COND
  ((|HasCategory| |#1| (LIST '|Evalable| (|devaluate| |#1|)))
    (PROGN
      (QSETREFV $ 34
        (CONS (|dispatchFunction| |EQ;eval;3$;10|) $))
      (QSETREFV $ 37
        (CONS (|dispatchFunction| |EQ;eval;$L$;11|) $))))))
(COND
  ((|testBitVector| |pv$| 2)
    (PROGN
      (QSETREFV $ 40
        (CONS (|dispatchFunction| |EQ;=;2$B;12|) $))
      (QSETREFV $ 44
        (CONS (|dispatchFunction| |EQ;coerce;$0f;13|) $))
      (QSETREFV $ 45
        (CONS (|dispatchFunction| |EQ;coerce;$B;14|) $))))))
(COND
  ((|testBitVector| |pv$| 23)
    (PROGN
      (QSETREFV $ 47 (CONS (|dispatchFunction| |EQ;+;3$;15|) $))
      (QSETREFV $ 48
        (CONS (|dispatchFunction| |EQ;+;S2$;16|) $))
      (QSETREFV $ 49
        (CONS (|dispatchFunction| |EQ;+;$S$;17|) $))))))
(COND
  ((|testBitVector| |pv$| 20)
    (PROGN
      (QSETREFV $ 51 (CONS (|dispatchFunction| |EQ;-;2$;18|) $))
      (QSETREFV $ 53
        (CONS (|dispatchFunction| |EQ;-;S2$;19|) $))
      (QSETREFV $ 54
        (CONS (|dispatchFunction| |EQ;-;$S$;20|) $))
      (QSETREFV $ 57
        (CONS (|dispatchFunction| |EQ;leftZero;2$;21|) $))
      (QSETREFV $ 8
        (CONS (|dispatchFunction| |EQ;rightZero;2$;22|) $))
      (QSETREFV $ 55

```

```

(CONS IDENTITY
  (FUNCALL (|dispatchFunction| |EQ;Zero;$;23|) $)))
(QSETREFV $ 52 (CONS (|dispatchFunction| |EQ;-;3$;24|) $))))
(COND
  ((|testBitVector| |pv$| 18)
    (PROGN
      (QSETREFV $ 59 (CONS (|dispatchFunction| |EQ;*;3$;25|) $))
      (QSETREFV $ 60
        (CONS (|dispatchFunction| |EQ;*;S2$;26|) $))
      (QSETREFV $ 60
        (CONS (|dispatchFunction| |EQ;*;S2$;27|) $))
      (QSETREFV $ 61
        (CONS (|dispatchFunction| |EQ;*;S$;28|) $))))))
(COND
  ((|testBitVector| |pv$| 16)
    (PROGN
      (QSETREFV $ 63
        (CONS IDENTITY
          (FUNCALL (|dispatchFunction| |EQ;One;$;29|) $)))
      (QSETREFV $ 66
        (CONS (|dispatchFunction| |EQ;recip;$U;30|) $))
      (QSETREFV $ 67
        (CONS (|dispatchFunction| |EQ;leftOne;$U;31|) $))
      (QSETREFV $ 68
        (CONS (|dispatchFunction| |EQ;rightOne;$U;32|) $))))))
(COND
  ((|testBitVector| |pv$| 6)
    (PROGN
      (QSETREFV $ 70
        (CONS (|dispatchFunction| |EQ;inv;2$;33|) $))
      (QSETREFV $ 67
        (CONS (|dispatchFunction| |EQ;leftOne;$U;34|) $))
      (QSETREFV $ 68
        (CONS (|dispatchFunction| |EQ;rightOne;$U;35|) $))))))
(COND
  ((|testBitVector| |pv$| 3)
    (PROGN
      (QSETREFV $ 73
        (CONS (|dispatchFunction| |EQ;characteristic;Nni;36|)
          $))
      (QSETREFV $ 76
        (CONS (|dispatchFunction| |EQ;*;I2$;37|) $))))))
(COND
  ((|testBitVector| |pv$| 9)
    (QSETREFV $ 19
      (CONS (|dispatchFunction| |EQ;factorAndSplit;$L;38|) $))))
(COND
  ((|testBitVector| |pv$| 4)
    (QSETREFV $ 85
      (CONS (|dispatchFunction| |EQ;differentiate;S$;39|) $))))

```



```

(COND
  ((|testBitVector| |pv$| 1)
   (PROGN
    (QSETREFV $ 88
     (CONS (|dispatchFunction| |EQ;dimension;Cn;40|) $))
    (QSETREFV $ 90 (CONS (|dispatchFunction| |EQ;/;3$;41|) $))
    (QSETREFV $ 70
     (CONS (|dispatchFunction| |EQ;inv;2$;42|) $))))))
(COND
  ((|testBitVector| |pv$| 10)
   (QSETREFV $ 93
    (CONS (|dispatchFunction| |EQ;subst;3$;43|) $))))
$)))

(setf (get '|Equation| '|infovec|)
  (LIST '#(NIL NIL NIL NIL NIL NIL (|local| |#1|) '|Rep|
    (0 . |rightZero|) |EQ;lhs;$;4| (|Factored| $)
    (5 . |factor|)
    (|Record| (|:| |factor| 6) (|:| |exponent| 74))
    (|List| 12) (|Factored| 6) (10 . |factors|) (15 . |Zero|)
    |EQ;equation;2S$;3| (|List| $) (19 . |factorAndSplit|)
    |EQ;=;2S$;2| |EQ;rhs;$;5| |EQ;swap;2$;6| (|Mapping| 6 6)
    |EQ;map;M2$;7| (|Symbol|) (24 . |eval|) (31 . |eval|)
    (|List| 25) (|List| 6) (38 . |eval|) (45 . |eval|)
    (|Equation| 6) (52 . |eval|) (58 . |eval|) (|List| 32)
    (64 . |eval|) (70 . |eval|) (|Boolean|) (76 . =) (82 . =)
    (|OutputForm|) (88 . |coerce|) (93 . =) (99 . |coerce|)
    (104 . |coerce|) (109 . +) (115 . +) (121 . +) (127 . +)
    (133 . -) (138 . -) (143 . -) (149 . -) (155 . -)
    (161 . |Zero|) (165 . -) (171 . |leftZero|) (176 . *)
    (182 . *) (188 . *) (194 . *) (200 . |One|) (204 . |One|)
    (|Union| $ '|"failed"|) (208 . |recip|) (213 . |recip|)
    (218 . |leftOne|) (223 . |rightOne|) (228 . |inv|)
    (233 . |inv|) (|NonNegativeInteger|)
    (238 . |characteristic|) (242 . |characteristic|)
    (|Integer|) (246 . |coerce|) (251 . *) (|Factored| 78)
    (|Polynomial| 74)
    (|MultivariateFactorize| 25 (|IndexedExponents| 25) 74 78)
    (257 . |factor|)
    (|Record| (|:| |factor| 78) (|:| |exponent| 74))
    (|List| 81) (262 . |factors|) (267 . |differentiate|)
    (273 . |differentiate|) (|CardinalNumber|)
    (279 . |coerce|) (284 . |dimension|) (288 . /) (294 . /)
    (|Equation| $) (300 . |subst|) (306 . |subst|)
    (|PositiveInteger|) (|List| 71) (|SingleInteger|)
    (|String|))
  '#(= 312 |zero?| 318 |swap| 323 |subtractIfCan| 328 |subst|
    334 |sample| 340 |rightZero| 344 |rightOne| 349 |rhs| 354
    |recip| 359 |one?| 364 |map| 369 |lhs| 375 |leftZero| 380
    |leftOne| 385 |latex| 390 |inv| 395 |hash| 400

```

```

|factorAndSplit| 405 |eval| 410 |equation| 436 |dimension|
442 |differentiate| 446 |conjugate| 472 |commutator| 478
|coerce| 484 |characteristic| 499 ^ 503 |Zero| 521 |One|
525 D 529 = 555 / 567 - 579 + 602 ** 620 * 638)
'((|unitsKnown| . 12) (|rightUnitary| . 3)
(|leftUnitary| . 3))
(CONS (|makeByteWordVec2| 25
      '(1 15 4 14 5 14 3 5 3 21 21 6 21 17 24 19 25 0 2
        25 2 7))
      (CONS '#(|VectorSpace&| |Module&|
                |PartialDifferentialRing&| NIL |Ring&| NIL NIL
                NIL NIL |AbelianGroup&| NIL |Group&|
                |AbelianMonoid&| |Monoid&| |AbelianSemiGroup&|
                |SemiGroup&| |SetCategory&| NIL NIL
                |BasicType&| NIL |InnerEvalable&|)
            (CONS '#(|VectorSpace| 6) (|Module| 6)
                  (|PartialDifferentialRing| 25)
                  (|BiModule| 6 6) (|Ring|)
                  (|LeftModule| 6) (|RightModule| 6)
                  (|Rng|) (|LeftModule| $$)
                  (|AbelianGroup|)
                  (|CancellationAbelianMonoid|) (|Group|)
                  (|AbelianMonoid|) (|Monoid|)
                  (|AbelianSemiGroup|) (|SemiGroup|)
                  (|SetCategory|) (|Type|)
                  (|CoercibleTo| 41) (|BasicType|)
                  (|CoercibleTo| 38)
                  (|InnerEvalable| 25 6))
            (|makeByteWordVec2| 97
              '(1 0 0 0 8 1 6 10 0 11 1 14 13 0 15 0
                6 0 16 1 0 18 0 19 3 6 0 0 25 6 26 3
                0 0 0 25 6 27 3 6 0 0 28 29 30 3 0 0
                0 28 29 31 2 6 0 0 32 33 2 0 0 0 0 34
                2 6 0 0 35 36 2 0 0 0 18 37 2 6 38 0
                0 39 2 0 38 0 0 40 1 6 41 0 42 2 41 0
                0 0 43 1 0 41 0 44 1 0 38 0 45 2 6 0
                0 0 46 2 0 0 0 0 47 2 0 0 6 0 48 2 0
                0 0 6 49 1 6 0 0 50 1 0 0 0 51 2 0 0
                0 0 52 2 0 0 6 0 53 2 0 0 0 6 54 0 0
                0 55 2 6 0 0 0 56 1 0 0 0 57 2 6 0 0
                0 58 2 0 0 0 0 59 2 0 0 6 0 60 2 0 0
                0 6 61 0 6 0 62 0 0 0 63 1 6 64 0 65
                1 0 64 0 66 1 0 64 0 67 1 0 64 0 68 1
                6 0 0 69 1 0 0 0 70 0 6 71 72 0 0 71
                73 1 6 0 74 75 2 0 0 74 0 76 1 79 77
                78 80 1 77 82 0 83 2 6 0 0 25 84 2 0
                0 0 25 85 1 86 0 71 87 0 0 86 88 2 6
                0 0 0 89 2 0 0 0 0 90 2 6 0 0 91 92 2
                0 0 0 0 93 2 2 38 0 0 1 1 20 38 0 1 1
                0 0 0 22 2 20 64 0 0 1 2 10 0 0 0 93

```

```

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

```

2.4 The code.o file

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

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

2.5 The info file

```

(((* (($ $ $) (|arguments| (|eq2| . $) (|eq1| . $)) (S (* S S S))
($ (= $ S S)))
(($ $ S) (|arguments| (|l| . S) (|eqn| . $)) (S (* S S S))
($ (= $ S S)))
(($ #0=(|Integer|) $) (|arguments| (|l| . #0#) (|eq| . $))
(S (|coerce| S (|Integer|))) ($ (* $ S S)))
(($ S $) (|arguments| (|l| . S) (|eqn| . $)) (S (* S S S))
($ (= $ S S)))
(+ (($ $ $) (|arguments| (|eq2| . $) (|eq1| . $)) (S (+ S S S))
($ (= $ S S))
(($ $ S) (|arguments| (|s| . S) (|eq1| . $)) ($ (+ $ $ $)))
(($ S $) (|arguments| (|s| . S) (|eq2| . $)) ($ (+ $ $ $))))

```

```

(- (($ $ $) (|arguments| (|eq2| . $) (|eq1| . $)) (S (- S S S))
  ($ (= $ S S)))
  (($ $ S) (|arguments| (|s| . S) (|eq1| . $)) ($ (- $ $ $)))
  (($ $) (|arguments| (|eq1| . $)) (S (- S S))
    ($ (|rhs| S $) (|lhs| S $) (= $ S S)))
  (($ S $) (|arguments| (|s| . S) (|eq2| . $)) ($ (- $ $ $))))
(/ (($ $ $) (|arguments| (|eq2| . $) (|eq1| . $)) (S (/ S S S))
  ($ (= $ S S)))
(= (($ S S) (|arguments| (|r| . S) (|l| . S)))
  (((|Boolean|) $ $) ((|Boolean|) (|false| (|Boolean|)))
    (|locals| (#:G1393 |Boolean|))
    (|arguments| (|eq2| . $) (|eq1| . $)) (S (= (|Boolean|) S S))))
(|One| (($) (S (|One| S)) ($ (|equation| $ S S))))
(|Zero| (($) (S (|Zero| S)) ($ (|equation| $ S S))))
(|characteristic|
  (((|NonNegativeInteger|))
    (S (|characteristic| (|NonNegativeInteger|)))))
(|coerce|
  (((|Boolean|) $) (|arguments| (|eqn| . $))
    (S (= (|Boolean|) S S)))
  (((|OutputForm|) $)
    ((|OutputForm|) (= (|OutputForm|) (|OutputForm|) (|OutputForm|)))
    (|arguments| (|eqn| . $)) (S (|coerce| (|OutputForm|) S))))
(|constructor|
  (NIL (|locals|
    (|Rep| |Join| (|SetCategory|)
      (CATEGORY |domain|
        (SIGNATURE |construct|
          ((|Record| (|:| |lhs| S) (|:| |rhs| S)) S
            S))
        (SIGNATURE |coerce|
          ((|OutputForm|)
            (|Record| (|:| |lhs| S) (|:| |rhs| S))))
        (SIGNATURE |elt|
          (S (|Record| (|:| |lhs| S) (|:| |rhs| S))
            "lhs"))
        (SIGNATURE |elt|
          (S (|Record| (|:| |lhs| S) (|:| |rhs| S))
            "rhs"))
        (SIGNATURE |setelt|
          (S (|Record| (|:| |lhs| S) (|:| |rhs| S))
            "lhs" S))
        (SIGNATURE |setelt|
          (S (|Record| (|:| |lhs| S) (|:| |rhs| S))
            "rhs" S))
        (SIGNATURE |copy|
          ((|Record| (|:| |lhs| S) (|:| |rhs| S))
            (|Record| (|:| |lhs| S) (|:| |rhs| S)))))))
(|differentiate|
  (($ $ #1=(|Symbol|)) (|arguments| (|sym| . #1#) (|eq1| . $))

```

```

(S (|differentiate| S S (|Symbol|))) ($ (|rhs| S $) (|lhs| S $)))
(|dimension|
  ((#2=(|CardinalNumber|))
    (#2# (|coerce| (|CardinalNumber|) (|NonNegativeInteger|))))
(|equation| (($ S S) (|arguments| (|r| . S) (|l| . S))))
(|eval| (($ $ $) (|arguments| (|eqn2| . $) (|eqn1| . $))
  (S (|eval| S S (|Equation| S)) ($ (= $ S S)))
  (($ $ #3=(|List| $))
    (|arguments| (|eqn2| . #3#) (|eqn1| . $))
    (S (|eval| S S (|List| (|Equation| S)) ($ (= $ S S)))
      (($ $ #4=(|List| #5=(|Symbol|)) #6=(|List| S))
        (|arguments| (|lx| . #6#) (|ls| . #4#) (|eqn| . $))
        (S (|eval| S S (|List| (|Symbol|)) (|List| S))
          ($ (= $ S S)))
        (($ $ #5# S) (|arguments| (|x| . S) (|s| . #5#) (|eqn| . $))
          (S (|eval| S S (|Symbol| S)) ($ (= $ S S))))
(|factorAndSplit|
  (((|List| $) $)
    ((|MultivariateFactorize| (|Symbol|)
      (|IndexedExponents| (|Symbol|)) (|Integer|)
      (|Polynomial| (|Integer|)))
      (|factor| (|Factored| (|Polynomial| (|Integer|))
        (|Polynomial| (|Integer|))))
    (|Factored| S)
    (|factors|
      (|List| (|Record| (|:| |factor| S)
        (|:| |exponent| (|Integer|))))
      (|Factored| S)))
    (|Factored| (|Polynomial| (|Integer|))
      (|factors|
        (|List| (|Record| (|:| |factor| (|Polynomial| (|Integer|))
          (|:| |exponent| (|Integer|))))
          (|Factored| (|Polynomial| (|Integer|))))
        (|locals| (|p| |Polynomial| (|Integer|)) (|eq0| . $))
        (|arguments| (|eq| . $))
        (S (|factor| (|Factored| S) S) (|Zero| S))
        ($ (|rightZero| $ $) (|lhs| S $) (|equation| $ S S)))
(|inv| (($ $) (|arguments| (|eq| . $)) (S (|inv| S S))
  ($ (|rhs| S $) (|lhs| S $)))
(|leftOne|
  (((|Union| $ "failed") $) (|locals| (|re| |Union| S "failed"))
    (|arguments| (|eq| . $))
    (S (|recip| (|Union| S "failed") S) (|inv| S S) (|One| S)
      (* S S S))
    ($ (|rhs| S $) (|lhs| S $) (|One| $) (= $ S S)))
(|leftZero|
  (($ $) (|arguments| (|eq| . $)) (S (|Zero| S) (- S S S))
    ($ (|rhs| S $) (|lhs| S $) (|Zero| $) (= $ S S S)))
(|lhs| ((S $) (|arguments| (|eqn| . $))))
(|map| (($ #7=(|Mapping| S S) $)

```

```

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

```

2.6 The EQ.fn file

```

(in-package 'compiler)(init-fn)
(ADD-FN-DATA '(
#S(FN NAME BOOT::|EQ;*;S2$;26| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
  CALLEES
  (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
    BOOT:SPADCALL)
  RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
  (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;rightOne;$U;32| DEF DEFUN VALUE-TYPE T FUN-VALUES
  NIL CALLEES
  (BOOT::|spadConstant| VMLISP:QCDR CONS VMLISP:QCAR EQL
    BOOT::|EQQCAR COND VMLISP:EXIT CDR CAR SVREF VMLISP:QREFELT
    BOOT:SPADCALL BOOT::|LETT VMLISP:SEQ RETURN)
  RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
  (BOOT::|spadConstant| VMLISP:QCDR VMLISP:QCAR BOOT::|EQQCAR COND
    VMLISP:EXIT VMLISP:QREFELT BOOT:SPADCALL BOOT::|LETT
    VMLISP:SEQ RETURN))
#S(FN NAME BOOT::|EQ;lhs;$S;4| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
  CALLEES (CAR VMLISP:QCAR) RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT
  NIL MACROS (VMLISP:QCAR))
#S(FN NAME BOOT::|EQ;+;3$;15| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
  CALLEES
  (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT

```

```

        BOOT:SPADCALL)
    RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
    (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;dimension;Cn;40| DEF DEFUN VALUE-TYPE T FUN-VALUES
    NIL CALLEES (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
    RETURN-TYPE NIL ARG-TYPES (T) NO-EMIT NIL MACROS
    (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;rightZero;2$;22| DEF DEFUN VALUE-TYPE T FUN-VALUES
    NIL CALLEES
    (BOOT::|spadConstant| CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
    RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
    (BOOT::|spadConstant| VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;coerce;$Of;13| DEF DEFUN VALUE-TYPE T FUN-VALUES
    NIL CALLEES
    (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
     BOOT:SPADCALL)
    RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
    (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;One;$;29| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES
    (CDR BOOT::|spadConstant| CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
    RETURN-TYPE NIL ARG-TYPES (T) NO-EMIT NIL MACROS
    (BOOT::|spadConstant| VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;inv;2$;42| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL CONS)
    RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
    (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;-;$S$;20| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES (CDR CONS CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
    RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
    (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;=;2$B;12| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES
    (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
     BOOT:SPADCALL COND)
    RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
    (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL COND))
#S(FN NAME BOOT::|EQ;/;$3$;41| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES
    (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
     BOOT:SPADCALL)
    RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
    (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;recip;$U;30| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES
    (VMLISP:QCDR LIST* CONS VMLISP:QCAR EQL BOOT::|EQQCAR COND
     VMLISP:EXIT CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL
     BOOT::|LETT VMLISP:SEQ RETURN)
    RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
    (VMLISP:QCDR VMLISP:QCAR BOOT::|EQQCAR COND VMLISP:EXIT

```

```

        VMLISP:QREFELT BOOT:SPADCALL BOOT::LETT VMLISP:SEQ RETURN))
#S(FN NAME BOOT::|EQ;-;3$;24| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES
    (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
     BOOT:SPADCALL)
    RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
    (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;eval;$L$;11| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES
    (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
     BOOT:SPADCALL)
    RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
    (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;leftZero;2$;21| DEF DEFUN VALUE-TYPE T FUN-VALUES
    NIL CALLEES
    (CDR BOOT::|spadConstant| CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
    RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
    (BOOT::|spadConstant| VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;*;S2$;27| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES
    (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
     BOOT:SPADCALL)
    RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
    (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;*;I2$;37| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL) RETURN-TYPE
    NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
    (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;eval;3$;10| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES
    (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
     BOOT:SPADCALL)
    RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
    (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;eval;$SS$;8| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES
    (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
     BOOT:SPADCALL)
    RETURN-TYPE NIL ARG-TYPES (T T T T) NO-EMIT NIL MACROS
    (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;factorAndSplit;$L;38| DEF DEFUN VALUE-TYPE T
    FUN-VALUES NIL CALLEES
    (BOOT:|Integer| BOOT:|Polynomial| EQUAL BOOT:NREVERSEO
     BOOT::|spadConstant| VMLISP:QCAR CONS ATOM VMLISP:EXIT CDR
     CAR BOOT:SPADCALL BOOT::LETT BOOT::|devaluate| LIST SVREF
     VMLISP:QREFELT BOOT::|HasSignature| COND VMLISP:SEQ RETURN)
    RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
    (BOOT::|spadConstant| VMLISP:QCAR VMLISP:EXIT BOOT:SPADCALL
     BOOT::LETT VMLISP:QREFELT COND VMLISP:SEQ RETURN))
#S(FN NAME BOOT::|EQ;differentiate;$S$;39| DEF DEFUN VALUE-TYPE T

```



```

FUN-VALUES NIL CALLEES
(CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL CONS) RETURN-TYPE NIL
ARG-TYPES (T T T) NO-EMIT NIL MACROS
(VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;eval;$LL$;9| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
  CALLEES
  (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
   BOOT:SPADCALL)
  RETURN-TYPE NIL ARG-TYPES (T T T T) NO-EMIT NIL MACROS
  (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;leftOne;$U;34| DEF DEFUN VALUE-TYPE T FUN-VALUES
  NIL CALLEES
  (CDR BOOT::|spadConstant| CAR SVREF VMLISP:QREFELT BOOT:SPADCALL
   CONS)
  RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
  (BOOT::|spadConstant| VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;map;M2$;7| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
  CALLEES
  (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
   BOOT:SPADCALL)
  RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
  (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;-;S2$;19| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
  CALLEES (CDR CONS CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
  RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
  (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;equation;2S$;3| DEF DEFUN VALUE-TYPE T FUN-VALUES
  NIL CALLEES (CONS) RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL
  MACROS NIL)
#S(FN NAME BOOT::|EQ;+;$S$;17| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
  CALLEES (CDR CONS CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
  RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
  (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;factorAndSplit;$L;1| DEF DEFUN VALUE-TYPE T
  FUN-VALUES NIL CALLEES
  (BOOT:NREVERSEO BOOT::|spadConstant| VMLISP:QCAR CONS ATOM
   VMLISP:EXIT CDR CAR BOOT:SPADCALL BOOT::LETT
   BOOT::|devaluate| LIST SVREF VMLISP:QREFELT
   BOOT::|HasSignature| COND VMLISP:SEQ RETURN)
  RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
  (BOOT::|spadConstant| VMLISP:QCAR VMLISP:EXIT BOOT:SPADCALL
   BOOT::LETT VMLISP:QREFELT COND VMLISP:SEQ RETURN))
#S(FN NAME BOOT::|EQ;*;3$;25| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
  CALLEES
  (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
   BOOT:SPADCALL)
  RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
  (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;Zero;$;23| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
  CALLEES

```

```

(CDR BOOT::|spadConstant| CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
RETURN-TYPE NIL ARG-TYPES (T) NO-EMIT NIL MACROS
(BOOT::|spadConstant| VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;characteristic;Nni;36| DEF DEFUN VALUE-TYPE T
FUN-VALUES NIL CALLEES
(CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL) RETURN-TYPE NIL
ARG-TYPES (T) NO-EMIT NIL MACROS (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;leftOne;$U;31| DEF DEFUN VALUE-TYPE T FUN-VALUES
NIL CALLEES
(VMLISP:QCDR BOOT::|spadConstant| CONS VMLISP:QCAR EQL
BOOT::|EQQCAR COND VMLISP:EXIT CDR CAR SVREF VMLISP:QREFELT
BOOT:SPADCALL BOOT::LETT VMLISP:SEQ RETURN)
RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
(VMLISP:QCDR BOOT::|spadConstant| VMLISP:QCAR BOOT::|EQQCAR COND
VMLISP:EXIT VMLISP:QREFELT BOOT:SPADCALL BOOT::LETT
VMLISP:SEQ RETURN))
#S(FN NAME BOOT::|EQ;swap;2$;6| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
CALLEES (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL CONS)
RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
(VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;-;2$;18| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
CALLEES (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL) RETURN-TYPE
NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
(VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;subst;3$;43| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
CALLEES
(CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL CONS VMLISP:EXIT
BOOT::LETT VMLISP:SEQ RETURN)
RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
(VMLISP:QREFELT BOOT:SPADCALL VMLISP:EXIT BOOT::LETT VMLISP:SEQ
RETURN))
#S(FN NAME BOOT::|EQ;=;2S$;2| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
CALLEES (CONS) RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL
MACROS NIL)
#S(FN NAME BOOT::|EQ;*;$S$;28| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
CALLEES
(VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
BOOT:SPADCALL)
RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
(VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;+;S2$;16| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
CALLEES (CDR CONS CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
(VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|Equation;| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
CALLEES
(BOOT::|EQ;One;$;29| BOOT::|EQ;Zero;$;23|
BOOT::|dispatchFunction| BOOT::|testBitVector| COND
BOOT::|Record0| BOOT::|Record| BOOT::|stuffDomainSlots| CONS
BOOT::|haddProp| BOOT::|HasCategory| BOOT::|buildPredVector|

```

```

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

```

2.7 The index.kaf file

Each constructor (e.g. EQ) had one library directory (e.g. EQ.nrlib). This directory contained a random access file called the index.kaf file. These files contain runtime information such as the operationAlist and the ConstructorModemap. At system build time we merge all of these .nrlib/index.kaf files into one database, INTERP.daase. Requests to get information from this database are cached so that multiple references do not cause additional disk i/o.

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

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

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

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

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

So to read the “operationAlist” from this file you would:

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

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

The index offset byte

35695

The “loadTimeStuff”

```
(setf (get 'Equation| 'infovec|)
  (LIST '#(NIL NIL NIL NIL NIL NIL (|local| |#1|) 'Rep|
    (0 . |rightZero|) |EQ;lhs;$S;4| (|Factored| $)
    (5 . |factor|)
    (|Record| (|:| |factor| 6) (|:| |exponent| 74))
    (|List| 12) (|Factored| 6) (10 . |factors|) (15 . |Zero|)
    |EQ;equation;2S$;3| (|List| $) (19 . |factorAndSplit|)
    |EQ;=;2S$;2| |EQ;rhs;$S;5| |EQ;swap;2$;6| (|Mapping| 6 6)
    |EQ;map;M2$;7| (|Symbol|) (24 . |eval|) (31 . |eval|)
    (|List| 25) (|List| 6) (38 . |eval|) (45 . |eval|)
    (|Equation| 6) (52 . |eval|) (58 . |eval|) (|List| 32)
    (64 . |eval|) (70 . |eval|) (|Boolean|) (76 . =) (82 . =)
    (|OutputForm|) (88 . |coerce|) (93 . =) (99 . |coerce|)
    (104 . |coerce|) (109 . +) (115 . +) (121 . +) (127 . +)
    (133 . -) (138 . -) (143 . -) (149 . -) (155 . -)
    (161 . |Zero|) (165 . -) (171 . |leftZero|) (176 . *)
    (182 . *) (188 . *) (194 . *) (200 . |One|) (204 . |One|)
    (|Union| $ 'failed") (208 . |recip|) (213 . |recip|)
    (218 . |leftOne|) (223 . |rightOne|) (228 . |inv|)
    (233 . |inv|) (|NonNegativeInteger|)
    (238 . |characteristic|) (242 . |characteristic|)
    (|Integer|) (246 . |coerce|) (251 . *) (|Factored| 78)
    (|Polynomial| 74)
    (|MultivariateFactorize| 25 (|IndexedExponents| 25) 74 78)
    (257 . |factor|)
    (|Record| (|:| |factor| 78) (|:| |exponent| 74))
    (|List| 81) (262 . |factors|) (267 . |differentiate|)
    (273 . |differentiate|) (|CardinalNumber|)
    (279 . |coerce|) (284 . |dimension|) (288 . /) (294 . /)
    (|Equation| $) (300 . |subst|) (306 . |subst|)
    (|PositiveInteger|) (|List| 71) (|SingleInteger|)
    (|String|))
  '#(~= 312 |zero?| 318 |swap| 323 |subtractIfCan| 328 |subst|
    334 |sample| 340 |rightZero| 344 |rightOne| 349 |rhs| 354
    |recip| 359 |one?| 364 |map| 369 |lhs| 375 |leftZero| 380
    |leftOne| 385 |latex| 390 |inv| 395 |hash| 400
    |factorAndSplit| 405 |eval| 410 |equation| 436 |dimension|
    442 |differentiate| 446 |conjugate| 472 |commutator| 478
    |coerce| 484 |characteristic| 499 ^ 503 |Zero| 521 |One|
    525 D 529 = 555 / 567 - 579 + 602 ** 620 * 638)
  '((|unitsKnown| . 12) (|rightUnitary| . 3)
    (|leftUnitary| . 3))
  (CONS (|makeByteWordVec2| 25
```

```

'(1 15 4 14 5 14 3 5 3 21 21 6 21 17 24 19 25 0 2
  25 2 7))
(CONS '#(|VectorSpace&| |Module&|
         |PartialDifferentialRing&| NIL |Ring&| NIL NIL
         NIL NIL |AbelianGroup&| NIL |Group&|
         |AbelianMonoid&| |Monoid&| |AbelianSemiGroup&|
         |SemiGroup&| |SetCategory&| NIL NIL
         |BasicType&| NIL |InnerEvalable&|)
      (CONS '#(|VectorSpace| 6) (|Module| 6)
              (|PartialDifferentialRing| 25)
              (|BiModule| 6 6) (|Ring|)
              (|LeftModule| 6) (|RightModule| 6)
              (|Rng|) (|LeftModule| $$)
              (|AbelianGroup|)
              (|CancellationAbelianMonoid|) (|Group|)
              (|AbelianMonoid|) (|Monoid|)
              (|AbelianSemiGroup|) (|SemiGroup|)
              (|SetCategory|) (|Type|)
              (|CoercibleTo| 41) (|BasicType|)
              (|CoercibleTo| 38)
              (|InnerEvalable| 25 6))
      (|makeByteWordVec2| 97
        '(1 0 0 0 8 1 6 10 0 11 1 14 13 0 15 0
          6 0 16 1 0 18 0 19 3 6 0 0 25 6 26 3
          0 0 0 25 6 27 3 6 0 0 28 29 30 3 0 0
          0 28 29 31 2 6 0 0 32 33 2 0 0 0 0 34
          2 6 0 0 35 36 2 0 0 0 18 37 2 6 38 0
          0 39 2 0 38 0 0 40 1 6 41 0 42 2 41 0
          0 0 43 1 0 41 0 44 1 0 38 0 45 2 6 0
          0 0 46 2 0 0 0 0 47 2 0 0 6 0 48 2 0
          0 0 6 49 1 6 0 0 50 1 0 0 0 51 2 0 0
          0 0 52 2 0 0 6 0 53 2 0 0 0 6 54 0 0
          0 55 2 6 0 0 0 56 1 0 0 0 57 2 6 0 0
          0 58 2 0 0 0 0 59 2 0 0 6 0 60 2 0 0
          0 6 61 0 6 0 62 0 0 0 63 1 6 64 0 65
          1 0 64 0 66 1 0 64 0 67 1 0 64 0 68 1
          6 0 0 69 1 0 0 0 70 0 6 71 72 0 0 71
          73 1 6 0 74 75 2 0 0 74 0 76 1 79 77
          78 80 1 77 82 0 83 2 6 0 0 25 84 2 0
          0 0 25 85 1 86 0 71 87 0 0 86 88 2 6
          0 0 0 89 2 0 0 0 0 90 2 6 0 0 91 92 2
          0 0 0 0 93 2 2 38 0 0 1 1 20 38 0 1 1
          0 0 0 22 2 20 64 0 0 1 2 10 0 0 0 93
          0 22 0 1 1 20 0 0 8 1 16 64 0 68 1 0
          6 0 21 1 16 64 0 66 1 16 38 0 1 2 0 0
          23 0 24 1 0 6 0 9 1 20 0 0 57 1 16 64
          0 67 1 2 97 0 1 1 11 0 0 70 1 2 96 0
          1 1 9 18 0 19 2 8 0 0 0 34 2 8 0 0 18
          37 3 7 0 0 25 6 27 3 7 0 0 28 29 31 2
          0 0 6 6 17 0 1 86 88 2 4 0 0 28 1 2 4

```

```

0 0 25 85 3 4 0 0 28 95 1 3 4 0 0 25
71 1 2 6 0 0 0 1 2 6 0 0 0 1 1 3 0 74
1 1 2 41 0 44 1 2 38 0 45 0 3 71 73 2
6 0 0 74 1 2 16 0 0 71 1 2 18 0 0 94
1 0 20 0 55 0 16 0 63 2 4 0 0 28 1 2
4 0 0 25 1 3 4 0 0 28 95 1 3 4 0 0 25
71 1 2 2 38 0 0 40 2 0 0 6 6 20 2 11
0 0 0 90 2 1 0 0 6 1 1 20 0 0 51 2 20
0 0 0 52 2 20 0 6 0 53 2 20 0 0 6 54
2 23 0 0 0 47 2 23 0 6 0 48 2 23 0 0
6 49 2 6 0 0 74 1 2 16 0 0 71 1 2 18
0 0 94 1 2 20 0 71 0 1 2 20 0 74 0 76
2 23 0 94 0 1 2 18 0 0 0 59 2 18 0 0
6 61 2 18 0 6 0 60))))))
'lookupComplete))

```

The “compilerInfo”

```

(SETQ |$CategoryFrame|
  (|put| 'Equation| 'isFunctor|
    '(((|eval| ($ $ (|List| (|Symbol|)) (|List| |#1|)))
      (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|))
      (ELT $ 31))
    (|eval| ($ $ (|Symbol|) |#1|))
    (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|))
    (ELT $ 27))
    (~= ((|Boolean|) $ $)) (|has| |#1| (|SetCategory|))
    (ELT $ NIL))
    (= ((|Boolean|) $ $)) (|has| |#1| (|SetCategory|))
    (ELT $ 40))
    (|coerce| ((|OutputForm|) $))
    (|has| |#1| (|SetCategory|)) (ELT $ 44))
    (|hash| ((|SingleInteger|) $))
    (|has| |#1| (|SetCategory|)) (ELT $ NIL))
    (|latex| ((|String|) $)) (|has| |#1| (|SetCategory|))
    (ELT $ NIL))
    (|coerce| ((|Boolean|) $)) (|has| |#1| (|SetCategory|))
    (ELT $ 45))
    (+ ($ $ $)) (|has| |#1| (|AbelianSemiGroup|))
    (ELT $ 47))
    (* ($ (|PositiveInteger|) $))
    (|has| |#1| (|AbelianSemiGroup|)) (ELT $ NIL))
    (|Zero| ($)) (|has| |#1| (|AbelianGroup|))
    (CONST $ 55))
    (|sample| ($))
    (OR (|has| |#1| (|AbelianGroup|))
      (|has| |#1| (|Monoid|)))
    (CONST $ NIL))
    (|zero?| ((|Boolean|) $)) (|has| |#1| (|AbelianGroup|))

```

```

(ELT $ NIL))
(* ($ (|NonNegativeInteger|) $))
(|has| |#1| (|AbelianGroup|)) (ELT $ NIL))
(|subtractIfCan| ((|Union| $ "failed") $ $))
(|has| |#1| (|AbelianGroup|)) (ELT $ NIL))
((- ($ $)) (|has| |#1| (|AbelianGroup|)) (ELT $ 51))
((- ($ $ $)) (|has| |#1| (|AbelianGroup|)) (ELT $ 52))
(* ($ (|Integer|) $)) (|has| |#1| (|AbelianGroup|))
(ELT $ 76))
(* ($ $ $)) (|has| |#1| (|SemiGroup|)) (ELT $ 59))
(** ($ $ (|PositiveInteger|)))
(|has| |#1| (|SemiGroup|)) (ELT $ NIL))
(^ ($ $ (|PositiveInteger|)))
(|has| |#1| (|SemiGroup|)) (ELT $ NIL))
(|One| ($)) (|has| |#1| (|Monoid|)) (CONST $ 63))
(|one?| ((|Boolean|) $)) (|has| |#1| (|Monoid|))
(ELT $ NIL))
(** ($ $ (|NonNegativeInteger|)))
(|has| |#1| (|Monoid|)) (ELT $ NIL))
(^ ($ $ (|NonNegativeInteger|)))
(|has| |#1| (|Monoid|)) (ELT $ NIL))
(|recip| ((|Union| $ "failed") $))
(|has| |#1| (|Monoid|)) (ELT $ 66))
(|inv| ($ $))
(OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|)))
(ELT $ 70))
(/ ($ $ $))
(OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|)))
(ELT $ 90))
(** ($ $ (|Integer|))) (|has| |#1| (|Group|))
(ELT $ NIL))
(^ ($ $ (|Integer|))) (|has| |#1| (|Group|))
(ELT $ NIL))
(|conjugate| ($ $ $)) (|has| |#1| (|Group|))
(ELT $ NIL))
(|commutator| ($ $ $)) (|has| |#1| (|Group|))
(ELT $ NIL))
(|characteristic| ((|NonNegativeInteger|)))
(|has| |#1| (|Ring|)) (ELT $ 73))
(|coerce| ($ (|Integer|))) (|has| |#1| (|Ring|))
(ELT $ NIL))
(* ($ |#1| $)) (|has| |#1| (|SemiGroup|)) (ELT $ 60))
(* ($ $ |#1|)) (|has| |#1| (|SemiGroup|)) (ELT $ 61))
(|differentiate| ($ $ (|Symbol|)))
(|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
(ELT $ 85))
(|differentiate| ($ $ (|List| (|Symbol|))))
(|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
(ELT $ NIL))
(|differentiate|

```



```

($ $ (|Symbol|) (|NonNegativeInteger|))
(|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
(ELT $ NIL))
((|differentiate|
  ($ $ (|List| (|Symbol|))
    (|List| (|NonNegativeInteger|))))
  (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
  (ELT $ NIL))
((D ($ $ (|Symbol|))
  (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
  (ELT $ NIL))
  (D ($ $ (|List| (|Symbol|)))
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
    (ELT $ NIL))
  (D ($ $ (|Symbol|) (|NonNegativeInteger|))
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
    (ELT $ NIL))
  (D ($ $ (|List| (|Symbol|))
    (|List| (|NonNegativeInteger|)))
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
    (ELT $ NIL))
  (/( $ $ |#1|)) (|has| |#1| (|Field|)) (ELT $ NIL))
  (|dimension| ((|CardinalNumber|))
    (|has| |#1| (|Field|)) (ELT $ 88))
  (|subst| ($ $ $)) (|has| |#1| (|ExpressionSpace|))
  (ELT $ 93))
  (|factorAndSplit| ((|List| $) $))
  (|has| |#1| (|IntegralDomain|)) (ELT $ 19))
  (|rightOne| ((|Union| $ "failed") $))
  (|has| |#1| (|Monoid|)) (ELT $ 68))
  (|leftOne| ((|Union| $ "failed") $))
  (|has| |#1| (|Monoid|)) (ELT $ 67))
  (- ($ $ |#1|)) (|has| |#1| (|AbelianGroup|))
  (ELT $ 54))
  (- ($ |#1| $)) (|has| |#1| (|AbelianGroup|))
  (ELT $ 53))
  (|rightZero| ($ $)) (|has| |#1| (|AbelianGroup|))
  (ELT $ 8))
  (|leftZero| ($ $)) (|has| |#1| (|AbelianGroup|))
  (ELT $ 57))
  (+ ($ $ |#1|)) (|has| |#1| (|AbelianSemiGroup|))
  (ELT $ 49))
  (+ ($ |#1| $)) (|has| |#1| (|AbelianSemiGroup|))
  (ELT $ 48))
  (|eval| ($ $ (|List| $)))
  (AND (|has| |#1| (|Evalable| |#1|))
    (|has| |#1| (|SetCategory|)))
  (ELT $ 37))
  (|eval| ($ $ $))
  (AND (|has| |#1| (|Evalable| |#1|))

```

```

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

```

```

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

```

```

T '|Equation|
(|put| '|Equation| '|mode|
  '|Mapping|
    (|Join| (|Type|)
      (CATEGORY |domain|
        (SIGNATURE = ($ |#1| |#1|))
        (SIGNATURE |equation|
          ($ |#1| |#1|))
        (SIGNATURE |swap| ($ $))
        (SIGNATURE |lhs| (|#1| $))
        (SIGNATURE |rhs| (|#1| $))
        (SIGNATURE |map|
          ($ (|Mapping| |#1| |#1|) $))
      (IF
        (|has| |#1|
          (|InnerEvalable| (|Symbol|)
            |#1|))
        (ATTRIBUTE
          (|InnerEvalable| (|Symbol|)
            |#1|))
        |noBranch|)
      (IF (|has| |#1| (|SetCategory|))
        (PROGN
          (ATTRIBUTE (|SetCategory|))
          (ATTRIBUTE
            (|CoercibleTo| (|Boolean|)))
          (IF
            (|has| |#1|
              (|Evalable| |#1|))
            (PROGN
              (SIGNATURE |eval| ($ $ $))
              (SIGNATURE |eval|
                ($ $ (|List| $))))
            |noBranch|))
        |noBranch|)
      (IF
        (|has| |#1|
          (|AbelianSemiGroup|))
        (PROGN
          (ATTRIBUTE
            (|AbelianSemiGroup|))
          (SIGNATURE + ($ |#1| $))
          (SIGNATURE + ($ $ |#1|)))
        |noBranch|)
      (IF (|has| |#1| (|AbelianGroup|))
        (PROGN
          (ATTRIBUTE (|AbelianGroup|))
          (SIGNATURE |leftZero| ($ $))
          (SIGNATURE |rightZero| ($ $))
          (SIGNATURE - ($ |#1| $))

```

```

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

```

```

(|VectorSpace| |#1|))
(SIGNATURE / ($ $ $))
(SIGNATURE |inv| ($ $))
|noBranch|)
(IF
(|has| |#1| (|ExpressionSpace|))
(SIGNATURE |subst| ($ $ $))
|noBranch|)))
(|Type|))
|$CategoryFrame|))))

```

The “constructorForm”

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

The “constructorKind”

```
|domain|
```

The “constructorModemap”

```

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

```

```

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

(|Type|)
(T |Equation|)

```

The “constructorCategory”

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



```

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

```

The “sourceFile”

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

The “modemaps”

```

((= (*1 *1 *2 *2)
  (AND (|isDomain| *1 (|Equation| *2)) (|ofCategory| *2 (|Type|))))
  (|equation| (*1 *1 *2 *2)
    (AND (|isDomain| *1 (|Equation| *2)) (|ofCategory| *2 (|Type|))))
  (|swap| (*1 *1 *1)
    (AND (|isDomain| *1 (|Equation| *2))
      (|ofCategory| *2 (|Type|))))
  (|lhs| (*1 *2 *1)
    (AND (|isDomain| *1 (|Equation| *2))
      (|ofCategory| *2 (|Type|))))
  (|rhs| (*1 *2 *1)
    (AND (|isDomain| *1 (|Equation| *2))
      (|ofCategory| *2 (|Type|))))
  (|map| (*1 *1 *2 *1)
    (AND (|isDomain| *2 (|Mapping| *3 *3))
      (|ofCategory| *3 (|Type|))
      (|isDomain| *1 (|Equation| *3))))
  (|eval| (*1 *1 *1 *1)

```

```

(AND (|ofCategory| *2 (|Evalable| *2))
      (|ofCategory| *2 (|SetCategory|))
      (|ofCategory| *2 (|Type|))
      (|isDomain| *1 (|Equation| *2)))
(|eval| (*1 *1 *1 *2)
        (AND (|isDomain| *2 (|List| (|Equation| *3)))
              (|ofCategory| *3 (|Evalable| *3))
              (|ofCategory| *3 (|SetCategory|))
              (|ofCategory| *3 (|Type|))
              (|isDomain| *1 (|Equation| *3))))
(+ (*1 *1 *2 *1)
   (AND (|isDomain| *1 (|Equation| *2))
         (|ofCategory| *2 (|AbelianSemiGroup|))
         (|ofCategory| *2 (|Type|))))
(+ (*1 *1 *1 *2)
   (AND (|isDomain| *1 (|Equation| *2))
         (|ofCategory| *2 (|AbelianSemiGroup|))
         (|ofCategory| *2 (|Type|))))
(|leftZero| (*1 *1 *1)
            (AND (|isDomain| *1 (|Equation| *2))
                  (|ofCategory| *2 (|AbelianGroup|))
                  (|ofCategory| *2 (|Type|))))
(|rightZero| (*1 *1 *1)
             (AND (|isDomain| *1 (|Equation| *2))
                   (|ofCategory| *2 (|AbelianGroup|))
                   (|ofCategory| *2 (|Type|))))
(- (*1 *1 *2 *1)
   (AND (|isDomain| *1 (|Equation| *2))
         (|ofCategory| *2 (|AbelianGroup|)) (|ofCategory| *2 (|Type|))))
(- (*1 *1 *1 *2)
   (AND (|isDomain| *1 (|Equation| *2))
         (|ofCategory| *2 (|AbelianGroup|)) (|ofCategory| *2 (|Type|))))
(|leftOne| (*1 *1 *1)
           (|partial| AND (|isDomain| *1 (|Equation| *2))
                          (|ofCategory| *2 (|Monoid|)) (|ofCategory| *2 (|Type|))))
(|rightOne| (*1 *1 *1)
            (|partial| AND (|isDomain| *1 (|Equation| *2))
                          (|ofCategory| *2 (|Monoid|)) (|ofCategory| *2 (|Type|))))
(|factorAndSplit| (*1 *2 *1)
                  (AND (|isDomain| *2 (|List| (|Equation| *3)))
                        (|isDomain| *1 (|Equation| *3))
                        (|ofCategory| *3 (|IntegralDomain|))
                        (|ofCategory| *3 (|Type|))))
(|subst| (*1 *1 *1 *1)
         (AND (|isDomain| *1 (|Equation| *2))
               (|ofCategory| *2 (|ExpressionSpace|))
               (|ofCategory| *2 (|Type|))))
(* (*1 *1 *1 *2)
   (AND (|isDomain| *1 (|Equation| *2))
         (|ofCategory| *2 (|SemiGroup|)) (|ofCategory| *2 (|Type|))))

```

```

(* (*1 *1 *2 *1)
  (AND (|isDomain| *1 (|Equation| *2))
        (|ofCategory| *2 (|SemiGroup|)) (|ofCategory| *2 (|Type|))))
(/ (*1 *1 *1 *1)
  (OR (AND (|isDomain| *1 (|Equation| *2))
            (|ofCategory| *2 (|Field|)) (|ofCategory| *2 (|Type|)))
      (AND (|isDomain| *1 (|Equation| *2))
            (|ofCategory| *2 (|Group|)) (|ofCategory| *2 (|Type|)))))
(|inv| (*1 *1 *1)
  (OR (AND (|isDomain| *1 (|Equation| *2))
            (|ofCategory| *2 (|Field|))
            (|ofCategory| *2 (|Type|)))
      (AND (|isDomain| *1 (|Equation| *2))
            (|ofCategory| *2 (|Group|))
            (|ofCategory| *2 (|Type|)))))

```

The “operationAlist”

```

((~= (((|Boolean|) $ $) NIL (|has| |#1| (|SetCategory|))))
(|zero?| (((|Boolean|) $) NIL (|has| |#1| (|AbelianGroup|))))
(|swap| (($ $) 22))
(|subtractIfCan|
  (((|Union| $ "failed") $ $) NIL (|has| |#1| (|AbelianGroup|))))
(|subst| (($ $ $) 93 (|has| |#1| (|ExpressionSpace|))))
(|sample|
  (($) NIL
    (OR (|has| |#1| (|AbelianGroup|)) (|has| |#1| (|Monoid|))) CONST))
(|rightZero| (($ $) 8 (|has| |#1| (|AbelianGroup|))))
(|rightOne| (((|Union| $ "failed") $) 68 (|has| |#1| (|Monoid|))))
(|rhs| ((|#1| $) 21))
(|recip| (((|Union| $ "failed") $) 66 (|has| |#1| (|Monoid|))))
(|one?| (((|Boolean|) $) NIL (|has| |#1| (|Monoid|))))
(|map| (($ (|Mapping| |#1| |#1|) $) 24) (|lhs| ((|#1| $) 9))
(|leftZero| (($ $) 57 (|has| |#1| (|AbelianGroup|))))
(|leftOne| (((|Union| $ "failed") $) 67 (|has| |#1| (|Monoid|))))
(|latex| (((|String|) $) NIL (|has| |#1| (|SetCategory|))))
(|inv| (($ $) 70 (OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|)))))
(|hash| (((|SingleInteger|) $) NIL (|has| |#1| (|SetCategory|))))
(|factorAndSplit| (((|List| $) $) 19 (|has| |#1| (|IntegralDomain|))))
(|eval| (($ $ $) 34
  (AND (|has| |#1| (|Evalable| |#1|))
        (|has| |#1| (|SetCategory|))))
  (($ $ (|List| $)) 37
  (AND (|has| |#1| (|Evalable| |#1|))
        (|has| |#1| (|SetCategory|))))
  (($ $ (|Symbol|) |#1|) 27
  (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|)))
  (($ $ (|List| (|Symbol|)) (|List| |#1|)) 31
  (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|))))

```

```

(equation| (($ |#1| |#1|) 17))
(dimension| (((|CardinalNumber|) 88 (|has| |#1| (|Field|))))
(differentiate|
  (($ $ (|List| (|Symbol|)) (|List| (|NonNegativeInteger|))) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
  (($ $ (|Symbol|) (|NonNegativeInteger|)) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
  (($ $ (|List| (|Symbol|))) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
  (($ $ (|Symbol|)) 85
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
(conjugate| (($ $ $) NIL (|has| |#1| (|Group|)))
(commutator| (($ $ $) NIL (|has| |#1| (|Group|)))
(coerce| (($ (|Integer|)) NIL (|has| |#1| (|Ring|)))
  (((|Boolean|) $) 45 (|has| |#1| (|SetCategory|)))
  (((|OutputForm|) $) 44 (|has| |#1| (|SetCategory|)))
(characteristic| (((|NonNegativeInteger|) 73 (|has| |#1| (|Ring|)))
  ^ (($ $ (|Integer|)) NIL (|has| |#1| (|Group|)))
    (($ $ (|NonNegativeInteger|)) NIL (|has| |#1| (|Monoid|)))
    (($ $ (|PositiveInteger|)) NIL (|has| |#1| (|SemiGroup|)))
(|Zero| (($) 55 (|has| |#1| (|AbelianGroup|)) CONST))
(|One| (($) 63 (|has| |#1| (|Monoid|)) CONST))
(D (($ $ (|List| (|Symbol|)) (|List| (|NonNegativeInteger|))) NIL
  (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
  (($ $ (|Symbol|) (|NonNegativeInteger|)) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
  (($ $ (|List| (|Symbol|))) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
  (($ $ (|Symbol|)) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
(= (($ |#1| |#1|) 20)
  (((|Boolean|) $ $) 40 (|has| |#1| (|SetCategory|)))
(/ (($ $ |#1|) NIL (|has| |#1| (|Field|)))
  (($ $ $) 90 (OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|))))
(- (($ |#1| $) 53 (|has| |#1| (|AbelianGroup|)))
  (($ $ |#1|) 54 (|has| |#1| (|AbelianGroup|)))
  (($ $ $) 52 (|has| |#1| (|AbelianGroup|)))
  (($ $) 51 (|has| |#1| (|AbelianGroup|)))
(+ (($ |#1| $) 48 (|has| |#1| (|AbelianSemiGroup|)))
  (($ $ |#1|) 49 (|has| |#1| (|AbelianSemiGroup|)))
  (($ $ $) 47 (|has| |#1| (|AbelianSemiGroup|)))
(** (($ $ (|Integer|)) NIL (|has| |#1| (|Group|)))
  (($ $ (|NonNegativeInteger|)) NIL (|has| |#1| (|Monoid|)))
  (($ $ (|PositiveInteger|)) NIL (|has| |#1| (|SemiGroup|)))
(* (($ $ |#1|) 61 (|has| |#1| (|SemiGroup|)))
  (($ |#1| $) 60 (|has| |#1| (|SemiGroup|)))
  (($ $ $) 59 (|has| |#1| (|SemiGroup|)))
  (($ (|Integer|) $) 76 (|has| |#1| (|AbelianGroup|)))
  (($ (|NonNegativeInteger|) $) NIL (|has| |#1| (|AbelianGroup|)))
  (($ (|PositiveInteger|) $) NIL (|has| |#1| (|AbelianSemiGroup|))))

```

The “superDomain”

The “signaturesAndLocals”

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

The “attributes”

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

The “predicates”

```
((|HasCategory| |#1| '(|Field|)) (|HasCategory| |#1| '(|SetCategory|))
(|HasCategory| |#1| '(|Ring|))
(|HasCategory| |#1| (LIST 'PartialDifferentialRing| '(|Symbol|)))
(OR (|HasCategory| |#1| (LIST 'PartialDifferentialRing| '(|Symbol|)))
(|HasCategory| |#1| '(|Ring|)))
(|HasCategory| |#1| '(|Group|))
(|HasCategory| |#1|
(LIST 'InnerEvalable| '(|Symbol|) (|devaluate| |#1|))))
```

```

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

```

The “abbreviation”

EQ

The “parents”

```

(((|Type|) . T)
 ( (|InnerEvalable| (|Symbol|) S) |has| S
   (|InnerEvalable| (|Symbol|) S))
 (|CoercibleTo| (|Boolean|)) |has| S (|SetCategory|))

```

```

(|SetCategory|) |has| S (|SetCategory|)
(|AbelianSemiGroup|) |has| S (|AbelianSemiGroup|)
(|AbelianGroup|) |has| S (|AbelianGroup|)
(|SemiGroup|) |has| S (|SemiGroup|) (|Monoid|) |has| S (|Monoid|)
(|Group|) |has| S (|Group|) (|BiModule| S S) |has| S (|Ring|)
(|Ring|) |has| S (|Ring|) (|Module| S) |has| S (|CommutativeRing|)
(|PartialDifferentialRing| (|Symbol|)) |has| S
  (|PartialDifferentialRing| (|Symbol|))
(|VectorSpace| S) |has| S (|Field|)

```

The “ancestors”

```

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

```

The “documentation”

```

(|constructor|
  (NIL "Equations as mathematical objects. All properties of the basis
    domain,{ } \spadignore{e.g.} being an abelian group are carried
    over the equation domain,{ } by performing the structural operations
    on the left and on the right hand side.")
  (|subst| (($ $ $)
    "\spad{subst(eq1,{ }eq2)} substitutes \spad{eq2} into both sides
    of \spad{eq1} the \spad{lhs} of \spad{eq2} should be a kernel")
  (|inv| (($ $)
    "\spad{inv(x)} returns the multiplicative inverse of \spad{x}.")
  (/ (($ $ $)
    "\spad{e1/e2} produces a new equation by dividing the left and right

```

```

    hand sides of equations \spad{e1} and \spad{e2}."))
(|factorAndSplit|
  (((|List| $) $)
    "\spad{factorAndSplit(eq)} make the right hand side 0 and factors the
    new left hand side. Each factor is equated to 0 and put into the
    resulting list without repetitions."))
(|rightOne|
  (((|Union| $ "failed") $)
    "\spad{rightOne(eq)} divides by the right hand side."))
  (((|Union| $ "failed") $)
    "\spad{rightOne(eq)} divides by the right hand side,{ } if possible."))
(|leftOne|
  (((|Union| $ "failed") $)
    "\spad{leftOne(eq)} divides by the left hand side."))
  (((|Union| $ "failed") $)
    "\spad{leftOne(eq)} divides by the left hand side,{ } if possible."))
(* (($ $ |#1|)
  "\spad{eqn*x} produces a new equation by multiplying both sides of
  equation eqn by \spad{x}."))
(($ |#1| $)
  "\spad{x*eqn} produces a new equation by multiplying both sides of
  equation eqn by \spad{x}."))
(- (($ $ |#1|)
  "\spad{eqn-x} produces a new equation by subtracting \spad{x} from
  both sides of equation eqn."))
(($ |#1| $)
  "\spad{x-eqn} produces a new equation by subtracting both sides of
  equation eqn from \spad{x}."))
(|rightZero|
  (($ $) "\spad{rightZero(eq)} subtracts the right hand side."))
(|leftZero|
  (($ $) "\spad{leftZero(eq)} subtracts the left hand side."))
(+ (($ $ |#1|)
  "\spad{eqn+x} produces a new equation by adding \spad{x} to both
  sides of equation eqn."))
(($ |#1| $)
  "\spad{x+eqn} produces a new equation by adding \spad{x} to both
  sides of equation eqn."))
(|eval| (($ $ (|List| $))
  "\spad{eval(eqn,{ } [x1=v1,{ } ... xn=vn])} replaces \spad{xi}
  by \spad{vi} in equation \spad{eqn}."))
  (($ $ $)
    "\spad{eval(eqn,{ } x=f)} replaces \spad{x} by \spad{f} in
    equation \spad{eqn}."))
(|map| (($ (|Mapping| |#1| |#1|) $)
  "\spad{map(f,{ }eqn)} constructs a new equation by applying
  \spad{f} to both sides of \spad{eqn}."))
(|rhs| (($ |#1| $)
  "\spad{rhs(eqn)} returns the right hand side of equation
  \spad{eqn}."))

```



```
(|lhs| ((|#1| $)
  "\\spad{lhs(eqn)} returns the left hand side of equation
  \\spad{eqn}.")
(|swap| (($ $)
  "\\spad{swap(eq)} interchanges left and right hand side of
  equation \\spad{eq}.".))
(|equation|
  (($ |#1| |#1|) "\\spad{equation(a,{b})} creates an equation.".))
(= (($ |#1| |#1|) "\\spad{a=b} creates an equation.")))
```

The “slotInfo”

```
(|Equation|
  (NIL (~= ((38 0 0) NIL (|has| |#1| (|SetCategory|))))
  (|zero?| ((38 0) NIL (|has| |#1| (|AbelianGroup|))))
  (|swap| ((0 0) 22))
  (|subtractIfCan| ((64 0 0) NIL (|has| |#1| (|AbelianGroup|))))
  (|subst| ((0 0 0) 93 (|has| |#1| (|ExpressionSpace|))))
  (|sample|
    ((0) NIL
      (OR (|has| |#1| (|AbelianGroup|))
          (|has| |#1| (|Monoid|)))
      CONST))
  (|rightZero| ((0 0) 8 (|has| |#1| (|AbelianGroup|))))
  (|rightOne| ((64 0) 68 (|has| |#1| (|Monoid|))))
  (|rhs| ((6 0) 21))
  (|recip| ((64 0) 66 (|has| |#1| (|Monoid|))))
  (|one?| ((38 0) NIL (|has| |#1| (|Monoid|))))
  (|map| ((0 23 0) 24)) (|lhs| ((6 0) 9))
  (|leftZero| ((0 0) 57 (|has| |#1| (|AbelianGroup|))))
  (|leftOne| ((64 0) 67 (|has| |#1| (|Monoid|))))
  (|latex| ((97 0) NIL (|has| |#1| (|SetCategory|))))
  (|inv| ((0 0) 70
    (OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|)))))
  (|hash| ((96 0) NIL (|has| |#1| (|SetCategory|))))
  (|factorAndSplit| ((18 0) 19 (|has| |#1| (|IntegralDomain|))))
  (|eval| ((0 0 28 29) 31
    (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|)))
    ((0 0 25 6) 27
      (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|)))
    ((0 0 18) 37
      (AND (|has| |#1| (|Evalable| |#1|))
          (|has| |#1| (|SetCategory|))))
    ((0 0 0) 34
      (AND (|has| |#1| (|Evalable| |#1|))
          (|has| |#1| (|SetCategory|)))))
  (|equation| ((0 6 6) 17))
  (|dimension| ((86) 88 (|has| |#1| (|Field|))))
  (|differentiate|
```

```

((0 0 25 71) NIL
  (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
((0 0 28 95) NIL
  (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
((0 0 25) 85
  (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
((0 0 28) NIL
  (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
(|conjugate| ((0 0 0) NIL (|has| |#1| (|Group|))))
(|commutator| ((0 0 0) NIL (|has| |#1| (|Group|))))
(|coerce| ((38 0) 45 (|has| |#1| (|SetCategory|))))
  ((41 0) 44 (|has| |#1| (|SetCategory|)))
  ((0 74) NIL (|has| |#1| (|Ring|)))
(|characteristic| ((71) 73 (|has| |#1| (|Ring|))))
(^ ((0 0 94) NIL (|has| |#1| (|SemiGroup|)))
  ((0 0 71) NIL (|has| |#1| (|Monoid|)))
  ((0 0 74) NIL (|has| |#1| (|Group|))))
(|Zero| ((0) 55 (|has| |#1| (|AbelianGroup|)) CONST))
(|One| ((0) 63 (|has| |#1| (|Monoid|)) CONST))
(D ((0 0 25 71) NIL
  (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
  ((0 0 28 95) NIL
  (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
  ((0 0 25) NIL
  (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
  ((0 0 28) NIL
  (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
  (= ((0 6 6) 20) ((38 0 0) 40 (|has| |#1| (|SetCategory|))))
  (/ ((0 0 6) NIL (|has| |#1| (|Field|)))
  ((0 0 0) 90
  (OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|)))))
(- ((0 0 6) 54 (|has| |#1| (|AbelianGroup|)))
  ((0 6 0) 53 (|has| |#1| (|AbelianGroup|)))
  ((0 0 0) 52 (|has| |#1| (|AbelianGroup|)))
  ((0 0) 51 (|has| |#1| (|AbelianGroup|))))
(+ ((0 0 6) 49 (|has| |#1| (|AbelianSemiGroup|)))
  ((0 6 0) 48 (|has| |#1| (|AbelianSemiGroup|)))
  ((0 0 0) 47 (|has| |#1| (|AbelianSemiGroup|))))
(** ((0 0 94) NIL (|has| |#1| (|SemiGroup|)))
  ((0 0 71) NIL (|has| |#1| (|Monoid|)))
  ((0 0 74) NIL (|has| |#1| (|Group|))))
(* ((0 6 0) 60 (|has| |#1| (|SemiGroup|)))
  ((0 0 6) 61 (|has| |#1| (|SemiGroup|)))
  ((0 0 0) 59 (|has| |#1| (|SemiGroup|)))
  ((0 94 0) NIL (|has| |#1| (|AbelianSemiGroup|)))
  ((0 74 0) 76 (|has| |#1| (|AbelianGroup|)))
  ((0 71 0) NIL (|has| |#1| (|AbelianGroup|))))))

```

The “index”

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


Chapter 3

Compiler top level

3.1 Global Data Structures

3.2 Pratt Parsing

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

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

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

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

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

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

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

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

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

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

For tokens like “and”, “or”, +, *, and the Booleans and Algebras can be subdivided into pseudo-classes so that

terms < factors < primaries

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

3.3)compile

This is the implementation of the)compile command.

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

User Level Required: compiler

Command Syntax:

```
)compile
)compile fileName
)compile fileName.spad
)compile directory/fileName.spad
)compile fileName )old
)compile fileName )translate
)compile fileName )quiet
)compile fileName )noquiet
```

```

)compile fileName )moreargs
)compile fileName )onlyargs
)compile fileName )break
)compile fileName )nobreak
)compile fileName )library
)compile fileName )nolibrary
)compile fileName )vartrace
)compile fileName )constructor nameOrAbbrev

```

These command forms invoke the Aldor compiler.

```

)compile fileName.as
)compile directory/fileName.as
)compile fileName.ao
)compile directory/fileName.ao
)compile fileName.al
)compile directory/fileName.al
)compile fileName.lsp
)compile directory/fileName.lsp
)compile fileName )new

```

Command Description:

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

```

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

```

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

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

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

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

```

)compile mycode
)co mycode
)co mycode.spad

```

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

This is frequently all you need to compile your file.

This simple command:

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

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

```

)compile mycode )nolibrary

```

By default, the `)library` system command *exposes* all domains and categories it processes. This means that the Axiom 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 .

```


Spad compiler

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

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

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

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

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

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

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

If you do not give any options, all constructors within a file are compiled. Each constructor should have an `)abbreviation` command in the file in which it is defined. We suggest that you place the `)abbreviation` commands at the top of the file in the order in which the constructors are defined.

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

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

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

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

or

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

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

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

3.4 Operator Precedence Table Initialization

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

LED and NUD Tables

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

; ** TABLE PURPOSE

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

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

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

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

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

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

— LEDNUDTables —

```
; ** TABLE CREATION
```

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

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

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

```
(mapcar #'(LAMBDA(J) (MAKENEWOP J '|Led|))
  '((* 800 801) (|rem| 800 801) (|mod| 800 801)
    (|quo| 800 801) (|div| 800 801)
    (/ 800 801) (** 900 901) (^ 900 901)
    (|exquo| 800 801) (+ 700 701)
    (\- 700 701) (\-> 1001 1002) (\<\- 1001 1002)
    (\: 996 997) (\:\: 996 997)
    (\@ 996 997) (|pretend| 995 996)
    (\.) (\! \! 1002 1001)
    (\, 110 111)
    (\; 81 82 (|PARSE-SemiColon|))
    (\< 400 400) (\> 400 400)
    (\<\< 400 400) (\>\> 400 400)
    (\<= 400 400) (\>= 400 400)
    (= 400 400) (~= 400 400)
    (\~= 400 400))
```

```

(|in| 400 400)      (|case| 400 400)
(|add| 400 120)     (|with| 2000 400 (|PARSE-InfixWith|))
(|has| 400 400)
(|where| 121 104)    ; must be 121 for SPAD, 126 for boot--> nboot
(|when| 112 190)
(|otherwise| 119 190 (|PARSE-Suffix|))
(|is| 400 400)      (|isnt| 400 400)
(|and| 250 251)     (|or| 200 201)
(|/\ 250 251)      (|\/ 200 201)
(|\.\. SEGMENT 401 699 (|PARSE-Seg|))
(=> 123 103)
(+-> 995 112)
(== DEF 122 121)
(==> MDEF 122 121)
(| 108 111)         ;was 190 190
(|:- LETD 125 124) (|:= LET 125 124)))

(mapcar #'(LAMBDA (J) (MAKENEWOP J 'Nud)))
'((|for| 130 350 (|PARSE-Loop|))
  (|while| 130 190 (|PARSE-Loop|))
  (|until| 130 190 (|PARSE-Loop|))
  (|repeat| 130 190 (|PARSE-Loop|))
  (|import| 120 0 (|PARSE-Import|) )
  (|unless|)
  (|add| 900 120)
  (|with| 1000 300 (|PARSE-With|))
  (|has| 400 400)
  (\- 701 700) ; right-prec. wants to be -1 + left-prec
  (+ 701 700)
  (\# 999 998)
  (\! 1002 1001)
  (\' 999 999 (|PARSE-Data|))
  (\<\< 122 120 (|PARSE-LabelExpr|))
  (\>\>)
  (^ 260 259 NIL)
  (\-> 1001 1002)
  (\: 194 195)
  (|not| 260 259 NIL)
  (\~ 260 259 nil)
  (\= 400 700)
  (|return| 202 201 (|PARSE-Return|))
  (|leave| 202 201 (|PARSE-Leave|))
  (|exit| 202 201 (|PARSE-Exit|))
  (|from|)
  (|iterate|)
  (|yield|)
  (|if| 130 0 (|PARSE-Conditional|)) ; was 130
  (\ 0 190)
  (|suchthat|)
  (|then| 0 114)

```

```
(|else| 0 114)))
```

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

— GLIPHTable —

```
(mapcar #'(lambda (x) (put (car x) 'glyph (cdr x)))
  '(
    ( \| (\|) )
    ( * (*) )
    ( \ ( (<) (\|) )
    ( + (- (>)) )
    ( - (>) )
    ( < (=) (<) )
    ;; ( / (\|) ) breaks */xxx
    ( \| (/) )
    ( > (=) (>) (\|))
    ( = (= (>)) (>) )
    ( \. (\.) )
    ( ^ (=) )
    ( \~ (=) )
    ( \: (=) (-) (\:)))
```

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

— RENAMETOKTable —

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

Generic function table

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

— **GENERICTable** —

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

3.6 Giant steps, Baby steps

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

Chapter 4

The Parser

4.1 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)]

```

4.2 boot transformations

defun string2BootTree

```

[new2OldLisp p72]
[def-rename p587]
[boot-line-stack p??]
[xtokenreader p??]
[line-handler p??]
[$boot p??]

```

[[\\$spad p573](#)]

— defun string2BootTree —

```
(defun string2BootTree (s)
  (init-boot/spad-reader)
  (let* ((boot-line-stack (list (cons 1 s)))
        ($boot t)
        ($spad nil)
        (xtokenreader 'get-boot-token)
        (line-handler 'next-boot-line)
        (parseout (progn (|PARSE-Expression|) (pop-stack-1))))
    (declare (special boot-line-stack $boot $spad xtokenreader line-handler))
    (def-rename (new2OldLisp parseout))))
```

—————

defun new2OldLisp

[[new2OldTran p72](#)]
 [[postTransform p365](#)]

— defun new2OldLisp —

```
(defun new2OldLisp (x)
  (new2OldTran (postTransform x)))
```

—————

defun new2OldTran

[[dcq p??](#)]
 [[new2OldTran p72](#)]
 [[newDef2Def p74](#)]
 [[newIf2Cond p73](#)]
 [[newConstruct p74](#)]
 [[\\$new2OldRenameAssoc p??](#)]

— defun new2OldTran —

```
(defun new2OldTran (x)
  (prog (tmp1 tmp2 tmp3 tmp4 tmp5 tmp6 a b c d)
    (declare (special |$new2OldRenameAssoc|))
    (return
```

```

(prog nil
  (if (atom x)
    (return (let ((y (assoc x |$new2OldRenameAssoc|)))
      (if y (cdr y) x))))
  (if (and (dcq (tmp1 a b . tmp2) x)
    (null tmp2)
    (eq tmp1 '|where|)
    (dcq (tmp3 . tmp4) b)
    (dcq ((tmp5 d . tmp6) . c) (reverse tmp4))
    (null tmp6)
    (eq tmp5 '|exit|)
    (eq tmp3 'seq)
    (or (setq c (nreverse c)) t))
    (return
      '(|where| ,(new2OldTran a) ,@(new2OldTran c)
        ,(new2OldTran d))))
  (return
    (case (car x)
      (quote x)
      (def (newDef2Def x))
      (if (newIf2Cond x))
      ; construct == #'list (see patches.lisp) TPD 12/2011
      (|construct| (newConstruct (new2OldTran (cdr x))))
      (t '(',(new2OldTran (car x)) . ,(new2OldTran (cdr x)))))))

```

defun newIf2Cond

[letError p??]
[new2OldTran p72]

— defun newIf2Cond —

```

(defun newIf2Cond (cond-expr)
  (if (not (and (= (length cond-expr) 4) (eq (car cond-expr) 'if)))
    (letError "(IF,a,b,c)" cond-expr)
    (let ((a (second cond-expr))
      (b (third cond-expr))
      (c (fourth cond-expr)))
      (setq a (new2OldTran a) b (new2OldTran b) c (new2OldTran c))
      (if (eq c '|noBranch|)
        '(if ,a ,b)
        '(if ,a ,b ,c))))

```

```
(defun letError (form val) (—systemError— (format nil " S is not matched by structure S
```

defun newDef2Def

```
[letError p??]  
[new2OldDefForm p74]  
[new2OldTran p72]
```

— defun newDef2Def —

```
(defun newDef2Def (def-expr)  
  (if (not (and (= (length def-expr) 5) (eq (car def-expr) 'def)))  
      (letError "(DEF,form,a,b,c)" def-expr)  
      (let ((form (second def-expr))  
            (a (third def-expr))  
            (b (fourth def-expr))  
            (c (fifth def-expr)))  
        '(def ,(new2OldDefForm form) ,(new2OldTran a)  
              ,(new2OldTran b) ,(new2OldTran c)))))
```

defun new2OldDefForm

```
[new2OldTran p72]  
[new2OldDefForm p74]
```

— defun new2OldDefForm —

```
(defun new2OldDefForm (x)  
  (cond  
    ((atom x) (new2OldTran x))  
    ((and (listp x) (listp (car x)) (eq (caar x) '|is|) (= (length (car x)) 3))  
      (let ((a (second (car x))) (b (third (car x))) (y (cdr x)))  
        (new2OldDefForm '( (spadlet ,a ,b) ,@y))))  
    ((cons (new2OldTran (car x)) (new2OldDefForm (cdr x))))))
```

defun newConstruct

— defun newConstruct —

```
(defun newConstruct (z)
  (if (atom z)
      z
      '(cons ,(car z) ,(newConstruct (cdr z)))))
```

4.3 preparse

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

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

defvar \$index

— initvars —

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

defvar \$linelist

— initvars —

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

defvar \$echolinestack

— initvars —

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

```
defvar $preparse-last-line
```

```
— initvars —
```

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

4.4 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")
```

```
defun initialize-preparse
```

```
[get-a-line p638]
[$index p75]
[$linelist p75]
[$echolinestack p75]
[$preparse-last-line p76]
```

```
— 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 . "      1:S * eqn:$ == 1 * eqn.lhs = 1 * eqn.rhs;")
(159 . "      1:S * eqn:$ == 1 * eqn.lhs = 1 * eqn.rhs;")
(160 . "      eqn:$ * 1:S == eqn.lhs * 1 = eqn.rhs * 1);")
(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)])))))

```

defun preparse

```

[preparse p80]
[preparse1 p84]
[parseprint p556]
[ifcar p??]
[$comblocklist p553]
[$skipme p??]
[$preparse-last-line p76]
[$index p75]
[$docList p??]
[$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 (consp $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 . "        1:S * eqn:$ == 1 * eqn.lhs = 1 * eqn.rhs;")
(159 . "        1:S * eqn:$ == 1 * eqn.lhs = 1 * eqn.rhs;")
(160 . "        eqn:$ * 1:S == eqn.lhs * 1 = eqn.rhs * 1);")
(161 . "    if S has Monoid then")
(162 . "        (1 == equation(1$S,1$S);")
(163 . "        recip eq ==")
(164 . "            ((lh := recip lhs eq) case \"failed\" => \"failed\");")
(165 . "            (rh := recip rhs eq) case \"failed\" => \"failed\");")
(166 . "            [lh :: S, rh :: S]);")
(167 . "        leftOne eq ==")
(168 . "            ((re := recip lhs eq) case \"failed\" => \"failed\");")
(169 . "            1 = rhs eq * re);")
(170 . "        rightOne eq ==")
(171 . "            ((re := recip rhs eq) case \"failed\" => \"failed\");")
(172 . "            lhs eq * re = 1));")
(173 . "    if S has Group then")
(174 . "        (inv eq == [inv lhs eq, inv rhs eq];")
(175 . "        leftOne eq == 1 = rhs eq * inv rhs eq;")
(176 . "        rightOne eq == lhs eq * inv rhs eq = 1);")
(177 . "    if S has Ring then")
(178 . "        (characteristic() == characteristic()$S;")
(179 . "        i:Integer * eq:$ == (i::S) * eq);")
(180 . "    if S has IntegralDomain then")
(181 . "        factorAndSplit eq ==")
(182 . "            ((S has factor : S -> Factored S) =>")
(183 . "                (eq0 := rightZero eq;")
(184 . "                [equation(rcf.factor,0)"]
(185 . "                    for rcf in factors factor lhs eq0]));")
(186 . "            (S has Polynomial Integer) =>")
(187 . "                (eq0 := rightZero eq;")
(188 . "                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)])))"))

```

defun Build the lines from the input for piles

The READLOOP calls `preparseReadLine` which returns a pair of the form

```
(number . string)
```

```

[preparseReadLine p89]
[preparse-echo p92]
[fincomblock p553]
[parsepiles p87]
[preparse1 doSystemCommand (vol5)]
[escaped p553]
[indent-pos p554]
[make-full-cvec p??]
[maxindex p??]
[preparse1 strposl (vol5)]
[is-console p555]
[spad-reader p??]
[$echolinestack p75]
[$byConstructors p629]
[$skipme p??]
[$constructorsSeen p629]
[$preparse-last-line p76]
[$preparse-last-line p76]
[$index p75]
[$index p75]
[$linelist p75]
[$in-stream p??]

```


— defun preparse1 —

```
(defun preparse1 (linelist)
  (labels (
    (isSystemCommand (line)
      (and (> (length line) 0) (eq (char line 0) #\ ) )))
    (executeSystemCommand (line)
      (catch 'spad_reader (|doSystemCommand| (subseq line 1)))))
  )
  (prog (($linelist linelist) $echolinestack num line i l psloc
    instring pcount comsym strsym oparsym cparsym n ncomsym tmp1
    (sloc -1) continue (parenlev 0) ncomblock lines locs nums functor)
    (declare (special $linelist $echolinestack |$byConstructors| $skipme
      |$constructorsSeen| $preparse-last-line $index in-stream))
  READLOOP
    (setq tmp1 (preparseReadLine linelist))
    (setq num (car tmp1))
    (setq line (cdr tmp1))
    (unless (stringp line)
      (preparse-echo linelist)
      (cond
        ((null lines) (return nil))
        (ncomblock (fincomblock nil nums locs ncomblock nil)))
      (return
        (pair (nreverse nums) (parsepiles (nreverse locs) (nreverse lines)))))
    (when (and (null lines) (isSystemCommand line))
      (preparse-echo linelist)
      (setq $preparse-last-line nil) ;don't reread this line
      (executeSystemCommand line)
      (go READLOOP))
    (setq l (length line))
    ; if we get a null line, read the next line
    (when (eq l 0) (go READLOOP))
    ; otherwise we have to parse this line
    (setq psloc sloc)
    (setq i 0)
    (setq instring nil)
    (setq pcount 0)
  STRLOOP ;; handle things that need ignoring, quoting, or grouping
    ; are we in a comment, quoting, or grouping situation?
    (setq strsym (or (position #\" line :start i ) l))
    (setq comsym (or (search "--" line :start2 i ) l))
    (setq ncomsym (or (search "++" line :start2 i ) l))
    (setq oparsym (or (position #\"( line :start i ) l))
    (setq cparsym (or (position #\\) line :start i ) l))
    (setq n (min strsym comsym ncomsym oparsym cparsym))
    (cond
      ; nope, we found no comment, quoting, or grouping
      ((= n l) (go NOCOMS))
```

```

((escaped line n))
; scan until we hit the end of the string
((= n strsym) (setq instring (not instring)))
; we are in a string, just continue looping
(instring)
;; handle -- comments by ignoring them
((= n comsym)
 (setq line (subseq line 0 n))
 (go NOCOMS)) ; discard trailing comment
;; handle ++ comments by chunking them together
((= n ncomsym)
 (setq sloc (indent-pos line))
 (cond
  ((= sloc n)
   (when (and ncomblock (not (= n (car ncomblock))))
    (fincomblock num nums locs ncomblock linelist)
    (setq ncomblock nil))
   (setq ncomblock (cons n (cons line (ifcdr ncomblock))))
   (setq line ""))
  (t
   (push (strconc (make-full-cvec n " ") (substring line n ())) $linelist)
   (setq $index (1- $index))
   (setq line (subseq line 0 n))))
 (go NOCOMS))
; know how deep we are into parens
((= n oparsym) (setq pcount (1+ pcount)))
((= n cparsym) (setq pcount (1- pcount)))
(setq i (1+ n))
(go STRLOOP)
NOCOMS
; remember the indentation level
(setq sloc (indent-pos line))
(setq line (string-right-trim " " line))
(when (null sloc)
 (setq sloc psloc)
 (go READLOOP))
; handle line that ends in a continuation character
(cond
 ((eq (elt line (maxindex line)) #\_)
  (setq continue t)
  (setq line (subseq line (maxindex line))))
 ((setq continue nil)))
; test for skipping constructors
(when (and (null lines) (= sloc 0))
 (if (and |$byConstructors|
  (null (search "==" line))
  (not
   (member
    (setq functor
     (intern (substring line 0 (strpos1 ": (" line 0 nil))))

```

```

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

```

defun parsepiles

Add parens and semis to lines to aid parsing. [add-parens-and-semis-to-line p88]

— defun parsepiles —

```

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

```

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 p555]
[drop p553]
[addclose p552]
[nonblankloc p555]
```

— 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) #\ ) ))))))))
```

defun preparseReadLine

```
[preparseReadLine1 p90]
[initial-substring p637]
[string2BootTree p71]
[storeblanks p637]
[skip-to-endif p556]
[preparseReadLine p89]
[*eof* p??]
```

— defun preparseReadLine —

```
(defun preparseReadLine (x)
  (let (line ind tmp1)
    (declare (special *eof*))
    (setq tmp1 (preparseReadLine1))
    (setq ind (car tmp1))
    (setq line (cdr tmp1))
    (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)))
```

—————

defun skip-ifblock

```
[preparseReadLine1 p90]
[skip-ifblock p89]
[initial-substring p637]
[string2BootTree p71]
[storeblanks p637]
```

— defun skip-ifblock —

```

(defun skip-ifblock (x)
  (let (line ind tmp1)
    (setq tmp1 (preparseReadLine1))
    (setq ind (car tmp1))
    (setq line (cdr tmp1))
    (cond
      ((not (stringp line))
       (cons ind line))
      ((zerop (size line))
       (skip-ifblock x))
      ((char= (elt line 0) #\ )
       (cond
         ((initial-substring ")if" line)
          (cond
            ((eval (string2BootTree (storeblanks line 3)))
             (preparseReadLine X))
            (t (skip-ifblock x))))
         ((initial-substring ")elseif" line)
          (cond
            ((eval (string2BootTree (storeblanks line 7)))
             (preparseReadLine X))
            (t (skip-ifblock x))))
         ((initial-substring ")else" line)
          (preparseReadLine x))
         ((initial-substring ")endif" line)
          (preparseReadLine x))
         ((initial-substring ")fin" line)
          (cons ind nil))))
      (t (skip-ifblock x)))))

```

defun preparseReadLine1

```

[get-a-line p638]
[expand-tabs p91]
[maxindex p??]
[strconc p??]
[preparseReadLine1 p90]
[$linelist p75]
[$linelist p75]
[$preparse-last-line p76]
[$index p75]
[$index p75]
[$EchoLineStack p??]

```

— defun preparseReadLine1 —

```

(defun preparsedReadLine1 ()
  (labels (
    (accumulateLinesWithTrailingEscape (line)
      (let (ind)
        (declare (special $preparse-last-line))
        (if (and (> (setq ind (maxindex line)) -1) (char= (elt line ind) #\_))
            (setq $preparse-last-line
              (strconc (substring line 0 ind) (cdr (preparsedReadLine1))))
            line))))
    (let (line)
      (declare (special $linelist $preparse-last-line $index $EchoLineStack))
      (setq line
        (if $linelist
          (pop $linelist)
          (expand-tabs (get-a-line in-stream))))
      (setq $preparse-last-line line)
      (if (stringp line)
        (progn
          (incf $index) ;; $index is the current line number
          (setq line (string-right-trim " " line))
          (push (copy-seq line) $EchoLineStack)
          (cons $index (accumulateLinesWithTrailingEscape line)))
        (cons $index line))))))

```

defun expand-tabs

[nonblankloc p555]

[indent-pos p554]

— defun expand-tabs —

```

(defun expand-tabs (str)
  (if (and (stringp str) (> (length str) 0))
    (let ((bpos (nonblankloc str))
          (tpos (indent-pos str)))
      (setq str
        (if (eql bpos tpos)
          str
          (concatenate 'string (make-string tpos :initial-element #\space)
            (subseq str bpos))))
      ;; remove dos CR
      (let ((lpos (maxindex str)))
        (if (eq (char str lpos) #\Return)
          (subseq str 0 lpos)
          str))))

```

```
str))
```

4.5 I/O Handling

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 ()))
```

Parsing stack

defstruct \$stack

— initvars —

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

defun stack-load

```
[$stack p92]
```


— defun stack-load —

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

—————

defun stack-clear

[[\\$stack p92](#)]

— defun stack-clear —

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

—————

defmacro stack-/-empty

[[\\$stack p92](#)]

— defmacro stack-/-empty —

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

—————

defun stack-push

[[\\$stack p92](#)]

— defun stack-push —

```
(defun stack-push (x stack)
  (push x (stack-store stack))
  (setf (stack-top stack) x))
```

```
(setf (stack-updated stack) t)
(incf (stack-size stack))
x)
```

defun stack-pop

[[\\$stack](#) [p92](#)]

— defun stack-pop —

```
(defun stack-pop (stack)
  (let ((y (pop (stack-store stack))))
    (decf (stack-size stack))
    (setf (stack-top stack)
          (if (stack-/empty stack) (car (stack-store stack)))
          y))
```

Parsing token

defstruct \$token

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

— initvars —

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

defvar \$prior-token

[[\\$token](#) [p94](#)]

— initvars —

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

defvar \$nonblank

— initvars —

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

defvar \$current-token

Token at head of input stream. [Token p94]

— initvars —

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

defvar \$next-token

[Token p94]

— initvars —

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

defvar \$valid-tokens

[Token p94]

— initvars —

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

defun token-install

[\$token p94]

— defun token-install —

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

defun token-print

[\$token p94]

— defun token-print —

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

Parsing reduction**defstruct \$reduction**

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

— **initvars** —

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

Chapter 5

Parse Transformers

5.1 Direct called parse routines

defun parseTransform

[parseTran p97]
[\$defOp p??]

— defun parseTransform —

```
(defun |parseTransform| (x)
  (let (|$defOp|)
    (declare (special |$defOp|))
    (setq |$defOp| nil)
    (setq x (subst '$ '% x :test #'equal)) ; for new compiler compatibility
    (|parseTran| x)))
```

—————

defun parseTran

[parseAtom p98]
[parseConstruct p99]
[parseTran p97]
[parseTranList p99]
[getl p??]
[\$op p??]

— defun parseTran —

```

(defun |parseTran| (x)
  (labels (
    (g (op)
      (let (tmp1 tmp2 x)
        (seq
          (if (and (consp op) (eq (qfirst op) '|elt|)
              (progn
                (setq tmp1 (qrest op))
                (and (consp tmp1)
                  (progn
                    (setq op (qfirst tmp1))
                    (setq tmp2 (qrest tmp1))
                    (and (consp tmp2)
                      (eq (qrest tmp2) nil)
                      (progn (setq x (qfirst 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 |$op|))
          (cond
            ((eq u '|construct|)
             (setq r (|parseConstruct| argl))
             (if (and (consp |$op|) (eq (qfirst |$op|) '|elt|))
                 (cons (|parseTran| |$op|) (cdr r))
                 r))
            ((and (atom u) (setq fn (getl u '|parseTran|)))
             (funcall fn argl))
            (t (cons (|parseTran| |$op|) (|parseTranList| argl)))))))

```

defun parseAtom

[parseLeave p125]
 [\$NoValue p??]

— defun parseAtom —

```

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

```

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

defun parseTranList

```
[parseTran p97]
[parseTranList p99]
```

— defun parseTranList —

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

deflist parseConstruct

— postvars —

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

defun parseConstruct

```
[parseTranList p99]
[$insideConstructIfTrue p??]
```

— defun parseConstruct —

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

5.2 Indirect called parse routines

In the `parseTran` function there is the code:

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

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

<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>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>


```

SEQ          parseSeq
;;control-V  parseUpArrow
VCONS       parseVCONS
where        parseWhere

```

defplist parseAnd

— postvars —

```

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

```

—————

defun parseAnd

```

[parseTran p97]
[parseAnd p101]
[parseTranList p99]
[parseIf p118]
[$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| )))))

```

—————

defplist parseAtSign

— postvars —

```

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

```

defun parseAtSign

```
[parseTran p97]
[parseType p102]
[$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)))))
```

defun parseType

```
[parseTran p97]

— defun parseType —

(defun |parseType| (x)
  (declare (special |$EmptyMode| |$quadSymbol|))
  (setq x (subst |$EmptyMode| |$quadSymbol| x :test #'equal))
  (if (and (consp x) (eq (qfirst x) '|typeOf|)
        (consp (qrest x)) (eq (qcddr x) nil))
      (list '|typeOf| (|parseTran| (qsecond x)))
      x))
```

defplist parseCategory

```
— postvars —

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

defun parseCategory

[parseTranList p99]
 [parseDropAssertions p103]
 [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))))
```

—————

defun parseDropAssertions

[parseDropAssertions p103]

— defun parseDropAssertions —

```
(defun |parseDropAssertions| (x)
  (cond
    ((not (consp x)) x)
    ((and (consp (qfirst x)) (eq (qcaar x) 'if)
          (consp (qcдар x))
          (eq (qcadar x) '|asserted|))
     (|parseDropAssertions| (qrest x)))
    (t (cons (qfirst x) (|parseDropAssertions| (qrest x))))))
```

—————

defplist parseCoerce

— postvars —

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

—————

defun parseCoerce

```
[parseType p102]
[parseTran p97]
[$InteractiveMode p??]

— defun parseCoerce —

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

defplist parseColon

```
— postvars —

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

defun parseColon

```
[parseTran p97]
[parseType p102]
[$InteractiveMode p??]
[$insideConstructIfTrue p??]

— defun parseColon —

(defun |parseColon| (arg)
  (declare (special |$insideConstructIfTrue|))
  (cond
    ((and (consp arg) (eq (qrest arg) nil))
     (list '|::| (|parseTran| (first arg))))
    ((and (consp arg) (consp (qrest arg)) (eq (qcddr arg) nil))
     (if |$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)))))

```

defplist parseDEF

— postvars —

```

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

```

defun parseDEF

```

[setDefOp p400]
[parseLhs p106]
[parseTranList p99]
[parseTranCheckForRecord p545]
[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|)))))

```

defun parseLhs

[parseTran p97]
[transIs p106]

— defun parseLhs —

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

—————

defun transIs

[isListConstructor p107]
[transIs1 p106]

— defun transIs —

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

—————

defun transIs1

[nreverse0 p??]
[transIs p106]
[transIs1 p106]

— defun transIs1 —

```
(defun |transIs1| (u)
  (let (x h v tmp3)
    (cond
```

```

((and (consp u) (eq (qfirst u) '|construct|))
  (dolist (x (qrest u) (nreverse0 tmp3))
    (push (|transIs| x) tmp3)))
((and (consp u) (eq (qfirst u) '|append|) (consp (qrest u))
  (consp (qcddr u)) (eq (qcdddr u) nil))
  (setq x (qsecond u))
  (setq h (list '|:| (|transIs| x)))
  (setq v (|transIs1| (qthird u)))
  (cond
    ((and (consp v) (eq (qfirst v) '|:|)
      (consp (qrest v)) (eq (qcddr v) nil))
      (list h (qsecond v)))
    ((eq v '|nil|) (car (cdr h)))
    ((atom v) (list h (list '|:| v)))
    (t (cons h v))))
((and (consp u) (eq (qfirst u) '|cons|) (consp (qrest u))
  (consp (qcddr u)) (eq (qcdddr u) nil))
  (setq h (|transIs| (qsecond u)))
  (setq v (|transIs1| (qthird u)))
  (cond
    ((and (consp v) (eq (qfirst v) '|:|) (consp (qrest v))
      (eq (qcddr v) nil))
      (cons h (list (qsecond v))))
    ((eq v '|nil|) (cons h nil))
    ((atom v) (list h (list '|:| v)))
    (t (cons h v))))
(t u))))

```

defun isListConstructor

[member p??]

— defun isListConstructor —

```

(defun |isListConstructor| (u)
  (and (consp u) (|member| (qfirst u) '(|construct| |append| |cons|))))

```

defplist parseDollarGreaterthan

— postvars —

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

defun parseDollarGreaterThan

```
[parseTran p97]
[$op p??]
```

— defun parseDollarGreaterThan —

```
(defun |parseDollarGreaterThan| (arg)
  (declare (special |$op|))
  (list (subst '$<' '$>' |$op| :test #'equal)
        (|parseTran| (second arg))
        (|parseTran| (first arg))))
```

deflist parseDollarGreaterEqual

— postvars —

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

defun parseDollarGreaterEqual

```
[parseTran p97]
[$op p??]
```

— defun parseDollarGreaterEqual —

```
(defun |parseDollarGreaterEqual| (arg)
  (declare (special |$op|))
  (|parseTran| (list '|not| (cons (subst '$<' '$>=' |$op| :test #'equal) arg)))))
```

— postvars —

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

defun parseDollarLessEqual

```
[parseTran p97]
[$op p??]
```

— defun parseDollarLessEqual —

```
(defun |parseDollarLessEqual| (arg)
  (declare (special |$op|))
  (|parseTran| (list '|not| (cons (subst '$> '$<= |$op| :test #'equal) arg)))))
```

deflist parseDollarNotEqual

— postvars —

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

defun parseDollarNotEqual

```
[parseTran p97]
[$op p??]
```

— defun parseDollarNotEqual —

```
(defun |parseDollarNotEqual| (arg)
  (declare (special |$op|))
  (|parseTran| (list '|not| (cons (subst '$= '$^= |$op| :test #'equal) arg)))))
```

defplist parseEquivalence

— postvars —

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

defun parseEquivalence

[parseIf p118]

— defun parseEquivalence —

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

defplist parseExit

— postvars —

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

defun parseExit

```
[parseTran p97]
[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 )))))
```

defpllist parseGreaterEqual

— postvars —

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

defun parseGreaterEqual

```
[parseTran p97]
[$op p??]
```

— defun parseGreaterEqual —

```
(defun |parseGreaterEqual| (arg)
  (declare (special |$op|))
  (|parseTran| (list '|not| (cons (subst '<' '>=' |$op| :test #'equal) arg)))))
```

defpllist parseGreaterThan

— postvars —

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

defun parseGreaterThan

```
[parseTran p97]
[$op p??]
```

— defun parseGreaterThan —

```
(defun |parseGreaterThan| (arg)
  (declare (special |$op|))
  (list (subst '<' '>' |$op| :test #'equal)
        (|parseTran| (second arg)) (|parseTran| (first arg))))
```

defplist parseHas

— postvars —

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

defun parseHas

```
[unabbrevAndLoad p??]
[getdatabase p??]
[opOf p??]
[makeNonAtomic p??]
[parseHasRhs p114]
[member p??]
[parseType p102]
[nreverse0 p??]
[$InteractiveMode p??]
[$CategoryFrame p??]
```

— defun parseHas —

```

(defun |parseHas| (arg)
  (labels (
    (fn (arg)
      (let (tmp4 tmp6 map op kk)
        (declare (special |$InteractiveMode|))
        (when |$InteractiveMode| (setq arg (|unabbrevAndLoad| arg)))
        (cond
          ((and (consp arg) (eq (qfirst arg) '|:|) (consp (qrest arg))
            (consp (qcddr arg)) (eq (qcdddr arg) nil)
            (consp (qthird arg))
            (eq (qcaaddr 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 (consp arg) (eq (qfirst arg) '|Join|))
           (dolist (z (rest arg) tmp4)
             (setq tmp4 (append tmp4 (fn z))))))
          ((and (consp arg) (eq (qfirst 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 (consp arg) (eq (qfirst arg) 'attribute))
             (list arg))
            ((and (consp arg) (eq (qfirst 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 (consp tmp1) (consp (qrest tmp1)) (consp (qcddr tmp1))
            (eq (qcdddr 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 (consp tmp2) (eq (qrest tmp2) nil))

```

```
(qfirst tmp2)
(cons '|and| tmp2))))))
```

defun parseHasRhs

```
[get p??]
[member p??]
[abbreviation? p??]
[loadIfNecessary p114]
[unabbrevAndLoad p??]
[$CategoryFrame p??]
```

— defun parseHasRhs —

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

defun loadIfNecessary

```
[loadLibIfNecessary p115]
```

— defun loadIfNecessary —

```
(defun |loadIfNecessary| (u)
  (|loadLibIfNecessary| u t))
```

defun loadLibIfNecessary

```
[loadLibIfNecessary p115]
[canFuncall? p??]
[macrop p??]
[getl p??]
[loadLib p??]
[lassoc p??]
[getProplist p??]
[getdatabase p??]
[updateCategoryFrameForCategory p117]
[updateCategoryFrameForConstructor p116]
[throwKeyedMsg p??]
[$CategoryFrame p??]
[$InteractiveMode p??]
```

— defun loadLibIfNecessary —

```
(defun |loadLibIfNecessary| (u mustExist)
  (let (value y)
    (declare (special |$CategoryFrame| |$InteractiveMode|))
    (cond
      ((eq u '|$EmptyMode|) u)
      ((null (atom u)) (|loadLibIfNecessary| (car u) mustExist))
      (t
       (setq value
        (cond
          ((or (canFuncall? u) (|macrop| u)) u)
          ((getl u 'loaded) u)
          ((|loadLib| u) u)))
        (cond
          ((and (null |$InteractiveMode|)
               (or (null (setq y (|getProplist| u |$CategoryFrame|)))
                   (and (null (lassoc '|isFunctor| y))
                        (null (lassoc '|isCategory| y))))))
           (if (setq y (getdatabase u 'constructorkind))
               (if (eq y '|category|)
                   (|updateCategoryFrameForCategory| u)
                   (|updateCategoryFrameForConstructor| u))
               (|throwKeyedMsg| 's2il0005 (list u)))
           (t value))))))
```

—————

defun updateCategoryFrameForConstructor

```
[getdatabase p??]
[put p??]
[convertOpAlist2compilerInfo p116]
[addModemap p269]
[$CategoryFrame p??]
[$CategoryFrame p??]
```

— defun updateCategoryFrameForConstructor —

```
(defun |updateCategoryFrameForConstructor| (constructor)
  (let (opAlist tmp1 dc sig pred impl)
    (declare (special |$CategoryFrame|))
    (setq opalist (getdatabase constructor 'operationalist))
    (setq tmp1 (getdatabase constructor 'constructormodemap))
    (setq dc (caar tmp1))
    (setq sig (cdar tmp1))
    (setq pred (caadr tmp1))
    (setq impl (cadadr tmp1))
    (setq |$CategoryFrame|
      (|put| constructor '|isFunctor|
        (|convertOpAlist2compilerInfo| opalist)
        (|addModemap| constructor dc sig pred impl
          (|put| constructor '|mode| (cons '|Mapping| sig) |$CategoryFrame|))))))
```

defun convertOpAlist2compilerInfo

— defun convertOpAlist2compilerInfo —

```
(defun |convertOpAlist2compilerInfo| (opalist)
  (labels (
    (formatSig (op arg2)
      (let (typelist slot stuff pred impl)
        (setq typelist (car arg2))
        (setq slot (cadr arg2))
        (setq stuff (cddr arg2))
        (setq pred (if stuff (car stuff) t))
        (setq impl (if (cdr stuff) (cadr stuff) 'elt))
        (list (list op typelist) pred (list impl '$ slot)))))
    (let (data result)
      (setq data
        (loop for item in opalist
          collect
```



```

(loop for sig in (rest item)
  collect (formatSig (car item) sig))))
(dolist (term data result)
  (setq result (append result term))))))

```

defun updateCategoryFrameForCategory

```

[getdatabase p??]
[put p??]
[addModemap p269]
[$CategoryFrame p??]
[$CategoryFrame p??]

```

— defun updateCategoryFrameForCategory —

```

(defun |updateCategoryFrameForCategory| (category)
  (let (tmp1 dc sig pred impl)
    (declare (special |$CategoryFrame|))
    (setq tmp1 (getdatabase category 'constructormodemap))
    (setq dc (caar tmp1))
    (setq sig (cdar tmp1))
    (setq pred (caadr tmp1))
    (setq impl (cadadr tmp1))
    (setq |$CategoryFrame|
      (|put| category '|isCategory| t
        (|addModemap| category dc sig pred impl |$CategoryFrame|))))))

```

deflist parseIf

— postvars —

```

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

```

defun parseIf

[parseIf,ifTran p118]
 [parseTran p97]

— defun parseIf —

```
(defun |parseIf| (arg)
  (if (null (and (consp arg) (consp (qrest arg))
                (consp (qcddr arg)) (eq (qcdddr arg) nil)))
      arg
      (|parseIf,ifTran|
       (|parseTran| (first arg))
       (|parseTran| (second arg))
       (|parseTran| (third arg)))))
```

—————

defun parseIf,ifTran

[parseIf,ifTran p118]
 [incExitLevel p??]
 [makeSimplePredicateOrNil p546]
 [incExitLevel p??]
 [parseTran p97]
 [\$InteractiveMode p??]

— defun parseIf,ifTran —

```
(defun |parseIf,ifTran| (pred a b)
  (let (pp z ap bp tmp1 tmp2 tmp3 tmp4 tmp5 tmp6 val s)
    (declare (special |$InteractiveMode|))
    (cond
      ((and (null |$InteractiveMode|) (eq pred '|true|))
       a)
      ((and (null |$InteractiveMode|) (eq pred '|false|))
       b)
      ((and (consp pred) (eq (qfirst pred) '|not|)
            (consp (qrest pred)) (eq (qcddr pred) nil))
       (|parseIf,ifTran| (second pred) b a))
      ((and (consp pred) (eq (qfirst pred) '|if|)
            (progn
              (setq tmp1 (qrest pred))
              (and (consp tmp1)
                    (progn
                     (setq pp (qfirst tmp1))
                     (setq tmp2 (qrest tmp1))
```

```

      (and (consp tmp2)
        (progn
          (setq ap (qfirst tmp2))
          (setq tmp3 (qrest tmp2))
          (and (consp tmp3)
            (eq (qrest tmp3) nil)
            (progn (setq bp (qfirst tmp3)) t))))))
    (|parseIf,ifTran| pp
      (|parseIf,ifTran| ap (copy a) (copy b))
      (|parseIf,ifTran| bp a b)))
  ((and (consp pred) (eq (qfirst pred) 'seq)
    (consp (qrest pred)) (progn (setq tmp2 (reverse (qrest pred))) t)
    (and (consp tmp2)
      (consp (qfirst tmp2))
      (eq (qcaar tmp2) '|exit|)
      (progn
        (setq tmp4 (qcddar tmp2))
        (and (consp tmp4)
          (equal (qfirst tmp4) 1)
          (progn
            (setq tmp5 (qrest tmp4))
            (and (consp tmp5)
              (eq (qrest tmp5) nil)
              (progn (setq pp (qfirst tmp5)) t))))
        (progn (setq z (qrest tmp2)) t))
      (progn (setq z (nreverse z)) t))
    (cons 'seq
      (append z
        (list
          (list '|exit| 1 (|parseIf,ifTran| pp
            (|incExitLevel| a)
            (|incExitLevel| b))))))
    ((and (consp a) (eq (qfirst a) 'if) (consp (qrest a))
      (equal (qsecond a) pred) (consp (qcddr a))
      (consp (qcdddr a))
      (eq (qcdddr a) nil))
      (list 'if pred (third a) b))
    ((and (consp b) (eq (qfirst b) 'if)
      (consp (qrest b)) (equal (qsecond b) pred)
      (consp (qcddr b))
      (consp (qcdddr b))
      (eq (qcdddr b) nil))
      (list 'if pred a (fourth b)))
    ((progn
      (setq tmp1 (|makeSimplePredicateOrNil| pred))
      (and (consp tmp1) (eq (qfirst tmp1) 'seq)
        (progn
          (setq tmp2 (qrest tmp1))
          (and (consp tmp2)
            (progn (setq tmp3 (reverse tmp2)) t))

```

```

      (and (consp tmp3)
        (progn
          (setq tmp4 (qfirst tmp3))
          (and (consp tmp4) (eq (qfirst tmp4) '|exit|)
            (progn
              (setq tmp5 (qrest tmp4))
              (and (consp tmp5) (equal (qfirst tmp5) 1)
                (progn
                  (setq tmp6 (qrest tmp5))
                  (and (consp tmp6) (eq (qrest tmp6) nil)
                    (progn (setq val (qfirst tmp6)) t)))))))
              (progn (setq s (qrest tmp3)) t))))))
      (setq s (nreverse s))
      (|parseTran|
        (cons 'seq
          (append s
            (list (list '|exit| 1 (|incExitLevel| (list 'if val a b)))))))
      (t
        (list 'if pred a b )))))

```

defplist parseImplies

— postvars —

```

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

```

defun parseImplies

[parseIf [p118](#)]

— defun parseImplies —

```

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

```

defplist parseIn

— postvars —

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

—————

defun parseIn

```
[parseTran p97]
[postError p370]
```

— defun parseIn —

```
(defun |parseIn| (arg)
  (let (i n)
    (setq i (|parseTran| (first arg)))
    (setq n (|parseTran| (second arg)))
    (cond
      ((and (consp n) (eq (qfirst n) 'segment)
            (consp (qrest n)) (eq (qcddr n) nil))
       (list 'step i (second n) 1))
      ((and (consp n) (eq (qfirst n) '|reverse|)
            (consp (qrest n)) (eq (qcddr n) nil)
            (consp (qsecond n)) (eq (qcaadr n) 'segment)
            (consp (qcdadr n))
            (eq (qcddadr n) nil))
       (postError (list " You cannot reverse an infinite sequence." )))
      ((and (consp n) (eq (qfirst n) 'segment)
            (consp (qrest n)) (consp (qcddr n))
            (eq (qcdddr n) nil))
       (if (third n)
           (list 'step i (second n) 1 (third n))
           (list 'step i (second n) 1)))
      ((and (consp n) (eq (qfirst n) '|reverse|)
            (consp (qrest n)) (eq (qcddr n) nil)
            (consp (qsecond n)) (eq (qcaadr n) 'segment)
            (consp (qcdadr n))
            (consp (qcddadr n))
            (eq (qrest (qcddadr 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 (consp n) (eq (qfirst n) '|tails|)
```

```

      (cons (qrest n) (eq (qcddr n) nil))
    (list 'on i (second n)))
  (t
    (list 'in i n))))

```

defplist parseInBy

— postvars —

```

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

```

defun parseInBy

```

[postError p370]
[parseTran p97]
[bright p??]
[parseIn p121]

```

— 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 (consp u) (eq (qfirst u) 'step)
                  (consp (qrest u))
                  (consp (qcddr u))
                  (consp (qcdddr 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) (cdddr u)))))))))
```

deflist parseIs

— postvars —

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

defun parseIs

```
[parseTran p97]
[transIs p106]
```

— defun parseIs —

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

deflist parseIsnt

— postvars —

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

defun parseIsnt

```
[parseTran p97]
[transIs p106]
```

— defun parseIsnt —

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

—————

defplist parseJoin

— postvars —

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

—————

defun parseJoin

[parseTranList [p99](#)]

— defun parseJoin —

```
(defun |parseJoin| (thejoin)
  (labels (
    (fn (arg)
      (cond
        ((null arg)
         nil)
        ((and (consp arg) (consp (qfirst arg)) (eq (qcaar arg) '|Join|))
         (append (cdar arg) (fn (rest arg))))
        (t
         (cons (first arg) (fn (rest arg))))))
    )
  (cons '|Join| (fn (|parseTranList| thejoin)))))
```

—————

defplist parseLeave

— postvars —


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

defun parseLeave

[parseTran p97]

— 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)))))
```

defplist parseLessEqual

— postvars —

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

defun parseLessEqual

[parseTran p97]

[\$op p??]

— defun parseLessEqual —

```
(defun |parseLessEqual| (arg)
  (declare (special |$op|))
  (|parseTran| (list '|not| (cons (subst '>' '<=' |$op| :test #'equal) arg)))))
```

defplist parseLET

— postvars —

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

defun parseLET

```
[parseTran p97]
[parseTranCheckForRecord p545]
[opOf p??]
[transIs p106]
```

— 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)))
```

defplist parseLETD

— postvars —

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

defun parseLETD

[parseTran p97]
 [parseType p102]

— **defun parseLETD** —

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

defplist parseMDEF

— **postvars** —

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

defun parseMDEF

[parseTran p97]
 [parseTranList p99]
 [parseTranCheckForRecord p545]
 [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|))))))
```

defplist parseNot

— postvars —

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

defplist parseNot

— postvars —

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

defun parseNot

```
[parseTran p97]
[$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|))))))
```

defplist parseNotEqual

— postvars —

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

—————

defun parseNotEqual

```
[parseTran p97]
[$op p??]
```

— defun parseNotEqual —

```
(defun |parseNotEqual| (arg)
  (declare (special |$op|))
  (|parseTran| (list '|not| (cons (subst '= '^= |$op| :test #'equal) arg))))
```

—————

defplist parseOr

— postvars —

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

—————

defun parseOr

```
[parseTran p97]
[parseTranList p99]
[parseIf p118]
[parseOr p129]
```

— defun parseOr —

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

defplist parsePretend

— postvars —

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

defun parsePretend

[[parseTran](#) p97]
[[parseType](#) p102]

— defun parsePretend —

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

defplist parseReturn

— postvars —

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

—————

defun parseReturn

```
[parseTran p97]
[moan p??]
```

— defun parseReturn —

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

—————

defplist parseSegment

— postvars —

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

—————

defun parseSegment

```
[parseTran p97]
```

— defun parseSegment —

```
(defun |parseSegment| (arg)
  (if (and (consp arg) (consp (qrest arg)) (eq (qcddr arg) nil))
      (if (second arg)
          (list 'segment (|parseTran| (first arg)) (|parseTran| (second arg)))
          (list 'segment (|parseTran| (first arg))))
      (cons 'segment arg)))
```

defplist parseSeq

— postvars —

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

defun parseSeq

```
[postError p370]
[transSeq p??]
[mapInto p??]
[last p??]
```

— defun parseSeq —

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

defplist parseVCONS

— postvars —


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

defun parseVCONS

```
[parseTranList p99]
```

— defun parseVCONS —

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

defplist parseWhere

— postvars —

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

defun parseWhere

```
[mapInto p??]
```

— defun parseWhere —

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

Chapter 6

Compile Transformers

With some specific exceptions most compile transformers are invoked through the property list item “**special**”. When a specific keyword is encountered in a list form the **compExpression** function looks up the keyword on the property list and funcalls the handler function, passing the form, the mode, and the environment.

If a handler for the keyword is not found then the **compForm** function is called to attempt to compile the form.

defun compExpression

```
[getl p??]  
[compForm p602]  
[$insideExpressionIfTrue p??]
```

— defun compExpression —

```
(defun |compExpression| (form mode env)  
  (let (|$insideExpressionIfTrue| fn)  
    (declare (special |$insideExpressionIfTrue|))  
    (setq |$insideExpressionIfTrue| t)  
    (if (and (atom (car form)) (setq fn (getl (car form) 'special)))  
        (funcall fn form mode env)  
        (|compForm| form mode env))))
```

The functions in this section are called through the symbol-plist of the symbol being parsed. In general, each of these functions takes 3 arguments

1. the **form** which is specific to the function

2. the **mode** a ---Join--- , which is a set of categories and domains
3. the **env** which is a list of functions and their modemaps

and the functions return modified versions of the three arguments suitable for further pro-

	DEF	(p140)	compDefine
	add	(p271)	compAdd
	@	(p357)	compAtSign
	CAPSULE	(p274)	compCapsule
	case	(p283)	compCase
	Mapping	(p285)	compCat
	Record	(p285)	compCat
	Union	(p285)	compCat
	CATEGORY	(p286)	compCategory
	::	(p358)	compCoerce
	:	(p290)	compColon
	CONS	(p294)	compCons
	construct	(p295)	compConstruct
	ListCategory	(p297)	compConstructorCategory
	RecordCategory	(p297)	compConstructorCategory
	UnionCategory	(p297)	compConstructorCategory
	VectorCategory	(p297)	compConstructorCategory
	elt	(p306)	compElt
	exit	(p308)	compExit
	has	(p309)	compHas(pred mode \$e)
cessing.	IF	(p311)	compIf
	import	(p318)	compImport
	is	(p318)	compIs
	Join	(p319)	compJoin
	+->	(p321)	compLambda
	leave	(p323)	compLeave
	MDEF	(p323)	compMacro
	pretend	(p324)	compPretend
	QUOTE	(p326)	compQuote
	REDUCE	(p326)	compReduce
	COLLECT	(p329)	compRepeatOrCollect
	REPEAT	(p329)	compRepeatOrCollect
	return	(p331)	compReturn
	SEQ	(p332)	compSeq
	LET	(p335)	compSetq
	SETQ	(p335)	compSetq
	String	(p345)	compString
	SubDomain	(p346)	compSubDomain
	SubsetCategory	(p348)	compSubsetCategory
		(p349)	compSuchthat
	VECTOR	(p349)	compVector
	where	(p350)	compWhere

6.1 Handline Category DEF forms

This is the graph of the functions used for compDefine. The syntax is a graphviz dot file. To generate this graph as a JPEG file, type:

```
tangle v9compDefine.dot bookvol9.pamphlet >v9compdefine.dot
dot -Tjpg v9compdefine.dot >v9compdefine.jpg
```

— v9compDefine.dot —

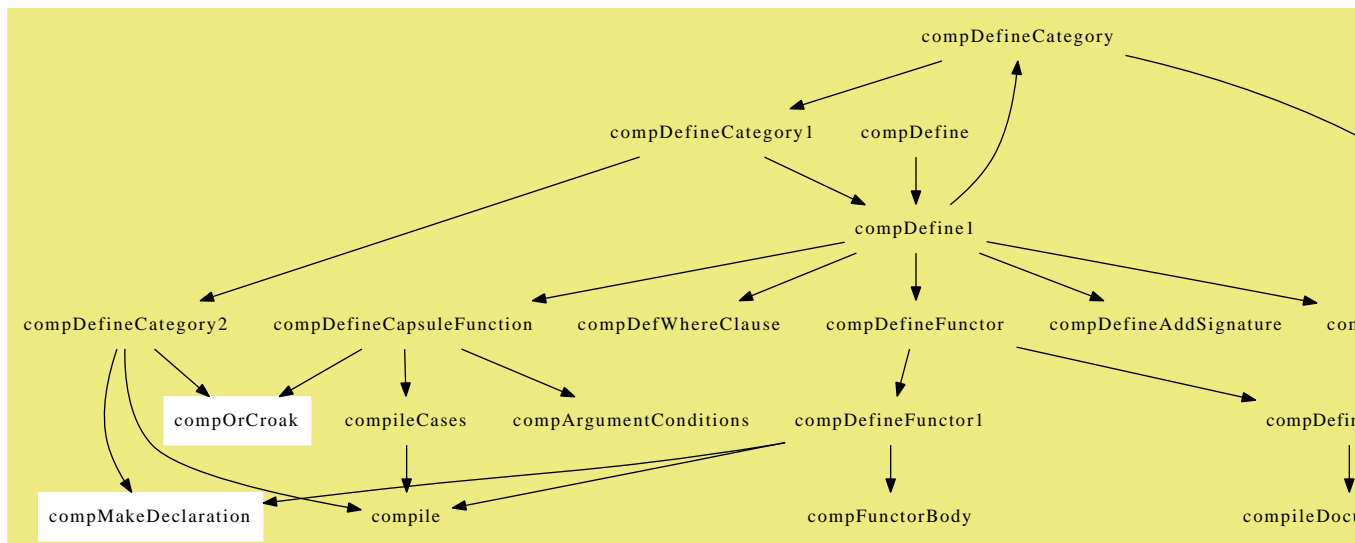
```
digraph pic {
    fontsize=10;
    bgcolor="#ECEA81";
    node [shape=box, color=white, style=filled];

    "compArgumentConditions"      [color="#ECEA81"]
    "compDefWhereClause"          [color="#ECEA81"]
    "compDefine"                  [color="#ECEA81"]
    "compDefine1"                 [color="#ECEA81"]
    "compDefineAddSignature"      [color="#ECEA81"]
    "compDefineCapsuleFunction"   [color="#ECEA81"]
    "compDefineCategory"         [color="#ECEA81"]
    "compDefineCategory1"        [color="#ECEA81"]
    "compDefineCategory2"        [color="#ECEA81"]
    "compDefineFunctor"          [color="#ECEA81"]
    "compDefineFunctor1"         [color="#ECEA81"]
    "compDefineLisplib"          [color="#ECEA81"]
    "compInternalFunction"       [color="#ECEA81"]
    "compMakeDeclaration"        [color="#FFFFFF"]
    "compFunctorBody"            [color="#ECEA81"]
    "compOrCroak"                [color="#FFFFFF"]
    "compile"                    [color="#ECEA81"]
    "compileCases"               [color="#ECEA81"]
    "compileDocumentation"        [color="#ECEA81"]

    "compDefine" -> "compDefine1"
    "compDefine1" -> "compDefineCapsuleFunction"
    "compDefine1" -> "compDefWhereClause"
    "compDefine1" -> "compDefineAddSignature"
    "compDefine1" -> "compDefineCategory"
    "compDefine1" -> "compDefineFunctor"
    "compDefine1" -> "compInternalFunction"
    "compDefineCapsuleFunction" -> "compArgumentConditions"
    "compDefineCapsuleFunction" -> "compOrCroak"
    "compDefineCapsuleFunction" -> "compileCases"
    "compDefineCategory" -> "compDefineCategory1"
    "compDefineCategory" -> "compDefineLisplib"
    "compDefineCategory1" -> "compDefine1"
```

```
"compDefineCategory1" -> "compDefineCategory2"
"compDefineCategory2" -> "compMakeDeclaration"
"compDefineCategory2" -> "compOrCroak"
"compDefineCategory2" -> "compile"
"compDefineFunctor" -> "compDefineFunctor1"
"compDefineFunctor" -> "compDefineLisplib"
"compDefineFunctor1" -> "compMakeDeclaration"
"compDefineFunctor1" -> "compFunctorBody"
"compDefineFunctor1" -> "compile"
"compDefineLisplib" -> "compileDocumentation"
"compileCases" -> "compile"

}
```



A Category is represented by a DEF form with 4 parts:

- a name
- a distnature
- an SC
- a body

For example, the BasicType category is written as

```
BasicType(): Category == with
  "==" (%,% ) -> Boolean    ++ x=y tests if x and y are equal.
  "~==" (%,% ) -> Boolean   ++ x~=y tests if x and y are not equal.
```

```
add
  _~_(x:%,y:%) : Boolean == not(x=y)
```

Which compiles to the DEF form:

```
(DEF
  (|BasicType|)
  ((|Category|))
  (NIL)
  (|add|
    (CATEGORY |domain|
      (SIGNATURE = ((|Boolean|) $ $))
      (SIGNATURE ~ = ((|Boolean|) $ $)))
    (CAPSULE
      (DEF
        (~ = |x| |y|)
        ((|Boolean|) $ $)
        (NIL NIL NIL)
        (IF (= |x| |y|) |false| |true|))))))
```

defplist compDefine plist

We set up the `compDefine` function to handle the DEF keyword by setting the `special` keyword on the DEF symbol property list.

— postvars —

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

—————

defun compDefine

The `compDefine` function expects three arguments:

1. the **form** which is an def specifying the domain to define.
2. the **mode** a —Join—, which is a set of categories and domains
3. the **env** which is a list of functions and their modemaps

```
[compDefine1 p141]
[$tripleCache p??]
[$tripleHits p??]
[$macroIfTrue p??]
[$packagesUsed p??]
```


— defun compDefine —

```
(defun |compDefine| (form mode env)
  (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 mode env)))
```

defun compDefine1

```
[macroExpand p174]
[isMacro p282]
[getSignatureFromMode p299]
[compDefine1 p141]
[compInternalFunction p155]
[compDefineAddSignature p143]
[compDefWhereClause p155]
[compDefineCategory p158]
[isDomainForm p344]
[getTargetFromRhs p173]
[giveFormalParametersValues p174]
[addEmptyCapsuleIfNecessary p173]
[compDefineFunctor p144]
[stackAndThrow p??]
[strconc p??]
[getAbbreviation p298]
[length p??]
[compDefineCapsuleFunction p151]
[$insideExpressionIfTrue p??]
[$formalArgList p??]
[$form p??]
[$op p??]
[$prefix p??]
[$insideFunctorIfTrue p??]
[$Category p??]
[$insideCategoryIfTrue p??]
[$insideCapsuleFunctionIfTrue p??]
[$ConstructorNames p??]
```

`[$NoValueMode p172]`
`[$EmptyMode p172]`
`[$insideWhereIfTrue p??]`
`[$insideExpressionIfTrue p??]`

— defun compDefine1 —

```

(defun |compDefine1| (form mode env)
  (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 env))
    (setq lhs (second form))
    (setq signature (third form))
    (setq specialCases (fourth form))
    (setq rhs (fifth form))
    (cond
      ((and |$insideWhereIfTrue|
            (|isMacro| form env)
            (or (equal mode |$EmptyMode|) (equal mode |$NoValueMode|)))
        (list lhs mode (|put| (car lhs) '|macro| rhs env)))
      ((and (null (car signature)) (consp rhs)
            (null (member (qfirst rhs) |$ConstructorNames|))
            (setq sig (|getSignatureFromMode| lhs env)))
        (|compDefine1|
         (list 'def lhs (cons (car sig) (cdr signature)) specialCases rhs)
         mode env))
      (|$insideCapsuleFunctionIfTrue| (|compInternalFunction| form mode env))
      (t
       (when (equal (car signature) |$Category|) (setq |$insideCategoryIfTrue| t))
       (setq env (|compDefineAddSignature| lhs signature env))
       (cond
         ((null (dolist (x (rest signature) tmp1) (setq tmp1 (and tmp1 (null x)))))
          (|compDefWhereClause| form mode env))
         ((equal (car signature) |$Category|)
          (|compDefineCategory| form mode env nil |$formalArgList|))
         ((and (|isDomainForm| rhs env) (null |$insideFunctorIfTrue|))
          (when (null (car signature))
            (setq signature
              (cons (|getTargetFromRhs| lhs rhs
                    (|giveFormalParametersValues| (cdr lhs) env))
                    (cdr signature))))
          (setq rhs (|addEmptyCapsuleIfNecessary| (car signature) rhs))
          (|compDefineFunctor|

```

```

      (list 'def lhs signature specialCases rhs)
      mode env NIL |$formalArgList|))
((null |$form|)
 (|stackAndThrow| (list "bad == form " form)))
(t
 (setq newPrefix
  (if |$prefix|
    (intern (strconc (|encodeItem| |$prefix|) "," (|encodeItem| |$op|)))
    (|getAbbreviation| |$op| (|#| (cdr |$form|)))))
 (|compDefineCapsuleFunction|
  form mode env newPrefix |$formalArgList|))))))

```

defun compDefineAddSignature

[hasFullSignature p172]
 [assoc p??]
 [lassoc p??]
 [getProplist p??]
 [comp p590]
 [\$EmptyMode p172]

— defun compDefineAddSignature —

```

(defun |compDefineAddSignature| (form signature env)
  (let (sig declForm)
    (declare (special |$EmptyMode|))
    (if
      (and (setq sig (|hasFullSignature| (rest form) signature env))
        (null (|assoc| (cons '$ sig)
          (|lassoc| '|modemap| (|getProplist| (car form) env))))))
      (progn
        (setq declForm
          (list '|:|
            (cons (car form)
              (loop for x in (rest form)
                for m in (rest sig)
                collect (list '|:| x m)))
            (car signature)))
          (third (|comp| declForm |$EmptyMode| env)))
        env)))

```

defun compDefineFunctor

```
[compDefineLisplib p163]
[compDefineFunctor1 p144]
[$domainShell p??]
[$profileCompiler p??]
[$lisplib p??]
[$profileAlist p??]
```

— defun compDefineFunctor —

```
(defun |compDefineFunctor| (df mode env prefix fal)
  (let (|$domainShell| |$profileCompiler| |$profileAlist|)
    (declare (special |$domainShell| |$profileCompiler| $lisplib |$profileAlist|))
    (setq |$domainShell| nil)
    (setq |$profileCompiler| t)
    (setq |$profileAlist| nil)
    (if $lisplib
      (|compDefineLisplib| df mode env prefix fal '|compDefineFunctor1|)
      (|compDefineFunctor1| df mode env prefix fal))))
```

defun compDefineFunctor1

```
[isCategoryPackageName p210]
[getArgumentModeOrMoan p187]
[getModemap p254]
[giveFormalParametersValues p174]
[compMakeCategoryObject p208]
[sayBrightly p??]
[pp p??]
[strconc p??]
[pname p??]
[disallowNilAttribute p214]
[remdup p??]
[NRTgenInitialAttributeAlist p??]
[NRTgetLocalIndex p211]
[compMakeDeclaration p624]
[augModemapsFromCategoryRep p267]
[augModemapsFromCategory p260]
[sublis p??]
[maxindex p??]
[makeFunctorArgumentParameters p217]
[compFunctorBody p168]
```

[reportOnFunctorCompilation p215]
 [compile p169]
 [augmentLisplibModemapsFromFunctor p212]
 [reportOnFunctorCompilation p215]
 [getParentsFor p??]
 [computeAncestorsOf p??]
 [constructor? p??]
 [NRTmakeSlot1Info p??]
 [isCategoryPackageName p210]
 [lisplibWrite p209]
 [mkq p??]
 [getdatabase p??]
 [NRTgetLookupFunction p210]
 [simpBool p??]
 [removeZeroOne p??]
 [evalAndRwriteLispForm p200]
 [\$lisplib p??]
 [\$top-level p??]
 [\$bootStrapMode p??]
 [\$CategoryFrame p??]
 [\$CheckVectorList p??]
 [\$FormalMapVariableList p266]
 [\$LocalDomainAlist p??]
 [\$NRTaddForm p??]
 [\$NRTaddList p??]
 [\$NRTattributeAlist p??]
 [\$NRTbase p??]
 [\$NRTdeltaLength p??]
 [\$NRTdeltaListComp p??]
 [\$NRTdeltaList p??]
 [\$NRTdomainFormList p??]
 [\$NRTloadTimeAlist p??]
 [\$NRTslot1Info p??]
 [\$NRTslot1PredicateList p??]
 [\$Representation p??]
 [\$addForm p??]
 [\$attributesName p??]
 [\$byteAddress p??]
 [\$byteVec p??]
 [\$compileOnlyCertainItems p??]
 [\$condAlist p??]
 [\$domainShell p??]
 [\$form p??]
 [\$functionLocations p??]
 [\$functionStats p??]
 [\$functorForm p??]

```

[$functorLocalParameters p??]
[$functorStats p??]
[$functorSpecialCases p??]
[$functorTarget p??]
[$functorsUsed p??]
[$genFVar p??]
[$genSDVar p??]
[$getDomainCode p??]
[$goGetList p??]
[$insideCategoryPackageIfTrue p??]
[$insideFunctorIfTrue p??]
[$isOpPackageName p??]
[$libFile p??]
[$lisplibAbbreviation p??]
[$lisplibAncestors p??]
[$lisplibCategoriesExtended p??]
[$lisplibCategory p??]
[$lisplibForm p??]
[$lisplibKind p??]
[$lisplibMissingFunctions p??]
[$lisplibModemap p??]
[$lisplibOperationAlist p??]
[$lisplibParents p??]
[$lisplibSlot1 p??]
[$lookupFunction p??]
[$myFunctorBody p??]
[$mutableDomain p??]
[$mutableDomains p??]
[$op p??]
[$pairlis p??]
[$QuickCode p??]
[$setelt p??]
[$signature p??]
[$template p??]
[$uncondAlist p??]
[$viewNames p??]
[$lisplibFunctionLocations p??]

```

— **defun compDefineFunctor1** —

```

(defun |compDefineFunctor1| (df mode |$e| |$prefix| |$formalArgList|)
  (declare (special |$e| |$prefix| |$formalArgList|))
  (labels (
    (FindRep (cb)
      (loop while cb do
        (when (atom cb) (return nil))

```

```

    (when (and (consp cb) (consp (qfirst cb)) (eq (qcaar cb) 'let)
              (consp (qcddar cb)) (eq (qcadar cb) 'Rep|)
              (consp (qcddar cb)))
      (return (caddar cb)))
    (pop cb)))
(let (|$addForm| |$viewNames| |$functionStats| |$functorStats|
      |$form| |$op| |$signature| |$functorTarget|
      |$Representation| |$LocalDomainAlist| |$functorForm|
      |$functorLocalParameters| |$CheckVectorList|
      |$getDomainCode| |$insideFunctorIfTrue| |$functorsUsed|
      |$setelt| $TOP_LEVEL |$genFVar| |$genSDVar|
      |$mutableDomain| |$attributesName| |$goGetList|
      |$condAlist| |$uncondAlist| |$NRTslot1PredicateList|
      |$NRTattributeAlist| |$NRTslot1Info| |$NRTbase|
      |$NRTaddForm| |$NRTdeltaList| |$NRTdeltaListComp|
      |$NRTaddList| |$NRTdeltaLength| |$NRTloadTimeAlist|
      |$NRTdomainFormList| |$template| |$functionLocations|
      |$isOpPackageName| |$lookupFunction| |$byteAddress|
      |$byteVec| form signature body originale argl signaturep target ds
      attributeList parSignature parForm
      argPars opp rettype tt bodyp lamOrSlam fun
      operationAlist modemap libFn tmp1)
(declare (special $lisplib $top_level |$bootStrapMode| |$CategoryFrame|
  |$CheckVectorList| |$FormalMapVariableList| | | | |
  |$LocalDomainAlist| |$NRTaddForm| |$NRTaddList|
  |$NRTattributeAlist| |$NRTbase| |$NRTdeltaLength|
  |$NRTdeltaListComp| |$NRTdeltaList| |$NRTdomainFormList|
  |$NRTloadTimeAlist| |$NRTslot1Info| |$NRTslot1PredicateList|
  |$Representation| |$addForm| |$attributesName|
  |$byteAddress| |$byteVec| |$compileOnlyCertainItems|
  |$condAlist| |$domainShell| |$form| |$functionLocations|
  |$functionStats| |$functorForm| |$functorLocalParameters|
  |$functorStats| |$functorSpecialCases| |$functorTarget|
  |$functorsUsed| |$genFVar| |$genSDVar| |$getDomainCode|
  |$goGetList| |$insideCategoryPackageIfTrue|
  |$insideFunctorIfTrue| |$isOpPackageName| |$libFile|
  |$lisplibAbbreviation| |$lisplibAncestors|
  |$lisplibCategoriesExtended| |$lisplibCategory|
  |$lisplibForm| |$lisplibKind| |$lisplibMissingFunctions|
  |$lisplibModemap| |$lisplibOperationAlist| |$lisplibParents|
  |$lisplibSlot1| |$lookupFunction| |$myFunctorBody|
  |$mutableDomain| |$mutableDomains| |$op| |$pairlis|
  |$QuickCode| |$setelt| |$signature| |$template|
  |$uncondAlist| |$viewNames| |$lisplibFunctionLocations|))
(setq form (second df))
(setq signature (third df))
(setq |$functorSpecialCases| (fourth df))
(setq body (fifth df))
(setq |$addForm| nil)
(setq |$viewNames| nil)

```

```

(setq |$functionStats| (list 0 0))
(setq |$functorStats| (list 0 0))
(setq |$form| nil)
(setq |$op| nil)
(setq |$signature| nil)
(setq |$functorTarget| nil)
(setq |$Representation| nil)
(setq |$LocalDomainAlist| nil)
(setq |$functorForm| nil)
(setq |$functorLocalParameters| nil)
(setq |$myFunctorBody| body)
(setq |$CheckVectorList| nil)
(setq |$getDomainCode| nil)
(setq |$insideFunctorIfTrue| t)
(setq |$functorsUsed| nil)
(setq |$setelt| (if |$QuickCode| 'qsetrefv 'setelt))
(setq $top_level nil)
(setq |$genFVar| 0)
(setq |$genSDVar| 0)
(setq originale |$e|)
(setq |$op| (first form))
(setq argl (rest form))
(setq |$formalArgList| (append argl |$formalArgList|))
(setq |$pairlis|
  (loop for a in argl for v in |$FormalMapVariableList|
    collect (cons a v)))
(setq |$mutableDomain|
  (OR (|isCategoryPackageName| |$op|)
    (COND
      ((boundp '|$mutableDomains|)
       (member |$op| |$mutableDomains|))
      ('T NIL))))

(setq signaturep
  (cons (car signature)
    (loop for a in argl collect (|getArgumentModeOrMoan| a form |$e|))))
(setq |$form| (cons |$op| argl))
(setq |$functorForm| |$form|)
(unless (car signaturep)
  (setq signaturep (cdar (|getModemap| |$form| |$e|))))
(setq target (first signaturep))
(setq |$functorTarget| target)
(setq |$e| (|giveFormalParametersValues| argl |$e|))
(setq tmp1 (|compMakeCategoryObject| target |$e|))
(if tmp1
  (progn
    (setq ds (first tmp1))
    (setq |$e| (third tmp1))
    (setq |$domainShell| (copy-seq ds))
    (setq |$attributesName| (intern (strconc (pname |$op|) ";attributes"))))
  (setq attributeList (|disallowNilAttribute| (elt ds 2)))

```



```

(setq |$goGetList| nil)
(setq |$condAlist| nil)
(setq |$uncondAlist| nil)
(setq |$NRTslot1PredicateList|
  (remdup (loop for x in attributeList collect (second x))))
(setq |$NRTattributeAlist| (|NRTgenInitialAttributeAlist| attributeList))
(setq |$NRTslot1Info| nil)
(setq |$NRTbase| 6)
(setq |$NRTaddForm| nil)
(setq |$NRTdeltaList| nil)
(setq |$NRTdeltaListComp| nil)
(setq |$NRTaddList| nil)
(setq |$NRTdeltaLength| 0)
(setq |$NRTloadTimeAlist| nil)
(setq |$NRTdomainFormList| nil)
(setq |$template| nil)
(setq |$functionLocations| nil)
(loop for x in argl do (|NRTgetLocalIndex| x))
(setq |$e|
  (third (|compMakeDeclaration| (list '||:| '$ target) mode |$e|)))
(unless |$insideCategoryPackageIfTrue|
  (if
    (and (consp body) (eq (qfirst body) '|add|)
      (consp (qrest body))
      (consp (qsecond body))
      (consp (qcddr body))
      (eq (qcdddr body) nil)
      (consp (qthird body))
      (eq (qcaaddr body) '|capsule|)
      (member (qcaadr body) '(|List| |Vector|))
      (equal (FindRep (qcdaddr body)) (second body)))
    (setq |$e| (|augModemapsFromCategoryRep| '$
      (second body) (cdaddr body) target |$e|))
    (setq |$e| (|augModemapsFromCategory| '$ '$ target |$e|))))
(setq |$signature| signaturep)
(setq operationAlist (sublis |$pairlis| (elt |$domainShell| 1)))
(setq parSignature (sublis |$pairlis| signaturep))
(setq parForm (sublis |$pairlis| form))
(setq argPars (|makeFunctorArgumentParameters| argl
  (cdr signaturep) (car signaturep)))
(setq |$functorLocalParameters| argl)
(setq opp |$op|)
(setq rettype (CAR signaturep))
(setq tt (|compFunctorBody| body rettype |$e| parForm))
(cond
  (|$compileOnlyCertainItems|
    (|reportOnFunctorCompilation|
      (list nil (cons '|Mapping| signaturep) originale)))
  (t
    (setq bodyp (first tt))

```

```

(setq lamOrSlam (if |$mutableDomain| 'lam 'spadslam))
(setq fun
  (|compile| (sublis |$pairlis| (list opp (list lamOrSlam arg1 body))))))
(setq operationAlist (sublis |$pairlis| |$lisplibOperationAlist|))
(cond
  ($lisplib
    (|augmentLisplibModemapsFromFunctor| parForm
      operationAlist parSignature)))
(|reportOnFunctorCompilation|)
(cond
  ($lisplib
    (setq modemap (list (cons parForm parSignature) (list t opp)))
    (setq |$lisplibModemap| modemap)
    (setq |$lisplibCategory| (cadar modemap))
    (setq |$lisplibParents|
      (|getParentsFor| |$op| |$FormalMapVariableList| |$lisplibCategory|))
    (setq |$lisplibAncestors| (|computeAncestorsOf| |$form| NIL))
    (setq |$lisplibAbbreviation| (|constructor?| |$op|)))
    (setq |$insideFunctorIfTrue| NIL)
  (cond
    ($lisplib
      (setq |$lisplibKind|
        (if (and (consp |$functorTarget|)
          (eq (qfirst |$functorTarget|) 'category)
          (consp (qrest |$functorTarget|))
          (not (eq (qsecond |$functorTarget|) 'domain))))
        '|package|
        '|domain|))
      (setq |$lisplibForm| form)
      (cond
        ((null |$bootStrapMode|)
          (setq |$NRTslot1Info| (|NRTmakeSlot1Info|))
          (setq |$isOpPackageName| (|isCategoryPackageName| |$op|))
          (when |$isOpPackageName|
            (|lisplibWrite| "slot1DataBase"
              (list '|updateSlot1DataBase| (mkq |$NRTslot1Info|))
              |$libFile|))
            (setq |$lisplibFunctionLocations|
              (sublis |$pairlis| |$functionLocations|))
            (setq |$lisplibCategoriesExtended|
              (sublis |$pairlis| |$lisplibCategoriesExtended|))
            (setq libFn (getdatabase opp 'abbreviation))
            (setq |$lookupFunction|
              (|NRTgetLookupFunction| |$functorForm|
                (cadar |$lisplibModemap|) |$NRTaddForm|))
            (setq |$byteAddress| 0)
            (setq |$byteVec| NIL)
            (setq |$NRTslot1PredicateList|
              (loop for x in |$NRTslot1PredicateList|
                collect (|simpBool| x)))

```

```

(|rewriteLispForm| 'loadTimeStuff|
  '(setf (get ,(mkq |$op|) '|infovec|) ,(|getInfovecCode|))))
(setq |$lisplibSlot1| |$NRTslot1Info|)
(setq |$lisplibOperationAlist| operationAlist)
(setq |$lisplibMissingFunctions| |$CheckVectorList|))
(|lisplibWrite| "compilerInfo"
  (|removeZeroOne|
    (list 'setq '|$CategoryFrame|
      (list '|put| (list 'quote opp) ''|isFunctor|
        (list 'quote operationAlist)
        (list '|addModemap|
          (list 'quote opp)
          (list 'quote parForm)
          (list 'quote parSignature)
          t
          (list 'quote opp)
          (list '|put| (list 'quote opp) ''|mode|
            (list 'quote (cons '|Mapping| parSignature))
            '|$CategoryFrame|))))
        |$libFile|)
      (unless arg1
        (|evalAndRewriteLispForm| 'niladic
          '(setf (get ',opp 'niladic) t)))
      (list fun (cons '|Mapping| signaturep) originale))))
  (progn
    (|sayBrightly| "    cannot produce category object:")
    (|pp| target)
    nil))))

```

defun compDefineCapsuleFunction

```

[length p??]
[get p??]
[profileRecord p??]
[compArgumentConditions p166]
[addDomain p248]
[giveFormalParametersValues p174]
[getSignature p302]
[put p??]
[getArgumentModeOrMoan p187]
[checkAndDeclare p304]
[hasSigInTargetCategory p304]
[stripOffSubdomainConditions p301]
[stripOffArgumentConditions p302]
[resolve p361]

```

```

[member p??]
[getmode p??]
[formatUnabbreviated p??]
[sayBrightly p??]
[compOrCroak p588]
[NRTassignCapsuleFunctionSlot p??]
[mkq p??]
[replaceExitEtc p333]
[addArgumentConditions p300]
[compileCases p167]
[addStats p??]
[$semanticErrorStack p??]
[$DomainsInScope p??]
[$op p??]
[$formalArgList p??]
[$signatureOfForm p??]
[$functionLocations p??]
[$profileCompiler p??]
[$compileOnlyCertainItems p??]
[$returnMode p??]
[$functorStats p??]
[$functionStats p??]
[$form p??]
[$functionStats p??]
[$argumentConditionList p??]
[$finalEnv p??]
[$initCapsuleErrorCount p??]
[$insideCapsuleFunctionIfTrue p??]
[$CapsuleModemapFrame p??]
[$CapsuleDomainsInScope p??]
[$insideExpressionIfTrue p??]
[$returnMode p??]
[$op p??]
[$formalArgList p??]
[$signatureOfForm p??]
[$functionLocations p??]

```

— **defun compDefineCapsuleFunction** —

```

(defun |compDefineCapsuleFunction| (df m oldE |$prefix| |$formalArgList|)
  (declare (special |$prefix| |$formalArgList|))
  (let (|$form| |$op| |$functionStats| |$argumentConditionList| |$finalEnv|
        |$initCapsuleErrorCount| |$insideCapsuleFunctionIfTrue|
        |$CapsuleModemapFrame| |$CapsuleDomainsInScope|
        |$insideExpressionIfTrue| form signature body tmp1 lineNumber
        specialCases argl identSig argModeList signaturep e rettype tmp2

```

```

    localOrExported formattedSig tt catchTag bodyp finalBody fun val)
(declare (special |$form| |$op| |$functionStats| |$functorStats|
    |$argumentConditionList| |$finalEnv| |$returnMode|
    |$initCapsuleErrorCount| |$newCompCompare| |$NoValueMode|
    |$insideCapsuleFunctionIfTrue|
    |$CapsuleModemapFrame| |$CapsuleDomainsInScope| | |
    |$insideExpressionIfTrue| |$compileOnlyCertainItems|
    |$profileCompiler| |$functionLocations| |$finalEnv|
    |$signatureOfForm| |$semanticErrorStack|))
(setq form (second df))
(setq signature (third df))
(setq specialCases (fourth df))
(setq body (fifth df))
(setq tmp1 specialCases)
(setq lineNumber (first tmp1))
(setq specialCases (rest tmp1))
(setq e oldE)
;-1. bind global variables
(setq |$form| nil)
(setq |$op| nil)
(setq |$functionStats| (list 0 0))
(setq |$argumentConditionList| nil)
(setq |$finalEnv| nil)
; used by ReplaceExitEtc to get a common environment
(setq |$initCapsuleErrorCount| (|#| |$semanticErrorStack|))
(setq |$insideCapsuleFunctionIfTrue| t)
(setq |$CapsuleModemapFrame| e)
(setq |$CapsuleDomainsInScope| (|get| '|$DomainsInScope| 'special e))
(setq |$insideExpressionIfTrue| t)
(setq |$returnMode| m)
(setq |$op| (first form))
(setq argl (rest form))
(setq |$form| (cons |$op| argl))
(setq argl (|stripOffArgumentConditions| argl))
(setq |$formalArgList| (append argl |$formalArgList|))
; let target and local signatures help determine modes of arguments
(setq argModeList
  (cond
    ((setq identSig (|hasSigInTargetCategory| argl form (car signature) e))
      (setq e (|checkAndDeclare| argl form identSig e))
      (cdr identSig))
    (t
      (loop for a in argl
        collect (|getArgumentModeOrMoan| a form e))))))
(setq argModeList (|stripOffSubdomainConditions| argModeList argl))
(setq signaturep (cons (car signature) argModeList))
(unless identSig
  (setq oldE (|put| |$op| '|mode| (cons '|Mapping| signaturep) oldE)))
; obtain target type if not given
(cond

```

```

((null (car signaturep))
 (setq signaturep
  (cond
   (identSig identSig)
   (t (|getSignature| |$op| (cdr signaturep) e))))))
(when signaturep
 (setq e (|giveFormalParametersValues| argl e))
 (setq |$signatureOfForm| signaturep)
 (setq |$functionLocations|
  (cons (cons (list |$op| |$signatureOfForm|) lineNumber)
   |$functionLocations|))
 (setq e (|addDomain| (car signaturep) e))
 (setq e (|compArgumentConditions| e))
 (when |$profileCompiler|
  (loop for x in argl for y in signaturep
   do (|profileRecord| 'arguments| x y)))
; 4. introduce needed domains into extendedEnv
(loop for domain in signaturep
 do (setq e (|addDomain| domain e)))
; 6. compile body in environment with extended environment
(setq rettype (|resolve| (car signaturep) |$returnMode|))
(setq localOrExported
 (cond
  ((and (null (|member| |$op| |$formalArgList|))
   (progn
    (setq tmp2 (|getmode| |$op| e))
    (and (consp tmp2) (eq (qfirst tmp2) '|Mapping|))))
   '|local|)
  (t '|exported|)))
; 6a skip if compiling only certain items but not this one
; could be moved closer to the top
(setq formattedSig (|formatUnabbreviated| (cons '|Mapping| signaturep)))
(cond
 ((and |$compileOnlyCertainItems|
  (null (|member| |$op| |$compileOnlyCertainItems|)))
  (|sayBrightly|
   (cons " skipping " (cons localOrExported (|bright| |$op|))))
  (list nil (cons '|Mapping| signaturep) oldE))
 (t
  (|sayBrightly|
   (cons " compiling " (cons localOrExported (append (|bright| |$op|)
    (cons ": " formattedSig)))))
  (setq tt (catch '|compCapsuleBody| (|compOrCroak| body rettype e)))
  (|NRTassignCapsuleFunctionSlot| |$op| signaturep)
  ; A THROW to the above CATCH occurs if too many semantic errors occur
  ; see stackSemanticError
  (setq catchTag (mkq (gensym)))
  (setq fun
   (progn
    (setq bodyp

```

```

      (|replaceExitEtc| (car tt) catchTag '|TAGGEDreturn| |$returnMode|))
      (setq bodyp (|addArgumentConditions| bodyp |$op|))
      (setq finalBody (list 'catch catchTag bodyp))
      (|compileCases|
        (list |$op| (list 'lam (append argl (list '$)) finalBody))
        oldE)))
      (setq |$functorStats| (|addStats| |$functorStats| |$functionStats|))
; 7. give operator a 'value property
      (setq val (list fun signaturep e))
      (list fun (list '|Mapping| signaturep) oldE))))))

```

defun compInternalFunction

```

[identp p??]
[stackAndThrow p??]

```

— defun compInternalFunction —

```

(defun |compInternalFunction| (df m env)
  (let (form signature specialCases body op argl nbody nf ress)
    (setq form (second df))
    (setq signature (third df))
    (setq specialCases (fourth df))
    (setq body (fifth df))
    (setq op (first form))
    (setq argl (rest form))
    (cond
      ((null (identp op))
        (|stackAndThrow| (list '|Bad name for internal function:| op)))
      ((eql (|#| argl) 0)
        (|stackAndThrow|
          (list '|Argumentless internal functions unsupported:| op )))
      (t
        (setq nbody (list '+-> argl body))
        (setq nf (list 'let (list '|:| op (cons '|Mapping| signature)) nbody))
        (setq ress (|comp| nf m env) ress))))))

```

defun compDefWhereClause

```

[getmode p??]
[userError p??]

```

```

[concat p??]
[lassoc p??]
[pairList p??]
[union p??]
[listOfIdentifiersIn p??]
[delete p??]
[orderByDependency p222]
[assocleft p??]
[assocright p??]
[comp p590]
[$sigAlist p??]
[$predAlist p??]

```

— defun compDefWhereClause —

```

(defun |compDefWhereClause| (arg mode env)
  (labels (
    (transformType (x)
      (declare (special |$sigAlist|))
      (cond
        ((atom x) x)
        ((and (consp x) (eq (qfirst x) '|:|) (consp (qrest x))
          (consp (qcddr x)) (eq (qcdddr x) nil))
         (setq |$sigAlist|
           (cons (cons (second x) (transformType (third x)))
                 |$sigAlist|))
         x)
        ((and (consp x) (eq (qfirst x) '|Record|)) x)
        (t
         (cons (first x)
               (loop for y in (rest x)
                     collect (transformType y))))))
    (removeSuchthat (x)
      (declare (special |$predAlist|))
      (if (and (consp x) (eq (qfirst x) '|\\|') (consp (qrest x))
        (consp (qcddr x)) (eq (qcdddr x) nil))
        (progn
          (setq |$predAlist| (cons (cons (second x) (third x)) |$predAlist|))
          (second x))
        x))
    (fetchType (a x env form)
      (if x
        x
        (or (|getmode| a env)
          (|userError| (|concat|
            "There is no mode for argument" a "of function" (first form))))))
    (addSuchthat (x y)
      (let (p)
        (declare (special |$predAlist|))

```



```

    (if (setq p (lassoc x |$predAlist|)) (list '|\\| y p) y)))
  )
  (let (|$sigAlist| |$predAlist| form signature specialCases body sigList
        argList argSigAlist argDepAlist varList whereList formxx signaturex
        deform formx)
    (declare (special |$sigAlist| |$predAlist|))
    ; form is lhs (f a1 ... an) of definition; body is rhs;
    ; signature is (t0 t1 ... tn) where t0= target type, ti=type of ai, i > 0;
    ; specialCases is (NIL l1 ... ln) where li is list of special cases
    ; which can be given for each ti
    ;
    ; removes declarative and assignment information from form and
    ; signature, placing it in list L, replacing form by ("where",form',:L),
    ; signature by a list of NILs (signifying declarations are in e)
    (setq form (second arg))
    (setq signature (third arg))
    (setq specialCases (fourth arg))
    (setq body (fifth arg))
    (setq |$sigAlist| nil)
    (setq |$predAlist| nil)
    ; 1. create sigList= list of all signatures which have embedded
    ;   declarations moved into global variable $sigAlist
    (setq sigList
      (loop for a in (rest form) for x in (rest signature)
        collect (transformType (fetchType a x env form))))
    ; 2. replace each argument of the form (| x p) by x, recording
    ;   the given predicate in global variable $predAlist
    (setq argList
      (loop for a in (rest form)
        collect (removeSuchthat a)))
    (setq argSigAlist (append |$sigAlist| (|pairList| argList sigList)))
    (setq argDepAlist
      (loop for pear in argSigAlist
        collect
          (cons (car pear)
                (|union| (|listOfIdentifiersIn| (cdr pear))
                        (|delete| (car pear)
                                (|listOfIdentifiersIn| (lassoc (car pear) |$predAlist|)))))))
    ; 3. obtain a list of parameter identifiers (x1 .. xn) ordered so that
    ;   the type of xi is independent of xj if i < j
    (setq varList
      (|orderByDependency| (assocleft argDepAlist) (assocright argDepAlist)))
    ; 4. construct a WhereList which declares and/or defines the xi's in
    ;   the order constructed in step 3
    (setq whereList
      (loop for x in varList
        collect (addSuchthat x (list '|:| x (lassoc x argSigAlist)))))
    (setq formxx (cons (car form) argList))
    (setq signaturex
      (cons (car signature)
            (|listOfIdentifiersIn| (lassoc (car signature) |$predAlist|))))
  )

```

```

      (loop for x in (rest signature) collect nil)))
      (setq deform (list 'def formxx signaturex specialCases body))
      (setq formx (cons '|where| (cons deform whereList)))
; 5. compile new ('DEF,("where",form',:WhereList),..) where
;   all argument parameters of form' are bound/declared in WhereList
      (|comp| formx mode env)))

```

defun compDefineCategory

```

[compDefineLisplib p163]
[compDefineCategory1 p158]
[$domainShell p??]
[$lisplibCategory p??]
[$lisplib p??]
[$insideFunctorIfTrue p??]

```

— defun compDefineCategory —

```

(defun |compDefineCategory| (df mode env prefix fal)
  (let (|$domainShell| |$lisplibCategory|)
    (declare (special |$domainShell| |$lisplibCategory| $lisplib
                      |$insideFunctorIfTrue|))
    (setq |$domainShell| nil) ; holds the category of the object being compiled
    (setq |$lisplibCategory| nil)
    (if (and (null |$insideFunctorIfTrue|) $lisplib)
        (|compDefineLisplib| df mode env prefix fal '|compDefineCategory1|)
        (|compDefineCategory1| df mode env prefix fal))))

```

defun compDefineCategory1

```

[compDefineCategory2 p159]
[makeCategoryPredicates p175]
[compDefine1 p141]
[mkCategoryPackage p176]
[$insideCategoryPackageIfTrue p??]
[$EmptyMode p172]
[$categoryPredicateList p??]
[$lisplibCategory p??]
[$bootStrapMode p??]

```

— defun compDefineCategory1 —

```

(defun |compDefineCategory1| (df mode env prefix fal)
  (let (|$insideCategoryPackageIfTrue| |$categoryPredicateList| form
        sig sc cat body categoryCapsule d tmp1 tmp3)
    (declare (special |$insideCategoryPackageIfTrue| |$EmptyMode|
                      |$categoryPredicateList| |$lisplibCategory|
                      |$bootStrapMode|))
    ;; a category is a DEF form with 4 parts:
    ;; ((DEF (|BasicType|) ((|Category|)) (NIL)
    ;;      (|add| (CATEGORY |domain| (SIGNATURE = ((|Boolean|) $ $))
    ;;              (SIGNATURE ~= ((|Boolean|) $ $)))
    ;;      (CAPSULE (DEF (~= |x| |y|) ((|Boolean|) $ $) (NIL NIL NIL)
    ;;                (IF (= |x| |y|) |false| |true|)))))
    (setq form (second df))
    (setq sig (third df))
    (setq sc (fourth df))
    (setq body (fifth df))
    (setq categoryCapsule
      (when (and (consp body) (eq (qfirst body) '|add|)
                (consp (qrest body)) (consp (qcddr body))
                (eq (qcddr body) nil))
        (setq tmp1 (third body))
        (setq body (second body))
        tmp1))
    (setq tmp3 (|compDefineCategory2| form sig sc body mode env prefix fal))
    (setq d (first tmp3))
    (setq mode (second tmp3))
    (setq env (third tmp3))
    (when (and categoryCapsule (null |$bootStrapMode|))
      (setq |$insideCategoryPackageIfTrue| t)
      (setq |$categoryPredicateList|
        (|makeCategoryPredicates| form |$lisplibCategory|))
      (setq env (third
        (|compDefine1|
          (|mkCategoryPackage| form cat categoryCapsule) |$EmptyMode| env))))
    (list d mode env)))

```

defun compDefineCategory2

```

[addBinding p??]
[getArgumentModeOrMoan p187]
[giveFormalParametersValues p174]
[take p??]
[sublis p??]
[compMakeDeclaration p624]
[opOf p??]

```

```

[optFunctorBody p??]
[compOrCroak p588]
[mkConstructor p201]
[compile p169]
[lisplibWrite p209]
[removeZeroOne p??]
[mkq p??]
[evalAndRwriteLispForm p200]
[eval p??]
[getParentsFor p??]
[computeAncestorsOf p??]
[constructor? p??]
[augLisplibModemapsFromCategory p187]
[$prefix p??]
[$formalArgList p??]
[$definition p??]
[$form p??]
[$op p??]
[$extraParms p??]
[$lisplibCategory p??]
[$FormalMapVariableList p266]
[$libFile p??]
[$TriangleVariableList p??]
[$lisplib p??]
[$formalArgList p??]
[$insideCategoryIfTrue p??]
[$top-level p??]
[$definition p??]
[$form p??]
[$op p??]
[$extraParms p??]
[$functionStats p??]
[$functorStats p??]
[$frontier p??]
[$getDomainCode p??]
[$addForm p??]
[$lisplibAbbreviation p??]
[$functorForm p??]
[$lisplibAncestors p??]
[$lisplibCategory p??]
[$lisplibParents p??]
[$lisplibModemap p??]
[$lisplibKind p??]
[$lisplibForm p??]
[$domainShell p??]

```

— defun compDefineCategory2 —

```

(defun |compDefineCategory2|
  (form signature specialCases body mode env |$prefix| |$formalArgList|)
  (declare (special |$prefix| |$formalArgList|) (ignore specialCases))
  (let (|$insideCategoryIfTrue| $TOP_LEVEL |$definition| |$form| |$op|
        |$extraParms| |$functionStats| |$functorStats| |$frontier|
        |$getDomainCode| |$addForm| argl sargl aList signaturep opp formp
        formalBody formals actuals g fun pairlis parSignature parForm modemap)
    (declare (special |$insideCategoryIfTrue| $top_level |$definition|
                      |$form| |$op| |$extraParms| |$functionStats|
                      |$functorStats| |$frontier| |$getDomainCode|
                      |$addForm| |$lisplibAbbreviation| |$functorForm|
                      |$lisplibAncestors| |$lisplibCategory|
                      |$FormalMapVariableList| |$lisplibParents|
                      |$lisplibModemap| |$lisplibKind| |$lisplibForm|
                      $lisplib |$domainShell| |$libFile|
                      |$TriangleVariableList|))
      ; 1. bind global variables
      (setq |$insideCategoryIfTrue| t)
      (setq $top_level nil)
      (setq |$definition| nil)
      (setq |$form| nil)
      (setq |$op| nil)
      (setq |$extraParms| nil)
      ; 1.1 augment e to add declaration $: <form>
      (setq |$definition| form)
      (setq |$op| (car |$definition|))
      (setq argl (cdr |$definition|))
      (setq env (|addBinding| '$ (list (cons '|mode| |$definition|)) env))
      ; 2. obtain signature
      (setq signaturep
        (cons (car signature)
              (loop for a in argl
                    collect (|getArgumentModeOrMoan| a |$definition| env))))
      (setq env (|giveFormalParametersValues| argl env))
      ; 3. replace arguments by $1,..., substitute into body,
      ;    and introduce declarations into environment
      (setq sargl (take (|#| argl) |$TriangleVariableList|))
      (setq |$form| (cons |$op| sargl))
      (setq |$functorForm| |$form|)
      (setq |$formalArgList| (append sargl |$formalArgList|))
      (setq aList (loop for a in argl for sa in sargl collect (cons a sa)))
      (setq formalBody (sublis aList body))
      (setq signaturep (sublis aList signaturep))
      ; Begin lines for category default definitions
      (setq |$functionStats| (list 0 0))
      (setq |$functorStats| (list 0 0))
      (setq |$frontier| 0)
      (setq |$getDomainCode| nil)

```

```

(setq |$addForm| nil)
(loop for x in sargl for r in (rest signaturep)
  do (setq env (third (|compMakeDeclaration| (list '|:| x r) mode env))))
; 4. compile body in environment of %type declarations for arguments
(setq opp |$op|)
(when (and (not (eq (|opOf| formalBody) '|Join|))
  (not (eq (|opOf| formalBody) '|mkCategory|))))
  (setq formalBody (list '|Join| formalBody)))
(setq body
  (|optFunctorBody| (car (|compOrCroak| formalBody (car signaturep) env))))
(when |$extraParms|
  (setq actuals nil)
  (setq formals nil)
  (loop for u in |$extraParms| do
    (setq formals (cons (car u) formals))
    (setq actuals (cons (mkq (cdr u)) actuals)))
  (setq body
    (list '|sublisV| (list 'pair (list 'quote formals) (cons 'list actuals))
      body)))
; always subst for args after extraparms
(when argl
  (setq body
    (list '|sublisV|
      (list 'pair
        (list 'quote sargl)
        (cons 'list (loop for u in sargl collect (list '|devalueate| u))))
      body)))
(setq body
  (list 'prog1 (list 'let (setq g (gensym)) body)
    (list 'setelt g 0 (|mkConstructor| |$form|))))
(setq fun (|compile| (list opp (list 'lam sargl body))))
; 5. give operator a 'modemap property
(setq pairlis
  (loop for a in argl for v in |$FormalMapVariableList|
    collect (cons a v)))
(setq parSignature (sublis pairlis signaturep))
(setq parForm (sublis pairlis form))
(|lisplibWrite| "compilerInfo"
  (|removeZeroOne|
    (list 'setq '|$CategoryFrame|
      (list '|put| (list 'quote opp) '|isCategory| t
        (list '|addModemap| (mkq opp) (mkq parForm)
          (mkq parSignature) t (mkq fun) '|$CategoryFrame|))))
    |$libFile|)
(unless sargl
  (|evalAndRwriteLispForm| 'niladic
    '(setf (get ',opp 'niladic) t)))
;; 6 put modemaps into InteractiveModemapFrame
(setq |$domainShell| (|eval| (cons opp (mapcar 'mkq sargl))))
(setq |$lisplibCategory| formalBody)

```

```

(when $lisplib
  (setq |$lisplibForm| form)
  (setq |$lisplibKind| '|category|)
  (setq modemap (list (cons parForm parSignature) (list t opp)))
  (setq |$lisplibModemap| modemap)
  (setq |$lisplibParents|
    (|getParentsFor| |$op| |$FormalMapVariableList| |$lisplibCategory|))
  (setq |$lisplibAncestors| (|computeAncestorsOf| |$form| nil))
  (setq |$lisplibAbbreviation| (|constructor?| |$op|))
  (setq formp (cons opp sargl))
  (|augLisplibModemapsFromCategory| formp formalBody signaturep))
(list fun '(|Category|) env)))

```

defun compDefineLisplib

```

[sayMSG p??]
[fillerSpaces p??]
[getConstructorAbbreviation p??]
[compileDocumentation p165]
[bright p??]
[finalizeLisplib p203]
[rshut p??]
[lisplibDoRename p201]
[filep p??]
[rpackfile p??]
[unloadOneConstructor p201]
[localdatabase p??]
[getdatabase p??]
[updateCategoryFrameForCategory p117]
[updateCategoryFrameForConstructor p116]
[$compileDocumentation p165]
[$filep p??]
[$spadLibFT p??]
[$algebraOutputStream p??]
[$newConlist p557]
[$lisplibKind p??]
[$lisplib p??]
[$op p??]
[$lisplibParents p??]
[$lisplibPredicates p??]
[$lisplibCategoriesExtended p??]
[$lisplibForm p??]
[$lisplibKind p??]

```

```

[$lisplibAbbreviation p??]
[$lisplibAncestors p??]
[$lisplibModemap p??]
[$lisplibModemapAlist p??]
[$lisplibSlot1 p??]
[$lisplibOperationAlist p??]
[$lisplibSuperDomain p??]
[$libFile p??]
[$lisplibVariableAlist p??]
[$lisplibCategory p??]
[$newConlist p557]

```

— defun compDefineLisplib —

```

(defun |compDefineLisplib| (df m env prefix fal fn)
  (let ($LISPLIB |$op| |$lisplibAttributes| |$lisplibPredicates|
        |$lisplibCategoriesExtended| |$lisplibForm| |$lisplibKind|
        |$lisplibAbbreviation| |$lisplibParents| |$lisplibAncestors|
        |$lisplibModemap| |$lisplibModemapAlist| |$lisplibSlot1|
        |$lisplibOperationAlist| |$lisplibSuperDomain| |$libFile|
        |$lisplibVariableAlist| |$lisplibCategory| op libname res ok filearg)
    (declare (special $lisplib |$op| |$lisplibAttributes| |$newConlist|
                      |$lisplibPredicates| |$lisplibCategoriesExtended| | |
                      |$lisplibForm| |$lisplibKind| |$algebraOutputStream|
                      |$lisplibAbbreviation| |$lisplibParents| |$spadLibFT|
                      |$lisplibAncestors| |$lisplibModemap| $filep
                      |$lisplibModemapAlist| |$lisplibSlot1|
                      |$lisplibOperationAlist| |$lisplibSuperDomain|
                      |$libFile| |$lisplibVariableAlist|
                      |$lisplibCategory| |$compileDocumentation|))
      (when (eq (car df) 'def) (car df))
      (setq op (caadr df))
      (|sayMSG| (|fillerSpaces| 72 "-"))
      (setq $lisplib t)
      (setq |$op| op)
      (setq |$lisplibAttributes| nil)
      (setq |$lisplibPredicates| nil)
      (setq |$lisplibCategoriesExtended| nil)
      (setq |$lisplibForm| nil)
      (setq |$lisplibKind| nil)
      (setq |$lisplibAbbreviation| nil)
      (setq |$lisplibParents| nil)
      (setq |$lisplibAncestors| nil)
      (setq |$lisplibModemap| nil)
      (setq |$lisplibModemapAlist| nil)
      (setq |$lisplibSlot1| nil)
      (setq |$lisplibOperationAlist| nil)
      (setq |$lisplibSuperDomain| nil)
      (setq |$libFile| nil)

```



```

(setq |$lisplibVariableAlist| nil)
(setq |$lisplibCategory| nil)
(setq libname (|getConstructorAbbreviation| op))
(cond
  ((and (boundp '|$compileDocumentation|) |$compileDocumentation|)
    (|compileDocumentation| libname))
  (t
    (|sayMSG| (cons "    initializing " (cons |$spadLibFT|
      (append (|bright| libname) (cons "for" (|bright| op))))))
    (|initializeLisplib| libname)
    (|sayMSG|
      (cons "    compiling into " (cons |$spadLibFT| (|bright| libname))))
    (setq ok nil)
    (unwind-protect
      (progn
        (setq res (funcall fn df m env prefix fal))
        (|sayMSG| (cons "    finalizing " (cons |$spadLibFT| (|bright| libname))))
        (|finalizeLisplib| libname)
        (setq ok t))
      (rshut |$libFile|))
    (when ok (|lisplibDoRename| libname))
    (setq filearg ($filep libname |$spadLibFT| 'a))
    (rpackfile filearg)
    (fresh-line |$algebraOutputStream|)
    (|sayMSG| (|fillerSpaces| 72 "-"))
    (|unloadOneConstructor| op libname)
    (localdatabase (list (getdatabase op 'abbreviation)) nil)
    (setq |$newConlist| (cons op |$newConlist|))
    (when (eq |$lisplibKind| '|category|)
      (|updateCategoryFrameForCategory| op)
      (|updateCategoryFrameForConstructor| op))
    res))))

```

defun compileDocumentation

```

[makeInputFilename p??]
[rdefiostream p??]
[lisplibWrite p209]
[finalizeDocumentation p492]
[rshut p??]
[rpackfile p??]
[replaceFile p??]
[$fcopy p??]
[$spadLibFT p??]
[$EmptyMode p172]

```

`[$\$e$ p??]`

— **defun compileDocumentation** —

```
(defun |compileDocumentation| (libName)
  (let (filename stream)
    (declare (special | $\$e$ | |$EmptyMode| |$spadLibFT| $fcopy))
    (setq filename (makeInputFilename libName |$spadLibFT|))
    ($fcopy filename (cons libname (list 'doclb)))
    (setq stream
      (rdefiostream (cons (list 'file libName 'doclb) (list (cons 'mode 'o)))))
    (|lisplibWrite| "documentation" (|finalizeDocumentation|) stream)
    (rshut stream)
    (rpackfile (list libName 'doclb))
    (replaceFile (list libName |$spadLibFT|) (list libName 'doclb))
    (list 'dummy| |$EmptyMode| | $\$e$ |)))
```

defun compArgumentConditions

`[compOrCroak p588]`
`[$\$$ Boolean p??]`
`[$\$$ argumentConditionList p??]`
`[$\$$ argumentConditionList p??]`

— **defun compArgumentConditions** —

```
(defun |compArgumentConditions| (env)
  (let (n a x tmp1)
    (declare (special |$Boolean| |$argumentConditionList|))
    (setq |$argumentConditionList|
      (loop for item in |$argumentConditionList|
        do
          (setq n (first item))
          (setq a (second item))
          (setq x (third item))
          (setq y (subst a '|#1| x :test #'equal))
          (setq tmp1 (|compOrCroak| y |$Boolean| env))
          (setq env (third tmp1))
      collect
        (list n x (first tmp1))))
    env))
```

defun compileCases

```
[eval p??]
[compile p169]
[getSpecialCaseAssoc p300]
[get p??]
[assocleft p??]
[outerProduct p??]
[assocright p??]
[mkpf p??]
[$getDomainCode p??]
[$insideFunctorIfTrue p??]
[$specialCaseKeyList p??]
```

— defun compileCases —

```
(defun |compileCases| (x |$e|)
  (declare (special |$e|))
  (labels (
    (isEltArgumentIn (Rlist x)
      (cond
        ((atom x) nil)
        ((and (consp x) (eq (qfirst x) 'elt) (consp (qrest x))
              (consp (qcddr x)) (eq (qcdddr x) nil))
          (or (member (second x) Rlist)
              (isEltArgumentIn Rlist (cdr x))))
        ((and (consp x) (eq (qfirst x) 'qrefelt) (consp (qrest x))
              (consp (qcddr x)) (eq (qcdddr x) nil))
          (or (member (second x) Rlist)
              (isEltArgumentIn Rlist (cdr x))))
        (t
         (or (isEltArgumentIn Rlist (car x))
             (isEltArgumentIn Rlist (cdr x))))))
    (FindNamesFor (r rp)
      (let (v u)
        (declare (special |$getDomainCode|))
        (cons r
              (loop for item in |$getDomainCode|
                    do
                      (setq v (second item))
                      (setq u (third item))
                      when (and (equal (second u) r) (|eval| (subst rp r u :test #'equal)))
                        collect v))))))
  (let (|$specialCaseKeyList| specialCaseAssoc listOfDomains listOfAllCases cl)
    (declare (special |$specialCaseKeyList| |$true| |$insideFunctorIfTrue|))
    (setq |$specialCaseKeyList| nil)
    (cond
      ((null (eq |$insideFunctorIfTrue| t)) (|compile| x))
      (t
```

```

(setq specialCaseAssoc
  (loop for y in (|getSpecialCaseAssoc|)
    when (and (null (|get| (first y) '|specialCase| |$e|))
      (isEltArgumentIn (FindNamesFor (first y) (second y)) x))
    collect y))
(cond
  ((null specialCaseAssoc) (|compile| x))
  (t
    (setq listOfDomains (assocleft specialCaseAssoc))
    (setq listOfAllCases (|outerProduct| (assocright specialCaseAssoc)))
    (setq cl
      (loop for z in listOfAllCases
        collect
          (progn
            (setq |$specialCaseKeyList|
              (loop for d in listOfDomains for c in z
                collect (cons d c)))
            (cons
              (mkpf
                (loop for d in listOfDomains for c in z
                  collect (list 'equal d c))
                'and)
              (list (|compile| (copy x)))))))
    (setq |$specialCaseKeyList| nil)
    (cons 'cond (append cl (list (list |$true| (|compile| x))))))))))

```

defun compFunctorBody

```

[bootstrapError p215]
[compOrCroak p588]
[/editfile p??]
[$NRTaddForm p??]
[$functorForm p??]
[$bootstrapMode p??]

```

— defun compFunctorBody —

```

(defun |compFunctorBody| (form mode env parForm)
  (declare (ignore parForm))
  (let (tt)
    (declare (special |$NRTaddForm| |$functorForm| |$bootstrapMode| /editfile))
    (if |$bootstrapMode|
      (list (|bootstrapError| |$functorForm| /editfile) mode env)
      (progn
        (setq tt (|compOrCroak| form mode env))

```

```

(if (and (consp form) (member (qfirst form) '(|add| capsule)))
  tt
  (progn
    (setq |$NRTaddForm|
      (if (and (consp form) (eq (qfirst form) '|SubDomain|)
              (consp (qrest form)) (consp (qcddr form))
              (eq (qcdddr form) nil))
          (qsecond form)
          form))
    tt))))))

```

defun compile

```

[member p??]
[getmode p??]
[get p??]
[modeEqual p362]
[userError p??]
[encodeItem p181]
[strconc p??]
[kar p??]
[encodeFunctionName p179]
[splitEncodedFunctionName p180]
[sayBrightly p??]
[optimizeFunctionDef p223]
[putInLocalDomainReferences p185]
[constructMacro p182]
[spadCompileOrSetq p182]
[elapsedTime p??]
[addStats p??]
[printStats p??]
[$functionStats p??]
[$macroIfTrue p??]
[$doNotCompileJustPrint p??]
[$insideCapsuleFunctionIfTrue p??]
[$saveableItems p??]
[$lisplibItemsAlreadyThere p??]
[$splitUpItemsAlreadyThere p??]
[$lisplib p??]
[$compileOnlyCertainItems p??]
[$functorForm p??]
[$signatureOfForm p??]
[$suffix p??]

```

```

[$prefix p??]
[$signatureOfForm p??]
[$e p??]
[$functionStats p??]
[$savableItems p??]
[$suffix p??]

```

— defun compile —

```

(defun |compile| (u)
  (labels (
    (isLocalFunction (op)
      (let (tmp1)
        (declare (special |$e| |$formalArgList|))
        (and (null (|member| op |$formalArgList|))
          (progn
            (setq tmp1 (|getmode| op |$e|))
            (and (consp tmp1) (eq (qfirst tmp1) '|Mapping|)))))))
    (let (op lamExpr DC sig sel opexport opmodes opp parts s tt unew
      optimizedBody stuffToCompile result functionStats)
      (declare (special |$functionStats| |$macroIfTrue| |$doNotCompileJustPrint|
        |$insideCapsuleFunctionIfTrue| |$saveableItems| |$e|
        |$lisplibItemsAlreadyThere| |$splitUpItemsAlreadyThere|
        |$compileOnlyCertainItems| $LISPLIB |$suffix|
        |$signatureOfForm| |$functorForm| |$prefix|
        |$savableItems|))
        (setq op (first u))
        (setq lamExpr (second u))
        (when |$suffix|
          (setq |$suffix| (1+ |$suffix|))
          (setq opp
            (progn
              (setq opexport nil)
              (setq opmodes
                (loop for item in (|get| op '|modemap| |$e|)
                  do
                    (setq dc (caar item))
                    (setq sig (cdar item))
                    (setq sel (cadadr item))
                    when (and (eq dc '$)
                      (setq opexport t)
                      (let ((result t))
                        (loop for x in sig for y in |$signatureOfForm|
                          do (setq result (|modeEqual| x y)))
                        result)))
                    collect sel)))
              (cond
                ((isLocalFunction op)
                  (when opexport

```

```

      (|userError| (list '|%b| op '|%d| " is local and exported"))
      (intern (strconc (|encodeItem| |$prefix|) ";" (|encodeItem| op))))
    (t
      (|encodeFunctionName| op |$functorForm| |$signatureOfForm|
        '|;| |$suffix|))))
    (setq u (list opp lamExpr))
    (when (and $lisplib $compileOnlyCertainItems|)
      (setq parts (|splitEncodedFunctionName| (elt u 0) '|;|))
      (cond
        ((eq parts 'inner|)
          (setq |$savableItems| (cons (elt u 0) |$savableItems|)))
        (t
          (setq unew nil)
          (loop for item in |$splitUpItemsAlreadyThere|
            do
              (setq s (first item))
              (setq tt (second item))
              (when
                (and (equal (elt parts 0) (elt s 0))
                     (equal (elt parts 1) (elt s 1))
                     (equal (elt parts 2) (elt s 2)))
                (setq unew tt)))
          (cond
            ((null unew)
              (|sayBrightly| (list " Error: Item did not previously exist"))
              (|sayBrightly| (cons " Item not saved: " (|bright| (elt u 0))))
              (|sayBrightly|
                (list " What's there is: " |$lisplibItemsAlreadyThere|))
              nil)
            (t
              (|sayBrightly| (list " Renaming " (elt u 0) " as " unew))
              (setq u (cons unew (cdr u)))
              (setq |$savableItems| (cons unew |$saveableItems|))))))
      (setq optimizedBody (|optimizeFunctionDef| u))
      (setq stuffToCompile
        (if |$insideCapsuleFunctionIfTrue|
          (|putInLocalDomainReferences| optimizedBody
            optimizedBody))
        (cond
          ((eq |$doNotCompileJustPrint| t)
            (prettyprint stuffToCompile)
            opp)
          (|$macroIfTrue| (|constructMacro| stuffToCompile)))
      (t
        (setq result (|spadCompileOrSetq| stuffToCompile))
        (setq functionStats (list 0 (|elapsedTime|)))
        (setq |$functionStats| (|addStats| |$functionStats| functionStats))
        (|printStats| functionStats
          result))))

```

defvar \$NoValueMode

— initvars —

```
(defvar |$NoValueMode| ' |NoValueMode|)
```

defvar \$EmptyMode

`$EmptyMode` is a constant whose value is `$EmptyMode`. It is used by `isPartialMode` to decide if a modemap is partially constructed. If the `$EmptyMode` constant occurs anywhere in the modemap structure at any depth then the modemap is still incomplete. To find this constant the `isPartialMode` function calls `CONTAINED $EmptyMode Y` which will walk the structure `Y` looking for this constant.

— initvars —

```
(defvar |$EmptyMode| ' |EmptyMode|)
```

defun hasFullSignature

TPDHERE: test with BASTYPE [get p??]

— defun hasFullSignature —

```
(defun |hasFullSignature| (arg1 signature env)
  (let (target ml u)
    (setq target (first signature))
    (setq ml (rest signature))
    (when target
      (setq u
        (loop for x in arg1 for m in ml
              collect (or m (|get| x ' |mode| env) (return 'failed))))
      (unless (eq u 'failed) (cons target u)))))
```

defun addEmptyCapsuleIfNecessary

```
[kar p??]
[$SpecialDomainNames p??]
```

— defun addEmptyCapsuleIfNecessary —

```
(defun |addEmptyCapsuleIfNecessary| (target rhs)
  (declare (special |$SpecialDomainNames|) (ignore target))
  (if (member (kar rhs) |$SpecialDomainNames|)
      rhs
      (list 'add| rhs (list 'capsule))))
```

—

defun getTargetFromRhs

```
[stackSemanticError p??]
[getTargetFromRhs p173]
[compOrCroak p588]
```

— defun getTargetFromRhs —

```
(defun |getTargetFromRhs| (lhs rhs env)
  (declare (special |$EmptyMode|))
  (cond
    ((and (consp rhs) (eq (qfirst rhs) 'capsule))
     (|stackSemanticError|
      (list "target category of " lhs
            " cannot be determined from definition")
      nil))
    ((and (consp rhs) (eq (qfirst rhs) '|SubDomain|) (consp (qrest rhs)))
     (|getTargetFromRhs| lhs (second rhs) env))
    ((and (consp rhs) (eq (qfirst rhs) 'add|)
          (consp (qrest rhs)) (consp (qcddr rhs))
          (eq (qcdddr rhs) nil)
          (consp (qthird rhs))
          (eq (qcaaddr rhs) 'capsule))
     (|getTargetFromRhs| lhs (second rhs) env))
    ((and (consp rhs) (eq (qfirst rhs) '|Record|))
     (cons '|RecordCategory| (rest rhs)))
    ((and (consp rhs) (eq (qfirst rhs) '|Union|))
     (cons '|UnionCategory| (rest rhs)))
    ((and (consp rhs) (eq (qfirst rhs) '|List|))
     (cons '|ListCategory| (rest rhs)))
    ((and (consp rhs) (eq (qfirst rhs) '|Vector|))
     (cons '|VectorCategory| (rest rhs)))
```

```
(t
  (second (|compOrCroak| rhs |$EmptyMode| env))))
```

defun giveFormalParametersValues

```
[put p??]
[get p??]
```

— defun giveFormalParametersValues —

```
(defun |giveFormalParametersValues| (arg1 env)
  (dolist (x arg1)
    (setq env
      (|put| x '|value|
        (list (|genSomeVariable|) (|get| x '|mode| env) nil) env)))
  env)
```

defun macroExpandInPlace

```
[macroExpand p174]
```

— defun macroExpandInPlace —

```
(defun |macroExpandInPlace| (form env)
  (let (y)
    (setq y (|macroExpand| form env))
    (if (or (atom form) (atom y))
      y
      (progn
        (rplaca form (car y))
        (rplacd form (cdr y))
        form
        ))))
```

defun macroExpand

```
[macroExpand p174]
[macroExpandList p175]
```

— defun macroExpand —

```
(defun |macroExpand| (form env)
  (let (u)
    (cond
      ((atom form)
       (if (setq u (|get| form '|macro| env))
           (|macroExpand| u env)
           form))
      ((and (consp form) (eq (qfirst form) 'def)
            (consp (qrest form))
            (consp (qcddr form))
            (consp (qcdddr form))
            (consp (qcdddr form))
            (eq (qrest (qcdddr form)) nil))
       (list 'def (|macroExpand| (second form) env)
              (|macroExpandList| (third form) env)
              (|macroExpandList| (fourth form) env)
              (|macroExpand| (fifth form) env)))
      (t (|macroExpandList| form env))))
```

defun macroExpandList

[macroExpand p174]
[getdatabase p??]

— defun macroExpandList —

```
(defun |macroExpandList| (lst env)
  (let (tmp)
    (if (and (consp lst) (eq (qrest lst) nil)
            (identp (qfirst lst)) (getdatabase (qfirst lst) 'niladic)
            (setq tmp (|get| (qfirst lst) '|macro| env)))
        (|macroExpand| tmp env)
        (loop for x in lst collect (|macroExpand| x env))))
```

defun makeCategoryPredicates

[\$FormalMapVariableList p266]
[\$TriangleVariableList p??]

```
[$mvl p??]
[$tv1 p??]
```

— **defun makeCategoryPredicates** —

```
(defun |makeCategoryPredicates| (form u)
  (labels (
    (fn (u pl)
      (declare (special |$tv1| |$mvl|))
      (cond
        ((and (consp u) (eq (qfirst u) '|Join|) (consp (qrest u)))
          (fn (car (reverse (qrest u))) pl))
        ((and (consp u) (eq (qfirst u) '|has|))
          (|insert| (eqsubstlist |$mvl| |$tv1| u) pl))
        ((and (consp u) (member (qfirst u) '(signature attribute))) pl)
        ((atom u) pl)
        (t (fn1 u pl))))
    (fn1 (u pl)
      (dolist (x u) (setq pl (fn x pl)))
      pl))
  (declare (special |$FormalMapVariableList| |$mvl| |$tv1|
    |$TriangleVariableList|))
  (setq |$tv1| (take (|#| (cdr form)) |$TriangleVariableList|))
  (setq |$mvl| (take (|#| (cdr form)) (cdr |$FormalMapVariableList|)))
  (fn u nil)))
```

—————

defun mkCategoryPackage

```
[strconc p??]
[pname p??]
[getdatabase p??]
[abbreviationsSpad2Cmd p??]
[JoinInner p??]
[assoc p??]
[sublislis p??]
[$options p??]
[$categoryPredicateList p??]
[$e p??]
[$FormalMapVariableList p266]
```

— **defun mkCategoryPackage** —

```
(defun |mkCategoryPackage| (form cat def)
  (labels (
```

```

(fn (x oplist)
  (cond
    ((atom x) oplist)
    ((and (consp x) (eq (qfirst x) 'def) (consp (qrest x)))
      (cons (second x) oplist))
    (t
      (fn (cdr x) (fn (car x) oplist))))))
(gn (cat)
  (cond
    ((and (consp cat) (eq (qfirst cat) 'category)) (cddr cat))
    ((and (consp cat) (eq (qfirst cat) '|Join|)) (gn (|last| (qrest cat))))
    (t nil)))
(let (|$options| op argl packageName packageAbb nameForDollar packageArgl
      capsuleDefAlist explicitCatPart catvec fullCatOpList op1 sig
      catOpList packageCategory nils packageSig)
  (declare (special |$options| |$categoryPredicateList| |$e|
                    |$FormalMapVariableList|))
  (setq op (car form))
  (setq argl (cdr form))
  (setq packageName (intern (strconc (pname op) "&")))
  (setq packageAbb (intern (strconc (getdatabase op 'abbreviation) "-")))
  (setq |$options| nil)
  (|abbreviationsSpad2Cmd| (list '|domain| packageAbb packageName))
  (setq nameForDollar (car (setdifference '(s a b c d e f g h i) argl)))
  (setq packageArgl (cons nameForDollar argl))
  (setq capsuleDefAlist (fn def nil))
  (setq explicitCatPart (gn cat))
  (setq catvec (|eval| (|mkEvalableCategoryForm| form)))
  (setq fullCatOpList (elt (|JoinInner| (list catvec) |$e|) 1))
  (setq catOpList
    (loop for x in fullCatOpList do
      (setq op1 (caar x))
      (setq sig (cadar x))
      when (|assoc| op1 capsuleDefAlist)
        collect (list 'signature op1 sig)))
  (when catOpList
    (setq packageCategory
      (cons 'category
        (cons '|domain| (sublislis argl |$FormalMapVariableList| catOpList))))
    (setq nils (loop for x in argl collect nil))
    (setq packageSig (cons packageCategory (cons form nils)))
    (setq |$categoryPredicateList|
      (subst nameForDollar '$ |$categoryPredicateList| :test #'equal))
    (subst nameForDollar '$
      (list 'def (cons packageName packageArgl)
        packageSig (cons nil nils) def) :test #'equal))))

```

defun mkEvalableCategoryForm

```

[mkEvalableCategoryForm p178]
[compOrCroak p588]
[getdatabase p??]
[get p??]
[mkq p??]
[$Category p??]
[$e p??]
[$EmptyMode p172]
[$CategoryFrame p??]
[$Category p??]
[$CategoryNames p??]
[$e p??]

```

— defun mkEvalableCategoryForm —

```

(defun |mkEvalableCategoryForm| (c)
  (let (op arg1 tmp1 x m)
    (declare (special |$Category| |$e| |$EmptyMode| |$CategoryFrame|
                      |$CategoryNames|))
    (if (consp c)
        (progn
          (setq op (qfirst c))
          (setq arg1 (qrest c))
          (cond
            ((eq op '|Join|)
             (cons '|Join|
                   (loop for x in arg1
                        collect (|mkEvalableCategoryForm| x))))
            ((eq op '|DomainSubstitutionMacro|)
             (|mkEvalableCategoryForm| (cadr arg1)))
            ((eq op '|mkCategory|) c)
            ((member op |$CategoryNames|)
             (setq tmp1 (|compOrCroak| c |$EmptyMode| |$e|))
             (setq x (car tmp1))
             (setq m (cadr tmp1))
             (setq |$e| (caddr tmp1))
             (when (equal m |$Category|) x))
            ((or (eq (getdatabase op 'constructorkind) '|category|)
                  (|get| op '|isCategory| |$CategoryFrame|))
             (cons op
                   (loop for x in arg1
                        collect (mkq x))))
            (t
             (setq tmp1 (|compOrCroak| c |$EmptyMode| |$e|))
             (setq x (car tmp1))
             (setq m (cadr tmp1))
             (setq |$e| (caddr tmp1))

```

```
(when (equal m |$Category| x)))
(mkq c)))
```

defun encodeFunctionName

Code for encoding function names inside package or domain [mkRepetitionAssoc p180]

```
[encodeItem p181]
[stringimage p??]
[internl p??]
[getAbbreviation p298]
[length p??]
[$lisplib p??]
[$lisplibSignatureAlist p??]
[$lisplibSignatureAlist p??]
```

— defun encodeFunctionName —

```
(defun |encodeFunctionName| (fun package signature sep count)
  (let (packageName arglist signaturep reducedSig n x encodedSig encodedName)
    (declare (special |$lisplibSignatureAlist| $lisplib))
    (setq packageName (car package))
    (setq arglist (cdr package))
    (setq signaturep (subst '$ package signature :test #'equal))
    (setq reducedSig
      (|mkRepetitionAssoc| (append (cdr signaturep) (list (car signaturep)))))
    (setq encodedSig
      (let ((result ""))
        (loop for item in reducedSig
          do
            (setq n (car item))
            (setq x (cdr item))
            (setq result
              (strconc result
                (if (eql n 1)
                  (|encodeItem| x)
                  (strconc (stringimage n) (|encodeItem| x))))))
        result))
    (setq encodedName
      (internl (|getAbbreviation| packageName (|#| arglist))
        '|;| (|encodeItem| fun) '|;| encodedSig sep (stringimage count)))
    (when $lisplib
      (setq |$lisplibSignatureAlist|
        (cons (cons encodedName signaturep) |$lisplibSignatureAlist|)))
    encodedName))
```

defun mkRepititionAssoc

[mkRepfun p??]

— defun mkRepititionAssoc —

```
(defun |mkRepititionAssoc| (z)
  (labels (
    (mkRepfun (z n)
      (cond
        ((null z) nil)
        ((and (consp z) (eq (qrest z) nil) (list (cons n (qfirst z)))))
        ((and (consp z) (consp (qrest z)) (equal (qsecond z) (qfirst z)))
         (mkRepfun (cdr z) (1+ n)))
        (t (cons (cons n (car z)) (mkRepfun (cdr z) 1))))))
    (mkRepfun z 1)))
```

defun splitEncodedFunctionName

[stringimage p??]

[strpos p??]

— defun splitEncodedFunctionName —

```
(defun |splitEncodedFunctionName| (encodedName sep)
  (let (sep0 p1 p2 p3 s1 s2 s3 s4)
    ; sep0 is the separator used in "encodeFunctionName".
    (setq sep0 ";")
    (unless (stringp encodedName) (setq encodedName (stringimage encodedName)))
    (cond
      ((null (setq p1 (strpos sep0 encodedName 0 "*"))) nil)
      ; This is picked up in compile for inner functions in partial compilation
      ((null (setq p2 (strpos sep0 encodedName (1+ p1) "*"))) '|inner|)
      ((null (setq p3 (strpos sep encodedName (1+ p2) "*"))) nil)
      (t
       (setq s1 (substring encodedName 0 p1))
       (setq s2 (substring encodedName (1+ p1) (- p2 p1 1)))
       (setq s3 (substring encodedName (1+ p2) (- p3 p2 1)))
       (setq s4 (substring encodedName (1+ p3) nil))
       (list s1 s2 s3 s4)))))
```

defun encodeItem

```
[getCaps p181]
[identp p??]
[pname p??]
[stringimage p??]
```

— defun encodeItem —

```
(defun |encodeItem| (x)
  (cond
    ((consp x) (|getCaps| (qfirst x)))
    ((identp x) (pname x))
    (t (stringimage x))))
```

defun getCaps

```
[stringimage p??]
[maxindex p??]
[l-case p??]
[strconc p??]
```

— defun getCaps —

```
(defun |getCaps| (x)
  (let (s c clist tmp1)
    (setq s (stringimage x))
    (setq clist
      (loop for i from 0 to (maxindex s)
            when (upper-case-p (setq c (elt s i)))
              collect c))
    (cond
      ((null clist) "_")
      (t
       (setq tmp1
         (cons (first clist) (loop for u in (rest clist) collect (l-case u))))
       (let ((result ""))
         (loop for u in tmp1
               do (setq result (strconc result u)))
         result))))))
```

defun constructMacro

```
constructMacro (form is [nam,[lam,vl,body]]) [stackSemanticError p??]
[identp p??]
```

— defun constructMacro —

```
(defun |constructMacro| (form)
  (let (vl body)
    (setq vl (cadadr form))
    (setq body (car (cddadr form)))
    (cond
      ((null (let ((result t))
                (loop for x in vl
                      do (setq result (and result (atom x))))
               result))
        (|stackSemanticError| (list '|illegal parameters for macro: | vl) nil))
      (t
       (list 'xlam (loop for x in vl when (identp x) collect x) body))))))
```

—————

defun spadCompileOrSetq

```
[contained p??]
[sayBrightly p??]
[bright p??]
[LAM,EVALANDFILEACTQ p??]
[mkq p??]
[comp p590]
[compileConstructor p183]
[$insideCapsuleFunctionIfTrue p??]
```

— defun spadCompileOrSetq —

```
(defun |spadCompileOrSetq| (form)
  (let (nam lam vl body namp tmp1 e vlp macform)
    (declare (special |$insideCapsuleFunctionIfTrue|))
    (setq nam (car form))
    (setq lam (caadr form))
    (setq vl (cadadr form))
    (setq body (car (cddadr form)))
    (cond
      ((and (consp vl) (progn (setq tmp1 (reverse vl)) t)
            (consp tmp1)
            (progn
```

```

      (setq e (qfirst tmp1))
      (setq vlp (qrest tmp1))
      t)
    (progn (setq vlp (nreverse vlp)) t)
    (consp body)
    (progn (setq namp (qfirst body)) t)
    (equal (qrest body) vlp))
  (|LAM,EVALANDFILEACTQ|
   (list 'put (mkq nam) (mkq '|SPADreplace|) (mkq namp)))
  (|sayBrightly|
   (cons "      " (append (|bright| nam)
                          (cons "is replaced by" (|bright| namp))))))
  ((and (or (atom body)
            (let ((result t))
              (loop for x in body
                    do (setq result (and result (atom x))))
              result)))
        (consp vl)
        (progn (setq tmp1 (reverse vl)) t)
        (consp tmp1)
        (progn
         (setq e (qfirst tmp1))
         (setq vlp (qrest tmp1))
         t)
        (progn (setq vlp (nreverse vlp)) t)
        (null (contained e body)))
   (setq macform (list 'xlam vlp body))
   (|LAM,EVALANDFILEACTQ|
    (list 'put (mkq nam) (mkq '|SPADreplace|) (mkq macform)))
   (|sayBrightly| (cons "      " (append (|bright| nam)
                                          (cons "is replaced by" (|bright| body))))))
  (t nil))
(if |$insideCapsuleFunctionIfTrue|
  (car (comp (list form)))
  (|compileConstructor| form)))

```

defun compileConstructor

[compileConstructor1 p184]
 [clearClams p??]

— defun compileConstructor —

```

(defun |compileConstructor| (form)
  (let (u)

```

```
(setq u (|compileConstructor1| form))
(|clearClams|)
u))
```

defun compileConstructor1

```
[getdatabase p??]
[compAndDefine p185]
[comp p590]
[clearConstructorCache p??]
[$mutableDomain p??]
[$ConstructorCache p??]
[$clamList p??]
[$clamList p??]
```

— defun compileConstructor1 —

```
(defun |compileConstructor1| (form)
  (let (|$clamList| fn key vl body1 lambdaOrSlam compForm u)
    (declare (special |$clamList| |$ConstructorCache| |$mutableDomain|))
    (setq fn (car form))
    (setq key (caadr form))
    (setq vl (cadadr form))
    (setq body1 (cddadr form))
    (setq |$clamList| nil)
    (setq lambdaOrSlam
      (cond
        ((eq (getdatabase fn 'constructorkind) '|category|) 'spadslam)
        (|$mutableDomain| 'lambda)
        (t
         (setq |$clamList|
              (cons (list fn '|$ConstructorCache| '|domainEqualList| '|count|)
                    |$clamList|))
          'lambda)))
    (setq compForm (list (list fn (cons lambdaOrSlam (cons vl body1)))))
    (if (eq (getdatabase fn 'constructorkind) '|category|)
        (setq u (|compAndDefine| compForm))
        (setq u (comp compForm)))
    (|clearConstructorCache| fn)
    (car u)))
```

defun compAndDefine

This function is used but never defined. We define a dummy function here. All references to it should be removed. **TPDHERE: This function is used but never defined. Remove it.**

— defun compAndDefine —

```
(defun compAndDefine (arg)
  (declare (ignore arg))
  nil)
```

—————

defun putInLocalDomainReferences

```
[NRTputInTail p185]
[$QuickCode p??]
[$elt p306]
```

— defun putInLocalDomainReferences —

```
(defun |putInLocalDomainReferences| (def)
  (let (|$elt| opName lam var1 body)
    (declare (special |$elt| |$QuickCode|))
    (setq opName (car def))
    (setq lam (caadr def))
    (setq var1 (cadadr def))
    (setq body (car (cddadr def)))
    (setq |$elt| (if |$QuickCode| 'qrefelt 'elt))
    (|NRTputInTail| (cddadr def))
    def))
```

—————

defun NRTputInTail

```
[lassoc p??]
[NRTassocIndex p342]
[rplaca p??]
[NRTputInHead p186]
[$elt p306]
[$devalueList p??]
```

— defun NRTputInTail —

```

(defun |NRTputInTail| (x)
  (let (u k)
    (declare (special |$elt| |$devalueList|))
    (maplist #'(lambda (y)
      (cond
        ((atom (setq u (car y)))
         (cond
           ((or (eq u '$) (lassoc u |$devalueList|))
            nil)
           ((setq k (|NRTassocIndex| u))
            (cond
              ; u atomic means that the slot will always contain a vector
              ((atom u) (rplaca y (list |$elt| '$ k)))
              ; this reference must check that slot is a vector
              (t (rplaca y (list 'spadcheckelt '$ k))))
            (t nil))))
        (t (|NRTputInHead| u))))
    x)
  x))

```

defun NRTputInHead

[\[NRTputInTail p185\]](#)
[\[NRTassocIndex p342\]](#)
[\[NRTputInHead p186\]](#)
[\[lastnode p??\]](#)
[\[keyedSystemError p??\]](#)
[\[\\$elt p306\]](#)

— defun NRTputInHead —

```

(defun |NRTputInHead| (bod)
  (let (fn clauses dom tmp2 ind k)
    (declare (special |$elt|))
    (cond
      ((atom bod) bod)
      ((and (consp bod) (eq (qcar bod) 'spadcall) (consp (qcdr bod))
        (progn (setq tmp2 (reverse (qcdr bod))) t) (consp tmp2))
       (setq fn (qcar tmp2))
       (|NRTputInTail| (cdr bod))
       (cond
         ((and (consp fn) (consp (qcdr fn)) (consp (qcdr (qcdr fn)))
          (eq (qcdddr fn) nil) (null (eq (qsecond fn) '$))
          (member (qcar fn) '(elt qrefelt const)))
          (when (setq k (|NRTassocIndex| (qsecond fn)))

```

```

      (rplaca (lastnode bod) (list |$elt| '$ k))))
    (t (|NRTputInHead| fn) bod)))
  ((and (consp bod) (eq (qcar bod) 'cond))
    (setq clauses (qcdr bod))
    (loop for cc in clauses do (|NRTputInTail| cc))
    bod)
  ((and (consp bod) (eq (qcar bod) 'quote)) bod)
  ((and (consp bod) (eq (qcar bod) 'closedfn)) bod)
  ((and (consp bod) (eq (qcar bod) 'spadconst) (consp (qcdr bod))
    (consp (qcddr bod)) (eq (qcdddr bod) nil))
    (setq dom (qsecond bod))
    (setq ind (qthird bod))
    (rplaca bod |$elt|)
    (cond
      ((eq dom '$) nil)
      ((setq k (|NRTassocIndex| dom))
        (rplaca (lastnode bod) (list |$elt| '$ k))
        bod)
      (t
        (|keyedSystemError| 'S2GE0016
          (list "NRTputInHead" "unexpected SPADCONST form")))))
  (t
    (|NRTputInHead| (car bod))
    (|NRTputInTail| (cdr bod) bod)))

```

defun getArgumentModeOrMoan

[getArgumentMode p³⁰⁵]
 [stackSemanticError p??]

— defun getArgumentModeOrMoan —

```

(defun |getArgumentModeOrMoan| (x form env)
  (or (|getArgumentMode| x env)
    (|stackSemanticError|
      (list '|argument| x '| of | form '| is not declared|) nil)))

```

defun augLisplibModemapsFromCategory

[sublis p??]
 [mkAlistOfExplicitCategoryOps p¹⁸⁹]

```

[isCategoryForm p??]
[lassoc p??]
[member p??]
[mkpf p??]
[interactiveModemapForm p191]
[$lisplibModemapAlist p??]
[$EmptyEnvironment p??]
[$domainShell p??]
[$PatternVariableList p??]
[$lisplibModemapAlist p??]

```

— defun augLisplibModemapsFromCategory —

```

(defun |augLisplibModemapsFromCategory| (form body signature)
  (let (arg1 sl opAlist nonCategorySigAlist domainList catPredList op sig
        pred sel predp modemap)
    (declare (special |$lisplibModemapAlist| |$EmptyEnvironment|
                      |$domainShell| |$PatternVariableList|))
    (setq op (car form))
    (setq arg1 (cdr form))
    (setq sl
      (cons (cons '$ '*1)
            (loop for a in arg1 for p in (rest |$PatternVariableList|)
                  collect (cons a p))))
    (setq form (sublis sl form))
    (setq body (sublis sl body))
    (setq signature (sublis sl signature))
    (when (setq opAlist (sublis sl (elt |$domainShell| 1)))
      (setq nonCategorySigAlist
        (|mkAlistOfExplicitCategoryOps| (subst '*1 '$ body :test #'equal)))
      (setq domainList
        (loop for a in (rest form) for m in (rest signature)
              when (|isCategoryForm| m |$EmptyEnvironment|)
              collect (list a m)))
      (setq catPredList
        (loop for u in (cons (list '*1 form) domainList)
              collect (cons '|ofCategory| u)))
      (loop for entry in opAlist
            when (|member| (cadar entry) (lassoc (caar entry) nonCategorySigAlist))
            do
              (setq op (caar entry))
              (setq sig (cadar entry))
              (setq pred (cadr entry))
              (setq sel (caddr entry))
              (setq predp (mkpf (cons pred catPredList) 'and))
              (setq modemap (list (cons '*1 sig) (list predp sel)))
              (setq |$lisplibModemapAlist|
                (cons (cons op (|interactiveModemapForm| modemap))
                      |$lisplibModemapAlist|))))))

```


defun mkAlistOfExplicitCategoryOps

```

[keyedSystemError p??]
[union p??]
[mkAlistOfExplicitCategoryOps p189]
[flattenSignatureList p190]
[nreverse0 p??]
[remdup p??]
[assocleft p??]
[isCategoryForm p??]
[$e p??]

```

— defun mkAlistOfExplicitCategoryOps —

```

(defun |mkAlistOfExplicitCategoryOps| (target)
  (labels (
    (atomizeOp (op)
      (cond
        ((atom op) op)
        ((and (consp op) (eq (qrest op) nil)) (qfirst op))
        (t (|keyedSystemError| 'S2GE0016
          (list "mkAlistOfExplicitCategoryOps" "bad signature")))))
    (fn (op u)
      (if (and (consp u) (consp (qfirst u)))
        (if (equal (qcaar u) op)
          (cons (qcdar u) (fn op (qrest u)))
          (fn op (qrest u)))))
  (let (z tmp1 op sig u opList)
    (declare (special |$e|))
    (when (and (consp target) (eq (qfirst target) '|add|) (consp (qrest target)))
      (setq target (second target)))
    (cond
      ((and (consp target) (eq (qfirst target) '|Join|))
        (setq z (qrest target))
        (PROG (tmp1)
          (RETURN
            (DO ((G167566 z (CDR G167566)) (cat nil))
              ((OR (ATOM G167566) (PROGN (setq cat (CAR G167566)) nil))
                tmp1)
              (setq tmp1 (|union| tmp1 (|mkAlistOfExplicitCategoryOps| cat)))))))
      ((and (consp target) (eq (qfirst target) 'category)
        (progn
          (setq tmp1 (qrest target))
          (and (consp tmp1)

```

```

      (progn (setq z (qrest tmp1)) t))))
(setq z (|flattenSignatureList| (cons 'progn z)))
(setq u
  (prog (G167577)
    (return
      (do ((G167583 z (cdr G167583)) (x nil))
        ((or (atom G167583)) (nreverse0 G167577))
        (setq x (car G167583))
        (cond
          ((and (consp x) (eq (qfirst x) 'signature) (consp (qrest x))
            (consp (qcddr x)))
            (setq op (qsecond x))
            (setq sig (qthird x))
            (setq G167577 (cons (cons (atomizeOp op) sig) G167577)))))))
(setq opList (remdup (assocleft u)))
(prog (G167593)
  (return
    (do ((G167598 opList (cdr G167598)) (x nil))
      ((or (atom G167598)) (nreverse0 G167593))
      (setq x (car G167598))
      (setq G167593 (cons (cons x (fn x u)) G167593))))))
(|isCategoryForm| target |$e|) nil)
(t
  (|keyedSystemError| 'S2GE0016
    (list "mkAlistOfExplicitCategoryOps" "bad signature"))))

```

defun flattenSignatureList

[flattenSignatureList p190]

— defun flattenSignatureList —

```

(defun |flattenSignatureList| (x)
  (let (zz)
    (cond
      ((atom x) nil)
      ((and (consp x) (eq (qfirst x) 'signature)) (list x))
      ((and (consp x) (eq (qfirst x) 'if) (consp (qrest x))
        (consp (qcddr x)) (consp (qcdddr x))
        (eq (qcdddr x) nil))
        (append (|flattenSignatureList| (third x))
          (|flattenSignatureList| (fourth x))))
      ((and (consp x) (eq (qfirst x) 'progn))
        (loop for x in (qrest x)
          do
            (if (and (consp x) (eq (qfirst x) 'signature))

```

```

      (setq zz (cons x zz))
      (setq zz (append (|flattenSignatureList| x) zz)))
    zz)
  (t nil))))

```

defun interactiveModemapForm

Create modemap form for use by the interpreter. This function replaces all specific domains mentioned in the modemap with pattern variables, and predicates [replaceVars p192]

[modemapPattern p199]

[substVars p198]

[fixUpPredicate p192]

[\$PatternVariableList p??]

[\$FormalMapVariableList p266]

— defun interactiveModemapForm —

```

(defun |interactiveModemapForm| (mm)
  (labels (
    (fn (x)
      (if (and (consp x) (consp (qrest x))
              (consp (qcddr x)) (eq (qcddr x) nil)
              (not (eq (qfirst x) '|isFreeFunction|))
              (atom (qthird x)))
          (list (first x) (second x) (list (third x)))
          x)))
    (let (pattern dc sig mmpat patternAlist partial patvars
          domainPredicateList tmp1 pred dependList cond)
      (declare (special |$PatternVariableList| |$FormalMapVariableList|))
      (setq mm
        (|replaceVars| (copy mm) |$PatternVariableList| |$FormalMapVariableList|))
      (setq pattern (car mm))
      (setq dc (caar mm))
      (setq sig (cdar mm))
      (setq pred (cadr mm))
      (setq pred
        (prog ()
          (return
            (do ((x pred (cdr x)) (result nil))
              ((atom x) (nreverse0 result))
              (setq result (cons (fn (car x)) result))))))
      (setq tmp1 (|modemapPattern| pattern sig))
      (setq mmpat (car tmp1))
      (setq patternAlist (cadr tmp1))
      (setq partial (caddr tmp1))

```

Replace every identifier in oldvars with the corresponding identifier in newvars in the expression x

```
[length p??]  
[orderPredicateItems p193]  
[moveORsOutside p197]
```

```
(defun |fixUpPredicate| (predClause domainPreds partial sig)
  (let (predicate fn skip predicates tmp1 dependList pred)
    (setq predicate (car predClause))
    (setq fn (cadr predClause))
    (setq skip (cddr predClause))
    (cond
      ((eq (car predicate) 'and)
        (setq predicates (append domainPreds (cdr predicate))))
      ((not (equal predicate (mkq t)))
        (setq predicates (cons predicate domainPreds)))
      (t
        (setq predicates (or domainPreds (list predicate))))))
```

```

(cond
  (> (|#| predicates) 1)
  (setq pred (cons 'and predicates))
  (setq tmp1 (|orderPredicateItems| pred sig skip))
  (setq pred (car tmp1))
  (setq dependlist (cdr tmp1))
  tmp1)
(t
  (setq pred (|orderPredicateItems| (car predicates) sig skip))
  (setq dependList
    (when (and (consp pred) (eq (qfirst pred) '|isDomain|)
              (consp (qrest pred)) (consp (qcddr pred))
              (eq (qcdddr pred) nil)
              (consp (qthird pred))
              (eq (qcdaddr pred) nil))
          (list (second pred)))))
  (setq pred (|moveORsOutside| pred))
  (when partial (setq pred (cons '|partial| pred)))
  (cons (cons pred (cons fn skip)) dependList)))

```

defun orderPredicateItems

[signatureTran p193]
 [orderPredTran p194]

— defun orderPredicateItems —

```

(defun |orderPredicateItems| (pred1 sig skip)
  (let (pred)
    (setq pred (|signatureTran| pred1))
    (if (and (consp pred) (eq (qfirst pred) 'and))
        (|orderPredTran| (qrest pred) sig skip
                          pred)))

```

defun signatureTran

[signatureTran p193]
 [isCategoryForm p??]
 [\$e p??]

— defun signatureTran —

```
(defun |signatureTran| (pred)
  (declare (special |$e|))
  (cond
    ((atom pred) pred)
    ((and (consp pred) (eq (qfirst pred) '|has|) (CONSP (qrest pred))
      (consp (qcddr pred))
      (eq (qcdddr pred) nil)
      (|isCategoryForm| (third pred) |$e|))
      (list '|ofCategory| (second pred) (third pred)))
    (t
      (loop for p in pred
        collect (|signatureTran| p))))))
```

defun orderPredTran

```
[member p??]
[delete p??]
[unionq p??]
[listOfPatternIds p??]
[intersectionq p??]
[setdifference p??]
[insertWOC p??]
[isDomainSubst p196]
```

— defun orderPredTran —

```
(defun |orderPredTran| (oldList sig skip)
  (let (lastDependList somethingDone lastPreds indepvl depvl dependList
        noldList x ids fullDependList newList answer)
    ; --(1) make two kinds of predicates appear last:
    ; ----- (op *target ..) when *target does not appear later in sig
    ; ----- (isDomain *1 ..)
    (SEQ
      (loop for pred in oldList
        do (cond
          ((or (and (consp pred) (consp (qrest pred))
            (consp (qcddr pred))
            (eq (qcdddr pred) nil)
            (member (qfirst pred) '|isDomain| |ofCategory|))
            (equal (qsecond pred) (car sig))
            (null (|member| (qsecond pred) (cdr sig))))
            (and (null skip) (consp pred) (eq (qfirst pred) '|isDomain|)
              (consp (qrest pred)) (consp (qcddr pred))
              (eq (qcdddr pred) nil)
              (equal (qsecond pred) '*1)))
```

```

      (setq oldList (|delete| pred oldList))
      (setq lastPreds (cons pred lastPreds))))))
; --(2a) lastDependList=list of all variables that lastPred forms depend upon
(setq lastDependList
  (let (result)
    (loop for x in lastPreds
      do (setq result (unionq result (|listOfPatternIds| x))))
    result))
; --(2b) dependList=list of all variables that isDom/ofCat forms depend upon
(setq dependList
  (let (result)
    (loop for x in oldList
      do (when
          (and (consp x)
               (or (eq (qfirst x) '|isDomain|) (eq (qfirst x) '|ofCategory|))
               (consp (qrest x)) (consp (qcddr x))
               (eq (qcdddr x) nil))
            (setq result (unionq result (|listOfPatternIds| (third x))))))
    result))
; --(3a) newList= list of ofCat/isDom entries that don't depend on
(loop for x in oldList
  do
    (cond
      ((and (consp x)
            (or (eq (qfirst x) '|ofCategory|) (eq (qfirst x) '|isDomain|))
            (consp (qrest x)) (consp (qcddr x))
            (eq (qcdddr x) nil))
        (setq indepvl (|listOfPatternIds| (second x)))
        (setq depvl (|listOfPatternIds| (third x)))
        (t
         (setq indepvl (|listOfPatternIds| x))
         (setq depvl nil)))
      (when
        (and (null (intersectionq indepvl dependList))
              (intersectionq indepvl lastDependList))
          (setq somethingDone t)
          (setq lastPreds (append lastPreds (list x)))
          (setq oldList (|delete| x oldList))))))
; --(3b) newList= list of ofCat/isDom entries that don't depend on
(loop while oldList do
  (loop for x in oldList do
    (cond
      ((and (consp x)
            (or (eq (qfirst x) '|ofCategory|) (eq (qfirst x) '|isDomain|))
            (consp (qrest x))
            (consp (qcddr x)) (eq (qcdddr x) nil))
        (setq indepvl (|listOfPatternIds| (second x)))
        (setq depvl (|listOfPatternIds| (third x)))
        (t
         (setq indepvl (|listOfPatternIds| x))

```

```

      (setq depvl nil)))
    (when (null (intersectionq indepvl dependList))
      (setq dependList (SETDIFFERENCE dependList depvl))
      (setq newList (APPEND newList (list x))))))
; --(4) noldList= what is left over
(cond
  ((equal (setq noldList (setdifference oldList newList)) oldList)
    (setq newList (APPEND newList oldList))
    (return nil))
  (t
   (setq oldList noldList))))
(loop for pred in newList do
  (when
    (and (consp pred)
         (or (eq (qfirst pred) '|isDomain|) (eq (qfirst x) '|ofCategory|))
         (consp (qrest pred))
         (consp (qcddr pred))
         (eq (qcdddr pred) nil))
         (setq ids (|listOfPatternIds| (third pred))))
    (when
      (let (result)
        (loop for id in ids do
          (setq result (and result (|member| id fullDependList))))
        result)
      (setq fullDependList (|insertWOC| (second pred) fullDependList)))
    (setq fullDependList (unionq fullDependList ids))))
(setq newList (append newList lastPreds))
(setq newList (|isDomainSubst| newList))
(setq answer
  (cons (cons 'and newList) (intersectionq fullDependList sig))))

```

defun isDomainSubst

— defun isDomainSubst —

```

(defun |isDomainSubst| (u)
  (labels (
    (findSub (x alist)
      (cond
        ((null alist) nil)
        ((and (consp alist) (consp (qfirst alist))
          (eq (qcaar alist) '|isDomain|)
          (consp (qcdar alist))
          (consp (qcddar alist))
          (eq (qcdddar alist) nil)

```



```

        (equal x (cadar alist)))
      (caddr alist))
    (t (findSub x (cdr alist))))))
(fn (x alist)
  (let (s)
    (declare (special |$PatternVariableList|))
    (if (atom x)
      (if
        (and (identp x)
              (member x |$PatternVariableList|)
              (setq s (findSub x alist)))
          s
          x)
      (cons (car x)
            (loop for y in (cdr x)
                  collect (fn y alist))))))
(let (head tail nhead)
  (if (consp u)
    (progn
      (setq head (qfirst u))
      (setq tail (qrest u))
      (setq nhead
        (cond
          ((and (consp head) (eq (qfirst head) '|isDomain|)
                (consp (qrest head)) (consp (qcddr head))
                (eq (qcddr head) nil))
            (list '|isDomain| (second head)
                  (fn (third head) tail)))
          (t head))))
      (cons nhead (|isDomainSubst| (cdr u))))
    u))))

```

defun moveORsOutside

[moveORsOutside p197]

— defun moveORsOutside —

```

(defun |moveORsOutside| (p)
  (let (q x)
    (cond
      ((and (consp p) (eq (qfirst p) 'and))
        (setq q
          (prog (G167169)
            (return
              (do ((G167174 (cdr p) (cdr G167174)) (|r| nil))

```

```

      ((or (atom G167174)) (nreverse0 G167169))
      (setq |r| (CAR G167174))
      (setq G167169 (cons (|moveORsOutside| |r|) G167169))))))
(cond
  ((setq x
    (let (tmp1)
      (loop for r in q
        when (and (consp r) (eq (qfirst r) 'or))
        do (setq tmp1 (or tmp1 r)))
      tmp1))
    (|moveORsOutside|
     (cons 'or
      (let (tmp1)
        (loop for tt in (cdr x)
          do (setq tmp1 (cons (cons 'and (subst tt x q :test #'equal)) tmp1)))
        (nreverse0 tmp1))))))
    (t (cons 'and q))))
(t p))))

(defun |moveORsOutside| (p)
  (let (q s x tmp1)
    (cond
      ((and (consp p) (eq (qfirst p) 'and))
       (setq q (loop for r in (qrest p) collect (|moveORsOutside| r)))
       (setq tmp1
        (loop for r in q
          when (and (consp r) (eq (qrest r) 'or))
          collect r))
       (setq x (mapcar #'(lambda (a b) (or a b)) tmp1))
       (if x
        (|moveORsOutside|
         (cons 'or
          (loop for tt in (cdr x)
            collect (cons 'and (subst tt x q :test #'equal))))))
        (cons 'and q))))
    ('t p))))

```

defun substVars

Make pattern variable substitutions. [nsubst p??]
 [contained p??]
 [\$FormalMapVariableList p266]

— defun substVars —

```
(defun |substVars| (pred patternAlist patternVarList)
```

```

(let (patVar value everything replacementVar domainPredicates)
(declare (special |$FormalMapVariableList|))
(setq domainPredicates NIL)
(maplist
 #'(lambda (x)
   (setq patVar (caar x))
   (setq value (cdar x))
   (setq pred (subst patVar value pred :test #'equal))
   (setq patternAlist (|subst| patVar value patternAlist))
   (setq domainPredicates
    (subst patVar value domainPredicates :test #'equal))
   (unless (member value |$FormalMapVariableList|)
    (setq domainPredicates
     (cons (list 'isDomain| patVar value) domainPredicates))))
 patternAlist)
(setq everything (list pred patternAlist domainPredicates))
(dolist (var |$FormalMapVariableList|)
 (cond
  ((contained var everything)
   (setq replacementVar (car patternVarList))
   (setq patternVarList (cdr patternVarList))
   (setq pred (subst replacementVar var pred :test #'equal))
   (setq domainPredicates
    (subst replacementVar var domainPredicates :test #'equal))))
 (list pred domainPredicates)))

```

defun modemapPattern

```

[rassoc p??]
[$PatternVariableList p??]

```

— defun modemapPattern —

```

(defun |modemapPattern| (mmPattern sig)
(let (partial patvar patvars mmpat patternAlist)
(declare (special |$PatternVariableList|))
(setq patternAlist nil)
(setq mmpat nil)
(setq patvars |$PatternVariableList|)
(setq partial nil)
(maplist
 #'(lambda (xTails)
   (let ((x (car xTails)))
    (when (and (consp x) (eq (qfirst x) '|Union|)
              (consp (qrest x)) (consp (qcddr x))

```

```

      (eq (qcdddr x) nil)
      (equal (third x) "failed")
      (equal xTails sig))
    (setq x (second x))
    (setq partial t))
  (setq patvar (|rassoc| x patternAlist))
  (cond
    ((null (null patvar))
     (setq mmpat (cons patvar mmpat)))
    (t
     (setq patvar (car patvars))
     (setq patvars (cdr patvars))
     (setq mmpat (cons patvar mmpat))
     (setq patternAlist (cons (cons patvar x) patternAlist))))))
  mmPattern)
(list (nreverse mmpat) patternAlist partial patvars)))

```

defun evalAndRwriteLispForm

```

[eval p??]
[rwriteLispForm p200]

```

— defun evalAndRwriteLispForm —

```

(defun |evalAndRwriteLispForm| (key form)
  (|eval| form)
  (|rwriteLispForm| key form))

```

defun rwriteLispForm

```

[$libFile p??]
[$lisplib p??]

```

— defun rwriteLispForm —

```

(defun |rwriteLispForm| (key form)
  (declare (special |$libFile| $lisplib))
  (when $lisplib
    (|rwrite| key form |$libFile|)
    (|LAM,FILEACTQ| key form)))

```

defun mkConstructor

[mkConstructor p201]

— defun mkConstructor —

```

(defun |mkConstructor| (form)
  (cond
    ((atom form) (list '|devaluate| form))
    ((null (rest form)) (list 'quote (list (first form))))
    (t
     (cons 'list
           (cons (mkq (first form))
                 (loop for x in (rest form) collect (|mkConstructor| x)))))))

```

defun unloadOneConstructor

[remprop p??]

[mkAutoLoad p??]

— defun unloadOneConstructor —

```

(defun |unloadOneConstructor| (cnam fn)
  (remprop cnam 'loaded)
  (setf (symbol-function cnam) (|mkAutoLoad| fn cnam)))

```

defun lisplibDoRename

[replaceFile p??]

[\$spadLibFT p??]

— defun lisplibDoRename —

```

(defun |lisplibDoRename| (libName)
  (declare (special |$spadLibFT|))
  (replaceFile (list libName |$spadLibFT| 'a) (list libName 'errorlib 'a)))

```

defun initializeLisplib

```

[erase p??]
[writeLib1 p203]
[adoptions p??]
[pathnameTypeId p??]
[LAM,FILEACTQ p??]
[$erase p??]
[$libFile p??]
[$libFile p??]
[$lisplibForm p??]
[$lisplibModemap p??]
[$lisplibKind p??]
[$lisplibModemapAlist p??]
[$lisplibAbbreviation p??]
[$lisplibAncestors p??]
[$lisplibOpAlist p??]
[$lisplibOperationAlist p??]
[$lisplibSuperDomain p??]
[$lisplibVariableAlist p??]
[$lisplibSignatureAlist p??]
[/editfile p??]
[/major-version p??]
[errors p??]

```

— defun initializeLisplib —

```

(defun |initializeLisplib| (libName)
  (declare (special $erase |$libFile| |$lisplibForm|
                    |$lisplibModemap| |$lisplibKind| |$lisplibModemapAlist|
                    |$lisplibAbbreviation| |$lisplibAncestors|
                    |$lisplibOpAlist| |$lisplibOperationAlist|
                    |$lisplibSuperDomain| |$lisplibVariableAlist| errors
                    |$lisplibSignatureAlist| /editfile /major-version errors))
  ($erase libName 'errorlib 'a)
  (setq errors 0)
  (setq |$libFile| (|writeLib1| libname 'errorlib 'a))
  (adoptions 'file |$libFile|)
  (setq |$lisplibForm| nil)
  (setq |$lisplibModemap| nil)
  (setq |$lisplibKind| nil)
  (setq |$lisplibModemapAlist| nil)
  (setq |$lisplibAbbreviation| nil)
  (setq |$lisplibAncestors| nil)
  (setq |$lisplibOpAlist| nil)
  (setq |$lisplibOperationAlist| nil)
  (setq |$lisplibSuperDomain| nil)
  (setq |$lisplibVariableAlist| nil)

```

```
(setq $lisplibSignatureAlist nil)
(when (eq (pathnameTypeId /editfile) 'spad)
  (|LAM,FILEACTQ| 'version (list '/versioncheck /major-version))))
```

defun writeLib1

[rdefiostream p??]

— defun writeLib1 —

```
(defun |writeLib1| (fn ft fm)
  (rdefiostream (cons (list 'file fn ft fm) (list '(mode . output))))))
```

defun finalizeLisplib

[lisplibWrite p209]
 [removeZeroOne p??]
 [namestring p??]
 [getConstructorOpsAndAtts p205]
 [NRTgenInitialAttributeAlist p??]
 [mergeSignatureAndLocalVarAlists p209]
 [finalizeDocumentation p492]
 [profileWrite p??]
 [sayMSG p??]
 [\$lisplibForm p??]
 [\$libFile p??]
 [\$lisplibKind p??]
 [\$lisplibModemap p??]
 [\$lisplibCategory p??]
 [\$/editfile p??]
 [\$lisplibModemapAlist p??]
 [\$lisplibForm p??]
 [\$lisplibModemap p??]
 [\$FormalMapVariableList p266]
 [\$lisplibSuperDomain p??]
 [\$lisplibSignatureAlist p??]
 [\$lisplibVariableAlist p??]
 [\$lisplibAttributes p??]
 [\$lisplibPredicates p??]

```

[$lisplibAbbreviation p??]
[$lisplibParents p??]
[$lisplibAncestors p??]
[$lisplibSlot1 p??]
[$profileCompiler p??]
[$spadLibFT p??]
[$lisplibCategory p??]
[$pairlis p??]
[$NRTslot1PredicateList p??]

```

— **defun finalizeLisplib** —

```

(defun |finalizeLisplib| (libName)
  (let (|$pairlis| |$NRTslot1PredicateList| kind opsAndAtts)
    (declare (special |$pairlis| |$NRTslot1PredicateList| |$spadLibFT|
                  |$lisplibForm| |$profileCompiler| |$libFile|
                  |$lisplibSlot1| |$lisplibAncestors| |$lisplibParents|
                  |$lisplibAbbreviation| |$lisplibPredicates|
                  |$lisplibAttributes| |$lisplibVariableAlist|
                  |$lisplibSignatureAlist| |$lisplibSuperDomain|
                  |$formalMapVariableList| |$lisplibModemap|
                  |$lisplibModemapAlist| /editfile |$lisplibCategory|
                  |$lisplibKind| errors))
      (|lisplibWrite| "constructorForm"
        (|removeZeroOne| |$lisplibForm|) |$libFile|)
      (|lisplibWrite| "constructorKind"
        (setq kind (|removeZeroOne| |$lisplibKind|)) |$libFile|)
      (|lisplibWrite| "constructorModemap"
        (|removeZeroOne| |$lisplibModemap|) |$libFile|)
      (setq |$lisplibCategory| (or |$lisplibCategory| (cadar |$lisplibModemap|)))
      (|lisplibWrite| "constructorCategory" |$lisplibCategory| |$libFile|)
      (|lisplibWrite| "sourceFile" (|namestring| /editfile) |$libFile|)
      (|lisplibWrite| "modemaps"
        (|removeZeroOne| |$lisplibModemapAlist|) |$libFile|)
      (setq opsAndAtts
        (|getConstructorOpsAndAtts| |$lisplibForm| kind |$lisplibModemap|))
      (|lisplibWrite| "operationAlist"
        (|removeZeroOne| (car opsAndAtts)) |$libFile|)
      (when (eq kind '|category|)
        (setq |$pairlis|
          (loop for a in (rest |$lisplibForm|)
                for v in |$formalMapVariableList|
                collect (cons a v)))
          (setq |$NRTslot1PredicateList| nil)
          (|NRTgenInitialAttributeAlist| (cdr opsAndAtts)))
      (|lisplibWrite| "superDomain"
        (|removeZeroOne| |$lisplibSuperDomain|) |$libFile|)
      (|lisplibWrite| "signaturesAndLocals"
        (|removeZeroOne|

```



```

      (|mergeSignatureAndLocalVarAlists| |$lisplibSignatureAlist|
        |$lisplibVariableAlist|))
      |$libFile|)
(|lisplibWrite| "attributes"
  (|removeZeroOne| |$lisplibAttributes|) |$libFile|)
(|lisplibWrite| "predicates"
  (|removeZeroOne| |$lisplibPredicates|) |$libFile|)
(|lisplibWrite| "abbreviation" |$lisplibAbbreviation| |$libFile|)
(|lisplibWrite| "parents" (|removeZeroOne| |$lisplibParents|) |$libFile|)
(|lisplibWrite| "ancestors" (|removeZeroOne| |$lisplibAncestors|) |$libFile|)
(|lisplibWrite| "documentation" (|finalizeDocumentation|) |$libFile|)
(|lisplibWrite| "slot1Info" (|removeZeroOne| |$lisplibSlot1|) |$libFile|)
(when |$profileCompiler| (|profileWrite|))
(when (and |$lisplibForm| (null (cdr |$lisplibForm|)))
  (setf (get (car |$lisplibForm|) 'niladic) t))
(unless (eql errors 0)
  (|sayMSG| (list "    Errors in processing " kind " " libName ":"))
  (|sayMSG| (list "    not replacing " |$spadLibFT| " for" libName))))

```

defun getConstructorOpsAndAtts

[getCategoryOpsAndAtts p205]
 [getFunctorOpsAndAtts p208]

— defun getConstructorOpsAndAtts —

```

(defun |getConstructorOpsAndAtts| (form kind modemap)
  (if (eq kind '|category|)
    (|getCategoryOpsAndAtts| form)
    (|getFunctorOpsAndAtts| form modemap)))

```

defun getCategoryOpsAndAtts

[transformOperationAlist p206]
 [getSlotFromCategoryForm p206]
 [getSlotFromCategoryForm p206]

— defun getCategoryOpsAndAtts —

```

(defun |getCategoryOpsAndAtts| (catForm)
  (cons (|transformOperationAlist| (|getSlotFromCategoryForm| catForm 1))

```

```
(|getSlotFromCategoryForm| catForm 2)))
```

defun getSlotFromCategoryForm

```
[eval p??]
[take p??]
[systemErrorHere p??]
[$FormalMapVariableList p266]
```

— defun getSlotFromCategoryForm —

```
(defun |getSlotFromCategoryForm| (opargs index)
  (let (op argl u)
    (declare (special |$FormalMapVariableList|))
    (setq op (first opargs))
    (setq argl (rest opargs))
    (setq u
      (|eval| (cons op (mapcar 'mkq (take (|#| argl) |$FormalMapVariableList|)))))
    (if (null (vecp u))
      (|systemErrorHere| "getSlotFromCategoryForm")
      (elt u index))))
```

defun transformOperationAlist

This transforms the operationAlist which is written out onto LISPLIBs. The original form of this list is a list of items of the form:

```
((<op> <signature>) (<condition> (ELT $ n)))
```

The new form is an op-Alist which has entries

```
(<op> . signature-Alist)
```

where signature-Alist has entries

```
(<signature> . item)
```

where item has form

```
(<slotNumber> <condition> <kind>)
```

```

where <kind> =
  NIL => function
  CONST => constant ... and others

```

```

[member p??]
[keyedSystemError p??]
[assoc p??]
[lassq p??]
[insertAlist p??]
[$functionLocations p??]

```

— defun transformOperationAlist —

```

(defun |transformOperationAlist| (operationAlist)
  (let (op sig condition implementation eltEtc impOp kind u n signatureItem
        itemList newAlist)
    (declare (special |$functionLocations|))
    (setq newAlist nil)
    (dolist (item operationAlist)
      (setq op (caar item))
      (setq sig (cadar item))
      (setq condition (cadr item))
      (setq implementation (caddr item))
      (setq kind
        (cond
          ((and (consp implementation) (consp (qrest implementation))
                (consp (qcddr implementation))
                (eq (qcdddr implementation) nil)
                (progn (setq n (qthird implementation)) t)
                (|member| (setq eltEtc (qfirst implementation)) '(const elt)))
            eltEtc)
          ((consp implementation)
            (setq impOp (qfirst implementation))
            (cond
              ((eq impOp 'xlam) implementation)
              ((|member| impOp '(const |Subsumed|)) impOp)
              (t (|keyedSystemError| 's2il0025 (list impOp))))
            ((eq implementation '|mkRecord|) '|mkRecord|)
            (t (|keyedSystemError| 's2il0025 (list implementation)))))
      (when (setq u (|assoc| (list op sig) |$functionLocations|))
        (setq n (cons n (cdr u))))
      (setq signatureItem
        (if (eq kind 'elt)
          (if (eq condition t)
            (list sig n)
            (list sig n condition))
          (list sig n condition kind)))
      (setq itemList (cons signatureItem (lassq op newAlist)))
      (setq newAlist (|insertAlist| op itemList newAlist)))

```

```
newAlist))
```

defun getFunctorOpsAndAtts

```
[transformOperationAlist p206]
[getSlotFromFunctor p208]
```

— defun getFunctorOpsAndAtts —

```
(defun |getFunctorOpsAndAtts| (form modemap)
  (cons (|transformOperationAlist| (|getSlotFromFunctor| form 1 modemap))
        (|getSlotFromFunctor| form 2 modemap)))
```

defun getSlotFromFunctor

```
[compMakeCategoryObject p208]
[systemErrorHere p??]
[$e p??]
[$lisplibOperationAlist p??]
```

— defun getSlotFromFunctor —

```
(defun |getSlotFromFunctor| (arg1 slot arg2)
  (declare (ignore arg1))
  (let (tt)
    (declare (special |$e| |$lisplibOperationAlist|))
    (cond
      ((eql slot 1) |$lisplibOperationAlist|)
      (t
       (setq tt (or (|compMakeCategoryObject| (cadar arg2) |$e|)
                    (|systemErrorHere| "getSlotFromFunctor"))
              (elt (car tt) slot)))))
```

defun compMakeCategoryObject

```
[isCategoryForm p??]
[mkEvaluableCategoryForm p178]
```

```
[$e p??]
[$Category p??]
```

— **defun compMakeCategoryObject** —

```
(defun |compMakeCategoryObject| (c |$e|)
  (declare (special |$e|))
  (let (u)
    (declare (special |$Category|))
    (cond
      ((null (|isCategoryForm| c |$e|)) nil)
      ((setq u (|mkEvaluableCategoryForm| c)) (list (|eval| u) |$Category| |$e|))
      (t nil))))
```

—————

defun mergeSignatureAndLocalVarAlists

```
[lassoc p??]
```

— **defun mergeSignatureAndLocalVarAlists** —

```
(defun |mergeSignatureAndLocalVarAlists| (signatureAlist localVarAlist)
  (loop for item in signatureAlist
    collect
      (cons (first item)
            (cons (rest item)
                  (lassoc (first item) localVarAlist)))))
```

—————

defun lisplibWrite

```
[rewrite128 p??]
[$lisplib p??]
```

— **defun lisplibWrite** —

```
(defun |lisplibWrite| (prop val filename)
  (declare (special $lisplib))
  (when $lisplib (|rewrite| prop val filename)))
```

—————

defun isCategoryPackageName

```
[pname p??]
[maxindex p??]
[char p??]
```

— defun isCategoryPackageName —

```
(defun |isCategoryPackageName| (nam)
  (let (p)
    (setq p (pname (lop0f| nam)))
    (equal (elt p (maxindex p)) (|char| '&))))
```

—————

defun NRTgetLookupFunction

Compute the lookup function (complete or incomplete) [sublis p??]

```
[NRTextendsCategory1 p??]
[getExportCategory p??]
[sayBrightly p??]
[sayBrightlyNT p??]
[bright p??]
[form2String p??]
[$why p??]
[$why p??]
[$pairlis p??]
```

— defun NRTgetLookupFunction —

```
(defun |NRTgetLookupFunction| (domform exCategory addForm)
  (let (|$why| extends u msg v)
    (declare (special |$why| |$pairlis|))
    (setq domform (sublis |$pairlis| domform))
    (setq addForm (sublis |$pairlis| addForm))
    (setq |$why| nil)
    (cond
      ((atom addForm) '|lookupComplete|)
      (t
       (setq extends
         (|NRTextendsCategory1| domform exCategory (|getExportCategory| addForm)))
       (cond
         ((null extends)
          (setq u (car |$why|))
          (setq msg (cadr |$why|))
          (setq v (cddr |$why|))
```

```

(|sayBrightly|
  "-----non extending category-----")
(|sayBrightlyNT|
  (cons ".."
    (append (|bright| (|form2String| domform)) (list '|of cat |))))
(print u)
(|sayBrightlyNT| (|bright| msg))
(if v (print (car v)) (terpri)))
(if extends
  '|lookupIncomplete|
  '|lookupComplete|))))

```

defun NRTgetLocalIndex

```

[NRTassocIndex p342]
[NRTaddInner p??]
[compOrCroak p588]
[rplaca p??]
[$NRTaddForm p??]
[$formalArgList p??]
[$NRTdeltaList p??]
[$NRTdeltaListComp p??]
[$NRTdeltaLength p??]
[$NRTbase p??]
[$EmptyMode p172]
[$e p??]

```

— defun NRTgetLocalIndex —

```

(defun |NRTgetLocalIndex| (item)
  (let (k value saveNRTdeltaListComp saveIndex compEntry)
    (declare (special |$e| |$EmptyMode| |$NRTdeltaLength| |$NRTbase|
                      |$NRTdeltaListComp| |$NRTdeltaList| |$formalArgList|
                      |$NRTaddForm|))
    (cond
      ((setq k (|NRTassocIndex| item)) k)
      ((equal item |$NRTaddForm|) 5)
      ((eq item '$) 0)
      ((eq item '$$) 2)
      (t
       (when (member item |$formalArgList|) (setq value item))
       (cond
         ((and (atom item) (null (member item '($ $))) (null value))
          (setq |$NRTdeltaList|
                (cons (cons '|domain| (cons (|NRTaddInner| item) value))

```

```

      |$NRTdeltaList|))
    (setq |$NRTdeltaListComp| (cons item |$NRTdeltaListComp|))
    (setq |$NRTdeltaLength| (1+ |$NRTdeltaLength|))
    (1- (+ |$NRTbase| |$NRTdeltaLength|)))
  (t
    (setq |$NRTdeltaList|
      (cons (cons '|domain| (cons (|NRTaddInner| item) value))
        |$NRTdeltaList|))
    (setq saveNRTdeltaListComp
      (setq |$NRTdeltaListComp| (cons nil |$NRTdeltaListComp|)))
    (setq saveIndex (+ |$NRTbase| |$NRTdeltaLength|))
    (setq |$NRTdeltaLength| (1+ |$NRTdeltaLength|))
    (setq compEntry (car (|compOrCroak| item |$EmptyMode| |$e|)))
    (rplaca saveNRTdeltaListComp compEntry)
    saveIndex))))))

```

defun augmentLisplibModemapsFromFunctor

```

[formal2Pattern p214]
[mkAlistOfExplicitCategoryOps p189]
[allLASSOCs p213]
[member p??]
[mkDatabasePred p214]
[mkpf p??]
[listOfPatternIds p??]
[interactiveModemapForm p191]
[$lisplibModemapAlist p??]
[$PatternVariableList p??]
[$e p??]
[$lisplibModemapAlist p??]
[$e p??]

```

— defun augmentLisplibModemapsFromFunctor —

```

(defun |augmentLisplibModemapsFromFunctor| (form opAlist signature)
  (let (argl nonCategorySigAlist op pred sel predList sig predp z skip modemap)
    (declare (special |$lisplibModemapAlist| |$PatternVariableList| |$e|))
    (setq form (|formal2Pattern| form))
    (setq argl (cdr form))
    (setq opAlist (|formal2Pattern| opAlist))
    (setq signature (|formal2Pattern| signature))
    ; We are going to be EVALing categories containing these pattern variables
    (loop for u in form for v in signature
      do (when (member u |$PatternVariableList|)

```



```

      (setq |$e| (|put| u '|mode| v |$e|)))
(when
  (setq nonCategorySigAlist (|mkAlistOfExplicitCategoryOps| (CAR signature)))
  (loop for entry in opAlist
    do
      (setq op (caar entry))
      (setq sig (cadar entry))
      (setq pred (cadr entry))
      (setq sel (caddr entry))
      (when
        (let (result)
          (loop for catSig in (|allLASSOCs| op nonCategorySigAlist)
            do (setq result (or result (|member| sig catSig))))
          result)
        (setq skip (when (and argl (contained '$ (cdr sig))) 'skip))
        (setq sel (subst form '$ sel :test #'equal))
        (setq predList
          (loop for a in argl for m in (rest signature)
            when (|member| a |$PatternVariableList|)
            collect (list a m)))
        (setq sig (subst form '$ sig :test #'equal))
        (setq predp
          (mkpf
            (cons pred (loop for y in predList collect (|mkDatabasePred| y)))
            'and))
        (setq z (|listOfPatternIds| predList))
        (when (some #'(lambda (u) (null (member u z))) argl)
          (|sayMSG| (list "cannot handle modemap for " op "by pattern match"))
          (setq skip 'skip))
        (setq modemap (list (cons form sig) (cons predp (cons sel skip))))
        (setq |$lisplibModemapAlist|
          (cons
            (cons op (|interactiveModemapForm| modemap))
            |$lisplibModemapAlist|))))))

```

defun allLASSOCs

— defun allLASSOCs —

```

(defun |allLASSOCs| (op alist)
  (loop for value in alist
    when (equal (car value) op)
    collect value))

```

defun formal2Pattern

```
[sublis p??]
[pairList p??]
[$PatternVariableList p??]
```

— defun formal2Pattern —

```
(defun |formal2Pattern| (x)
  (declare (special |$PatternVariableList|))
  (sublis (|pairList| |$FormalMapVariableList| (cdr |$PatternVariableList|)) x))
```

defun mkDatabasePred

```
[isCategoryForm p??]
[$e p??]
```

— defun mkDatabasePred —

```
(defun |mkDatabasePred| (arg)
  (let (a z)
    (declare (special |$e|))
    (setq a (car arg))
    (setq z (cadr arg))
    (if (|isCategoryForm| z |$e|)
        (list '|ofCategory| a z)
        (list '|ofType| a z))))
```

defun disallowNilAttribute

— defun disallowNilAttribute —

```
(defun |disallowNilAttribute| (x)
  (loop for y in x when (and (car y) (not (eq (car y) '|nil|)))
    collect y))
```

defun bootStrapError

```
[mkq p??]
[namestring p??]
[mkDomainConstructor p??]
```

— defun bootStrapError —

```
(defun |bootStrapError| (functorForm sourceFile)
  (list 'cond
    (list '|$bootStrapMode|
      (list 'vector (|mkDomainConstructor| functorForm) nil nil nil nil nil))
    (list ''t
      (list '|systemError|
        (list 'list ''|%b| (MKQ (CAR functorForm)) ''|%d| "from" ''|%b|
          (mkq (|namestring| sourceFile)) ''|%d| "needs to be compiled")))))
```

—————

defun reportOnFunctorCompilation

```
[displayMissingFunctions p216]
[sayBrightly p??]
[displaySemanticErrors p??]
[displayWarnings p??]
[addStats p??]
[normalizeStatAndStringify p??]
[$op p??]
[$functorStats p??]
[$functionStats p??]
[$warningStack p??]
[$semanticErrorStack p??]
```

— defun reportOnFunctorCompilation —

```
(defun |reportOnFunctorCompilation| ()
  (declare (special |$op| |$functorStats| |$functionStats|
    |$warningStack| |$semanticErrorStack|))
  (|displayMissingFunctions|)
  (when |$semanticErrorStack| (|sayBrightly| " "))
  (|displaySemanticErrors|)
  (when |$warningStack| (|sayBrightly| " "))
  (|displayWarnings|)
  (setq |$functorStats| (|addStats| |$functorStats| |$functionStats|))
  (|sayBrightly|
    (cons ''|%l|
```

```

      (append (|bright| " Cumulative Statistics for Constructor")
        (list |$op|)))
    (|sayBrightly|
      (cons " Time:"
        (append (|bright| (|normalizeStatAndStringify| (second |$functorStats|))
          (list "seconds"))))
    (|sayBrightly| " ")
    '|done|)

```

defun displayMissingFunctions

```

[member p??]
[getmode p??]
[sayBrightly p??]
[bright p??]
[formatUnabbreviatedSig p??]
[$env p??]
[$formalArgList p??]
[$CheckVectorList p??]

```

— defun displayMissingFunctions —

```

(defun |displayMissingFunctions| ()
  (let (i loc exp)
    (declare (special |$env| |$formalArgList| |$CheckVectorList|))
    (unless |$CheckVectorList|
      (setq loc nil)
      (setq exp nil)
      (loop for cvl in |$CheckVectorList| do
        (unless (cdr cvl)
          (if (and (null (|member| (caar cvl) |$formalArgList|))
            (consp (|getmode| (caar cvl) |$env|))
            (eq (qfirst (|getmode| (caar cvl) |$env|)) '|Mapping|))
            (push (list (caar cvl) (cadar cvl)) loc)
            (push (list (caar cvl) (cadar cvl)) exp))))
      (when loc
        (|sayBrightly| (cons '|%l| (|bright| " Missing Local Functions:"))
          (setq i 0)
          (loop for item in loc do
            (|sayBrightly|
              (cons " [" (cons (incf i) (cons "]"
                (append (|bright| (first item))
                  (cons '|:| (|formatUnabbreviatedSig| (second item))))))))))
      (when exp
        (|sayBrightly| (cons '|%l| (|bright| " Missing Exported Functions:"))

```

```
(setq i 0)
(loop for item in exp do
  (|sayBrightly|
    (cons "      [" (cons (incf i) (cons "]"
      (append (|bright| (first item))
        (cons '|: | (|formatUnabbreviatedSig| (second item))))))))))
```

defun makeFunctorArgumentParameters

```
[assq p??]
[isCategoryForm p??]
[genDomainViewList0 p219]
[union p??]
[$ConditionalOperators p??]
[$alternateViewList p??]
[$forceAdd p??]
```

— defun makeFunctorArgumentParameters —

```
(defun |makeFunctorArgumentParameters| (argl sigl target)
  (labels (
    (augmentSig (s ss)
      (let (u)
        (declare (special |$ConditionalOperators|))
        (if ss
          (progn
            (loop for u in ss do (push (rest u) |$ConditionalOperators|))
            (if (and (consp s) (eq (qfirst s) '|Join|))
              (progn
                (if (setq u (assq 'category ss))
                  (subst (append u ss) u s :test #'equal)
                  (cons '|Join|
                    (append (rest s) (list (cons 'category (cons '|package| ss)))))))
              (list '|Join| s (cons 'category (cons '|package| ss))))))
          s)))
    (fn (a s)
      (declare (special |$CategoryFrame|))
      (if (|isCategoryForm| s |$CategoryFrame|)
        (if (and (consp s) (eq (qfirst s) '|Join|))
          (|genDomainViewList0| a (rest s))
          (list (|genDomainView| a s '|getDomainView|)))
        (list a)))
    (findExtras (a target)
      (cond
        ((and (consp target) (eq (qfirst target) '|Join|))
```

```

      (reduce #'|union|
        (loop for x in (qrest target)
          collect (findExtras a x))))
    ((and (consp target) (eq (qfirst target) 'category))
      (reduce #'|union|
        (loop for x in (qcddr target)
          collect (findExtras1 a x))))))
  (findExtras1 (a x)
    (cond
      ((and (consp x) (or (eq (qfirst x) 'and) (eq (qfirst x) 'or))
        (reduce #'|union|
          (loop for y in (rest x) collect (findExtras1 a y))))
        ((and (consp x) (eq (qfirst x) 'if)
          (consp (qrest x)) (consp (qcddr x))
          (consp (qcdddr x))
          (eq (qcddddr x) nil))
          (|union| (findExtrasP a (second x))
            (|union|
              (findExtras1 a (third x))
              (findExtras1 a (fourth x)))))))
    (findExtrasP (a x)
      (cond
        ((and (consp x) (or (eq (qfirst x) 'and) (eq (qfirst x) 'or))
          (reduce #'|union|
            (loop for y in (rest x) collect (findExtrasP a y))))
          ((and (consp x) (eq (qfirst x) '|has|)
            (consp (qrest x)) (consp (qcddr x))
            (consp (qcdddr x))
            (eq (qcddddr x) nil))
            (|union| (findExtrasP a (second x))
              (|union|
                (findExtras1 a (third x))
                (findExtras1 a (fourth x))))))
          ((and (consp x) (eq (qfirst x) '|has|)
            (consp (qrest x)) (equal (qsecond x) a)
            (consp (qcddr x))
            (eq (qcdddr x) nil)
            (consp (qthird x))
            (eq (qcaaddr x) 'signature))
            (list (third x))))))
    )
  (let (|$alternateViewList| |$forceAdd| |$ConditionalOperators|)
    (declare (special |$alternateViewList| |$forceAdd| |$ConditionalOperators|))
    (setq |$alternateViewList| nil)
    (setq |$forceAdd| t)
    (setq |$ConditionalOperators| nil)
    (mapcar #'reduce
      (loop for a in argl for s in sigl do
        (fn a (augmentSig s (findExtras a target))))))

```

defun genDomainViewList0

[getDomainViewList p??]

— defun genDomainViewList0 —

```
(defun |genDomainViewList0| (id catlist)
  (|genDomainViewList| id catlist t))
```

defun genDomainViewList[isCategoryForm p??]
[genDomainView p219]
[genDomainViewList p219]
[\$EmptyEnvironment p??]

— defun genDomainViewList —

```
(defun |genDomainViewList| (id catlist firsttime)
  (declare (special |$EmptyEnvironment|) (ignore firsttime))
  (cond
    ((null catlist) nil)
    ((and (consp catlist) (eq (qrest catlist) nil)
      (null (|isCategoryForm| (first catlist) |$EmptyEnvironment|)))
     nil)
    (t
     (cons
      (|genDomainView| id (first catlist) '|getDomainView|)
      (|genDomainViewList| id (rest catlist) nil))))))
```

defun genDomainView[genDomainOps p220]
[augModemapsFromCategory p260]
[mkDomainConstructor p??]

```
[member p??]
[$e p??]
[$getDomainCode p??]
```

— defun genDomainView —

```
(defun |genDomainView| (name c viewSelector)
  (let (code cd)
    (declare (special |$getDomainCode| |$e|))
    (cond
      ((and (consp c) (eq (qfirst c) 'category) (consp (qrest c)))
        (|genDomainOps| name name c))
      (t
        (setq code
          (if (and (consp c) (eq (qfirst c) '|SubsetCategory|)
            (consp (qrest c)) (consp (qcddr c))
            (eq (qcdddr c) nil))
          (second c)
          c))
        (setq |$e| (|augModemapsFromCategory| name nil c |$e|))
        (setq cd
          (list 'let name (list viewSelector name (|mkDomainConstructor| code))))
        (unless (|member| cd |$getDomainCode|)
          (setq |$getDomainCode| (cons cd |$getDomainCode|)))
        name))))
```

—————

defun genDomainOps

```
[getOperationAlist p265]
[substNames p266]
[mkq p??]
[mkDomainConstructor p??]
[addModemap p269]
[$e p??]
[$ConditionalOperators p??]
[$getDomainCode p??]
```

— defun genDomainOps —

```
(defun |genDomainOps| (viewName dom cat)
  (let (siglist oplist cd i)
    (declare (special |$e| |$ConditionalOperators| |$getDomainCode|))
    (setq oplist (|getOperationAlist| dom dom cat))
    (setq siglist (loop for lst in oplist collect (first lst)))
```



```

(setq oplist (|substNames| dom viewName dom oplist))
(setq cd
  (list 'let viewName
    (list '|mkOpVec| dom
      (cons 'list
        (loop for opsig in siglist
          collect
            (list 'list (mkq (first opsig))
              (cons 'list
                (loop for mode in (rest opsig)
                  collect (|mkDomainConstructor| mode))))))))))
(setq |$getDomainCode| (cons cd |$getDomainCode|))
(setq i 0)
(loop for item in oplist do
  (if (|member| (first item) |$ConditionalOperators|)
    (setq |$e| (|addModemap| (caar item) dom (cadar item) nil
      (list 'elt viewName (incf i)) |$e|))
    (setq |$e| (|addModemap| (caar item) dom (cadar item) (second item)
      (list 'elt viewName (incf i)) |$e|))))
viewName))

```

defun mkOpVec

```

[getPrincipalView p??]
[getOperationAlistFromLisplib p??]
[opOf p??]
[length p??]
[assq p??]
[assoc p??]
[sublis p??]
[AssocBarGensym p222]
[$FormalMapVariableList p266]
[Undef p??]

```

— defun mkOpVec —

```

(defun |mkOpVec| (dom siglist)
  (let (substargs oplist ops u noplist i tmp1)
    (declare (special |$FormalMapVariableList| |Undef|))
    (setq dom (|getPrincipalView| dom))
    (setq substargs
      (cons (cons '$ (elt dom 0))
        (loop for a in |$FormalMapVariableList| for x in (rest (elt dom 0))
          collect (cons a x))))
    (setq oplist (|getOperationAlistFromLisplib| (|opOf| (elt dom 0))))
  )

```

```

(setq ops (make-array (|#| siglist)))
(setq i -1)
(loop for opSig in siglist do
  (incf i)
  (setq u (assq (first opSig) oplist))
  (setq tmp1 (|assoc| (second opSig) u))
  (cond
    ((and (consp tmp1) (consp (qrest tmp1))
          (consp (qcddr tmp1)) (consp (qcdddr tmp1))
          (eq (qcdddr tmp1) nil)
          (eq (qfourth tmp1) 'elt))
     (setelt ops i (elt dom (second tmp1))))
    (t
     (setq noplist (sublis substargs u))
     (setq tmp1
      (|AssocBarGensym|
       (subst (elt dom 0) '$ (second opSig) :test #'equal) noplist))
     (cond
       ((and (consp tmp1) (consp (qrest tmp1)) (consp (qcddr tmp1))
             (consp (qcdddr tmp1))
             (eq (qcdddr tmp1) nil)
             (eq (qfourth tmp1) 'elt))
        (setelt ops i (elt dom (second tmp1))))
       (t
        (setelt ops i (cons |Undef| (cons (list (elt dom 0) i) opSig)))))))
  ops))

```

defun AssocBarGensym

[EqualBarGensym p244]

— defun AssocBarGensym —

```

(defun |AssocBarGensym| (key z)
  (loop for x in z
    do (when (and (consp x) (|EqualBarGensym| key (car x))) (return x))))

```

defun orderByDependency

```

[say p??]
[userError p??]

```

```
[intersection p??]
[member p??]
[remdup p??]
```

— defun orderByDependency —

```
(defun |orderByDependency| (vl dl)
  (let (selfDependents fatalError newl orderedVarList vlp dlp)
    (setq selfDependents
      (loop for v in vl for d in dl
        when (member v d)
        collect v))
    (loop for v in vl for d in dl
      when (member v d)
      do (say v "depends on itself")
        (setq fatalError t))
    (cond
      (fatalError (|userError| "Parameter specification error"))
      (t
        (loop until (null vl) do
          (setq newl
            (loop for v in vl for d in dl
              when (null (|intersection| d vl))
              collect v))
          (if (null newl)
            (setq vl nil) ; force loop exit
            (progn
              (setq orderedVarList (append newl orderedVarList))
              (setq vlp (setdifference vl newl))
              (setq dlp
                (loop for x in vl for d in dl
                  when (|member| x vlp)
                  collect (setdifference d newl)))
              (setq vl vlp)
              (setq dl dlp))))
          (when (and newl orderedVarList) (remdup (nreverse orderedVarList)))))))
```

—

6.2 Code optimization routines

defun optimizeFunctionDef

```
[rplac p??]
[sayBrightlyI p??]
[optimize p225]
```

```
[pp p??]
[bright p??]
[$reportOptimization p??]
```

— defun optimizeFunctionDef —

```
(defun |optimizeFunctionDef| (def)
  (labels (
    (fn (x g)
      (cond
        ((and (consp x) (eq (qfirst x) 'throw) (consp (qrest x))
          (equal (qsecond x) g))
          (|rplac| (car x) 'return)
          (|rplac| (cdr x)
            (replaceThrowByReturn (qcddr x) g)))
        ((atom x) nil)
        (t
          (replaceThrowByReturn (car x) g)
          (replaceThrowByReturn (cdr x) g))))
    (replaceThrowByReturn (x g)
      (fn x g)
        x)
    (removeTopLevelCatch (body)
      (if (and (consp body) (eq (qfirst body) 'catch) (consp (qrest body))
        (consp (qcddr body)) (eq (qcdddr body) nil))
        (removeTopLevelCatch
          (replaceThrowByReturn
            (qthird body) (qsecond body)))
          body)))
  (let (defp name slamOrLam args body bodyp)
    (declare (special |$reportOptimization|))
    (when |$reportOptimization|
      (|sayBrightlyI| (|bright| "Original LISP code:"))
      (|pp| def))
    (setq defp (|optimize| (copy def)))
    (when |$reportOptimization|
      (|sayBrightlyI| (|bright| "Optimized LISP code:"))
      (|pp| defp)
      (|sayBrightlyI| (|bright| "Final LISP code:")))
    (setq name (car defp))
    (setq slamOrLam (caadr defp))
    (setq args (cadadr defp))
    (setq body (car (cddadr defp)))
    (setq bodyp (removeTopLevelCatch body))
    (list name (list slamOrLam args bodyp))))
```

defun optimize

[optimize p225]
 [say p??]
 [prettyprint p??]
 [rplac p??]
 [optIF2COND p227]
 [getl p??]
 [subrname p228]

— defun optimize —

```
(defun |optimize| (x)
  (labels (
    (opt (x)
      (let (argl body a y op)
        (cond
          ((atom x) nil)
          ((eq (setq y (car x)) 'quote) nil)
          ((eq y 'closedfn) nil)
          ((and (consp y) (consp (qfirst y)) (eq (qcaar y) 'xlam)
                (consp (qcddar y)) (consp (qcddar y))
                (eq (qcdddar y) nil))
           (setq argl (qcadar y))
           (setq body (qcaddar y))
           (setq a (qrest y))
           (|optimize| (cdr x))
           (cond
            ((eq argl '|ignore|) (rplac (car x) body))
            (t
             (when (null (<= (length argl) (length a)))
               (say "length mismatch in XLAM expression")
               (prettyprint y))
             (rplac (car x)
                    (|optimize|
                     (|optXLAMCond|
                      (sublis (|pairList| argl a) body)))))))
          ((atom y)
           (|optimize| (cdr x))
           (cond
            ((eq y '|true|) (rplac (car x) ''T))
            ((eq y '|false|) (rplac (car x) nil))))
          ((eq (car y) 'if)
           (rplac (car x) (|optIF2COND| y))
           (setq y (car x))
           (when (setq op (getl (|subrname| (car y)) 'optimize))
             (|optimize| (cdr x))
             (rplac (car x) (funcall op (|optimize| (car x)))))
           ((setq op (getl (|subrname| (car y)) 'optimize))
```

```

      (|optimize| (cdr x))
      (rplac (car x) (funcall op (|optimize| (car x)))))
    (t
      (rplac (car x) (|optimize| (car x)))
      (|optimize| (cdr x)))))
  (opt x)
  x))

```

defun optXLAMCond

[optCONDtail p226]
 [optPredicateIfTrue p227]
 [optXLAMCond p226]
 [rplac p??]

— defun optXLAMCond —

```

(defun |optXLAMCond| (x)
  (cond
    ((and (consp x) (eq (qfirst x) 'cond) (consp (qrest x))
          (consp (qsecond x)) (consp (qcdadr x))
          (eq (qcddadr x) nil))
      (if (|optPredicateIfTrue| (qcaadr x))
          (qcadadr x)
          (cons 'cond (cons (qsecond x) (|optCONDtail| (qcddr x)))))
    ((atom x) x)
    (t
      (rplac (car x) (|optXLAMCond| (car x)))
      (rplac (cdr x) (|optXLAMCond| (cdr x)))
      x)))

```

defun optCONDtail

[optCONDtail p226]
 [\$true p??]

— defun optCONDtail —

```

(defun |optCONDtail| (z)
  (declare (special |$true|))
  (when z

```

```
(cond
  ((|optPredicateIfTrue| (caar z)) (list (list |$true| (cadar z))))
  ((null (cdr z)) (list (car z) (list |$true| (list '|CondError|))))
  (t (cons (car z) (|optCONDtail| (cdr z))))))
```

defvar \$BasicPredicates

If these predicates are found in an expression the code optimizer routine `optPredicateIfTrue` then `optXLAM` will replace the call with the argument. This is used for predicates that test the type of their argument so that, for instance, a call to `integerp` on an integer will be replaced by that integer if it is true. This represents a simple kind of compile-time type evaluation.

— **initvars** —

```
(defvar |$BasicPredicates| '(integerp stringp floatp))
```

defun optPredicateIfTrue

[`$BasicPredicates` [p227](#)]

— **defun optPredicateIfTrue** —

```
(defun |optPredicateIfTrue| (p)
  (declare (special |$BasicPredicates|))
  (cond
    ((and (consp p) (eq (qfirst p) 'quote)) T)
    ((and (consp p) (consp (qrest p)) (eq (qcddr p) nil)
      (member (qfirst p) |$BasicPredicates|) (funcall (qfirst p) (qsecond p)))
     t)
    (t nil)))
```

defun optIF2COND

[`optIF2COND` [p227](#)]

[`$true p??`]

— **defun optIF2COND** —

```
(defun |optIF2COND| (arg)
  (let (a b c)
    (declare (special |$true|))
    (setq a (cadr arg))
    (setq b (caddr arg))
    (setq c (cadddr arg))
    (cond
      ((eq b '|noBranch|) (list 'cond (list (list 'null a ) c)))
      ((eq c '|noBranch|) (list 'cond (list a b)))
      ((and (consp c) (eq (qfirst c) 'if))
        (cons 'cond (cons (list a b) (cdr (|optIF2COND| c)))))
      ((and (consp c) (eq (qfirst c) 'cond))
        (cons 'cond (cons (list a b) (qrest c)))))
    (t
      (list 'cond (list a b) (list |$true| c))))))
```

defun subname

```
[identp p??]
[compiled-function-p p??]
[mbpip p??]
[bpname p??]
```

— defun subname —

```
(defun |subname| (u)
  (cond
    ((identp u) u)
    ((or (compiled-function-p u) (mbpip u)) (bpname u))
    (t nil)))
```

Special case optimizers

Optimization functions are called through the OPTIMIZE property on the symbol property list. The current list is:

call	optCall
seq	optSEQ
eq	optEQ
minus	optMINUS
qsminus	optQSMINUS

-	opt-
lessp	optLESSP
spadcall	optSPADCALL
	optSuchthat
catch	optCatch
cond	optCond
mkRecord	optMkRecord
recordelt	optRECORDELT
setrecordelt	optSETRECORDELT
recordcopy	optRECORDCOPY

Be aware that there are case-sensitivity issues. When found in the s-expression, each symbol in the left column will call a custom optimization routine in the right column. The optimization routines are below. Note that each routine has a special chunk in postvars using eval-when to set the property list at load time.

These optimizations are done destructively. That is, they modify the function in-place using rplac.

Not all of the optimization routines are called through the property list. Some are called only from other optimization routines, e.g. optPackageCall.

defplist optCall

— postvars —

```
(eval-when (eval load)
  (setf (get '|call| 'optimize) '|optCall|))
```

—————

defun Optimize “call” expressions

```
[optimize p225]
[rplac p??]
[optPackageCall p230]
[optCallSpecially p231]
[systemErrorHere p??]
[$QuickCode p??]
[$bootStrapMode p??]
```

— defun optCall —

```
(defun |optCall| (x)
  (let (u tmp1 fn a name q r n w)
```

```

(declare (special |$QuickCode| |$bootStrapMode|))
(setq u (cdr x))
(setq x (|optimize| (list u)))
(cond
  ((atom (car x)) (car x))
  (t
   (setq tmp1 (car x))
   (setq fn (car tmp1))
   (setq a (cdr tmp1))
   (cond
    ((atom fn) (rplac (cdr x) a) (rplac (car x) fn))
    ((and (consp fn) (eq (qfirst fn) 'pac)) (|optPackageCall| x fn a))
    ((and (consp fn) (eq (qfirst fn) '|applyFun|)
          (consp (qrest fn)) (eq (qcddr fn) nil))
     (setq name (qsecond fn))
     (rplac (car x) 'spadcall)
     (rplac (cdr x) (append a (cons name nil))))
    x)
   ((and (consp fn) (consp (qrest fn)) (consp (qcddr fn))
        (eq (qcddr fn) nil)
        (member (qfirst fn) '(elt qrefelt const)))
    (setq q (qfirst fn))
    (setq r (qsecond fn))
    (setq n (qthird fn))
    (cond
     ((and (null |$bootStrapMode|) (setq w (|optCallSpecially| q x n r)))
      w)
     ((eq q 'const)
      (list '|spadConstant| r n))
     (t
      (rplac (car x) 'spadcall)
      (when |$QuickCode| (rplaca fn 'qrefelt))
      (rplac (cdr x) (append a (list fn)))
      x)))
   (t (|systemErrorHere| "optCall"))))))

```

defun optPackageCall

```

[rplaca p??]
[rplacd p??]

```

— defun optPackageCall —

```

(defun |optPackageCall| (x arg2 arglist)
  (let (packageVariableOrForm functionName)

```

```

(setq packageVariableOrForm (second arg2))
(setq functionName (third arg2))
(rplaca x functionName)
(rplacd x (append arglist (list packageVariableOrForm)))
x))

```

defun optCallSpecially

```

[lassoc p??]
[kar p??]
[get p??]
[opOf p??]
[optSpecialCall p232]
[$specialCaseKeyList p??]
[$getDomainCode p??]
[$optimizableConstructorNames p??]
[$e p??]

```

— defun optCallSpecially —

```

(defun |optCallSpecially| (q x n r)
  (declare (ignore q))
  (labels (
    (lookup (a z)
      (let (zp)
        (when z
          (setq zp (car z))
          (setq z (cdr x))
          (if (and (consp zp) (eq (qfirst zp) 'let) (consp (qrest zp))
              (equal (qsecond zp) a) (consp (qcddr zp)))
              (qthird zp)
              (lookup a z))))))
    (let (tmp1 op y prop yy)
      (declare (special |$specialCaseKeyList| |$getDomainCode| |$e|
                        |$optimizableConstructorNames|))
      (cond
        ((setq y (lassoc r |$specialCaseKeyList|))
         (|optSpecialCall| x y n))
        ((member (kar r) |$optimizableConstructorNames|)
         (|optSpecialCall| x r n))
        ((and (setq y (|get| r '|value| |$e|))
              (member (|opOf| (car y)) |$optimizableConstructorNames|))
         (|optSpecialCall| x (car y) n))
        ((and (setq y (lookup r |$getDomainCode|))
              (progn

```

```

      (setq tmp1 y)
      (setq op (first tmp1))
      (setq y (second tmp1))
      (setq prop (third tmp1))
      tmp1)
    (setq yy (lassoc y |$specialCaseKeyList|)))
  (|optSpecialCall| x (list op yy prop) n))
  (t nil))))

```

defun optSpecialCall

```

[|optCallEval| p233]
[function p??]
[keyedSystemError p??]
[mkq p??]
[getl p??]
[compileTimeBindingOf p233]
[rplac p??]
[optimize p225]
[rplacw p??]
[rplaca p??]
[$QuickCode p??]
[$Undef p??]

```

— defun optSpecialCall —

```

(defun |optSpecialCall| (x y n)
  (let (yval args tmp1 fn a)
    (declare (special |$QuickCode| |Undef|))
    (setq yval (|optCallEval| y))
    (cond
      ((eq (caaar x) 'const)
        (cond
          ((equal (kar (elt yval n)) (|function| |Undef|))
            (|keyedSystemError| 'S2GE0016
              (list "optSpecialCall" "invalid constant"))))
          (t (mkq (elt yval n)))))
      ((setq fn (getl (|compileTimeBindingOf| (car (elt yval n))) '|SPADreplace|))
        (|rplac| (cdr x) (cdar x))
        (|rplac| (car x) fn)
        (when (and (consp fn) (eq (qfirst fn) 'xlam))
          (setq x (car (|optimize| (list x)))))
        (if (and (consp x) (eq (qfirst x) 'equal) (progn (setq args (qrest x)) t))
          (rplacw x (def-equal args))
          x))

```

```
(t
  (setq tmp1 (car x))
  (setq fn (car tmp1))
  (setq a (cdr tmp1))
  (rplac (car x) 'spadcall)
  (when |$QuickCode| (rplaca fn 'qrefelt))
  (rplac (cdr x) (append a (list fn)))
  x)))
```

defun compileTimeBindingOf

```
[bpiname p??]
[keyedSystemError p??]
[moan p??]
```

— defun compileTimeBindingOf —

```
(defun |compileTimeBindingOf| (u)
  (let (name)
    (cond
      ((null (setq name (bpiname u)))
        (|keyedSystemError| 'S2000001 (list u)))
      ((eq name '|Undef|)
        (moan "optimiser found unknown function"))
      (t name))))
```

defun optCallEval

```
[List p??]
[Integer p??]
[Vector p??]
[PrimititveArray p??]
[FactoredForm p??]
[Matrix p??]
[eval p??]
```

— defun optCallEval —

```
(defun |optCallEval| (u)
  (cond
```

```

((and (consp u) (eq (qfirst u) '|List|))
  (|List| (|Integer|)))
((and (consp u) (eq (qfirst u) '|Vector|))
  (|Vector| (|Integer|)))
((and (consp u) (eq (qfirst u) '|PrimitiveArray|))
  (|PrimitiveArray| (|Integer|)))
((and (consp u) (eq (qfirst u) '|FactoredForm|))
  (|FactoredForm| (|Integer|)))
((and (consp u) (eq (qfirst u) '|Matrix|))
  (|Matrix| (|Integer|)))
(t
  (|eval| u)))

```

defplist optSEQ

— postvars —

```

(eval-when (eval load)
  (setf (get 'seq 'optimize) '|optSEQ|))

```

defun optSEQ

— defun optSEQ —

```

(defun |optSEQ| (arg)
  (labels (
    (tryToRemoveSEQ (z)
      (if (and (consp z) (eq (qfirst z) 'seq) (consp (qrest z))
        (eq (qcddr z) nil) (consp (qsecond z))
        (consp (qcadadr z))
        (eq (qcddadr z) nil)
        (member (qcaadr z) '(exit return throw)))
      (qcadadr z)
      z))
    (SEQToCOND (z)
      (let (transform before aft)
        (setq transform
          (loop for x in z
            while

```

```

      (and (consp x) (eq (qfirst x) 'cond) (consp (qrest x))
           (eq (qcddr x) nil) (consp (qsecond x))
           (consp (qcaddr x))
           (eq (qcddadr x) nil)
           (consp (qcadadr x))
           (eq (qfirst (qcadadr x)) 'exit)
           (consp (qrest (qcadadr x)))
           (eq (qcddr (qcadadr x)) nil))
    collect
      (list (qcaadr x)
            (qsecond (qcadadr x))))
  (setq before (take (|#| transform) z))
  (setq aft (|after| z before))
  (cond
   ((null before) (cons 'seq aft))
   ((null aft)
    (cons 'cond (append transform (list '(t (|conderr|))))))
   (t
    (cons 'cond (append transform
                        (list (list 't (|optSEQ| (cons 'seq aft))))))))
  (getRidOfTemps (z))
  (let (g x r)
    (cond
     ((null z) nil)
     ((and (consp z) (consp (qfirst z)) (eq (qcaar z) 'let)
           (consp (qcdar z)) (consp (qcddar z))
           (gensymp (qcadar z))
           (> 2 (|numOfOccurencesOf| (qcadar z) (qrest z))))
      (setq g (qcadar z))
      (setq x (qcaddar z))
      (setq r (qrest z))
      (getRidOfTemps (subst x g r :test #'equal)))
     ((eq (car z) '||/throwAway|)
      (getRidOfTemps (cdr z)))
     (t
      (cons (car z) (getRidOfTemps (cdr z))))))
  (tryToRemoveSEQ (SEQToCOND (getRidOfTemps (cdr arg)))))

```

defplist optEQ

— postvars —

```

(eval-when (eval load)
  (setf (get 'eq 'optimize) '|optEQ|))

```

defun optEQ

— defun optEQ —

```
(defun |optEQ| (u)
  (let (z r)
    (cond
      ((and (consp u) (eq (qfirst u) 'eq) (consp (qrest u))
        (consp (qcddr u)) (eq (qcdddr u) nil))
       (setq z (qsecond u))
       (setq r (qthird u)))
      ; That undoes some weird work in Boolean to do with the definition of true
      (if (and (numberp z) (numberp r))
          (list 'quote (eq z r))
          u)))
  (t u))))
```

defplist optMINUS

— postvars —

```
(eval-when (eval load)
  (setf (get 'minus 'optimize) '|optMINUS|))
```

defun optMINUS

— defun optMINUS —

```
(defun |optMINUS| (u)
  (let (v)
    (cond
      ((and (consp u) (eq (qfirst u) 'minus) (consp (qrest u))
        (eq (qcddr u) nil))
       (setq v (qsecond u))
       (cond ((numberp v) (- v)) (t u)))
```



```
(t u))))
```

defplist optQSMINUS

— postvars —

```
(eval-when (eval load)
  (setf (get 'qsminus 'optimize) '|optQSMINUS|))
```

defun optQSMINUS

— defun optQSMINUS —

```
(defun |optQSMINUS| (u)
  (let (v)
    (cond
      ((and (consp u) (eq (qfirst u) 'qsminus) (consp (qrest u))
            (eq (qcddr u) nil))
       (setq v (qsecond u))
       (cond ((numberp v) (- v)) (t u)))
      (t u))))
```

defplist opt-

— postvars —

```
(eval-when (eval load)
  (setf (get '- 'optimize) '|opt-|))
```

defun opt-

— defun opt- —

```
(defun |opt-| (u)
  (let (v)
    (cond
      ((and (consp u) (eq (qfirst u) '-') (consp (qrest u))
            (eq (qcddr u) NIL))
       (setq v (qsecond u))
       (cond ((numberp v) (- v)) (t u)))
      (t u))))
```

—————

defplist optLESSP

— postvars —

```
(eval-when (eval load)
  (setf (get 'lessp 'optimize) '|optLESSP|))
```

—————

defun optLESSP

— defun optLESSP —

```
(defun |optLESSP| (u)
  (let (a b)
    (cond
      ((and (consp u) (eq (qfirst u) 'lessp) (consp (qrest u))
            (consp (qcddr u))
            (eq (qcdddr u) nil))
       (setq a (qsecond u))
       (setq b (qthird u))
       (if (eql b 0)
           (list 'minusp a)
           (list '> b a)))
      (t u))))
```

—————

defplist optSPADCALL

— postvars —

```
(eval-when (eval load)
  (setf (get 'spadcall 'optimize) '|optSPADCALL|))
```

—————

defun optSPADCALL

[optCall p229]
 [\$InteractiveMode p??]

— defun optSPADCALL —

```
(defun |optSPADCALL| (form)
  (let (fun arg1 tmp1 dom slot)
    (declare (special |$InteractiveMode|))
    (setq arg1 (cdr form))
    (cond
      ; last arg is function/env, but may be a form
      ((null |$InteractiveMode|) form)
      ((and (consp arg1)
            (progn (setq tmp1 (reverse arg1)) t)
            (consp tmp1))
       (setq fun (qfirst tmp1))
       (setq arg1 (qrest tmp1))
       (setq arg1 (nreverse arg1))
       (cond
         ((and (consp fun)
               (or (eq (qfirst fun) 'elt) (eq (qfirst fun) 'lispelt)))
          (progn
            (and (consp (qrest fun))
                 (progn
                  (setq dom (qsecond fun))
                  (and (consp (qcddr fun))
                       (eq (qcdddr fun) nil)
                       (progn
                        (setq slot (qthird fun))
                        t))))))
          (|optCall| (cons '|call| (cons (list 'elt dom slot) arg1))))
         (t form))))
      (t form))))
```

—————

defplist optSuchthat

— postvars —

```
(eval-when (eval load)
  (setf (get '|\\|' 'optimize) '|optSuchthat|))
```

—————

defun optSuchthat

— defun optSuchthat —

```
(defun |optSuchthat| (arg)
  (cons 'suchthat (cdr arg)))
```

—————

defplist optCatch

— postvars —

```
(eval-when (eval load)
  (setf (get 'catch 'optimize) '|optCatch|))
```

—————

defun optCatch

```
[rplac p??]
[optimize p225]
[$InteractiveMode p??]
```

— defun optCatch —

```
(defun |optCatch| (x)
  (labels (
    (changeThrowToExit (s g)
      (cond
```

```

((or (atom s) (member (car s) '(quote seq repeat collect))) nil)
((and (consp s) (eq (qfirst s) 'throw) (consp (qrest s))
      (equal (qsecond s) g))
  (|rplac| (car s) 'exit)
  (|rplac| (cdr s) (qcddr s)))
(t
  (changeThrowToExit (car s) g)
  (changeThrowToExit (cdr s) g)))
(hasNoThrows (a g)
  (cond
    ((and (consp a) (eq (qfirst a) 'throw) (consp (qrest a))
          (equal (qsecond a) g))
      nil)
    ((atom a) t)
    (t
      (and (hasNoThrows (car a) g)
            (hasNoThrows (cdr a) g))))))
(changeThrowToGo (s g)
  (let (u)
    (cond
      ((or (atom s) (eq (car s) 'quote)) nil)
      ((and (consp s) (eq (qfirst s) 'throw) (consp (qrest s))
            (equal (qsecond s) g) (consp (qcddr s))
            (eq (qcdddr s) nil))
        (setq u (qthird s))
        (changeThrowToGo u g)
        (|rplac| (car s) 'progn)
        (|rplac| (cdr s) (list (list 'let (cadr g) u) (list 'go (cadr g)))))
      (t
        (changeThrowToGo (car s) g)
        (changeThrowToGo (cdr s) g))))))
(let (g tmp2 u s tmp6 a)
  (declare (special |$InteractiveMode|))
  (setq g (cadr x))
  (setq a (caddr x))
  (cond
    (|$InteractiveMode| x)
    ((atom a) a)
    (t
      (cond
        ((and (consp a) (eq (qfirst a) 'seq) (consp (qrest a))
              (progn (setq tmp2 (reverse (qrest a))) t)
              (consp tmp2) (consp (qfirst tmp2)) (eq (qcaar tmp2) 'throw)
              (consp (qcddar tmp2))
              (equal (qcadar tmp2) g)
              (consp (qcddar tmp2))
              (eq (qcdddr tmp2) nil))
          (setq u (qcaddar tmp2))
          (setq s (qrest tmp2))
          (setq s (nreverse s))

```

```

(changeThrowToExit s g)
(|rplac| (cdr a) (append s (list (list 'exit u))))
(setq tmp6 (|optimize| x))
(setq a (caddr tmp6)))
(cond
  ((hasNoThrows a g)
   (|rplac| (car x) (car a))
   (|rplac| (cdr x) (cdr a)))
  (t
   (changeThrowToGo a g)
   (|rplac| (car x) 'seq)
   (|rplac| (cdr x)
    (list (list 'exit a) (cadr g) (list 'exit (cadr g))))))
x))))

```

defplist optCond

— postvars —

```

(eval-when (eval load)
  (setf (get 'cond 'optimize) '|optCond|))

```

defun optCond

```

[rplacd p??]
[TruthP p264]
[EqualBarGensym p244]
[rplac p??]

```

— defun optCond —

```

(defun |optCond| (x)
  (let (z p1 p2 c3 c1 c2 a result)
    (setq z (cdr x))
    (when
      (and (consp z) (consp (qrest z)) (eq (qcddr z) nil)
           (consp (qsecond z)) (consp (qcdadr z))
           (eq (qrest (qcdadr z)) nil)
           (|TruthP| (qcaadr z))
           (consp (qcadadr z)))

```

```

      (eq (qfirst (qcadr z)) 'cond))
    (rplacd (cdr x) (qrest (qcadr z))))
  (cond
    ((and (consp z) (consp (qfirst z)) (consp (qrest z)) (consp (qsecond z)))
      (setq p1 (qcaar z))
      (setq c1 (qcdar z))
      (setq p2 (qcaadr z))
      (setq c2 (qcdadr z))
      (when
        (or (and (consp p1) (eq (qfirst p1) 'null) (consp (qrest p1))
              (eq (qcddr p1) nil)
              (equal (qsecond p1) p2))
            (and (consp p2) (eq (qfirst p2) 'null) (consp (qrest p2))
              (eq (qcddr p2) nil)
              (equal (qsecond p2) p1))))
        (setq z (list (cons p1 c1) (cons 't c2)))
        (rplacd x z))
      (when
        (and (consp c1) (eq (qrest c1) nil) (equal (qfirst c1) 'nil)
              (equal p2 't) (equal (car c2) 't))
          (if (and (consp p1) (eq (qfirst p1) 'null) (consp (qrest p1))
                (eq (qcddr p1) nil))
              (setq result (qsecond p1))
              (setq result (list 'null p1))))))
    (if result
      result
      (cond
        ((and (consp z) (consp (qfirst z)) (consp (qrest z)) (consp (qsecond z))
              (consp (qcddr z)) (eq (qcddr z) nil)
              (consp (qthird z))
              (|TruthP| (qcaaddr z)))
          (setq p1 (qcaar z))
          (setq c1 (qcdar z))
          (setq p2 (qcaadr z))
          (setq c2 (qcdadr z))
          (setq c3 (qcdaddr z))
          (cond
            ((|EqualBarGensym| c1 c3)
              (list 'cond
                (cons (list 'or p1 (list 'null p2)) c1) (cons (list 'quote t) c2)))
            ((|EqualBarGensym| c1 c2)
              (list 'cond (cons (list 'or p1 p2) c1) (cons (list 'quote t) c3)))
            (t x)))
          (t
            (do ((y z (cdr y)))
              ((atom y) nil)
              (do ()
                ((null (and (consp y) (consp (qfirst y)) (consp (qcdar y))
                          (eq (qcddr y) nil) (consp (qrest y))
                          (consp (qsecond y)) (consp (qcdadr y))

```

```

      (eq (qcddadr y) nil)
      (|EqualBarGensym| (qcadar y)
                        (qcadadr y))))
    nil)
  (setq a (list 'or (qcaar y) (qcaadr y)))
  (rplac (car (car y)) a)
  (rplac (cdr y) (qcddr y)))
x)))))

```

defun EqualBarGensym

```

[gensymp p??]
[$GensymAssoc p??]
[$GensymAssoc p??]

```

— defun EqualBarGensym —

```

(defun |EqualBarGensym| (x y)
  (labels (
    (fn (x y)
      (let (z)
        (declare (special |$GensymAssoc|))
        (cond
          ((equal x y) t)
          ((and (gensymp x) (gensymp y))
           (if (setq z (|assoc| x |$GensymAssoc|))
               (if (equal y (cdr z)) t nil)
               (progn
                  (setq |$GensymAssoc| (cons (cons x y) |$GensymAssoc|))
                  t)))
          ((null x) (and (consp y) (eq (qrest y) nil) (gensymp (qfirst y))))
          ((null y) (and (consp x) (eq (qrest x) nil) (gensymp (qfirst x))))
          ((or (atom x) (atom y)) nil)
          (t
           (and (fn (car x) (car y))
                  (fn (cdr x) (cdr y))))))))
    (let (|$GensymAssoc|)
      (declare (special |$GensymAssoc|))
      (setq |$GensymAssoc| NIL)
      (fn x y)))

```

defplist optMkRecord

— postvars —

```
(eval-when (eval load)
  (setf (get '|mkRecord| 'optimize) '|optMkRecord|))
```

—————

defun optMkRecord

[length p??]

— defun optMkRecord —

```
(defun |optMkRecord| (arg)
  (let (u)
    (setq u (cdr arg))
    (cond
      ((and (consp u) (eq (qrest u) nil)) (list 'list (qfirst u)))
      ((eql (|#| u) 2) (cons 'cons u))
      (t (cons 'vector u)))))
```

—————

defplist optRECORDELT

— postvars —

```
(eval-when (eval load)
  (setf (get '|recordelt| 'optimize) '|optRECORDELT|))
```

—————

defun optRECORDELT

[keyedSystemError p??]

— defun optRECORDELT —

```
(defun |optRECORDELT| (arg)
  (let (name ind len)
    (setq name (cadr arg))
    (setq ind (caddr arg))
    (setq len (caddr arg))
    (cond
      ((eq len 1)
       (cond
         ((eq ind 0) (list 'qcar name))
         (t (|keyedSystemError| 'S2000002 (list ind)))))
      ((eq len 2)
       (cond
         ((eq ind 0) (list 'qcar name))
         ((eq ind 1) (list 'qcdr name))
         (t (|keyedSystemError| 'S2000002 (list ind)))))
      (t (list 'qvlt name ind)))))
```

defpllist optSETRECORDELT

— postvars —

```
(eval-when (eval load)
  (setf (get 'setrecordelt 'optimize) '|optSETRECORDELT|))
```

defun optSETRECORDELT

[keyedSystemError p??]

— defun optSETRECORDELT —

```
(defun |optSETRECORDELT| (arg)
  (let (name ind len expr)
    (setq name (cadr arg))
    (setq ind (caddr arg))
    (setq len (caddr arg))
    (setq expr (car (caddr arg)))
    (cond
      ((eq len 1)
       (if (eq ind 0)
           (list 'progn (list 'rplaca name expr) (list 'qcar name))
```

```

(|keyedSystemError| 'S2000002 (list ind))))
((eq1 len 2)
 (cond
  ((eq1 ind 0)
   (list 'progn (list 'rplaca name expr) (list 'qcar name)))
  ((eq1 ind 1)
   (list 'progn (list 'rplacd name expr) (list 'qcdr name)))
  (t (|keyedSystemError| 'S2000002 (list ind)))))
(t
 (list 'qsetvelt name ind expr))))))

```

deflist optRECORDCOPY

— postvars —

```

(eval-when (eval load)
 (setf (get 'recordcopy 'optimize) '|optRECORDCOPY|))

```

defun optRECORDCOPY

— defun optRECORDCOPY —

```

(defun |optRECORDCOPY| (arg)
 (let (name len)
  (setq name (cadr arg))
  (setq len (caddr arg))
  (cond
   ((eq1 len 1) (list 'list (list 'car name)))
   ((eq1 len 2) (list 'cons (list 'car name) (list 'cdr name)))
   (t
    (list 'replace (list 'make-array len) name))))))

```

6.3 Functions to manipulate modemaps

defun addDomain

```
[identp p??]
[qslessp p??]
[getDomainsInScope p250]
[domainMember p260]
[isLiteral p??]
[addNewDomain p251]
[getmode p??]
[isCategoryForm p??]
[isFunctor p249]
[constructor? p??]
[member p??]
[unknownTypeError p249]
```

— defun addDomain —

```
(defun |addDomain| (domain env)
  (let (s name tmp1)
    (cond
      ((atom domain)
        (cond
          ((eq domain '|$EmptyMode|) env)
          ((eq domain '|$NoValueMode|) env)
          ((or (null (identp domain))
              (and (qslessp 2 (|#| (setq s (princ-to-string domain))))
                  (eq (|char| '|#|) (elt s 0))
                  (eq (|char| '|#|) (elt s 1))))
            env)
          ((member domain (|getDomainsInScope| env)) env)
          ((|isLiteral| domain env) env)
          (t (|addNewDomain| domain env))))
      ((eq (setq name (car domain)) '|Category|) env)
      ((|domainMember| domain (|getDomainsInScope| env)) env)
      ((and (progn
              (setq tmp1 (|getmode| name env))
              (and (consp tmp1) (eq (qfirst tmp1) '|Mapping|)
                  (consp (qrest tmp1))))
            (|isCategoryForm| (second tmp1) env))
        (|addNewDomain| domain env))
      ((or (|isFunctor| name) (|constructor?| name))
        (|addNewDomain| domain env))
      (t
        (when (and (null (|isCategoryForm| domain env))
                    (null (|member| name '(|Mapping| category))))
          (|unknownTypeError| name))
```

```
env))))
```

defun unknownTypeError

```
[stackSemanticError p??]
```

— defun unknownTypeError —

```
(defun |unknownTypeError| (name)
  (let (op)
    (setq name
      (if (and (consp name) (setq op (qfirst name)))
          op
          name))
    (|stackSemanticError| (list '|%b| name '|%d| '|is not a known type|) nil)))
```

defun isFunctor

```
[opOf p??]
[identp p??]
[getdatabase p??]
[get p??]
[constructor? p??]
[updateCategoryFrameForCategory p117]
[updateCategoryFrameForConstructor p116]
[$CategoryFrame p??]
[$InteractiveMode p??]
```

— defun isFunctor —

```
(defun |isFunctor| (x)
  (let (op u prop)
    (declare (special |$CategoryFrame| |$InteractiveMode|))
    (setq op (|opOf| x))
    (cond
      ((null (identp op)) nil)
      (|$InteractiveMode|
       (if (member op '(|Union| |SubDomain| |Mapping| |Record|))
           t
           (member (getdatabase op 'constructorkind) '(|domain| |package|)))))
```

```

((setq u
  (or (|get| op '|isFunction| |$CategoryFrame|)
      (member op '(|SubDomain| |Union| |Record|))))
  u)
(|constructor?| op)
(cond
  ((setq prop (|get| op '|isFunction| |$CategoryFrame|)) prop)
  (t
   (if (eq (getdatabase op 'constructorkind) '|category|)
       (|updateCategoryFrameForCategory| op)
       (|updateCategoryFrameForConstructor| op))
   (|get| op '|isFunction| |$CategoryFrame|)))
  (t nil))))

```

defun getDomainsInScope

The way XLAMs work:

```
((XLAM ($1 $2 $3) (SETELT $1 0 $3)) X "c" V) ==> (SETELT X 0 V)
```

```

[get p??]
[$CapsuleDomainsInScope p??]
[$insideCapsuleFunctionIfTrue p??]

```

— defun getDomainsInScope —

```

(defun |getDomainsInScope| (env)
  (declare (special |$CapsuleDomainsInScope| |$insideCapsuleFunctionIfTrue|))
  (if |$insideCapsuleFunctionIfTrue|
      |$CapsuleDomainsInScope|
      (|get| '|$DomainsInScope| 'special env)))

```

defun putDomainsInScope

```

[getDomainsInScope p250]
[put p??]
[delete p??]
[say p??]
[member p??]
[$CapsuleDomainsInScope p??]

```

[*\$insideCapsuleFunctionIfTrue* p??]

— **defun putDomainsInScope** —

```
(defun |putDomainsInScope| (x env)
  (let (z newValue)
    (declare (special |$CapsuleDomainsInScope| |$insideCapsuleFunctionIfTrue|))
    (setq z (|getDomainsInScope| env))
    (when (|member| x z) (say "***** Domain: " x " already in scope"))
    (setq newValue (cons x (|delete| x z)))
    (if |$insideCapsuleFunctionIfTrue|
        (progn
          (setq |$CapsuleDomainsInScope| newValue)
          env)
        (|put| '|$DomainsInScope| 'special newValue env))))
```

—————

defun isSuperDomain

[*isSubset* p??]
 [*lassoc* p??]
 [*opOf* p??]
 [*get* p??]

— **defun isSuperDomain** —

```
(defun |isSuperDomain| (domainForm domainFormp env)
  (cond
    ((|isSubset| domainFormp domainForm env) t)
    ((and (eq domainForm '|Rep|) (eq domainFormp '$)) t)
    (t (lassoc (|opOf| domainFormp) (|get| domainForm '|SubDomain| env))))))
```

—————

defun addNewDomain

[*augModemapsFromDomain* p252]

— **defun addNewDomain** —

```
(defun |addNewDomain| (domain env)
  (|augModemapsFromDomain| domain domain env))
```

—————

defun augModemapsFromDomain

```

[member p??]
[kar p??]
[getDomainsInScope p250]
[getdatabase p??]
[opOf p??]
[addNewDomain p251]
[listOrVectorElementNode p??]
[stripUnionTags p??]
[augModemapsFromDomain1 p252]
[$Category p??]
[$DummyFunctorNames p??]

```

— **defun augModemapsFromDomain** —

```

(defun |augModemapsFromDomain| (name functorForm env)
  (let (curDomainsInScope u innerDom)
    (declare (special |$Category| |$DummyFunctorNames|))
    (cond
      ((|member| (or (kar name) name) |$DummyFunctorNames|)
       env)
      ((or (equal name |$Category|) (|isCategoryForm| name env))
       env)
      ((|member| name (setq curDomainsInScope (|getDomainsInScope| env)))
       env)
      (t
       (when (setq u (getdatabase (|opOf| functorForm) 'superdomain))
         (setq env (|addNewDomain| (car u) env)))
       (when (setq innerDom (|listOrVectorElementNode| name))
         (setq env (|addDomain| innerDom env)))
       (when (and (consp name) (eq (qfirst name) '|Union|))
         (dolist (d (|stripUnionTags| (qrest name)))
           (setq env (|addDomain| d env))))
       (|augModemapsFromDomain1| name functorForm env))))))

```

—————

defun augModemapsFromDomain1

```

[getl p??]
[kar p??]
[addConstructorModemaps p254]
[getmode p??]
[augModemapsFromCategory p260]
[getmodeOrMapping p??]

```


[substituteCategoryArguments p253]
 [stackMessage p??]

— defun augModemapsFromDomain1 —

```
(defun |augModemapsFromDomain1| (name functorForm env)
  (let (mappingForm categoryForm functArgTypes catform)
    (cond
      ((get1 (kar functorForm) '|makeFunctionList|)
        (|addConstructorModemaps| name functorForm env))
      ((and (atom functorForm) (setq catform (|getmode| functorForm env)))
        (|augModemapsFromCategory| name functorForm catform env))
      ((setq mappingForm (|getmodeOrMapping| (kar functorForm) env))
        (when (eq (car mappingForm) '|Mapping|) (car mappingForm))
        (setq categoryForm (cadr mappingForm))
        (setq functArgTypes (cddr mappingForm))
        (setq catform
          (|substituteCategoryArguments| (cdr functorForm) categoryForm))
        (|augModemapsFromCategory| name functorForm catform env))
      (t
        (|stackMessage| (list functorForm '| is an unknown mode|))
        env))))
```

—

defun substituteCategoryArguments

[internal p??]
 [stringimage p??]
 [sublis p??]

— defun substituteCategoryArguments —

```
(defun |substituteCategoryArguments| (argl catform)
  (let (arglAssoc (i 0))
    (setq argl (subst '$$ '$ argl :test #'equal))
    (setq arglAssoc
      (loop for a in argl
        collect (cons (internal '|#| (stringimage (incf i))) a)))
    (sublis arglAssoc catform)))
```

—

defun addConstructorModemaps

```
[putDomainsInScope p250]
[getl p??]
[addModemap p269]
[$InteractiveMode p??]
```

— **defun addConstructorModemaps** —

```
(defun |addConstructorModemaps| (name form env)
  (let (|$InteractiveMode| functorName fn tmp1 funList op sig nsig opcode)
    (declare (special |$InteractiveMode|))
    (setq functorName (car form))
    (setq |$InteractiveMode| nil)
    (setq env (|putDomainsInScope| name env))
    (setq fn (getl functorName '|makeFunctionList|))
    (setq tmp1 (funcall fn name form env))
    (setq funList (car tmp1))
    (setq env (cadr tmp1))
    (dolist (item funList)
      (setq op (first item))
      (setq sig (second item))
      (setq opcode (third item))
      (when (and (consp opcode) (consp (qrest opcode))
                  (consp (qcddr opcode))
                  (eq (qcddr opcode) nil)
                  (eq (qfirst opcode) 'elt))
        (setq nsig (subst '$$$ name sig :test #'equal))
        (setq nsig
          (subst '$ '$$$ (subst '$$ '$ nsig :test #'equal) :test #'equal))
        (setq opcode (list (first opcode) (second opcode) nsig)))
      (setq env (|addModemap| op name sig t opcode env)))
    env))
```

—————

defun getModemap

```
[get p??]
[compApplyModemap p255]
[sublis p??]
```

— **defun getModemap** —

```
(defun |getModemap| (x env)
  (let (u)
    (dolist (modemap (|get| (first x) '|modemap| env))
```

```
(when (setq u (|compApplyModemap| x modemap env nil))
  (return (sublis (third u) modemap))))))
```

defun compApplyModemap

```
[length p??]
[pmatchWithSl p??]
[sublis p??]
[comp p590]
[coerce p351]
[compMapCond p256]
[member p??]
[genDeltaEntry p??]
[$e p??]
[$bindings p??]
[$e p??]
[$bindings p??]
```

— defun compApplyModemap —

```
(defun |compApplyModemap| (form modemap |$e| sl)
  (declare (special |$e|))
  (let (op argl mc mr margl fnsel g mp lt ltp temp1 f)
    (declare (special |$bindings| |$e|))
    ; -- $e      is the current environment
    ; -- sl      substitution list, nil means bottom-up, otherwise top-down
    ; -- 0. fail immediately if #argl=#margl
    (setq op (car form))
    (setq argl (cdr form))
    (setq mc (caar modemap))
    (setq mr (cadar modemap))
    (setq margl (cddar modemap))
    (setq fnsel (cdr modemap))
    (when (= (|#| argl) (|#| margl))
      ; 1. use modemap to evaluate arguments, returning failed if not possible
      (setq lt
        (prog (t0)
          (return
            (do ((t1 argl (cdr t1)) (y NIL) (t2 margl (cdr t2)) (m nil))
              ((or (atom t1) (atom t2)) (nreverse0 t0))
              (setq y (car t1))
              (setq m (car t2))
              (setq t0
                (cons
                  (progn
```

```

      (setq sl (|pmatchWithSl| mp m sl))
      (setq g (sublis sl m))
      (setq temp1 (or (|comp| y g |$e|) (return '|failed|)))
      (setq mp (cadr temp1))
      (setq |$e| (caddr temp1))
      temp1)
      t0))))))
; 2. coerce each argument to final domain, returning failed
; if not possible
(unless (eq lt '|failed|)
  (setq ltp
    (loop for y in lt for d in (sublis sl margl)
      collect (or (|coerce| y d) (return '|failed|))))
  (unless (eq ltp '|failed|)
    ; 3. obtain domain-specific function, if possible, and return
    ; $bindings is bound by compMapCond
    (setq temp1 (|compMapCond| op mc sl fnsl))
    (when temp1
      ; can no longer trust what the modemap says for a reference into
      ; an exterior domain (it is calculating the displacement based on view
      ; information which is no longer valid; thus ignore this index and
      ; store the signature instead.)
      (setq f (car temp1))
      (setq |$bindings| (cadr temp1))
      (if (and (consp f) (consp (qcdr f)) (consp (qcddr f)) ; f is [op1,.]
        (eq (qcdddr f) nil)
        (|member| (qcar f) '(elt const |Subsumed|)))
        (list (|genDeltaEntry| (cons op modemap)) ltp |$bindings|)
        (list f ltp |$bindings|))))))

```

defun compMapCond

[compMapCond' p²⁵⁷]
 [\$bindings p??]

— defun compMapCond —

```

(defun |compMapCond| (op mc |$bindings| fnsl)
  (declare (special |$bindings|))
  (let (t0)
    (do ((t1 nil t0) (t2 fnsl (cdr t2)) (u nil))
      ((or t1 (atom t2) (progn (setq u (car t2)) nil)) t0)
      (setq t0 (or t0 (|compMapCond'| u op mc |$bindings|)))))

```

defun compMapCond'

```
[compMapCond" p257]
[compMapConfFun p??]
[stackMessage p??]
```

— defun compMapCond' —

```
(defun |compMapCond'| (t0 op dc bindings)
  (let ((cexpr (car t0)) (fexpr (cadr t0)))
    (if (|compMapCond''| cexpr dc)
        (|compMapCondFun| fexpr op dc bindings)
        (|stackMessage| '("not known that" %b ,dc %d "has" %b ,cexpr %d))))))
```

—————

defun compMapCond"

```
[compMapCond" p257]
[knownInfo p??]
[get p??]
[stackMessage p??]
[$Information p??]
[$e p??]
```

— defun compMapCond" —

```
(defun |compMapCond''| (cexpr dc)
  (let (l u tmp1 tmp2)
    (declare (special |$Information| |$e|))
    (cond
      ((eq cexpr t) t)
      ((and (consp cexpr)
            (eq (qcar cexpr) 'and)
            (progn (setq l (qcdr cexpr)) t))
        (prog (t0)
          (setq t0 t)
          (return
            (do ((t1 nil (null t0)) (t2 l (cdr t2)) (u nil))
              ((or t1 (atom t2) (progn (setq u (car t2)) nil)) t0)
              (setq t0 (and t0 (|compMapCond''| u dc)))))))
      ((and (consp cexpr)
            (eq (qcar cexpr) 'or)
            (progn (setq l (qcdr cexpr)) t))
        (prog (t3)
          (setq t3 nil)
          (return
```

```

      (do ((t4 nil t3) (t5 1 (cdr t5)) (u nil))
          ((or t4 (atom t5) (progn (setq u (car t5)) nil)) t3)
          (setq t3 (or t3 (|compMapCond''| u dc))))))
  ((and (consp cexpr)
        (eq (qcar cexpr) '|not|)
        (progn
          (setq tmp1 (qcdr cexpr))
          (and (consp tmp1)
                (eq (qcdr tmp1) nil)
                (progn (setq u (qcar tmp1)) t))))
    (null (|compMapCond''| u dc)))
  ((and (consp cexpr)
        (eq (qcar cexpr) '|has|)
        (progn
          (setq tmp1 (qcdr cexpr))
          (and (consp tmp1)
                (progn
                  (setq tmp2 (qcdr tmp1))
                  (and (consp tmp2)
                        (eq (qcdr tmp2) nil))))))
    (cond
      ((|knownInfo| cexpr) t)
      (t nil)))
  ((|member|
    (cons 'attribute (cons dc (cons cexpr nil)))
    (|get| '|$Information| 'special |$e|))
   t)
  (t
   (|stackMessage| '("not known that" %b ,dc %d "has" %b ,cexpr %d)
   nil)))

```

defun compMapCondFun

— defun compMapCondFun —

```

(defun |compMapCondFun| (fnexpr op dc bindings)
  (declare (ignore op) (ignore dc))
  (cons fnexpr (cons bindings nil)))

```

defun getUniqueSignature

[getUniqueModemap p259]

— defun getUniqueSignature —

```
(defun |getUniqueSignature| (form env)
  (cdar (|getUniqueModemap| (first form) (|#| (rest form)) env)))
```

—————

defun getUniqueModemap

[getModemapList p259]

[qslessp p??]

[stackWarning p??]

— defun getUniqueModemap —

```
(defun |getUniqueModemap| (op numOfArgs env)
  (let (mml)
    (cond
      ((eq 1 (|#| (setq mml (|getModemapList| op numOfArgs env))))
       (car mml))
      ((qslessp 1 (|#| mml))
       (|stackWarning|
        (list numOfArgs " argument form of: " op " has more than one modemap"))
       (car mml))
      (t nil))))
```

—————

defun getModemapList

[getModemapListFromDomain p260]

[nreverse0 p??]

[get p??]

— defun getModemapList —

```
(defun |getModemapList| (op numOfArgs env)
  (let (result)
    (cond
      ((and (consp op) (eq (qfirst op) '|elt|) (consp (qrest op))
```

```

      (consp (qcddr op)) (eq (qcddr op) nil))
    (|getModemapListFromDomain| (third op) numOfArgs (second op) env))
  (t
   (dolist (term (|get| op '|modemap| env) (nreverse0 result))
    (when (eql numOfArgs (|#| (cddar term))) (push term result))))))

```

defun getModemapListFromDomain

[get p??]

— defun getModemapListFromDomain —

```

(defun |getModemapListFromDomain| (op numOfArgs d env)
  (loop for term in (|get| op '|modemap| env)
    when (and (equal (caar term) d) (eql (|#| (cddar term)) numOfArgs))
    collect term))

```

defun domainMember

[modeEqual p362]

— defun domainMember —

```

(defun |domainMember| (dom domList)
  (let (result)
    (dolist (d domList result)
      (setq result (or result (|modeEqual| dom d)))))

```

defun augModemapsFromCategory

[evalAndSub p265]
 [compilerMessage p??]
 [putDomainsInScope p250]
 [addModemapKnown p268]
 [\$base p??]

— defun augModemapsFromCategory —


```
(defun |augModemapsFromCategory| (domainName functorform categoryForm env)
  (let (tmp1 op sig cond fnsel)
    (declare (special |$base|))
    (setq tmp1 (|evalAndSub| domainName domainName functorform categoryForm env))
    (|compilerMessage| (list '|Adding | domainName '| modemaps|))
    (setq env (|putDomainsInScope| domainName (second tmp1)))
    (setq |$base| 4)
    (dolist (u (first tmp1))
      (setq op (caar u))
      (setq sig (cadar u))
      (setq cond (cadr u))
      (setq fnsel (caddr u))
      (setq env (|addModemapKnown| op domainName sig cond fnsel env)))
    env))
```

defun addEltModemap

This is a hack to change selectors from strings to identifiers; and to add flag identifiers as literals in the environment [makeLiteral p??]

```
[addModemap1 p270]
[systemErrorHere p??]
[$insideCapsuleFunctionIfTrue p??]
[$e p??]
```

— defun addEltModemap —

```
(defun |addEltModemap| (op mc sig pred fn env)
  (let (tmp1 v sel lt id)
    (declare (special |$e| |$insideCapsuleFunctionIfTrue|))
    (cond
      ((and (eq op '|elt|) (consp sig))
        (setq tmp1 (reverse sig))
        (setq sel (qfirst tmp1))
        (setq lt (nreverse (qrest tmp1)))
        (cond
          ((stringp sel)
            (setq id (intern sel))
            (if |$insideCapsuleFunctionIfTrue|
              (setq |$e| (|makeLiteral| id |$e|))
              (setq env (|makeLiteral| id env)))
            (|addModemap1| op mc (append lt (list id)) pred fn env))
          (t (|addModemap1| op mc sig pred fn env))))
      ((and (eq op '|setelt|) (consp sig))
        (setq tmp1 (reverse sig))
        (setq v (qfirst tmp1))
```

```

(setq sel (qsecond tmp1))
(setq lt (nreverse (qcddr tmp1)))
(cond
  ((stringp sel) (setq id (intern sel))
    (if |$insideCapsuleFunctionIfTrue|
      (setq |$e| (|makeLiteral| id |$e|))
      (setq env (|makeLiteral| id env)))
    (|addModemap1| op mc (append lt (list id v)) pred fn env))
  (t (|addModemap1| op mc sig pred fn env))))
(t (|systemErrorHere| "addEltModemap"))))

```

defun mkNewModemapList

```

[member p??]
[assoc p??]
[mergeModemap p263]
[nreverse0 p??]
[insertModemap p263]
[$InteractiveMode p??]
[$forceAdd p??]

```

— defun mkNewModemapList —

```

(defun |mkNewModemapList| (mc sig pred fn curModemapList env filenameOrNil)
  (let (map entry oldMap opred result)
    (declare (special |$InteractiveMode| |$forceAdd|))
    (setq entry
      (cons (setq map (cons mc sig)) (cons (list pred fn) filenameOrNil)))
    (cond
      ((|member| entry curModemapList) curModemapList)
      ((and (setq oldMap (|assoc| map curModemapList))
        (consp oldMap) (consp (qrest oldMap))
        (consp (qsecond oldMap))
        (consp (qcddr oldMap))
        (eq (qcddadr oldMap) nil)
        (equal (qcadadr oldMap) fn))
        (setq opred (qcaadr oldMap))
        (cond
          (|$forceAdd| (|mergeModemap| entry curModemapList env))
          ((eq opred t) curModemapList)
          (t
            (when (and (not (eq pred t)) (not (equal pred opred)))
              (setq pred (list 'or pred opred)))
            (dolist (x curModemapList (nreverse0 result))
              (push

```

```

      (if (equal x oldMap)
          (cons map (cons (list pred fn) filenameOrNil))
          x)
      result))))))
(|$InteractiveMode|
 (|insertModemap| entry curModemapList))
(t
 (|mergeModemap| entry curModemapList env))))

```

defun insertModemap

— defun insertModemap —

```

(defun |insertModemap| (new mmList)
  (if (null mmList) (list new) (cons new mmList)))

```

defun mergeModemap

[isSuperDomain p251]
 [TruthP p264]
 [\$forceAdd p??]

— defun mergeModemap —

```

(defun |mergeModemap| (entry modemapList env)
  (let (mc sig pred mcp sigp predp newmm mm)
    (declare (special |$forceAdd|))
    ; break out the condition, signature, and predicate fields of the new entry
    (setq mc (caar entry))
    (setq sig (cdar entry))
    (setq pred (caadr entry))
    (seq
     ; walk across the successive tails of the modemap list
     (do ((mmtail modemapList (cdr mmtail))
          ((atom mmtail) nil)
          (setq mcp (caaar mmtail))
          (setq sigp (cdaar mmtail))
          (setq predp (caadar mmtail))
          (cond
           ((or (equal mc mcp) (|isSuperDomain| mcp mc env))

```

```

; if this is a duplicate condition
(exit
  (progn
    (setq newmm nil)
    (setq mm modemapList)
    ; copy the unique modemap terms
    (loop while (not (eq mm mmtail)) do
      (setq newmm (cons (car mm) newmm))
      (setq mm (cdr mm)))
    ; if the conditions and signatures are equal
    (when (and (equal mc mcp) (equal sig sigp))
      ; we only need one of these unless the conditions are hairy
      (cond
        ((and (null |$forceAdd|) (|TruthP| predp))
          ; the new predicate buys us nothing
          (setq entry nil)
          (return modemapList))
        ((|TruthP| pred)
          ; the thing we matched against is useless, by comparison
          (setq mmtail (cdr mmtail))))))
    (setq modemapList (nconc (nreverse newmm) (cons entry mmtail)))
    (setq entry nil)
    (return modemapList))))))
; if the entry is still defined, add it to the modemap
(if entry
  (append modemapList (list entry))
  modemapList)))

```

defun TruthP

— defun TruthP —

```

(defun |TruthP| (x)
  (cond
    ((null x) nil)
    ((eq x t) t)
    ((and (consp x) (eq (qfirst x) 'quote)) t)
    (t nil)))

```

defun evalAndSub

[isCategory p??]
 [substNames p266]
 [contained p??]
 [put p??]
 [get p??]
 [getOperationAlist p265]
 [\$lhsOfColon p??]

— defun evalAndSub —

```
(defun |evalAndSub| (domainName viewName functorForm form |$e|)
  (declare (special |$e|))
  (let (|$lhsOfColon| opAlist substAlist)
    (declare (special |$lhsOfColon|))
    (setq |$lhsOfColon| domainName)
    (cond
      ((|isCategory| form)
       (list (|substNames| domainName viewName functorForm (elt form 1)) |$e|))
      (t
       (when (contained '$$ form)
         (setq |$e| (|put| '$$ '|model| (|get| '$ '|model| |$e|) |$e|)))
       (setq opAlist (|getOperationAlist| domainName functorForm form))
       (setq substAlist (|substNames| domainName viewName functorForm opAlist))
       (list substAlist |$e|))))))
```

—————

defun getOperationAlist

[getdatabase p??]
 [isFunctor p249]
 [systemError p??]
 [compMakeCategoryObject p208]
 [stackMessage p??]
 [\$e p??]
 [\$domainShell p??]
 [\$insideFunctorIfTrue p??]
 [\$functorForm p??]

— defun getOperationAlist —

```
(defun |getOperationAlist| (name functorForm form)
  (let (u tt)
    (declare (special |$e| |$domainShell| |$insideFunctorIfTrue| |$functorForm|))
```

```

(when (and (atom name) (getdatabase name 'niladic))
  (setq functorform (list functorForm)))
(cond
  ((and (setq u (|isFunction| functorForm))
        (null (and (|$insideFunctorIfTrue|
                    (equal (first functorForm) (first |$functorForm|))))))
   u)
  ((and (|$insideFunctorIfTrue| (eq name '$))
        (if (|$domainShell|
              (elt (|$domainShell| 1)
                    (|systemError| "$ has no shell now"))))
        (|compMakeCategoryObject| form |$e|))
   (setq tt (|compMakeCategoryObject| form |$e|))
   (setq |$e| (third tt))
   (elt (first tt) 1))
  (t
   (|stackMessage| (list 'not a category form: | form|))))))

```

defvar \$FormalMapVariableList

— initvars —

```

(defvar |$FormalMapVariableList|
  '(\#1 \#2 \#3 \#4 \#5 \#6 \#7 \#8 \#9 \#10 \#11 \#12 \#13 \#14 \#15))

```

defun substNames

```

[isCategoryPackageName p210]
[eqsubstlist p??]
[nreverse0 p??]
[$FormalMapVariableList p266]

```

— defun substNames —

```

(defun |substNames| (domainName viewName functorForm opalist)
  (let (nameForDollar sel pos modemapform tmp0 tmp1)
    (declare (special |$FormalMapVariableList|))
    (setq functorForm (subst '$$ '$ functorForm))
    (setq nameForDollar
      (if (|isCategoryPackageName| functorForm)
          (second functorForm)

```

```

    domainName))
; following calls to SUBSTQ must copy to save RPLAC's in
; putInLocalDomainReferences
(dolist (term
        (eqsubstlist (kdr functorForm) |$FormalMapVariableList| opalist)
        (nreverse0 tmp0)))
(setq tmp1 (reverse term))
(setq sel (caar tmp1))
(setq pos (caddar tmp1))
(setq modemapform (nreverse (cdr tmp1)))
(push
 (append
  (subst '$ '$$ (subst nameForDollar '$ modemapform))
  (list
   (list sel viewName (if (eq domainName '$) pos (cadar modemapform))))
 tmp0))))

```

defun augModemapsFromCategoryRep

```

[evalAndSub p265]
[isCategory p??]
[compilerMessage p??]
[putDomainsInScope p250]
[assoc p??]
[addModemap p269]
[$base p??]

```

— defun augModemapsFromCategoryRep —

```

(defun |augModemapsFromCategoryRep|
  (domainName repDefn functorBody categoryForm env)
  (labels (
    (redefinedList (op z)
      (let (result)
        (dolist (u z result)
          (setq result (or result (redefined op u))))))
    (redefined (opname u)
      (let (op z result)
        (when (consp u)
          (setq op (qfirst u))
          (setq z (qrest u))
          (cond
            ((eq op 'def) (equal opname (caar z)))
            ((member op '(progn seq)) (redefinedList opname z))
            ((eq op 'cond)

```

```

      (dolist (v z result)
        (setq result (or result (redefinedList opname (cdr v)))))))))
(let (fnAlist tmp1 repFnAlist catform lhs op sig cond fnset u)
(declare (special |$base|))
(setq tmp1 (|evalAndSub| domainName domainName domainName categoryForm env))
(setq fnAlist (car tmp1))
(setq env (cadr tmp1))
(setq tmp1 (|evalAndSub| '|Rep| '|Rep| repDefn (|getmode| repDefn env) env))
(setq repFnAlist (car tmp1))
(setq env (cadr tmp1))
(setq catform
  (if (|isCategory| categoryForm) (elt categoryForm 0) categoryForm))
(|compilerMessage| (list '|Adding| domainName '| modemap|))
(setq env (|putDomainsInScope| domainName env))
(setq |$base| 4)
(dolist (term fnAlist)
  (setq lhs (car term))
  (setq op (caar term))
  (setq sig (cadar term))
  (setq cond (cadr term))
  (setq fnset (caddr term))
  (setq u (|assoc| (subst '|Rep| domainName lhs :test #'equal) repFnAlist))
  (if (and u (null (redefinedList op functorBody)))
    (setq env (|addModemap| op domainName sig cond (caddr u) env))
    (setq env (|addModemap| op domainName sig cond fnset env))))
env)))

```

6.4 Maintaining Modemaps

defun addModemapKnown

```

[addModemap0 p269]
[$e p??]
[$insideCapsuleFunctionIfTrue p??]
[$CapsuleModemapFrame p??]

```

— defun addModemapKnown —

```

(defun |addModemapKnown| (op mc sig pred fn |$e|)
(declare (special |$e| |$CapsuleModemapFrame| |$insideCapsuleFunctionIfTrue|))
(if (eq |$insideCapsuleFunctionIfTrue| t)
  (progn
    (setq |$CapsuleModemapFrame|
      (|addModemap0| op mc sig pred fn |$CapsuleModemapFrame|))
  )
)

```



```

    |$e|)
  (|addModemap0| op mc sig pred fn |$e|)))

```

defun addModemap

```

[addModemap0 p269]
[knownInfo p??]
[$e p??]
[$InteractiveMode p??]
[$insideCapsuleFunctionIfTrue p??]
[$CapsuleModemapFrame p??]
[$CapsuleModemapFrame p??]

```

— defun addModemap —

```

(defun |addModemap| (op mc sig pred fn |$e|)
  (declare (special |$e| |$CapsuleModemapFrame| |$InteractiveMode|
                  |$insideCapsuleFunctionIfTrue|))
  (cond
    (|$InteractiveMode| |$e|)
    (t
     (when (|knownInfo| pred) (setq pred t))
     (cond
       ((eq |$insideCapsuleFunctionIfTrue| t)
        (setq |$CapsuleModemapFrame|
              (|addModemap0| op mc sig pred fn |$CapsuleModemapFrame|))
        |$e|)
       (t
        (|addModemap0| op mc sig pred fn |$e|)))))))

```

defun addModemap0

```

[addEltModemap p261]
[addModemap1 p270]
[$functorForm p??]

```

— defun addModemap0 —

```

(defun |addModemap0| (op mc sig pred fn env)
  (declare (special |$functorForm|))

```

```
(cond
  ((and (consp |$functorForm|)
        (eq (qfirst |$functorForm|) '|CategoryDefaults|)
        (eq mc '$))
    env)
  ((or (eq op '|elt|) (eq op '|setelt|))
    (|addEltModemap| op mc sig pred fn env))
  (t (|addModemap1| op mc sig pred fn env))))
```

defun addModemap1

```
[getProplist p??]
[mkNewModemapList p262]
[lassoc p??]
[augProplist p??]
[unErrorRef p??]
[addBinding p??]
```

— defun addModemap1 —

```
(defun |addModemap1| (op mc sig pred fn env)
  (let (currentProplist newModemapList newProplist newProplistp)
    (when (eq mc '|Rep|) (setq sig (subst '$ '|Rep| sig :test #'equal)))
    (setq currentProplist (or (|getProplist| op env) nil))
    (setq newModemapList
      (|mkNewModemapList| mc sig pred fn
        (lassoc '|modemap| currentProplist) env nil))
    (setq newProplist (|augProplist| currentProplist '|modemap| newModemapList))
    (setq newProplistp (|augProplist| newProplist 'fluid t))
    (|unErrorRef| op)
    (|addBinding| op newProplistp env)))
```

6.5 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)))
```

defplist compAdd plist

We set up the `compAdd` function to handle the `add` keyword by setting the `special` keyword on the `add` symbol property list.

— **postvars** —

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

—————

defun compAdd

The `compAdd` function expects three arguments:

1. the **form** which is an `—add—` specifying the domain to extend and a set of functions to be added
2. the **mode** a `—Join—`, which is a set of categories and domains
3. the **env** which is a list of functions and their modemaps

The bulk of the work is performed by a call to `compOrCroak` which compiles the functions in the `add` form capsule.

The `compAdd` function returns a triple, the result of a call to `compCapsule`.

1. the **compiled capsule** which is a progn form which returns the domain
2. the **mode** from the input argument
3. the **env** prepended with the signatures of the functions in the body of the `add`.

```
[comp p590]
[compSubDomain1 p346]
[nreverse0 p??]
[NRTgetLocalIndex p211]
[compTuple2Record p273]
[compOrCroak p588]
[compCapsule p274]
[/editfile p??]
[$addForm p??]
[$addFormLhs p??]
[$EmptyMode p172]
[$NRTaddForm p??]
[$packagesUsed p??]
[$functorForm p??]
```

[\$bootStrapMode p??]

— defun compAdd —

```
(defun compAdd (form mode env)
  (let (|$addForm| |$addFormLhs| code domainForm predicate tmp3 tmp4)
    (declare (special |$addForm| |$addFormLhs| |$EmptyMode| |$NRTaddForm|
                      |$packagesUsed| |$functorForm| |$bootStrapMode| /editfile))
    (setq |$addForm| (second form))
    (cond
      ((eq |$bootStrapMode| t)
        (cond
          ((and (consp |$addForm|) (eq (qfirst |$addForm|) '|@Tuple|))
            (setq code nil))
          (t
            (setq tmp3 (|comp| |$addForm| mode env))
            (setq code (first tmp3))
            (setq mode (second tmp3))
            (setq env (third tmp3)) tmp3))
        (list
          (list 'cond
            (list '|$bootStrapMode| code)
            (list 't
              (list '|systemError|
                (list 'list '|%b| (mkq (car |$functorForm|)) '|%d| "from"
                  '|%b| (mkq (|namestring| /editfile)) '|%d|
                    "needs to be compiled"))))
              mode env))
          (t
            (setq |$addFormLhs| |$addForm|)
            (cond
              ((and (consp |$addForm|) (eq (qfirst |$addForm|) '|SubDomain|)
                (consp (qrest |$addForm|)) (consp (qcddr |$addForm|))
                (eq (qcddr |$addForm|) nil))
                (setq domainForm (second |$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 mode env))
                (setq |$addForm| (first tmp3))
                (setq env (third tmp3)) tmp3)
              (t
                (setq |$packagesUsed|
                  (if (and (consp |$addForm|) (eq (qfirst |$addForm|) '|@Tuple|))
                    (append (qrest |$addForm|) |$packagesUsed|)
                    (cons |$addForm| |$packagesUsed|)))
                  (setq |$NRTaddForm| |$addForm|))
```

```

(setq tmp3
  (cond
    ((and (consp |$addForm|) (eq (qfirst |$addForm|) '|@Tuple|))
      (setq |$NRTaddForm|
        (cons '|@Tuple|
          (dolist (x (cdr |$addForm|) (nreverse0 tmp4))
            (push (|NRTgetLocalIndex| x) tmp4))))
      (|compOrCroak| (|compTuple2Record| |$addForm|) |$EmptyMode| env))
    (t
      (|compOrCroak| |$addForm| |$EmptyMode| env))))
(setq |$addForm| (first tmp3))
(setq env (third tmp3))
tmp3))
(|compCapsule| (third form) mode env))))

```

defun compTuple2Record

— defun compTuple2Record —

```

(defun |compTuple2Record| (u)
  (let ((i 0))
    (cons '|Record|
      (loop for x in (rest u)
        collect (list '|:| (incf i) x)))))

```

defplist compCapsule plist

We set up the `compCapsule` function to handle the `capsule` keyword by setting the `special` keyword on the `capsule` symbol property list.

— postvars —

```

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

```

defun compCapsule

```
[bootStrapError p215]
[compCapsuleInner p274]
[addDomain p248]
[editfile p??]
[$insideExpressionIfTrue p??]
[$functorForm p??]
[$bootStrapMode p??]
```

— defun compCapsule —

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

—————

defun compCapsuleInner

```
[addInformation p??]
[compCapsuleItems p275]
[processFunctor p275]
[mkpf p??]
[$getDomainCode p??]
[$signature p??]
[$form p??]
[$addForm p??]
[$insideCategoryPackageIfTrue p??]
[$insideCategoryIfTrue p??]
[$functorLocalParameters p??]
```

— defun compCapsuleInner —

```
(defun |compCapsuleInner| (form mode env)
  (let (localParList data code)
    (declare (special |$getDomainCode| |$signature| |$form| |$addForm|
                    |$insideCategoryPackageIfTrue| |$insideCategoryIfTrue|
```

```

(|$functorLocalParameters|))
(setq env (|addInformation| mode env))
(setq data (cons 'progn form))
(setq env (|compCapsuleItems| form nil env))
(setq localParList |$functorLocalParameters|)
(when |$addForm| (setq data (list '|add| |$addForm| data)))
(setq code
  (if (and |$insideCategoryIfTrue| (null |$insideCategoryPackageIfTrue|))
      data
      (|processFunctor| |$form| |$signature| data localParList env)))
(cons (mkpf (append |$getDomainCode| (list code)) 'progn) (list mode env)))

```

defun processFunctor

```

[error p??]
[buildFunctor p??]

```

— defun processFunctor —

```

(defun |processFunctor| (form signature data localParList e)
  (cond
    ((and (consp form) (eq (qrest form) nil)
          (eq (qfirst form) '|CategoryDefaults|))
     (|error| '|CategoryDefaults is a reserved name|))
    (t (|buildFunctor| form signature data localParList e))))

```

defun compCapsuleItems

The variable data appears to be unbound at runtime. Optimized code won't check for this but interpreted code fails. We should PROVE that data is unbound at runtime but have not done so yet. Rather than remove the code entirely (since there MIGHT be a path where it is used) we check for the runtime bound case and assign `$myFunctorBody` if data has a value.

The `compCapsuleInner` function in this file LOOKS like it sets data and expects code to manipulate the assigned data structure. Since we can't be sure we take the least disruptive course of action.

```

[compSingleCapsuleItem p276]
[$stop-level p??]
[$myFunctorBody p??]
[$signatureOfForm p??]

```

```
[$suffix p??]
[$e p??]
[$pred p??]
[$e p??]
```

— defun compCapsuleItems —

```
(defun |compCapsuleItems| (itemlist |$predl| |$e|)
  (declare (special |$predl| |$e|))
  (let ($top_level |$myFunctorBody| |$signatureOfForm| |$suffix|)
    (declare (special $top_level |$myFunctorBody| |$signatureOfForm| |$suffix|))
    (setq $top_level nil)
    (setq |$myFunctorBody| nil)
    (when (boundp '|data|) (setq |$myFunctorBody| |data|))
    (setq |$signatureOfForm| nil)
    (setq |$suffix| 0)
    (loop for item in itemlist do
      (setq |$e| (|compSingleCapsuleItem| item |$predl| |$e|)))
    |$e|))
```

defun compSingleCapsuleItem

```
[doit p??]
[$pred p??]
[$e p??]
[macroExpandInPlace p174]
```

— defun compSingleCapsuleItem —

```
(defun |compSingleCapsuleItem| (item |$predl| |$e|)
  (declare (special |$predl| |$e|))
  (|doIt| (|macroExpandInPlace| item |$e|) |$predl|)
  |$e|)
```

defun doIt

```
[lastnode p??]
[compSingleCapsuleItem p276]
[isDomainForm p344]
[stackWarning p??]
```



```

[doIt p276]
[compOrCroak p588]
[stackSemanticError p??]
[bright p??]
[member p??]
[kar p??]
[—isFunction p??]
[insert p??]
[opOf p??]
[get p??]
[NRTgetLocalIndex p211]
[sublis p??]
[compOrCroak p588]
[sayBrightly p??]
[formatUnabbreviated p??]
[doItIf p281]
[isMacro p282]
[put p??]
[cannotDo p??]
[$pred1 p??]
[$e p??]
[$EmptyMode p172]
[$NonMentionableDomainNames p??]
[$functorLocalParameters p??]
[$functorsUsed p??]
[$packagesUsed p??]
[$NRTopt p??]
[$Representation p??]
[$LocalDomainAlist p??]
[$QuickCode p??]
[$signatureOfForm p??]
[$genno p??]
[$e p??]
[$functorLocalParameters p??]
[$functorsUsed p??]
[$packagesUsed p??]
[$Representation p??]
[$LocalDomainAlist p??]

```

— defun doIt —

```

(defun |doIt| (item |$pred1|)
  (declare (special |$pred1|))
  (prog ($genno x rhs lhsp lhs rhsp rhsCode z tmp1 tmp2 tmp6 op body tt
        functionPart u code)
    (declare (special $genno |$e| |$EmptyMode| |$signatureOfForm|

```

```

|$QuickCode| |$LocalDomainAlist| |$Representation|
|$NRTopt| |$packagesUsed| |$functorsUsed|
|$functorLocalParameters| |$NonMentionableDomainNames|))

(setq $genno 0)
(cond
  ((and (consp item) (eq (qfirst item) 'seq) (consp (qrest item))
    (progn (setq tmp6 (reverse (qrest item))) t)
    (consp tmp6) (consp (qfirst tmp6))
    (eq (qcaar tmp6) 'exit)
    (consp (qcдар tmp6))
    (equal (qcadar tmp6) 1)
    (consp (qcddar tmp6))
    (eq (qcdddar tmp6) nil))
    (setq x (qcaddar tmp6))
    (setq z (qrest tmp6))
    (setq z (nreverse z))
    (rplaca item 'progn)
    (rplaca (lastnode item) x)
    (loop for it1 in (rest item)
      do (setq |$e| (|compSingleCapsuleItem| it1 |$predl| |$e|))))
  ((|isDomainForm| item |$e|)
    (setq u (list 'import (cons (car item) (cdr item))))
    (|stackWarning| (list 'Use: import (cons (car item) (cdr item))))
    (rplaca item (car u))
    (rplacd item (cdr u))
    (|doIt| item |$predl|))
  ((and (consp item) (eq (qfirst item) 'let) (consp (qrest item))
    (consp (qcddr item)))
    (setq lhs (qsecond item))
    (setq rhs (qthird item))
    (cond
      ((null (progn
        (setq tmp2 (|compOrCroak| item |$EmptyMode| |$e|))
        (and (consp tmp2)
          (progn
            (setq code (qfirst tmp2))
            (and (consp (qrest tmp2))
              (progn
                (and (consp (qcddr tmp2))
                  (eq (qcdddar tmp2) nil)
                  (PROGN
                    (setq |$e| (qthird tmp2))
                    t))))))))
        (|stackSemanticError|
          (cons 'cannot compile assigned value to (|bright| lhs))
          nil))
      (null (and (consp code) (eq (qfirst code) 'let)
        (progn
          (and (consp (qrest code))
            (progn

```

```

                (setq lhsp (qsecond code))
                (and (consp (qcddr code))))))
                (atom (qsecond code))))
(cond
  ((and (consp code) (eq (qfirst code) 'progn))
   (|stackSemanticError|
    (list '|multiple assignment | item '| not allowed|)
    nil))
  (t
   (rplaca item (car code))
   (rplacd item (cdr code)))))
(t
 (setq lhs lhsp)
 (cond
  ((and (null (|member| (kar rhs) |$NonMentionableDomainNames|))
        (null (member lhs |$functorLocalParameters|)))
   (setq |$functorLocalParameters|
         (append |$functorLocalParameters| (list lhs)))))
  (cond
   ((and (consp code) (eq (qfirst code) 'let))
    (progn
     (setq tmp2 (qrest code))
     (and (consp tmp2)
          (progn
           (setq tmp6 (qrest tmp2))
           (and (consp tmp6)
                (progn
                 (setq rhsp (qfirst tmp6))
                 t))))))
     (|isDomainForm| rhsp |$e|))
    (cond
     ((|isFunctor| rhsp)
      (setq |$functorsUsed| (|insert| (|opOf| rhsp) |$functorsUsed|))
      (setq |$packagesUsed| (|insert| (list (|opOf| rhsp))
                                       |$packagesUsed|))))
     (cond
      ((eq lhs '|Rep|)
       (setq |$Representation| (elt (|get| '|Rep| '|value| |$e|) 0))
       (cond
        ((eq |$NRTopt| t)
         (|NRTgetLocalIndex| |$Representation|)
         (t nil))))
      (setq |$LocalDomainAlist|
            (cons (cons lhs
                        (sublis |$LocalDomainAlist| (elt (|get| lhs '|value| |$e|) 0))
                        |$LocalDomainAlist|))))
      (cond
       ((and (consp code) (eq (qfirst code) 'let))
        (rplaca item (if |$QuickCode| 'qsetrefv 'setelt))
        (setq rhsCode rhsp)

```

```

      (rplacd item (list '$ (|NRTgetLocalIndex| lhs) rhsCode)))
    (t
      (rplaca item (car code))
      (rplacd item (cdr code))))))
  ((and (consp item) (eq (qfirst item) '|:|') (consp (qrest item))
    (consp (qcddr item)) (eq (qcdddr item) nil))
    (setq tmp1 (|compOrCroak| item |$EmptyMode| |$e|))
    (setq |$e| (caddr tmp1))
    tmp1)
  ((and (consp item) (eq (qfirst item) '|import|'))
    (loop for dom in (qrest item)
      do (|sayBrightly| (cons "    importing " (|formatUnabbreviated| dom))))
    (setq tmp1 (|compOrCroak| item |$EmptyMode| |$e|))
    (setq |$e| (caddr tmp1))
    (rplaca item 'progn)
    (rplacd item nil))
  ((and (consp item) (eq (qfirst item) '|if|'))
    (|doItIf| item |$pred1| |$e|))
  ((and (consp item) (eq (qfirst item) '|where|') (consp (qrest item)))
    (|compOrCroak| item |$EmptyMode| |$e|))
  ((and (consp item) (eq (qfirst item) '|mdef|'))
    (setq tmp1 (|compOrCroak| item |$EmptyMode| |$e|))
    (setq |$e| (caddr tmp1)) tmp1)
  ((and (consp item) (eq (qfirst item) '|def|') (consp (qrest item))
    (consp (qsecond item)))
    (setq op (qcaadr item))
    (cond
      ((setq body (|isMacro| item |$e|))
        (setq |$e| (|put| op '|macro| body |$e|)))
      (t
        (setq tt (|compOrCroak| item |$EmptyMode| |$e|))
        (setq |$e| (caddr tt))
        (rplaca item '|CodeDefine|)
        (rplacd (cadr item) (list |$signatureOfForm|))
        (setq functionPart (list '|dispatchFunction| (car tt)))
        (rplaca (cddr item) functionPart)
        (rplacd (cddr item) nil))))
  ((setq u (|compOrCroak| item |$EmptyMode| |$e|))
    (setq code (car u))
    (setq |$e| (caddr u))
    (rplaca item (car code))
    (rplacd item (cdr code)))
  (t (|cannotDo|))))

```

defun doItIf

```
[comp p590]
[userError p??]
[compSingleCapsuleItem p276]
[getSuccessEnvironment p314]
[localExtras p??]
[rplaca p??]
[rplacd p??]
[$e p??]
[$functorLocalParameters p??]
[$predl p??]
[$e p??]
[$functorLocalParameters p??]
[$getDomainCode p??]
[$Boolean p??]
```

— defun doItIf —

```
(defun |doItIf| (item |$predl| |$e|)
  (declare (special |$predl| |$e|))
  (labels (
    (localExtras (oldFLP)
      (let (oldFLPp flp1 gv ans nils n)
        (declare (special |$functorLocalParameters| |$getDomainCode|))
        (unless (eq oldFLP |$functorLocalParameters|)
          (setq flp1 |$functorLocalParameters|)
          (setq oldFLPp oldFLP)
          (setq n 0)
          (loop while oldFLPp
            do
              (setq oldFLPp (cdr oldFLPp))
              (setq n (1+ n)))
          (setq nils (setq ans nil))
          (loop for u in flp1
            do
              (if (or (atom u)
                (let (result)
                  (loop for v in |$getDomainCode|
                    do
                      (setq result (or result
                        (and (consp v) (consp (qrest v))
                          (equal (qsecond v) u))))
                  result)))
              ; Now we have to add code to compile all the elements of
              ; functorLocalParameters that were added during the conditional compilation
              (setq nils (cons u nils))
              (progn
```

```

      (setq gv (gensym))
      (setq ans (cons (list 'let gv u) ans))
      (setq nils (CONS gv nils)))
    (setq n (1+ n)))
  (setq |$functorLocalParameters| (append oldFLP (nreverse nils)))
  (nreverse ans))))
(let (p x y olde tmp1 pp xp oldFLP yp)
(declare (special |$functorLocalParameters| |$Boolean|))
  (setq p (second item))
  (setq x (third item))
  (setq y (fourth item))
  (setq olde |$e|)
  (setq tmp1
    (or (|comp| p |$Boolean| |$e|)
        (|userError| (list "not a Boolean:" p))))
  (setq pp (first tmp1))
  (setq |$e| (third tmp1))
  (setq oldFLP |$functorLocalParameters|)
  (unless (eq x '|noBranch|)
    (|compSingleCapsuleItem| x |$pred| (|getSuccessEnvironment| p |$e|))
    (setq xp (localExtras oldFLP)))
  (setq oldFLP |$functorLocalParameters|)
  (unless (eq y '|noBranch|)
    (|compSingleCapsuleItem| y |$pred| (|getInverseEnvironment| p olde))
    (setq yp (localExtras oldFLP)))
  (rplaca item 'cond)
  (rplacd item (list (cons pp (cons x xp)) (cons ''t (cons y yp))))))

```

defun isMacro

[get p??]

— defun isMacro —

```

(defun |isMacro| (x env)
  (let (op args signature body)
    (when
      (and (consp x) (eq (qfirst x) 'def) (consp (qrest x))
           (consp (qsecond x)) (consp (qcddr x))
           (consp (qcdddr x))
           (consp (qcddddr x))
           (eq (qrest (qcddddr x)) nil))
      (setq op (qcaadr x))
      (setq args (qcdadr x))
      (setq signature (qthird x))
      (setq body (qfirst (qcddddr x)))

```

```
(when
  (and (null (|get| op '|modemap| env))
        (null args)
        (null (|get| op '|mode| env))
        (consp signature)
        (eq (qrest signature) nil)
        (null (qfirst signature))))
  body))))
```

defplist compCase plist

We set up the `compCase` function to handle the `case` keyword by setting the `special` keyword on the `case` symbol property list.

— **postvars** —

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

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 p248]
 [compCase1 p284]
 [coerce p351]

— **defun compCase** —

```
(defun |compCase| (form mode env)
  (let (mp td)
    (setq mp (third form))
    (setq env (|addDomain| mp env))
    (when (setq td (|compCase1| (second form) mp env)) (|coerce| td mode))))
```

defun compCase1

```
[comp p590]
[getModemapList p259]
[nreverse0 p??]
[modeEqual p362]
[$Boolean p??]
[$EmptyMode p172]
```

— defun compCase1 —

```
(defun |compCase1| (form mode env)
  (let (xp mp ep map tmp3 tmp5 tmp6 u fn)
    (declare (special |$Boolean| |$EmptyMode|))
    (when (setq tmp3 (|comp| form |$EmptyMode| env))
      (setq xp (first tmp3))
      (setq mp (second tmp3))
      (setq ep (third tmp3))
      (when
        (setq u
          (dolist (modemap (|getModemapList| 'case 2 ep) (nreverse0 tmp5))
            (setq map (first modemap))
            (when
              (and (consp map) (consp (qrest map)) (consp (qcddr map))
                (consp (qcdddr map))
                (eq (qcdddr map) nil)
                (|modeEqual| (fourth map) mode)
                (|modeEqual| (third map) mp))
              (push (second modemap) tmp5))))
          (when
            (setq fn
              (dolist (onepair u tmp6)
                (when (first onepair) (setq tmp6 (or tmp6 (second onepair))))))
              (list (list 'call fn xp) |$Boolean| ep))))))
```

—————

defplist compCat plist

We set up the `compCat` function to handle the `Record` keyword by setting the `special` keyword on the `Record` symbol property list.

— postvars —

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

—————

defplist compCat plist

We set up the `compCat` function to handle the `Mapping` keyword by setting the `special` keyword on the `Mapping` symbol property list.

— **postvars** —

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

—————

defplist compCat plist

We set up the `compCat` function to handle the `Union` keyword by setting the `special` keyword on the `Union` symbol property list.

— **postvars** —

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

—————

defun compCat

[getl p??]

— **defun compCat** —

```
(defun |compCat| (form mode env)
  (declare (ignore mode))
  (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 env))
      (setq funList (first tmp1))
      (setq env (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 env))))
```

defplist compCategory plist

We set up the `compCategory` function to handle the `category` keyword by setting the `special` keyword on the `category` symbol property list.

— postvars —

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

defun compCategory

```
[resolve p361]
[compCategoryItem p287]
[mkExplicitCategoryFunction p288]
[systemErrorHere p??]
[$sigList p??]
[$atList p??]
[$top-level p??]
[$sigList p??]
[$atList p??]
```

— defun compCategory —

```
(defun |compCategory| (form mode env)
  (let ($top_level |$sigList| |$atList| domainOrPackage z rep)
    (declare (special $top_level |$sigList| |$atList|))
    (setq $top_level t)
    (cond
      ((and
        (equal (setq mode (|resolve| mode (list '|Category|)))
              (list '|Category|))
        (consp form)
        (eq (qfirst form) 'category)
        (consp (qrest form)))
        (setq domainOrPackage (second form))
        (setq z (qcddr form))
        (setq |$sigList| nil)
        (setq |$atList| nil)
        (dolist (x z) (|compCategoryItem| x nil))
        (setq rep
          (|mkExplicitCategoryFunction| domainOrPackage |$sigList| |$atList|))
        (list rep mode env))
```

```
(t
  (|systemErrorHere| "compCategory")))))
```

defun compCategoryItem

```
[compCategoryItem p287]
[mkpf p??]
[$sigList p??]
[$atList p??]
```

— defun compCategoryItem —

```
(defun |compCategoryItem| (x predl)
  (let (p e a b c predlp pred y z op sig)
    (declare (special |$sigList| |$atList|))
    (cond
      ((null x) nil)
      ; 1. if x is a conditional expression, recurse; otherwise, form the predicate
      ((and (consp x) (eq (qfirst x) 'cond)
        (consp (qrest x)) (eq (qcddr x) nil)
        (consp (qsecond x))
        (consp (qcadadr x))
        (eq (qcddadr x) nil))
        (setq p (qcaadr x))
        (setq e (qcadadr x))
        (setq predlp (cons p predl))
        (cond
          ((and (consp e) (eq (qfirst e) 'progn))
            (setq z (qrest e))
            (dolist (y z) (|compCategoryItem| y predlp)))
          (t (|compCategoryItem| e predlp))))
      ((and (consp x) (eq (qfirst x) 'if) (consp (qrest x))
        (consp (qcddr x)) (consp (qcdddr x))
        (eq (qcdddr x) nil))
        (setq a (qsecond x))
        (setq b (qthird x))
        (setq c (qfourth x))
        (setq predlp (cons a predl))
        (unless (eq b '|noBranch|)
          (cond
            ((and (consp b) (eq (qfirst b) 'progn))
              (setq z (qrest b))
              (dolist (y z) (|compCategoryItem| y predlp)))
            (t (|compCategoryItem| b predlp))))
        (cond
```

```

((eq c '|noBranch|) nil)
(t
  (setq predlp (cons (list '|not| a) predl))
  (cond
    ((and (consp c) (eq (qfirst c) 'progn))
      (setq z (qrest c))
      (dolist (y z) (|compCategoryItem| y predlp)))
    (t (|compCategoryItem| c predlp))))))
(t
  (setq pred (if predl (mkpf predl 'and) t))
  (cond
; 2. if attribute, push it and return
    ((and (consp x) (eq (qfirst x) 'attribute)
      (consp (qrest x)) (eq (qcddr x) nil))
      (setq y (qsecond x))
      (push (mkq (list y pred)) |$atList|))
; 3. it may be a list, with PROGN as the CAR, and some information as the CDR
    ((and (consp x) (eq (qfirst x) 'progn))
      (setq z (qrest x))
      (dolist (u z) (|compCategoryItem| u predl)))
    (t
; 4. otherwise, x gives a signature for a single operator name or a list of
; names; if a list of names, recurse
      (cond ((eq (car x) 'signature) (car x)))
      (setq op (cadr x))
      (setq sig (cddr x))
      (cond
        ((null (atom op))
          (dolist (y op)
            (|compCategoryItem| (cons 'signature (cons y sig)) predl)))
        (t
; 5. branch on a single type or a signature %with source and target
          (push (mkq (list (cdr x) pred)) |$sigList|)))))))))

```

defun mkExplicitCategoryFunction

```

[mkq p??]
[union p??]
[mustInstantiate p289]
[remdup p??]
[identp p??]
[wrapDomainSub p290]

```

— defun mkExplicitCategoryFunction —

```

(defun |mkExplicitCategoryFunction| (domainOrPackage sigList atList)
  (let (body sig parameters)
    (setq body
      (list '|mkCategory| (mkq domainOrPackage)
        (cons 'list (reverse sigList))
        (cons 'list (reverse atList))
        (mkq
          (let (result)
            (loop for item in sigList
              do
                (setq sig (car (cdaadr item)))
                (setq result
                  (|union| result
                    (loop for d in sig
                      when (|mustInstantiate| d)
                        collect d))))
            result)))
          nil)))
    (setq parameters
      (remdup
        (let (result)
          (loop for item in sigList
            do
              (setq sig (car (cdaadr item)))
              (setq result
                (append result
                  (loop for x in sig
                    when (and (identp x) (not (eq x '$)))
                      collect x))))
            result)))
        (|wrapDomainSub| parameters body)))

```

defun mustInstantiate

```

[getl p??]
[$DummyFunctorNames p??]

```

— defun mustInstantiate —

```

(defun |mustInstantiate| (d)
  (declare (special |$DummyFunctorNames|))
  (and (consp d)
    (null (or (member (qfirst d) |$DummyFunctorNames|)
      (getl (qfirst d) '|makeFunctionList|))))))

```

defun wrapDomainSub

— defun wrapDomainSub —

```
(defun |wrapDomainSub| (parameters x)
  (list '|DomainSubstitutionMacro| parameters x))
```

defplist compColon plist

We set up the `compColon` function to handle the `:` keyword by setting the `special` keyword on the `:` symbol property list.

— postvars —

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

defun compColon

```
[compColonInside p596]
[assoc p??]
[getDomainsInScope p250]
[isDomainForm p344]
[compColon member (vol5)]
[addDomain p248]
[isCategoryForm p??]
[unknownTypeError p249]
[compColon p290]
[eqsubstlist p??]
[take p??]
[length p??]
[nreverse0 p??]
[getmode p??]
[systemErrorHere p??]
[put p??]
[makeCategoryForm p293]
[genSomeVariable p??]
```

```

[$lhsOfColon p??]
[$noEnv p??]
[$insideFunctorIfTrue p??]
[$bootStrapMode p??]
[$FormalMapVariableList p266]
[$insideCategoryIfTrue p??]
[$insideExpressionIfTrue p??]

```

— defun compColon —

```

(defun |compColon| (form mode env)
  (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 form))
    (setq argt (third form))
    (if |$insideExpressionIfTrue|
        (|compColonInside| argf mode env argt)
        (progn
          (setq |$lhsOfColon| argf)
          (setq argt
            (cond
              ((and (atom argt)
                    (setq tprime (|assoc| argt (|getDomainsInScope| env))))
               tprime)
              ((and (|isDomainForm| argt env) (null |$insideCategoryIfTrue|))
               (unless (|member| argt (|getDomainsInScope| env))
                 (setq env (|addDomain| argt env)))
               argt)
              ((or (|isDomainForm| argt env) (|isCategoryForm| argt env))
               argt)
              ((and (consp argt) (eq (qfirst argt) '|Mapping|)
                   (progn
                     (setq tmp2 (qrest argt))
                     (and (consp tmp2)
                         (progn
                           (setq mprime (qfirst tmp2))
                           (setq r (qrest tmp2))
                           t))))
               argt)
              (t
               (|unknownTypeError| argt)
               argt)))
          (cond
            ((eq (car argf) 'listof)
             (dolist (x (cdr argf) td)
              (setq td (|compColon| (list '|:| x argt) mode env))

```

```

    (setq env (third td))))
(t
 (setq env
  (cond
   ((and (consp argf)
        (progn
         (setq op (qfirst argf))
         (setq argl (qrest argf))
         t)
        (null (and (consp argt) (eq (qfirst argt) '|Mapping|))))
   (setq newTarget
    (eqsubstlist (take (|#| argl) |$FormalMapVariableList|)
     (dolist (x argl (nreverse0 g2))
      (setq g2
       (cons
        (cond
         ((and (consp x) (eq (qfirst x) '|:|)
          (progn
           (setq tmp2 (qrest x))
           (and (consp tmp2)
                (progn
                 (setq a (qfirst tmp2))
                 (setq tmp3 (qrest tmp2))
                 (and (consp tmp3)
                      (eq (qrest tmp3) nil)
                      (progn
                       (setq mode (qfirst tmp3))
                       t))))))
          a)
         (t x))
        g2)))
    argt))
 (setq signature
  (cons '|Mapping|
   (cons newTarget
    (dolist (x argl (nreverse0 g5))
     (setq g5
      (cons
       (cond
        ((and (consp x) (eq (qfirst x) '|:|)
         (progn
          (setq tmp2 (qrest x))
          (and (consp tmp2)
               (progn
                (setq a (qfirst tmp2))
                (setq tmp3 (qrest tmp2))
                (and (consp tmp3)
                     (eq (qrest tmp3) nil)
                     (progn
                      (setq mode (qfirst tmp3))
                      t))))))
         (t x))
       g5)))
    signature)))

```



```

t))))))
mode)
(t
  (or (|getmode| x env)
      (|systemErrorHere| "compColonOld"))))
g5))))))
(|put| op '|mode| signature env))
(t (|put| argf '|mode| argt env)))
(cond
  ((and (null |$bootStrapMode|) |$insideFunctorIfTrue|
        (progn
          (setq tmp2 (|makeCategoryForm| argt env))
          (and (consp tmp2)
                (progn
                  (setq catform (qfirst tmp2))
                  (setq tmp3 (qrest tmp2))
                  (and (consp tmp3)
                        (eq (qrest tmp3) nil)
                        (progn
                          (setq env (qfirst tmp3))
                          t)))))))
   (setq env
    (|put| argf '|value| (list (|genSomeVariable|) argt |$noEnv|)
      env)))
  (list '|/throwAway| (|getmode| argf env) env ))))))))

```

defun makeCategoryForm

[isCategoryForm p??]
 [compOrCroak p588]
 [\$EmptyMode p172]

— defun makeCategoryForm —

```

(defun |makeCategoryForm| (c env)
  (let (tmp1)
    (declare (special |$EmptyMode|))
    (when (|isCategoryForm| c env)
      (setq tmp1 (|compOrCroak| c |$EmptyMode| env))
      (list (first tmp1) (third tmp1)))))

```

defplist compCons plist

We set up the `compCons` function to handle the `cons` keyword by setting the `special` keyword on the `cons` symbol property list.

— postvars —

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

—————

defun compCons

[[compCons1](#) p294]
[[compForm](#) p602]

— defun compCons —

```
(defun |compCons| (form mode env)
  (or (|compCons1| form mode env) (|compForm| form mode env)))
```

—————

defun compCons1

[[comp](#) p590]
[[convert](#) p600]
[[\\$EmptyMode](#) p172]

— defun compCons1 —

```
(defun |compCons1| (arg mode env)
  (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| env))
      (setq x (first tmp1))
      (setq mx (second tmp1))
      (setq env (third tmp1))
      (cond
        ((null y)
         (|convert| (list (list 'list x) (list '|List| mx) env ) mode))
        (t
```

```

(when (setq yt (|comp| y |$EmptyMode| env))
  (setq y (first yt))
  (setq my (second yt))
  (setq env (third yt))
  (setq td
    (cond
      ((and (consp my) (eq (qfirst my) '|List|) (consp (qrest 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 env (third tmp1))
              (cond
                ((and (consp (car ytp)) (eq (qfirst (car ytp)) 'list))
                  (list (cons 'list (cons x (cdr (car ytp)))) mr env))
                (t
                  (list (list 'cons x (car ytp)) mr env)))))))
      (t
        (list (list 'cons x y) (list '|Pair| mx my) env ))))
    (|convert| td mode))))))

```

defplist compConstruct plist

We set up the `compConstruct` function to handle the `construct` keyword by setting the `special` keyword on the `construct` symbol property list.

— postvars —

```

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

```

defun compConstruct

```

[modeIsAggregateOf p??]
[compList p602]
[convert p600]
[compForm p602]
[compVector p349]
[getDomainsInScope p250]

```

— defun compConstruct —

```

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

```

defplist compConstructorCategory plist

We set up the `compConstructorCategory` function to handle the `ListCategory` keyword by setting the `special` keyword on the `ListCategory` symbol property list.

— postvars —

```

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

```

defplist compConstructorCategory plist

We set up the `compConstructorCategory` function to handle the `RecordCategory` keyword by setting the `special` keyword on the `RecordCategory` symbol property list.

— postvars —

```

(eval-when (eval load)

```

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

deflist compConstructorCategory plist

We set up the `compConstructorCategory` function to handle the `UnionCategory` keyword by setting the `special` keyword on the `UnionCategory` symbol property list.

— postvars —

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

deflist compConstructorCategory plist

We set up the `compConstructorCategory` function to handle the `VectorCategory` keyword by setting the `special` keyword on the `VectorCategory` symbol property list.

— postvars —

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

defun compConstructorCategory

[[resolve p361](#)]
 [\$Category p??]

— defun compConstructorCategory —

```
(defun |compConstructorCategory| (form mode env)
  (declare (special |$Category|))
  (list form (|resolve| |$Category| mode) env))
```

defun getAbbreviation

```
[constructor? p??]
[assq p??]
[mkAbbrev p298]
[rplac p??]
[$abbreviationTable p??]
[$abbreviationTable p??]
```

— defun getAbbreviation —

```
(defun |getAbbreviation| (name c)
  (let (cname x n upc newAbbreviation)
    (declare (special |$abbreviationTable|))
    (setq cname (|constructor?| name))
    (cond
      ((setq x (assq cname |$abbreviationTable|))
       (cond
         ((setq n (assq name (cdr x)))
          (cond
            ((setq upc (assq c (cdr n)))
             (cdr upc))
            (t
             (setq newAbbreviation (|mkAbbrev| x cname))
             (rplac (cdr n) (cons (cons c newAbbreviation) (cdr n)))
             newAbbreviation)))
         (t
          (setq newAbbreviation (|mkAbbrev| x x))
          (rplac (cdr x)
                 (cons (cons name (list (cons c newAbbreviation))) (cdr x)))
                 newAbbreviation)))
      (t
       (setq |$abbreviationTable|
              (cons (list cname (list name (cons c cname))) |$abbreviationTable|)
                    cname))))))
```

—

defun mkAbbrev

```
[addSuffix p299]
[alistSize p299]
```

— defun mkAbbrev —

```
(defun |mkAbbrev| (x z)
  (|addSuffix| (|alistSize| (cdr x)) z))
```

defun addSuffix

— defun addSuffix —

```

(defun |addSuffix| (n u)
  (let (s)
    (if (alpha-char-p (elt (spadlet s (stringimage u)) (maxindex s)))
        (intern (strconc s (stringimage n)))
        (intern1 (strconc s (stringimage '|;|) (stringimage n))))))

```

defun alistSize

— defun alistSize —

```

(defun |alistSize| (c)
  (labels (
    (count (x level)
      (cond
        ((eq1 level 2) (|#| x))
        ((null x) 0)
        (+ (count (cdar x) (1+ level))
           (count (cdr x) level))))))
    (count c 1)))

```

defun getSignatureFromMode

```

[getmode p??]
[opOf p??]
[length p??]
[stackAndThrow p??]
[eqsubstlist p??]
[take p??]
[$FormalMapVariableList p266]

```

— defun getSignatureFromMode —

```
(defun |getSignatureFromMode| (form env)
  (let (tmp1 signature)
    (declare (special |$FormalMapVariableList|))
    (setq tmp1 (|getmode| (|opOf| form) env))
    (when (and (consp tmp1) (eq (qfirst tmp1) '|Mapping|))
      (setq signature (qrest tmp1))
      (if (not (eql (|#| form) (|#| signature)))
          (|stackAndThrow| (list '|Wrong number of arguments: | form))
          (eqsubstlist (cdr form)
                        (take (|#| (cdr form)) |$FormalMapVariableList|
                             signature))))))
```

—————

defun getSpecialCaseAssoc

```
[|$functorForm p??|
 |$functorSpecialCases p??|
```

— defun getSpecialCaseAssoc —

```
(defun |getSpecialCaseAssoc| ()
  (declare (special |$functorSpecialCases| |$functorForm|))
  (loop for r in (rest |$functorForm|)
        for z in (rest |$functorSpecialCases|)
        when z
        collect (cons r z)))
```

—————

defun addArgumentConditions

```
[mkq p??]
[|systemErrorHere p??|
|$true p??|
|$functionName p??|
|$body p??|
|$argumentConditionList p??|
|$argumentConditionList p??|
```

— defun addArgumentConditions —


```

(defun |addArgumentConditions| (|$body| |$functionName|)
  (declare (special |$body| |$functionName| |$argumentConditionList| |$true|))
  (labels (
    (fn (clist)
      (let (n untypedCondition typedCondition)
        (cond
          ((and (consp clist) (consp (qfirst clist)) (consp (qcdar clist))
              (consp (qcddar clist))
              (eq (qcdddar clist) nil))
            (setq n (qcaar clist))
            (setq untypedCondition (qcadar clist))
            (setq typedCondition (qcaddar clist))
            (list 'cond
              (list typedCondition (fn (cdr clist)))
              (list |$true|
                (list '|argumentDataError| n
                  (mkq untypedCondition) (mkq |$functionName|))))))
          ((null clist) |$body|)
          (t (|systemErrorHere| "addArgumentConditions")))))
    (if |$argumentConditionList|
      (fn |$argumentConditionList|
        (|$body|)))
    )
  )

```

defun stripOffSubdomainConditions

```

[assoc p??]
[mkpf p??]
|$argumentConditionList p??]
|$argumentConditionList p??]

```

— defun stripOffSubdomainConditions —

```

(defun |stripOffSubdomainConditions| (margl argl)
  (let (pair (i 0))
    (declare (special |$argumentConditionList|))
    (loop for x in margl for arg in argl
      do (incf i)
      collect
      (cond
        ((and (consp x) (eq (qfirst x) '|SubDomain|) (consp (qrest x))
              (consp (qcddr x)) (eq (qcdddr x) nil))
          (cond
            ((setq pair (|assoc| i |$argumentConditionList|))
              (rplac (cadr pair) (mkpf (list (third x) (cadr pair)) 'and))
            )
          )
        )
    )
  )

```

```

      (second x))
    (t
      (setq |$argumentConditionList|
        (cons (list i arg (third x)) |$argumentConditionList|))
      (second x))))
    (t x))))

```

defun stripOffArgumentConditions

```

[$argumentConditionList p??]
[$argumentConditionList p??]

```

— defun stripOffArgumentConditions —

```

(defun |stripOffArgumentConditions| (arg1)
  (let (condition (i 0))
    (declare (special |$argumentConditionList|))
    (loop for x in arg1
      do (incf i)
      collect
      (cond
        ((and (consp x) (eq (qfirst x) '|\\|') (consp (qrest x))
          (consp (qcddr x)) (eq (qcdddr x) nil))
          (setq condition (subst '|#1| (second x) (third x) :test #'equal))
          (setq |$argumentConditionList|
            (cons (list i (second x) condition) |$argumentConditionList|))
          (second x))
        (t x)))))

```

defun getSignature

Try to return a signature. If there isn't one, complain and return nil. If there are more than one then remove any that are subsumed. If there is still more than one complain else return the only signature. [get p??]

```

[length p??]
[remdup p??]
[knownInfo p??]
[getmode p??]
[say p??]
[printSignature p??]

```

```
[SourceLevelSubsume p??]
[stackSemanticError p??]
[$e p??]
```

— defun getSignature —

```
(defun |getSignature| (op argModeList |$e|)
  (declare (special |$e|))
  (let (mmList pred u tmp1 dc sig sigl)
    (setq mmList (|get| op '|modemap| |$e|))
    (cond
      ((eq 1
        (|#| (setq sigl (remdup
          (loop for item in mmList
            do
              (setq dc (caar item))
              (setq sig (cdar item))
              (setq pred (caadr item))
              when (and (eq dc '$) (equal (cdr sig) argModeList) (|knownInfo| pred))
                collect sig))))))
        (car sigl))
      ((null sigl)
        (cond
          ((progn
              (setq tmp1 (setq u (|getmode| op |$e|)))
              (and (consp tmp1) (eq (qfirst tmp1) '|Mapping|)))
            (qrest tmp1))
          (t
            (say "***** USER ERROR *****")
            (say "available signatures for " op ": ")
            (cond
              ((null mmList) (say "    NONE"))
              (t
                (loop for item in mmList
                  do (|printSignature| '|      | op (cdar item)))
                (|printSignature| '|NEED | op (cons '? argModeList))))
            nil)))
      (t
        ; Before we complain about duplicate signatures, we should
        ; check that we do not have for example, a partial - as
        ; well as a total one. SourceLevelSubsume should do this
        (loop for u in sigl do
          (loop for v in sigl
            when (null (equal u v))
            do (when (|SourceLevelSubsume| u v) (setq sigl (|delete| v sigl)))))
        (cond
          ((eq 1 (|#| sigl)) (car sigl))
          (t
            (|stackSemanticError|
```

```
(list '|duplicate signatures for | op '|: | argModeList) nil))))))
```

defun checkAndDeclare

```
[getArgumentMode p305]
[modeEqual p362]
[put p??]
[sayBrightly p??]
[bright p??]
```

— defun checkAndDeclare —

```
(defun |checkAndDeclare| (argl form sig env)
  (let (m1 stack)
    (loop for a in argl for m in (rest sig)
      do
        (if (setq m1 (|getArgumentMode| a env))
          (if (null (|modeEqual| m1 m))
            (setq stack
              (cons '| | (append (|bright| a)
                (cons "must have type "
                  (cons m
                    (cons " not "
                      (cons m1
                        (cons '|%1| stack))))))))
            (setq env (|put| a '|mode| m env)))
          (when stack
            (|sayBrightly|
              (cons " Parameters of "
                (append (|bright| (car form))
                  (cons " are of wrong type:"
                    (cons '|%1| stack))))))
            env)))
```

defun hasSigInTargetCategory

```
[getArgumentMode p305]
[remdup p??]
[length p??]
[getSignatureFromMode p299]
[stackWarning p??]
```

```
[compareMode2Arg p??]
[bright p??]
[$domainShell p??]
```

— **defun hasSigInTargetCategory** —

```
(defun |hasSigInTargetCategory| (argl form opsig env)
  (labels (
    (fn (opName sig opsig mList form)
      (declare (special |$op|))
      (and
        (and
          (and (equal opName |$op|) (equal (|#| sig) (|#| form)))
          (or (null opsig) (equal opsig (car sig))))
        (let ((result t))
          (loop for x in mList for y in (rest sig)
            do (setq result (and result (or (null x) (|modeEqual| x y))))
          result))))
    (let (mList potentialSigList c sig)
      (declare (special |$domainShell|))
      (setq mList
        (loop for x in argl
          collect (|getArgumentMode| x env)))
      (setq potentialSigList
        (remdup
          (loop for item in (elt |$domainShell| 1)
            when (fn (caar item) (cadar item) opsig mList form)
              collect (cadar item))))
      (setq c (|#| potentialSigList))
      (cond
        ((eql 1 c) (car potentialSigList))
        ((eql 0 c)
          (when (equal (|#| (setq sig (|getSignatureFromMode| form env))) (|#| form))
            sig))
        ((> c 1)
          (setq sig (car potentialSigList))
          (|stackWarning|
            (cons '|signature of lhs not unique:|
              (append (|bright| sig) (list '|chosen|))))
          sig)
        (t nil))))))
```

—————

defun getArgumentMode

```
[get p??]
```

— defun getArgumentMode —

```
(defun |getArgumentMode| (x e)
  (if (stringp x) x (|get| x '|mode| e)))
```

—————

defplist compElt plist

We set up the `compElt` function to handle the `elt` keyword by setting the `special` keyword on the `elt` symbol property list.

— postvars —

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

—————

defun compElt

```
[compForm p602]
[isDomainForm p344]
[addDomain p248]
[getModemapListFromDomain p260]
[length p??]
[stackMessage p??]
[stackWarning p??]
[convert p600]
[opOf p??]
[getDeltaEntry p??]
[$One p??]
[$Zero p??]
```

— defun compElt —

```
(defun |compElt| (form mode env)
  (let (aDomain anOp mmList n modemap sig pred val)
    (declare (special |$One| |$Zero|))
    (setq anOp (third form))
    (setq aDomain (second form))
    (cond
      ((null (and (consp form) (eq (qfirst form) '|elt|)
                  (consp (qrest form)) (consp (qcddr form))
                  (eq (qcdddr form) nil))))
```

```

(|compForm| form mode env))
((eq aDomain '|Lisp|)
 (list (cond
        ((equal anOp '$Zero|) 0)
        ((equal anOp '$One|) 1)
        (t anOp))
      mode env))
(|isDomainForm| aDomain env)
(setq env (|addDomain| aDomain env))
(setq mmList (|getModemapListFromDomain| anOp 0 aDomain env))
(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 (not (eql (|#| sig) 2))
              (null (and (consp val) (eq (qfirst val) '|elt|))))
  (setq val (|genDeltaEntry| (cons (|opOf| anOp) modemap)))
  (|convert| (list (list '|call| val) (second sig) env) mode))))
(t
 (|compForm| form mode env))))

```

defplist compExit plist

We set up the `compExit` function to handle the `exit` keyword by setting the `special` keyword on the `exit` symbol property list.

— **postvars** —

```

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

```

defun compExit

```
[comp p590]
[modifyModeStack p625]
[stackMessageIfNone p??]
[$exitModeStack p??]
```

— defun compExit —

```
(defun |compExit| (form mode env)
  (let (exitForm index m1 u)
    (declare (special |$exitModeStack|))
    (setq index (1- (second form)))
    (setq exitForm (third form))
    (cond
      ((null |$exitModeStack|)
       (|comp| exitForm mode env))
      (t
       (setq m1 (elt |$exitModeStack| index))
       (setq u (|comp| exitForm m1 env))
       (cond
         (u
          (|modifyModeStack| (second u) index)
          (list (list ' |TAGGEDexit| index u) mode env))
         (t
          (|stackMessageIfNone|
           (list ' |cannot compile exit expression| exitForm ' |in mode| m1))))))))))
```

defplist compHas plist

We set up the `compHas` function to handle the `has` keyword by setting the `special` keyword on the `has` symbol property list.

— postvars —

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

defun compHas

```
[chaseInferences p??]
[compHasFormat p309]
[coerce p351]
[$e p??]
[$e p??]
[$Boolean p??]
```

— defun compHas —

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

—————

defun compHasFormat

```
[take p??]
[length p??]
[sublislis p??]
[comp p590]
[mkList p310]
[mkDomainConstructor p??]
[isDomainForm p344]
[$FormalMapVariableList p266]
[$EmptyMode p172]
[$e p??]
[$form p??]
[$EmptyEnvironment p??]
```

— defun compHasFormat —

```
(defun |compHasFormat| (pred)
  (let (olda b argl formals tmp1 a)
    (declare (special |$EmptyEnvironment| |$e| |$EmptyMode|
                      |$FormalMapVariableList| |$form|))
    (when (eq (car pred) '|has|) (car pred))
    (setq olda (second pred))
    (setq b (third pred))
```

```

(setq arg1 (rest |$form|))
(setq formals (take (|#| arg1) |$FormalMapVariableList|))
(setq a (sublislis arg1 formals olda))
(setq tmp1 (|comp| a |$EmptyModel| |$e|))
(when tmp1
  (setq a (car tmp1))
  (setq a (sublislis formals arg1 a))
  (cond
    ((and (consp b) (eq (qfirst b) 'attribute) (consp (qrest b))
      (eq (qcddr b) nil))
      (list '|HasAttribute| a (list 'quote (qsecond b))))
    ((and (consp b) (eq (qfirst b) 'signature) (consp (qrest b))
      (consp (qcddr b)) (eq (qcdddr b) NIL))
      (list '|HasSignature| a
        (|mkList|
          (list (MKQ (qsecond b))
            (|mkList|
              (loop for type in (qthird b)
                collect (|mkDomainConstructor| type)))))))
    ((|isDomainForm| b |$EmptyEnvironment|)
      (list 'equal a b))
    (t
      (list '|HasCategory| a (|mkDomainConstructor| b))))))

```

defun mkList

— defun mkList —

```

(defun |mkList| (u)
  (when u (cons 'list u)))

```

defplist compIf plist

We set up the `compIf` function to handle the `if` keyword by setting the `special` keyword on the `if` symbol property list.

— postvars —

```

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

```

defun compIf

```
[canReturn p312]
[intersectionEnvironment p??]
[compBoolean p314]
[compFromIf p312]
[resolve p361]
[coerce p351]
[quotify p??]
[$Boolean p??]
```

— defun compIf —

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

defun compFromIf

[comp p590]

— defun compFromIf —

```
(defun |compFromIf| (a m env)
  (if (eq a '|noBranch|)
      (list '|noBranch| m env)
      (|comp| a m env)))
```

—————

defun canReturn

[say p??]

[canReturn p312]

[systemErrorHere p??]

— defun canReturn —

```
(defun |canReturn| (expr level exitCount ValueFlag)
  (labels (
    (findThrow (gs expr level exitCount ValueFlag)
      (cond
        ((atom expr) nil)
        ((and (consp expr) (eq (qfirst expr) 'throw) (consp (qrest expr))
          (equal (qsecond expr) gs) (consp (qcddr expr))
          (eq (qcddr expr) nil))
        t)
      ((and (consp expr) (eq (qfirst expr) 'seq))
        (let (result)
          (loop for u in (qrest expr)
            do (setq result
              (or result
                (findThrow gs u (1+ level) exitCount ValueFlag))))
          result)))
    (t
      (let (result)
        (loop for u in (rest expr)
          do (setq result
            (or result
              (findThrow gs u level exitCount ValueFlag))))
        result))))))
  (let (op gs)
    (cond
      ((atom expr) (and ValueFlag (equal level exitCount)))
```

```

((eq (setq op (car expr)) 'quote) (and ValueFlag (equal level exitCount)))
((eq op '|TAGGEDexit|)
 (cond
  ((and (consp expr) (consp (qrest expr)) (consp (qcddr expr))
        (eq (qcdddr expr) nil))
   (|canReturn| (car (third expr)) level (second expr)
                 (equal (second expr) level))))
  ((and (equal level exitCount) (null ValueFlag))
   nil)
  ((eq op 'seq)
   (let (result)
     (loop for u in (rest expr)
            do (setq result (or result (|canReturn| u (1+ level) exitCount nil))))
    result))
  ((eq op '|TAGGEDreturn|) nil)
  ((eq op 'catch)
   (cond
    ((findThrow (second expr) (third expr) level
                 exitCount ValueFlag)
     t)
    (t
     (|canReturn| (third expr) level exitCount ValueFlag))))
  ((eq op 'cond)
   (cond
    ((equal level exitCount)
     (let (result)
       (loop for u in (rest expr)
              do (setq result (or result
                                   (|canReturn| (|last| u) level exitCount ValueFlag))))
      result))
    (t
     (let (outer)
       (loop for v in (rest expr)
              do (setq outer (or outer
                                   (let (inner)
                                     (loop for u in v
                                            do (setq inner
                                                  (or inner
                                                       (findThrow gs u level exitCount ValueFlag))))
                                      inner))))
        outer))))
    ((eq op 'if)
     (and (consp expr) (consp (qrest expr)) (consp (qcddr expr))
           (consp (qcdddr expr))
           (eq (qcddddd expr) nil))
     (cond
      ((null (|canReturn| (second expr) 0 0 t))
       (say "IF statement can not cause consequents to be executed")
       (|pp| expr)))
      (or (|canReturn| (second expr) level exitCount nil)

```

```

      (|canReturn| (third expr) level exitCount ValueFlag)
      (|canReturn| (fourth expr) level exitCount ValueFlag)))
((atom op)
 (let ((result t))
  (loop for u in expr
    do (setq result
      (and result (|canReturn| u level exitCount ValueFlag))))
  result))
((and (consp op) (eq (qfirst op) 'xlam) (consp (qrest op))
  (consp (qcddr op)) (eq (qcdddr op) nil))
 (let ((result t))
  (loop for u in expr
    do (setq result
      (and result (|canReturn| u level exitCount ValueFlag))))
  result))
(t (|systemErrorHere| "canReturn")))))))

```

defun compBoolean

[comp p590]
 [getSuccessEnvironment p314]
 [getInverseEnvironment p316]

— defun compBoolean —

```

(defun |compBoolean| (p mode env)
  (let (tmp1 pp)
    (when (setq tmp1 (OR (|comp| p mode env)))
      (setq pp (car tmp1))
      (setq mode (cadr tmp1))
      (setq env (caddr tmp1))
      (list pp mode (|getSuccessEnvironment| p env)
        (|getInverseEnvironment| p env)))))

```

defun getSuccessEnvironment

[isDomainForm p344]
 [put p??]
 [identp p??]
 [getProplist p??]
 [comp p590]

```
[consProplistOf p??]
[removeEnv p??]
[addBinding p??]
[get p??]
[$EmptyEnvironment p??]
[$EmptyMode p172]
```

— defun `getSuccessEnvironment` —

```
(defun |getSuccessEnvironment| (a env)
  (let (id currentProplist tt newProplist x m)
    (declare (special |$EmptyMode| |$EmptyEnvironment|))
    (cond
      ((and (consp a) (eq (qfirst a) '|has|) (consp (qrest a))
        (consp (qcddr a)) (eq (qcdddr a) nil))
        (if
          (and (identp (second a)) (|isDomainForm| (third a) |$EmptyEnvironment|))
          (|put| (second a) '|specialCase| (third a) env)
          env))
      ((and (consp a) (eq (qfirst a) '|is|) (consp (qrest a))
        (consp (qcddr a)) (eq (qcdddr a) nil))
        (setq id (qsecond a))
        (setq m (qthird a))
        (cond
          ((and (identp id) (|isDomainForm| m |$EmptyEnvironment|))
            (setq env (|put| id '|specialCase| m env))
            (setq currentProplist (|getProplist| id env))
            (setq tt (|comp| m |$EmptyMode| env))
            (when tt
              (setq env (caddr tt))
              (setq newProplist
                (|consProplistOf| id currentProplist '|value|
                  (cons m (cdr (|removeEnv| tt))))))
              (|addBinding| id newProplist env)))
          (t env)))
      ((and (consp a) (eq (qfirst a) '|case|) (consp (qrest a))
        (consp (qcddr a)) (eq (qcdddr a) nil)
        (identp (qsecond a)))
        (setq x (qsecond a))
        (setq m (qthird a))
        (|put| x '|condition| (cons a (|get| x '|condition| env)) env))
      (t env))))
```

defun getInverseEnvironment

```

[identp p??]
[isDomainForm p344]
[put p??]
[get p??]
[member p??]
[mkpf p??]
[delete p??]
[getUnionMode p317]
[$EmptyEnvironment p??]

```

— **defun getInverseEnvironment** —

```

(defun |getInverseEnvironment| (a env)
  (let (op argl x m oldpred tmp1 zz newpred)
    (declare (special |$EmptyEnvironment|))
    (cond
      ((atom a) env)
      (t
       (setq op (car a))
       (setq argl (cdr a))
       (cond
         ((eq op '|has|)
          (setq x (car argl))
          (setq m (cadr argl))
          (cond
            ((and (identp x) (|isDomainForm| m |$EmptyEnvironment|))
             (|put| x '|specialCase| m env))
            (t env)))
         ((and (consp a) (eq (qfirst a) '|case|) (consp (qrest a))
              (consp (qcddr a)) (eq (qcdddr a) nil)
              (identp (qsecond a)))
          (setq x (qsecond a))
          (setq m (qthird a))
          (setq tmp1 (|get| x '|condition| env))
          (cond
            ((and tmp1 (consp tmp1) (eq (qrest tmp1) nil) (consp (qfirst tmp1))
              (eq (qcaar tmp1) '|or|) (|member| a (qcdr tmp1)))
             (setq oldpred (qcdr tmp1))
             (|put| x '|condition| (list (mkpf (|delete| a oldpred) '|or|) env))
            (t
             (setq tmp1 (|getUnionMode| x env))
             (setq zz (|delete| m (qrest tmp1)))
             (loop for u in zz
                   when (and (consp u) (eq (qfirst u) '|:|)
                           (consp (qrest u)) (equal (qsecond u) m))
                   do (setq zz (|delete| u zz)))
             (setq newpred

```



```

      (mkpf (loop for mp in zz collect (list '|case| x mp)) 'or))
      (|put| x '|condition|
        (cons newpred (|get| x '|condition| env)) env))))
    (t env))))))

```

defun getUnionMode

[isUnionMode p317]
[getmode p??]

— defun getUnionMode —

```

(defun |getUnionMode| (x env)
  (let (m)
    (setq m (when (atom x) (|getmode| x env)))
    (when m (|isUnionMode| m env))))

```

defun isUnionMode

[getmode p??]
[get p??]

— defun isUnionMode —

```

(defun |isUnionMode| (m env)
  (let (mp v tmp1)
    (cond
      ((and (consp m) (eq (qfirst m) '|Union|)) m)
      ((progn
         (setq tmp1 (setq mp (|getmode| m env)))
         (and (consp tmp1) (eq (qfirst tmp1) '|Mapping|)
              (consp (qrest tmp1)) (eq (qcddr tmp1) nil)
              (consp (qsecond tmp1))
              (eq (qcaadr tmp1) '|UnionCategory|)))
         (second mp))
      ((setq v (|get| (if (eq m '$) '|Rep| m) '|value| env))
       (when (and (consp (car v)) (eq (qfirst (car v)) '|Union|)) (car v)))))

```

defplist compImport plist

We set up the `compImport` function to handle the `import` keyword by setting the `special` keyword on the `import` symbol property list.

— **postvars** —

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

—————

defun compImport

[[addDomain](#) p248]
 [[\\$NoValueMode](#) p172]

— **defun compImport** —

```
(defun |compImport| (form mode env)
  (declare (ignore mode))
  (declare (special |$NoValueMode|))
  (dolist (dom (cdr form)) (setq env (|addDomain| dom env)))
  (list '|/throwAway| |$NoValueMode| env))
```

—————

defplist compIs plist

We set up the `compIs` function to handle the `is` keyword by setting the `special` keyword on the `is` symbol property list.

— **postvars** —

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

—————

defun compIs

[[comp](#) p590]
 [[coerce](#) p351]
 [[\\$Boolean](#) p??]

[[\\$EmptyMode](#) [p172](#)]

— **defun compIs** —

```
(defun |compIs| (form mode env)
  (let (a b aval am tmp1 bval bm td)
    (declare (special |$Boolean| |$EmptyMode|))
    (setq a (second form))
    (setq b (third form))
    (when (setq tmp1 (|comp| a |$EmptyMode| env))
      (setq aval (first tmp1))
      (setq am (second tmp1))
      (setq env (third tmp1))
      (when (setq tmp1 (|comp| b |$EmptyMode| env))
        (setq bval (first tmp1))
        (setq bm (second tmp1))
        (setq env (third tmp1))
        (setq td (list (list '|domainEqual| aval bval) |$Boolean| env ))
        (|coerce| td mode))))))
```

—————

defplist compJoin plist

We set up the `compJoin` function to handle the `Join` keyword by setting the `special` keyword on the `Join` symbol property list.

— **postvars** —

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

—————

defun compJoin

```
[nreverse0 p??]
[compForMode p321]
[stackSemanticError p??]
[nreverse0 p??]
[isCategoryForm p??]
[union p??]
[compJoin,getParms p??]
[wrapDomainSub p290]
[convert p600]
```

[\$Category p??]

— defun compJoin —

```
(defun |compJoin| (form mode env)
  (labels (
    (getParms (y env)
      (cond
        ((atom y)
          (when (|isDomainForm| y env) (list y)))
        ((and (consp y) (eq (qfirst y) 'length)
          (consp (qrest y)) (eq (qcddr y) nil))
          (list y (second y)))
        (t (list y))))))
  (let (arg1 catList p1 tmp3 tmp4 tmp5 body parameters catListp td)
    (declare (special |$Category|))
    (setq arg1 (cdr form))
    (setq catList
      (dolist (x arg1 (nreverse0 tmp3))
        (push (car (or (|compForMode| x |$Category| env) (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 env)
                    (setq parameters
                      (|union|
                        (dolist (y (cdr x) tmp5)
                          (setq tmp5 (append tmp5 (getParms y env))))
                      parameters))
                  (x)
                ((and (consp x) (eq (qfirst x) '|DomainSubstitutionMacro|)
                  (consp (qrest x)) (consp (qcddr x))
                  (eq (qcdddr x) nil))
                  (setq p1 (second x))
                  (setq body (third x))
                  (setq parameters (|union| p1 parameters)) body)
                ((and (consp x) (eq (qfirst x) '|mkCategory|))
                  x)
                ((and (atom x) (equal (|getmode| x env) |$Category|))
                  x)
                (t
                  (|stackSemanticError| (list '|invalid argument to Join: | x) nil)
                  x))
```

```

      tmp4))))
    (setq td (list (|wrapDomainSub| parameters (cons '|Join| catListp))
                  |$Category| env))
    (|convert| td mode))))))

```

defun compForMode

[[comp p590](#)]

[[\\$compForModeIfTrue p??](#)]

— defun compForMode —

```

(defun |compForMode| (x m e)
  (let (|$compForModeIfTrue|)
    (declare (special |$compForModeIfTrue|))
    (setq |$compForModeIfTrue| t)
    (|comp| x m e)))

```

defplist compLambda plist

We set up the `compLambda` function to handle the `+->` keyword by setting the `special` keyword on the `+->` symbol property list.

— postvars —

```

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

```

defun compLambda

[[argsToSig p623](#)]

[[compAtSign p357](#)]

[[stackAndThrow p??](#)]

— defun compLambda —

```

(defun |compLambda| (form mode env)

```

```

(let (vl body tmp1 tmp2 tmp3 target args arg1 sig1 ress)
  (setq vl (second form))
  (setq body (third form))
  (cond
   ((and (consp vl) (eq (qfirst vl) '|:|)
    (progn
     (setq tmp1 (qrest vl))
     (and (consp tmp1)
      (progn
       (setq args (qfirst tmp1))
       (setq tmp2 (qrest tmp1))
       (and (consp tmp2)
        (eq (qrest tmp2) nil)
        (progn
         (setq target (qfirst tmp2))
         t))))))
    (when (and (consp args) (eq (qfirst args) '|@Tuple|))
      (setq args (qrest args)))
    (cond
     ((listp args)
      (setq tmp3 (|argsToSig| args))
      (setq arg1 (first tmp3))
      (setq sig1 (second tmp3))
      (cond
       (sig1
        (setq ress
         (compAtSign
          (list '@
           (list '+-> arg1 body)
           (cons '|Mapping| (cons target sig1))) mode env))
         ress)
        (t (|stackAndThrow| (list '|compLambda| form )))))
      (t (|stackAndThrow| (list '|compLambda| form )))))
    (t (|stackAndThrow| (list '|compLambda| form ))))))

```

defplist compLeave plist

We set up the `compLeave` function to handle the `leave` keyword by setting the `special` keyword on the `leave` symbol property list.

— postvars —

```

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

```

defun compLeave

```
[comp p590]
[modifyModeStack p625]
[$exitModeStack p??]
[$leaveLevelStack p??]
```

— **defun compLeave** —

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

—————

defplist compMacro plist

We set up the `compMacro` function to handle the `MDEF` keyword by setting the `special` keyword on the `MDEF` symbol property list.

— **postvars** —

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

—————

defun compMacro

```
[formatUnabbreviated p??]
[sayBrightly p??]
[put p??]
[macroExpand p174]
[$macroIfTrue p??]
[$NoValueMode p172]
[$EmptyMode p172]
```

— **defun compMacro** —

```

(defun |compMacro| (form mode env)
  (let (|$macroIfTrue| lhs signature specialCases rhs prhs)
    (declare (special |$macroIfTrue| |$NoValueMode| |$EmptyMode|))
    (setq |$macroIfTrue| t)
    (setq lhs (second form))
    (setq signature (third form))
    (setq specialCases (fourth form))
    (setq rhs (fifth form))
    (setq prhs
      (cond
        ((and (consp rhs) (eq (qfirst rhs) 'category))
         (list "-- the constructor category"))
        ((and (consp rhs) (eq (qfirst rhs) '|Join|))
         (list "-- the constructor category"))
        ((and (consp rhs) (eq (qfirst rhs) 'capsule))
         (list "-- the constructor capsule"))
        ((and (consp rhs) (eq (qfirst rhs) '|add|))
         (list "-- the constructor capsule"))
        (t (|formatUnabbreviated| rhs))))
    (|sayBrightly|
     (cons "    processing macro definition"
       (cons '|%b|
         (append (|formatUnabbreviated| lhs)
           (cons " ==> "
             (append prhs (list '|%d|))))))))
    (when (or (equal mode |$EmptyMode|) (equal mode |$NoValueMode|))
      (list '|/throwAway| |$NoValueMode|
        (|put| (CAR lhs) '|macro| (|macroExpand| rhs env) env))))))

```

defplist compPretend plist

We set up the `compPretend` function to handle the `pretend` keyword by setting the `special` keyword on the `pretend` symbol property list.

— postvars —

```

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

```

defun compPretend

[addDomain p248]
 [comp p590]


```
[opOf p??]
[stackSemanticError p??]
[stackWarning p??]
[$newCompilerUnionFlag p??]
[$EmptyMode p172]
```

— **defun compPretend** —

```
(defun |compPretend| (form mode env)
  (let (x tt warningMessage td tp)
    (declare (special |$newCompilerUnionFlag| |$EmptyMode|))
    (setq x (second form))
    (setq tt (third form))
    (setq env (|addDomain| tt env))
    (when (setq td (or (|comp| x tt env) (|comp| x |$EmptyMode| env)))
      (when (equal (second td) tt)
        (setq warningMessage (list '|pretend| tt '| -- should replace by @|)))
      (cond
        ((and |$newCompilerUnionFlag|
              (eq (|opOf| (second td)) '|Union|)
              (not (eq (|opOf| mode) '|Union|)))
          (|stackSemanticError|
           (list '|cannot pretend | x '| of mode | (second td) '| to mode | mode)
           nil))
        (t
         (setq td (list (first td) tt (third td)))
         (when (setq tp (|coerce| td mode))
           (when warningMessage (|stackWarning| warningMessage)
             tp))))))
```

—————

deflist compQuote plist

We set up the `compQuote` function to handle the `QUOTE` keyword by setting the `special` keyword on the `QUOTE` symbol property list.

— **postvars** —

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

—————

defun compQuote

— defun compQuote —

```
(defun |compQuote| (form mode env)
  (list form mode env))
```

—————

defplist compReduce plist

We set up the `compReduce` function to handle the `REDUCE` keyword by setting the `special` keyword on the `REDUCE` symbol property list.

— postvars —

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

—————

defun compReduce

```
[compReduce1 p326]
[$formalArgList p??]
```

— defun compReduce —

```
(defun |compReduce| (form mode env)
  (declare (special |$formalArgList|))
  (|compReduce1| form mode env |$formalArgList|))
```

—————

defun compReduce1

```
[systemError p??]
[nreverse0 p??]
[compIterator p??]
[comp p590]
[parseTran p97]
[getIdentity p??]
```

```

[$sideEffectsList p??]
[$until p??]
[$initList p??]
[$Boolean p??]
[$e p??]
[$sendTestList p??]

```

— defun compReduce1 —

```

(defun |compReduce1| (form mode env |$formalArgList|)
  (declare (special |$formalArgList|))
  (let (|$sideEffectsList| |$until| |$initList| |$sendTestList| collectForm
        collectOp body op itl acc afterFirst bodyVal part1 part2 part3 id
        identityCode untilCode finalCode tmp1 tmp2)
    (declare (special |$sideEffectsList| |$until| |$initList| |$Boolean| |$e|
                      |$sendTestList|))
    (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 |$initList| nil)
       (setq |$sendTestList| nil)
       (setq |$e| env)
       (setq itl
        (dolist (x itl (nreverse0 tmp2))
          (setq tmp1 (or (|compIterator| x |$e|) (return '|failed|)))
          (setq |$e| (second tmp1))
          (push (elt tmp1 0) tmp2)))
        (unless (eq itl '|failed|)
          (setq env |$e|)
          (setq acc (gensym))
          (setq afterFirst (gensym))
          (setq bodyVal (gensym))
          (when (setq tmp1 (|comp| (list 'let bodyVal body ) mode env))
            (setq part1 (first tmp1))
            (setq mode (second tmp1))
            (setq env (third tmp1))
            (when (setq tmp1 (|comp| (list 'let acc bodyVal) mode env))
              (setq part2 (first tmp1))
              (setq env (third tmp1))

```

```

(when (setq tmp1
        (|comp| (list 'let acc (|parseTran| (list op acc bodyVal)))
                mode env))
      (setq part3 (first tmp1))
      (setq env (third tmp1))
      (when (setq identityCode
                (if (setq id (|getIdentity| op env))
                    (car (|comp| id mode env))
                    (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| |$Boolean| env))
                          (setq untilCode (first tmp1))
                          (setq env (third tmp1))
                          (setq finalCode
                                (subst (list 'until untilCode) '|$until| finalCode :test #'equal)))
                          (list finalCode mode env ))))))))

```

defplist compRepeatOrCollect plist

We set up the `compRepeatOrCollect` function to handle the `COLLECT` keyword by setting the `special` keyword on the `COLLECT` symbol property list.

— postvars —

```

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

```

defplist compRepeatOrCollect plist

We set up the `compRepeatOrCollect` function to handle the `REPEAT` keyword by setting the `special` keyword on the `REPEAT` symbol property list.

— postvars —

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

defun compRepeatOrCollect

```
[length p??]
[compIterator p??]
[modeIsAggregateOf p??]
[stackMessage p??]
[compOrCroak p588]
[comp p590]
[coerceExit p357]
[ p??]
[ p??]
[$until p??]
[$Boolean p??]
[$NoValueMode p172]
[$exitModeStack p??]
[$leaveLevelStack p??]
[$formalArgList p??]
```

— defun compRepeatOrCollect —

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

```

(if (eq repeatOrCollect 'collect)
  (cond
    ((eq targetMode '|$EmptyMode|)
     '|$EmptyMode|)
    ((setq u (|modeIsAggregateOf| '|List| targetMode env))
     (second u))
    ((setq u (|modeIsAggregateOf| '|PrimitiveArray| targetMode env))
     (setq repeatOrCollect 'collectv)
     (second u))
    ((setq u (|modeIsAggregateOf| '|Vector| targetMode env))
     (setq repeatOrCollect 'collectvec)
     (second u))
    (t
     (|stackMessage| "Invalid collect bodytype")
     '|failed|))
    '$NoValueMode|))
(unless (eq bodyMode '|failed|)
  (when (setq tmp1 (|compOrCroak| body bodyMode env))
    (setq bodyp (first tmp1))
    (setq mp (second tmp1))
    (setq ep (third tmp1))
    (when |$until|
      (setq tmp1 (|comp| |$until| |$Boolean| ep))
      (setq untilCode (first tmp1))
      (setq ep (third tmp1))
      (setq itlp
        (subst (list 'until untilCode) '|$until| itlp :test #'equal)))
    (setq formp (cons repeatOrCollect (append itlp (list bodyp))))
    (setq mpp
      (cond
        ((eq repeatOrCollect 'collect)
         (if (setq u (|modeIsAggregateOf| '|List| targetMode env))
             (car u)
             (list '|List| mp)))
        ((eq repeatOrCollect 'collectv)
         (if (setq u (|modeIsAggregateOf| '|PrimitiveArray| targetMode env))
             (car u)
             (list '|PrimitiveArray| mp)))
        ((eq repeatOrCollect 'collectvec)
         (if (setq u (|modeIsAggregateOf| '|Vector| targetMode env))
             (car u)
             (list '|Vector| mp)))
        (t mp)))
    (|coerceExit| (list formp mpp ep) targetMode))))))
(declare (special |$exitModeStack| |$leaveLevelStack| |$formalArgList|))
(fn form
  (cons mode |$exitModeStack|)
  (cons (|#| |$exitModeStack|) |$leaveLevelStack|)
  |$formalArgList|
  env)))

```

defplist compReturn plist

We set up the `compReturn` function to handle the `return` keyword by setting the `special` keyword on the `return` symbol property list.

— **postvars** —

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

defun compReturn

```
[stackSemanticError p??]
[userError p??]
[resolve p361]
[comp p590]
[modifyModeStack p625]
[$exitModeStack p??]
[$returnMode p??]
```

— **defun compReturn** —

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

```

(setq mp (second u))
(setq ep (third u))
(when (>= index 0)
  (setq |$returnMode| (|resolve| mp |$returnMode|))
  (|modifyModeStack| mp index))
(list (list 'TAGGEDreturn| 0 u) mode ep))))))

```

defplist compSeq plist

We set up the `compSeq` function to handle the `SEQ` keyword by setting the `special` keyword on the `SEQ` symbol property list.

— **postvars** —

```

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

```

defun compSeq

```

[compSeq1 p332]
[$exitModeStack p??]

```

— **defun compSeq** —

```

(defun |compSeq| (form mode env)
  (declare (special |$exitModeStack|))
  (|compSeq1| (cdr form) (cons mode |$exitModeStack|) env))

```

defun compSeq1

```

[nreverse0 p??]
[compSeqItem p334]
[mkq p??]
[replaceExitEtc p333]
[$exitModeStack p??]
[$insideExpressionIfTrue p??]
[$finalEnv p??]

```


[[\\$NoValueMode](#) p172]

— **defun compSeq1** —

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

—————

defun replaceExitEtc

[[rplac](#) p??]

[[replaceExitEtc](#) p333]

[[intersectionEnvironment](#) p??]

[[convertOrCroak](#) p334]

[[\\$finalEnv](#) p??]

[[\\$finalEnv](#) p??]

— **defun replaceExitEtc** —

```
(defun |replaceExitEtc| (x tag opFlag opMode)
  (declare (special |$finalEnv|))
  (cond
    ((atom x) nil)
    ((and (consp x) (eq (qfirst x) 'quote)) nil)
    ((and (consp x) (equal (qfirst x) opFlag) (consp (qrest x))
      (consp (qcddr x)) (eq (qcdddr x) nil))
      (|rplac| (caaddr x) (|replaceExitEtc| (caaddr x) tag opFlag opMode))
      (cond
        ((eq1 (second x) 0)
```

```

(setq |$finalEnv|
  (if |$finalEnv|
    (|intersectionEnvironment| |$finalEnv| (third (third x)))
    (third (third x))))
(|rplac| (car x) 'throw)
(|rplac| (cadr x) tag)
(|rplac| (caddr x) (car (|convertOrCroak| (caddr x) opMode))))
(t
  (|rplac| (cadr x) (1- (cadr x)))))
((and (consp x) (consp (qrest x)) (consp (qcddr x))
  (eq (qcddr x) nil)
  (member (qfirst x) '(|TAGGEDreturn| |TAGGEDexit|)))
  (|rplac| (car (caddr x))
    (|replaceExitEtc| (car (caddr x)) tag opFlag opMode)))
(t
  (|replaceExitEtc| (car x) tag opFlag opMode)
  (|replaceExitEtc| (cdr x) tag opFlag opMode)))
x)

```

defun convertOrCroak

[convert p600]
[userError p??]

— defun convertOrCroak —

```

(defun |convertOrCroak| (tt m)
  (let (u)
    (if (setq u (|convert| tt m))
      u
      (|userError|
        (list 'CANNOT CONVERT: | (first tt) '|%1| ' OF MODE: | (second tt)
          '|%1| ' TO MODE: | m '|%1|))))))

```

defun compSeqItem

[comp p590]
[macroExpand p174]

— defun compSeqItem —

```
(defun |compSeqItem| (form mode env)
  (|comp| (|macroExpand| form env) mode env))
```

defplist compSetq plist

We set up the `compSetq` function to handle the `LET` keyword by setting the `special` keyword on the `LET` symbol property list.

— **postvars** —

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

defplist compSetq plist

We set up the `compSetq` function to handle the `SETQ` keyword by setting the `special` keyword on the `SETQ` symbol property list.

— **postvars** —

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

defun compSetq

[`compSetq1` p335]

— **defun compSetq** —

```
(defun |compSetq| (form mode env)
  (|compSetq1| (second form) (third form) mode env))
```

defun compSetq1

[`setqSingle` p340]

[`compSetq1` `identp` (`vol5`)]

[compMakeDeclaration p624]
 [compSetq p335]
 [setqMultiple p337]
 [setqSetelt p340]
 [\$EmptyMode p172]

— defun compSetq1 —

```
(defun |compSetq1| (form val mode env)
  (let (x y ep op z)
    (declare (special |$EmptyMode|))
    (cond
      ((identp form) (|setqSingle| form val mode env))
      ((and (consp form) (eq (qfirst form) '|:|) (consp (qrest form))
        (consp (qcddr form)) (eq (qcdddr form) nil))
        (setq x (second form))
        (setq y (third form))
        (setq ep (third (|compMakeDeclaration| form |$EmptyMode| env)))
        (|compSetq| (list 'let x val) mode ep))
      ((consp form)
        (setq op (qfirst form))
        (setq z (qrest form))
        (cond
          ((eq op 'cons) (|setqMultiple| (|uncons| form) val mode env))
          ((eq op '|@Tuple|) (|setqMultiple| z val mode env))
          (t (|setqSetelt| form val mode env))))))
```

defun uncons

[uncons p336]

— defun uncons —

```
(defun |uncons| (x)
  (cond
    ((atom x) x)
    ((and (consp x) (eq (qfirst x) 'cons) (consp (qrest x))
      (consp (qcddr x)) (eq (qcdddr x) nil))
      (cons (second x) (|uncons| (third x)))))
```

defun setqMultiple

```

[nreverse0 p??]
[stackMessage p??]
[setqMultipleExplicit p339]
[genVariable p??]
[addBinding p??]
[compSetq1 p335]
[convert p600]
[put p??]
[genSomeVariable p??]
[length p??]
[mkprogn p??]
[$EmptyMode p172]
[$NoValueMode p172]
[$noEnv p??]

```

— defun setqMultiple —

```

(defun |setqMultiple| (nameList val m env)
  (labels (
    (decompose (tt len env)
      (declare (ignore len))
      (let (tmp1 z)
        (declare (special |$EmptyMode|))
        (cond
          ((and (consp tt) (eq (qfirst tt) '|Record|)
            (progn (setq z (qrest tt)) t))
          (loop for item in z
            collect (cons (second item) (third item))))
        ((progn
          (setq tmp1 (|comp| tt |$EmptyMode| env))
          (and (consp tmp1) (consp (qrest tmp1)) (consp (qsecond tmp1))
            (eq (qcaadr tmp1) '|RecordCategory|)
            (consp (qcddr tmp1)) (eq (qcdddr tmp1) nil)))
          (loop for item in z
            collect (cons (second item) (third item))))
        (t (|stackMessage| (list '|no multiple assigns to mode: | tt))))))
    (let (g m1 tt x mp selectorModePairs tmp2 assignList)
      (declare (special |$noEnv| |$EmptyMode| |$NoValueMode|))
      (cond
        ((and (consp val) (eq (qfirst val) '|cons|) (equal m |$NoValueMode|))
          (|setqMultipleExplicit| nameList (|uncons| val) m env))
        ((and (consp val) (eq (qfirst val) '|@Tuple|) (equal m |$NoValueMode|))
          (|setqMultipleExplicit| nameList (qrest val) m env))
        ; 1 create a gensym, %add to local environment, compile and assign rhs
        (t
          (setq g (|genVariable|))

```

```

(setq env (|addBinding| g nil env))
(setq tmp2 (|compSetq1| g val |$EmptyMode| env))
(when tmp2
  (setq tt tmp2)
  (setq m1 (cadr tmp2))
  (setq env (|put| g 'mode m1 env))
  (setq tmp2 (|convert| tt m))
; 1.1 --exit if result is a list
  (when tmp2
    (setq x (first tmp2))
    (setq mp (second tmp2))
    (setq env (third tmp2))
    (cond
      ((and (consp m1) (eq (qfirst m1) '|List|) (consp (qrest m1))
        (eq (qcddr m1) nil))
        (loop for y in nameList do
          (setq env
            (|put| y '|value| (list (|genSomeVariable|) (second m1) |$noEnv|
              env))))
        (|convert| (list (list 'progn x (list 'let nameList g) g) mp env) m))
      (t
        ; 2 --verify that the #nameList = number of parts of right-hand-side
        (setq selectorModePairs
          (decompose m1 (|#| nameList) env))
        (when selectorModePairs
          (cond
            ((not (eq1 (|#| nameList) (|#| selectorModePairs)))
              (|stackMessage|
                (list val '| must decompose into |
                  (|#| nameList) '| components| )))
            (t
              ; 3 --generate code
              (setq assignList
                (loop for x in nameList
                  for item in selectorModePairs
                    collect (car
                      (progn
                        (setq tmp2
                          (or (|compSetq1| x (list '|elt| g (first item))
                            (rest item) env)
                            (return '|failed|)))
                        (setq env (third tmp2))
                        tmp2))))
              (unless (eq assignList '|failed|)
                (list (mkprogn (cons x (append assignList (list g)))) mp env)
                ))))))))

```

defun setqMultipleExplicit

```
[stackMessage p??]
[genVariable p??]
[compSetq1 p335]
[last p??]
[$EmptyMode p172]
[$NoValueMode p172]
```

— defun setqMultipleExplicit —

```
(defun |setqMultipleExplicit| (nameList valList m env)
  (declare (ignore m))
  (let (gensymList assignList tmp1 reAssignList)
    (declare (special |$NoValueMode| |$EmptyMode|))
    (cond
      ((not (eq1 (|#| nameList) (|#| valList))))
      (|stackMessage|
        (list '|Multiple assignment error; # of items in: | nameList
              '|must = # in: | valList))))
    (t
      (setq gensymList
        (loop for name in nameList
              collect (|genVariable|)))
      (setq assignList
        (loop for g in gensymList
              for val in valList
              collect (progn
                (setq tmp1
                  (or (|compSetq1| g val |$EmptyMode| env)
                    (return '|failed|)))
                (setq env (third tmp1))
                tmp1)))
      (unless (eq assignList '|failed|)
        (setq reAssignList
          (loop for g in gensymList
                for name in nameList
                collect (progn
                  (setq tmp1
                    (or (|compSetq1| name g |$EmptyMode| env)
                      (return '|failed|)))
                  (setq env (third tmp1))
                  tmp1)))
        (unless (eq reAssignList '|failed|)
          (list
            (cons 'progn
              (append
                (loop for tt in assignList
                      collect (car tt))
```

```
(loop for tt in reAssignList
  collect (car tt)))
|$NoValueMode| (third (|last| reAssignList)))))))))
```

defun setqSetelt

[comp p590]

— defun setqSetelt —

```
(defun |setqSetelt| (form val mode env)
  (|comp| (cons '|setelt| (cons (car form) (append (cdr form) (list val))))
    mode env))
```

defun setqSingle

```
[setqSingle getProplist (vol5)]
[getmode p??]
[get p??]
[maxSuperType p344]
[comp p590]
[getmode p??]
[assignError p342]
[convert p600]
[setqSingle identp (vol5)]
[profileRecord p??]
[consProplistOf p??]
[removeEnv p??]
[setqSingle addBinding (vol5)]
[isDomainForm p344]
[isDomainInScope p??]
[stackWarning p??]
[augModemapsFromDomain1 p252]
[NRTassocIndex p342]
[isDomainForm p344]
[outputComp p343]
[$insideSetqSingleIfTrue p??]
[$QuickLet p??]
[$form p??]
```


[\$profileCompiler p??]
 [\$EmptyMode p172]
 [\$NoValueMode p172]

— defun setqSingle —

```
(defun |setqSingle| (form val mode env)
  (let (|$insideSetqSingleIfTrue| currentProplist mpp maxmpp td x mp tp key
        newProplist ep k newform)
    (declare (special |$insideSetqSingleIfTrue| |$QuickLet| |$form|
                      |$profileCompiler| |$EmptyMode| |$NoValueMode|))
    (setq |$insideSetqSingleIfTrue| t)
    (setq currentProplist (|getProplist| form env))
    (setq mpp
      (or (|get| form '|mode| env) (|getmode| form env)
          (if (equal mode |$NoValueMode|) |$EmptyMode| mode)))
    (when (setq td
      (cond
        ((setq td (|comp| val mpp env))
         td)
        ((and (null (|get| form '|mode| env))
              (not (equal mpp (setq maxmpp (|maxSuperType| mpp env))))
              (setq td (|comp| val maxmpp env)))
         td)
        ((and (setq td (|comp| val |$EmptyMode| env))
              (|getmode| (second td) env))
         (|assignError| val (second td) form mpp))))
      (when (setq tp (|convert| td mode))
        (setq x (first tp))
        (setq mp (second tp))
        (setq ep (third tp))
        (when (and |$profileCompiler| (identp form))
          (setq key (if (member form (cdr |$form|)) '|arguments| '|locals|))
          (|profileRecord| key form (second td)))
        (setq newProplist
          (|consProplistOf| form currentProplist '|value|
                            (|removeEnv| (cons val (cdr td)))))
        (setq ep (if (consp form) ep (|addBinding| form newProplist ep)))
        (when (|isDomainForm| val ep)
          (when (|isDomainInScope| form ep)
            (|stackWarning|
              (list '|domain valued variable| '|%b| form '|%d|
                    '|has been reassigned within its scope| )))
            (setq ep (|augModemapsFromDomain1| form val ep)))
        (if (setq k (|NRTassocIndex| form))
          (setq newform (list 'setelt '$ k x))
          (setq newform
            (if |$QuickLet|
              (list 'let form x))
```

```

(list 'let form x
  (if (|isDomainForm| x ep)
    (list 'elt form 0)
    (car (|outputComp| form ep))))))
(list newform mp ep))))

```

defun NRTassocIndex

This function returns the index of domain entry *x* in the association list [*\$NRTaddForm* *p??*] [*\$NRTdeltaList* *p??*] [*\$found* *p??*] [*\$NRTbase* *p??*] [*\$NRTdeltaLength* *p??*]

— defun NRTassocIndex —

```

(defun |NRTassocIndex| (x)
  (let (k (i 0))
    (declare (special |$NRTdeltaLength| |$NRTbase| |$found| |$NRTdeltaList|
                      |$NRTaddForm|))
    (cond
      ((null x) x)
      ((equal x |$NRTaddForm|) 5)
      ((setq k
        (let (result)
          (loop for y in |$NRTdeltaList|
            when (and (incf i)
                      (eq (elt y 0) '|domain|)
                      (equal (elt y 1) x)
                      (setq |$found| y))
            do (setq result (or result i)))
          result))
        (- (+ |$NRTbase| |$NRTdeltaLength|) k))
      (t nil))))

```

defun assignError

[stackMessage *p??*]

— defun assignError —

```
(defun |assignError| (val mp form m)
  (let (message)
    (setq message
      (if val
        (list '|CANNOT ASSIGN: | val '|%1|
              '| OF MODE: | mp '|%1|
              '| TO: | form '|%1| '| OF MODE: | m)
        (list '|CANNOT ASSIGN: | val '|%1|
              '| TO: | form '|%1| '| OF MODE: | m)))
      (|stackMessage| message)))
```

defun outputComp

```
[comp p590]
[nreverse0 p??]
[outputComp p343]
[get p??]
[$Expression p??]
```

— defun outputComp —

```
(defun |outputComp| (x env)
  (let (arg1 v)
    (declare (special |$Expression|))
    (cond
      ((|comp| (list '|::| x |$Expression|) |$Expression| env))
      ((and (consp x) (eq (qfirst x) '|construct|))
        (setq arg1 (qrest x))
        (list (cons 'list
          (let (result tmp1)
            (loop for x in arg1
              do (setq result
                (cons (car
                  (progn
                    (setq tmp1 (|outputComp| x env))
                    (setq env (third tmp1))
                    tmp1))
                  result))))
            (nreverse0 result)))
          |$Expression| env))
      ((and (setq v (|get| x '|value| env))
        (consp (cadr v)) (eq (qfirst (cadr v)) '|Union|))
        (list (list '|coerceUn2E| x (cadr v)) |$Expression| env))
      (t (list x |$Expression| env))))))
```

defun maxSuperType

```
[get p??]
[maxSuperType p344]
```

— defun maxSuperType —

```
(defun |maxSuperType| (m env)
  (let (typ)
    (if (setq typ (|get| m '|SuperDomain| env))
        (|maxSuperType| typ env)
        m)))
```

defun isDomainForm

```
[kar p??]
[isFunctor p249]
[isCategoryForm p??]
[isDomainConstructorForm p344]
[$SpecialDomainNames p??]
```

— defun isDomainForm —

```
(defun |isDomainForm| (d env)
  (let (tmp1)
    (declare (special |$SpecialDomainNames|))
    (or (member (kar d) |$SpecialDomainNames|) (|isFunctor| d)
        (and (progn
              (setq tmp1 (|getmode| d env))
              (and (consp tmp1) (eq (qfirst tmp1) '|Mapping|) (consp (qrest tmp1))))
          (|isCategoryForm| (qsecond tmp1) env))
        (|isCategoryForm| (|getmode| d env) env)
        (|isDomainConstructorForm| d env))))
```

defun isDomainConstructorForm

```
[isCategoryForm p??]
[eqsubstlist p??]
```

[[\\$FormalMapVariableList](#) p266]

— **defun isDomainConstructorForm** —

```
(defun |isDomainConstructorForm| (d env)
  (let (u)
    (declare (special |$FormalMapVariableList|))
    (when
      (and (consp d)
            (setq u (|get| (qfirst d) '|value| env))
            (consp u)
            (consp (qrest u))
            (consp (qsecond u))
            (eq (qcaadr u) '|Mapping|)
            (consp (qcdadr u)))
      (|isCategoryForm|
       (eqsubstlist (rest d) |$FormalMapVariableList| (cadadr u)) env))))
```

—————

defplist compString plist

We set up the `compString` function to handle the `String` keyword by setting the `special` keyword on the `String` symbol property list.

— **postvars** —

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

—————

defun compString

[[resolve](#) p361]

[[\\$StringCategory](#) p??]

— **defun compString** —

```
(defun |compString| (form mode env)
  (declare (special |$StringCategory|))
  (list form (|resolve| |$StringCategory| mode) env))
```

—————

defplist compSubDomain plist

We set up the `compSubDomain` function to handle the `SubDomain` keyword by setting the `special` keyword on the `SubDomain` symbol property list.

— **postvars** —

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

—————

defun compSubDomain

```
[compSubDomain1 p346]
[compCapsule p274]
[$addFormLhs p??]
[$NRTaddForm p??]
[$addForm p??]
[$addFormLhs p??]
```

— **defun compSubDomain** —

```
(defun |compSubDomain| (form mode env)
  (let (|$addFormLhs| |$addForm| domainForm predicate tmp1)
    (declare (special |$addFormLhs| |$addForm| |$NRTaddForm| |$addFormLhs|))
    (setq domainForm (second form))
    (setq predicate (third form))
    (setq |$addFormLhs| domainForm)
    (setq |$addForm| nil)
    (setq |$NRTaddForm| domainForm)
    (setq tmp1 (|compSubDomain1| domainForm predicate mode env))
    (setq |$addForm| (first tmp1))
    (setq env (third tmp1))
    (|compCapsule| (list 'capsule) mode env)))
```

—————

defun compSubDomain1

```
[compMakeDeclaration p624]
[addDomain p248]
[compOrCroak p588]
[stackSemanticError p??]
[lispize p347]
```

[evalAndRwriteLispForm p200]
 [\$CategoryFrame p??]
 [\$op p??]
 [\$lisplibSuperDomain p??]
 [\$Boolean p??]
 [\$EmptyMode p172]

— defun compSubDomain1 —

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

—————

defun lispize

[optimize p225]

— defun lispize —

```
(defun |lispize| (x)
  (car (|optimize| (list x))))
```

—————

defplist compSubsetCategory plist

We set up the `compSubsetCategory` function to handle the `SubsetCategory` keyword by setting the `special` keyword on the `SubsetCategory` symbol property list.

— postvars —

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

defun compSubsetCategory

TPDHERE: See `LocalAlgebra` for an example call `[put p??]`

`[comp p590]`

`[$lhsOfColon p??]`

— defun compSubsetCategory —

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

defplist compSuchthat plist

We set up the `compSuchthat` function to handle the `|` keyword by setting the `special` keyword on the `|` symbol property list.

— postvars —


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

defun compSuchthat

```
[comp p590]
[put p??]
[$Boolean p??]
```

— defun compSuchthat —

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

defplist compVector plist

We set up the `compVector` function to handle the `VECTOR` keyword by setting the `special` keyword on the `VECTOR` symbol property list.

— postvars —

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

defun compVector

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

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

```
[comp p590]
[$EmptyVector p??]
```

— **defun compVector** —

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

—————

defplist compWhere plist

We set up the `compWhere` function to handle the `where` keyword by setting the `special` keyword on the `where` symbol property list.

— **postvars** —

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

—————

defun compWhere

```
[comp p590]
[macroExpand p174]
```

```
[deltaContour p??]
[addContour p??]
[$insideExpressionIfTrue p??]
[$insideWhereIfTrue p??]
[$EmptyMode p172]
```

— defun compWhere —

```
(defun |compWhere| (form mode eInit)
  (let (|$insideExpressionIfTrue| |$insideWhereIfTrue| newform exprList e
        eBefore tmp1 x eAfter del eFinal)
    (declare (special |$insideExpressionIfTrue| |$insideWhereIfTrue|
                      |$EmptyMode|))
    (setq newform (second form))
    (setq exprlist (cddr form))
    (setq |$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| newform (setq eBefore e)) mode e))
    (when tmp1
      (setq x (first tmp1))
      (setq mode (second tmp1))
      (setq eAfter (third tmp1))
      (setq del (|deltaContour| eAfter eBefore))
      (if del
        (setq eFinal (|addContour| del eInit))
        (setq eFinal eInit))
      (list x mode eFinal))))
```

6.6 Functions for coercion

defun coerce

The function `coerce` is used by the old compiler for coercions. The function `coerceInteractive` is used by the interpreter. One should always call the correct function, since the representation of basic objects may not be the same. [keyedSystemError p??]

```
[rplac p??]
[coerceEasy p352]
[coerceSubset p353]
```

```
[coerceHard p354]
[isSomeDomainVariable p??]
[stackMessage p??]
[$InteractiveMode p??]
[$Rep p??]
[$fromCoerceable p??]
```

— defun coerce —

```
(defun |coerce| (tt mode)
  (labels (
    (fn (x m1 m2)
      (list '|Cannot coerce| '|%b| x '|%d| '|%l| '|          of mode| '|%b| m1
            '|%d| '|%l| '|          to mode| '|%b| m2 '|%d|)))
    (let (tp)
      (declare (special |$fromCoerceable$| |$Rep| |$InteractiveMode|))
      (if |$InteractiveMode|
        (|keyedSystemError| 'S2GE0016
          (list "coerce" "function coerce called from the interpreter."))
        (progn
          (|rplac| (cadr tt) (subst '$ |$Rep| (cadr tt) :test #'equal))
          (cond
            ((setq tp (|coerceEasy| tt mode)) tp)
            ((setq tp (|coerceSubset| tt mode)) tp)
            ((setq tp (|coerceHard| tt mode)) tp)
            ((or (eq (car tt) '|$fromCoerceable$|) (|isSomeDomainVariable| mode)) nil)
            (t (|stackMessage| (fn (first tt) (second tt) mode))))))))))
```

—

defun coerceEasy

```
[modeEqualSubst p363]
[$EmptyMode p172]
[$Exit p??]
[$NoValueMode p172]
[$Void p??]
```

— defun coerceEasy —

```
(defun |coerceEasy| (tt m)
  (declare (special |$EmptyMode| |$Exit| |$NoValueMode| |$Void|))
  (cond
    ((equal m |$EmptyMode|) tt)
    ((or (equal m |$NoValueMode|) (equal m |$Void|))
     (list (car tt) m (third tt)))
```

```

(equal (second tt) m) tt)
(equal (second tt) |$NoValueMode|) tt)
(equal (second tt) |$Exit|)
(list
  (list 'progn (car tt) (list '|userError| "Did not really exit."))
  m (third tt)))
(or (equal (second tt) |$EmptyMode|)
    (|modeEqualSubst| (second tt) m (third tt)))
(list (car tt) m (third tt))))

```

defun coerceSubset

```

[isSubset p??]
[lassoc p??]
[get p??]
[opOf p??]
[eval p??]
[isSubset p??]
[maxSuperType p344]

```

— defun coerceSubset —

```

(defun |coerceSubset| (arg1 mp)
  (let (x m env pred)
    (setq x (first arg1))
    (setq m (second arg1))
    (setq env (third arg1))
    (cond
      ((or (|isSubset| m mp env) (and (eq m '|Rep|) (eq mp '$)))
        (list x mp env))
      ((and (consp m) (eq (qfirst m) '|SubDomain|)
            (consp (qrest m)) (equal (qsecond m) mp))
        (list x mp env))
      ((and (setq pred (lassoc (|opOf| mp) (|get| (|opOf| m) '|SubDomain| env)))
            (integerp x) (|eval| (subst x '|#1| pred :test #'equal)))
        (list x mp env))
      ((and (setq pred (|isSubset| mp (|maxSuperType| m env) env))
            (integerp x) (|eval| (subst x '|* pred :test #'equal)))
        (list x mp env))
      (t nil))))

```

defun coerceHard

```
[modeEqual p362]
[get p??]
[getmode p??]
[isCategoryForm p??]
[extendsCategoryForm p??]
[coerceExtraHard p355]
[$e p??]
[$e p??]
[$String p345]
[$bootStrapMode p??]
```

— defun coerceHard —

```
(defun |coerceHard| (tt m)
  (let (|$e| mp tmp1 mpp)
    (declare (special |$e| |$String| |$bootStrapMode|))
    (setq |$e| (third tt))
    (setq mp (second tt))
    (cond
      ((and (stringp mp) (|modeEqual| m |$String|))
        (list (car tt) m |$e|))
      ((or (|modeEqual| mp m)
        (and (or (progn
          (setq tmp1 (|get| mp '|value| |$e|))
          (and (consp tmp1)
            (progn (setq mpp (qfirst tmp1)) t)))
          (progn
            (setq tmp1 (|getmode| mp |$e|))
            (and (consp tmp1)
              (eq (qfirst tmp1) '|Mapping|)
              (and (consp (qrest tmp1))
                (eq (qcddr tmp1) nil)
                (progn (setq mpp (qsecond tmp1)) t))))))
          (|modeEqual| mpp m))
        (and (or (progn
          (setq tmp1 (|get| m '|value| |$e|))
          (and (consp tmp1)
            (progn (setq mpp (qfirst tmp1)) t)))
          (progn
            (setq tmp1 (|getmode| m |$e|))
            (and (consp tmp1)
              (eq (qfirst tmp1) '|Mapping|)
              (and (consp (qrest tmp1))
                (eq (qcddr tmp1) nil)
                (progn (setq mpp (qsecond tmp1)) t))))))
          (|modeEqual| mpp mp)))
        (list (car tt) m (third tt)))
```

```

((and (stringp (car tt)) (equal (car tt) m))
 (list (car tt) m |$e|))
((|isCategoryForm| m |$e|)
 (cond
  ((eq |$bootStrapMode| t)
   (list (car tt) m |$e|))
  ((|extendsCategoryForm| (car tt) (cadr tt) m)
   (list (car tt) m |$e|))
  (t (|coerceExtraHard| tt m))))
(t (|coerceExtraHard| tt m))))

```

defun coerceExtraHard

[\[autoCoerceByModemap p360\]](#)
[\[isUnionMode p317\]](#)
[\[hasType p356\]](#)
[\[member p??\]](#)
[\[autoCoerceByModemap p360\]](#)
[\[coerce p351\]](#)
[\[\\$Expression p??\]](#)

— defun coerceExtraHard —

```

(defun |coerceExtraHard| (tt m)
  (let (x mp e tmp1 z ta tp tpp)
    (declare (special |$Expression|))
    (setq x (first tt))
    (setq mp (second tt))
    (setq e (third tt))
    (cond
     ((setq tp (|autoCoerceByModemap| tt m)) tp)
     ((and (progn
            (setq tmp1 (|isUnionMode| mp e))
            (and (consp tmp1) (eq (qfirst tmp1) '|Union|))
            (progn
             (setq z (qrest tmp1)) t)))
      (setq ta (|hasType| x e))
      (|member| ta z)
      (setq tp (|autoCoerceByModemap| tt ta))
      (setq tpp (|coerce| tp m)))
     tpp)
    ((and (consp mp) (eq (qfirst mp) '|Record|) (equal m |$Expression|))
     (list (list '|coerceRe2E| x (list 'elt (copy mp) 0)) m e))
    (t nil)))

```

defun hasType

[get p??]

— defun hasType —

```
(defun |hasType| (x e)
  (labels (
    (fn (x)
      (cond
        ((null x) nil)
        ((and (consp x) (consp (qfirst x)) (eq (qcaar x) '|case|)
              (consp (qcddar x)) (consp (qcddar x))
              (eq (qcdddar x) nil))
          (qcaddar x))
        (t (fn (cdr x))))))
    (fn (|get| x '|condition| e))))
```

defun coerceable

[pmatch p??]
 [sublis p??]
 [coerce p³⁵¹]
 [\$fromCoerceable p??]

— defun coerceable —

```
(defun |coerceable| (m mp env)
  (let (sl)
    (declare (special |$fromCoerceable$|))
    (cond
      ((equal m mp) m)
      ((setq sl (|pmatch| mp m)) (sublis sl mp))
      ((|coerce| (list '|$fromCoerceable$| m env) mp) mp)
      (t nil))))
```


defun coerceExit

[[resolve p361](#)]
 [replaceExitEsc p??]
 [coerce [p351](#)]
 [\$exitMode p??]

— defun coerceExit —

```
(defun |coerceExit| (arg1 mp)
  (let (x m e catchTag xp)
    (declare (special |$exitMode|))
    (setq x (first arg1))
    (setq m (second arg1))
    (setq e (third arg1))
    (setq mp (|resolve| m mp))
    (setq xp
      (|replaceExitEtc| x
        (setq catchTag (mkq (gensym))) '|TAGGEDexit| |$exitMode|))
    (|coerce| (list (list 'catch catchTag xp) m e) mp)))
```

—————

defplist compAtSign plist

We set up the `compAtSign` function to handle the `@` keyword by setting the `special` keyword on the `@` symbol property list.

— postvars —

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

—————

defun compAtSign

[[addDomain p248](#)]
 [comp [p590](#)]
 [coerce [p351](#)]

— defun compAtSign —

```
(defun compAtSign (form mode env)
  (let ((newform (second form)) (mprime (third form)) tmp)
```

```
(setq env (|addDomain| mprime env))
(when (setq tmp (|comp| newform mprime env)) (|coerce| tmp mode)))
```

defplist compCoerce plist

We set up the `compCoerce` function to handle the `::` keyword by setting the `special` keyword on the `::` symbol property list.

— postvars —

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

defun compCoerce

```
[addDomain p248]
[getmode p??]
[compCoerce1 p359]
[coerce p351]
```

— defun compCoerce —

```
(defun |compCoerce| (form mode env)
  (let (newform newmode tmp1 tmp4 z td)
    (setq newform (second form))
    (setq newmode (third form))
    (setq env (|addDomain| newmode env))
    (setq tmp1 (|getmode| newmode env))
    (cond
      ((setq td (|compCoerce1| newform newmode env))
        (|coerce| td mode))
      ((and (consp tmp1) (eq (qfirst tmp1) '|Mapping|)
            (consp (qrest tmp1)) (eq (qcddr tmp1) nil)
            (consp (qsecond tmp1))
            (eq (qcaadr tmp1) '|UnionCategory|))
        (setq z (qcdadr tmp1))
        (when
          (setq td
            (dolist (mode1 z tmp4)
              (setq tmp4 (or tmp4 (|compCoerce1| newform mode1 env))))
            (|coerce| (list (car td) newmode (third td)) mode))))))
```

defun compCoerce1

[comp p590]
 [resolve p361]
 [coerce p351]
 [coerceByModemap p359]
 [mkq p??]

— defun compCoerce1 —

```
(defun |compCoerce1| (form mode env)
  (let (m1 td tp gg pred code)
    (declare (special |$String| |$EmptyMode|))
    (when (setq td (or (|comp| form mode env) (|comp| form |$EmptyMode| env)))
      (setq m1 (if (stringp (second td)) |$String| (second td)))
      (setq mode (|resolve| m1 mode))
      (setq td (list (car td) m1 (third td)))
      (cond
        ((setq tp (|coerce| td mode)) tp)
        ((setq tp (|coerceByModemap| td mode)) tp)
        ((setq pred (|isSubset| mode (second td) env))
         (setq gg (gensym))
         (setq pred (subst gg '* pred :test #'equal))
         (setq code
          (list 'prog1
              (list 'let gg (first td))
              (cons '|check-subtype| (cons pred (list (mkq mode) gg))))))
        (list code mode (third td))))))
```

defun coerceByModemap

[modeEqual p362]
 [isSubset p??]
 [genDeltaEntry p??]

— defun coerceByModemap —

```
(defun |coerceByModemap| (arg1 mp)
  (let (x m env map cexpr u mm fn)
    (setq x (first arg1))
    (setq m (second arg1))
    (setq env (third arg1))
```

```

(setq u
(loop for modemap in (|getModemapList| '|coerce| 1 env)
do
  (setq map (first modemap))
  (setq cexpr (second modemap))
  when
    (and (consp map) (consp (qrest map))
         (consp (qcddr map))
         (eq (qcdddr map) nil)
         (or (|modeEqual| (second map) mp) (|isSubset| (second map) mp env))
         (or (|modeEqual| (third map) m) (|isSubset| m (third map) env)))
    collect modemap))
(when u
  (setq mm (first u))
  (setq fn (|genDeltaEntry| (cons '|coerce| mm)))
  (list (list '|call| fn x) mp env)))

```

defun autoCoerceByModemap

```

[getModemapList p259]
[modeEqual p362]
[member p??]
[get p??]
[stackMessage p??]
[$fromCoerceable p??]

```

— defun autoCoerceByModemap —

```

(defun |autoCoerceByModemap| (arg1 target)
  (let (x source e map cexpr u fn y)
    (declare (special |$fromCoerceable$|))
    (setq x (first arg1))
    (setq source (second arg1))
    (setq e (third arg1))
    (setq u
      (loop for modemap in (|getModemapList| '|autoCoerce| 1 e)
        do
          (setq map (first modemap))
          (setq cexpr (second modemap))
          when
            (and (consp map) (consp (qrest map)) (consp (qcddr map))
                 (eq (qcdddr map) nil)
                 (|modeEqual| (second map) target)
                 (|modeEqual| (third map) source))
            collect cexpr))

```

```

(when u
  (setq fn
    (let (result)
      (loop for item in u
        do
          (when (first item) (setq result (or result (second item))))))
    result))
(when fn
  (cond
    ((and (consp source) (eq (qfirst source) '|Union|)
      (|member| target (qrest source))))
    (cond
      ((and (setq y (|get| x '|condition| e))
        (let (result)
          (loop for u in y do
            (setq result
              (or result
                (and (consp u) (eq (qfirst u) '|case|) (consp (qrest u))
                  (consp (qcddr u))
                  (eq (qcdddr u) nil)
                  (equal (qthird u) target))))))
          result))
        (list (list '|call| fn x) target e))
      ((eq x '|$fromCoerceable$|) nil)
      (t
        (|stackMessage|
          (list '|cannot coerce: | x '|%1| '|          of mode: | source
            '|%1| '|          to: | target '| without a case statement|))))))
    (t
      (list (list '|call| fn x) target e))))))

```

defun resolve

[\[modeEqual p362\]](#)
[\[mkUnion p362\]](#)
[\[\\$String p345\]](#)
[\[\\$EmptyMode p172\]](#)
[\[\\$NoValueMode p172\]](#)

— defun resolve —

```

(defun |resolve| (din dout)
  (declare (special |$String| |$EmptyMode| |$NoValueMode|))
  (cond
    ((or (equal din |$NoValueMode|) (equal dout |$NoValueMode|)) |$NoValueMode|)

```

```

(equal dout |$EmptyModel|) din)
((and (not (equal din dout)) (or (stringp din) (stringp dout)))
 (cond
  ((|modeEqual| dout |$String|) dout)
  ((|modeEqual| din |$String|) nil)
  (t (|mkUnion| din dout))))
(t dout))

```

defun mkUnion

```

[union p??]
[$Rep p??]

```

— defun mkUnion —

```

(defun |mkUnion| (a b)
  (declare (special |$Rep|))
  (cond
   ((and (eq b '$) (consp |$Rep|) (eq (qfirst |$Rep|) '|Union|))
    (qrest |$Rep|))
   ((and (consp a) (eq (qfirst a) '|Union|))
    (cond
     ((and (consp b) (eq (qfirst b) '|Union|))
      (cons '|Union| (|union| (qrest a) (qrest b))))
     (t (cons '|Union| (|union| (list b) (qrest a))))))
   ((and (consp b) (eq (qfirst b) '|Union|))
    (cons '|Union| (|union| (list a) (qrest b))))
   (t (list '|Union| a b))))

```

defun This orders Unions

This orders Unions

— defun modeEqual —

```

(defun |modeEqual| (x y)
  (let (x1 y1)
    (cond
     ((or (atom x) (atom y)) (equal x y))
     ((not (eq1 (|#| x) (|#| y))) nil)
     ((and (consp x) (eq (qfirst x) '|Union|) (consp y) (eq (qfirst y) '|Union|))
      (setq x1 (qrest x))

```

```

(setq y1 (qrest y))
(loop for a in x1 do
  (loop for b in y1 do
    (when (|modeEqual| a b)
      (setq x1 (|delete| a x1))
      (setq y1 (|delete| b y1))
      (return nil))))
(unless (or x1 y1) t))
(t
 (let ((result t))
  (loop for u in x for v in y
    do (setq result (and result (|modeEqual| u v))))
  result))))

```

defun modeEqualSubst

```

[modeEqual p362]
[modeEqualSubst p363]
[length p??]

```

— defun modeEqualSubst —

```

(defun |modeEqualSubst| (m1 m env)
  (let (mp op z1 z2)
    (cond
      ((|modeEqual| m1 m) t)
      ((atom m1)
       (when (setq mp (car (|get| m1 '|value| env)))
         (|modeEqual| mp m)))
      ((and (consp m1) (consp m) (equal (qfirst m) (qfirst m1))
        (equal (|#| (qrest m1)) (|#| (qrest m))))
       (setq op (qfirst m1))
       (setq z1 (qrest m1))
       (setq z2 (qrest m))
       (let ((result t))
        (loop for xm1 in z1 for xm2 in z2
          do (setq result (and result (|modeEqualSubst| xm1 xm2 env))))
        result))
      (t nil))))

```

Chapter 7

Post Transformers

7.1 Direct called postparse routines

defun postTransform

[postTran p366]
[postTransform identp (vol5)]
[postTransformCheck p369]
[aplTran p401]

— defun postTransform —

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

```

      (setq tmp5 (qrest tmp4))
      (and (consp tmp5)
            (eq (qrest tmp5) nil)
            (progn (setq tt (qfirst tmp5)) t))))))
      (progn (setq l (qrest 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)))

```

defun postTran

```

[postAtom p367]
[postTran p366]
[unTuple p409]
[postTranList p368]
[postForm p370]
[postOp p367]
[postScriptsForm p368]

```

— 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 (consp op) (eq (qfirst op) '|elt|))
              (progn
                (setq tmp1 (qrest op))
                (and (consp tmp1)
                     (progn
                      (setq a (qfirst tmp1))
                      (setq tmp2 (qrest tmp1))
                      (and (consp tmp2)
                           (eq (qrest tmp2) nil)
                           (progn (setq b (qfirst tmp2)) t))))))
              (cons (|postTran| op) (cdr (|postTran| (cons b (cdr x))))))
            ((and (consp op) (eq (qfirst op) '|Scripts|))
              (postScriptsForm op

```

```

      (dolist (y (rest x) tmp3)
        (setq tmp3 (append tmp3 (|unTuple| (|postTran| y))))))
    ((not (equal op (setq y (postOp op))))
     (cons y (postTranList (cdr x))))
    (t (postForm x))))))

```

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)))

```

defun postAtom

[\$boot p??]

— defun postAtom —

```

(defun postAtom (x)
  (declare (special $boot))
  (cond
    ($boot x)
    ((eq1 x 0) '(|Zero|))
    ((eq1 x 1) '(|One|))
    ((eq x t) 't$)
    ((and (identp x) (getdatabase x 'niladic)) (list x))
    (t x)))

```

defun postTranList

[postTran p366]

— defun postTranList —

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

—————

defun postScriptsForm

[getScriptName p404]

[length p??]

[postTranScripts p368]

— defun postScriptsForm —

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

—————

defun postTranScripts

[postTranScripts p368]

[postTran p366]

— defun postTranScripts —

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

```

      (and (consp tmp1) (eq (qrest tmp1) nil))))
    (postTranScripts (qfirst tmp1)))
  ((and (consp a) (eq (qfirst a) '|;|))
   (dolist (y (qrest a) tmp2)
    (setq tmp2 (append tmp2 (postTranScripts y)))))
  ((and (consp a) (eq (qfirst a) '|,|))
   (dolist (y (qrest a) tmp3)
    (setq tmp3 (append tmp3 (fn (|postTran| y)))))
   (t (list (|postTran| a))))))

```

defun postTransformCheck

[postcheck p369]
 [\$defOp p??]

— defun postTransformCheck —

```

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

```

defun postcheck

[setDefOp p400]
 [postcheck p369]

— defun postcheck —

```

(defun postcheck (x)
  (cond
    ((atom x) nil)
    ((and (consp x) (eq (qfirst x) 'def) (consp (qrest x)))
     (setDefOp (qsecond x))
     (postcheck (qcddr x)))
    ((and (consp x) (eq (qfirst x) 'quote)) nil)
    (t (postcheck (car x)) (postcheck (cdr x)))))

```

defun postError

```
[bumperrorcount p545]
[$defOp p??]
[$InteractiveMode p??]
[$postStack p??]
```

— defun postError —

```
(defun postError (msg)
  (let (xmsg)
    (declare (special |$defOp| |$postStack| |$InteractiveMode|))
    (bumperrorcount '|precompilation|)
    (setq xmsg
      (if (and (not (eq |$defOp| '|$defOp|)) (null |$InteractiveMode|))
          (cons |$defOp| (cons ": " msg))
          msg))
    (push xmsg |$postStack|)
    nil))
```

defun postForm

```
[postTranList p368]
[internal p??]
[postTran p366]
[postError p370]
[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)
              (exit op))))
          (t
           (setq arglp (postTranList argl))
           (setq opp
             (seq
              (exit op)
              (exit op)))))))
    (exit op))
```

```

(when $boot (exit op))
(when (or (get1 op '|Led|) (get1 op '|Nud|) (eq op 'in)) (exit op))
(setq numOfArgs
  (cond
    ((and (consp arglp) (eq (qrest arglp) nil) (consp (qfirst arglp))
      (eq (qcaar arglp) '@Tuple|))
      (|#| (qcdar arglp)))
    (t 1)))
(internal '* (princ-to-string numOfArgs) (pname op)))
(cons opp arglp))
((and (consp op) (eq (qfirst op) '|Scripts|))
  (append (|postTran| op) (postTranList argl)))
(t
  (setq u (postTranList u))
  (cond
    ((and (consp u) (consp (qfirst u)) (eq (qcaar u) '@Tuple|))
      (postError
        (cons " "
          (append (|bright| u)
            (list "is illegal because tuples cannot be applied!" '|%1|
              " Did you misuse infix dot?")))))
    (t)))
(cond
  ((and (consp x) (consp (qrest x)) (eq (qcddr x) nil)
    (consp (qsecond x)) (eq (qcaadr x) '@Tuple|))
    (cons (car x) (qcdadr x)))
  (t x))))

```

7.2 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:

<code>add</code>	<code>postAdd</code>
<code>@</code>	<code>postAtSign</code>
<code>:BF:</code>	<code>postBigFloat</code>
<code>Block</code>	<code>postBlock</code>
<code>CATEGORY</code>	<code>postCategory</code>
<code>COLLECT</code>	<code>postCollect</code>

```

:          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

```

defplist postAdd plist

— postvars —

```

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

```

—————

defun postAdd

```

[postTran p366]
[postCapsule p373]

```

— defun postAdd —

```

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

```

defun postCapsule

[checkWarning p549]
 [postBlockItem p373]
 [postBlockItemList p373]
 [postFlatten p382]

— defun postCapsule —

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

defun postBlockItemList

[postBlockItem p373]

— defun postBlockItemList —

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

defun postBlockItem

[postTran p366]

— defun postBlockItem —

```

(defun |postBlockItem| (x)
  (let ((tmp1 t) tmp2 y tt z)
    (setq x (|postTran| x))
    (if
      (and (consp x) (eq (qfirst x) '|@Tuple|))
      (progn
        (and (consp (qrest x))
          (progn (setq tmp2 (reverse (qrest x))) t)
          (consp tmp2)
          (progn
            (and (consp (qfirst tmp2)) (eq (qcaar tmp2) '|:|)
              (progn
                (and (consp (qcdar tmp2))
                  (progn
                    (setq y (qcadar tmp2))
                    (and (consp (qcddar tmp2))
                      (eq (qcdddar tmp2) nil)
                      (progn (setq tt (qcaddar tmp2)) t)))))))
            (progn (setq z (qrest tmp2)) t)
            (progn (setq z (nreverse z)) T)))
        (do ((tmp6 nil (null tmp1)) (tmp7 z (cdr tmp7)) (x nil))
          ((or tmp6 (atom tmp7)) tmp1)
          (setq x (car tmp7))
          (setq tmp1 (and tmp1 (identp x)))))
      (cons '|:| (cons (cons 'listof (append z (list y))) (list tt)))
      x)))

```

defplist postAtSign plist

— postvars —

```

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

```

defun postAtSign

[postTran p366]
 [postType p375]

— defun postAtSign —

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

defun postType

[postTran p366]
[unTuple p409]

— defun postType —

```
(defun |postType| (typ)
  (let (source target)
    (cond
      ((and (consp typ) (eq (qfirst typ) '->) (consp (qrest typ))
        (consp (qcddr typ)) (eq (qcdddr typ) nil))
       (setq source (qsecond typ))
       (setq target (qthird typ))
       (cond
         ((eq source '|constant|)
          (list (list (|postTran| target)) '|constant|))
         (t
          (list (cons '|Mapping|
                     (cons (|postTran| target)
                           (|unTuple| (|postTran| source)))))))
      ((and (consp typ) (eq (qfirst typ) '->)
        (consp (qrest typ)) (eq (qcddr typ) nil))
       (list (list '|Mapping| (|postTran| (qsecond typ))))
      (t (list (|postTran| typ)))))
```

defplist postBigFloat plist

— postvars —

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

defun postBigFloat

[postTran p366]

[\$boot p??]

[\$InteractiveMode p??]

— defun postBigFloat —

```

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

```

defplist postBlock plist

— postvars —

```

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

```

defun postBlock

[postBlockItemList p373]

[postTran p366]

— 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))))))
```

defplist postCategory plist

— postvars —

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

defun postCategory

```
[postTran p366]
[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))))))))
```

defun postCollect,finish

```
[postMakeCons p378]
[tuple2List p549]
```

[postTranList p368]

— defun postCollect,finish —

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

—————

defun postMakeCons

[postMakeCons p378]

[postTran p366]

— defun postMakeCons —

```
(defun |postMakeCons| (args)
  (let (a b)
    (cond
      ((null args) '|nil|)
      ((and (consp args) (consp (qfirst args)) (eq (qcaar args) '|:|)
        (consp (qcдар args)) (eq (qcddar args) nil))
       (setq a (qcдар args))
       (setq b (qrest args))
       (if b
        (list '|append| (|postTran| a) (|postMakeCons| b))
        (|postTran| a)))
      (t (list '|cons| (|postTran| (car args)) (|postMakeCons| (cdr args))))))
```

defplist postCollect plist

— postvars —

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

defun postCollect

```
[postCollect,finish p377]
[postCollect p379]
[postIteratorList p380]
[postTran p366]
```

— 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 (consp x) (consp (qfirst x)) (eq (qcaar x) '|elt|)
            (consp (qcddar x)) (consp (qcdddar x))
            (eq (qcddddar x) nil)
            (eq (qcaddar x) '|construct|))
        (|postCollect|
         (cons (list '|elt| (qcadar x) 'collect)
               (append m (list (cons '|construct| (qrest x)))))))
      (t
       (setq itl (|postIteratorList| m))
       (setq x
        (if (and (consp x) (eq (qfirst x) '|construct|)
                  (consp (qrest x)) (eq (qcddr x) nil))
            (qsecond x)
            x))
       (|postCollect,finish| constructOp itl (|postTran| x))))))
```

defun postIteratorList

[postTran p366]
 [postInSeq p388]
 [postIteratorList p380]

— defun postIteratorList —

```
(defun |postIteratorList| (args)
  (let (z p y u a b)
    (cond
      ((consp args)
       (setq p (|postTran| (qfirst args)))
       (setq z (qrest args))
       (cond
         ((and (consp p) (eq (qfirst p) 'in) (consp (qrest p))
          (consp (qcddr p)) (eq (qcdddr p) nil))
          (setq y (qsecond p))
          (setq u (qthird p))
          (cond
            ((and (consp u) (eq (qfirst u) '|\\|') (consp (qrest u))
             (consp (qcddr u)) (eq (qcdddr u) nil))
             (setq a (qsecond u))
             (setq b (qthird u))
             (cons (list 'in y (|postInSeq| a))
                   (cons (list '|\\|' b)
                         (|postIteratorList| z))))
            (t (cons (list 'in y (|postInSeq| u)) (|postIteratorList| z))))))
         (t (cons p (|postIteratorList| z))))))
    (t args)))
```

—

defplist postColon plist

— postvars —

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

—

defun postColon

[postTran p366]
 [postType p375]

— defun postColon —

```
(defun |postColon| (u)
  (cond
    ((and (consp u) (eq (qfirst u) '|:|)
          (consp (qrest u)) (eq (qcddr u) nil))
      (list '|:| (|postTran| (qsecond u))))
    ((and (consp u) (eq (qfirst u) '|:|) (consp (qrest u))
          (consp (qcddr u)) (eq (qcdddr u) nil))
      (cons '|:| (cons (|postTran| (second u)) (|postType| (third u)))))))
```

—————

defplist postColonColon plist

— postvars —

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

—————

defun postColonColon

[postForm p370]
 [\$boot p??]

— defun postColonColon —

```
(defun |postColonColon| (u)
  (if (and $boot (consp u) (eq (qfirst u) '|::|) (consp (qrest u))
        (consp (qcddr u)) (eq (qcdddr u) nil))
      (intern (princ-to-string (third u)) (second u))
      (postForm u)))
```

—————

defplist postComma plist

— postvars —

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

—————

defun postComma

[postTuple p398]
[comma2Tuple p382]

— defun postComma —

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

—————

defun comma2Tuple

[postFlatten p382]

— defun comma2Tuple —

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

—————

defun postFlatten

[postFlatten p382]

— defun postFlatten —

```
(defun |postFlatten| (x op)
  (let (a b)
    (cond
```

```
((and (consp x) (equal (qfirst x) op) (consp (qrest x))
      (consp (qcddr x)) (eq (qcdddr x) nil))
  (setq a (qsecond x))
  (setq b (qthird x))
  (append (|postFlatten| a op) (|postFlatten| b op)))
(t (list x))))
```

deflist postConstruct plist

— postvars —

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

defun postConstruct

```
[comma2Tuple p382]
[postTranSegment p384]
[postMakeCons p378]
[tuple2List p549]
[postTranList p368]
[postTran p366]
```

— defun postConstruct —

```
(defun |postConstruct| (u)
  (let (b a tmp4 tmp7)
    (cond
      ((and (consp u) (eq (qfirst u) '|construct|)
            (consp (qrest u)) (eq (qcddr u) nil))
        (setq b (qsecond u))
        (setq a
          (if (and (consp b) (eq (qfirst b) '|,|'))
              (|comma2Tuple| b)
              b))
        (cond
          ((and (consp a) (eq (qfirst a) 'segment) (consp (qrest a))
                (consp (qcddr a)) (eq (qcdddr a) nil))
            (list '|construct| (|postTranSegment| (second a) (third a))))
          ((and (consp a) (eq (qfirst a) '@Tuple|))
```

```

(cond
  ((dolist (x (qrest a) tmp4)
    (setq tmp4
      (or tmp4
        (and (consp x) (eq (qfirst x) '|:|)
          (consp (qrest x)) (eq (qcddr x) nil))))))
    (|postMakeCons| (qrest a)))
  ((dolist (x (qrest a) tmp7)
    (setq tmp7 (or tmp7 (and (consp x) (eq (qfirst x) 'segment)))))
    (|tuple2List| (qrest a)))
  (t (cons '|construct| (postTranList (qrest a)))))
(t (list '|construct| (|postTran| a))))
(t u)))

```

defun postTranSegment

[postTran p366]

— defun postTranSegment —

```

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

```

defplist postDef plist

— postvars —

```

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

```

defun postDef

[postMDef p391]

[recordHeaderDocumentation p472]

[postTran p366]

[postDefArgs p386]

```
[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 (consp lhs) (eq (qfirst lhs) '|macro|)
            (consp (qrest lhs)) (eq (qcddr lhs) nil))
        (|postMDef| (list '==> (second lhs) rhs))
        (progn
          (unless $boot (|recordHeaderDocumentation| nil))
          (when (not (eql |$maxSignatureLineNumber| 0))
            (setq |$docList|
                  (cons (cons '|constructor| |$headerDocumentation|) |$docList|))
            (setq |$maxSignatureLineNumber| 0))
          (setq lhs (|postTran| lhs))
          (setq tmp1
            (if (and (consp lhs) (eq (qfirst lhs) '|:|') (cdr lhs) (list lhs nil)))
            (setf form (first tmp1))
            (setf targetType (second tmp1))
            (when (and (null |$InteractiveMode|) (atom form)) (setf form (list form)))
            (setf newLhs
              (if (atom form)
                  form
                  (progn
                    (setf tmp1
                      (dolist (x form (nreverse0 tmp4))
                        (push
                          (if (and (consp x) (eq (qfirst x) '|:|') (consp (qrest x))
                                      (consp (qcddr x)) (eq (qcdddr x) nil))
                              (second x)
                              x)
                          tmp4)))
                    (setf op (car tmp1))
                    (setf arg1 (cdr tmp1))
                    (cons op (|postDefArgs| arg1))))))
          (setf argTypeList
            (unless (atom form)
```

```

(dolist (x (cdr form) (nreverse0 tmp6))
  (push
    (when (and (consp x) (eq (qfirst x) '[:]) (consp (qrest x))
      (consp (qcddr x)) (eq (qcdddr 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))))))

```

defun postDefArgs

[postError p370]
 [postDefArgs p386]

— defun postDefArgs —

```

(defun |postDefArgs| (args)
  (let (a b)
    (cond
      ((null args) args)
      ((and (consp args) (consp (qfirst args)) (eq (qcaar args) '[:])
        (consp (qcddr args)) (eq (qcdddr args) nil))
        (setq a (qcadar args))
        (setq b (qrest args))
        (cond
          (b (postError
            (list " Argument" a "of indefinite length must be last"))
            ((or (atom a) (and (consp a) (eq (qfirst a) 'quote)))
              a)
            (t
              (postError
                (list " Argument" a "of indefinite length must be a name")))))
          (t (cons (car args) (|postDefArgs| (cdr args)))))))

```

defplist postExit plist

— postvars —

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

defun postExit

[postTran p366]

— defun postExit —

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

defplist postIf plist

— postvars —

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

defun postIf

[nreverse0 p??]
 [postTran p366]
 [\$boot p??]

— defun postIf —

```
(defun |postIf| (arg)
  (let (tmp1)
    (if (null (and (consp arg) (eq (qfirst arg) '|if|)))
        arg
        (cons 'if
              (dolist (x (qrest arg) (nreverse0 tmp1))
                (push
```

```
(if (and (null (setq x (|postTran| x))) (null $boot)) 'noBranch| x)
tmp1))))))
```

defplist postin plist

— postvars —

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

defun postin

```
[systemErrorHere p??]
[postTran p366]
[postInSeq p388]
```

— defun postin —

```
(defun |postin| (arg)
  (if (null (and (consp arg) (eq (qfirst arg) 'in|) (consp (qrest arg))
                (consp (qcddr arg)) (eq (qcddr arg) nil)))
      (|systemErrorHere| "postin")
      (list 'in| (|postTran| (second arg)) (|postInSeq| (third arg)))))
```

defun postInSeq

```
[postTranSegment p384]
[tuple2List p549]
[postTran p366]
```

— defun postInSeq —

```
(defun |postInSeq| (seq)
  (cond
    ((and (consp seq) (eq (qfirst seq) 'segment) (consp (qrest seq))
```



```

      (consp (qcddr seq)) (eq (qcdddr seq) nil))
    (|postTranSegment| (second seq) (third seq)))
  ((and (consp seq) (eq (qfirst seq) '@Tuple|))
    (|tuple2List| (qrest seq)))
  (t (|postTran| seq))))

```

defplist postIn plist

— postvars —

```

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

```

defun postIn

```

[systemErrorHere p??]
[postTran p366]
[postInSeq p388]

```

— defun postIn —

```

(defun |postIn| (arg)
  (if (null (and (consp arg) (eq (qfirst arg) 'in) (consp (qrest arg))
    (consp (qcddr arg)) (eq (qcdddr arg) nil)))
    (|systemErrorHere| "postIn")
    (list 'in (|postTran| (second arg)) (|postInSeq| (third arg)))))

```

defplist postJoin plist

— postvars —

```

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

```

defun postJoin

[postTran p366]
 [postTranList p368]

— defun postJoin —

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

—————

defplist postMapping plist

— postvars —

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

—————

defun postMapping

[postTran p366]
 [unTuple p409]

— defun postMapping —

```
(defun |postMapping| (u)
  (if (null (and (consp u) (eq (qfirst u) '|->|) (consp (qrest u))
                (consp (qcddr u)) (eq (qcdddr u) nil))))
      u
      (cons '|Mapping|
            (cons (|postTran| (third u))
                  (|unTuple| (|postTran| (second u)))))))
```

—————

defplist postMDef plist

— postvars —

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

—————

defun postMDef

```
[postTran p366]
[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 (consp lhs) (eq (qfirst lhs) '|:|) (cdr lhs) (list lhs nil)))
        (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 (consp x) (eq (qfirst x) '|:|) (consp (qrest x))) (second x) x)
           tmp4)))
       (setq typeList
        (cons targetType
         (dolist (x (qrest form) (nreverse0 tmp5))
           (push
```

```

      (when (and (consp x) (eq (qfirst x) '|:|) (consp (qrest x))
                (consp (qcddr x)) (eq (qcddr x) nil))
            (third x))
      tmp5))))
(list 'mdef newLhs typeList
      (dolist (x form (nreverse0 tmp8)) (push nil tmp8))
      (|postTran| rhs))))))

```

defplist postPretend plist

— postvars —

```

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

```

defun postPretend

[postTran [p366](#)]
 [postType [p375](#)]

— defun postPretend —

```

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

```

defplist postQUOTE plist

— postvars —

```

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

```

defun postQUOTE

— defun postQUOTE —

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

—————

defplist postReduce plist

— postvars —

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

—————

defun postReduce

```
[postTran p366]
[postReduce p393]
[$InteractiveMode p??]
```

— defun postReduce —

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

—————

defplist postRepeat plist

— postvars —

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

—————

defun postRepeat

[postIteratorList p380]

[postTran p366]

— 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))))))
```

—————

defplist postScripts plist

— postvars —

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

—————

defun postScripts

[getScriptName p404]

[postTranScripts p368]

— defun postScripts —

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

defplist postSemiColon plist

— postvars —

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

defun postSemiColon

```
[postBlock p376]
[postFlattenLeft p395]
```

— defun postSemiColon —

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

defun postFlattenLeft

```
[postFlattenLeft p395]
```

— defun postFlattenLeft —

```
(defun |postFlattenLeft| (x op)
  (let (a b)
    (cond
      ((and (consp x) (equal (qfirst x) op) (consp (qrest x))
            (consp (qcddr x)) (eq (qcdddr x) nil))
       (setq a (qsecond x))
       (setq b (qthird x))
       (append (|postFlattenLeft| a op) (list b)))
      (t (list x)))))
```

defplist postSignature plist

— postvars —

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

defun postSignature

[postType p375]
 [removeSuperfluousMapping p396]
 [killColons p397]

— defun postSignature —

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

defun removeSuperfluousMapping

— defun removeSuperfluousMapping —

```
(defun |removeSuperfluousMapping| (sig1)
  (if (and (consp sig1) (consp (qfirst sig1))) (eq (qcaar sig1) '|Mapping|))
      (cons (cdr (qfirst sig1)) (qrest sig1))
      sig1))
```

defun killColons

[killColons p397]

— defun killColons —

```
(defun |killColons| (x)
  (cond
    ((atom x) x)
    ((and (consp x) (eq (qfirst x) '|Record|)) x)
    ((and (consp x) (eq (qfirst x) '|Union|)) x)
    ((and (consp x) (eq (qfirst x) '|:|) (consp (qrest x))
          (consp (qcddr x)) (eq (qcddr x) nil))
     (|killColons| (third x)))
    (t (cons (|killColons| (car x)) (|killColons| (cdr x))))))
```

defplist postSlash plist

— postvars —

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

defun postSlash

[postTran p366]

— defun postSlash —

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

defplist postTuple plist

— postvars —

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

—————

defun postTuple

[postTranList p368]

— defun postTuple —

```
(defun |postTuple| (arg)
  (cond
    ((and (consp arg) (eq (qrest arg) nil) (eq (qfirst arg) '|@Tuple|))
     arg)
    ((and (consp arg) (eq (qfirst arg) '|@Tuple|) (consp (qrest arg)))
     (cons '|@Tuple| (postTranList (cdr arg))))))
```

—————

defplist postTupleCollect plist

— postvars —

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

—————

defun postTupleCollect

[postCollect p379]

— 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)))))))
```

defplist postWhere plist

— postvars —

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

defun postWhere

[postTran p366]
[postTranList p368]

— defun postWhere —

```
(defun |postWhere| (arg)
  (let (b x)
    (setq b (third arg))
    (setq x (if (and (consp b) (eq (qfirst b) '|Block|)) (qrest b) (list b)))
    (cons '|where| (cons (|postTran| (second arg)) (postTranList x)))))
```

defplist postWith plist

— postvars —

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

defun postWith

```
[postTran p366]
[$insidePostCategoryIfTrue p??]
```

— defun postWith —

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

7.3 Support routines**defun setDefOp**

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

— defun setDefOp —

```
(defun setDefOp (f)
  (let (tmp1)
    (declare (special |$defOp| |$topOp|))
    (when (and (consp f) (eq (qfirst f) '|:|)
              (consp (setq tmp1 (qrest f)))))
      (setq f (qfirst tmp1)))
    (unless (atom f) (setq f (car f)))
    (if |$topOp|
      (setq |$defOp| f)
      (setq |$topOp| f))))
```

defun aplTran

```
[aplTran1 p401]
[containsBang p404]
[$genno p??]
[$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))))))
```

defun aplTran1

```
[aplTranList p403]
[aplTran1 p401]
[hasAplExtension p403]
[nreverse0 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 (consp argl)
                  (progn
```

```

      (setq f (qfirst arg1))
      (setq tmp1 (qrest arg1))
      (and (consp tmp1)
            (eq (qrest tmp1) nil)
            (progn
              (setq y (qfirst tmp1))
              t))))
    (cond
      ((and (consp y)
            (progn
              (setq opprime (qfirst y))
              (setq yprime (qrest 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 (consp tmp1)
          (progn
            (setq arglAssoc (qfirst tmp1))
            (setq futureArgl (qrest tmp1))
            t)))
    (cons '|reshape|
      (cons
        (cons 'collect
          (append
            (do ((tmp3 arglAssoc (cdr tmp3)) (tmp4 nil))
                ((or (atom tmp3)
                     (progn (setq tmp4 (car tmp3)) nil)
                     (progn
                       (setq g (car tmp4))
                       (setq a (cdr tmp4))
                       nil))
                  (nreverse0 tmp2))
            (push (list 'in g (list '|ravel| a))) tmp2))
          (list (aplTran1 (cons op futureArgl))))
          (list (cdar arglAssoc))))
      (t (cons op argl))))))

```

defun aplTranList

[aplTran1 p401]

[aplTranList p403]

— defun aplTranList —

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

—

defun hasAplExtension

[nreverse0 p??]

[deepestExpression p404]

[genvar p??]

[aplTran1 p401]

— defun hasAplExtension —

```
(defun hasAplExtension (arg1)
  (let (tmp2 tmp3 y z g arglAssoc u)
    (when
      (dolist (x arg1 tmp2)
        (setq tmp2 (or tmp2 (and (consp x) (eq (qfirst x) '!)))))
      (setq u
        (dolist (x arg1 (nreverse0 tmp3))
          (push
            (if (and (consp x) (eq (qfirst x) '!)
                (consp (qrest x)) (eq (qcddr x) nil))
                (progn
                  (setq y (qsecond x))
                  (setq z (deepestExpression y))
                  (setq arglAssoc
                    (cons (cons (setq g (genvar)) (aplTran1 z)) arglAssoc))
                  (subst g z y :test #'equal))
                  x)
            tmp3)))
      (cons arglAssoc u))))
```

—

defun deepestExpression

[deepestExpression p404]

— defun deepestExpression —

```
(defun deepestExpression (x)
  (if (and (consp x) (eq (qfirst x) '!))
      (consp (qrest x)) (eq (qcddr x) nil))
      (deepestExpression (qsecond x)
                          x)))
```

defun containsBang

[containsBang p404]

— defun containsBang —

```
(defun containsBang (u)
  (let (tmp2)
    (cond
      ((atom u) (eq u '!))
      ((and (consp u) (equal (qfirst u) 'quote)
            (consp (qrest u)) (eq (qcddr u) nil))
       nil)
      (t
       (dolist (x u tmp2)
         (setq tmp2 (or tmp2 (containsBang x))))))))
```

defun getScriptName

```
[getScriptName identp (vol5)]
[postError p370]
[internl p??]
[decodeScripts p405]
[getScriptName pname (vol5)]
```

— defun getScriptName —

```
(defun getScriptName (op a numberOfFunctionalArgs)
```



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

defun decodeScripts

[strconc p??]
[decodeScripts p[405](#)]

— defun decodeScripts —

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

Chapter 8

DEF forms

defvar \$defstack

— initvars —

(defvar \$defstack nil)

—————

defvar \$is-spill

— initvars —

(defvar \$is-spill nil)

—————

defvar \$is-spill-list

— initvars —

(defvar \$is-spill-list nil)

—————

defvar \$vl

— initvars —

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

—————

defvar \$is-gensymlist

— initvars —

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

—————

defvar \$initial-gensym

— initvars —

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

—————

defvar \$is-eqlist

— initvars —

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

—————

defun hackforis

[hackforis1 p[409](#)]

— defun hackforis —

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

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))
```

defun unTuple

— defun unTuple —

```
(defun |unTuple| (x)  
  (if (and (consp x) (eq (qfirst x) '|@Tuple|))  
      (qrest x)  
      (list x)))
```

defun errhuh

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

— defun errhuh —

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

Chapter 9

PARSE forms

9.1 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(tmp 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)>>!' ') ;

NBGlyphTok{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 ;

```

9.2 The PARSE code

defvar \$tmptok

— initvars —

```
(defvar |tmptok| nil)
```

—————

defvar \$tok

— initvars —

```
(defvar tok nil)
```

—————

defvar \$ParseMode

— initvars —

```
(defvar |ParseMode| nil)
```

—————

defvar \$definition-name

— initvars —

```
(defvar definition-name nil)
```

—————

defvar \$lablasoc

— initvars —

```
(defvar lablasoc nil)
```

—————

defun PARSE-NewExpr

```
[match-string p454]
[action p467]
[PARSE-NewExpr processSynonyms (vol5)]
[must p466]
[current-symbol p460]
[PARSE-Statement p422]
[definition-name p417]
```

— defun PARSE-NewExpr —

```
(defun |PARSE-NewExpr| ()
  (or (and (match-string ")") (action (|processSynonyms|))
      (must (|PARSE-Command|))))
```

```
(and (action (setq definition-name (current-symbol)))
      (|PARSE-Statement|)))
```

defun PARSE-Command

```
[match-advance-string p455]
[must p466]
[PARSE-SpecialKeyWord p418]
[PARSE-SpecialCommand p419]
[push-reduction p468]
```

— defun PARSE-Command —

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

defun PARSE-SpecialKeyWord

```
[match-current-token p459]
[action p467]
[token-symbol p??]
[current-token p461]
[PARSE-SpecialKeyWord unAbbreviateKeyword (vol5)]
[current-symbol p460]
```

— defun PARSE-SpecialKeyWord —

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

defun PARSE-SpecialCommand

```

[match-advance-string p455]
[bang p??]
[optional p467]
[PARSE-Expression p425]
[push-reduction p468]
[PARSE-SpecialCommand p419]
[pop-stack-1 p550]
[PARSE-CommandTail p421]
[must p466]
[current-symbol p460]
[action p467]
[PARSE-TokenList p420]
[PARSE-TokenCommandTail p419]
[star p467]
[PARSE-PrimaryOrQM p421]
[PARSE-CommandTail p421]
[$noParseCommands p??]
[$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|))))))

```

—————

defun PARSE-TokenCommandTail

```

[bang p??]
[optional p467]

```

```

[star p467]
[PARSE-TokenOption p420]
[atEndOfLine p??]
[push-reduction p468]
[PARSE-TokenCommandTail p419]
[pop-stack-2 p550]
[pop-stack-1 p550]
[action p467]
[PARSE-TokenCommandTail systemCommand (vol5)]

```

— **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)))))

```

—————→

defun PARSE-TokenOption

```

[match-advance-string p455]
[must p466]
[PARSE-TokenList p420]

```

— **defun PARSE-TokenOption** —

```

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

```

—————→

defun PARSE-TokenList

```

[star p467]
[isTokenDelimiter p457]
[push-reduction p468]
[current-symbol p460]
[action p467]
[advance-token p462]

```

— **defun PARSE-TokenList** —


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

defun PARSE-CommandTail

```
[bang p??]
[optional p467]
[star p467]
[push-reduction p468]
[PARSE-Option p422]
[PARSE-CommandTail p421]
[pop-stack-2 p550]
[pop-stack-1 p550]
[action p467]
[PARSE-CommandTail systemCommand (vol5)]
```

— 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)))))
```

defun PARSE-PrimaryOrQM

```
[match-advance-string p455]
[push-reduction p468]
[PARSE-PrimaryOrQM p421]
[PARSE-Primary p434]
```

— defun PARSE-PrimaryOrQM —

```
(defun |PARSE-PrimaryOrQM| ()
  (or (and (match-advance-string "?")
    (push-reduction '|PARSE-PrimaryOrQM| '?)
```

```
(|PARSE-Primary|))
```

defun PARSE-Option

```
[match-advance-string p455]
[must p466]
[star p467]
[PARSE-PrimaryOrQM p421]
```

— defun PARSE-Option —

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

defun PARSE-Statement

```
[PARSE-Expr p426]
[optional p467]
[star p467]
[match-advance-string p455]
[must p466]
[push-reduction p468]
[pop-stack-2 p550]
[pop-stack-1 p550]
```

— 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))))))))))
```

defun PARSE-InfixWith

[PARSE-With p423]
 [push-reduction p468]
 [pop-stack-2 p550]
 [pop-stack-1 p550]

— defun PARSE-InfixWith —

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

—————

defun PARSE-With

[match-advance-string p455]
 [must p466]
 [push-reduction p468]
 [pop-stack-1 p550]

— 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))))))
```

—————

defun PARSE-Category

[match-advance-string p455]
 [must p466]
 [bang p??]
 [optional p467]
 [push-reduction p468]
 [PARSE-Expression p425]
 [PARSE-Category p423]
 [pop-stack-3 p551]
 [pop-stack-2 p550]
 [pop-stack-1 p550]

— defun PARSE-Category —

defun PARSE-Expression

[PARSE-Expr p426]
 [PARSE-rightBindingPowerOf p428]
 [make-symbol-of p460]
 [push-reduction p468]
 [pop-stack-1 p550]
 [ParseMode p417]
 [prior-token p94]

— 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))))
```

defun PARSE-Import

[match-advance-string p455]
 [must p466]
 [PARSE-Expr p426]
 [bang p??]
 [optional p467]
 [star p467]
 [push-reduction p468]
 [pop-stack-2 p550]
 [pop-stack-1 p550]

— 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))))))
```

defun PARSE-Expr

[PARSE-NudPart p426]
 [PARSE-LedPart p426]
 [optional p467]
 [star p467]
 [push-reduction p468]
 [pop-stack-1 p550]

— 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))))
```

defun PARSE-LedPart

[PARSE-Operation p427]
 [push-reduction p468]
 [pop-stack-1 p550]

— defun PARSE-LedPart —

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

defun PARSE-NudPart

[PARSE-Operation p427]
 [PARSE-Reduction p431]
 [PARSE-Form p431]
 [push-reduction p468]
 [pop-stack-1 p550]

[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))))
```

—————

defun PARSE-Operation

```
[match-current-token p459]
[current-symbol p460]
[PARSE-leftBindingPowerOf p427]
[lt p??]
[getl p??]
[action p467]
[PARSE-rightBindingPowerOf p428]
[PARSE-getSemanticForm p428]
[elemn p??]
[ParseMode p417]
[rbp p??]
[tmptok p416]
```

— 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))))
```

—————

defun PARSE-leftBindingPowerOf

```
[getl p??]
[elemn p??]
```

— defun PARSE-leftBindingPowerOf —

```
(defun |PARSE-leftBindingPowerOf| (x ind)
  (declare (special x ind))
  (let ((y (get1 x ind))) (if y (elemn y 3 0) 0)))
```

—————

defun PARSE-rightBindingPowerOf

```
[get1 p??]
[elemn p??]
```

— defun PARSE-rightBindingPowerOf —

```
(defun |PARSE-rightBindingPowerOf| (x ind)
  (declare (special x ind))
  (let ((y (get1 x ind))) (if y (elemn y 4 105) 105)))
```

—————

defun PARSE-getSemanticForm

```
[PARSE-Prefix p428]
[PARSE-Infix p429]
```

— 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|))))
```

—————

defun PARSE-Prefix

```
[push-reduction p468]
[current-symbol p460]
[action p467]
[advance-token p462]
```


[optional p467]
 [PARSE-TokTail p430]
 [must p466]
 [PARSE-Expression p425]
 [push-reduction p468]
 [pop-stack-2 p550]
 [pop-stack-1 p550]

— 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))))))
```

defun PARSE-Infix

[push-reduction p468]
 [current-symbol p460]
 [action p467]
 [advance-token p462]
 [optional p467]
 [PARSE-TokTail p430]
 [must p466]
 [PARSE-Expression p425]
 [pop-stack-2 p550]
 [pop-stack-1 p550]

— 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) )))))
```

defun PARSE-TokTail

[current-symbol p460]
 [current-char p463]
 [char-eq p464]
 [copy-token p??]
 [action p467]
 [PARSE-Qualification p430]
 [\$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)))))
```

—————

defun PARSE-Qualification

[match-advance-string p455]
 [must p466]
 [PARSE-Primary1 p434]
 [push-reduction p468]
 [dollarTran p465]
 [pop-stack-1 p550]

— defun PARSE-Qualification —

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

—————

defun PARSE-Reduction

[PARSE-ReductionOp p431]
 [must p466]
 [PARSE-Expr p426]
 [push-reduction p468]
 [pop-stack-2 p550]
 [pop-stack-1 p550]

— 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) )))))
```

—————

defun PARSE-ReductionOp

[getl p??]
 [current-symbol p460]
 [match-next-token p460]
 [action p467]
 [advance-token p462]

— 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)))))
```

—————

defun PARSE-Form

[match-advance-string p455]
 [bang p??]
 [optional p467]
 [must p466]
 [push-reduction p468]
 [pop-stack-1 p550]

[PARSE-Application p432]

— **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|)))
```

—————

defun PARSE-Application

[PARSE-Primary p434]

[optional p467]

[star p467]

[PARSE-Selector p433]

[PARSE-Application p432]

[push-reduction p468]

[pop-stack-2 p550]

[pop-stack-1 p550]

— **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)))))))
```

—————

[match-advance-string p455]
[must p466]
[PARSE-Name p441]

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

```
[current-symbol p460]
[char-ne p464]
[current-char p463]
[match-advance-string p455]
[must p466]
[PARSE-PrimaryNoFloat p434]
[push-reduction p468]
[pop-stack-2 p550]
[pop-stack-1 p550]
[PARSE-Float p435]
[PARSE-Primary p434]
[$boot p??]
```

```
(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))))))

```

defun PARSE-PrimaryNoFloat

[PARSE-Primary1 p434]
 [optional p467]
 [PARSE-TokTail p430]

— defun PARSE-PrimaryNoFloat —

```

(defun |PARSE-PrimaryNoFloat| ()
  (and (|PARSE-Primary1|) (optional (|PARSE-TokTail|))))

```

defun PARSE-Primary

[PARSE-Float p435]
 [PARSE-PrimaryNoFloat p434]

— defun PARSE-Primary —

```

(defun |PARSE-Primary| ()
  (or (|PARSE-Float|) (|PARSE-PrimaryNoFloat|)))

```

defun PARSE-Primary1

[PARSE-VarForm p440]
 [optional p467]
 [current-symbol p460]
 [PARSE-Primary1 p434]
 [must p466]
 [pop-stack-2 p550]
 [pop-stack-1 p550]
 [push-reduction p468]
 [PARSE-Quad p439]

[\[PARSE-String p439\]](#)
[\[PARSE-IntegerTok p438\]](#)
[\[PARSE-FormalParameter p439\]](#)
[\[match-string p454\]](#)
[\[PARSE-Data p442\]](#)
[\[match-advance-string p455\]](#)
[\[PARSE-Expr p426\]](#)
[\[PARSE-Sequence p445\]](#)
[\[PARSE-Enclosure p438\]](#)
[\[\\$boot p??\]](#)

— **defun PARSE-Primary1** —

```

(defun |PARSE-Primary1| ()
  (declare (special $boot))
  (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|)))

```

—————

defun PARSE-Float

[\[PARSE-FloatBase p436\]](#)
[\[must p466\]](#)
[\[PARSE-FloatExponent p437\]](#)
[\[push-reduction p468\]](#)
[\[make-float p??\]](#)
[\[pop-stack-4 p551\]](#)
[\[pop-stack-3 p551\]](#)
[\[pop-stack-2 p550\]](#)
[\[pop-stack-1 p550\]](#)

— **defun PARSE-Float** —

```
(defun |PARSE-Float| ()
  (and (|PARSE-FloatBase|)
    (must (or (and nonblank (|PARSE-FloatExponent|))
      (push-reduction '|PARSE-Float| 0)))
    (push-reduction '|PARSE-Float|
      (make-float (pop-stack-4) (pop-stack-2) (pop-stack-2)
        (pop-stack-1))))))
```

defun PARSE-FloatBase

```
[current-symbol p460]
[char-eq p464]
[current-char p463]
[char-ne p464]
[next-char p463]
[PARSE-IntegerTok p438]
[must p466]
[PARSE-FloatBasePart p436]
[PARSE-IntegerTok p438]
[push-reduction p468]
[PARSE-FloatBase digitp (vol5)]
```

— defun PARSE-FloatBase —

```
(defun |PARSE-FloatBase| ()
  (or (and (integerp (current-symbol)) (char-eq (current-char) ".")
    (char-ne (next-char) ".") (|PARSE-IntegerTok|)
    (must (|PARSE-FloatBasePart|)))
    (and (integerp (current-symbol))
      (char-eq (char-upcase (current-char)) 'e)
      (|PARSE-IntegerTok|) (push-reduction '|PARSE-FloatBase| 0)
      (push-reduction '|PARSE-FloatBase| 0))
    (and (digitp (current-char)) (eq (current-symbol) '|.|)
      (push-reduction '|PARSE-FloatBase| 0)
      (|PARSE-FloatBasePart|))))
```

defun PARSE-FloatBasePart

```
[match-advance-string p455]
[must p466]
```


— defun PARSE-FloatBasePart —

— 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))))

```

defun PARSE-Enclosure

```

[match-advance-string p455]
[must p466]
[PARSE-Expr p426]
[push-reduction p468]
[pop-stack-1 p550]

```

— 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|)))))))

```

defun PARSE-IntegerTok

```

[parse-number p547]

```

— defun PARSE-IntegerTok —

```

(defun |PARSE-IntegerTok| () (parse-number))

```

defun PARSE-FormalParameter

[PARSE-FormalParameterTok p439]

— defun PARSE-FormalParameter —

```
(defun |PARSE-FormalParameter| () (|PARSE-FormalParameterTok|))
```

defun PARSE-FormalParameterTok

[parse-argument-designator p548]

— defun PARSE-FormalParameterTok —

```
(defun |PARSE-FormalParameterTok| () (parse-argument-designator))
```

defun PARSE-Quad

[match-advance-string p455]

[push-reduction p468]

[PARSE-GlyphTok p444]

[\$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| '|.|))))
```

defun PARSE-String

[parse-spadstring p546]

— defun PARSE-String —

```
(defun |PARSE-String| () (parse-spadstring))
```

defun PARSE-VarForm

[PARSE-Name p441]
 [optional p467]
 [PARSE-Scripts p440]
 [push-reduction p468]
 [pop-stack-2 p550]
 [pop-stack-1 p550]

— 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))))
```

defun PARSE-Scripts

[match-advance-string p455]
 [must p466]
 [PARSE-ScriptItem p441]

— defun PARSE-Scripts —

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

defun PARSE-ScriptItem

[PARSE-Expr p426]
 [optional p467]
 [star p467]
 [match-advance-string p455]
 [must p466]
 [PARSE-ScriptItem p441]
 [push-reduction p468]
 [pop-stack-2 p550]
 [pop-stack-1 p550]

— 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)))))))
```

—————

defun PARSE-Name

[parse-identifier p547]
 [push-reduction p468]
 [pop-stack-1 p550]

— defun PARSE-Name —

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

—————

defun PARSE-Data

[action p467]
 [PARSE-Sexpr p442]
 [push-reduction p468]
 [translabel p543]
 [pop-stack-1 p550]
 [labasoc p??]

— defun PARSE-Data —

```
(defun |PARSE-Data| ()
  (declare (special labasoc))
  (and (action (setq labasoc nil)) (|PARSE-Sexpr|)
    (push-reduction '|PARSE-Data|
      (list 'quote (translabel (pop-stack-1) labasoc)))))
```

—————

defun PARSE-Sexpr

[PARSE-Sexpr1 p442]

— defun PARSE-Sexpr —

```
(defun |PARSE-Sexpr| ()
  (and (action (advance-token)) (|PARSE-Sexpr1|)))
```

—————

defun PARSE-Sexpr1

[PARSE-AnyId p444]
 [optional p467]
 [PARSE-NBGlyphTok p443]
 [must p466]
 [PARSE-Sexpr1 p442]
 [action p467]
 [pop-stack-2 p550]
 [nth-stack p551]
 [match-advance-string p455]
 [push-reduction p468]
 [PARSE-IntegerTok p438]
 [pop-stack-1 p550]

[PARSE-String p439]
 [bang p??]
 [star p467]
 [PARSE-GlyphTok p444]

— 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 ")")))))
```

—————

defun PARSE-NBGlyphTok

[match-current-token p459]
 [action p467]
 [advance-token p462]
 [tok p416]

— defun PARSE-NBGlyphTok —

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

defun PARSE-GlyphTok

```
[match-current-token p459]
[action p467]
[advance-token p462]
[tok p416]
```

— defun PARSE-GlyphTok —

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

defun PARSE-AnyId

```
[parse-identifier p547]
[match-string p454]
[push-reduction p468]
[current-symbol p460]
[action p467]
[advance-token p462]
[parse-keyword p548]
```

— defun PARSE-AnyId —

```
(defun |PARSE-AnyId| ()
  (or (parse-identifier)
      (or (and (match-string "$")
                (push-reduction '|PARSE-AnyId| (current-symbol))
                (action (advance-token)))
          (parse-keyword)))))
```

defun PARSE-Sequence

[PARSE-OpenBracket p446]
 [must p466]
 [PARSE-Sequence1 p445]
 [match-advance-string p455]
 [PARSE-OpenBrace p446]
 [push-reduction p468]
 [pop-stack-1 p550]

— defun PARSE-Sequence —

```
(defun |PARSE-Sequence| ()
  (or (and (|PARSE-OpenBracket|) (must (|PARSE-Sequence1|))
           (must (match-advance-string "]")))
      (and (|PARSE-OpenBrace|) (must (|PARSE-Sequence1|))
           (must (match-advance-string "}"))
           (push-reduction '|PARSE-Sequence|
                           (list '|brace| (pop-stack-1))))))
```

—————

defun PARSE-Sequence1

[PARSE-Expression p425]
 [push-reduction p468]
 [pop-stack-2 p550]
 [pop-stack-1 p550]
 [optional p467]
 [PARSE-IteratorTail p447]

— 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))))))))))
```

—————

defun PARSE-OpenBracket

[getToken p458]
 [current-symbol p460]
 [eqcar p??]
 [push-reduction p468]
 [action p467]
 [advance-token p462]

— 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)))))
```

—————

defun PARSE-OpenBrace

[getToken p458]
 [current-symbol p460]
 [eqcar p??]
 [push-reduction p468]
 [action p467]
 [advance-token p462]

— 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)))))
```

—————

defun PARSE-IteratorTail

```
[match-advance-string p455]
[bang p??]
[optional p467]
[star p467]
[PARSE-Iterator p447]
```

— defun PARSE-IteratorTail —

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

—————

defun PARSE-Iterator

```
[match-advance-string p455]
[must p466]
[PARSE-Primary p434]
[PARSE-Expression p425]
[PARSE-Expr p426]
[pop-stack-3 p551]
[pop-stack-2 p550]
[pop-stack-1 p550]
[optional p467]
```

— 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))))))

```

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.

defun PARSE-Suffix

```

[push-reduction p468]
[current-symbol p460]
[action p467]
[advance-token p462]
[optional p467]
[PARSE-TokTail p430]
[pop-stack-1 p550]

```

— 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))))))

```

defun PARSE-SemiColon

```
[match-advance-string p455]
[must p466]
[PARSE-Expr p426]
[push-reduction p468]
[pop-stack-2 p550]
[pop-stack-1 p550]
```

— defun PARSE-SemiColon —

```
(defun |PARSE-SemiColon| ()
  (and (match-advance-string ";")
        (must (or (|PARSE-Expr| 82)
                   (push-reduction '|PARSE-SemiColon| '|/throwAway|))))
        (push-reduction '|PARSE-SemiColon|
                          (list '|;| (pop-stack-2) (pop-stack-1)))))
```

—————

defun PARSE-Return

```
[match-advance-string p455]
[must p466]
[PARSE-Expression p425]
[push-reduction p468]
[pop-stack-1 p550]
```

— defun PARSE-Return —

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

—————

defun PARSE-Exit

```
[match-advance-string p455]
[must p466]
[PARSE-Expression p425]
[push-reduction p468]
[pop-stack-1 p550]
```

— 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)))))
```

—————

defun PARSE-Leave

```
[match-advance-string p455]
[PARSE-Expression p425]
[must p466]
[push-reduction p468]
[PARSE-Label p433]
[pop-stack-1 p550]
```

— 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))))))
```

—————

defun PARSE-Seg

```
[PARSE-GlyphTok p444]
[bang p??]
[optional p467]
[PARSE-Expression p425]
[push-reduction p468]
[pop-stack-2 p550]
```

[pop-stack-1 p550]

— 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)))))
```

—————

defun PARSE-Conditional

[match-advance-string p455]
 [must p466]
 [PARSE-Expression p425]
 [bang p??]
 [optional p467]
 [PARSE-ElseClause p451]
 [push-reduction p468]
 [pop-stack-3 p551]
 [pop-stack-2 p550]
 [pop-stack-1 p550]

— 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)))))
```

—————

defun PARSE-ElseClause

[current-symbol p460]
 [PARSE-Conditional p451]
 [PARSE-Expression p425]

— defun PARSE-ElseClause —

```
(defun |PARSE-ElseClause| ()
  (or (and (eq (current-symbol) '|if|) (|PARSE-Conditional|))
      (|PARSE-Expression|)))
```

—————

defun PARSE-Loop

```
[star p467]
[PARSE-Iterator p447]
[must p466]
[match-advance-string p455]
[PARSE-Expr p426]
[push-reduction p468]
[pop-stack-2 p550]
[pop-stack-1 p550]
```

— defun PARSE-Loop —

```
(defun |PARSE-Loop| ()
  (or (and (star repeater (|PARSE-Iterator|))
          (must (match-advance-string "repeat"))
          (must (|PARSE-Expr| 110))
          (push-reduction '|PARSE-Loop|
                        (cons 'repeat
                            (append (pop-stack-2) (list (pop-stack-1))))))
      (and (match-advance-string "repeat") (must (|PARSE-Expr| 110))
          (push-reduction '|PARSE-Loop|
                        (list 'repeat (pop-stack-1))))))
```

—————

defun PARSE-LabelExpr

```
[PARSE-Label p433]
[must p466]
[PARSE-Expr p426]
[push-reduction p468]
[pop-stack-2 p550]
[pop-stack-1 p550]
```

— 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))))))
```

defun PARSE-FloatTok

```
[parse-number p547]
[push-reduction p468]
[pop-stack-1 p550]
[bfp- p??]
[$boot p??]
```

— defun PARSE-FloatTok —

```
(defun |PARSE-FloatTok| ()
  (declare (special $boot))
  (and (parse-number)
    (push-reduction '|PARSE-FloatTok|
      (if $boot (pop-stack-1) (bfp- (pop-stack-1))))))
```

9.3 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

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).

defun match-string

The match-string function returns length of X if X matches initial segment of inputstream.

[unget-tokens p458]
 [skip-blanks p454]
 [line-past-end-p p635]
 [current-char p463]
 [initial-substring-p p456]
 [subseq p??]
 [\$line p633]
 [line p633]

— 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))))))
```

—————

defun skip-blanks

[current-char p463]
 [token-lookahead-type p455]
 [advance-char p??]

— defun skip-blanks —

```
(defun skip-blanks ()
  (loop (let ((cc (current-char)))
        (if (not cc) (return nil))
        (if (eq (token-lookahead-type cc) 'white)
            (if (not (advance-char)) (return nil))
            (return t)))))
```

— initvars —

```
(defvar Escape-Character #\\ "Superquoting character.")
```

defun token-lookahead-type

[Escape-Character p??]

— defun token-lookahead-type —

```
(defun token-lookahead-type (char)
  "Predicts the kind of token to follow, based on the given initial character."
  (declare (special Escape-Character))
  (cond
    ((not char) 'eof)
    ((or (char= char Escape-Character) (alpha-char-p char)) 'id)
    ((digitp char) 'num)
    ((char= char #\') 'string)
    ((char= char #\[) 'bstring)
    ((member char '(#\Space #\Tab #\Return) :test #'char=) 'white)
    (t 'special-char)))
```

defun match-advance-string

The match-string function returns length of X if X matches initial segment of inputstream. If it is successful, advance inputstream past X. [quote-if-string p456]

[current-token p461]
 [match-string p454]
 [line-current-index p??]
 [line-past-end-p p635]
 [line-current-char p??]
 [\$token p94]
 [\$line p633]

— defun match-advance-string —

```
(defun match-advance-string (x)
```

```

(let ((y (if (>= (length (string x))
                (length (string (quote-if-string (current-token))))
              (match-string x)
              nil))) ; must match at least the current token
  (when y
    (incf (line-current-index current-line) y)
    (if (not (line-past-end-p current-line))
        (setf (line-current-char current-line)
              (elt (line-buffer current-line)
                    (line-current-index current-line)))
        (setf (line-current-char current-line) #\space))
    (setq prior-token
          (make-token :symbol (intern (string x))
                      :type 'identifier
                      :nonblank nonblank))
    t)))

```

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)))

```

defun quote-if-string

[token-type p??]
 [strconc p??]
 [token-symbol p??]
 [underscore p[458](#)]
 [token-nonblank p??]
 [pack p??]
 [escape-keywords p[457](#)]
 [\$boot p??]
 [\$spad p[573](#)]

— 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))))))

```

defun escape-keywords

[\$keywords p??]

— defun escape-keywords —

```

(defun escape-keywords (pname id)
  (declare (special keywords))
  (if (member id keywords)
      (concatenate 'string "-" pname)
      pname))

```

defun isTokenDelimiter

NIL needed below since END_UNIT is not generated by current parser [current-symbol p460]

— defun isTokenDelimiter —

```

(defun |isTokenDelimiter| ()
  (member (current-symbol) '(\ end\_unit nil)))

```

defun underscore

[vector-push p??]

— defun underscore —

```

(defun underscore (string)
  (if (every #'alpha-char-p string)
      string
      (let* ((size (length string))
              (out-string (make-array (* 2 size)
                                      :element-type 'string-char
                                      :fill-pointer 0))
              next-char)
        (dotimes (i size)
          (setq next-char (char string i))
          (unless (alpha-char-p next-char) (vector-push #\_ out-string))
          (vector-push next-char out-string))
        out-string)))

```

Token Handling**defun getToken**

[eqcar p??]

— defun getToken —

```

(defun |getToken| (x)
  (if (eqcar x '|elt|) (third x) x))

```

defun unget-tokens

```

[quote-if-string p456]
[line-current-segment p636]
[strconc p??]
[line-number p??]

```

[token-nonblank p??]
 [line-new-line p636]
 [line-number p??]
 [valid-tokens p95]

— 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?"))))
```

—————

defun match-current-token

This returns the current token if it has EQ type and (optionally) equal symbol. [current-token p461]

[match-token p460]

— defun match-current-token —

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

—————

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)))
```

—————→

defun match-next-token

This returns the next token if it has equal type and (optionally) equal symbol. [next-token p462]
 [match-token p460]

— defun match-next-token —

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

—————→

defun current-symbol

[make-symbol-of p460]
 [current-token p461]

— defun current-symbol —

```
(defun current-symbol ()
  (make-symbol-of (current-token)))
```

—————→

defun make-symbol-of

[\$token p94]

— 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))))
```

—————

defun current-token

This returns the current token getting a new one if necessary. [try-get-token p461]
 [valid-tokens p95]
 [current-token p461]

— defun current-token —

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

—————

defun try-get-token

[get-token p463]
 [valid-tokens p95]

— defun try-get-token —

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

—————

defun next-token

This returns the token after the current token, or NIL if there is none after. [try-get-token p461]

[current-token p461]
 [valid-tokens p95]
 [next-token p462]

— 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)))
```

—————

defun advance-token

This makes the next token be the current token. [current-token p461]

[copy-token p??]
 [try-get-token p461]
 [valid-tokens p95]
 [current-token p461]

— 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))))
```

—————

defvar \$XTokenReader

— initvars —

```
(defvar XTokenReader 'get-meta-token "Name of tokenizing function")
```

—————

defun get-token

[XTokenReader p463]
[XTokenReader p463]

— defun get-token —

```
(defun get-token (token)
  (funcall XTokenReader token))
```

—————

Character handling

defun current-char

This returns the current character of the line, initially blank for an unread line. [line p633]
[current-line p634]

— defun current-char —

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

—————

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 p634]

[line-next-char p635]
 [current-line p634]

— defun next-char —

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

—————

defun char-eq

— defun char-eq —

```
(defun char-eq (x y)
  (char= (character x) (character y)))
```

—————

defun char-ne

— defun char-ne —

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

—————

Error handling

defvar \$meta-error-handler

— initvars —

```
(defvar meta-error-handler 'meta-meta-error-handler)
```

—————

defun meta-syntax-error

[meta-error-handler p464]
 [meta-error-handler p464]

— defun meta-syntax-error —

```
(defun meta-syntax-error (&optional (wanted nil) (parsing nil))
  (declare (special meta-error-handler))
  (funcall meta-error-handler wanted parsing))
```

—————

Floating Point Support**defun floatexpid**

TPDHERE: The use of and in spadreduce is undefined. rewrite this to loop

[floatexpid identp (vol5)]
 [floatexpid pname (vol5)]
 [spadreduce p??]
 [collect p379]
 [step p??]
 [maxindex p??]
 [floatexpid digitp (vol5)]

— defun floatexpid —

```
(defun floatexpid (x &aux s)
  (when (and (identp x) (char= (char-upcase (elt (setq s (pname x)) 0)) #\E)
    (> (length s) 1)
    (spadreduce and 0 (collect (step i 1 1 (maxindex s))
                               (digitp (elt s i))))))
  (read-from-string s t nil :start 1)))
```

—————

Dollar Translation**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))))
```

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.

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)))
```

defmacro must

[meta-syntax-error p465]

— **defmacro must** —

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

defun action

— defun action —

```
(defun action (dothis) (or dothis t))
```

defun optional

— defun optional —

```
(defun optional (dothis) (or dothis t))
```

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 p468]
[pop-stack-1 p550]

— defmacro star —

```
(defmacro star (lab prod)
  '(prog ((oldstacksize (stack-size reduce-stack)))
    (if (not ,prod) (return nil))
  loop
    (if (not ,prod)
      (let* ((newstacksize (stack-size reduce-stack))
             (number-of-new-reductions (- newstacksize oldstacksize)))
        (if (> number-of-new-reductions 0)
          (return (do ((i 0 (1+ i)) (accum nil))
                      ((= i number-of-new-reductions)
                       (push-reduction ',lab accum)
                       (return t))
                    (push (pop-stack-1) accum)))
          (return t)))
```

```
(go loop))))
```

Stacking and retrieving reductions of rules.

defvar \$reduce-stack

Stack of results of reduced productions. [Stack p92]

— **initvars** —

```
(defvar reduce-stack (make-stack) )
```

defmacro reduce-stack-clear

— **defmacro reduce-stack-clear** —

```
(defmacro reduce-stack-clear () '(stack-load nil reduce-stack))
```

defun push-reduction

```
[stack-push p93]
[make-reduction p??]
[reduce-stack p468]
```

— **defun push-reduction** —

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

Chapter 10

Comment Recording

This is the graph of the functions used for recording comments. The syntax is a graphviz dot file. To generate this graph as a JPEG file, type:

```
tangle v9CommentRecording.dot bookvol9.pamphlet >v9cr.dot
dot -Tjpg v9cr.dot >v9cr.jpg
```

— v9CommentRecording.dot —

```
digraph pic {
    fontsize=10;
    bgcolor="#ECEA81";
    node [shape=box, color=white, style=filled];

    "postDef" [color="#FFFFFF"]
    "PARSE-Category" [color="#FFFFFF"]

    "recordAttributeDocumentation" [color="#FF6600"]
    "recordSignatureDocumentation" [color="#FF6600"]
    "recordDocumentation" [color="#2222DD"]
    "collectComBlock" [color="#22EE22"]
    "recordHeaderDocumentation" [color="#FFFF66"]
    "collectAndDeleteAssoc" [color="#FFFF66"]

    "postDef" -> "recordHeaderDocumentation"
    "PARSE-Category" -> "recordSignatureDocumentation"
    "PARSE-Category" -> "recordAttributeDocumentation"

    "recordAttributeDocumentation" -> "recordDocumentation"
    "recordSignatureDocumentation" -> "recordDocumentation"
    "recordDocumentation" -> "recordHeaderDocumentation"
    "recordDocumentation" -> "collectComBlock"
```

```
"collectComBlock" -> "collectAndDeleteAssoc"
}
```

10.1 Comment Recording Layer 0 – API

defun recordSignatureDocumentation

This function is called externally by PARSE-Category. [recordDocumentation p471]
[postTransform p365]

— defun recordSignatureDocumentation —

```
(defun |recordSignatureDocumentation| (opSig lineno)
  (|recordDocumentation| (cdr (postTransform opSig)) lineno))
```

defun recordAttributeDocumentation

This function is called externally by PARSE-Category. [opOf p??]
[pname p??]
[upper-case-p p??]
[recordDocumentation p471]
[ifcdr p??]
[postTransform p365]

— defun recordAttributeDocumentation —

```
(defun |recordAttributeDocumentation| (arg lineno)
  (let (att name)
    (setq att (cadr arg))
    (setq name (|opOf| att))
    (cond
      ((upper-case-p (elt (pname name) 0)) nil)
      (t
       (|recordDocumentation|
        (list name (cons 'attribute (ifcdr (postTransform att)))) lineno))))))
```

10.2 Comment Recording Layer 1

defun recordDocumentation

[recordHeaderDocumentation p472]
 [collectComBlock p471]
 [\$maxSignatureLineNumber p??]
 [\$docList p??]

— defun recordDocumentation —

```
(defun |recordDocumentation| (key lineno)
  (let (u)
    (declare (special |$docList| |$maxSignatureLineNumber|))
    (|recordHeaderDocumentation| lineno)
    (setq u (|collectComBlock| lineno))
    (setq |$maxSignatureLineNumber| lineno)
    (setq |$docList| (cons (cons key u) |$docList|))))
```

—————

10.3 Comment Recording Layer 2

defun collectComBlock

[collectAndDeleteAssoc p472]
 [\$comblocklist p553]

— defun collectComBlock —

```
(defun |collectComBlock| (x)
  (let (val u)
    (declare (special $comblocklist))
    (cond
      ((and (consp $comblocklist)
            (consp (qcar $comblocklist))
            (equal (qcaar $comblocklist) x))
       (setq val (qcdar $comblocklist))
       (setq u (append val (|collectAndDeleteAssoc| x)))
       (setq $comblocklist (cdr $comblocklist))
       u)
      (t (|collectAndDeleteAssoc| x)))))
```

—————

10.4 Comment Recording Layer 3

defun recordHeaderDocumentation

This function is called externally by postDef. [assocright p??]
 [\$maxSignatureLineNumber p??]
 [\$comblocklist p553]
 [\$headerDocumentation p??]
 [\$headerDocumentation p??]
 [\$comblocklist p553]

— defun recordHeaderDocumentation —

```
(defun |recordHeaderDocumentation| (lineno)
  (let (al)
    (declare (special |$headerDocumentation| |$maxSignatureLineNumber|
                     $comblocklist))
    (when (eql |$maxSignatureLineNumber| 0)
      (setq al
        (loop for p in $comblocklist
              when (or (null (car p)) (null lineno) (> lineno (car p)))
              collect p))
      (setq $comblocklist (setdifference $comblocklist al))
      (setq |$headerDocumentation| (assocright al))
      (when |$headerDocumentation| (setq |$maxSignatureLineNumber| 1))
      |$headerDocumentation|)))
```

defun collectAndDeleteAssoc

u is (.. (x . a) .. (x . b) ..) ==> (a b ..)

deleting entries from u assumes that the first element is useless [\$comblocklist p553]

— defun collectAndDeleteAssoc —

```
(defun |collectAndDeleteAssoc| (x)
  (let (r res s)
    (declare (special $comblocklist))
    (maplist
      #'(lambda (y)
        (when (setq s (cdr y))
          (do ()
            ((null (and s (consp (car s)) (equal (qcar (car s)) x))) nil)
            (setq r (qcdr (car s))))
```

```
      (setq res (append res r))
      (setq s (cdr s))
      (rplacd y s)))
$comblocklist)
res))
```

—————

Chapter 11

Category handling

defun getConstructorExports

— defun getConstructorExports —

```
(defun |getConstructorExports| (&rest arg)
  (let (options conform)
    (setq conform (car arg))
    (setq options (cdr arg))
    (|categoryParts| conform
      (getdatabase (|opOf| conform) 'constructorcategory)
      (ifcar options))))
```

—————

Chapter 12

Building libdb.text

defun extendLocalLibdb

```
[buildLibdb p478]
[union p??]
[purgeNewConstructorLines p481]
[dbReadLines p481]
[dbWriteLines p481]
[extendLocalLibdb deleteFile (vol5)]
[msort p??]
[$createLocalLibDb p??]
[$newConstructorList p??]
[$newConstructorList p??]
```

— **defun extendLocalLibdb** —

```
(defun |extendLocalLibdb| (conlist)
  (let (localLibdb oldlines newlines)
    (declare (special |$createLocalLibDb| |$newConstructorList|))
    (cond
      ((null |$createLocalLibDb|) nil)
      ((null conlist) nil)
      (t
       (|buildLibdb| conlist)
       (setq |$newConstructorList| (|union| conlist |$newConstructorList|))
       (setq localLibdb "libdb.text")
       (cond
         ((null (probe-file "libdb.text"))
          (rename-file "temp.text" "libdb.text"))
         (t
          (setq oldlines
                 (|purgeNewConstructorLines| (|dbReadLines| localLibdb) conlist))
```

```
(setq newlines (|dbReadLines| "temp.text"))
(|dbWriteLines| (msort (|union| oldlines newlines)) "libdb.text")
(|deleteFile| "temp.text")))))))
```

defun buildLibdb

This function appears to have two use cases, one in which the domainList variable is undefined, in which case it writes out all of the constructors, and the other case where it writes out a single constructor. Formal for libdb.text:

```
constructors      Cname\#\I\sig \args  \abb \comments (C is C, D, P, X)
operations        Op  \#\E\sig \conname\pred\comments (E is one of U/E)
attributes         Aname\#\E\args\conname\pred\comments
I = <x if exposed><d if category with a default package>
```

```
[dsetq p??]
[ifcar p??]
[buildLibdb deleteFile (vol5)]
[buildLibdb make-outstream (vol5)]
[writedb p??]
[buildLibdbString p480]
[buildLibdb allConstructors (vol5)]
[buildLibdbConEntry p482]
[getConstructorExports p475]
[buildLibOps p484]
[buildLibAttrs p485]
[shut p??]
[obey p??]
[deleteFile p??]
[$outStream p??]
[$conform p??]
[$kind p??]
[$doc p??]
[$exposed? p??]
[$conform p??]
[$conname p??]
[$outStream p??]
[$DefLst p??]
[$PakLst p??]
[$catLst p??]
[$DomLst p??]
[$AttrLst p??]
[$OpLst p??]
```

— defun buildLibdb —

```

(defun |buildLibdb| (&rest G168131 &AUX options)
  (dsetq options G168131)
  (let (|$OpLst| |$AttrLst| |$DomLst| |$CatLst| |$PakLst| |$DefLst|
        |$outStream| |$conname| |$conform| |$exposed?| |$doc|
        |$kind| domainList comments constructorList tmp1 attrlist oplist)
    (declare (special |$OpLst| |$AttrLst| |$DomLst| |$CatLst| |$PakLst|
                      |$DefLst| |$outStream| |$conname| |$conform|
                      |$exposed?| |$doc| |$kind|))
    (setq domainList (ifcar options))
    (setq |$OpLst| nil)
    (setq |$AttrLst| nil)
    (setq |$DomLst| nil)
    (setq |$CatLst| nil)
    (setq |$PakLst| nil)
    (setq |$DefLst| nil)
    (|deleteFile| "temp.text")
    (setq |$outStream| (make-outstream "temp.text"))
    (unless domainList
      (setq comments
        (concatenate 'string
          "\\spad{Union(A,B,...,C)} is a primitive type in AXIOM used to "
          "represent objects of type \\spad{A} or of type \\spad{B} or...or "
          "of type \\spad{C}."))
      (|writedb|
        (|buildLibdbString|
          (list "dUnion" 1 "x" "special" "(A,B,...,C)" 'UNION comments)))
      (setq comments
        (concatenate 'string
          "\\spad{Record(a:A,b:B,...,c:C)} is a primitive type in AXIOM used "
          "to represent composite objects made up of objects of type "
          "\\spad{A}, \\spad{B},..., \\spad{C} which are indexed by \"keys\" "
          "(identifiers) \\spad{a},\\spad{b},...,\\spad{c}."))
      (|writedb|
        (|buildLibdbString|
          (list "dRecord" 1 "x" "special" "(a:A,b:B,...,c:C)" 'RECORD comments)))
      (setq comments
        (concatenate 'string
          "\\spad{Mapping(T,S)} is a primitive type in AXIOM used to represent "
          "mappings from source type \\spad{S} to target type \\spad{T}. "
          "Similarly, \\spad{Mapping(T,A,B)} denotes a mapping from source "
          "type \\spad{(A,B)} to target type \\spad{T}."))
      (|writedb|
        (|buildLibdbString|
          (list "dMapping" 1 "x" "special" "(T,S)" 'MAPPING comments)))
      (setq comments
        (concatenate 'string
          "\\spad{Enumeration(a,b,...,c)} is a primitive type in AXIOM used to "

```

```

    "represent the object composed of the symbols \\spad{a},\\spad{b},"
    "..., and \\spad{c}.")
  (|writedb|
    (|buildLibdbString|
      (list "dEnumeration" 1 "x" "special" "(a,b,...,c)" 'ENUM comments))))
  (setq |$conname| nil)
  (setq |$conform| nil)
  (setq |$exposed?| nil)
  (setq |$doc| nil)
  (setq |$kind| nil)
  (setq constructorList (or domainList (|allConstructors|)))
  (loop for con in constructorList do
    (|writedb| (|buildLibdbConEntry| con))
    (setq tmp1 (|getConstructorExports| |$conform|))
    (setq attrlist (car tmp1))
    (setq oplist (cdr tmp1))
    (|buildLibOps| oplist)
    (|buildLibAttrs| attrlist))
  (shut |$outStream|)
  (unless domainList
    (obey "sort \"temp.text\" > \"libdb.text\"")
    (rename-file "libdb.text" "olibdb.text")
    (|deleteFile| "temp.text"))))

```

defun buildLibdbString

```

[strconc p??]
[stringimage p??]

```

— defun buildLibdbString —

```

(defun |buildLibdbString| (arg)
  (let (x u)
    (setq x (car arg))
    (setq u (cdr arg))
    (strconc (stringimage x)
      (let ((result ""))
        (loop for y in u
          collect (setq result (strconc result (strconc "\"" (stringimage y))))
          result))))

```

defun dbReadLines

```
[eofp p??]
[readline p??]
```

— defun dbReadLines —

```
(defun |dbReadLines| (target)
  (let (instream lines)
    (setq instream (open target))
    (setq lines
      (loop while (not (eofp instream))
        collect (readline instream)))
    (close instream)
    lines))
```

defun purgeNewConstructorLines

```
[screenLocalLine p486]
```

— defun purgeNewConstructorLines —

```
(defun |purgeNewConstructorLines| (lines conlist)
  (loop for x in lines
    when (null (|screenLocalLine| x conlist))
    collect x))
```

defun dbWriteLines

```
[ifcar p??]
[getTempPath p??]
[make-outstream p??]
[writedb p??]
[shut p??]
[$outStream p??]
[$outStream p??]
```

— defun dbWriteLines —

```
(defun |dbWriteLines| (&rest G176369 &aux options s)
```

```

(dsetq (s . options) G176369)
(let (|$outStream| pathname)
(declare (special |$outStream|))
  (setq pathname (or (ifcar options) (|getTempPath| ' |source|)))
  (setq |$outStream| (make-outstream pathname))
  (loop for x in s do (|writedb| x))
  (shut |$outStream|)
  pathname))

```

defun buildLibdbConEntry

```

[getdatabase p??]
[dbMkForm p??]
[msubst p??]
[isExposedConstructor p??]
[pname p??]
[maxindex p??]
[downcase p??]
[lassoc p??]
[libdbTrim p??]
[concatWithBlanks p??]
[form2HtString p??]
[libConstructorSig p??]
[strconc p??]
[buildLibdbString p480]
[length p??]
[$exposed? p??]
[$kind p??]
[$conform p??]
[$kind p??]
[$doc p??]
[$exposed? p??]
[$conname p??]

```

— defun buildLibdbConEntry —

```

(defun |buildLibdbConEntry| (conname)
  (let (abb conform pname kind arg1 tmp1 conComments argpart sigpart header)
    (declare (special |$exposed?| |$doc| |$kind| |$conname| |$conform|))
    (cond
      ((null (getdatabase conname 'constructormodemap)) nil)
      (t
       (setq abb (getdatabase conname 'abbreviation))
       (setq |$conname| conname)

```

```

(setq conform (or (getdatabase conname 'constructorform) (list conname)))
(setq |$conform| (|dbMkForm| (msubst 't 'T$ conform)))
(cond
  ((null |$conform|) nil)
  (t
   (setq |$exposed?| (if (|isExposedConstructor| conname) "x" "n"))
   (setq |$doc| (getdatabase conname 'documentation))
   (setq pname (pname conname))
   (setq kind (getdatabase conname 'constructorkind))
   (cond
     ((and (eq kind '|domain|)
            (progn
              (setq tmp1 (getdatabase conname 'constructormodemap))
              (and (consp tmp1)
                    (consp (qcar tmp1))
                    (consp (qcdar tmp1))))
              (consp (qcadar tmp1)) (eq (qcaadar tmp1) 'category)
              (progn
                (and (consp (qcdadar tmp1))
                      (eq (qcar (qcdadar tmp1)) '|package|))))
             (setq kind '|package|))))
     (t
      (setq |$kind|
        (if (char= (elt pname (maxindex pname)) #\&)
            '|x|
            (downcase (elt (pname kind) 0))))
      (setq arg1 (cdr |$conform|))
      (setq conComments
        (cond
          ((progn
              (setq tmp1 (lassoc '|constructor| |$doc|))
              (and (consp tmp1)
                    (eq (qcdr tmp1) nil)
                    (consp (qcar tmp1))
                    (equal (qcaar tmp1) nil)))
              (|libdbTrim| (|concatWithBlanks| (qcdar tmp1))))
          (t "")))
      (setq argpart (substring (|form2HtString| (cons '|f| arg1)) 1 nil))
      (setq sigpart (|libConstructorSig| |$conform|))
      (setq header (strconc |$kind| (pname conname)))
      (|buildLibdbString|
        (list header (|#| arg1) |$exposed?|
              sigpart argpart abb conComments))))))

```

defun buildLibOps

[buildLibOp p484]

— defun buildLibOps —

```
(defun |buildLibOps| (oplist)
  (loop for item in oplist
        do (|buildLibOp| (car item) (cadr item) (cddr item))))
```

—————

defun buildLibOp

```
[sublislis p??]
[msubst p??]
[form2LispString p??]
[stringimage p??]
[strconc p??]
[libdbTrim p??]
[concatWithBlanks p??]
[lassoc p??]
[checkCommentsForBraces p??]
[writedb p??]
[buildLibdbString p480]
[$kind p??]
[$doc p??]
[$exposed? p??]
[$conform p??]
```

— defun buildLibOp —

```
(defun |buildLibOp| (op sig pred)
  (let (nsig sigpart predString s sop header conform comments)
    (declare (special |$kind| |$doc| |$exposed?| |$conform|))
    (setq nsig (sublislis (cdr |$conform|) |$FormalMapVariableList| sig))
    (setq pred (sublislis (cdr |$conform|) |$FormalMapVariableList| pred))
    (setq nsig (msubst 't 't$ nsig))
    (setq pred (msubst 't 't$ pred))
    (setq sigpart (|form2LispString| (cons '|Mapping| nsig)))
    (setq predString (if (eq pred t) "" (|form2LispString| pred)))
    (setq sop
      (cond
        ((string= (setq s (stringimage op)) "One") "1")
        ((string= s "Zero") "0")
        (t s)))
```



```

(setq header (strconc "o" sop))
(setq conform (strconc |$kind| (|form2LispString| |$conform|)))
(setq comments
  (|libdbTrim| (|concatWithBlanks| (lassoc sig (lassoc op |$doc|)))))
(|checkCommentsForBraces| '|operation| sop sigpart comments)
(|writedb|
  (|buildLibdbString|
    (list header (|#| (cdr sig)) |$exposed?| sigpart
      conform predString comments))))

```

defun buildLibAttrs

[buildLibAttr p485]

— defun buildLibAttrs —

```

(defun |buildLibAttrs| (attrlist)
  (let (name arg1 pred)
    (loop for item in attrlist
      do (|buildLibAttr| (car item) (cadr item) (cddr item)))))

```

defun buildLibAttr

attributes AKname\#\args\conname\pred\comments (K is U or C)

```

[stringimage p??]
[form2LispString p??]
[sublislis p??]
[concatWithBlanks p??]
[lassoc p??]
[checkCommentsForBraces p??]
[writedb p??]
[buildLibdbString p480]
[length p??]
[$conform p??]
[$FormalMapVariableList p266]
[$kind p??]
[$doc p??]
[$exposed? p??]
[$conname p??]

```

— defun buildLibAttr —

```
(defun |buildLibAttr| (name arg1 pred)
  (let (argPart predString header conname comments)
    (declare (special |$kind| |$conname| |$doc| |$conform|
                      |$FormalMapVariableList| |$exposed?|))
    (setq header (strconc "a" (stringimage name)))
    (setq argPart (substring (|form2LispString| (cons '|f| arg1)) 1 nil))
    (setq pred (sublislis (cdr |$conform|) |$FormalMapVariableList| pred))
    (setq predString (if (eq pred t) "" (|form2LispString| pred)))
    (setq header (strconc "a" (stringimage name)))
    (setq conname (strconc |$kind| (|form2LispString| |$conname|)))
    (setq comments
      (|concatWithBlanks| (lassoc (cons '|attribute| arg1) (lassoc name |$doc|))))
    (|checkCommentsForBraces| '|attribute| (stringimage name) arg1 comments)
    (|writedb|
      (|buildLibdbString|
        (list header (|#| arg1) |$exposed?| argPart
              conname predString comments)))))
```

—————

defun screenLocalLine

```
[dbPart p??]
[charPosition p??]
[dbName p??]
[dbKind p??]
```

— defun screenLocalLine —

```
(defun |screenLocalLine| (line conlist)
  (let (s k con)
    (setq k (|dbKind| line))
    (setq con
      (intern (cond ((or (char= k #\o) (char= k #\a))
                      (setq s (|dbPart| line 5 1))
                      (setq k (|charPosition| #\ ( s 1))
                            (substring s 1 (1- k)))
                      (t (|dbName| line))))))
    (member con conlist)))
```

—————

Chapter 13

Comment Syntax Checking

This is the graph of the functions used for comment syntax checking. The syntax is a graphviz dot file. To generate this graph as a JPEG file, type:

```
tangle v9CommentSyntaxChecking.dot bookvol9.pamphlet >v9csc.dot
dot -Tjpg v9csc.dot >v9csc.jpg
```

— v9CommentSyntaxChecking.dot —

```
digraph hierarchy {
  fontsize=10;
  bgcolor="#ECEA81";
  node [shape=box, color=white, style=filled];

  "compileDocumentation" [color="#FFFFFF"]
  "finalizeLisplib"      [color="#FFFFFF"]

  {rank=same; "compileDocumentation" "finalizeLisplib"}

  "checkAddBackSlashes" [color="#FFFF66"]
  "checkAddMacros"      [color="#FFFF66"]
  "checkAddPeriod"      [color="#FFFF66"]
  "checkAddSpaceSegments" [color="#FFFF66"]
  "checkAddSpaces"      [color="#FFFF66"]
  "checkAlphabetic"     [color="#FFFF66"]
  "checkIeEgfun"        [color="#FFFF66"]
  "checkIsValidType"    [color="#FFFF66"]
  "checkLookForLeftBrace" [color="#FFFF66"]
  "checkLookForRightBrace" [color="#FFFF66"]
  "checkNumOfArgs"      [color="#FFFF66"]
  "checkSayBracket"     [color="#FFFF66"]
  "checkSkipBlanks"     [color="#FFFF66"]
}
```

```

"checkSplitBackslash" [color="#FFFF66"]
"checkSplitOn" [color="#FFFF66"]
"checkSplitPunctuation" [color="#FFFF66"]
"firstNonBlankPosition" [color="#FFFF66"]
"getMatchingRightPren" [color="#FFFF66"]
"hasNoVowels" [color="#FFFF66"]
"htcharPosition" [color="#FFFF66"]
"newWordFrom" [color="#FFFF66"]
"removeBackslashes" [color="#FFFF66"]
"whoOwns" [color="#FFFF66"]

{rank=same;
  "checkAddBackSlashes"
  "checkAddMacros"
  "checkAddPeriod"
  "checkAddSpaceSegments"
  "checkAddSpaces"
  "checkAlphabetic"
  "checkIeEgfun"
  "checkIsValidType"
  "checkLookForLeftBrace"
  "checkLookForRightBrace"
  "checkNumOfArgs"
  "checkSayBracket"
  "checkSkipBlanks"
  "checkSplitBackslash"
  "checkSplitOn"
  "checkSplitPunctuation"
  "firstNonBlankPosition"
  "getMatchingRightPren"
  "hasNoVowels"
  "htcharPosition"
  "newWordFrom"
  "removeBackslashes"
  "whoOwns"
}

"checkAddIndented" [color="#22EE22"]
"checkDocMessage" [color="#22EE22"]
"checkExtract" [color="#22EE22"]
"checkGetArgs" [color="#22EE22"]
"checkGetMargin" [color="#22EE22"]
"checkGetParse" [color="#22EE22"]
"checkGetStringBeforeRightBrace" [color="#22EE22"]
"checkIeEg" [color="#22EE22"]
"checkIndentedLines" [color="#22EE22"]
"checkSkipIdentifierToken" [color="#22EE22"]
"checkSkipOpToken" [color="#22EE22"]
"checkSplitBrace" [color="#22EE22"]
"checkTrimCommented" [color="#22EE22"]

```

```

"newString2Words" [color="#22EE22"]

{rank=same;
  "checkAddIndented"
  "checkDocMessage"
  "checkExtract"
  "checkGetArgs"
  "checkGetMargin"
  "checkGetParse"
  "checkGetStringBeforeRightBrace"
  "checkIeEg"
  "checkIndentedLines"
  "checkSkipIdentifierToken"
  "checkSkipOpToken"
  "checkSplitBrace"
  "checkTrimCommented"
  "newString2Words"
}

"checkDocError" [color="#2222DD"]
"checkRemoveComments" [color="#2222DD"]
"checkSkipToken" [color="#2222DD"]
"checkSplit2Words" [color="#2222DD"]

{rank=same;
  "checkDocError"
  "checkRemoveComments"
  "checkSkipToken"
  "checkSplit2Words"
}

"checkBeginEnd" [color="#FF6600"]
"checkDecorate" [color="#FF6600"]
"checkDecorateForHt" [color="#FF6600"]
"checkDocError1" [color="#FF6600"]
"checkFixCommonProblem" [color="#FF6600"]
"checkGetLispFunctionName" [color="#FF6600"]
"checkHTargs" [color="#FF6600"]
"checkRecordHash" [color="#FF6600"]
"checkTexht" [color="#FF6600"]
"checkTransformFirsts" [color="#FF6600"]
"checkTrim" [color="#FF6600"]

{rank=same;
  "checkBeginEnd"
  "checkDecorate"
  "checkDecorateForHt"
  "checkDocError1"
  "checkFixCommonProblem"
  "checkGetLispFunctionName"

```

```

    "checkHTargs"
    "checkRecordHash"
    "checkTexht"
    "checkTransformFirsts"
    "checkTrim"
}

"checkArguments" [color="#0066FF"]
"checkBalance" [color="#0066FF"]

{rank=same;
  "checkArguments"
  "checkBalance"
}

"checkComments" [color="#006600"]
"checkRewrite" [color="#006600"]

{rank=same;
  "checkComments"
  "checkRewrite"
}

"transformAndRecheckComments" [color="#448822"]

"transDoc" [color="#448822"]

"transDocList" [color="#448822"]

"finalizeDocumentation" [color="#448822"]

"checkAddIndented" -> "firstNonBlankPosition"
"checkAddIndented" -> "checkAddSpaceSegments"
"checkArguments" -> "checkHTargs"
"checkBalance" -> "checkBeginEnd"
"checkBalance" -> "checkDocError"
"checkBalance" -> "checkSayBracket"
"checkBeginEnd" -> "checkDocError"
"checkComments" -> "checkGetMargin"
"checkComments" -> "checkTransformFirsts"
"checkComments" -> "checkIndentedLines"
"checkComments" -> "checkGetArgs"
"checkComments" -> "newString2Words"
"checkComments" -> "checkAddSpaces"
"checkComments" -> "checkIeEg"
"checkComments" -> "checkSplit2Words"
"checkComments" -> "checkBalance"
"checkComments" -> "checkArguments"
"checkComments" -> "checkFixCommonProblem"
"checkComments" -> "checkDecorate"

```

```

"checkComments" -> "checkAddPeriod"
"checkDecorate" -> "checkDocError"
"checkDecorate" -> "checkAddBackSlashes"
"checkDecorate" -> "hasNoVowels"
"checkDecorateForHt" -> "checkDocError"
"checkDocError" -> "checkDocMessage"
"checkDocError1" -> "checkDocError"
"checkDocMessage" -> "whoOwns"
"checkExtract" -> "firstNonBlankPosition"
"checkFixCommonProblem" -> "checkDocError"
"checkGetArgs" -> "firstNonBlankPosition"
"checkGetArgs" -> "getMatchingRightPren"
"checkGetLispFunctionName" -> "checkDocError"
"checkGetMargin" -> "firstNonBlankPosition"
"checkGetParse" -> "removeBackslashes"
"checkHTargs" -> "checkLookForLeftBrace"
"checkHTargs" -> "checkLookForRightBrace"
"checkHTargs" -> "checkDocError"
"checkIeEg" -> "checkIeEgfun"
"checkIndentedLines" -> "firstNonBlankPosition"
"checkIndentedLines" -> "checkAddSpaceSegments"
"checkRecordHash" -> "checkLookForLeftBrace"
"checkRecordHash" -> "checkLookForRightBrace"
"checkRecordHash" -> "checkGetLispFunctionName"
"checkRecordHash" -> "checkGetStringBeforeRightBrace"
"checkRecordHash" -> "checkGetParse"
"checkRecordHash" -> "checkDocError"
"checkRecordHash" -> "checkNumOfArgs"
"checkRecordHash" -> "checkIsValidType"
"checkRemoveComments" -> "checkTrimCommented"
"checkRewrite" -> "checkRemoveComments"
"checkRewrite" -> "checkAddIndented"
"checkRewrite" -> "checkGetArgs"
"checkRewrite" -> "newString2Words"
"checkRewrite" -> "checkAddSpaces"
"checkRewrite" -> "checkSplit2Words"
"checkRewrite" -> "checkAddMacros"
"checkRewrite" -> "checkTexht"
"checkRewrite" -> "checkArguments"
"checkRewrite" -> "checkFixCommonProblem"
"checkRewrite" -> "checkRecordHash"
"checkRewrite" -> "checkDecorateForHt"
"checkSkipIdentifierToken" -> "checkAlphabetic"
"checkSkipOpToken" -> "checkAlphabetic"
"checkSkipToken" -> "checkSkipIdentifierToken"
"checkSkipToken" -> "checkSkipOpToken"
"checkSplit2Words" -> "checkSplitBrace"
"checkSplitBrace" -> "checkSplitBackslash"
"checkSplitBrace" -> "checkSplitOn"
"checkSplitBrace" -> "checkSplitPunctuation"

```

```

"checkTexht" -> "checkDocError"
"checkTransformFirst" -> "checkSkipToken"
"checkTransformFirst" -> "checkSkipBlanks"
"checkTransformFirst" -> "getMatchingRightPren"
"checkTransformFirst" -> "checkDocError"
"checkTrim" -> "checkDocError"
"checkTrimCommented" -> "htcharPosition"
"finalizeDocumentation" -> "transDocList"
"newString2Words" -> "newWordFrom"
"transDoc" -> "checkDocError1"
"transDoc" -> "checkTrim"
"transDoc" -> "checkExtract"
"transDoc" -> "transformAndRecheckComments"
"transDocList" -> "transDoc"
"transDocList" -> "checkDocError"
"transDocList" -> "checkDocError1"
"transformAndRecheckComments" -> "checkComments"
"transformAndRecheckComments" -> "checkRewrite"

"compileDocumentation" -> "finalizeDocumentation"
"finalizeLisplib" -> "finalizeDocumentation"
}

```

13.1 Comment Checking Layer 0 – API

defun finalizeDocumentation

```

[bright p??]
[sayMSG p??]
[stringimage p??]
[strconc p??]
[sayKeyedMsg p??]
[form2String p??]
[formatOpSignature p??]
[transDocList p495]
[assocleft p??]
[remdup p??]
[macroExpand p174]
[sublislis p??]
[$e p??]
[$lisplibForm p??]
[$docList p??]
[$op p??]
[$comblocklist p553]

```


[`$FormalMapVariableList` p266]

— `defun finalizeDocumentation` —

```
(defun |finalizeDocumentation| ()
  (labels (
    (fn (x env)
      (declare (special |$lisplibForm| |$FormalMapVariableList|))
      (cond
        ((atom x) (list x nil))
        (t
         (when (> (|#| x) 2) (setq x (take 2 x)))
         (sublislis |$FormalMapVariableList| (cdr |$lisplibForm|)
                    (|macroExpand| x env))))))
    (hn (u)
      ; ((op,sig,doc), ...) --> ((op ((sig doc) ...)) ...)
      (let (opList op1 sig doc)
        (setq opList (remdup (assocleft u)))
        (loop for op in opList
          collect
            (cons op
              (loop for item in u
                do (setq op1 (first item))
                  (setq sig (second item))
                  (setq doc (third item))
                  when (equal op op1)
                    collect
                      (list sig doc))))))
        (let (unusedCommentLineNumbers docList u noHeading attributes
              signatures name bigcnt op s litcnt a n r sig)
          (declare (special |$e| |$lisplibForm| |$docList| |$op| |$comblocklist|))
          (setq unusedCommentLineNumbers
            (loop for x in $comblocklist
              when (cdr x)
                collect x))
          (setq docList (subst '$ '% (|transDocList| |$op| |$docList|) :test #'equal))
          (cond
            ((setq u
              (loop for item in docList
                when (null (cdr item))
                  collect (car item)))
              (loop for y in u
                do
                  (cond
                    ((eq y '|constructor|) (setq noHeading t))
                    ((and (consp y) (consp (qcdr y)) (eq (qcddr y) nil)
                      (consp (qcadr y)) (eq (qcaadr y) '|attribute|))
                     (setq attributes (cons (cons (qcar y) (qcdadr y)) attributes)))
                    (t (setq signatures (cons y signatures))))
                  (setq name (CAR |$lisplibForm|))
```

```

(when (or noHeading signatures attributes unusedCommentLineNumbers)
  (|sayKeyedMsg| 'S2CD0001 nil)
  (setq bigcnt 1)
  (when (or noHeading signatures attributes)
    (|sayKeyedMsg| 'S2CD0002 (list (strconc (stringimage bigcnt) ".") name))
    (setq bigcnt (1+ bigcnt))
    (setq litcnt 1)
    (when noHeading
      (|sayKeyedMsg| 'S2CD0003
        (list (strconc "(" (stringimage litcnt) ")") name))
      (setq litcnt (1+ litcnt)))
    (when signatures
      (|sayKeyedMsg| 'S2CD0004
        (list (strconc "(" (stringimage litcnt) ")"))
        (setq litcnt (1+ litcnt))
        (loop for item in signatures
          do
            (setq op (first item))
            (setq sig (second item))
            (setq s (|formatOpSignature| op sig))
            (|sayMSG|
              (if (atom s)
                (list '|%x9| s)
                (cons '|%x9| s))))))
    (when attributes
      (|sayKeyedMsg| 'S2CD0005
        (list (strconc "(" (stringimage litcnt) ")"))
        (setq litcnt (1+ litcnt))
        (DO ((G166491 attributes
              (CDR G166491))
            (x NIL))
          ((OR (ATOM G166491)
              (PROGN
                (SETQ x (CAR G166491))
                NIL))
           NIL)
         (SEQ (EXIT
              (PROGN
                (setq a (|form2String| x))
                (|sayMSG|
                  (COND
                    ((ATOM a)
                     (CONS '|%x9| (CONS a NIL)))
                    ('T (CONS '|%x9| a))))))))))
    (when unusedCommentLineNumbers
      (|sayKeyedMsg| 'S2CD0006
        (list (strconc (stringimage bigcnt) ".") name))
        (loop for item in unusedCommentLineNumbers
          do
            (setq r (second item))

```

```

      (|sayMSG| (cons " " (append (|bright| n) (list " " r)))))))))
(hn
 (loop for item in docList
  collect (append (fn (car item) |$e|) (cdr item))))))

```

13.2 Comment Checking Layer 1

defun transDocList

```

[sayBrightly p??]
[transDoc p496]
[checkDocError p517]
[checkDocError1 p507]
[$constructorName p??]

```

— defun transDocList —

```

(defun |transDocList| (|$constructorName| doclist)
  (declare (special |$constructorName|))
  (let (commentList conEntry acc)
    (|sayBrightly|
     (list " Processing " |$constructorName| " for Browser database:"))
    (setq commentList (|transDoc| |$constructorName| doclist))
    (setq acc nil)
    (loop for entry in commentList
      do
        (cond
         ((and (consp entry) (eq (qcar entry) '|constructor|)
              (consp (qcdr entry)) (eq (qcddr entry) nil))
          (if conEntry
              (|checkDocError| (list "Spurious comments: " (qcadr entry)))
              (setq conEntry entry)))
         (t (setq acc (cons entry acc)))))
    (if conEntry
        (cons conEntry acc)
        (progn
         (|checkDocError1| (list "Missing Description"))
         acc))))

```

13.3 Comment Checking Layer 2

defun transDoc

```
[checkDocError1 p507]
[checkTrim p516]
[checkExtract p520]
[transformAndRecheckComments p497]
[nreverse p??]
[$x p??]
[$attribute? p??]
[$x p??]
[$attribute? p??]
[$argl p??]
```

— defun transDoc —

```
(defun |transDoc| (conname doclist)
  (declare (ignore conname))
  (let (|$x| |$attribute?| |$argl| rlist lines u v longline acc)
    (declare (special |$x| |$attribute?| |$argl|))
    (setq |$x| nil)
    (setq rlist (reverse doclist))
    (loop for item in rlist
      do
        (setq |$x| (car item))
        (setq lines (cdr item))
        (setq |$attribute?|
          (and (consp |$x|) (consp (qcdr |$x|)) (eq (qcddr |$x|) nil)
              (consp (qcadr |$x|)) (eq (qcdadr |$x|) nil)
              (eq (qcaadr |$x|) '|attribute|))))
    (cond
      ((null lines)
        (unless |$attribute?| (|checkDocError1| (list "Not documented!!!!"))))
      (t
        (setq u
          (|checkTrim| |$x|
            (cond
              ((stringp lines) (list lines))
              ((eq |$x| '|constructor|) (car lines))
              (t lines))))
          (setq |$argl| nil) ;; possibly unused -- tpd
          (setq longline
            (cond
              ((eq |$x| '|constructor|)
                (setq v
                  (or
                    (|checkExtract| "Description:" u)
```

```

      (and u (|checkExtract| "Description:"
        (cons (strconc "Description: " (car u)) (cdr u))))))
    (|transformAndRecheckComments| ' |constructor| (or v u))
    (t (|transformAndRecheckComments| |$x| u)))
    (setq acc (cons (list |$x| longline) acc))))
  (nreverse acc)))

```

13.4 Comment Checking Layer 3

defun transformAndRecheckComments

```

[sayBrightly p??]
[checkComments p498]
[checkRewrite p499]
[$exposeFlagHeading p??]
[$checkingXmptex? p??]
[$x p??]
[$name p??]
[$origin p??]
[$recheckingFlag p??]
[$exposeFlagHeading p??]

```

— defun transformAndRecheckComments —

```

(defun |transformAndRecheckComments| (name lines)
  (let (|$x| |$name| |$origin| |$recheckingFlag| |$exposeFlagHeading| u)
    (declare (special |$x| |$name| |$origin| |$recheckingFlag|
      |$exposeFlagHeading| |$exposeFlag| |$checkingXmptex?|))
    (setq |$checkingXmptex?| nil)
    (setq |$x| name)
    (setq |$name| '|GlossaryPage|)
    (setq |$origin| '|gloss|)
    (setq |$recheckingFlag| nil)
    (setq |$exposeFlagHeading| (list "-----" name "-----"))
    (unless |$exposeFlag| (|sayBrightly| |$exposeFlagHeading|))
    (setq u (|checkComments| name lines))
    (setq |$recheckingFlag| t)
    (|checkRewrite| name (list u))
    (setq |$recheckingFlag| nil)
    u))

```

13.5 Comment Checking Layer 4

defun checkComments

```
[checkGetMargin p522]
[checkTransformFirsts p513]
[checkIndentedLines p524]
[checkGetArgs p521]
[newString2Words p527]
[checkAddSpaces p530]
[checkIeEg p523]
[checkSplit2Words p518]
[checkBalance p501]
[checkArguments p501]
[checkFixCommonProblems p??]
[checkDecorate p504]
[strconc p??]
[checkAddPeriod p529]
[pp p??]
[$attribute? p??]
[$checkErrorFlag p??]
[$argl p??]
[$checkErrorFlag p??]
```

— defun checkComments —

```
(defun |checkComments| (nameSig lines)
  (let (|$checkErrorFlag| margin w verbatim u2 okBefore u v res)
    (declare (special |$checkErrorFlag| |$argl| |$attribute?|))
    (setq |$checkErrorFlag| nil)
    (setq margin (|checkGetMargin| lines))
    (cond
      ((and (or (null (boundp '|$attribute?|)) (null |$attribute?|))
            (not (eq nameSig '|constructor|))))
      (setq lines
        (cons
          (|checkTransformFirsts| (car nameSig) (car lines) margin)
          (cdr lines))))))
    (setq u (|checkIndentedLines| lines margin))
    (setq |$argl| (|checkGetArgs| (car u)))
    (setq u2 nil)
    (setq verbatim nil)
    (loop for x in u
      do (setq w (|newString2Words| x))
        (cond
          (verbatim
            (cond
```

```

      ((and w (equal (car w) "\\end{verbatim}"))
       (setq verbatim nil)
       (setq u2 (append u2 w)))
      (t
       (setq u2 (append u2 (list x))))))
      ((and w (equal (car w) "\\begin{verbatim}"))
       (setq verbatim t)
       (setq u2 (append u2 w)))
      (t (setq u2 (append u2 w)))))
(setq u u2)
(setq u (|checkAddSpaces| u))
(setq u (|checkIeEg| u))
(setq u (|checkSplit2Words| u))
(|checkBalance| u)
(setq okBefore (null |$checkErrorFlag|))
(|checkArguments| u)
(when |$checkErrorFlag| (setq u (|checkFixCommonProblem| u)))
(setq v (|checkDecorate| u))
(setq res
  (let ((result ""))
    (loop for y in v
      do (setq result (strconc result y)))
    result))
(setq res (|checkAddPeriod| res))
(when |$checkErrorFlag| (lpp| res))
res))

```

defun checkRewrite

```

[checkRemoveComments p518]
[checkAddIndented p519]
[checkGetArgs p521]
[newString2Words p527]
[checkAddSpaces p530]
[checkSplit2Words p518]
[checkAddMacros p528]
[checkTexht p512]
[checkArguments p501]
[checkFixCommonProblem p508]
[checkRecordHash p509]
[checkDecorateForHt p506]
[$checkErrorFlag p??]
[$argl p??]
[$checkingXmptex? p??]

```

— defun checkRewrite —

```
(defun |checkRewrite| (name lines)
  (declare (ignore name))
  (prog (|$checkErrorFlag| margin w verbatim u2 okBefore u)
    (declare (special |$checkErrorFlag| |$argl| |$checkingXmptex?|))
    (setq |$checkErrorFlag| t)
    (setq margin 0)
    (setq lines (|checkRemoveComments| lines))
    (setq u lines)
    (when |$checkingXmptex?|
      (setq u
        (loop for x in u
          collect (|checkAddIndented| x margin))))
    (setq |$argl| (|checkGetArgs| (car u)))
    (setq u2 nil)
    (setq verbatim nil)
    (loop for x in u
      do
        (setq w (|newString2Words| x))
        (cond
          (verbatim
            (cond
              ((and w (equal (car w) "\\end{verbatim}"))
                (setq verbatim nil)
                (setq u2 (append u2 w)))
              (t
                (setq u2 (append u2 (list x))))))
            ((and w (equal (car w) "\\begin{verbatim}"))
              (setq verbatim t)
              (setq u2 (append u2 w)))
              (t (setq u2 (append u2 w))))))
        (setq u u2)
        (setq u (|checkAddSpaces| u))
        (setq u (|checkSplit2Words| u))
        (setq u (|checkAddMacros| u))
        (setq u (|checkTexht| u))
        (setq okBefore (null |$checkErrorFlag|))
        (|checkArguments| u)
        (when |$checkErrorFlag| (setq u (|checkFixCommonProblem| u)))
        (|checkRecordHash| u)
        (|checkDecorateForHt| u)))
```

13.6 Comment Checking Layer 5

defun checkArguments

```
[hget p??]
[checkHTargs p509]
[$htMacroTable p??]
```

— defun checkArguments —

```
(defun |checkArguments| (u)
  (let (x k)
    (declare (special |$htMacroTable|))
    (loop while u
      do (setq x (car u))
        (cond
          ((null (setq k (hget |$htMacroTable| x))) '|skip|)
          ((eql k 0) '|skip|)
          ((> k 0) (|checkHTargs| x (cdr u) k nil))
          (t (|checkHTargs| x (cdr u) (- k) t)))
        (pop u))
      u))
```

—————

defun checkBalance

```
[checkBeginEnd p502]
[assoc p??]
[rassoc p??]
[checkDocError p517]
[checkSayBracket p534]
[nreverse p??]
[$checkPrenAlist p??]
```

— defun checkBalance —

```
(defun |checkBalance| (u)
  (let (x openClose open top restStack stack)
    (declare (special |$checkPrenAlist|))
    (|checkBeginEnd| u)
    (setq stack nil)
    (loop while u
      do
        (setq x (car u))
        (cond
```

```

((setq openClose (|assoc| x |$checkPrenAlist|))
 (setq stack (cons (car openClose) stack)))
((setq open (|rassoc| x |$checkPrenAlist|))
 (cond
  ((consp stack)
   (setq top (qcar stack))
   (setq restStack (qcdr stack))
   (when (not (eq open top))
    (|checkDocError|
     (list "Mismatch: left " (|checkSayBracket| top)
           " matches right " (|checkSayBracket| open))))
   (setq stack restStack))
  (t
   (|checkDocError|
    (list "Missing left " (|checkSayBracket| open))))))
(pop u))
(when stack
 (loop for x in (nreverse stack)
  do
   (|checkDocError| (list "Missing right " (|checkSayBracket| x)))))
u))

```

13.7 Comment Checking Layer 6

defun checkBeginEnd

```

[length p??]
[hget p??]
[ifcar p??]
[ifcdr p??]
[substring? p??]
[checkDocError p517]
[member p??]
[$charRbrace p??]
[$charLbrace p??]
[$beginEndList p??]
[$htMacroTable p??]
[$charBack p??]

```

— defun checkBeginEnd —

```

(defun |checkBeginEnd| (u)
 (let (x y beginEndStack)
  (declare (special |$charRbrace| |$charLbrace| |$beginEndList| |$charBack|

```

```

(|$htMacroTable|))
(loop while u
  do
    (setq x (car u))
    (cond
      ((and (stringp x) (equal (elt x 0) |$charBack|) (> (|#| x) 2)
        (null (hget |$htMacroTable| x)) (null (equal x "\\spadignore"))
        (equal (ifcar (ifcdr u)) |$charLbrace|)
        (null (or (|substring?| "\\radiobox" x 0)
          (|substring?| "\\inputbox" x 0))))
        (|checkDocError| (list '|Unexpected HT command: | x)))
      ((equal x "\\beginitems")
        (setq beginEndStack (cons '|items| beginEndStack)))
      ((equal x "\\begin")
        (cond
          ((and (consp u) (consp (qcdr u)) (equal (qcar (qcdr u)) |$charLbrace|)
            (consp (qcddr u)) (equal (car (qcddr u)) |$charRbrace|))
            (setq y (qcaddr u))
            (cond
              ((null (|member| y |$beginEndList|))
                (|checkDocError| (list "Unknown begin type: \\begin{" y "}"))))
              (setq beginEndStack (cons y beginEndStack))
              (setq u (qcddr u)))
            (t (|checkDocError| (list "Improper \\begin command")))))
          ((equal x "\\item")
            (cond
              ((|member| (ifcar beginEndStack) '("items" "menu")) nil)
              (null beginEndStack)
              (|checkDocError| (list "\\item appears outside a \\begin-\\end")))
            (t
              (|checkDocError|
                (list "\\item appears within a \\begin{"
                  (ifcar beginEndStack) "}"..")))))
          ((equal x "\\end")
            (cond
              ((and (consp u) (consp (qcdr u)) (equal (qcar (qcdr u)) |$charLbrace|)
                (consp (qcddr u)) (equal (car (qcddr u)) |$charRbrace|))
                (setq y (qcaddr u))
                (cond
                  ((equal y (ifcar beginEndStack))
                    (setq beginEndStack (cdr beginEndStack))
                    (setq u (qcddr u)))
                  (t
                    (|checkDocError|
                      (list "Trying to match \\begin{" (ifcar beginEndStack)
                        "} with \\end{" y "}")))))
                (t
                  (|checkDocError| (list "Improper \\end command")))))
            (pop u))
          (cond

```

```
(beginEndStack
  (|checkDocError| (list "Missing \\end{" (car beginEndStack) "}"))))
(t '|ok|)))
```

defun checkDecorate

```
[checkDocError p517]
[member p??]
[checkAddBackSlashes p527]
[hasNoVowels p539]
[$checkingXmptex? p??]
[$charExclusions p??]
[$argl p??]
[$charBack p??]
[$charRbrace p??]
[$charLbrace p??]
```

— defun checkDecorate —

```
(defun |checkDecorate| (u)
  (let (x count mathSymbolsOk spadflag verbatim v xcount acc)
    (declare (special |$charLbrace| |$charRbrace| |$charBack| |$argl|
                      |$charExclusions| |$checkingXmptex?|))
    (setq count 0)
    (loop while u
      do
        (setq x (car u))
        (cond
          ((null verbatim)
           (cond
             ((string= x "\\em")
              (cond
                ((> count 0)
                 (setq mathSymbolsOk (1- count))
                 (setq spadflag (1- count))))
              (t
               (|checkDocError| (list "\\em must be enclosed in braces")))))
           (when (|member| x '("\\spadpaste" "\\spad" "\\spadop"))
             (setq mathSymbolsOk count))
           (cond
             ((|member| x '("\\s" "\\spadtype" "\\spadsys" "\\example" "\\andexample"
                           "\\spadop" "\\spad" "\\spadignore" "\\spadpaste"
                           "\\spadcommand" "\\footnote"))
              (setq spadflag count))
             ((equal x |$charLbrace|
```

```

    (setq count (1+ count)))
  ((equal x |$charRbrace|)
   (setq count (1- count))
   (when (eql mathSymbolsOk count) (setq mathSymbolsOk nil))
   (when (eql spadflag count) (setq spadflag nil)))
  ((and (null mathSymbolsOk)
        (|member| x '("|+" "*" "=" "==" "->"))
        (when |$checkingXmptex?|
          (|checkDocError|
           (list '|Symbol| x " appearing outside \\spad{ }"))))))))
(setq acc
 (cond
  ((string= x "\\end{verbatim}")
   (setq verbatim nil)
   (cons x acc))
  (verbatim (cons x acc))
  ((string= x "\\begin{verbatim}")
   (setq verbatim t)
   (cons x acc))
  ((and (string= x "\\begin")
        (equal (car (setq v (ifcdr u))) |$charLbrace|)
        (string= (car (setq v (ifcdr v))) "detail")
        (equal (car (setq v (ifcdr v))) |$charRbrace|))
   (setq u v)
   (cons "\\blankline " acc))
  ((and (string= x "\\end")
        (equal (car (setq v (ifcdr u))) |$charLbrace|)
        (string= (car (setq v (ifcdr v))) "detail")
        (equal (car (setq v (ifcdr v))) |$charRbrace|))
   (setq u v)
   acc)
  ((or (char= x #\$) (string= x "$"))
   (cons "\\$" acc))
  ((or (char= x #%) (string= x "%"))
   (cons "\\%" acc))
  ((or (char= x #\,) (string= x ","))
   (cons ",{" acc))
  ((string= x "\\spad")
   (cons "\\spad" acc))
  ((and (stringp x) (digitp (elt x 0)))
   (cons x acc))
  ((and (null spadflag)
        (or (and (charp x)
                  (alpha-char-p x)
                  (null (member x |$charExclusions|)))
            (|member| x |$argl|)))
   (cons |$charRbrace| (cons x (cons |$charLbrace| (cons "\\spad" acc)))))
  ((and (null spadflag)
        (or (and (stringp x)
                  (null (equal (elt x 0) |$charBack|)))
            (|member| x |$argl|)))
   (cons |$charRbrace| (cons x (cons |$charLbrace| (cons "\\spad" acc)))))
  ))

```

```

        (digitp (elt x (maxindex x))))
      (|member| x '("true" "false"))))
    (cons |$charRbrace| (cons x (cons |$charLbrace| (cons "\\spad" acc)))))
  (t
   (setq xcount (|#| x))
   (cond
    ((and (eql xcount 3)
          (char= (elt x 1) #\t)
          (char= (elt x 2) #\h))
     (cons "th" (cons |$charRbrace|
                      (cons (elt x 0) (cons |$charLbrace| (cons "\\spad" acc)))))))
    ((and (eql xcount 4)
          (char= (elt x 1) #\-)
          (char= (elt x 2) #\t)
          (char= (elt x 3) #\h))
     (cons "-th" (cons |$charRbrace|
                       (cons (elt x 0) (cons |$charLbrace| (cons "\\spad" acc)))))))
    ((or (and (eql xcount 2)
              (char= (elt x 1) #\i))
         (and (null spadflag)
              (> xcount 0)
              (> 4 xcount)
              (null (|member| x '("th" "rd" "st")))
              (|hasNoVowels| x)))
         (cons |$charRbrace|
               (cons x (cons |$charLbrace| (cons "\\spad" acc)))))
     (t
      (cons (|checkAddBackSlashes| x) acc))))))
  (setq u (cdr u)))
(nreverse acc)))

```

defun checkDecorateForHt

```

[checkDocError p517]
[member p??]
[$checkingXmptex? p??]
[$charRbrace p??]
[$charLbrace p??]

```

— defun checkDecorateForHt —

```

(defun |checkDecorateForHt| (u)
  (let (x count spadflag)
    (declare (special |$checkingXmptex?| |$charRbrace| |$charLbrace|))
    (setq count 0)

```

```

(setq spadflag nil)
(loop while u
  do
    (setq x (car u))
    (when (equal x "\\em")
      (if (> count 0)
        (setq spadflag (1- count))
        (|checkDocError| (list "\\em must be enclosed in braces"))))
    (cond
      ((|member| x '("\\s" "\\spadop" "\\spadtype" "\\spad" "\\spadpaste"
        "\\spadcommand" "\\footnote"))
        (setq spadflag count))
      ((equal x |$charLbrace|)
        (setq count (1+ count)))
      ((equal x |$charRbrace|)
        (setq count (1- count))
        (when (equal spadflag count) (setq spadflag nil))))
      ((and (null spadflag) (|member| x '("+ " *" "=" "==" "->"))
        (when |$checkingXmptex?|
          (|checkDocError| (list '|Symbol| x " appearing outside \\spad{}"))))
        (t nil))
      (when (or (equal x "$") (equal x "%"))
        (|checkDocError| (list "Unescaped " x)))
      (pop u))
  u))

```

defun checkDocError1

[checkDocError p517]

[\$compileDocumentation p165]

— defun checkDocError1 —

```

(defun |checkDocError1| (u)
  (declare (special |$compileDocumentation|))
  (if (and (boundp '|$compileDocumentation|) |$compileDocumentation|)
    nil
    (|checkDocError| u)))

```

defun checkFixCommonProblem

```
[member p??]
[ifcar p??]
[ifcdr p??]
[checkDocError p517]
[$charLbrace p??]
[$HTspadmacros p??]
```

— **defun checkFixCommonProblem** —

```
(defun |checkFixCommonProblem| (u)
  (let (x next acc)
    (declare (special |$charLbrace| |$HTspadmacros|))
    (loop while u
      do
        (setq x (car u))
        (cond
          ((and (equal x |$charLbrace|)
                (|member| (setq next (ifcar (cdr u))) |$HTspadmacros|)
                (not (equal (ifcar (ifcdr (cdr u))) |$charLbrace|)))
            (|checkDocError| (list "Reversing " next " and left brace"))
            (setq acc (cons |$charLbrace| (cons next acc)))
            (setq u (cddr u)))
          (t
            (setq acc (cons x acc))
            (setq u (cdr u))))))
    (nreverse acc)))
```

—————

defun checkGetLispFunctionName

```
[charPosition p??]
[checkDocError p517]
```

— **defun checkGetLispFunctionName** —

```
(defun |checkGetLispFunctionName| (s)
  (let (n k j)
    (setq n (|#| s))
    (cond
      ((and (setq k (|charPosition| #\| s 1))
            (> n k)
            (setq j (|charPosition| #\| s (1+ k)))
            (> n j))
        (substring s (1+ k) (1- (- j k)))))
```



```
(t
  (|checkDocError| (cons "Ill-formed lisp expression : " (list s)))
  '|illformed|)))
```

defun checkHTargs

Note that u should start with an open brace. [checkLookForLeftBrace p533]
 [checkLookForRightBrace p533]
 [checkDocError p517]
 [checkHTargs p509]
 [ifcdr p??]

— defun checkHTargs —

```
(defun |checkHTargs| (keyword u nargs inteerValue?)
  (cond
    ((eq1 nargs 0) '|ok|)
    ((null (setq u (|checkLookForLeftBrace| u)))
     (|checkDocError| (list "Missing argument for " keyword)))
    ((null (setq u (|checkLookForRightBrace| (ifcdr u))))
     (|checkDocError| (list "Missing right brace for " keyword)))
    (t
     (|checkHTargs| keyword (cdr u) (1- nargs) inteerValue?))))
```

defun checkRecordHash

```
[member p??]
[checkLookForLeftBrace p533]
[checkLookForRightBrace p533]
[ifcdr p??]
[intern p??]
[hget p??]
[hput p??]
[checkGetLispFunctionName p508]
[checkGetStringBeforeRightBrace p523]
[checkGetParse p522]
[checkDocError p517]
[opOf p??]
[spadSysChoose p??]
[checkNumOfArgs p534]
```

```

[checkIsValidType p532]
[form2HtString p??]
[getl p??]
[$charBack p??]
[$HTlinks p??]
[$htHash p??]
[$HTlisplinks p??]
[$lispHash p??]
[$glossHash p??]
[$currentSysList p??]
[$setOptions p??]
[$sysHash p??]
[$name p??]
[$origin p??]
[$sysHash p??]
[$glossHash p??]
[$lispHash p??]
[$htHash p??]

```

— defun checkRecordHash —

```

(defun |checkRecordHash| (u)
  (let (p q htname entry s parse n key x)
    (declare (special |$origin| |$name| |$sysHash| |$setOptions| |$glossHash|
                      |$currentSysList| |$lispHash| |$HTlisplinks| |$htHash|
                      |$HTlinks| |$charBack|))
    (loop while u
      do
        (setq x (car u))
        (when (and (stringp x) (equal (elt x 0) |$charBack|))
          (cond
            ((and (|member| x |$HTlinks|)
                  (setq u (|checkLookForLeftBrace| (ifcdr u)))
                  (setq u (|checkLookForRightBrace| (ifcdr u)))
                  (setq u (|checkLookForLeftBrace| (ifcdr u)))
                  (setq u (ifcdr u)))
              (setq htname (|intern| (ifcar u)))
              (setq entry (or (hget |$htHash| htname) (list nil)))
              (hput |$htHash| htname
                (cons (car entry) (cons (cons |$name| |$origin|) (cdr entry)))))
            ((and (|member| x |$HTlisplinks|)
                  (setq u (|checkLookForLeftBrace| (ifcdr u)))
                  (setq u (|checkLookForRightBrace| (ifcdr u)))
                  (setq u (|checkLookForLeftBrace| (ifcdr u)))
                  (setq u (ifcdr u)))
              (setq htname
                (|intern|
                 (|checkGetLispFunctionName|

```

```

        (|checkGetStringBeforeRightBrace| u))))
    (setq entry (or (hget |$lispHash| htname) (list nil)))
    (hput |$lispHash| htname
      (cons (car entry) (cons (cons |$name| |$origin|) (cdr entry)))))
  ((and (or (setq p (|member| x '("\gloss" "\spadglos")))
    (setq q (|member| x '("\glossSee" "\spadglosSee"))))
    (setq u (|checkLookForLeftBrace| (ifcdr u)))
    (setq u (ifcdr u)))
    (when q
      (setq u (|checkLookForRightBrace| u))
      (setq u (|checkLookForLeftBrace| (ifcdr u)))
      (setq u (ifcdr u)))
    (setq htname (|intern| (|checkGetStringBeforeRightBrace| u)))
    (setq entry
      (or (hget |$glossHash| htname) (list nil)))
      (hput |$glossHash| htname
        (cons (car entry) (cons (cons |$name| |$origin|) (cdr entry)))))
  ((and (boot-equal x "\spadsys")
    (setq u (|checkLookForLeftBrace| (ifcdr u)))
    (setq u (ifcdr u)))
    (setq s (|checkGetStringBeforeRightBrace| u))
    (when (char= (elt s 0) #\)) (setq s (substring s 1 nil)))
    (setq parse (|checkGetParse| s))
    (cond
      ((null parse)
        (|checkDocError| (list "Unparseable \spadtype: " s)))
      ((null (|member| (|lopOf| parse) |$currentSysList|))
        (|checkDocError| (list "Bad system command: " s)))
      ((or (atom parse)
        (null (and (consp parse) (eq (qcar parse) '|set|)
          (consp (qcdr parse))
          (eq (qcddr parse) nil))))
        '|ok|)
      ((null (|spadSysChoose| |$setOptions| (qcadr parse)))
        (progn
          (|checkDocError| (list "Incorrect \spadsys: " s))
          (setq entry (or (hget |$sysHash| htname) (list nil)))
          (hput |$sysHash| htname
            (cons (car entry) (cons (cons |$name| |$origin|) (cdr entry))))))
    ((and (boot-equal x "\spadtype")
      (setq u (|checkLookForLeftBrace| (ifcdr u)))
      (setq u (ifcdr u)))
      (setq s (|checkGetStringBeforeRightBrace| u))
      (setq parse (|checkGetParse| s))
      (cond
        ((null parse)
          (|checkDocError| (list "Unparseable \spadtype: " s)))
        (t
          (setq n (|checkNumOfArgs| parse))
          (cond

```

```

((null n)
  (|checkDocError| (list "Unknown \\spadtype: " s)))
((and (atom parse) (> n 0))
  '|skip|)
((null (setq key (|checkIsValidType| parse)))
  (|checkDocError| (list "Unknown \\spadtype: " s)))
((atom key) '|ok|)
(t
  (|checkDocError|
    (list "Wrong number of arguments: " (|form2HtString| key))))))
((and (|member| x '("\\spadop" "\\keyword"))
  (setq u (|checkLookForLeftBrace| (ifcdr u)))
  (setq u (ifcdr u)))
  (setq x (|intern| (|checkGetStringBeforeRightBrace| u)))
  (when (null (or (getl x '|Lcd|) (getl x '|Nud|)))
    (|checkDocError| (list "Unknown \\spadop: " x))))))
(pop u))
'|done|))

```

defun checkTexht

```

[ifcdr p??]
[ifcar p??]
[checkDocError p517]
[$charRbrace p??]
[$charLbrace p??]

```

— defun checkTexht —

```

(defun |checkTexht| (u)
  (let (count y x acc)
    (declare (special |$charRbrace| |$charLbrace|))
    (setq count 0)
    (loop while u
      do
        (setq x (car u))
        (when (and (string= x "\\texht") (setq u (ifcdr u)))
          (unless (equal (ifcar u) |$charLbrace|)
            (|checkDocError| "First left brace after \\texht missing"))
          ; drop first argument including braces of \texht
          (setq count 1)
          (loop while
            (or (not (equal (setq y (ifcar (setq u (cdr u)))) |$charRbrace|))
              (> count 1))
          do

```

```

      (when (equal y |$charLbrace|) (setq count (1+ count)))
      (when (equal y |$charRbrace|) (setq count (1- count))))
; drop first right brace of 1st arg
(setq x (ifcar (setq u (cdr u))))
(when (and (string= x "\\httex")
          (setq u (ifcdr u))
          (equal (ifcar u) |$charLbrace|))
; left brace: add it
(setq acc (cons (ifcar u) acc))
(loop while
  (not (equal (setq y (ifcar (setq u (cdr u)))) |$charRbrace|))
  do (setq acc (cons y acc)))
; right brace: add it
(setq acc (cons (ifcar u) acc))
; left brace: forget it
(setq x (ifcar (setq u (cdr u))))
(loop while (not (equal (ifcar (setq u (cdr u))) |$charRbrace|))
  do 'skip|)
; forget right brace: move to next char
(setq x (ifcar (setq u (cdr u))))
(setq acc (cons x acc))
(setq u (cdr u)))
(nreverse acc)))

```

defun checkTransformFirsts

```

[pname p??]
[leftTrim p??]
[fillerSpaces p??]
[checkTransformFirsts p513]
[maxindex p??]
[checkSkipToken p518]
[checkSkipBlanks p534]
[getMatchingRightPren p538]
[checkDocError p517]
[strconc p??]
[getl p??]
[lassoc p??]
[$checkPrenAlist p??]
[$charBack p??]

```

— defun checkTransformFirsts —

```
(defun |checkTransformFirsts| (opname u margin)
```

```

(prog (namestring s m infixOp p open close z n i prefixOp j k firstWord)
(declare (special |$checkPrenAlist| |$charBack|))
(return
  (progn
; case 1: \spad{...
; case 2: form(args)
    (setq namestring (pname opname))
    (cond
      ((equal namestring "Zero") (setq namestring "0"))
      ((equal namestring "One") (setq namestring "1"))
      (t nil))
    (cond
      ((> margin 0)
        (setq s (|leftTrim| u))
        (strconc (|fillerSpaces| margin) (|checkTransformFirsts| opname s 0)))
      (t
        (setq m (maxindex u))
        (cond
          ((> 2 m) u)
          ((equal (elt u 0) |$charBack|) u)
          ((alpha-char-p (elt u 0))
            (setq i (or (|checkSkipToken| u 0 m) (return u)))
            (setq j (or (|checkSkipBlanks| u i m) (return u)))
            (setq open (elt u j))
            (cond
              ((or (and (equal open #\[) (setq close #\]))
                    (and (equal open #\() (setq close #\))))
                (setq k (|getMatchingRightPren| u (1+ j) open close))
                (cond
                  ((not (equal namestring (setq firstWord (substring u 0 i))))
                    (|checkDocError|
                     (list "Improper first word in comments: " firstWord)
                     u)
                  (null k)
                  (cond
                     ((equal open (|char| '['))
                      (|checkDocError|
                       (list "Missing close bracket on first line: " u)))
                     (t
                      (|checkDocError|
                       (list "Missing close parenthesis on first line: " u))))
                  u)
                (t
                  (strconc "\\spad{" (substring u 0 (1+ k)) "}"
                    (substring u (1+ k) nil))))))
            (t
              (setq k (or (|checkSkipToken| u j m) (return u)))
              (setq infixOp (intern (substring u j (- k j))))
              (cond
                ; case 3: form arg

```

```

((null (get1 infixOp '|Led|)))
(cond
  ((not (equal namestring (setq firstWord (substring u 0 i))))
    (|checkDocError|
      (list "Improper first word in comments: " firstWord))
    u)
  ((and (eql (|#| (setq p (pname infixOp))) 1)
    (setq open (elt p 0))
    (setq close (lassoc open |$checkPrenAlist|)))
    (setq z (|getMatchingRightPren| u (1+ k) open close))
    (when (> z (maxindex u)) (setq z (1- k)))
    (strconc "\\spad{" (substring u 0 (1+ z)) "}"
      (substring u (1+ z) nil)))
  (t
    (strconc "\\spad{" (substring u 0 k) "}"
      (substring u k nil))))))
(t
  (setq z (or (|checkSkipBlanks| u k m) (return u)))
  (setq n (or (|checkSkipToken| u z m) (return u)))
  (cond
    ((not (equal namestring (pname infixOp)))
      (|checkDocError|
        (list "Improper initial operator in comments: " infixOp))
      u)
    (t
      (strconc "\\spad{" (substring u 0 n) "}"
        (substring u n nil)))))))))
; case 4: arg op arg
(t
  (setq i (or (|checkSkipToken| u 0 m) (return u)))
  (cond
    ((not (equal namestring (setq firstWord (substring u 0 i))))
      (|checkDocError|
        (list "Improper first word in comments: " firstWord))
      u)
    (t
      (setq prefixOp (intern (substring u 0 i)))
      (cond
        ((null (get1 prefixOp '|Nud|)) u)
        (t
          (setq j (or (|checkSkipBlanks| u i m) (return u)))
          (cond
            ((equal (elt u j) (|char| '|(|))
              (setq j
                (|getMatchingRightPren| u (1+ j) (|char| '|(|) (|char| '|(|)))))
              (cond
                ((> j m) u)
                (t
                  (strconc "\\spad{" (substring u 0 (1+ j)) "}"

```

```

                                (substring u (1+ j) nil))))))
(t
  (setq k (or (|checkSkipToken| u j m) (return u)))
  (cond
    ((not (equal namestring (setq firstWord (substring u 0 i))))
      (|checkDocError|
        (list "Improper first word in comments: " firstWord))
      u)
    (t
      (strconc "\\spad{" (substring u 0 k) "} "
        (substring u k nil))))))))))

```

defun checkTrim

```

[charPosition p??]
[systemError p??]
[checkDocError p517]
[$charBlank p??]
[$x p??]
[$charPlus p??]

```

— defun checkTrim —

```

(defun |checkTrim| (|x| lines)
  (declare (special |x|))
  (labels (
    (trim (s)
      (let (k)
        (declare (special |$charBlank|))
        (setq k (wherePP s))
        (substring s (+ k 2) nil)))
    (wherePP (u)
      (let (k)
        (declare (special |$charPlus|))
        (setq k (|charPosition| |$charPlus| u 0))
        (if (or (eql k (|#| u))
          (not (eql (|charPosition| |$charPlus| u (1+ k)) (1+ k))))
          (|systemError| " Improper comment found")
          k))))
    (let (j s)
      (setq s (list (wherePP (car lines))))
      (loop for x in (rest lines)
        do
          (setq j (wherePP x))
          (unless (member j s)

```



```

(|checkDocError| (list |$x| " has varying indentation levels"))
  (setq s (cons j s))))
(loop for y in lines
  collect (trim y))))

```

13.8 Comment Checking Layer 7

defun checkDocError

```

[checkDocMessage p519]
[concat p??]
[saybrightly1 p??]
[sayBrightly p??]
[$checkErrorFlag p??]
[$recheckingFlag p??]
[$constructorName p??]
[$exposeFlag p??]
[$exposeFlagHeading p??]
[$outStream p??]
[$checkErrorFlag p??]
[$exposeFlagHeading p??]

```

— defun checkDocError —

```

(defun |checkDocError| (u)
  (let (msg)
    (declare (special |$outStream| |$exposeFlag| |$exposeFlagHeading|
                      |$constructorName| |$recheckingFlag| |$checkErrorFlag|))
    (setq |$checkErrorFlag| t)
    (setq msg
      (cond
        (|$recheckingFlag|
          (if |$constructorName|
              (|checkDocMessage| u)
              (|concat| "> " u)))
        (|$constructorName| (|checkDocMessage| u))
        (t u)))
    (when (and |$exposeFlag| |$exposeFlagHeading|)
      (saybrightly1 |$exposeFlagHeading| |$outStream|)
      (|sayBrightly| |$exposeFlagHeading|)
      (setq |$exposeFlagHeading| nil))
    (|sayBrightly| msg)
    (when |$exposeFlag| (saybrightly1 msg |$outStream|))))

```

defun checkRemoveComments

[checkTrimCommented p526]

— defun checkRemoveComments —

```
(defun |checkRemoveComments| (lines)
  (let (line acc)
    (loop while lines
      do
        (setq line (|checkTrimCommented| (car lines)))
        (when (>= (|firstNonBlankPosition| line) 0) (push line acc))
        (pop lines))
    (nreverse acc)))
```

defun checkSkipToken

[checkSkipIdentifierToken p525]

[checkSkipOpToken p525]

— defun checkSkipToken —

```
(defun |checkSkipToken| (u i m)
  (if (alpha-char-p (elt u i))
      (|checkSkipIdentifierToken| u i m)
      (|checkSkipOpToken| u i m)))
```

defun checkSplit2Words

[checkSplitBrace p525]

— defun checkSplit2Words —

```
(defun |checkSplit2Words| (u)
  (let (x verbatim z acc)
    (setq acc nil)
    (loop while u
      do
```

```

(setq x (car u))
(setq acc
  (cond
    ((string= x "\\end{verbatim}")
      (setq verbatim nil)
      (cons x acc))
    (verbatim (cons x acc))
    ((string= x "\\begin{verbatim}")
      (setq verbatim t)
      (cons x acc))
    ((setq z (|checkSplitBrace| x))
      (append (nreverse z) acc))
    (t (cons x acc))))
(pop u))
(nreverse acc)))

```

13.9 Comment Checking Layer 8

defun checkAddIndented

[firstNonBlankPosition p538]

[strconc p??]

[stringimage p??]

[checkAddSpaceSegments p529]

TPDHERE: Note that this function was missing without error, so may be junk

— defun checkAddIndented —

```

(defun |checkAddIndented| (x margin)
  (let (k)
    (setq k (|firstNonBlankPosition| x))
    (cond
      ((eql k -1) "\\blankline ")
      ((eql margin k) x)
      (t
        (strconc "\\indented{" (stringimage (- k margin)) "}{"
          (|checkAddSpaceSegments| (substring x k nil) 0) "}")))))

```

defun checkDocMessage

[getdatabase p??]

[whoOwns p541]

```
[concat p??]
[$x p??]
[$constructorName p??]
```

— defun checkDocMessage —

```
(defun |checkDocMessage| (u)
  (let (sourcefile person middle)
    (declare (special |$constructorName| |$x|))
    (setq sourcefile (getdatabase |$constructorName| 'sourcefile))
    (setq person (or (|whoOwns| |$constructorName|) "---"))
    (setq middle
      (if (boundp '|$x|)
          (list "(" |$x| "): ")
          (list ": ")))
    (|concat| person ">" sourcefile "-->" |$constructorName| middle u)))
```

defun checkExtract

```
[firstNonBlankPosition p538]
[substring? p??]
[charPosition p??]
[length p??]
```

— defun checkExtract —

```
(defun |checkExtract| (header lines)
  (let (line u margin firstLines m k j i acc)
    (loop while lines
      do
        (setq line (car lines))
        (setq k (|firstNonBlankPosition| line)) ; gives margin of description
        (if (|substring?| header line k)
            (return nil)
            (setq lines (cdr lines))))
    (cond
      ((null lines) nil)
      (t
       (setq u (car lines))
       (setq j (|charPosition| #\: u k))
       (setq margin k)
       (setq firstLines
        (if (not (eq1 (setq k (|firstNonBlankPosition| u (1+ j))) -1))
            (cons (substring u (1+ j) nil) (cdr lines))
            (cdr lines)))))
```

```

; now look for another header; if found skip all rest of these lines
(setq acc nil)
(loop for line in firstLines
  do
    (setq m (|#| line))
    (cond
      ((eql (setq k (|firstNonBlankPosition| line)) -1) '|skip|)
      ((> k margin) '|skip|)
      ((null (upper-case-p (elt line k))) '|skip|)
      ((equal (setq j (|charPosition| #\: line k)) m) '|skip|)
      ((> j (setq i (|charPosition| #\space line (1+ k)))) '|skip|)
      (t (return nil)))
    (setq acc (cons line acc)))
(nreverse acc))))

```

defun checkGetArgs

```

[maxindex p??]
[firstNonBlankPosition p538]
[checkGetArgs p521]
[stringPrefix? p??]
[getMatchingRightPren p538]
[charPosition p??]
[trimString p??]
[$charComma p??]

```

— defun checkGetArgs —

```

(defun |checkGetArgs| (u)
  (let (m k acc i)
    (declare (special |$charComma|))
    (cond
      ((null (stringp u)) nil)
      (t
       (setq m (maxindex u))
       (setq k (|firstNonBlankPosition| u))
       (cond
         ((> k 0)
          (|checkGetArgs| (substring u k nil)))
         ((|stringPrefix?| "\\spad{" u)
          (setq k (or (|getMatchingRightPren| u 6 #{ #\} m))
                     (|checkGetArgs| (substring u 6 (- k 6)))))
         ((> (setq i (|charPosition| #\ ( u 0)) m)
          nil)
          ((not (eql (elt u m) #\)))

```

```

nil)
(t
  (do ()
    ((null (> m (setq k (|charPosition| |$charComma| u (1+ i)))))) nil)
    (setq acc
      (cons (|trimString| (substring u (1+ i) (1- (- k i)))) acc))
    (setq i k))
  (nreverse (cons (substring u (1+ i) (1- (- m i))) acc))))))

```

defun checkGetMargin

[firstNonBlankPosition p[538](#)]

— defun checkGetMargin —

```

(defun |checkGetMargin| (lines)
  (let (x k margin)
    (loop while lines
      do
        (setq x (car lines))
        (setq k (|firstNonBlankPosition| x))
        (unless (= k -1) (setq margin (if margin (min margin k) k)))
        (pop lines))
    (or margin 0)))

```

defun checkGetParse

[ncParseFromString p??]

[removeBackslashes p[541](#)]

— defun checkGetParse —

```

(defun |checkGetParse| (s)
  (|ncParseFromString| (|removeBackslashes| s)))

```

defun checkGetStringBeforeRightBrace

[\$charRbrace p??]

— defun checkGetStringBeforeRightBrace —

```

(defun |checkGetStringBeforeRightBrace| (u)
  (prog (x acc)
    (declare (special |$charRbrace|))
    (return
      (loop while u
        do
          (setq x (car u))
          (cond
            ((equal x |$charRbrace|)
             (let ((result ""))
               (loop for item in acc
                 do (setq result (concatenate 'string item result)))
               (return result)))
            (t
             (setq acc (cons x acc))
             (setq u (cdr u))))))))

```

—————

defun checkIeEg

[checkIeEgfun p531]

[nreverse p??]

— defun checkIeEg —

```

(defun |checkIeEg| (u)
  (let (x verbatim z acc)
    (setq acc nil)
    (setq verbatim nil)
    (loop while u
      do
        (setq x (car u))
        (setq acc
          (cond
            ((equal x "\\end{verbatim}")
             (setq verbatim nil)
             (cons x acc))
            (verbatim (cons x acc))
            ((equal x "\\begin{verbatim}")
             (setq verbatim t)

```

```

      (cons x acc))
    ((setq z (|checkIeEgfun| x))
     (append (nreverse z) acc))
    (t (cons x acc))))
  (setq u (cdr u)))
(nreverse acc))

```

defun checkIndentedLines

[firstNonBlankPosition p538]
 [strconc p??]
 [checkAddSpaceSegments p529]
 [\$charFauxNewline p??]

— defun checkIndentedLines —

```

(defun |checkIndentedLines| (u margin)
  (let (k s verbatim u2)
    (declare (special |$charFauxNewline|))
    (loop for x in u
      do
        (setq k (|firstNonBlankPosition| x))
        (cond
          ((eql k -1)
           (if verbatim
              (setq u2 (append u2 (list |$charFauxNewline|)))
              (setq u2 (append u2 (list "\\blankline "))))))
          (t
           (setq s (substring x k nil))
           (cond
             ((string= s "\\begin{verbatim}")
              (setq verbatim t)
              (setq u2 (append u2 (list s))))
             ((string= s "\\end{verbatim}")
              (setq verbatim nil)
              (setq u2 (append u2 (list s))))
             (verbatim
              (setq u2 (append u2 (list (substring x margin nil))))))
             ((eql margin k)
              (setq u2 (append u2 (list s))))
             (t
              (setq u2
                (append u2
                  (list (strconc "\\indented{" (stringimage (- k margin))
                               "}" (|checkAddSpaceSegments| s 0) "}")
                      ))))))))

```



```
u2))
```

defun checkSkipIdentifierToken

[checkAlphabetic p531]

— defun checkSkipIdentifierToken —

```
(defun |checkSkipIdentifierToken| (u i m)
  (do ()
    ((null (and (> m i) (|checkAlphabetic| (elt u i)))) nil)
    (setq i (1+ i)))
  (unless (= i m) i))
```

defun checkSkipOpToken

[checkAlphabetic p531]

[member p??]

[\$charDelimiters p??]

— defun checkSkipOpToken —

```
(defun |checkSkipOpToken| (u i m)
  (declare (special |$charDelimiters|))
  (do ()
    ((null (and (> m i)
                (null (|checkAlphabetic| (elt u i)))
                (null (|member| (elt u i) |$charDelimiters|))))
      nil)
    (setq i (1+ i)))
  (unless (= i m) i))
```

defun checkSplitBrace

[charp p??]

[length p??]

[checkSplitBackslash p535]
 [checkSplitBrace p525]
 [checkSplitOn p536]
 [checkSplitPunctuation p537]

— defun checkSplitBrace —

```
(defun |checkSplitBrace| (x)
  (let (m u)
    (cond
      ((charp x) (list x))
      ((eql (|#| x) 1) (list (elt x 0)))
      ((and (setq u (|checkSplitBackslash| x)) (cdr u))
        (let (result)
          (loop for y in u do (append result (|checkSplitBrace| y)))
          result))
      (t
        (setq m (maxindex x))
        (cond
          ((and (setq u (|checkSplitOn| x)) (cdr u))
            (let (result)
              (loop for y in u do (append result (|checkSplitBrace| y)))
              result))
          ((and (setq u (|checkSplitPunctuation| x)) (cdr u))
            (let (result)
              (loop for y in u do (append result (|checkSplitBrace| y)))
              result))
          (t (list x)))))))
```

—————

defun checkTrimCommented

[length p??]
 [htcharPosition p539]

— defun checkTrimCommented —

```
(defun |checkTrimCommented| (line)
  (let (n k)
    (setq n (|#| line))
    (setq k (|htcharPosition| (|char| '%) line 0))
    (cond
      ((eql k 0) "")
      ((or (>= k (1- n)) (not (eql (elt line (1+ k)) #\%))) line)
      ((> (|#| line) k) (substring line 0 k))
      (t line))))
```

defun newString2Words

[newWordFrom p540]
[nreverse0 p??]

— **defun newString2Words** —

```
(defun |newString2Words| (z)
  (let (m tmp1 w i result)
    (cond
      ((null (stringp z)) (list z))
      (t
       (setq m (maxindex z))
       (cond
         ((eql m -1) nil)
         (t
          (setq i 0)
          (do () ; [w while newWordFrom(l,i,m) is [w,i]]
              ((null (progn
                      (setq tmp1 (|newWordFrom| z i m))
                      (and (consp tmp1)
                          (progn
                           (setq w (qcar tmp1))
                           (and (consp (qcdr tmp1))
                               (eq (qcddr tmp1) nil)
                               (progn
                                (setq i (qcadr tmp1))
                                t))))))
              (nreverse0 result))
          (setq result (cons (qcar tmp1) result))))))))))
```

13.10 Comment Checking Layer 9**defun checkAddBackSlashes**

[strconc p??]
[maxindex p??]
[checkAddBackSlashes p527]
[\$charBack p??]

`[$charEscapeList p??]`

— **defun checkAddBackSlashes** —

```
(defun |checkAddBackSlashes| (s)
  (let (c m char insertIndex k)
    (declare (special |$charBack| |$charEscapeList|))
    (cond
      ((or (and (charp s) (setq c s))
           (and (eql (|#| s) 1) (setq c (elt s 0)))))
      (if (member s |$charEscapeList|)
          (strconc |$charBack| c)
          s))
    (t
     (setq k 0)
     (setq m (maxindex s))
     (setq insertIndex nil)
     (loop while (< k m)
       do
         (setq char (elt s k))
         (cond
           ((char= char |$charBack|) (setq k (+ k 2)))
           ((member char |$charEscapeList|) (return (setq insertIndex k))))
         (setq k (1+ k)))
     (cond
       (insertIndex
        (|checkAddBackSlashes|
         (strconc (substring s 0 insertIndex) |$charBack| (elt s k)
                  (substring s (1+ insertIndex) nil))))
       (T s))))))
```

—————

defun checkAddMacros

`[lassoc p??]`
`[nreverse p??]`
`[$HTmacros p??]`

— **defun checkAddMacros** —

```
(defun |checkAddMacros| (u)
  (let (x verbatim y acc)
    (declare (special |$HTmacros|))
    (loop while u
      do
        (setq x (car u))
```

```

(setq acc
  (cond
    ((string= x "\\end{verbatim}")
      (setq verbatim nil)
      (cons x acc))
    (verbatim
      (cons x acc))
    ((string= x "\\begin{verbatim}")
      (setq verbatim t)
      (cons x acc))
    ((setq y (lassoc x |$HTmacs|))
      (append y acc))
    (t (cons x acc))))
(pop u))
(nreverse acc)))

```

defun checkAddPeriod

```

[setelt p??]
[maxindex p??]

```

— defun checkAddPeriod —

```

(defun |checkAddPeriod| (s)
  (let (m lastChar)
    (setq m (maxindex s))
    (setq lastChar (elt s m))
    (cond
      ((or (char= lastChar #\!) (char= lastChar #\?) (char= lastChar #\.) s)
        ((or (char= lastChar #\,) (char= lastChar #\;))
          (setelt s m #\.)
          s)
      (t s))))

```

defun checkAddSpaceSegments

```

[checkAddSpaceSegments p529]
[maxindex p??]
[charPosition p??]
[strconc p??]
[$charBlank p??]

```

— defun checkAddSpaceSegments —

```
(defun |checkAddSpaceSegments| (u k)
  (let (m i j n)
    (declare (special |$charBlank|))
    (setq m (maxindex u))
    (setq i (|charPosition| |$charBlank| u k))
    (cond
      ((> i m) u)
      (t
       (setq j i)
       (loop while (and (incf j) (char= (elt u j) #\space)))
       (setq n (- j i)) ; number of blanks
       (if (> n 1)
         (strconc (substring u 0 i) "\\space{" (stringimage n) "}")
         (|checkAddSpaceSegments| (substring u (+ i n) nil) 0))
       (|checkAddSpaceSegments| u j))))))
```

—————

defun checkAddSpaces

```
[$charBlank p??]
[$charFauxNewline p??]
```

— defun checkAddSpaces —

```
(defun |checkAddSpaces| (u)
  (let (u2 space i)
    (declare (special |$charBlank| |$charFauxNewline|))
    (cond
      ((null u) nil)
      ((null (cdr u)) u)
      (t
       (setq space |$charBlank|)
       (setq i 0)
       (loop for f in u
         do
           (incf i)
           (when (string= f "\\begin{verbatim}")
             (setq space |$charFauxNewline|)
             (unless u2 (setq u2 (list space))))
           (if (> i 1)
             (setq u2 (append u2 (list space f)))
             (setq u2 (append u2 (list f))))
           (when (string= f "\\end{verbatim}")
```

```

      (setq u2 (append u2 (list space)))
      (setq space |$charBlank|)))
    u2)))

```

defun checkAlphabetic

[*\$charIdentifierEndings* *p??*]

— defun checkAlphabetic —

```

(defun |checkAlphabetic| (c)
  (declare (special |$charIdentifierEndings|))
  (or (alpha-char-p c) (digitp c) (member c |$charIdentifierEndings|)))

```

defun checkIeEgfun

[*maxindex* *p??*]
 [*checkIeEgFun* *p??*]
 [*\$charPeriod* *p??*]

— defun checkIeEgfun —

```

(defun |checkIeEgfun| (x)
  (let (m key firstPart result)
    (declare (special |$charPeriod|))
    (cond
      ((characterp x) nil)
      ((equal x "") nil)
      (t
       (setq m (maxindex x))
       (loop for k from 0 to (- m 3)
         do
           (cond
             ((and
              (equal (elt x (1+ k)) |$charPeriod|)
              (equal (elt x (+ k 3)) |$charPeriod|)
              (or
               (and
                (equal (elt x k) #\i)
                (equal (elt x (+ k 2)) #\e)
                (setq key "that is"))
              )
             )

```

```

      (and
        (equal (elt x k) #\e)
        (equal (elt x (+ k 2)) #\g)
        (setq key "for example"))))
    (progn
      (setq firstPart (when (> k 0) (cons (substring x 0 k) nil)))
      (setq result
        (append firstPart
          (cons "\\spadignore{"
            (cons (substring x k 4)
              (cons "}"
                (|checkIeEgfun| (substring x (+ k 4) nil))))))))))
    result)))

```

defun checkIsValidType

This function returns ok if correct, form is wrong number of arguments, nil if unknown

[length p??]

[checkIsValidType p⁵³²]

[constructor? p??]

[abbreviation? p??]

[getdatabase p??]

— defun checkIsValidType —

```

(defun |checkIsValidType| (form)
  (labels (
    (fn (form coSig)
      (cond
        ((not (eql (|#| form) (|#| coSig))) form)
        ((let (result)
            (loop for x in (rest form)
                  for flag in (rest coSig)
                  do (when flag (setq result (or result (null (|checkIsValidType| x))))))
            result)
          nil)
         (t '|ok|))))
    (let (op args conname)
      (cond
        ((atom form) '|ok|)
        (t
         (setq op (car form))
         (setq args (cdr form))
         (setq conname
          (if (|constructor?| op)

```



```

      op
      (|abbreviation?| op)))
    (when conname (fn form (getdatabase conname 'cosig)))))))))

```

defun checkLookForLeftBrace

```

[$charBlank p??]
[$charLbrace p??]

```

— defun checkLookForLeftBrace —

```

(defun |checkLookForLeftBrace| (u)
  (declare (special |$charBlank| |$charLbrace|))
  (loop while u
    do
      (cond
        ((equal (car u) |$charLbrace|) (return (car u)))
        ((not (eql (car u) |$charBlank|)) (return nil))
        (t (pop u))))
    u)

```

defun checkLookForRightBrace

This returns a line beginning with right brace [\$charLbrace p??]
[\$charRbrace p??]

— defun checkLookForRightBrace —

```

(defun |checkLookForRightBrace| (u)
  (let (found count)
    (declare (special |$charLbrace| |$charRbrace|))
    (setq count 0)
    (loop while u
      do
        (cond
          ((equal (car u) |$charRbrace|)
            (if (eql count 0)
              (return (setq found u))
              (setq count (1- count)))))
          ((equal (car u) |$charLbrace|)
            (setq count (1+ count)))))

```

```
(pop u))
found))
```

defun checkNumOfArgs

A nil return implies that the argument list length does not match [opOf p??]
 [constructor? p??]
 [abbreviation? p??]
 [getdatabase p??]

— defun checkNumOfArgs —

```
(defun |checkNumOfArgs| (conform)
  (let (conname)
    (setq conname (|opOf| conform))
    (when (or (|constructor?| conname) (setq conname (|abbreviation?| conname)))
      (|#| (getdatabase conname 'constructorargs))))))
```

defun checkSayBracket

— defun checkSayBracket —

```
(defun |checkSayBracket| (x)
  (cond
    ((or (char= x #\() (char= x #\))) "pren")
    ((or (char= x #\{ (char= x #\}) "brace")
    (t "bracket"))))
```

defun checkSkipBlanks

[\$charBlank p??]

— defun checkSkipBlanks —

```
(defun |checkSkipBlanks| (u i m)
```

```
(declare (special |$charBlank|))
(do ()
  ((null (and (> m i) (equal (elt u i) |$charBlank|))) nil)
  (setq i (1+ i)))
(unless (= i m) i))
```

—————

defun checkSplitBackslash

[checkSplitBackslash p⁵³⁵]
 [maxindex p??]
 [charPosition p??]
 [\$charBack p??]

— defun checkSplitBackslash —

```
(defun |checkSplitBackslash| (x)
  (let (m k u v)
    (declare (special |$charBack|))
    (cond
      ((null (stringp x)) (list x))
      (t
       (setq m (maxindex x))
       (cond
         ((> m (setq k (|charPosition| |$charBack| x 0)))
          (cond
            ((or (eql m 1) (alpha-char-p (elt x (1+ k)))) ;starts with backslash so
             (if (> m (setq k (|charPosition| |$charBack| x 1)))
                 ; yes, another backslash
                 (cons (substring x 0 k) (|checkSplitBackslash| (substring x k nil)))
                 ; no, just return the line
                 (list x)))
            ((eql k 0)
             ; starts with backspace but x.1 is not a letter; break it up
             (cons (substring x 0 2)
                   (|checkSplitBackslash| (substring x 2 nil))))
          (t
           (setq u (substring x 0 k))
           (setq v (substring x k 2))
           (if (= (1+ k) m)
               (list u v)
               (cons u
                     (cons v
                           (|checkSplitBackslash|
                            (substring x (+ k 2) nil)))))))
       (t (list x))))))
```

defun checkSplitOn

```
[checkSplitOn p536]
[charp p??]
[maxindex p??]
[charPosition p??]
[$charBack p??]
[$charSplitList p??]
```

— defun checkSplitOn —

```
(defun |checkSplitOn| (x)
  (let (m char k z)
    (declare (special |$charBack| |$charSplitList|))
    (cond
      ((charp x) (list x))
      (t
       (setq z |$charSplitList|)
       (setq m (maxindex x))
       (loop while z
         do
           (setq char (car z))
           (cond
             ((and (eql m 0) (equal (elt x 0) char))
              (return (setq k -1)))
             (t
              (setq k (|charPosition| char x 0))
              (cond
                ((and (> k 0) (equal (elt x (1- k)) |$charBack|)) (list x))
                ((<= k m) (return k))))
              (pop z))
           (cond
             ((null z) (list x))
             ((eql k -1) (list char))
             ((eql k 0) (list char (substring x 1 nil)))
             ((eql k (maxindex x)) (list (substring x 0 k) char))
             (t
              (cons (substring x 0 k)
                     (cons char (|checkSplitOn| (substring x (1+ k) nil))))))))))
```

defun checkSplitPunctuation

```
[charp p??]
[maxindex p??]
[checkSplitPunctuation p537]
[charPosition p??]
[hget p??]
[$charDash p??]
[$htMacroTable p??]
[$charQuote p??]
[$charPeriod p??]
[$charSemiColon p??]
[$charComma p??]
[$charBack p??]
```

— defun checkSplitPunctuation —

```
(defun |checkSplitPunctuation| (x)
  (let (m lastchar v k u)
    (declare (special |$charDash| |$htMacroTable| |$charBack| |$charQuote|
                      |$charComma| |$charSemiColon| |$charPeriod|))
    (cond
      ((charp x) (list x))
      (t
       (setq m (maxindex x))
       (cond
         ((> 1 m) (list x))
         (t
          (setq lastchar (elt x m))
          (cond
            ((and (equal lastchar |$charPeriod|)
                  (equal (elt x (1- m)) |$charPeriod|))
             (cond
               ((eql m 1) (list x))
               ((and (> m 3) (equal (elt x (- m 2)) |$charPeriod|))
                (append (|checkSplitPunctuation| (substring x 0 (- m 2)))
                        (list "..."))))
               (t
                (append (|checkSplitPunctuation| (substring x 0 (1- m)))
                        (list "..")))))
            ((or (equal lastchar |$charPeriod|)
                  (equal lastchar |$charSemiColon|)
                  (equal lastchar |$charComma|))
             (list (substring x 0 m) lastchar))
            ((and (> m 1) (equal (elt x (1- m)) |$charQuote|))
             (list (substring x 0 (1- m)) (substring x (1- m) nil)))
            ((> m (setq k (|charPosition| |$charBack| x 0)))
             (cond
               ((eql k 0)
```

```

(cond
  ((or (eql m 1) (hget |$htMacroTable| x) (alpha-char-p (elt x 1)))
    (list x))
  (t
    (setq v (substring x 2 nil))
    (cons (substring x 0 2) (|checkSplitPunctuation| v))))
(t
  (setq u (substring x 0 k))
  (setq v (substring x k nil))
  (append (|checkSplitPunctuation| u)
    (|checkSplitPunctuation| v))))
(> m (setq k (|charPosition| |$charDash| x 1)))
  (setq u (substring x (1+ k) nil))
  (cons (substring x 0 k)
    (cons |$charDash| (|checkSplitPunctuation| u))))
(t
  (list x))))))

```

defun firstNonBlankPosition

[maxindex p??]

— defun firstNonBlankPosition —

```

(defun |firstNonBlankPosition| (&rest therest)
  (let ((x (car therest)) (options (cdr therest)) start k)
    (declare (special |$charBlank|))
    (setq start (or (ifcar options) 0))
    (setq k -1)
    (loop for i from start to (maxindex x)
      do (when (not (eql (elt x i) |$charBlank|)) (return (setq k i))))
    k))

```

defun getMatchingRightPren

[maxindex p??]

— defun getMatchingRightPren —

```

(defun |getMatchingRightPren| (u j open close)
  (let (m c found count)

```

```

(setq count 0)
(setq m (maxindex u))
(loop for i from j to m
  do
    (setq c (elt u i))
    (cond
      ((equal c close)
        (if (eql count 0)
          (return (setq found i))
          (setq count (1- count)))))
      ((equal c open)
        (setq count (1+ count)))))
found))

```

defun hasNoVowels

[maxindex p??]

— defun hasNoVowels —

```

(defun |hasNoVowels| (x)
  (labels (
    (isVowel (c)
      (or (eq c #\a) (eq c #\e) (eq c #\i) (eq c #\o) (eq c #\u)
          (eq c #\A) (eq c #\E) (eq c #\I) (eq c #\O) (eq c #\U))))
    (let (max)
      (setq max (maxindex x))
      (cond
        ((char= (elt x max) #\y) nil)
        (t
         (let ((result t))
           (loop for i from 0 to max
             do (setq result (and result (null (isVowel (elt x i))))))
           result))))))

```

defun htcharPosition

[length p??]

[charPosition p??]

[htcharPosition p539]

[\$charBack p??]

— defun htcharPosition —

```
(defun |htcharPosition| (char line i)
  (let (m k)
    (declare (special |$charBack|))
    (setq m (|#| line))
    (setq k (|charPosition| char line i))
    (cond
      ((eql k m) k)
      (> k 0)
      (if (not (eql (elt line (1- k)) |$charBack|))
          k
          (|htcharPosition| char line (1+ k))))
      (t 0))))
```

—————

defun newWordFrom

```
[$stringFauxNewline p??]
[$charBlank p??]
[$charFauxNewline p??]
```

— defun newWordFrom —

```
(defun |newWordFrom| (z i m)
  (let (ch done buf)
    (declare (special |$charFauxNewline| |$charBlank| |$stringFauxNewline|))
    (loop while (and (<= i m) (char= (elt z i) #\space)) do (incf i))
    (cond
      (> i m) nil
      (t
       (setq buf "")
       (setq ch (elt z i))
       (cond
         ((equal ch |$charFauxNewline|)
          (list |$stringFauxNewline| (1+ i)))
         (t
          (setq done nil)
          (loop while (and (<= i m) (null done))
                do
                  (setq ch (elt z i))
                  (cond
                    ((or (equal ch |$charBlank|) (equal ch |$charFauxNewline|))
                     (setq done t))
                    (t
                     (t
```



```

      (setq buf (strconc buf ch))
      (setq i (1+ i))))
    (list buf i))))))

```

defun removeBackslashes

```

[charPosition p??]
[removeBackslashes p541]
[strconc p??]
[length p??]
[$charBack p??]

```

— defun removeBackslashes —

```

(defun |removeBackslashes| (s)
  (let (k)
    (declare (special |$charBack|))
    (cond
      ((string= s "") "")
      ((> (|#| s) (setq k (|charPosition| |$charBack| s 0)))
       (if (eql k 0)
           (|removeBackslashes| (substring s 1 nil))
           (strconc (substring s 0 k)
                     (|removeBackslashes| (substring s (1+ k) nil))))))
      (t s))))

```

defun whoOwns

This function always returns nil in the current system. Since it has no side effects we define it to return nil. [getdatabase p??]

```

[strconc p??]
[awk p??]
[shut p??]
[$exposeFlag p??]

```

— defun whoOwns —

```

(defun |whoOwns| (con)
  (declare (ignore con))
  nil)

```

```
; (let (filename quoteChar instream value)
; (declare (special |$exposeFlag|))
; (cond
;   ((null |$exposeFlag|) nil)
;   (t
;    (setq filename (getdatabase con 'sourcefile))
;    (setq quoteChar #\")
;    (obey (strconc "awk '$2 == \" quoteChar filename quoteChar
;                \" {print $1}' whofiles > /tmp/temp"))
;    (setq instream (make-instream "/tmp/temp"))
;    (setq value (unless (eofp instream) (readline instream)))
;    (shut instream)
;    value))))
```

Chapter 14

Utility Functions

defun translabel

[translabel1 p543]

— defun translabel —

```
(defun translabel (x al)
  (translabel1 x al) x)
```

—————

defun translabel1

[refvecp p??]

[maxindex p??]

[translabel1 p543]

[lassoc p??]

— defun translabel1 —

```
(defun translabel1 (x al)
  "Transforms X according to AL = ((<label> . Sexpr) ..)."
  (cond
    ((refvecp x)
     (do ((i 0 (1+ i)) (k (maxindex x)))
         ((> i k)
          (if (let ((y (lassoc (elt x i) al))) (setelt x i y))
              (translabel1 (elt x i) al))))
     ((atom x) nil)
    ((let ((y (lassoc (first x) al)))
```

```
(if y (setf (first x) y) (translabel1 (cdr x) al))))
((translabel1 (first x) al) (translabel1 (cdr x) al))))
```

defun displayPreCompilationErrors

```
[length p??]
[remdup p??]
[sayBrightly p??]
[sayMath p??]
[$postStack p??]
[$topOp p??]
[$InteractiveMode p??]
```

— defun displayPreCompilationErrors —

```
(defun |displayPreCompilationErrors| ()
  (let (n errors heading)
    (declare (special |$postStack| |$topOp| |$InteractiveMode|))
    (setq n (|#| (setq |$postStack| (remdup (nreverse |$postStack|)))))
    (unless (eql n 0)
      (setq errors (cond ((> n 1) "errors") (t "error")))
      (cond
        (|$InteractiveMode|
         (|sayBrightly| (list " Semantic " errors " detected: ")))
        (t
         (setq heading
           (if (not (eq |$topOp| '|$topOp|))
               (list " " |$topOp| " has")
               (list " You have")))
         (|sayBrightly|
          (append heading (list n "precompilation " errors ":" )))))
      (cond
        ((> n 1)
         (let ((i 1))
           (dolist (x |$postStack|)
             (|sayMath| (cons " " (cons i (cons " " x))))))
          (t (|sayMath| (cons " " (car |$postStack|)))))
        (terpri))))
```

defun bumperrorcount

[[\\$InteractiveMode](#) p??]
 [[\\$spad-errors](#) p??]

— defun bumperrorcount —

```
(defun bumperrorcount (kind)
  (declare (special |$InteractiveMode| $spad_errors))
  (unless |$InteractiveMode|
    (let ((index (case kind
                    (|syntax| 0)
                    (|precompilation| 1)
                    (|semantic| 2)
                    (t (error (break "BUMPERRORCOUNT: kind=~s~%" kind))))))
      (setelt $spad_errors index (1+ (elt $spad_errors index))))))
```

defun parseTranCheckForRecord

[[postError](#) p370]
 [[parseTran](#) p97]

— defun parseTranCheckForRecord —

```
(defun |parseTranCheckForRecord| (x op)
  (declare (ignore op))
  (let (tmp3)
    (setq x (|parseTran| x))
    (cond
      ((and (consp x) (eq (qfirst x) '|Record|))
        (cond
          ((do ((z nil tmp3) (tmp4 (qrest x) (cdr tmp4)) (y nil))
              ((or z (atom tmp4)) tmp3)
              (setq y (car tmp4))
              (cond
                ((null (and (consp y) (eq (qfirst y) '|:|) (consp (qrest y))
                           (consp (qcddr y)) (eq (qcddr y) nil)))
                 (setq tmp3 (or tmp3 y))))
            (postError (list " Constructor" x "has missing label" )))
          (t x)))
      (t x))))
```

defun makeSimplePredicateOrNil

```
[isSimple p??]
[isAlmostSimple p??]
[wrapSEQExit p??]
```

— **defun makeSimplePredicateOrNil** —

```
(defun |makeSimplePredicateOrNil| (p)
  (let (u g)
    (cond
      ((|isSimple| p) nil)
      ((setq u (|isAlmostSimple| p)) u)
      (t (|wrapSEQExit| (list (list 'let (setq g (gensym)) p) g))))))
```

—————→

defun parse-spadstring

```
[match-current-token p459]
[token-symbol p??]
[push-reduction p468]
[advance-token p462]
```

— **defun parse-spadstring** —

```
(defun parse-spadstring ()
  (let* ((tok (match-current-token 'spadstring))
        (symbol (if tok (token-symbol tok))))
    (when tok
      (push-reduction 'spadstring-token (copy-tree symbol))
      (advance-token)
      t)))
```

—————→

defun parse-string

```
[match-current-token p459]
[token-symbol p??]
[push-reduction p468]
[advance-token p462]
```

— **defun parse-string** —

```
(defun parse-string ()
  (let* ((tok (match-current-token 'string))
         (symbol (if tok (token-symbol tok))))
    (when tok
      (push-reduction 'string-token (copy-tree symbol))
      (advance-token)
      t)))
```

defun parse-identifier

[match-current-token p459]
 [token-symbol p??]
 [push-reduction p468]
 [advance-token p462]

— defun parse-identifier —

```
(defun parse-identifier ()
  (let* ((tok (match-current-token 'identifier))
         (symbol (if tok (token-symbol tok))))
    (when tok
      (push-reduction 'identifier-token (copy-tree symbol))
      (advance-token)
      t)))
```

defun parse-number

[match-current-token p459]
 [token-symbol p??]
 [push-reduction p468]
 [advance-token p462]

— defun parse-number —

```
(defun parse-number ()
  (let* ((tok (match-current-token 'number))
         (symbol (if tok (token-symbol tok))))
    (when tok
      (push-reduction 'number-token (copy-tree symbol))
      (advance-token)
      t)))
```

defun parse-keyword

[match-current-token p459]
 [token-symbol p??]
 [push-reduction p468]
 [advance-token p462]

— defun parse-keyword —

```
(defun parse-keyword ()
  (let* ((tok (match-current-token 'keyword))
        (symbol (if tok (token-symbol tok))))
    (when tok
      (push-reduction 'keyword-token (copy-tree symbol))
      (advance-token)
      t)))
```

defun parse-argument-designator

[push-reduction p468]
 [match-current-token p459]
 [token-symbol p??]
 [advance-token p462]

— defun parse-argument-designator —

```
(defun parse-argument-designator ()
  (let* ((tok (match-current-token 'argument-designator))
        (symbol (if tok (token-symbol tok))))
    (when tok
      (push-reduction 'argument-designator-token (copy-tree symbol))
      (advance-token)
      t)))
```

defun print-package

[[\\$out-stream p??](#)]

— defun print-package —

```
(defun print-package (package)
  (declare (special out-stream))
  (format out-stream "~&~%(IN-PACKAGE ~S )~%~%" package))
```

—————

defun checkWarning

[[postError p370](#)]

[[concat p??](#)]

— defun checkWarning —

```
(defun |checkWarning| (msg)
  (postError (|concat| "Parsing error: " msg)))
```

—————

defun tuple2List

[[tuple2List p549](#)]

[[postTranSegment p384](#)]

[[postTran p366](#)]

[[\\$boot p??](#)]

[[\\$InteractiveMode p??](#)]

— defun tuple2List —

```
(defun |tuple2List| (arg)
  (let (u p q)
    (declare (special |$InteractiveMode| $boot))
    (when (consp arg)
      (setq u (|tuple2List| (qrest arg)))
      (cond
        ((and (consp (qfirst arg)) (eq (qcaar arg) 'segment)
              (consp (qcdar arg))
              (consp (qcddar arg))
              (eq (qcdddar arg) nil))
```

```

(setq p (qcadar arg))
(setq q (qcaddar arg))
(cond
  ((null u) (list '|construct| (|postTranSegment| p q)))
  ((and |$InteractiveMode| (null $boot))
   (cons '|append|
         (cons (list '|construct| (|postTranSegment| p q))
               (list (|tuple2List| (qrest arg)))))))
  (t
   (cons '|nconc|
         (cons (list '|construct| (|postTranSegment| p q))
               (list (|tuple2List| (qrest arg)))))))
  ((null u) (list '|construct| (|postTran| (qfirst arg))))
  (t (list '|cons| (|postTran| (qfirst arg)) (|tuple2List| (qrest arg))))))

```

defmacro pop-stack-1

[reduction-value p??]
 [Pop-Reduction p552]

— defmacro pop-stack-1 —

```
(defmacro pop-stack-1 () '(reduction-value (Pop-Reduction)))
```

defmacro pop-stack-2

[stack-push p93]
 [reduction-value p??]
 [Pop-Reduction p552]

— defmacro pop-stack-2 —

```

(defmacro pop-stack-2 ()
  '(let* ((top (Pop-Reduction)) (next (Pop-Reduction)))
    (stack-push top Reduce-Stack)
    (reduction-value next)))

```

defmacro pop-stack-3

[stack-push p93]
 [reduction-value p??]
 [Pop-Reduction p552]

— **defmacro pop-stack-3** —

```
(defmacro pop-stack-3 ()
  '(let* ((top (Pop-Reduction)) (next (Pop-Reduction)) (nnext (Pop-Reduction)))
    (stack-push next Reduce-Stack)
    (stack-push top Reduce-Stack)
    (reduction-value nnext)))
```

defmacro pop-stack-4

[stack-push p93]
 [reduction-value p??]
 [Pop-Reduction p552]

— **defmacro pop-stack-4** —

```
(defmacro pop-stack-4 ()
  '(let* ((top (Pop-Reduction))
    (next (Pop-Reduction))
    (nnext (Pop-Reduction))
    (nnnext (Pop-Reduction)))
    (stack-push nnnext Reduce-Stack)
    (stack-push next Reduce-Stack)
    (stack-push top Reduce-Stack)
    (reduction-value nnnext)))
```

defmacro nth-stack

[stack-store p??]
 [reduction-value p??]

— **defmacro nth-stack** —

```
(defmacro nth-stack (x)
```

```
'(reduction-value (nth (1- ,x) (stack-store Reduce-Stack))))
```

defun Pop-Reduction

[stack-pop [p94](#)]

— defun Pop-Reduction —

```
(defun Pop-Reduction () (stack-pop Reduce-Stack))
```

defun addclose

[suffix p??]

— defun addclose —

```
(defun addclose (line char)
  (cond
    ((char= (char line (maxindex line)) #\; )
     (setelt line (maxindex line) char)
     (if (char= char #\;) line (suffix #\; line)))
    ((suffix char line))))
```

defun blankp

— defun blankp —

```
(defun blankp (char)
  (or (eq char #\Space) (eq char #\tab)))
```

defun drop

Return a pointer to the Nth cons of X, counting 0 as the first cons. [drop p[553](#)]
 [take p??]
 [croak p??]

— defun drop —

```
(defun drop (n x &aux m)
  (cond
    ((eq1 n 0) x)
    ((> n 0) (drop (1- n) (cdr x)))
    ((>= (setq m (+ (length x) n)) 0) (take m x))
    ((croak (list "Bad args to DROP" n x)))))
```

—————

defun escaped

— defun escaped —

```
(defun escaped (str n)
  (and (> n 0) (eq (char str (1- n)) #\_)))
```

—————

defvar \$comblocklist

— initvars —

```
(defvar $comblocklist nil "a dynamic lists of comments for this block")
```

—————

defun fincomblock

- NUM is the line number of the current line
- OLDNUMS is the list of line numbers of previous lines
- OLDLOCS is the list of previous indentation locations

- NCBLOCK is the current comment block

```
[preparse-echo p92]
[$comblocklist p553]
[$EchoLineStack p??]
```

— **defun fincomblock** —

```
(defun fincomblock (num oldnums oldlocs ncblock linelist)
  (declare (special $EchoLineStack $comblocklist))
  (push
   (cond
    ((eql (car ncblock) 0) (cons (1- num) (reverse (cdr ncblock))))
    ;; comment for constructor itself paired with 1st line -1
    (t
     (when $EchoLineStack
      (setq num (pop $EchoLineStack))
      (preparse-echo linelist)
      (setq $EchoLineStack (list num)))
     (cons
      ;; scan backwards for line to left of current
      (do ((onums oldnums (cdr onums))
          (olocs oldlocs (cdr olocs))
          (sloc (car ncblock)))
          ((null onums) nil)
          (when (and (numberp (car olocs)) (<= (car olocs) sloc))
            (return (car onums))))
      (reverse (cdr ncblock)))))
   $comblocklist))
```

—————

defun indent-pos

```
[next-tab-loc p555]
```

— **defun indent-pos** —

```
(defun indent-pos (str)
  (do ((i 0 (1+ i)) (pos 0))
      ((>= i (length str)) nil)
      (case (char str i)
        (#\space (incf pos))
        (#\tab (setq pos (next-tab-loc pos)))
        (otherwise (return pos)))))
```

—————

defun infixtok

[string2id-n p??]

— defun infixtok —

```
(defun infixtok (s)
  (member (string2id-n s 1) '(|then| |else|) :test #'eq))
```

—————

defun is-console

[fp-output-stream p??]

[*terminal-io* p??]

— defun is-console —

```
(defun is-console (stream)
  (and (streamp stream) (output-stream-p stream)
    (eq (system:fp-output-stream stream)
        (system:fp-output-stream *terminal-io*))))
```

—————

defun next-tab-loc

— defun next-tab-loc —

```
(defun next-tab-loc (i)
  (* (1+ (truncate i 8)) 8))
```

—————

defun nonblankloc[blankp [p552](#)]

— defun nonblankloc —

```
(defun nonblankloc (str)
  (position-if-not #'blankp str))
```

defun parseprint

— defun parseprint —

```
(defun parseprint (l)
  (when l
    (format t "~&~%          ***      PREPARSE      ***~%~%"
      (dolist (x l) (format t "~5d. ~a~%" (car x) (cdr x)))
      (format t "~%"))))
```

defun skip-to-endif

[initial-substring p637]
 [preparseReadLine p89]
 [preparseReadLine1 p90]
 [skip-to-endif p556]

— defun skip-to-endif —

```
(defun skip-to-endif (x)
  (let (line ind tmp1)
    (setq tmp1 (preparseReadLine1))
    (setq ind (car tmp1))
    (setq line (cdr tmp1))
    (cond
      ((not (stringp line)) (cons ind line))
      ((initial-substring line ")endif") (preparseReadLine x))
      ((initial-substring line ")fin") (cons ind nil))
      (t (skip-to-endif x)))))
```

Chapter 15

The Compiler

defvar \$newConlist

A list of new constructors discovered during compile. These are used in a call to `extendLocalLibdb` when a user compiles new local code.

— **initvars** —

```
(defvar |$newConlist| nil
  "A list of new constructors discovered during compile ")
```

—————

15.1 Compiling EQ.spad

Given the top level command:

```
)co EQ
```

The default call chain looks like:

```
1> (|compiler| ...)
2> (|compileSpad2Cmd| ...)
   Compiling AXIOM source code from file /tmp/A.spad using old system
   compiler.
3> (|compilerDoit| ...)
4> (|/RQ,LIB|)
5> (/RF-1 ...)
6> (SPAD ...)
   AXSERV abbreviates package AxiomServer
7> (S-PROCESS ...)
```

```

8> (|compTopLevel| ...)
9> (|compOrCroak| ...)
10> (|compOrCroak1| ...)
11> (|comp| ...)
12> (|compNoStacking| ...)
13> (|comp2| ...)
14> (|comp3| ...)
15> (|compExpression| ...)
* 16> (|compWhere| ...)
17> (|comp| ...)
18> (|compNoStacking| ...)
19> (|comp2| ...)
20> (|comp3| ...)
21> (|compExpression| ...)
22> (|compSeq| ...)
23> (|compSeq1| ...)
24> (|compSeqItem| ...)
25> (|comp| ...)
26> (|compNoStacking| ...)
27> (|comp2| ...)
28> (|comp3| ...)
29> (|compExpression| ...)
<29 (|compExpression| ...)
<28 (|comp3| ...)
<27 (|comp2| ...)
<26 (|compNoStacking| ...)
<25 (|comp| ...)
<24 (|compSeqItem| ...)
24> (|compSeqItem| ...)
25> (|comp| ...)
26> (|compNoStacking| ...)
27> (|comp2| ...)
28> (|comp3| ...)
29> (|compExpression| ...)
30> (|compExit| ...)
31> (|comp| ...)
32> (|compNoStacking| ...)
33> (|comp2| ...)
34> (|comp3| ...)
35> (|compExpression| ...)
<35 (|compExpression| ...)
<34 (|comp3| ...)
<33 (|comp2| ...)
<32 (|compNoStacking| ...)
<31 (|comp| ...)
31> (|modifyModeStack| ...)
<31 (|modifyModeStack| ...)
<30 (|compExit| ...)
<29 (|compExpression| ...)
<28 (|comp3| ...)

```

```

    <27 (|comp2| ...)
    <26 (|compNoStacking| ...)
    <25 (|comp| ...)
    <24 (|compSeqItem| ...)
    24> (|replaceExitEtc| ...)
    25> (|replaceExitEtc,fn| ...)
    26> (|replaceExitEtc| ...)
    27> (|replaceExitEtc,fn| ...)
    28> (|replaceExitEtc| ...)
    29> (|replaceExitEtc,fn| ...)
    <29 (|replaceExitEtc,fn| ...)
    <28 (|replaceExitEtc| ...)
    28> (|replaceExitEtc| ...)
    29> (|replaceExitEtc,fn| ...)
    <29 (|replaceExitEtc,fn| ...)
    <28 (|replaceExitEtc| ...)
    <27 (|replaceExitEtc,fn| ...)
    <26 (|replaceExitEtc| ...)
    26> (|replaceExitEtc| ...)
    27> (|replaceExitEtc,fn| ...)
    28> (|replaceExitEtc| ...)
    29> (|replaceExitEtc,fn| ...)
    30> (|replaceExitEtc| ...)
    31> (|replaceExitEtc,fn| ...)
    32> (|replaceExitEtc| ...)
    33> (|replaceExitEtc,fn| ...)
    <33 (|replaceExitEtc,fn| ...)
    <32 (|replaceExitEtc| ...)
    32> (|replaceExitEtc| ...)
    33> (|replaceExitEtc,fn| ...)
    <33 (|replaceExitEtc,fn| ...)
    <32 (|replaceExitEtc| ...)
    <31 (|replaceExitEtc,fn| ...)
    <30 (|replaceExitEtc| ...)
    30> (|convertOrCroak| ...)
    31> (|convert| ...)
    <31 (|convert| ...)
    <30 (|convertOrCroak| ...)
    <29 (|replaceExitEtc,fn| ...)
    <28 (|replaceExitEtc| ...)
    28> (|replaceExitEtc| ...)
    29> (|replaceExitEtc,fn| ...)
    <29 (|replaceExitEtc,fn| ...)
    <28 (|replaceExitEtc| ...)
    <27 (|replaceExitEtc,fn| ...)
    <26 (|replaceExitEtc| ...)
    <25 (|replaceExitEtc,fn| ...)
    <24 (|replaceExitEtc| ...)
    <23 (|compSeq1| ...)
    <22 (|compSeq| ...)

```

```

    <21 (|compExpression| ...)
    <20 (|comp3| ...)
    <19 (|comp2| ...)
    <18 (|compNoStacking| ...)
    <17 (|comp| ...)
    17> (|comp| ...)
    18> (|compNoStacking| ...)
    19> (|comp2| ...)
    20> (|comp3| ...)
    21> (|compExpression| ...)
    22> (|comp| ...)
    23> (|compNoStacking| ...)
    24> (|comp2| ...)
    25> (|comp3| ...)
    26> (|compColon| ...)
    <26 (|compColon| ...)
    <25 (|comp3| ...)
    <24 (|comp2| ...)
    <23 (|compNoStacking| ...)
    <22 (|comp| ...)

```

In order to explain the compiler we will walk through the compilation of EQ.spad, which handles equations as mathematical objects. We start the system. Most of the structure in Axiom are circular so we have to the **print-cycle** to true.

```
root@spiff:/tmp# axiom -nox
```

```
(1) -> )lisp (setq *print-circle* t)
```

```
Value = T
```

We trace the function we find interesting:

```
(1) -> )lisp (trace |compiler|)
```

```
Value = (|compiler|)
```

15.2 The top level compiler command

This is the graph of the functions used for compDefine. The syntax is a graphviz dot file. To generate this graph as a JPEG file, type:

```
tangle v9compDefine.dot bookvol9.pamphlet >v9compdefine.dot
dot -Tjpg v9compiler.dot >v9compiler.jpg
```

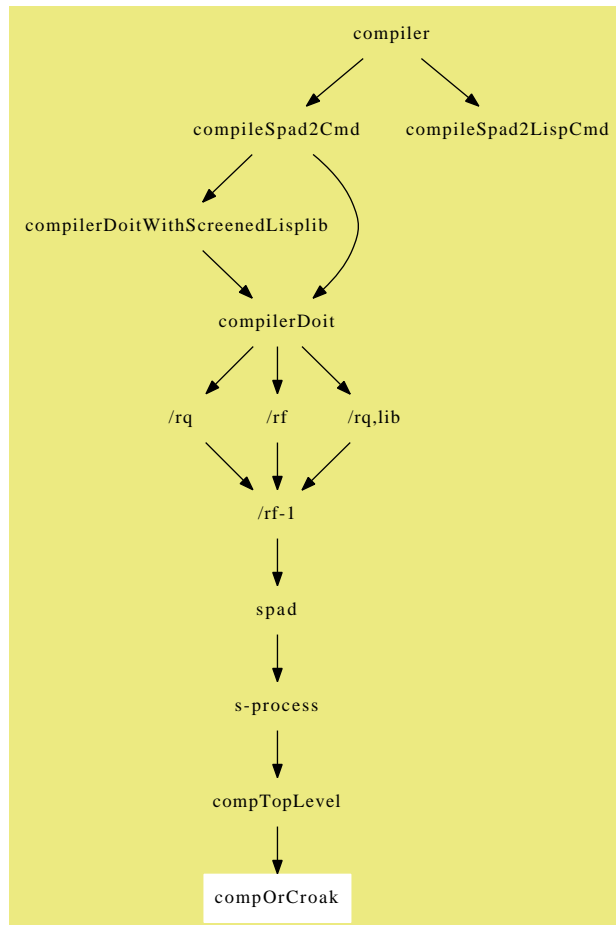
```

digraph pic {
    fontsize=10;
    bgcolor="#ECEA81";
    node [shape=box, color=white, style=filled];

    "compiler"                [color="#ECEA81"]
    "compileSpad2Cmd"         [color="#ECEA81"]
    "compileSpad2LispCmd"     [color="#ECEA81"]
    "compilerDoitWithScreenedLisplib" [color="#ECEA81"]
    "compilerDoit"            [color="#ECEA81"]
    "/rq"                    [color="#ECEA81"]
    "/rf"                    [color="#ECEA81"]
    "/rf-1"                 [color="#ECEA81"]
    "/rq,lib"               [color="#ECEA81"]
    "spad"                  [color="#ECEA81"]
    "s-process"             [color="#ECEA81"]
    "compTopLevel"          [color="#ECEA81"]
    "compOrCroak"           [color="#FFFFFF"]

    "compiler" -> "compileSpad2Cmd"
    "compiler" -> "compileSpad2LispCmd"
    "compileSpad2Cmd" -> "compilerDoitWithScreenedLisplib"
    "compileSpad2Cmd" -> "compilerDoit"
    "compilerDoitWithScreenedLisplib" -> "compilerDoit"
    "compilerDoit" -> "/rq"
    "compilerDoit" -> "/rf"
    "compilerDoit" -> "/rq,lib"
    "/rq" -> "/rf-1"
    "/rf" -> "/rf-1"
    "/rq,lib" -> "/rf-1"
    "/rf-1" -> "spad"
    "spad" -> "s-process"
    "s-process" -> "compTopLevel"
    "compTopLevel" -> "compOrCroak"
}

```



defun compiler

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. The `pathnameType` function eventually returns the fact that this is a `spad` source file. Once that is known we call the `compileSpad2Cmd` function with a list containing the full pathname as a string.

```
1> (|compiler| (EQ))
2> (|pathname| (EQ))
<2 (|pathname| #p"EQ")
```

```

2> (|pathnameType| #p"EQ")
3> (|pathname| #p"EQ")
<3 (|pathname| #p"EQ")
<2 (|pathnameType| NIL)
2> (|pathnameType| "/tmp/EQ.spad")
3> (|pathname| "/tmp/EQ.spad")
<3 (|pathname| #p"/tmp/EQ.spad")
<2 (|pathnameType| "spad")
2> (|pathnameType| "/tmp/EQ.spad")
3> (|pathname| "/tmp/EQ.spad")
<3 (|pathname| #p"/tmp/EQ.spad")
<2 (|pathnameType| "spad")
2> (|pathnameType| "/tmp/EQ.spad")
3> (|pathname| "/tmp/EQ.spad")
<3 (|pathname| #p"/tmp/EQ.spad")
<2 (|pathnameType| "spad")
2> (|compileSpad2Cmd| ("/tmp/EQ.spad"))

[compiler helpSpad2Cmd (vol5)]
[compiler selectOptionLC (vol5)]
[compiler pathname (vol5)]
[compiler mergePathnames (vol5)]
[compiler pathnameType (vol5)]
[compiler namestring (vol5)]
[throwKeyedMsg p??]
[findfile p??]
[compileSpad2Cmd p565]
[compileSpadLispCmd p568]
[$newConlist p557]
[$options p??]
[/editfile p??]

```

— defun compiler —

```

(defun |compiler| (args)
  "The top level compiler command"
  (let (|$newConlist| optlist optname optargs havenew haveold ef afl
        pathname pathtype)
    (declare (special |$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 pathname (|pathname| args))
   (setq pathtype (|pathnameType| pathname))
   (cond
     ((or haveold (string= pathtype "spad"))
      (if (null (setq af1 ($findfile pathname '(|spad|))))
          (|throwKeyedMsg| 's2il0003 (cons (namestring pathname) nil))
          (|compileSpad2Cmd| (list af1))))
     ((string= pathtype "nrllib")
      (if (null (setq af1 ($findfile pathname '(|nrllib|))))
          (|throwKeyedMsg| 'S2IL0003 (cons (namestring pathname) nil))
          (|compileSpadLispCmd| (list af1))))
    (t
     (setq af1 ($findfile pathname '(|spad|)))
     (cond
       ((and af1 (string= (|pathnameType| af1) "spad"))
        (|compileSpad2Cmd| (list af1)))
       (t
        (setq ef (|pathname| /editfile))
        (setq ef (|mergePathnames| pathname ef))
        (cond
          ((equal ef pathname) (|throwKeyedMsg| 's2iz0039 nil))
          (t
           (setq pathname ef)
           (cond
             ((string= (|pathnameType| pathname) "spad")
              (|compileSpad2Cmd| args))
             (t
              (setq af1 ($findfile pathname '(|spad|)))
              (cond
                ((and af1 (string= (|pathnameType| af1) "spad"))
                 (|compileSpad2Cmd| (cons af1 nil)))
                (t (|throwKeyedMsg| 's2iz0039 nil))))))))))))))

```

defun compileSpad2Cmd

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| |copyright|
<1 (|selectOptionLC| |compiler|)
1> (|selectOptionLC| |compiler| (|abbreviations| |boot| |browse| |cd| |clear| |close| |compiler| |copyright|
<1 (|selectOptionLC| |compiler|)
1> (|pathname| (EQ))
<1 (|pathname| #p"EQ")
1> (|pathnameType| #p"EQ")
2> (|pathname| #p"EQ")
<2 (|pathname| #p"EQ")
<1 (|pathnameType| NIL)
1> (|pathnameType| "/tmp/EQ.spad")
2> (|pathname| "/tmp/EQ.spad")
<2 (|pathname| #p"/tmp/EQ.spad")
<1 (|pathnameType| "spad")
1> (|pathnameType| "/tmp/EQ.spad")
2> (|pathname| "/tmp/EQ.spad")
<2 (|pathname| #p"/tmp/EQ.spad")
<1 (|pathnameType| "spad")
1> (|pathnameType| "/tmp/EQ.spad")
2> (|pathname| "/tmp/EQ.spad")
<2 (|pathname| #p"/tmp/EQ.spad")
<1 (|pathnameType| "spad")
1> (|compileSpad2Cmd| ("/tmp/EQ.spad"))
2> (|pathname| ("/tmp/EQ.spad"))
<2 (|pathname| #p"/tmp/EQ.spad")
2> (|pathnameType| #p"/tmp/EQ.spad")
3> (|pathname| #p"/tmp/EQ.spad")
<3 (|pathname| #p"/tmp/EQ.spad")
<2 (|pathnameType| "spad")
2> (|updateSourceFiles| #p"/tmp/EQ.spad")
3> (|pathname| #p"/tmp/EQ.spad")
<3 (|pathname| #p"/tmp/EQ.spad")
3> (|pathname| #p"/tmp/EQ.spad")
<3 (|pathname| #p"/tmp/EQ.spad")
3> (|pathnameType| #p"/tmp/EQ.spad")
4> (|pathname| #p"/tmp/EQ.spad")
<4 (|pathname| #p"/tmp/EQ.spad")
<3 (|pathnameType| "spad")
```

```

3> (|pathname| ("EQ" "spad" "*"))
<3 (|pathname| #p"EQ.spad")
3> (|pathnameType| #p"EQ.spad")
  4> (|pathname| #p"EQ.spad")
  <4 (|pathname| #p"EQ.spad")
  <3 (|pathnameType| "spad")
<2 (|updateSourceFiles| #p"EQ.spad")
2> (|namestring| ("/tmp/EQ.spad"))
  3> (|pathname| ("/tmp/EQ.spad"))
  <3 (|pathname| #p"/tmp/EQ.spad")
  <2 (|namestring| "/tmp/EQ.spad")
Compiling AXIOM source code from file /tmp/EQ.spad using old system
compiler.

```

Again we find a lot of redundant work. We finally end up calling **compilerDoit** with a constructed argument list:

```

2> (|compilerDoit| NIL (|rq| |lib|))

[compileSpad2Cmd pathname (vol5)]
[compileSpad2Cmd pathnameType (vol5)]
[compileSpad2Cmd namestring (vol5)]
[compileSpad2Cmd updateSourceFiles (vol5)]
[compileSpad2Cmd selectOptionLC (vol5)]
[compileSpad2Cmd terminateSystemCommand (vol5)]
[throwKeyedMsg p??]
[compileSpad2Cmd sayKeyedMsg (vol5)]
[error p??]
[strconc p??]
[object2String p??]
[browserAutoloadOnceTrigger p??]
[spad2AsTranslatorAutoloadOnceTrigger p??]
[compilerDoitWithScreenedLisplib p??]
[compilerDoit p570]
[extendLocalLibdb p477]
[spadPrompt p??]
[$newComp p??]
[$scanIfTrue p??]
[$compileOnlyCertainItems p??]
[$f p??]
[$m p??]
[$QuickLet p??]
[$QuickCode p??]
[$sourceFileTypes p??]
[$InteractiveMode p??]
[$options p??]
[$newConlist p557]

```

[/editfile p??]

— defun compileSpad2Cmd —

```
(defun |compileSpad2Cmd| (args)
  (let (|$newComp| |$scanIfTrue|
        |$compileOnlyCertainItems| |$f| |$m| |$QuickLet| |$QuickCode|
        |$sourceFileTypes| |$InteractiveMode| path optlist fun optname
        optargs fullopt constructor)
    (declare (special |$newComp| |$scanIfTrue|
                      |$compileOnlyCertainItems| |$f| |$m| |$QuickLet| |$QuickCode|
                      |$sourceFileTypes| |$InteractiveMode| /editfile |$options|
                      |$newConlist|))
    (setq path (|pathname| args))
    (cond
     ((not (string= (|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| |nbreak| |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 (not (eq (elt fun 0) '|c|)) (setelt fun 0 '|rq|)))
         (|noquiet| (when (not (eq (elt fun 0) '|c|)) (setelt fun 0 '|rf|)))
         (|nbreak| (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| 's2iz0038 (list ")constructor"))
              (setq |$compileOnlyCertainItems| 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|))))

```

defun compileSpadLispCmd

```

[compileSpadLispCmd pathname (vol5)]
[compileSpadLispCmd pathnameType (vol5)]
[compileSpadLispCmd selectOptionLC (vol5)]
[compileSpadLispCmd namestring (vol5)]
[compileSpadLispCmd terminateSystemCommand (vol5)]
[compileSpadLispCmd fnameMake (vol5)]
[compileSpadLispCmd pathnameDirectory (vol5)]
[compileSpadLispCmd pathnameName (vol5)]
[compileSpadLispCmd fnameReadable? (vol5)]
[compileSpadLispCmd localdatabase (vol5)]
[throwKeyedMsg p??]
[object2String p??]
[compileSpadLispCmd sayKeyedMsg (vol5)]
[recompile-lib-file-if-necessary p627]
[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|))))))

```

compilerDoitWithScreenedLisplib

```

compilerDoitWithScreenedLisplib [embed p??]
[rewrite p??]
[compilerDoit p570]
[unembed p??]
[$saveableItems p??]
[$libFile p??]

```

— **defun compilerDoitWithScreenedLisplib** —

```
(defun |compilerDoitWithScreenedLisplib| (constructor fun)
  (declare (special |$saveableItems| |$libFile|))
  (embed 'rwrite
    '(lambda (key value stream)
      (cond
        ((and (eq stream |$libFile|)
              (not (member key |$saveableItems|)))
         value)
        ((not nil) (rwrite key value stream))))))
  (unwind-protect
    (|compilerDoit| constructor fun)
    (unembed 'rwrite)))
```

defun compilerDoit

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) ->

[compilerDoit /rq (vol5)]
[compilerDoit /rf (vol5)]
[compilerDoit member (vol5)]
[sayBrightly p??]
[opOf p??]
[/RQ,LIB p572]
[$byConstructors p629]
[$constructorsSeen p629]
```

— **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 |$byConstructors| (loop for x in constructor collect (|op0f| x)))
    (|/RQ,LIB|)
    (dolist (x |$byConstructors|)
      (unless (|member| x |$constructorsSeen|)
        (|sayBrightly| '(">>> Warning " |%b| ,x |%d| " was not found"))))))))
```

defun /rq

Compile with quiet output [rf-1 p572]
 [echo-meta p??]

— defun /rq —

```
(defun /rq (&rest foo &aux (echo-meta nil))
  (declare (special Echo-Meta) (ignore foo))
  (/rf-1 nil))
```

defun /rf

Compile with noisy output [rf-1 p572]
 [echo-meta p??]

— defun /rf —

```
(defun /rf (&rest foo &aux (echo-meta t))
  (declare (special echo-meta) (ignore foo))
  (/rf-1 nil))
```

defun /RQ,LIB

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)

[/rf-1 p572]
[/RQ,LIB echo-meta (vol5)]
[$lisplib p??]
```

— **defun /RQ,LIB** —

```
(defun |/RQ,LIB| (&rest foo &aux (echo-meta nil) ($lisplib t))
  (declare (special echo-meta $lisplib) (ignore foo))
  (/rf-1 nil))
```

—————

defun /rf-1

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)

[/rf-1 makeInputFilename (vol5)]
[ncINTERPFILE p627]
calls/rf-1spad [/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)))))
```

defun spad

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.nrlib/EQ.fn">)
3> (OPEN #p"/tmp/EQ.nrlib/EQ.o" :DIRECTION :OUTPUT :IF-EXISTS :APPEND)
<3 (OPEN #<output stream "/tmp/EQ.nrlib/EQ.o">)
3> (OPEN #p"/tmp/EQ.nrlib/EQ.data")
<3 (OPEN #<input stream "/tmp/EQ.nrlib/EQ.data">)
3> (OPEN "/tmp/EQ.nrlib/index.kaf")
<3 (OPEN #<input stream "/tmp/EQ.nrlib/index.kaf">)
<2 (S-PROCESS NIL)
<1 (SPAD T)
1> (OPEN "temp.text" :DIRECTION :OUTPUT)
<1 (OPEN #<output stream "temp.text">)
1> (OPEN "libdb.text")
<1 (OPEN #<input stream "libdb.text">)
1> (OPEN "temp.text")
<1 (OPEN #<input stream "temp.text">)
1> (OPEN "libdb.text" :DIRECTION :OUTPUT)
<1 (OPEN #<output stream "libdb.text">)

```

The major steps in this process involve the **preparse** function. (See book volume 5 for more details). The **preparse** function returns a list of pairs of the form: ((linenumber . linestring) (linenumber . linestring)) For instance, for the file `EQ.spad`, we get:

```

<2 (PREPARSE (
(19 . "Equation(S: Type): public == private where")
(20 . " (Ex ==> OutputForm;")
(21 . "   public ==> Type with")
(22 . "   (\\"=\\": (S, S) -> $;")
...[skip]...
(202 . "           inv eq == [inv lhs eq, inv rhs eq]);")
(203 . "       if S has ExpressionSpace then")
(204 . "           subst(eq1,eq2) ==")
(205 . "               (eq3 := eq2 pretend Equation S;")
(206 . "               [subst(lhs eq1,eq3),subst(rhs eq1,eq3)])))))

```

```

[spad-reader p??]
[spad addBinding (vol5)]
[spad makeInitialModemapFrame (vol5)]
[spad init-boot/spad-reader (vol5)]
[initialize-preparse p76]
[preparse p80]
[PARSE-NewExpr p417]
[pop-stack-1 p550]
[s-process p576]
[ioclear p??]
[spad shut (vol5)]
[$noSubsumption p??]
[$InteractiveFrame p??]
[$InitialDomainsInScope p??]

```

— defun spad —

```
(defun spad (&optional (*spad-input-file* nil) (*spad-output-file* nil)
  &aux (*comp370-apply* #'print-defun)
  (*fileactq-apply* #'print-defun)
  ($spad t) ($boot nil) (optionlist nil) (*eof* nil)
  (file-closed nil) (/editfile *spad-input-file*)
  (|$noSubsumption| |$noSubsumption|) in-stream out-stream)

(declare (special echo-meta /editfile *comp370-apply* *eof* curoutstream
  file-closed |$noSubsumption| |$InteractiveFrame|
  |$InteractiveModel| optionlist
  boot-line-stack *fileactq-apply* $spad $boot))

;; only rebind |$InteractiveFrame| if compiling
(progv (if (not |$InteractiveModel|) '(|$InteractiveFrame|)
  (if (not |$InteractiveModel|)
    (list (|addBinding| '|$DomainsInScope|
      '((fluid . |true|))
      (|addBinding| '|$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-parse in-stream)
    (setq out-stream (if *spad-output-file*
      (open *spad-output-file* :direction :output
        *standard-output*))
    (when *spad-output-file*
      (format out-stream "~&;;; -*- Mode:Lisp; Package:Boot -*-~%~%")
      (print-package "BOOT"))
    (setq curoutstream out-stream)
```

```

(loop
  (if (or *eof* file-closed) (return nil))
  (catch 'spad_reader
    (if (setq boot-line-stack (preparse in-stream))
      (let ((line (cdar boot-line-stack)))
        (declare (special line))
        (|PARSE-NewExpr|)
        (let ((parseout (pop-stack-1)) )
          (when parseout
            (let ((*standard-output* out-stream))
              (s-process parseout))
            (format out-stream "~&"))))
      )))
  (ioclear in-stream out-stream)))
(if *spad-input-file* (shut in-stream))
(if *spad-output-file* (shut out-stream)))
t))

```

defun Interpreter interface to the compiler

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| (-> $ $)))
NIL)
(|if| (|has| S |ExpressionSpace|)
(|Signature| |subst| (-> (|,$ $ $ $)))
NIL)
))))
(==> |private|
(|add|
(|;|
(|;|
(|;|
(|;|
(|;|
(|;|
(|;|
(|;|
(|;|
(|;|
(|;|
(|;|
(|;|
(|;|
(|;|
(|;|
(|;|
(|:=| |Repl|
(|Record| (|,$ (|:| |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| |leqn2| (|List| (|Equation| S))))))
(|eval|
  (|,|
    (|eqn1| |rhs|)
    (|pretend| |leqn2| (|List| (|Equation| S))))))
NIL))
(|if| (|has| S |SetCategory|)
  (|;|
    (|;|
      (==
        (= |eq1| |eq2|)
        (|and|
          (@ (= (|eq1| |lhs|) (|eq2| |lhs|)) |Boolean|)
          (@ (= (|eq1| |rhs|) (|eq2| |rhs|)) |Boolean|)))
        (==
          (|::| (|coerce| (|::| |eqn| $)) |Ex|)
          (= (|::| (|eqn| |lhs|) |Ex|) (|::| (|eqn| |rhs|) |Ex|))))
        (==
          (|::| (|coerce| (|::| |eqn| $)) |Boolean|)
          (= (|eqn| |lhs|) (|eqn| |rhs|))))
      NIL))
(|if| (|has| S |AbelianSemiGroup|)
  (|;|
    (|;|
      (==
        (+ |eq1| |eq2|)
        (=
          (+ (|eq1| |lhs|) (|eq2| |lhs|))
          (+ (|eq1| |rhs|) (|eq2| |rhs|))))
        (== (+ |s| |eq2|) (+ (|construct| (|,| |s| |s|)) |eq2|)))
        (== (+ |eq1| |s|) (+ |eq1| (|construct| (|,| |s| |s|)))))
      NIL))
(|if| (|has| S |AbelianGroup|)
  (|;|
    (|;|
      (|;|
        (|;|
          (== (- |eq1|) (= (- (|lhs| |eq1|)) (- (|rhs| |eq1|))))
          (== (- |s| |eq2|) (- (|construct| (|,| |s| |s|)) |eq2|)))
          (== (- |eq1| |s|) (- |eq1| (|construct| (|,| |s| |s|)))))
          (== (|leftZero| |eq1|) (= 0 (- (|rhs| |eq1|) (|lhs| |eq1|))))
          (== (|rightZero| |eq1|) (= (- (|lhs| |eq1|) (|rhs| |eq1|)) 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|))))))
(==
  (* (|:| |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|))))))
  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))))))

```

```

[curstrm p??]
[def-rename p587]
[new2OldLisp p72]
[parseTransform p97]
[postTransform p365]
[displayPreCompilationErrors p544]
[prettyprint p??]
[s-process processInteractive (vol5)]
[compTopLevel p586]
[def-process p??]
[displaySemanticErrors p??]
[terpri p??]
[get-internal-run-time p??]
[$Index p??]
[$macroassoc p??]
[$newspad p??]
[$PolyMode p??]
[$EmptyMode p172]
[$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??]
[TranslateOnly p??]
[PrintOnly p??]
[currentLine p??]
[InteractiveFrame 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| ())
$top_level |$insideFunctorIfTrue| |$insideExpressionIfTrue|
|$insideCoerceInteractiveHardIfTrue| |$insideWhereIfTrue|
|$insideCategoryIfTrue| |$insideCapsuleFunctionIfTrue| |$form|
(|$DomainFrame| '((NIL)))
(|$e| |$EmptyEnvironment|)
(|$genFVar| 0)
(|$genSDVar| 0)
(|$VariableCount| 0)
(|$previousTime| (get-internal-run-time))
(|$LocalFrame| '((NIL)))
(curstrm curoutstream) |$s| |$x| |$m| u)
(declare (special |$Index| $macroassoc $newspad |$PolyMode| |$EmptyMode|
|$compUniquelyIfTrue| |$currentFunction| |$postStack| |$topOp|
|$semanticErrorStack| |$warningStack| |$exitMode| |$exitModeStack|
|$returnMode| |$leaveMode| |$leaveLevelStack| $top_level
|$insideFunctorIfTrue| |$insideExpressionIfTrue| | | | | | |
|$insideCoerceInteractiveHardIfTrue| |$insideWhereIfTrue|
|$insideCategoryIfTrue| |$insideCapsuleFunctionIfTrue| |$form|
|$DomainFrame| |$e| |$EmptyEnvironment| |$genFVar| |$genSDVar|
|$VariableCount| |$previousTime| |$LocalFrame|
curstrm |$s| |$x| |$m| curoutstream $traceflag |$Translation|
|$TranslateOnly| |$PrintOnly| |$currentLine| |$InteractiveFrame|))
(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))

```

defun compTopLevel

```
[compOrCroak p588]
[$NRTderivedTargetIfTrue p??]
[$killOptimizeIfTrue p??]
[$forceAdd p??]
[$compTimeSum p??]
[$resolveTimeSum p??]
[$packagesUsed p??]
[$envHashTable p??]
```

— defun compTopLevel —

```
(defun |compTopLevel| (form mode env)
  (let (|$NRTderivedTargetIfTrue| |$killOptimizeIfTrue| |$forceAdd|
        |$compTimeSum| |$resolveTimeSum| |$packagesUsed| |$envHashTable|
        t1 t2 t3 val newmode)
    (declare (special |$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 |$envHashTable| (make-hashtable 'equal))
    (dolist (u (car (car env)))
      (dolist (v (cdr u))
        (hput |$envHashTable| (cons (car u) (cons (car v) nil)) t)))
    (cond
      ((or (and (consp form) (eq (qfirst form) 'def))
           (and (consp form) (eq (qfirst form) '|where|)
                (progn
                  (setq t1 (qrest form))
                  (and (consp t1)
                       (progn
                        (setq t2 (qfirst t1))
                        (and (consp t2) (eq (qfirst t2) 'def)))))))
        (setq t3 (|compOrCroak| form mode env))
        (setq val (car t3))
        (setq newmode (second t3))
        (cons val (cons newmode (cons env nil))))
      (t (|compOrCroak| form mode env))))
```

defun print-defun

[is-console p555]
 [print-full p??]
 [vmlisp::optionlist p??]
 [\$PrettyPrint p??]

— defun print-defun —

```
(defun print-defun (name body)
  (let* ((sp (assoc 'vmlisp::compiler-output-stream vmlisp::optionlist))
        (st (if sp (cdr sp) *standard-output*)))
    (declare (special vmlisp::optionlist |$PrettyPrint|))
    (when (and (is-console st) (symbolp name) (fboundp name)
              (not (compiled-function-p (symbol-function name)))))
      (compile name))
    (when (or |$PrettyPrint| (not (is-console st)))
      (print-full body st) (force-output st))))
```

—————

defun def-rename

[def-rename p587]

— defun def-rename —

```
(defun def-rename (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-rename (first x)) (def-rename (cdr x)))))
    (x)))
```

—————

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.

defun compOrCroak

[compOrCroak1 p589]

— defun compOrCroak —

```
(defun |compOrCroak| (form mode env)
  (|compOrCroak1| form mode env nil nil))
```

This results in a call to the inner function with

```
(|compOrCroak1|
  (DEF (|CohenCategory|)
    ((|Category|))
    (NIL)
    (|Join|
      (|SetCategory|)
      (CATEGORY |package|
        (SIGNATURE |kind| ((|Boolean|) |CExpr|))
        (SIGNATURE |operand| (|CExpr| |CExpr| (|Integer|)))
```



```

(SIGNATURE |numberOfOperand| ((|Integer|) |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.

defun compOrCroak1

```

[comp p590]
[compOrCroak1,compactify p626]
[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 p588]

```

— defun compOrCroak1 —

```

(defun |compOrCroak1| (form mode env |$compStack| |$compErrorMessageStack|)
  (declare (special |$compStack| |$compErrorMessageStack|))
  (let (td errorMessage)
    (declare (special |$level| |$s| |$scanIfTrue| |$exitModeStack|))
    (cond
      ((setq td (catch '|compOrCroak| (|comp| form mode env))) td)
      (t
       (setq |$compStack|
              (cons (list form mode env |$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| mode env ))
  (t
    (|displaySemanticErrors|)
    (say "***** comp fails at level " |$level| " with expression: *****")
    (|displayComp| |$level|)
    (|userError| errorMessage))))))

```

defun comp

```

[compNoStacking p590]
[$compStack p??]
[$exitModeStack p??]

```

— defun comp —

```

(defun |comp| (form mode env)
  (let (td)
    (declare (special |$compStack| |$exitModeStack|))
    (if (setq td (|compNoStacking| form mode env))
      (setq |$compStack| nil)
      (push (list form mode env |$exitModeStack|) |$compStack|))
    td))

```

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 p591]
[compNoStacking1 p591]
[$compStack p??]
[$Representation p??]
[$EmptyMode p172]

```

— defun compNoStacking —

```
(defun |compNoStacking| (form mode env)
  (let (td)
    (declare (special |$compStack| |$Representation| |$EmptyMode|))
    (if (setq td (|comp2| form mode env))
      (if (and (equal mode |$EmptyMode|) (equal (second td) |$Representation|))
        (list (car td) '$ (third td))
        td)
      (|compNoStacking1| form mode env |$compStack|))))
```

defun compNoStacking1

```
[get p??]
[comp2 p591]
[$compStack p??]
```

— defun compNoStacking1 —

```
(defun |compNoStacking1| (form mode env |$compStack|)
  (declare (special |$compStack|))
  (let (u td)
    (if (setq u (|get| (if (eq mode '$) '|Rep| mode) '|value| env))
      (if (setq td (|comp2| form (car u) env))
        (list (car td) mode (third td))
        nil)
      nil)))
```

defun comp2

```
[comp3 p592]
[isDomainForm p344]
[isFunctor p249]
[insert p??]
[opOf p??]
[addDomain p248]
[$bootStrapMode p??]
[$packagesUsed p??]
[$lisplib p??]
```

— defun comp2 —

```
(defun |comp2| (form mode env)
  (let (tmp1)
    (declare (special |$bootStrapMode| |$packagesUsed| $lisplib))
    (when (setq tmp1 (|comp3| form mode env))
      (destructuring-bind (y mprime env) tmp1
        (when (and $lisplib (|isDomainForm| form env) (|isFunction| form))
          (setq |$packagesUsed| (|insert| (list (|opOf| form)) |$packagesUsed|)))
        ; isDomainForm test needed to prevent error while compiling Ring
        ; $bootStrapMode-test necessary for compiling Ring in $bootStrapMode
        (if (and (not (equal mode mprime))
                  (or |$bootStrapMode| (|isDomainForm| mprime env)))
            (list y mprime (|addDomain| mprime env))
            (list y mprime env))))))
```

defun comp3

```
[addDomain p248]
[compWithMappingMode p617]
[compAtom p597]
[getmode p??]
[applyMapping p593]
[compApply p595]
[compColon p290]
[compCoerce p358]
[stringPrefix? p??]
[comp3 pname (vol5)]
[compTypeOf p596]
[compExpression p135]
[comp3 member (vol5)]
[getDomainsInScope p250]
[$e p??]
[$insideCompTypeOf p??]
```

— defun comp3 —

```
(defun |comp3| (form mode |$e|)
  (declare (special |$e|))
  (let (env op ml u tt tmp1)
    (declare (special |$insideCompTypeOf|))
    (setq |$e| (|addDomain| mode |$e|))
    (setq env |$e|)
    (cond
      ((and (consp mode) (eq (qfirst mode) '|Mapping|))
        (|compWithMappingMode| form mode env))
```

```

((and (consp mode) (eq (qfirst mode) 'quote)
      (consp (qcdr mode)) (eq (qcddr mode) nil))
 (when (equal form (qcadr mode)) (list form mode |$e|)))
((stringp mode)
 (when (and (atom form)
            (or (equal mode form) (equal mode (princ-to-string form)))))
 (list mode mode env )))
((or (null form) (atom form)) (|compAtom| form mode env))
(t
 (setq op (car form))
 (cond
  ((and (progn
        (setq tmp1 (|getmode| op env))
        (and (consp tmp1)
              (eq (qfirst tmp1) '|Mapping|)
              (progn (setq ml (qrest tmp1)) t)))
        (setq u (|applyMapping| form mode env ml)))
   u)
  ((and (consp op) (eq (qfirst op) 'kappa)
        (consp (qcdr op)) (consp (qcddr op))
        (consp (qcdddr op)) (eq (qcdddr op) nil))
   (|compApply| (qcadr op) (qcaddr op) (qcaddr op) (cdr form) mode env))
  ((eq op '|:|) (|compColon| form mode env))
  ((eq op '|::|) (|compCoerce| form mode env))
  ((and (null (eq |$insideCompTypeOf| t))
        (|stringPrefix?| "TypeOf" (pname op)))
   (|compTypeOf| form mode env))
  (t
   (setq tt (|compExpression| form mode env))
   (cond
    ((and (consp tt) (consp (qcdr tt)) (consp (qcddr tt))
          (eq (qcdddr tt) nil)
          (null (|member| (qcadr tt) (|getDomainsInScope| (qcaddr tt)))))
     (list (qcar tt) (qcadr tt) (|addDomain| (qcadr tt) (qcaddr tt))))
    (t tt))))))

```

defun applyMapping

```

[isCategoryForm p??]
[sublis p??]
[comp p590]
[convert p600]
[member p??]
[get p??]
[getAbbreviation p298]

```

```
[encodeItem p181]
[$FormalMapVariableList p266]
[$form p??]
[$op p??]
[$prefix p??]
[$formalArgList p??]
```

— defun applyMapping —

```
(defun |applyMapping| (t0 m e ml)
  (prog (op argl mlp temp1 arglp nprefix opp form pairlis)
    (declare (special |$FormalMapVariableList| |$form| |$op| |$prefix|
                      |$formalArgList|))
    (return
     (progn
      (setq op (car t0))
      (setq argl (cdr t0))
      (cond
       ((not (eql (|#| argl) (1- (|#| ml)))) nil)
       ((|isCategoryForm| (car ml) e)
        (setq pairlis
         (loop for a in argl for v in |$FormalMapVariableList|
              collect (cons v a)))
        (setq mlp (sublis pairlis ml))
        (setq arglp
         (loop for x in argl for mp in (rest mlp)
              collect (car
                       (progn
                        (setq temp1 (or (|comp| x mp e) (return '|failed|)))
                        (setq e (caddr temp1))
                        temp1))))
        (when (eq arglp '|failed|) (return nil))
        (setq form (cons op arglp))
        (|convert| (list form (car mlp) e) m))
       (t
        (setq arglp
         (loop for x in argl for mp in (rest ml)
              collect (car
                       (progn
                        (setq temp1 (or (|comp| x mp e) (return '|failed|)))
                        (setq e (caddr temp1))
                        temp1))))
        (when (eq arglp '|failed|) (return nil))
        (setq form
         (cond
          ((and (null (|member| op |$formalArgList|))
               (atom op)
               (null (|get| op '|value| e)))
           (setq nprefix
```

```

      (or |$prefix| (|getAbbreviation| |$op| (|#| (cdr |$form|))))))
    (setq opp
      (intern (strconc
        (|encodeItem| nprefix) '|;| (|encodeItem| op))))
    (cons opp (append arglp (list '$))))
  (t
    (cons '|call| (cons (list '|applyFun| op) arglp))))
  (setq pairlis
    (loop for a in arglp for v in |$FormalMapVariableList|
      collect (cons v a)))
  (|convert| (list form (sublis pairlis (car ml)) e) m))))))

```

defun compApply

[\[comp p590\]](#)
[\[Pair p??\]](#)
[\[removeEnv p??\]](#)
[\[resolve p361\]](#)
[\[AddContour p??\]](#)
[\[\\$EmptyMode p172\]](#)

— defun compApply —

```

(defun |compApply| (sig varl body argl m e)
  (let (temp1 argtl contour code mq bodyq)
    (declare (special |$EmptyMode|))
    (setq argtl
      (loop for x in argl
        collect (progn
          (setq temp1 (|comp| x |$EmptyMode| e))
          (setq e (caddr temp1))
          temp1)))
    (setq contour
      (loop for x in varl
        for mq in (cdr sig)
        for a in argl
        collect
          (|Pair| x
            (list
              (list '|mode| mq)
              (list '|value| (|removeEnv| (|comp| a mq e)))))))
    (setq code
      (cons (list 'lambda varl bodyq)
        (loop for tt in argtl
          collect (car tt))))

```

```
(setq mq (|resolve| m (car sig)))
(setq bodyq (car (|comp| body mq (|addContour| contour e))))
(list code mq e)))
```

defun compTypeOf

```
[eqsubstlist p??]
[get p??]
[put p??]
[comp3 p592]
[$insideCompTypeOf p??]
[$FormalMapVariableList p266]
```

— defun compTypeOf —

```
(defun |compTypeOf| (form mode env)
  (let (|$insideCompTypeOf| op argl newModemap)
    (declare (special |$insideCompTypeOf| |$FormalMapVariableList|))
    (setq op (car form))
    (setq argl (cdr form))
    (setq |$insideCompTypeOf| t)
    (setq newModemap
      (eqsubstlist argl |$FormalMapVariableList| (|get| op '|modemap| env)))
    (setq env (|put| op '|modemap| newModemap env))
    (|comp3| form mode env)))
```

defun compColonInside

```
[addDomain p248]
[comp p590]
[coerce p351]
[stackWarning p??]
[opOf p??]
[stackSemanticError p??]
[$newCompilerUnionFlag p??]
[$EmptyMode p172]
```

— defun compColonInside —

```
(defun |compColonInside| (form mode env mprime)
```



```

(let (mpp warningMessage td tprime)
  (declare (special |$newCompilerUnionFlag| |$EmptyMode|))
  (setq env (|addDomain| mprime env))
  (when (setq td (|comp| form |$EmptyMode| env))
    (cond
      ((equal (setq mpp (second td)) mprime)
        (setq warningMessage
          (list '|:| mprime '| -- should replace by @|))))
      (setq td (list (car td) mprime (third td)))
      (when (setq tprime (|coerce| td mode))
        (cond
          (warningMessage (|stackWarning| warningMessage))
          ((and |$newCompilerUnionFlag| (eq (|opOf| mpp) '|Union|))
            (setq tprime
              (|stackSemanticError|
                (list '|cannot pretend | form '| of mode | mpp '| to mode | mprime )
                  nil))))
          (t
            (|stackWarning|
              (list '|:| mprime '| -- should replace by pretend|))))
          tprime))))

```

defun compAtom

```

[compAtomWithModemap p598]
[get p??]
[modeIsAggregateOf p??]
[compList p602]
[compVector p349]
[convert p600]
[isSymbol p??]
[compSymbol p600]
[primitiveType p600]
[primitiveType p600]
[$Expression p??]

```

— defun compAtom —

```

(defun |compAtom| (form mode env)
  (prog (tmp1 tmp2 r td tt)
    (declare (special |$Expression|))
    (return
      (cond
        ((setq td
          (|compAtomWithModemap| form mode env (|get| form '|modemap| env))) td)

```

```

((eq form '|nil|)
 (setq td
  (cond
   ((progn
    (setq tmp1 (|modeIsAggregateOf| '|List| mode env))
    (and (consp tmp1)
     (progn
      (setq tmp2 (qrest tmp1))
      (and (consp tmp2)
       (eq (qrest tmp2) nil)
       (progn
        (setq r (qfirst tmp2)) t))))))
    (|compList| form (list '|List| r) env))
   ((progn
    (setq tmp1 (|modeIsAggregateOf| '|Vector| mode env))
    (and (consp tmp1)
     (progn
      (setq tmp2 (qrest tmp1))
      (and (consp tmp2) (eq (qrest tmp2) nil)
       (progn
        (setq r (qfirst tmp2)) t))))))
    (|compVector| form (list '|Vector| r) env))))
 (when td (|convert| td mode)))
(t
 (setq tt
  (cond
   ((|isSymbol| form) (or (|compSymbol| form mode env) (return nil)))
   ((and (equal mode |$Expression|)
    (|primitiveType| form)) (list form mode env ))
   ((stringp form) (list form form env ))
   (t (list form (or (|primitiveType| form) (return nil)) env )))
  (|convert| tt mode))))

```

defun compAtomWithModemap

[transImplementation p599]

[modeEqual p362]

[convert p600]

[\$NoValueMode p172]

— defun compAtomWithModemap —

```

(defun |compAtomWithModemap| (x m env v)
  (let (tt transimp y)
    (declare (special |$NoValueMode|))

```

```

(cond
  ((setq transimp
    (loop for map in v
      when ; map is [[.,target],[.,fn]]
        (and (consp map) (consp (qcar map)) (consp (qcadr map))
          (eq (qcddar map) nil)
          (consp (qcdr map)) (eq (qcddr map) nil)
          (consp (qcadr map)) (consp (qcddadr map))
          (eq (qcddadr map) nil))
      collect
        (list (|transImplementation| x map (qcadadr map)) (qcadar map) env)))
  (cond
    ((setq tt
      (let (result)
        (loop for item in transimp
          when (|modeEqual| m (cadr item))
            do (setq result (or result item)))
        result))
      tt)
    ((eq 1 (|#| (setq transimp
      (loop for ta in transimp
        when (setq y (|convert| ta m))
          collect y))))
      (car transimp))
    ((and (< 0 (|#| transimp)) (equal m |$NoValueMode|))
      (car transimp))
    (t nil))))))

```

defun transImplementation

[genDeltaEntry p??]

— defun transImplementation —

```

(defun |transImplementation| (op map fn)
  (setq fn (|genDeltaEntry| (cons op map)))
  (if (and (consp fn) (eq (qcar fn) 'xlam))
    (cons fn nil)
    (cons '|call| (cons fn nil))))

```

defun convert

[resolve p361]
 [coerce p351]

— defun convert —

```
(defun |convert| (td mode)
  (let (res)
    (when (setq res (|resolve| (second td) mode))
      (|coerce| td res))))
```

—————

defun primitiveType

[\$DoubleFloat p??]
 [\$NegativeInteger p??]
 [\$PositiveInteger p??]
 [\$NonNegativeInteger p??]
 [\$String p345]
 [\$EmptyMode p172]

— defun primitiveType —

```
(defun |primitiveType| (form)
  (declare (special |$DoubleFloat| |$NegativeInteger| |$PositiveInteger|
                    |$NonNegativeInteger| |$String| |$EmptyMode|))
  (cond
    ((null form) |$EmptyMode|)
    ((stringp form) |$String|)
    ((integerp form)
     (cond
       ((= form 0) |$NonNegativeInteger|)
       ((> form 0) |$PositiveInteger|)
       (t |$NegativeInteger|)))
    ((floatp form) |$DoubleFloat|)
    (t nil)))
```

—————

defun compSymbol

[isFluid p??]
 [getmode p??]

```

[get p??]
[NRTgetLocalIndex p211]
[compSymbol member (vol5)]
[isFunction p??]
[errorRef p??]
[stackMessage p??]
[$Symbol p??]
[$Expression p??]
[$FormalMapVariableList p266]
[$compForModeIfTrue p??]
[$formalArgList p??]
[$NoValueMode p172]
[$functorLocalParameters p??]
[$Boolean p??]
[$NoValue p??]

```

— defun compSymbol —

```

(defun |compSymbol| (form mode env)
  (let (v mprime newmode)
    (declare (special |$Symbol| |$Expression| |$FormalMapVariableList|
                      |$compForModeIfTrue| |$formalArgList| |$NoValueMode|
                      |$functorLocalParameters| |$Boolean| |$NoValue|))
    (cond
      ((eq form '|$NoValue|) (list '|$NoValue| |$NoValueMode| env ))
      ((|isFluid| form)
       (setq newmode (|getmode| form env))
       (when newmode (list form (|getmode| form env) env)))
      ((eq form '|true|) (list '(quote t) |$Boolean| env ))
      ((eq form '|false|) (list nil |$Boolean| env ))
      ((or (equal form mode)
            (|get| form '|isLiteral| env)) (list (list 'quote form) form env))
      ((setq v (|get| form '|value| env))
       (cond
         ((member form |$functorLocalParameters|)
          ; s will be replaced by an ELT form in beforeCompile
          (|NRTgetLocalIndex| form)
          (list form (second v) env))
         (t
          ; form has been SETQd
          (list form (second v) env))))
      ((setq mprime (|getmode| form env))
       (cond
         ((and (null (|member| form |$formalArgList|))
                (null (member form |$FormalMapVariableList|))
                (null (|isFunction| form env))
                (null (eq |$compForModeIfTrue| t))))
          (|errorRef| form)))

```

```

(list form mprime env ))
((member form |$FormalMapVariableList|)
 (|stackMessage| (list '|no mode found for| form )))
((or (equal mode |$Expression|) (equal mode |$Symbol|))
 (list (list 'quote form) mode env ))
((null (|isFunction| form env)) (|errorRef| form))))

```

defun compList

[comp p590]

— defun compList —

```

(defun |compList| (form mode env)
  (let (tmp1 tmp2 t0 failed (newmode (second mode)))
    (if (null form)
      (list nil mode env)
      (progn
        (setq t0
          (do ((t3 form (cdr t3)) (x nil))
              ((or (atom t3) failed) (unless failed (nreverse0 tmp2)))
            (setq x (car t3))
            (if (setq tmp1 (|comp| x newmode env))
              (progn
                (setq newmode (second tmp1))
                (setq env (third tmp1))
                (push tmp1 tmp2)
                (setq failed t))))
          (unless failed
            (cons
              (cons 'list (loop for texpr in t0 collect (car texpr)))
              (list (list '|List| newmode) env)))))))

```

defun compForm

[compForm1 p603]

[compArgumentsAndTryAgain p616]

[stackMessageIfNone p??]

— defun compForm —

```
(defun |compForm| (form mode env)
  (cond
    ((|compForm1| form mode env))
    ((|compArgumentsAndTryAgain| form mode env))
    (t (|stackMessageIfNone| (list '|cannot compile| '|%b| form '|%d| )))))
```

defun compForm1

This function is called if a keyword is found in a compile form but there is no handler listed for the form (See 6). [length p??]

[outputComp p343]
 [compOrCroak p588]
 [compExpressionList p609]
 [coerceable p356]
 [comp p590]
 [coerce p351]
 [compForm2 p610]
 [augModemapsFromDomain1 p252]
 [getFormModemaps p607]
 [nreverse0 p??]
 [addDomain p248]
 [compToApply p605]
 [\$NumberOfArgsIfInteger p??]
 [\$Expression p??]
 [\$EmptyMode p172]

— defun compForm1 —

```
(defun |compForm1| (form mode env)
  (let (|$NumberOfArgsIfInteger| op argl domain tmp1 opprime ans mmList td
        tmp2 tmp3 tmp4 tmp5 tmp6 tmp7)
    (declare (special |$NumberOfArgsIfInteger| |$Expression| |$EmptyMode|))
    (setq op (car form))
    (setq argl (cdr form))
    (setq |$NumberOfArgsIfInteger| (|#| argl))
    (cond
      ((eq op '|error|)
        (list
          (cons op
            (dolist (x argl (nreverse0 tmp4))
              (setq tmp2 (|outputComp| x env))
              (setq env (third tmp2))
              (push (car tmp2) tmp4)))
            mode env)))
```

```

((and (consp op) (eq (qfirst op) '|elt|)
  (progn
    (setq tmp3 (qrest op))
    (and (consp tmp3)
      (progn
        (setq domain (qfirst tmp3))
        (setq tmp1 (qrest tmp3))
        (and (consp tmp1)
          (eq (qrest tmp1) nil)
          (progn
            (setq opprime (qfirst tmp1))
            t))))))
(cond
  ((eq domain '|Lisp|)
    (list
      (cons opprime
        (dolist (x arg1 (nreverse tmp7))
          (setq tmp2 (|compOrCroak| x |$EmptyMode| env))
          (setq env (third tmp2))
          (push (car tmp2) tmp7)))
        mode env))
    ((and (equal domain |$Expression|) (eq opprime '|construct|))
      (|compExpressionList| arg1 mode env))
    ((and (eq opprime 'collect) (|coerceable| domain mode env))
      (when (setq td (|comp| (cons opprime arg1) domain env))
        (|coerce| td mode)))
    ((and (consp domain) (eq (qfirst domain) '|Mapping|)
      (setq ans
        (|compForm2| (cons opprime arg1) mode
          (setq env (|augModemapsFromDomain1| domain domain env))
          (dolist (x (|getFormModemaps| (cons opprime arg1) env)
            (nreverse0 tmp6))
            (when
              (and (consp x)
                (and (consp (qfirst x)) (equal (qcaar x) domain)))
              (push x tmp6))))))
        ans)
      ((setq ans
        (|compForm2| (cons opprime arg1) mode
          (setq env (|addDomain| domain env))
          (dolist (x (|getFormModemaps| (cons opprime arg1) env)
            (nreverse0 tmp5))
            (when
              (and (consp x)
                (and (consp (qfirst x)) (equal (qcaar x) domain)))
              (push x tmp5))))))
        ans)
      ((and (eq opprime '|construct|) (|coerceable| domain mode env))
        (when (setq td (|comp| (cons opprime arg1) domain env))
          (|coerce| td mode)))

```



```

      (t nil)))
    (t
      (setq env (|addDomain| mode env))
      (cond
        ((and (setq mmList (|getFormModemaps| form env))
              (setq td (|compForm2| form mode env mmList)))
          td)
        (t
          (|compToApply| op arg1 mode env))))))

```

defun compToApply

[\[compNoStacking p590\]](#)
[\[compApplication p605\]](#)
[\[\\$EmptyMode p172\]](#)

— defun compToApply —

```

(defun |compToApply| (op arg1 m e)
  (let (tt m1)
    (declare (special |$EmptyMode|))
    (setq tt (|compNoStacking| op |$EmptyMode| e))
    (when tt
      (setq m1 (cadr tt))
      (cond
        ((and (consp (car tt)) (eq (qcar (car tt)) 'quote)
              (consp (qcdr (car tt))) (eq (qcddr (car tt)) nil)
              (equal (qcadr (car tt)) m1))
          nil)
        (t
          (|compApplication| op arg1 m (caddr tt) tt))))))

```

defun compApplication

[\[eltForm p??\]](#)
[\[resolve p361\]](#)
[\[coerce p351\]](#)
[\[strconc p??\]](#)
[\[encodeItem p181\]](#)
[\[getAbbreviation p298\]](#)
[\[length p??\]](#)

```

[member p??]
[comp p590]
[isCategoryForm p??]
[$Category p??]
[$formatArgList p??]
[$op p??]
[$form p??]
[$prefix p??]

```

— defun compApplication —

```

(defun |compApplication| (op argl m env tt)
  (let (argml retm temp1 argTl nprefix opp form eltForm)
    (declare (special |$form| |$op| |$prefix| |$formalArgList| |$Category|))
    (cond
      ((and (consp (cadr tt)) (eq (qcar (cadr tt)) '|Mapping|)
            (consp (qcdr (cadr tt)))))
        (setq retm (qcadr (cadr tt)))
        (setq argml (qcddr (cadr tt)))
        (cond
          ((not (eql (|#| argl) (|#| argml))) nil)
          (t
           (setq retm (|resolve| m retm))
           (cond
             ((or (equal retm |$Category|) (|isCategoryForm| retm env))
              nil)
             (t
              (setq argTl
                (loop for x in argl for m in argml
                  collect (progn
                    (setq temp1 (or (|comp| x m env) (return '|failed|)))
                    (setq env (caddr temp1))
                    temp1))))
              (cond
                ((eq argTl '|failed|) nil)
                (t
                 (setq form
                  (cond
                    ((and
                     (null
                      (or (|member| op |$formalArgList|)
                          (|member| (car tt) |$formalArgList|)))
                     (atom (car tt)))
                     (setq nprefix
                      (or |$prefix| (|getAbbreviation| |$op| (|#| (cdr |$form|)))))
                     (setq opp
                      (intern
                       (strconc (|encodeItem| nprefix) '|;' (|encodeItem| (car tt)))))
                      (cons opp

```

```

      (append
        (loop for item in argTl collect (car item))
        (list '$))))
    (t
      (cons '|call|
        (cons (list '|applyFun| (car tt))
          (loop for item in argTl collect (car item))))))
      (|coerce| (list form retm env) (|resolve| retm m))))))
((eq op '|elt|) nil)
(t
  (setq eltForm (cons '|elt| (cons op argl)))
  (|comp| eltForm m env))))

```

defun getFormModemaps

```

[getFormModemaps p607]
[nreverse0 p??]
[get p??]
[eltModemapFilter p608]
[last p??]
[length p??]
[stackMessage p??]
[$insideCategoryPackageIfTrue p??]

```

— defun getFormModemaps —

```

(defun |getFormModemaps| (form env)
  (let (op argl domain op1 modemapList nargs finalModemapList)
    (declare (special |$insideCategoryPackageIfTrue|))
    (setq op (car form))
    (setq argl (cdr form))
    (cond
      ((and (consp op) (eq (qfirst op) '|elt|) (CONSP (qrest op))
        (consp (qcddr op)) (eq (qcdddr op) nil))
        (setq op1 (third op))
        (setq domain (second op))
        (loop for x in (|getFormModemaps| (cons op1 argl) env)
          when (and (consp x) (consp (qfirst x)) (equal (qcaar x) domain))
            collect x))
      ((null (atom op)) nil)
      (t
        (setq modemapList (|get| op '|modemap| env))
        (when |$insideCategoryPackageIfTrue|
          (setq modemapList
            (loop for x in modemapList

```

```

      when (and (consp x) (consp (qfirst x)) (not (eq (qcaar x) '$)))
        collect x))))))
(cond
  ((eq op '|elt|)
   (setq modemapList (|eltModemapFilter| (|last| argl) modemapList env)))
  ((eq op '|setelt|)
   (setq modemapList (|seteltModemapFilter| (CADR argl) modemapList env))))
(setq nargs (|#| argl))
(setq finalModemapList
  (loop for mm in modemapList
    when (equal (|#| (cddar mm)) nargs)
    collect mm))
(when (and modemapList (null finalModemapList))
  (|stackMessage|
   (list '|no modemap for| '|%b| op '|%d| '|with | nargs '| arguments|)))
finalModemapList))

```

defun eltModemapFilter

```

[isConstantId p??]
[stackMessage p??]

```

— defun eltModemapFilter —

```

(defun |eltModemapFilter| (name mmList env)
  (let (z)
    (if (|isConstantId| name env)
      (cond
        ((setq z
          (loop for mm in mmList
            when (and (consp mm) (consp (qfirst mm)) (consp (qcdar mm))
              (consp (qcddar mm))
              (consp (qcdddar mm))
              (equal (fourth (first mm)) name))
          collect mm))
         z)
      (t
       (|stackMessage|
        (list '|selector variable: | name '| is undeclared and unbound|))
        nil))
      mmList)))

```

defun seteltModemapFilter

```
[isConstantId p??]
[stackMessage p??]
```

— defun seteltModemapFilter —

```
(defun |seteltModemapFilter| (name mmList env)
  (let (z)
    (if (|isConstantId| name env)
      (cond
        ((setq z
          (loop for mm in mmList
            when (equal (car (cdddar mm)) name)
              collect mm))
         z)
      (t
       (|stackMessage|
        (list '|selector variable: | name '| is undeclared and unbound|))
        nil))
      mmList)))
```

—————

defun compExpressionList

```
[nreverse0 p??]
[comp p590]
[convert p600]
[$Expression p??]
```

— defun compExpressionList —

```
(defun |compExpressionList| (arg1 m env)
  (let (tmp1 tlst)
    (declare (special |$Expression|))
    (setq tlst
      (prog (result)
        (return
          (do ((tmp2 arg1 (cdr tmp2)) (x nil))
            ((or (atom tmp2)) (nreverse0 result))
            (setq x (car tmp2))
            (setq result
              (cons
                (progn
                  (setq tmp1 (or (|comp| x |$Expression| env) (return '|failed|)))
                  (setq env (third tmp1))
```

```

      tmp1)
      result))))))
(unless (eq t1st '|failed|)
  (|convert|
    (list (cons 'list
      (prog (result)
        (return
          (do ((tmp3 t1st (cdr tmp3)) (y nil))
            ((or (atom tmp3)) (nreverse0 result))
            (setq y (car tmp3))
            (setq result (cons (car y) result))))))
      |$Expression| env)
    m)))

```

defun compForm2

```

[take p??]
[length p??]
[nreverse0 p??]
[sublis p??]
[assoc p??]
[PredImplies p??]
[isSimple p??]
[compUniquely p616]
[compFormPartiallyBottomUp p615]
[compForm3 p612]
[$EmptyMode p172]
[$TriangleVariableList p??]

```

— defun compForm2 —

```

(defun |compForm2| (form mode env modemapList)
  (let (op argl sargl 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 argl (cdr form))
    (setq sargl (take (|#| argl) |$TriangleVariableList|))
    (setq aList (mapcar #'(lambda (x y) (cons x y)) sargl argl))
    (setq modemapList (sublis aList modemapList))
    ; now delete any modemaps that are subsumed by something else, provided
    ; the conditions are right (i.e. subsumer true whenever subsumee true)
    (dolist (u modemapList)
      (cond
        ((and (consp u)

```

```

(progn
  (setq tmp6 (qfirst u))
  (and (consp tmp6) (progn (setq dc (qfirst tmp6)) t)))
(progn
  (setq tmp7 (qrest u))
  (and (consp tmp7) (eq (qrest tmp7) nil)
    (progn
      (setq tmp1 (qfirst tmp7))
      (and (consp tmp1)
        (progn
          (setq cond (qfirst tmp1))
          (setq tmp2 (qrest tmp1))
          (and (consp tmp2) (eq (qrest tmp2) nil)
            (progn
              (setq tmp3 (qfirst tmp2))
              (and (consp tmp3) (eq (qfirst tmp3) '|Subsumed|)
                (progn
                  (setq tmp4 (qrest tmp3))
                  (and (consp tmp4)
                    (progn
                      (setq tmp5 (qrest tmp4))
                      (and (consp tmp5)
                        (eq (qrest tmp5) nil)
                        (progn
                          (setq nsig (qfirst tmp5))
                          t))))))))))))))
      (setq v (|assoc| (cons dc nsig) modemapList))
      (consp v)
      (progn
        (setq tmp6 (qrest v))
        (and (consp tmp6) (eq (qrest tmp6) nil)
          (progn
            (setq tmp7 (qfirst tmp6))
            (and (consp tmp7)
              (progn
                (setq ncond (qfirst tmp7))
                t))))))
      (setq deleteList (cons u deleteList))
      (unless (|PredImplies| ncond cond)
        (setq newList (push '(', (car u) (,cond (elt ,dc nil))) newList))))))
(when deleteList
  (setq modemapList
    (remove-if #'(lambda (x) (member x deletelist)) modemapList)))
; it is important that subsumed ops (newList) be considered last
(when newList (setq modemapList (append modemapList newList)))
(setq tl
  (loop for x in arg1
    while (and (|isSimple| x)
      (setq td (|compUniquely| x |$EmptyMode| env)))
    collect td

```

```

      do (setq env (third td))))
(cond
  ((some #'identity tl)
   (setq partialModeList (loop for x in tl collect (when x (second x))))
   (or
    (|compFormPartiallyBottomUp| form mode env modemapList partialModeList)
    (|compForm3| form mode env modemapList)))
  (t (|compForm3| form mode env modemapList))))

```

defun compForm3

[compFormWithModemap p??]
 [compUniquely p616]
 [\$compUniquelyIfTrue p??]

— defun compForm3 —

```

(defun |compForm3| (form mode env modemapList)
  (let (op argl mml tt)
    (declare (special |$compUniquelyIfTrue|))
    (setq op (car form))
    (setq argl (cdr form))
    (setq tt
      (let (result)
        (maplist #'(lambda (mlst)
          (setq result (or result
            (|compFormWithModemap| form mode env (car (setq mml mlst))))))
          modemapList)
        result))
    (when |$compUniquelyIfTrue|
      (if (let (result)
          (mapcar #'(lambda (mm)
            (setq result (or result (|compFormWithModemap| form mode env mm))))
            (rest mml))
          result)
        (throw '|compUniquely| nil)
        tt))
    tt))

```

defun compFocompFormWithModemap

```

[isCategoryForm p??]
[isFunctor p249]
[substituteIntoFunctorModemap p614]
[listOfSharpVars p??]
[coerceable p356]
[compApplyModemap p255]
[isCategoryForm p??]
[identp p??]
[get p??]
[last p??]
[convert p600]
[$Category p??]
[$FormalMapVariableList p266]

```

— **defun compFormWithModemap** —

```

(defun |compFormWithModemap| (form m env modemap)
  (prog (op argl sv target cexpr targetp map temp1 f transimp sl mp formp z c
        xp ep tt)
    (declare (special |$Category| |$FormalMapVariableList|))
    (return
      (progn
        (setq op (car form))
        (setq argl (cdr form))
        (setq map (car modemap))
        (setq target (cadar modemap))
        (when (and (|isCategoryForm| target env) (|isFunctor| op))
          (setq temp1 (or (|substituteIntoFunctorModemap| argl modemap env)
                          (return nil))))
        (setq modemap (car temp1))
        (setq env (cadr temp1))
        (setq map (car modemap))
        (setq target (cadar modemap))
        (setq cexpr (cdr modemap))
        modemap)
      (setq sv (|listOfSharpVars| map))
      (when sv
        (loop for x in argl for ss in |$FormalMapVariableList|
          do (when (|member| ss sv)
              (setq modemap (subst x ss modemap :test #'equal))
              (setq map (car modemap))
              (setq target (cadar modemap))
              (setq cexpr (cdr modemap))
              modemap)))
      (cond
        ((null (setq targetp (|coerceable| target m env))) nil)

```

```

(t
  (setq map (cons targetp (cdr map)))
  (setq temp1 (or (|compApplyModemap| form modemap env nil)
                  (return nil)))
  (setq f (car temp1))
  (setq transimp (cadr temp1))
  (setq sl (caddr temp1))
  (setq mp (sublis sl (elt map 1)))
  (setq xp
    (progn
      (setq formp (cons f (loop for tt in transimp collect (car tt))))
      (cond
        ((or (equal mp |$Category|) (|isCategoryForm| mp env)) formp)
        ((and (eq op '|elt|) (consp f) (eq (qcar f) 'xlam)
              (identp (car argl))
              (setq c (|get| (car argl) '|condition| env))
              (consp c) (eq (qcdr c) nil)
              (consp (qcar c)) (eq (qcaar c) '|case|)
              (consp (qcdr c)) (equal (qcadar c) z)
              (consp (qcddar c)) (eq (qcdr (qcddar c)) nil)
              (or (and (consp (qcaddar c))
                      (eq (qcar (qcaddar c)) '|:|)
                      (consp (qcdr (qcaddar c)))
                      (equal (qcadr (qcaddar c)) (cadr argl))
                      (consp (qcddr (qcaddar c)))
                      (eq (qcdddr (qcaddar c)) nil)
                      (equal (qcaddr (qcaddar c)) m))
                  (eq (qcaddar c) (cadr argl))))
              (list 'cdr (car argl)))
        (t (cons '|call| formp))))))
  (setq ep
    (if transimp
      (caddr (|last| transimp))
      env))
  (setq tt (list xp mp ep))
  (|convert| tt m))))))

```

defun substituteIntoFunctorModemap

```

[keyedSystemError p??]
[eqsubstlist p??]
[compOrCroak p588]
[sublis p??]

```

— defun substituteIntoFunctorModemap —

```
(defun |substituteIntoFunctorModemap| (argl modemap env)
  (let (dc sig tmp1 tl substitutionList)
    (setq dc (caar modemap))
    (setq sig (cdar modemap))
    (cond
      ((not (eql (|#| dc) (|#| sig)))
        (|keyedSystemError| 'S2GE0016
          (list "substituteIntoFunctorModemap" "Incompatible maps"))))
      ((equal (|#| argl) (|#| (cdr sig)))
        (setq sig (eqsubstlist argl (cdr dc) sig))
        (setq tl
          (loop for a in argl for m in (rest sig)
            collect (progn
              (setq tmp1 (|compOrCroak| a m env))
              (setq env (caddr tmp1))
              tmp1)))
        (setq substitutionList
          (loop for x in (rest dc) for tt in tl
            collect (cons x (car tt))))
        (list (sublis substitutionList modemap) env))
      (t nil))))
```

defun compFormPartiallyBottomUp

[compForm3 p612]
 [compFormMatch p615]

— defun compFormPartiallyBottomUp —

```
(defun |compFormPartiallyBottomUp| (form mode env modemapList partialModeList)
  (let (mmList)
    (when (setq mmList (loop for mm in modemapList
      when (|compFormMatch| mm partialModeList)
      collect mm))
      (|compForm3| form mode env mmList))))
```

defun compFormMatch

— defun compFormMatch —

```
(defun |compFormMatch| (mm partialModeList)
  (labels (
    (ismatch (a b)
      (cond
        ((null b) t)
        ((null (car b)) (|compFormMatch,match| (cdr a) (cdr b)))
        ((and (equal (car a) (car b)) (ismatch (cdr a) (cdr b)))))))
    (and (consp mm) (consp (qfirst mm)) (consp (qcddar mm))
      (ismatch (qcddar mm) partialModeList))))
```

defun compUniquely

[compUniquely p616]
 [comp p590]
 [\$compUniquelyIfTrue p??]

— defun compUniquely —

```
(defun |compUniquely| (x m env)
  (let (|$compUniquelyIfTrue|)
    (declare (special |$compUniquelyIfTrue|))
    (setq |$compUniquelyIfTrue| t)
    (catch '|compUniquely| (|comp| x m env))))
```

defun compArgumentsAndTryAgain

[comp p590]
 [compForm1 p603]
 [\$EmptyMode p172]

— defun compArgumentsAndTryAgain —

```
(defun |compArgumentsAndTryAgain| (form mode env)
  (let (arg1 tmp1 a tmp2 tmp3 u)
    (declare (special |$EmptyMode|))
    (setq arg1 (cdr form))
    (cond
      ((and (consp form) (eq (qfirst form) '|elt|))
        (progn
          (setq tmp1 (qrest form))
          (and (consp tmp1)
```

```

      (progn
        (setq a (qfirst tmp1))
        (setq tmp2 (qrest tmp1))
        (and (consp tmp2) (eq (qrest tmp2) nil))))))
(when (setq tmp3 (|comp| a |$EmptyMode| env))
  (setq env (third tmp3))
  (|compForm1| form mode env)))
(t
  (setq u
    (dolist (x arg1)
      (setq tmp3 (or (|comp| x |$EmptyMode| env) (return '|failed|)))
      (setq env (third tmp3))
      tmp3))
    (unless (eq u '|failed|)
      (|compForm1| form mode env))))))

```

defun compWithMappingMode

[compWithMappingMode1 [p617](#)]
 [\$formalArgList p??]

— defun compWithMappingMode —

```

(defun |compWithMappingMode| (form mode oldE)
  (declare (special |$formalArgList|))
  (|compWithMappingMode1| form mode oldE |$formalArgList|))

```

defun compWithMappingMode1

[isFunctor [p249](#)]
 [get p??]
 [extendsCategoryForm p??]
 [compLambda [p321](#)]
 [stackAndThrow p??]
 [take p??]
 [compMakeDeclaration [p624](#)]
 [hasFormalMapVariable [p623](#)]
 [comp [p590](#)]
 [extractCodeAndConstructTriple [p622](#)]
 [optimizeFunctionDef [p223](#)]

```

[comp-tran p??]
[freelist p625]
[$formalArgList p??]
[$killOptimizeIfTrue p??]
[$funname p??]
[$funnameTail p??]
[$QuickCode p??]
[$EmptyMode p172]
[$FormalMapVariableList p266]
[$CategoryFrame p??]
[$formatArgList p??]

```

— defun compWithMappingModel1 —

```

(defun |compWithMappingModel1| (form mode oldE |$formalArgList|)
  (declare (special |$formalArgList|))
  (prog (|$killOptimizeIfTrue| $funname $funnameTail mprime s1 tmp1 tmp2
        tmp3 tmp4 tmp5 tmp6 target argModeList nx oldstyle ress v11 vl e tt
        u frees i scode locals body vec expandedFunction fname uu)
    (declare (special |$killOptimizeIfTrue| $funname $funnameTail
                      |$QuickCode| |$EmptyMode| |$FormalMapVariableList|
                      |$CategoryFrame| |$formatArgList|))

    (return
     (seq
      (progn
       (setq mprime (second mode))
       (setq s1 (caddr mode))
       (setq |$killOptimizeIfTrue| t)
       (setq e oldE)
       (cond
        ((|isFunctor| form)
         (cond
          ((and (progn
                  (setq tmp1 (|get| form '|modemap| |$CategoryFrame|))
                  (and (consp tmp1)
                       (progn
                        (setq tmp2 (qfirst tmp1))
                        (and (consp tmp2)
                            (progn
                             (setq tmp3 (qfirst tmp2))
                             (and (consp tmp3)
                                 (progn
                                  (setq tmp4 (qrest tmp3))
                                  (and (consp tmp4)
                                      (progn
                                       (setq target (qfirst tmp4))
                                       (setq argModeList (qrest tmp4))
                                       t))))))
                           (progn
                            (progn
                             (setq target (qfirst tmp4))
                             (setq argModeList (qrest tmp4))
                             t))))))
                     (progn
                      (progn
                       (setq target (qfirst tmp4))
                       (setq argModeList (qrest tmp4))
                       t))))))
          (progn
           (progn
            (setq target (qfirst tmp4))
            (setq argModeList (qrest tmp4))
            t))))))

```

```

                                (setq tmp5 (qrest tmp2))
                                (and (consp tmp5) (eq (qrest tmp5) nil))))))
  (prog (t1)
    (setq t1 t)
    (return
      (do ((t2 nil (null t1))
          (t3 argModeList (cdr t3))
          (newmode nil)
          (t4 sl (cdr t4))
          (s nil))
        ((or t2 (atom t3)
            (progn (setq newmode (car t3)) nil)
            (atom t4)
            (progn (setq s (car t4)) nil))
         t1)
        (seq (exit
              (setq t1
                (and t1 (|extendsCategoryForm| '$ s newmode))))))
          (|extendsCategoryForm| '$ target mprime))
        (return (list form mode e )))
      (t nil)))
  (t
    (when (stringp form) (setq form (intern form)))
    (setq ress nil)
    (setq oldstyle t)
    (cond
      ((and (consp form)
            (eq (qfirst form) '+-->)
            (progn
              (setq tmp1 (qrest form))
              (and (consp tmp1)
                (progn
                  (setq v1 (qfirst tmp1))
                  (setq tmp2 (qrest tmp1))
                  (and (consp tmp2)
                    (eq (qrest tmp2) nil)
                    (progn (setq nx (qfirst tmp2)) t))))))
        (setq oldstyle nil)
        (cond
          ((and (consp v1) (eq (qfirst v1) '|:|))
            (setq ress (|compLambda| form mode oldE)
              ress)
            (t
              (setq v1
                (cond
                  ((and (consp v1)
                        (eq (qfirst v1) '|@Tuple|)
                        (progn (setq v11 (qrest v1)) t))
                   v11)
                (t v1)))
            (t v1)))

```

```

(setq vl
  (cond
    ((symbolp vl) (cons vl nil))
    ((and
      (listp vl)
      (prog (t5)
        (setq t5 t)
        (return
          (do ((t7 nil (null t5))
              (t6 vl (cdr t6))
              (v nil))
            ((or t7 (atom t6) (progn (setq v (car t6)) nil)) t5)
            (seq
              (exit
                (setq t5 (and t5 (symbolp v))))))))))
    vl)
  (t
    (|stackAndThrow| (cons '|bad +-> arguments:| (list vl )))))
  (setq |$formatArgList| (append vl |$formalArgList|))
  (setq form 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| form vl)))
        (return
          (progn
            (setq tmp6 (or (|comp| (cons form vl) mprime e) (return nil)))
            (setq u (car tmp6))
            (|extractCodeAndConstructTriple| u mode oldE))))
        ((and (null vl) (setq tt (|comp| (cons form nil) mprime e)))
          (return
            (progn
              (setq u (car tt))
              (|extractCodeAndConstructTriple| u mode oldE))))

```



```

(t
  (setq tmp6 (or (|comp| form 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 '$$))
                            (caddr expandedFunction))))
      ((eql (|#| frees) 1)
       (setq vec (caar frees))
       (cons 'lambda (cons (append vl (list vec))
                            (caddr expandedFunction))))
      (t
       (setq scode nil)
       (setq vec nil)
       (setq locals nil)
       (setq i -1)
       (do ((t0 frees (cdr t0)) (v nil))
           ((or (atom t0) (progn (setq v (car t0)) nil)) nil)
         (seq
          (exit
           (progn
            (setq i (plus i 1))
            (setq vec (cons (car v) vec))
            (setq scode
              (cons
               (cons 'setq
                 (cons (car v)
                      (cons
                       (cons
                        (cond
                          (|$QuickCode| 'qrefelt)
                          (t 'elt))
                        (cons '$$ (cons i nil)))
                        nil)))
              scode)))
            (setq locals (cons (car v) locals))))))
       (setq body (caddr expandedFunction))
       (cond
        (locals
         (cond

```

```

((and (consp body)
      (progn
        (setq tmp1 (qfirst body))
        (and (consp tmp1)
              (eq (qfirst 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 mode oldE)))))))))

```

defun extractCodeAndConstructTriple

— defun extractCodeAndConstructTriple —

```
(defun |extractCodeAndConstructTriple| (form mode oldE)
  (let (tmp1 a fn op env)
    (cond
      ((and (consp form) (eq (qfirst form) '|call|))
        (progn
          (setq tmp1 (qrest form))
          (and (consp tmp1)
              (progn (setq fn (qfirst tmp1)) t))))
      (cond
        ((and (consp fn) (eq (qfirst fn) '|applyFun|))
          (progn
            (setq tmp1 (qrest fn))
            (and (consp tmp1) (eq (qrest tmp1) nil)
                (progn (setq a (qfirst tmp1)) t))))
          (setq fn a)))
      (list fn mode oldE))
    (t
     (setq op (car form))
     (setq env (car (reverse (cdr form))))
     (list (list 'cons (list '|function| op) env) mode oldE))))
```

defun hasFormalMapVariable

```
[hasFormalMapVariable ScanOrPairVec (vol5)]
[$formalMapVariables p??]
```

— defun hasFormalMapVariable —

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

defun argsToSig

— defun argsToSig —

```
(defun |argsToSig| (args)
  (let (tmp1 v tmp2 tt sig1 arg1 bad)
```

```

(cond
  ((and (consp args) (eq (qfirst args) '[:|]))
    (progn
      (setq tmp1 (qrest args))
      (and (consp tmp1)
        (progn
          (setq v (qfirst tmp1))
          (setq tmp2 (qrest tmp1))
          (and (consp tmp2)
            (eq (qrest tmp2) nil)
            (progn
              (setq tt (qfirst tmp2))
              t))))))
    (list (list v) (list tt)))
  (t
    (setq sig1 nil)
    (setq arg1 nil)
    (setq bad nil)
    (dolist (arg args)
      (cond
        ((and (consp arg) (eq (qfirst arg) '[:|]))
          (progn
            (setq tmp1 (qrest arg))
            (and (consp tmp1)
              (progn
                (setq v (qfirst tmp1))
                (setq tmp2 (qrest tmp1))
                (and (consp tmp2) (eq (qrest tmp2) nil)
                  (progn
                    (setq tt (qfirst tmp2))
                    t))))))
            (setq sig1 (cons tt sig1))
            (setq arg1 (cons v arg1)))
          (t (setq bad t))))
    (cond
      (bad (list nil nil ))
      (t (list (reverse arg1) (reverse sig1)))))))

```

—————

defun compMakeDeclaration

[compColon p290]
 [\$insideExpressionIfTrue p??]

— defun compMakeDeclaration —

```
(defun |compMakeDeclaration| (form mode env)
```

```
(let (|$insideExpressionIfTrue|)
  (declare (special |$insideExpressionIfTrue|))
  (setq |$insideExpressionIfTrue| nil)
  (|compColon| form mode env)))
```

defun modifyModeStack

```
[say p??]
[copy p??]
[setelt p??]
[resolve p361]
[$reportExitModeStack p??]
[$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)))))
```

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 p625]
[freelist assq (vol5)]
[freelist identp (vol5)]
[getmode p??]
[unionq p??]
```

— defun freelist —

```

(defun freelist (u bound free e)
  (let (v op)
    (if (atom u)
        (cond
          ((null (identp u)) free)
          ((member u bound) free)
          ; more than 1 free becomes alist (name . number)
          ((setq v (assq u free)) (rplacd v (+ 1 (cdr v))) free)
          ((null (|getmode| u e)) free)
          (t (cons (cons u 1) free)))
        (progn
          (setq op (car u))
          (cond
            ((member op '(quote go |function|)) free)
            ((eq op 'lambda) ; lambdas bind symbols
              (setq bound (unionq bound (second u)))
              (dolist (v (cddr u))
                (setq free (freelist v bound free e))))
            ((eq op 'prog) ; progs bind symbols
              (setq bound (unionq bound (second u)))
              (dolist (v (cddr u))
                (unless (atom v)
                  (setq free (freelist v bound free e))))))
            ((eq op 'seq)
              (dolist (v (cdr u))
                (unless (atom v)
                  (setq free (freelist v bound free e))))))
            ((eq op 'cond)
              (dolist (v (cdr u))
                (dolist (vv v)
                  (setq free (freelist vv bound free e))))))
            (t
              (when (atom op) (setq u (cdr u))) ; atomic functions aren't descended
              (dolist (v u)
                (setq free (freelist v bound free e))))
              free))))

```

defun compOrCroak1,compactify

[compOrCroak1,compactify p626]
 [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))))))
```

defun Compiler/Interpreter interface

```
[ncINTERPFILE SpadInterpretStream (vol5)]
[$EchoLines p??]
[$ReadingFile p??]
```

— defun ncINTERPFILE —

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

defun recompile-lib-file-if-necessary

```
[compile-lib-file p628]
[*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))))
    (declare (special *lisp-bin-filetype*))
    (unless (and ldate bdate (> bdate ldate))
      (compile-lib-file lfile)
      (list bfile))))
```

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)
```

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)))
```

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)))
```

defvar \$byConstructors

— initvars —

```
(defvar |$byConstructors| () "list of constructors to be compiled")
```

—————

defvar \$constructorsSeen

— initvars —

```
(defvar |$constructorsSeen| () "list of constructors found")
```

—————

Chapter 16

Level 1

defvar \$current-fragment

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

— **initvars** —

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

—————

defun read-a-line

```
[subseq p??]  
[Line-New-Line p??]  
[read-a-line p631]  
[*eof* p??]  
[File-Closed p??]
```

— **defun read-a-line** —

```
(defun read-a-line (&optional (stream t))  
  (let (cp)  
    (declare (special *eof* File-Closed))  
    (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)  
        (read-line stream))
```

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

—————→

Chapter 17

Level 0

17.1 Line Handling

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.

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))
```

—————

defvar \$current-line

The current input line.

— **initvars** —

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

—————

defmacro line-clear

[*\$line* p633]

— **defmacro line-clear** —

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

—————

defun line-print

[*\$line* p633]

[*\$out-stream* p??]

— **defun line-print** —

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

—————

defun line-at-end-p

[*\$line* p633]

— **defun line-at-end-p** —

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

defun line-past-end-p

[[line](#) [p633](#)]

— defun line-past-end-p —

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

defun line-next-char

[[line](#) [p633](#)]

— defun line-next-char —

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

defun line-advance-char

[[line](#) [p633](#)]

— defun line-advance-char —

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

defun line-current-segment

[\$line p633]

— defun line-current-segment —

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

—————

defun line-new-line

[\$line p633]

— 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)))))
```

—————

defun next-line

[\$in-stream p??]

[\$line-handler p??]

— defun next-line —

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

—————

defun Advance-Char

[Line-At-End-P p??]
 [Line-Advance-Char p??]
 [next-line p636]
 [current-char p463]
 [\$in-stream p??]
 [\$line p633]

— defun Advance-Char —

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

—————

defun storeblanks

— defun storeblanks —

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

—————

defun initial-substring

— defun initial-substring —

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

defun get-a-line

[is-console p555]
[get-a-line mkprompt (vol5)]
[read-a-line p631]

— **defun get-a-line** —

```
(defun get-a-line (stream)
  (when (is-console stream) (princ (mkprompt)))
  (let ((l1 (read-a-line stream)))
    (if (and (stringp l1) (adjustable-array-p l1))
        (make-array (array-dimensions l1) :element-type 'string-char
                     :adjustable t :initial-contents l1)
        l1)))
```

Chapter 18

The Chunks

— Compiler —

```
(in-package "BOOT")

\getchunk{initvars}

\getchunk{LEDNUDTables}
\getchunk{GLIPHTable}
\getchunk{RENAMETOKTable}
\getchunk{GENERICTable}

\getchunk{defmacro bang}
\getchunk{defmacro line-clear}
\getchunk{defmacro must}
\getchunk{defmacro nth-stack}
\getchunk{defmacro pop-stack-1}
\getchunk{defmacro pop-stack-2}
\getchunk{defmacro pop-stack-3}
\getchunk{defmacro pop-stack-4}
\getchunk{defmacro reduce-stack-clear}
\getchunk{defmacro stack-/empty}
\getchunk{defmacro star}

\getchunk{defun action}
\getchunk{defun addArgumentConditions}
\getchunk{defun addclose}
\getchunk{defun addConstructorModemaps}
\getchunk{defun addDomain}
\getchunk{defun addEltModemap}
\getchunk{defun addEmptyCapsuleIfNecessary}
\getchunk{defun addModemapKnown}
```

```

\getchunk{defun addModemap}
\getchunk{defun addModemap0}
\getchunk{defun addModemap1}
\getchunk{defun addNewDomain}
\getchunk{defun add-parens-and-semis-to-line}
\getchunk{defun addSuffix}
\getchunk{defun Advance-Char}
\getchunk{defun advance-token}
\getchunk{defun alistSize}
\getchunk{defun allLASSOCs}
\getchunk{defun aplTran}
\getchunk{defun aplTran1}
\getchunk{defun aplTranList}
\getchunk{defun applyMapping}
\getchunk{defun argsToSig}
\getchunk{defun assignError}
\getchunk{defun AssocBarGensym}
\getchunk{defun augLisplibModemapsFromCategory}
\getchunk{defun augmentLisplibModemapsFromFunctor}
\getchunk{defun augModemapsFromCategory}
\getchunk{defun augModemapsFromCategoryRep}
\getchunk{defun augModemapsFromDomain}
\getchunk{defun augModemapsFromDomain1}
\getchunk{defun autoCoerceByModemap}

\getchunk{defun blankp}
\getchunk{defun bootStrapError}
\getchunk{defun buildLibAttr}
\getchunk{defun buildLibAttrs}
\getchunk{defun buildLibdb}
\getchunk{defun buildLibdbConEntry}
\getchunk{defun buildLibdbString}
\getchunk{defun buildLibOp}
\getchunk{defun buildLibOps}
\getchunk{defun bumpererrorcount}

\getchunk{defun canReturn}
\getchunk{defun char-eq}
\getchunk{defun char-ne}
\getchunk{defun checkAddBackSlashes}
\getchunk{defun checkAddIndented}
\getchunk{defun checkAddMacros}
\getchunk{defun checkAddPeriod}
\getchunk{defun checkAddSpaces}
\getchunk{defun checkAddSpaceSegments}
\getchunk{defun checkAlphabetic}
\getchunk{defun checkAndDeclare}
\getchunk{defun checkArguments}
\getchunk{defun checkBalance}
\getchunk{defun checkBeginEnd}

```

```

\getchunk{defun checkComments}
\getchunk{defun checkDecorate}
\getchunk{defun checkDecorateForHt}
\getchunk{defun checkDocError}
\getchunk{defun checkDocError1}
\getchunk{defun checkDocMessage}
\getchunk{defun checkExtract}
\getchunk{defun checkFixCommonProblem}
\getchunk{defun checkGetArgs}
\getchunk{defun checkGetLispFunctionName}
\getchunk{defun checkGetMargin}
\getchunk{defun checkGetParse}
\getchunk{defun checkGetStringBeforeRightBrace}
\getchunk{defun checkHTargs}
\getchunk{defun checkIeEg}
\getchunk{defun checkIeEgfun}
\getchunk{defun checkIndentedLines}
\getchunk{defun checkIsValidType}
\getchunk{defun checkLookForLeftBrace}
\getchunk{defun checkLookForRightBrace}
\getchunk{defun checkNumOfArgs}
\getchunk{defun checkRecordHash}
\getchunk{defun checkRemoveComments}
\getchunk{defun checkRewrite}
\getchunk{defun checkSayBracket}
\getchunk{defun checkSkipBlanks}
\getchunk{defun checkSkipIdentifierToken}
\getchunk{defun checkSkipOpToken}
\getchunk{defun checkSkipToken}
\getchunk{defun checkSplitBackslash}
\getchunk{defun checkSplitBrace}
\getchunk{defun checkSplitOn}
\getchunk{defun checkSplitPunctuation}
\getchunk{defun checkSplit2Words}
\getchunk{defun checkTexht}
\getchunk{defun checkTransformFirsts}
\getchunk{defun checkTrim}
\getchunk{defun checkTrimCommented}
\getchunk{defun checkWarning}
\getchunk{defun coerce}
\getchunk{defun coerceable}
\getchunk{defun coerceByModemap}
\getchunk{defun coerceEasy}
\getchunk{defun coerceExit}
\getchunk{defun coerceExtraHard}
\getchunk{defun coerceHard}
\getchunk{defun coerceSubset}
\getchunk{defun collectAndDeleteAssoc}
\getchunk{defun collectComBlock}
\getchunk{defun comma2Tuple}

```

```
\getchunk{defun comp}  
\getchunk{defun comp2}  
\getchunk{defun comp3}  
\getchunk{defun compAdd}  
\getchunk{defun compAndDefine}  
\getchunk{defun compApplication}  
\getchunk{defun compApply}  
\getchunk{defun compApplyModemap}  
\getchunk{defun compArgumentConditions}  
\getchunk{defun compArgumentsAndTryAgain}  
\getchunk{defun compAtom}  
\getchunk{defun compAtomWithModemap}  
\getchunk{defun compAtSign}  
\getchunk{defun compBoolean}  
\getchunk{defun compCapsule}  
\getchunk{defun compCapsuleInner}  
\getchunk{defun compCapsuleItems}  
\getchunk{defun compCase}  
\getchunk{defun compCase1}  
\getchunk{defun compCat}  
\getchunk{defun compCategory}  
\getchunk{defun compCategoryItem}  
\getchunk{defun compCoerce}  
\getchunk{defun compCoerce1}  
\getchunk{defun compColon}  
\getchunk{defun compColonInside}  
\getchunk{defun compCons}  
\getchunk{defun compCons1}  
\getchunk{defun compConstruct}  
\getchunk{defun compConstructorCategory}  
\getchunk{defun compDefine}  
\getchunk{defun compDefine1}  
\getchunk{defun compDefineAddSignature}  
\getchunk{defun compDefineCapsuleFunction}  
\getchunk{defun compDefineCategory}  
\getchunk{defun compDefineCategory1}  
\getchunk{defun compDefineCategory2}  
\getchunk{defun compDefineFunctor}  
\getchunk{defun compDefineFunctor1}  
\getchunk{defun compDefineLisplib}  
\getchunk{defun compDefWhereClause}  
\getchunk{defun compElt}  
\getchunk{defun compExit}  
\getchunk{defun compExpression}  
\getchunk{defun compExpressionList}  
\getchunk{defun compForm}  
\getchunk{defun compForm1}  
\getchunk{defun compForm2}  
\getchunk{defun compForm3}  
\getchunk{defun compFormMatch}
```

```

\getchunk{defun compForMode}
\getchunk{defun compFormPartiallyBottomUp}
\getchunk{defun compFormWithModemap}
\getchunk{defun compFromIf}
\getchunk{defun compFunctorBody}
\getchunk{defun compHas}
\getchunk{defun compHasFormat}
\getchunk{defun compIf}
\getchunk{defun compile}
\getchunk{defun compileCases}
\getchunk{defun compileConstructor}
\getchunk{defun compileConstructor1}
\getchunk{defun compileDocumentation}
\getchunk{defun compileFileQuietly}
\getchunk{defun compile-lib-file}
\getchunk{defun compiler}
\getchunk{defun compilerDoit}
\getchunk{defun compilerDoitWithScreenedLispLib}
\getchunk{defun compileSpad2Cmd}
\getchunk{defun compileSpadLispCmd}
\getchunk{defun compileTimeBindingOf}
\getchunk{defun compImport}
\getchunk{defun compInternalFunction}
\getchunk{defun compIs}
\getchunk{defun compJoin}
\getchunk{defun compLambda}
\getchunk{defun compLeave}
\getchunk{defun compList}
\getchunk{defun compMacro}
\getchunk{defun compMakeCategoryObject}
\getchunk{defun compMakeDeclaration}
\getchunk{defun compMapCond}
\getchunk{defun compMapCond'}
\getchunk{defun compMapCond''}
\getchunk{defun compMapCondFun}
\getchunk{defun compNoStacking}
\getchunk{defun compNoStacking1}
\getchunk{defun compOrCroak}
\getchunk{defun compOrCroak1}
\getchunk{defun compOrCroak1,compactify}
\getchunk{defun compPretend}
\getchunk{defun compQuote}
\getchunk{defun compRepeatOrCollect}
\getchunk{defun compReduce}
\getchunk{defun compReduce1}
\getchunk{defun compReturn}
\getchunk{defun compSeq}
\getchunk{defun compSeqItem}
\getchunk{defun compSeq1}
\getchunk{defun compSetq}

```

```

\getchunk{defun compSetq1}
\getchunk{defun compSingleCapsuleItem}
\getchunk{defun compString}
\getchunk{defun compSubDomain}
\getchunk{defun compSubDomain1}
\getchunk{defun compSymbol}
\getchunk{defun compSubsetCategory}
\getchunk{defun compSuchthat}
\getchunk{defun compToApply}
\getchunk{defun compTopLevel}
\getchunk{defun compTuple2Record}
\getchunk{defun compTypeOf}
\getchunk{defun compUniquely}
\getchunk{defun compVector}
\getchunk{defun compWhere}
\getchunk{defun compWithMappingMode}
\getchunk{defun compWithMappingMode1}
\getchunk{defun constructMacro}
\getchunk{defun containsBang}
\getchunk{defun convert}
\getchunk{defun convertOpAlist2compilerInfo}
\getchunk{defun convertOrCroak}
\getchunk{defun current-char}
\getchunk{defun current-symbol}
\getchunk{defun current-token}

\getchunk{defun dbReadLines}
\getchunk{defun dbWriteLines}
\getchunk{defun decodeScripts}
\getchunk{defun deepestExpression}
\getchunk{defun def-rename}
\getchunk{defun disallowNilAttribute}
\getchunk{defun displayMissingFunctions}
\getchunk{defun displayPreCompilationErrors}
\getchunk{defun doIt}
\getchunk{defun doItIf}
\getchunk{defun dollarTran}
\getchunk{defun domainMember}
\getchunk{defun drop}

\getchunk{defun eltModemapFilter}
\getchunk{defun encodeItem}
\getchunk{defun encodeFunctionName}
\getchunk{defun EqualBarGensym}
\getchunk{defun errhuh}
\getchunk{defun escape-keywords}
\getchunk{defun escaped}
\getchunk{defun evalAndRwriteLispForm}
\getchunk{defun evalAndSub}
\getchunk{defun expand-tabs}

```



```

\getchunk{defun extendLocalLibdb}
\getchunk{defun extractCodeAndConstructTriple}

\getchunk{defun flattenSignatureList}
\getchunk{defun finalizeDocumentation}
\getchunk{defun finalizeLisplib}
\getchunk{defun fincomblock}
\getchunk{defun firstNonBlankPosition}
\getchunk{defun fixUpPredicate}
\getchunk{defun floatexpid}
\getchunk{defun formal2Pattern}
\getchunk{defun freelist}

\getchunk{defun get-a-line}
\getchunk{defun getAbbreviation}
\getchunk{defun getArgumentMode}
\getchunk{defun getArgumentModeOrMoan}
\getchunk{defun getCaps}
\getchunk{defun getCategoryOpsAndAtts}
\getchunk{defun getConstructorExports}
\getchunk{defun getConstructorOpsAndAtts}
\getchunk{defun getDomainsInScope}
\getchunk{defun getFormModemaps}
\getchunk{defun getFunctorOpsAndAtts}
\getchunk{defun getInverseEnvironment}
\getchunk{defun getMatchingRightPren}
\getchunk{defun getModemap}
\getchunk{defun getModemapList}
\getchunk{defun getModemapListFromDomain}
\getchunk{defun getOperationAList}
\getchunk{defun getScriptName}
\getchunk{defun getSignature}
\getchunk{defun getSignatureFromMode}
\getchunk{defun getSlotFromCategoryForm}
\getchunk{defun getSlotFromFunctor}
\getchunk{defun getSpecialCaseAssoc}
\getchunk{defun getSuccessEnvironment}
\getchunk{defun getTargetFromRhs}
\getchunk{defun get-token}
\getchunk{defun getToken}
\getchunk{defun getUnionMode}
\getchunk{defun getUniqueModemap}
\getchunk{defun getUniqueSignature}
\getchunk{defun genDomainOps}
\getchunk{defun genDomainViewList0}
\getchunk{defun genDomainViewList}
\getchunk{defun genDomainView}
\getchunk{defun giveFormalParametersValues}

\getchunk{defun hackforis}

```

```

\getchunk{defun hackforis1}
\getchunk{defun hasAplExtension}
\getchunk{defun hasFormalMapVariable}
\getchunk{defun hasFullSignature}
\getchunk{defun hasNoVowels}
\getchunk{defun hasSigInTargetCategory}
\getchunk{defun hasType}
\getchunk{defun htcharPosition}

\getchunk{defun indent-pos}
\getchunk{defun infixtok}
\getchunk{defun initialize-prepare}
\getchunk{defun initial-substring}
\getchunk{defun initial-substring-p}
\getchunk{defun initializeLisplib}
\getchunk{defun insertModemap}
\getchunk{defun interactiveModemapForm}
\getchunk{defun isCategoryPackageName}
\getchunk{defun is-console}
\getchunk{defun isDomainConstructorForm}
\getchunk{defun isDomainForm}
\getchunk{defun isDomainSubst}
\getchunk{defun isFunctor}
\getchunk{defun isListConstructor}
\getchunk{defun isMacro}
\getchunk{defun isSuperDomain}
\getchunk{defun isTokenDelimiter}
\getchunk{defun isUnionMode}

\getchunk{defun killColons}

\getchunk{defun line-advance-char}
\getchunk{defun line-at-end-p}
\getchunk{defun line-current-segment}
\getchunk{defun line-next-char}
\getchunk{defun line-past-end-p}
\getchunk{defun line-print}
\getchunk{defun line-new-line}
\getchunk{defun lispize}
\getchunk{defun lisplibDoRename}
\getchunk{defun lisplibWrite}
\getchunk{defun loadIfNecessary}
\getchunk{defun loadLibIfNecessary}

\getchunk{defun macroExpand}
\getchunk{defun macroExpandInPlace}
\getchunk{defun macroExpandList}
\getchunk{defun makeCategoryForm}
\getchunk{defun makeCategoryPredicates}
\getchunk{defun makeFunctorArgumentParameters}

```

```

\getchunk{defun makeSimplePredicateOrNil}
\getchunk{defun make-symbol-of}
\getchunk{defun match-advance-string}
\getchunk{defun match-current-token}
\getchunk{defun match-next-token}
\getchunk{defun match-string}
\getchunk{defun match-token}
\getchunk{defun maxSuperType}
\getchunk{defun mergeModemap}
\getchunk{defun mergeSignatureAndLocalVarAlists}
\getchunk{defun meta-syntax-error}
\getchunk{defun mkAbbrev}
\getchunk{defun mkAListOfExplicitCategoryOps}
\getchunk{defun mkCategoryPackage}
\getchunk{defun mkConstructor}
\getchunk{defun mkDatabasePred}
\getchunk{defun mkEvalableCategoryForm}
\getchunk{defun mkExplicitCategoryFunction}
\getchunk{defun mkList}
\getchunk{defun mkNewModemapList}
\getchunk{defun mkOpVec}
\getchunk{defun mkRepetitionAssoc}
\getchunk{defun mkUnion}
\getchunk{defun modifyModeStack}
\getchunk{defun modeEqual}
\getchunk{defun modeEqualSubst}
\getchunk{defun modemapPattern}
\getchunk{defun moveORsOutside}
\getchunk{defun mustInstantiate}

\getchunk{defun ncINTERPFILE}
\getchunk{defun newWordFrom}
\getchunk{defun next-char}
\getchunk{defun next-line}
\getchunk{defun next-tab-loc}
\getchunk{defun next-token}
\getchunk{defun newConstruct}
\getchunk{defun newDef2Def}
\getchunk{defun newIf2Cond}
\getchunk{defun newString2Words}
\getchunk{defun new2OldDefForm}
\getchunk{defun new2OldTran}
\getchunk{defun new2OldLisp}
\getchunk{defun nonblankloc}
\getchunk{defun NRTassocIndex}
\getchunk{defun NRTgetLocalIndex}
\getchunk{defun NRTgetLookupFunction}
\getchunk{defun NRTputInHead}
\getchunk{defun NRTputInTail}

```

```

\getchunk{defun optCall}
\getchunk{defun optCallEval}
\getchunk{defun optCallSpecially}
\getchunk{defun optCatch}
\getchunk{defun optCond}
\getchunk{defun optCONDtail}
\getchunk{defun optEQ}
\getchunk{defun optIF2COND}
\getchunk{defun optimize}
\getchunk{defun optimizeFunctionDef}
\getchunk{defun optional}
\getchunk{defun optLESSP}
\getchunk{defun optMINUS}
\getchunk{defun optMkRecord}
\getchunk{defun optPackageCall}
\getchunk{defun optPredicateIfTrue}
\getchunk{defun optQSMINUS}
\getchunk{defun optRECORDCOPY}
\getchunk{defun optRECORDELT}
\getchunk{defun optSETRECORDELT}
\getchunk{defun optSEQ}
\getchunk{defun optSPADCALL}
\getchunk{defun optSpecialCall}
\getchunk{defun optSuchthat}
\getchunk{defun optXLAMCond}
\getchunk{defun opt-}
\getchunk{defun orderByDependency}
\getchunk{defun orderPredicateItems}
\getchunk{defun orderPredTran}
\getchunk{defun outputComp}

\getchunk{defun PARSE-AnyId}
\getchunk{defun PARSE-Application}
\getchunk{defun parse-argument-designator}
\getchunk{defun parse-identifier}
\getchunk{defun parse-keyword}
\getchunk{defun parse-number}
\getchunk{defun parse-spadstring}
\getchunk{defun parse-string}
\getchunk{defun PARSE-Category}
\getchunk{defun PARSE-Command}
\getchunk{defun PARSE-CommandTail}
\getchunk{defun PARSE-Conditional}
\getchunk{defun PARSE-Data}
\getchunk{defun PARSE-ElseClause}
\getchunk{defun PARSE-Enclosure}
\getchunk{defun PARSE-Exit}
\getchunk{defun PARSE-Expr}
\getchunk{defun PARSE-Expression}
\getchunk{defun PARSE-Float}

```

```

\getchunk{defun PARSE-FloatBase}
\getchunk{defun PARSE-FloatBasePart}
\getchunk{defun PARSE-FloatExponent}
\getchunk{defun PARSE-FloatTok}
\getchunk{defun PARSE-Form}
\getchunk{defun PARSE-FormalParameter}
\getchunk{defun PARSE-FormalParameterTok}
\getchunk{defun PARSE-getSemanticForm}
\getchunk{defun PARSE-GlyphTok}
\getchunk{defun PARSE-Import}
\getchunk{defun PARSE-Infix}
\getchunk{defun PARSE-InfixWith}
\getchunk{defun PARSE-IntegerTok}
\getchunk{defun PARSE-Iterator}
\getchunk{defun PARSE-IteratorTail}
\getchunk{defun PARSE-Label}
\getchunk{defun PARSE-LabelExpr}
\getchunk{defun PARSE-Leave}
\getchunk{defun PARSE-LedPart}
\getchunk{defun PARSE-leftBindingPowerOf}
\getchunk{defun PARSE-Loop}
\getchunk{defun PARSE-Name}
\getchunk{defun PARSE-NBGlyphTok}
\getchunk{defun PARSE-NewExpr}
\getchunk{defun PARSE-NudPart}
\getchunk{defun PARSE-OpenBrace}
\getchunk{defun PARSE-OpenBracket}
\getchunk{defun PARSE-Operation}
\getchunk{defun PARSE-Option}
\getchunk{defun PARSE-Prefix}
\getchunk{defun PARSE-Primary}
\getchunk{defun PARSE-Primary1}
\getchunk{defun PARSE-PrimaryNoFloat}
\getchunk{defun PARSE-PrimaryOrQM}
\getchunk{defun PARSE-Qualification}
\getchunk{defun PARSE-Quad}
\getchunk{defun PARSE-Reduction}
\getchunk{defun PARSE-ReductionOp}
\getchunk{defun PARSE-Return}
\getchunk{defun PARSE-rightBindingPowerOf}
\getchunk{defun PARSE-ScriptItem}
\getchunk{defun PARSE-Scripts}
\getchunk{defun PARSE-Seg}
\getchunk{defun PARSE-Selector}
\getchunk{defun PARSE-SemiColon}
\getchunk{defun PARSE-Sequence}
\getchunk{defun PARSE-Sequence1}
\getchunk{defun PARSE-Sexpr}
\getchunk{defun PARSE-Sexpr1}
\getchunk{defun PARSE-SpecialCommand}

```

```

\getchunk{defun PARSE-SpecialKeyWord}
\getchunk{defun PARSE-Statement}
\getchunk{defun PARSE-String}
\getchunk{defun PARSE-Suffix}
\getchunk{defun PARSE-TokenCommandTail}
\getchunk{defun PARSE-TokenList}
\getchunk{defun PARSE-TokenOption}
\getchunk{defun PARSE-TokTail}
\getchunk{defun PARSE-VarForm}
\getchunk{defun PARSE-With}
\getchunk{defun parsepiles}
\getchunk{defun parseAnd}
\getchunk{defun parseAtom}
\getchunk{defun parseAtSign}
\getchunk{defun parseCategory}
\getchunk{defun parseCoerce}
\getchunk{defun parseColon}
\getchunk{defun parseConstruct}
\getchunk{defun parseDEF}
\getchunk{defun parseDollarGreaterEqual}
\getchunk{defun parseDollarGreaterThan}
\getchunk{defun parseDollarLessEqual}
\getchunk{defun parseDollarNotEqual}
\getchunk{defun parseDropAssertions}
\getchunk{defun parseEquivalence}
\getchunk{defun parseExit}
\getchunk{defun postFlatten}
\getchunk{defun postFlattenLeft}
\getchunk{defun postForm}
\getchunk{defun parseGreaterEqual}
\getchunk{defun parseGreaterThan}
\getchunk{defun parseHas}
\getchunk{defun parseHasRhs}
\getchunk{defun parseIf}
\getchunk{defun parseIf,ifTran}
\getchunk{defun parseImplies}
\getchunk{defun parseIn}
\getchunk{defun parseInBy}
\getchunk{defun parseIs}
\getchunk{defun parseIsnt}
\getchunk{defun parseJoin}
\getchunk{defun parseLeave}
\getchunk{defun parseLessEqual}
\getchunk{defun parseLET}
\getchunk{defun parseLETD}
\getchunk{defun parseLhs}
\getchunk{defun parseMDEF}
\getchunk{defun parseNot}
\getchunk{defun parseNotEqual}
\getchunk{defun parseOr}

```

```

\getchunk{defun parsePretend}
\getchunk{defun parseprint}
\getchunk{defun parseReturn}
\getchunk{defun parseSegment}
\getchunk{defun parseSeq}
\getchunk{defun parseTran}
\getchunk{defun parseTranCheckForRecord}
\getchunk{defun parseTranList}
\getchunk{defun parseTransform}
\getchunk{defun parseType}
\getchunk{defun parseVCONS}
\getchunk{defun parseWhere}
\getchunk{defun Pop-Reduction}
\getchunk{defun postAdd}
\getchunk{defun postAtom}
\getchunk{defun postAtSign}
\getchunk{defun postBigFloat}
\getchunk{defun postBlock}
\getchunk{defun postBlockItem}
\getchunk{defun postBlockItemList}
\getchunk{defun postCapsule}
\getchunk{defun postCategory}
\getchunk{defun postcheck}
\getchunk{defun postCollect}
\getchunk{defun postCollect,finish}
\getchunk{defun postColon}
\getchunk{defun postColonColon}
\getchunk{defun postComma}
\getchunk{defun postConstruct}
\getchunk{defun postDef}
\getchunk{defun postDefArgs}
\getchunk{defun postError}
\getchunk{defun postExit}
\getchunk{defun postIf}
\getchunk{defun postin}
\getchunk{defun postIn}
\getchunk{defun postInSeq}
\getchunk{defun postIteratorList}
\getchunk{defun postJoin}
\getchunk{defun postMakeCons}
\getchunk{defun postMapping}
\getchunk{defun postMDef}
\getchunk{defun postOp}
\getchunk{defun postPretend}
\getchunk{defun postQUOTE}
\getchunk{defun postReduce}
\getchunk{defun postRepeat}
\getchunk{defun postScripts}
\getchunk{defun postScriptsForm}
\getchunk{defun postSemiColon}

```

```

\getchunk{defun postSignature}
\getchunk{defun postSlash}
\getchunk{defun postTran}
\getchunk{defun postTranList}
\getchunk{defun postTranScripts}
\getchunk{defun postTranSegment}
\getchunk{defun postTransform}
\getchunk{defun postTransformCheck}
\getchunk{defun postTuple}
\getchunk{defun postTupleCollect}
\getchunk{defun postType}
\getchunk{defun postWhere}
\getchunk{defun postWith}
\getchunk{defun print-package}
\getchunk{defun preparse}
\getchunk{defun preparse1}
\getchunk{defun preparse-echo}
\getchunk{defun preparseReadLine}
\getchunk{defun preparseReadLine1}
\getchunk{defun primitiveType}
\getchunk{defun print-defun}
\getchunk{defun processFunctor}
\getchunk{defun purgeNewConstructorLines}
\getchunk{defun push-reduction}
\getchunk{defun putDomainsInScope}
\getchunk{defun putInLocalDomainReferences}

\getchunk{defun quote-if-string}

\getchunk{defun read-a-line}
\getchunk{defun recompile-lib-file-if-necessary}
\getchunk{defun recordAttributeDocumentation}
\getchunk{defun recordDocumentation}
\getchunk{defun recordHeaderDocumentation}
\getchunk{defun recordSignatureDocumentation}
\getchunk{defun replaceExitEtc}
\getchunk{defun removeBackslashes}
\getchunk{defun removeSuperfluousMapping}
\getchunk{defun replaceVars}
\getchunk{defun resolve}
\getchunk{defun reportOnFunctorCompilation}
\getchunk{defun /rf}
\getchunk{defun /rq}
\getchunk{defun /rf-1}
\getchunk{defun /RQ,LIB}
\getchunk{defun rwriteLispForm}

\getchunk{defun screenLocalLine}
\getchunk{defun setDefOp}
\getchunk{defun seteltModemapFilter}

```



```

\getchunk{defun setqMultiple}
\getchunk{defun setqMultipleExplicit}
\getchunk{defun setqSetelt}
\getchunk{defun setqSingle}
\getchunk{defun signatureTran}
\getchunk{defun skip-blanks}
\getchunk{defun skip-ifblock}
\getchunk{defun skip-to-endif}
\getchunk{defun spad}
\getchunk{defun spadCompileOrSetq}
\getchunk{defun spad-fixed-arg}
\getchunk{defun splitEncodedFunctionName}
\getchunk{defun stack-clear}
\getchunk{defun stack-load}
\getchunk{defun stack-pop}
\getchunk{defun stack-push}
\getchunk{defun storeblanks}
\getchunk{defun string2BootTree}
\getchunk{defun stripOffArgumentConditions}
\getchunk{defun stripOffSubdomainConditions}
\getchunk{defun subrname}
\getchunk{defun substituteCategoryArguments}
\getchunk{defun substituteIntoFunctorModemap}
\getchunk{defun substNames}
\getchunk{defun substVars}
\getchunk{defun s-process}

\getchunk{defun token-install}
\getchunk{defun token-lookahead-type}
\getchunk{defun token-print}
\getchunk{defun transDoc}
\getchunk{defun transDocList}
\getchunk{defun transformAndRecheckComments}
\getchunk{defun transformOperationAlist}
\getchunk{defun transImplementation}
\getchunk{defun transIs}
\getchunk{defun transIs1}
\getchunk{defun translabel}
\getchunk{defun translabel1}
\getchunk{defun TruthP}
\getchunk{defun try-get-token}
\getchunk{defun tuple2List}

\getchunk{defun uncons}
\getchunk{defun underscore}
\getchunk{defun unget-tokens}
\getchunk{defun unknownTypeError}
\getchunk{defun unloadOneConstructor}
\getchunk{defun unTuple}
\getchunk{defun updateCategoryFrameForCategory}

```

```
\getchunk{defun updateCategoryFrameForConstructor}  
  
\getchunk{defun whoOwns}  
\getchunk{defun wrapDomainSub}  
\getchunk{defun writeLib1}  
  
\getchunk{postvars}
```

Bibliography

- [1] Jenks, R.J. and Sutor, R.S. “Axiom – The Scientific Computation System” Springer-Verlag New York (1992) ISBN 0-387-97855-0
- [2] Knuth, Donald E., “Literate Programming” Center for the Study of Language and Information ISBN 0-937073-81-4 Stanford CA (1992)
- [3] Daly, Timothy, “The Axiom Wiki Website”
<http://axiom.axiom-developer.org>
- [4] Watt, Stephen, “Aldor”,
<http://www.aldor.org>
- [5] Lamport, Leslie, “Latex – A Document Preparation System”, Addison-Wesley, New York ISBN 0-201-52983-1
- [6] Ramsey, Norman “Noweb – A Simple, Extensible Tool for Literate Programming”
<http://www.eecs.harvard.edu/~nr/noweb>
- [7] Daly, Timothy, “The Axiom Literate Documentation”
<http://axiom.axiom-developer.org/axiom-website/documentation.html>
- [8] Pratt, Vaughn “Top down operator precedence” POPL ’73 Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages
hall.org.ua/halls/wizzard/pdf/Vaughan.Pratt.TDOP.pdf
- [9] Floyd, R. W. “Semantic Analysis and Operator Precedence” JACM 10, 3, 316-333 (1963)

Chapter 19

Index

Index

- +− >, [321](#)
 - defplist, [321](#)
- − >, [390](#)
 - defplist, [390](#)
- <=, [125](#)
 - defplist, [125](#)
- =>, [391](#)
 - defplist, [391](#)
- =>, [386](#)
 - defplist, [386](#)
- >, [111](#)
 - defplist, [111](#)
- >=, [110](#), [111](#)
 - defplist, [110](#), [111](#)
- *comp370-apply*
 - usedby spad, [575](#)
- *eof*
 - usedby read-a-line, [631](#)
 - usedby spad, [575](#)
- *fileactq-apply*
 - usedby spad, [575](#)
- *lisp-bin-filetype*
 - usedby recompile-lib-file-if-necessary, [627](#)
- *standard-output*
 - usedby compileFileQuietly, [628](#)
- *terminal-io*
 - usedby is-console, [555](#)
- ,, [382](#)
 - defplist, [382](#)
- −, [237](#)
 - defplist, [237](#)
- /, [397](#)
 - defplist, [397](#)
- /RQ,LIB, [572](#)
 - calledby compilerDoit, [570](#)
 - calls /rf-1, [572](#)
 - calls echo-meta[5], [572](#)
- uses \$lisplib, [572](#)
- defun, [572](#)
- /editfile
 - usedby /rf-1, [573](#)
 - usedby compFuncAdd, [271](#)
 - usedby compFuncorBody, [168](#)
 - usedby compileSpad2Cmd, [567](#)
 - usedby compiler, [563](#)
 - usedby initializeLisplib, [202](#)
 - usedby spad, [575](#)
- /major-version
 - usedby initializeLisplib, [202](#)
- /rf, [571](#)
 - calls /rf-1, [571](#)
 - uses echo-meta, [571](#)
 - defun, [571](#)
- /rf-1, [572](#)
 - calledby /RQ,LIB, [572](#)
 - calledby /rf, [571](#)
 - calledby /rq, [571](#)
 - calls makeInputFilename[5], [572](#)
 - calls ncINTERPFILE, [572](#)
 - uses /editfile, [573](#)
 - uses echo-meta, [573](#)
 - defun, [572](#)
- /rf[5]
 - called by compilerDoit, [570](#)
- /rq, [571](#)
 - calls /rf-1, [571](#)
 - uses echo-meta, [571](#)
 - defun, [571](#)
- /rq[5]
 - called by compilerDoit, [570](#)
- ., [104](#), [290](#), [380](#)
 - defplist, [104](#), [290](#), [380](#)
- ::, [103](#), [358](#), [381](#)
 - defplist, [103](#), [358](#), [381](#)

- :BF:, 375
 - defplist, 375
- ;;, 395
 - defplist, 395
- ==, 384
 - defplist, 384
- \$*eof*
 - local ref preparseReadLine, 89
- \$/editfile
 - local ref finalizeLisplib, 203
- \$AttrLst
 - local def buildLibdb, 478
- \$BasicPredicates, 227
 - local ref optPredicateIfTrue, 227
 - defvar, 227
- \$Boolean
 - local ref compArgumentConditions, 166
 - local ref compHas, 309
 - local ref doItIf, 281
 - usedby compCase1, 284
 - usedby compIf, 311
 - usedby compIs, 319
 - usedby compReduce1, 327
 - usedby compRepeatOrCollect, 329
 - usedby compSubDomain1, 347
 - usedby compSuchthat, 349
 - usedby compSymbol, 601
- \$CapsuleDomainsInScope
 - local def compDefineCapsuleFunction, 152
 - local def putDomainsInScope, 251
 - local ref getDomainsInScope, 250
- \$CapsuleModemapFrame
 - local def addModemapKnown, 268
 - local def addModemap, 269
 - local def compDefineCapsuleFunction, 152
 - local ref addModemap, 269
- \$CategoryFrame
 - local def updateCategoryFrameForCategory, 117
 - local def updateCategoryFrameForConstructor, 116
 - local ref isFunctor, 249
 - local ref loadLibIfNecessary, 115
 - local ref mkEvaluableCategoryForm, 178
 - local ref updateCategoryFrameForCategory, 117
 - local ref updateCategoryFrameForConstructor, 116
 - usedby compDefineFunctor1, 145
 - usedby compSubDomain1, 347
 - usedby compWithMappingMode1, 618
 - usedby parseHasRhs, 114
 - usedby parseHas, 112
- \$CategoryNames
 - local ref mkEvaluableCategoryForm, 178
- \$Category
 - local ref augModemapsFromDomain, 252
 - local ref compApplication, 606
 - local ref compFocompFormWithModemap, 613
 - local ref compMakeCategoryObject, 209
 - local ref mkEvaluableCategoryForm, 178
 - usedby compConstructorCategory, 297
 - usedby compDefine1, 141
 - usedby compJoin, 320
- \$CheckVectorList
 - usedby compDefineFunctor1, 145
 - usedby displayMissingFunctions, 216
- \$ConditionalOperators
 - usedby genDomainOps, 220
 - usedby makeFunctorArgumentParameters, 217
- \$ConstructorCache
 - local ref compileConstructor1, 184
- \$ConstructorNames
 - usedby compDefine1, 142
- \$DefLst
 - local def buildLibdb, 478
- \$DomLst
 - local def buildLibdb, 478
- \$DomainFrame
 - usedby s-process, 584
- \$DomainsInScope
 - local ref compDefineCapsuleFunction, 152
- \$DoubleFloat
 - usedby primitiveType, 600
- \$DummyFunctorNames
 - local ref augModemapsFromDomain, 252
 - local ref mustInstantiate, 289
- \$EchoLineStack
 - local def preparseReadLine1, 90
 - local ref preparse-echo, 92

- usedby fincomblock, 554
- `$EchoLines`
 - usedby ncINTERPFILE, 627
- `$EmptyEnvironment`
 - local ref augLisplibModemapsFromCategory, 188
 - local ref compHasFormat, 309
 - local ref getInverseEnvironment, 316
 - local ref getSuccessEnvironment, 315
 - usedby genDomainViewList, 219
 - usedby s-process, 584
- `$EmptyMode`, 172
 - local def NRTgetLocalIndex, 211
 - local ref coerceEasy, 352
 - local ref compApply, 595
 - local ref compHasFormat, 309
 - local ref compToApply, 605
 - local ref compileDocumentation, 166
 - local ref doIt, 277
 - local ref getSuccessEnvironment, 315
 - local ref makeCategoryForm, 293
 - local ref mkEvalableCategoryForm, 178
 - local ref resolve, 361
 - local ref setqMultipleExplicit, 339
 - local ref setqMultiple, 337
 - usedby compAdd, 271
 - usedby compArgumentsAndTryAgain, 616
 - usedby compCase1, 284
 - usedby compColonInside, 596
 - usedby compCons1, 294
 - usedby compDefine1, 142
 - usedby compDefineAddSignature, 143
 - usedby compDefineCategory1, 158
 - usedby compForm1, 603
 - usedby compForm2, 610
 - usedby compIs, 319
 - usedby compMacro, 323
 - usedby compNoStacking, 590
 - usedby compPretend, 325
 - usedby compSetq1, 336
 - usedby compSubDomain1, 347
 - usedby compWhere, 351
 - usedby compWithMappingModel1, 618
 - usedby primitiveType, 600
 - usedby s-process, 583
 - usedby setqSingle, 341
 - defvar, 172
- `$EmptyVector`
 - usedby compVector, 350
- `$Exit`
 - local ref coerceEasy, 352
- `$Expression`
 - local ref coerceExtraHard, 355
 - local ref compExpressionList, 609
 - local ref outputComp, 343
 - usedby compAtom, 597
 - usedby compForm1, 603
 - usedby compSymbol, 601
- `$FormalMapVariableList`, 266
 - local ref applyMapping, 594
 - local ref buildLibAttr, 485
 - local ref compDefineCategory2, 160
 - local ref compFocompFormWithModemap, 613
 - local ref compHasFormat, 309
 - local ref finalizeDocumentation, 493
 - local ref finalizeLisplib, 203
 - local ref getSignatureFromMode, 300
 - local ref getSlotFromCategoryForm, 206
 - local ref interactiveModemapForm, 191
 - local ref isDomainConstructorForm, 345
 - local ref substVars, 198
 - usedby compColon, 291
 - usedby compDefineFunctor1, 145
 - usedby compSymbol, 601
 - usedby compTypeOf, 596
 - usedby compWithMappingModel1, 618
 - usedby makeCategoryPredicates, 175
 - usedby mkCategoryPackage, 176
 - usedby mkOpVec, 221
 - usedby substNames, 266
 - defvar, 266
- `$GensymAssoc`
 - local def EqualBarGensym, 244
 - local ref EqualBarGensym, 244
- `$HTlinks`
 - local ref checkRecordHash, 510
- `$HTlisplinks`
 - local ref checkRecordHash, 510
- `$HTmacros`
 - local ref checkAddMacros, 528
- `$HTspadmacros`

- local ref checkFixCommonProblem, 508
- \$Index
 - usedby s-process, 583
- \$Information
 - local ref compMapCond", 257
- \$InitialDomainsInScope
 - usedby spad, 575
- \$InteractiveFrame
 - usedby s-process, 584
 - usedby spad, 574
- \$InteractiveMode
 - local def addConstructorModemaps, 254
 - local ref addModemap, 269
 - local ref coerce, 352
 - local ref displayPreCompilationErrors, 544
 - local ref isFunctor, 249
 - local ref loadLibIfNecessary, 115
 - local ref mkNewModemapList, 262
 - local ref optCatch, 240
 - local ref optSPADCALL, 239
 - usedby bumperrorcount, 545
 - usedby compileFileQuietly, 628
 - usedby compileSpad2Cmd, 566
 - usedby dollarTran, 465
 - usedby parseAnd, 101
 - usedby parseAtSign, 102
 - usedby parseCoerce, 104
 - usedby parseColon, 104
 - usedby parseHas, 112
 - usedby parseIf,ifTran, 118
 - usedby parseNot, 128
 - usedby postBigFloat, 376
 - usedby postDef, 385
 - usedby postError, 370
 - usedby postMDef, 391
 - usedby postReduce, 393
 - usedby spad, 575
 - usedby tuple2List, 549
- \$LocalDomainAlist
 - local def doIt, 277
 - local ref doIt, 277
 - usedby compDefineFunctor1, 145
- \$LocalFrame
 - usedby s-process, 584
- \$NRTaddForm
 - local ref NRTassocIndex, 342
- local ref NRTgetLocalIndex, 211
- usedby compAdd, 271
- usedby compDefineFunctor1, 145
- usedby compFunctorBody, 168
- usedby compSubDomain, 346
- \$NRTaddList
 - usedby compDefineFunctor1, 145
- \$NRTattributeAlist
 - usedby compDefineFunctor1, 145
- \$NRTbase
 - local def NRTgetLocalIndex, 211
 - local ref NRTassocIndex, 342
 - usedby compDefineFunctor1, 145
- \$NRTdeltaLength
 - local ref NRTassocIndex, 342
 - local ref NRTgetLocalIndex, 211
 - usedby compDefineFunctor1, 145
- \$NRTdeltaListComp
 - local ref NRTgetLocalIndex, 211
 - usedby compDefineFunctor1, 145
- \$NRTdeltaList
 - local ref NRTassocIndex, 342
 - local ref NRTgetLocalIndex, 211
 - usedby compDefineFunctor1, 145
- \$NRTderivedTargetIfTrue
 - usedby compTopLevel, 586
- \$NRTdomainFormList
 - usedby compDefineFunctor1, 145
- \$NRTloadTimeAlist
 - usedby compDefineFunctor1, 145
- \$NRTopt
 - local ref doIt, 277
- \$NRTslot1Info
 - usedby compDefineFunctor1, 145
- \$NRTslot1PredicateList
 - local def finalizeLisplib, 204
 - usedby compDefineFunctor1, 145
- \$NegativeInteger
 - usedby primitiveType, 600
- \$NoValueMode, 172
 - local ref coerceEasy, 352
 - local ref compAtomWithModemap, 598
 - local ref resolve, 361
 - local ref setqMultipleExplicit, 339
 - local ref setqMultiple, 337
 - usedby compDefine1, 142

- usedby compImport, 318
 - usedby compMacro, 323
 - usedby compRepeatOrCollect, 329
 - usedby compSeq1, 333
 - usedby compSymbol, 601
 - usedby setqSingle, 341
- defvar, 172
- \$NoValue
 - usedby compSymbol, 601
 - usedby parseAtom, 98
- \$NonMentionableDomainNames
 - local ref doIt, 277
- \$NonNegativeInteger
 - usedby primitiveType, 600
- \$NumberOfArgsIfInteger
 - usedby compForm1, 603
- \$One
 - usedby compElt, 306
- \$OpLst
 - local def buildLibdb, 479
- \$PakLst
 - local def buildLibdb, 478
- \$PatternVariableList
 - local ref augLisplibModemapsFromCategory, 188
 - local ref augmentLisplibModemapsFromFunctor, 212
 - local ref formal2Pattern, 214
 - local ref interactiveModemapForm, 191
 - local ref modemapPattern, 199
- \$PolyMode
 - usedby s-process, 583
- \$PositiveInteger
 - usedby primitiveType, 600
- \$PrettyPrint
 - usedby print-defun, 587
- \$PrintOnly
 - usedby s-process, 584
- \$QuickCode
 - local ref doIt, 277
 - local ref optCall, 229
 - local ref optSpecialCall, 232
 - local ref putInLocalDomainReferences, 185
 - usedby compDefineFunctor1, 146
 - usedby compWithMappingMode1, 618
 - usedby compileSpad2Cmd, 566
- \$QuickLet
 - usedby compileSpad2Cmd, 566
 - usedby setqSingle, 340
- \$ReadingFile
 - usedby ncINTERPFILE, 627
- \$Representation
 - local def doIt, 277
 - local ref doIt, 277
 - usedby compDefineFunctor1, 145
 - usedby compNoStacking, 590
- \$Rep
 - local ref coerce, 352
 - local ref mkUnion, 362
- \$SpecialDomainNames
 - local ref isDomainForm, 344
 - usedby addEmptyCapsuleIfNecessary, 173
- \$StringCategory
 - usedby compString, 345
- \$String
 - local ref coerceHard, 354
 - local ref resolve, 361
 - usedby primitiveType, 600
- \$Symbol
 - usedby compSymbol, 601
- \$TranslateOnly
 - usedby s-process, 584
- \$Translation
 - usedby s-process, 584
- \$TriangleVariableList
 - local ref compDefineCategory2, 160
 - usedby compForm2, 610
 - usedby makeCategoryPredicates, 176
- \$Undef
 - local ref optSpecialCall, 232
- \$VariableCount
 - usedby s-process, 584
- \$Void
 - local ref coerceEasy, 352
- \$Zero
 - usedby compElt, 306
- \$abbreviationTable
 - local def getAbbreviation, 298
 - local ref getAbbreviation, 298
- \$addFormLhs
 - usedby compAdd, 271
 - usedby compSubDomain, 346

- \$addForm
 - local def compDefineCategory2, 160
 - usedby compAdd, 271
 - usedby compCapsuleInner, 274
 - usedby compDefineFunctor1, 145
 - usedby compSubDomain, 346
- \$algebraOutputStream
 - local ref compDefineLisplib, 163
- \$alternateViewList
 - usedby makeFunctorArgumentParameters, 217
- \$argl
 - local def checkComments, 498
 - local def transDoc, 496
 - local ref checkDecorate, 504
 - local ref checkRewrite, 499
- \$argumentConditionList
 - local def addArgumentConditions, 300
 - local def compArgumentConditions, 166
 - local def compDefineCapsuleFunction, 152
 - local def stripOffArgumentConditions, 302
 - local def stripOffSubdomainConditions, 301
 - local ref addArgumentConditions, 300
 - local ref compArgumentConditions, 166
 - local ref stripOffArgumentConditions, 302
 - local ref stripOffSubdomainConditions, 301
- \$atList
 - local def compCategory, 286
 - local ref compCategoryItem, 287
 - local ref compCategory, 286
- \$attribute?
 - local def transDoc, 496
 - local ref checkComments, 498
 - local ref transDoc, 496
- \$attributesName
 - usedby compDefineFunctor1, 145
- \$base
 - local def augModemapsFromCategoryRep, \$byteVec 267
 - local def augModemapsFromCategory, 260
- \$beginEndList
 - local ref checkBeginEnd, 502
- \$bindings
 - local def compApplyModemap, 255
 - local ref compApplyModemap, 255
 - local ref compMapCond, 256
- \$body
 - local ref addArgumentConditions, 300
- \$bootStrapMode
 - local ref coerceHard, 354
 - local ref optCall, 229
 - usedby comp2, 591
 - usedby compAdd, 272
 - usedby compCapsule, 274
 - usedby compColon, 291
 - usedby compDefineCategory1, 158
 - usedby compDefineFunctor1, 145
 - usedby compFunctorBody, 168
- \$boot
 - local def string2BootTree, 72
 - local ref PARSE-FloatTok, 453
 - local ref PARSE-Primary1, 435
 - local ref aplTran1, 401
 - local ref postAtom, 367
 - usedby PARSE-Quad, 439
 - usedby PARSE-Selector, 433
 - usedby PARSE-TokTail, 430
 - usedby aplTran, 401
 - usedby postBigFloat, 376
 - usedby postColonColon, 381
 - usedby postDef, 385
 - usedby postForm, 370
 - usedby postIf, 387
 - usedby postMDef, 391
 - usedby quote-if-string, 456
 - usedby spad, 575
 - usedby tuple2List, 549
- \$byConstructors, 629
 - local ref prepare1, 84
 - usedby compilerDoit, 570
 - defvar, 629
- \$byteAddress
 - usedby compDefineFunctor1, 145
 - usedby compDefineFunctor1, 145
- \$catLst
 - local def buildLibdb, 478
- \$categoryPredicateList
 - usedby compDefineCategory1, 158
 - usedby mkCategoryPackage, 176

- \$charBack
 - local ref checkAddBackSlashes, 528
 - local ref checkBeginEnd, 502
 - local ref checkDecorate, 504
 - local ref checkRecordHash, 510
 - local ref checkSplitBackslash, 535
 - local ref checkSplitOn, 536
 - local ref checkSplitPunctuation, 537
 - local ref checkTransformFirsts, 513
 - local ref htcharPosition, 540
 - local ref removeBackslashes, 541
- \$charBlank
 - local ref checkAddSpaceSegments, 530
 - local ref checkAddSpaces, 530
 - local ref checkLookForLeftBrace, 533
 - local ref checkSkipBlanks, 534
 - local ref checkTrim, 516
 - local ref newWordFrom, 540
- \$charComma
 - local ref checkGetArgs, 521
 - local ref checkSplitPunctuation, 537
- \$charDash
 - local ref checkSplitPunctuation, 537
- \$charDelimiters
 - local ref checkSkipOpToken, 525
- \$charEscapeList
 - local ref checkAddBackSlashes, 528
- \$charExclusions
 - local ref checkDecorate, 504
- \$charFauxNewline
 - local ref checkAddSpaces, 530
 - local ref checkIndentedLines, 524
 - local ref newWordFrom, 540
- \$charIdentifierEndings
 - local ref checkAlphabetic, 531
- \$charLbrace
 - local ref checkBeginEnd, 502
 - local ref checkDecorateForHt, 506
 - local ref checkDecorate, 504
 - local ref checkFixCommonProblem, 508
 - local ref checkLookForLeftBrace, 533
 - local ref checkLookForRightBrace, 533
 - local ref checkTexht, 512
- \$charPeriod
 - local ref checkIeEgfun, 531
 - local ref checkSplitPunctuation, 537
- \$charPlus
 - local ref checkTrim, 516
- \$charQuote
 - local ref checkSplitPunctuation, 537
- \$charRbrace
 - local ref checkBeginEnd, 502
 - local ref checkDecorateForHt, 506
 - local ref checkDecorate, 504
 - local ref checkGetStringBeforeRightBrace, 523
 - local ref checkLookForRightBrace, 533
 - local ref checkTexht, 512
- \$charSemiColon
 - local ref checkSplitPunctuation, 537
- \$charSplitList
 - local ref checkSplitOn, 536
- \$checkErrorFlag
 - local def checkComments, 498
 - local def checkDocError, 517
 - local ref checkComments, 498
 - local ref checkDocError, 517
 - local ref checkRewrite, 499
- \$checkPrenAlist
 - local ref checkBalance, 501
 - local ref checkTransformFirsts, 513
- \$checkingXmptex?
 - local def transformAndRecheckComments, 497
 - local ref checkDecorateForHt, 506
 - local ref checkDecorate, 504
 - local ref checkRewrite, 500
- \$clamList
 - local def compileConstructor1, 184
 - local ref compileConstructor1, 184
- \$comblocklist, 553
 - local def collectComBlock, 471
 - local def recordHeaderDocumentation, 472
 - local ref collectAndDeleteAssoc, 472
 - local ref finalizeDocumentation, 493
 - local ref recordHeaderDocumentation, 472
 - usedby fincomblock, 554
 - usedby preparse, 80
 - defvar, 553
- \$compErrorMessageStack
 - usedby compOrCroak1, 589
- \$compForModelIfTrue

- local def compForMode, [321](#)
 - usedby compSymbol, [601](#)
- \$compStack
 - local def compOrCroak1, [589](#)
 - local ref compNoStacking1, [591](#)
 - local ref compNoStacking, [590](#)
 - local ref comp, [590](#)
- \$compTimeSum
 - usedby compTopLevel, [586](#)
- \$compUniquelyIfTrue
 - local def compUniquely, [616](#)
 - local ref compForm3, [612](#)
 - usedby s-process, [583](#)
- \$compileDocumentation
 - local ref checkDocError1, [507](#)
 - local ref compDefineLisplib, [163](#)
- \$compileOnlyCertainItems
 - local ref compDefineCapsuleFunction, [152](#)
 - local ref compile, [169](#)
 - usedby compDefineFunctor1, [145](#)
 - usedby compileSpad2Cmd, [566](#)
- \$condAlist
 - usedby compDefineFunctor1, [145](#)
- \$conform
 - local def buildLibdb, [478](#)
 - local ref buildLibAttr, [485](#)
 - local ref buildLibOp, [484](#)
 - local ref buildLibdbConEntry, [482](#)
 - local ref buildLibdb, [478](#)
- \$conname
 - local def buildLibdbConEntry, [482](#)
 - local def buildLibdb, [478](#)
 - local ref buildLibAttr, [486](#)
- \$constructorLineNumber
 - usedby preparse, [80](#)
- \$constructorName
 - local ref checkDocError, [517](#)
 - local ref checkDocMessage, [520](#)
 - local ref transDocList, [495](#)
- \$constructorsSeen, [629](#)
 - local ref preparse1, [84](#)
 - usedby compilerDoit, [570](#)
 - defvar, [629](#)
- \$createLocalLibDb
 - local ref extendLocalLibdb, [477](#)
- \$currentFunction
 - usedby s-process, [583](#)
- \$currentLine
 - usedby s-process, [584](#)
- \$currentSysList
 - local ref checkRecordHash, [510](#)
- \$defOp
 - usedby parseTransform, [97](#)
 - usedby postError, [370](#)
 - usedby postTransformCheck, [369](#)
 - usedby setDefOp, [400](#)
- \$definition
 - local def compDefineCategory2, [160](#)
 - local ref compDefineCategory2, [160](#)
- \$defstack, [407](#)
 - defvar, [407](#)
- \$devaluateList
 - local ref NRTputInTail, [185](#)
- \$doNotCompileJustPrint
 - local ref compile, [169](#)
- \$docList
 - local def recordDocumentation, [471](#)
 - local ref finalizeDocumentation, [492](#)
 - usedby postDef, [385](#)
 - usedby preparse, [80](#)
- \$doc
 - local def buildLibdbConEntry, [482](#)
 - local def buildLibdb, [478](#)
 - local ref buildLibAttr, [485](#)
 - local ref buildLibOp, [484](#)
- \$domainShell
 - local def compDefineCategory2, [160](#)
 - local ref augLisplibModemapsFromCategory, [188](#)
 - local ref hasSigInTargetCategory, [305](#)
 - usedby compDefineCategory, [158](#)
 - usedby compDefineFunctor1, [145](#)
 - usedby compDefineFunctor, [144](#)
 - usedby getOperationAlist, [265](#)
- \$echolinestack, [75](#)
 - local ref preparse1, [84](#)
 - usedby initialize-preparse, [76](#)
 - defvar, [75](#)
- \$elt
 - local def putInLocalDomainReferences, [185](#)
 - local ref NRTputInHead, [186](#)

- local ref NRTputInTail, [185](#)
- \$endTestList
 - usedby compReduce1, [327](#)
- \$envHashTable
 - usedby compTopLevel, [586](#)
- \$env
 - usedby displayMissingFunctions, [216](#)
- \$erase
 - local ref initializeLisplib, [202](#)
- \$exitModeStack
 - usedby compExit, [308](#)
 - usedby compLeave, [323](#)
 - usedby compOrCroak1, [589](#)
 - usedby compRepeatOrCollect, [329](#)
 - usedby compReturn, [331](#)
 - usedby compSeq1, [332](#)
 - usedby compSeq, [332](#)
 - usedby comp, [590](#)
 - usedby modifyModeStack, [625](#)
 - usedby s-process, [584](#)
- \$exitMode
 - local ref coerceExit, [357](#)
 - usedby s-process, [584](#)
- \$exposeFlagHeading
 - local def checkDocError, [517](#)
 - local def transformAndRecheckComments, [497](#)
 - local ref checkDocError, [517](#)
 - local ref transformAndRecheckComments, [497](#)
- \$exposeFlag
 - local ref checkDocError, [517](#)
 - local ref whoOwns, [541](#)
- \$exposed?
 - local def buildLibdbConEntry, [482](#)
 - local def buildLibdb, [478](#)
 - local ref buildLibAttr, [485](#)
 - local ref buildLibOp, [484](#)
 - local ref buildLibdbConEntry, [482](#)
- \$extraParms
 - local def compDefineCategory2, [160](#)
 - local ref compDefineCategory2, [160](#)
- \$e
 - local def NRTgetLocalIndex, [211](#)
 - local def addEltModemap, [261](#)
- local def augmentLisplibModemapsFrom-Functor, [212](#)
- local def coerceHard, [354](#)
- local def compApplyModemap, [255](#)
- local def compCapsuleItems, [276](#)
- local def compHas, [309](#)
- local def doItIf, [281](#)
- local def doIt, [277](#)
- local def mkEvaluableCategoryForm, [178](#)
- local ref addModemapKnown, [268](#)
- local ref addModemap, [269](#)
- local ref augmentLisplibModemapsFrom-Functor, [212](#)
- local ref coerceHard, [354](#)
- local ref compApplyModemap, [255](#)
- local ref compCapsuleItems, [276](#)
- local ref compHasFormat, [309](#)
- local ref compHas, [309](#)
- local ref compMakeCategoryObject, [209](#)
- local ref compMapCond", [257](#)
- local ref compSingleCapsuleItem, [276](#)
- local ref compileDocumentation, [166](#)
- local ref compile, [170](#)
- local ref doItIf, [281](#)
- local ref doIt, [277](#)
- local ref finalizeDocumentation, [492](#)
- local ref getSignature, [303](#)
- local ref getSlotFromFunctor, [208](#)
- local ref mkAlistOfExplicitCategoryOps, [189](#)
- local ref mkDatabasePred, [214](#)
- local ref mkEvaluableCategoryForm, [178](#)
- local ref optCallSpecially, [231](#)
- local ref signatureTran, [193](#)
- usedby comp3, [592](#)
- usedby compReduce1, [327](#)
- usedby genDomainOps, [220](#)
- usedby genDomainView, [220](#)
- usedby getOperationAlist, [265](#)
- usedby mkCategoryPackage, [176](#)
- usedby s-process, [584](#)
- \$fcopy
 - local ref compileDocumentation, [165](#)
- \$filep
 - local ref compDefineLisplib, [163](#)
- \$finalEnv

- local def compDefineCapsuleFunction, 152
- local def replaceExitEtc, 333
- local ref replaceExitEtc, 333
- usedby compSeq1, 333
- \$forceAdd
 - local ref mergeModemap, 263
 - local ref mkNewModemapList, 262
 - usedby compTopLevel, 586
 - usedby makeFunctorArgumentParameters,\$functionName
 - 217
- \$formalArgList
 - local def compDefineCapsuleFunction, 152
 - local def compDefineCategory2, 160
 - local ref NRTgetLocalIndex, 211
 - local ref applyMapping, 594
 - local ref compDefineCapsuleFunction, 152
 - local ref compDefineCategory2, 160
 - usedby compDefine1, 141
 - usedby compReduce, 326
 - usedby compRepeatOrCollect, 329
 - usedby compSymbol, 601
 - usedby compWithMappingMode1, 618
 - usedby compWithMappingMode, 617
 - usedby displayMissingFunctions, 216
- \$formalMapVariables
 - local def hasFormalMapVariable, 623
- \$formatArgList
 - local ref compApplication, 606
 - usedby compWithMappingMode1, 618
- \$form
 - local def compDefineCapsuleFunction, 152
 - local def compDefineCategory2, 160
 - local ref applyMapping, 594
 - local ref compApplication, 606
 - local ref compDefineCategory2, 160
 - local ref compHasFormat, 309
 - usedby compCapsuleInner, 274
 - usedby compDefine1, 141
 - usedby compDefineFunctor1, 145
 - usedby s-process, 584
 - usedby setqSingle, 341
- \$found
 - local ref NRTassocIndex, 342
- \$fromCoerceable
 - local ref autoCoerceByModemap, 360
 - local ref coerceable, 356
- local ref coerce, 352
- \$frontier
 - local def compDefineCategory2, 160
- \$functionLocations
 - local def compDefineCapsuleFunction, 152
 - local ref compDefineCapsuleFunction, 152
 - local ref transformOperationAlist, 207
 - usedby compDefineFunctor1, 145
- \$functionName
 - local ref addArgumentConditions, 300
- \$functionStats
 - local def compDefineCapsuleFunction, 152
 - local def compDefineCategory2, 160
 - local def compile, 170
 - local ref compDefineCapsuleFunction, 152
 - local ref compile, 169
 - usedby compDefineFunctor1, 145
 - usedby reportOnFunctorCompilation, 215
- \$functorForm
 - local def compDefineCategory2, 160
 - local ref addModemap0, 269
 - local ref compile, 169
 - local ref getSpecialCaseAssoc, 300
 - usedby compAdd, 272
 - usedby compCapsule, 274
 - usedby compDefineFunctor1, 146
 - usedby compFunctorBody, 168
 - usedby getOperationAlist, 265
- \$functorLocalParameters
 - local def doItIf, 281
 - local def doIt, 277
 - local ref doItIf, 281
 - local ref doIt, 277
 - usedby compCapsuleInner, 274
 - usedby compDefineFunctor1, 146
 - usedby compSymbol, 601
- \$functorSpecialCases
 - local ref getSpecialCaseAssoc, 300
 - usedby compDefineFunctor1, 146
- \$functorStats
 - local def compDefineCategory2, 160
 - local ref compDefineCapsuleFunction, 152
 - usedby compDefineFunctor1, 146
 - usedby reportOnFunctorCompilation, 215
- \$functorTarget
 - usedby compDefineFunctor1, 146

- \$functorsUsed
 - local def doIt, [277](#)
 - local ref doIt, [277](#)
 - usedby compDefineFunctor1, [146](#)
- \$funnameTail
 - usedby compWithMappingMode1, [618](#)
- \$funname
 - usedby compWithMappingMode1, [618](#)
- \$f
 - usedby compileSpad2Cmd, [566](#)
- \$genFVar
 - usedby compDefineFunctor1, [146](#)
 - usedby s-process, [584](#)
- \$genSDVar
 - usedby compDefineFunctor1, [146](#)
 - usedby s-process, [584](#)
- \$genno
 - local def aplTran, [401](#)
 - local def doIt, [277](#)
- \$getDomainCode
 - local def compDefineCategory2, [160](#)
 - local ref compileCases, [167](#)
 - local ref doItIf, [281](#)
 - local ref optCallSpecially, [231](#)
 - usedby compCapsuleInner, [274](#)
 - usedby compDefineFunctor1, [146](#)
 - usedby genDomainOps, [220](#)
 - usedby genDomainView, [220](#)
- \$glossHash
 - local def checkRecordHash, [510](#)
 - local ref checkRecordHash, [510](#)
- \$goGetList
 - usedby compDefineFunctor1, [146](#)
- \$headerDocumentation
 - local def recordHeaderDocumentation, [472](#)
 - local ref recordHeaderDocumentation, [472](#)
 - usedby postDef, [385](#)
 - usedby preparse, [80](#)
- \$htHash
 - local def checkRecordHash, [510](#)
 - local ref checkRecordHash, [510](#)
- \$htMacroTable
 - local ref checkArguments, [501](#)
 - local ref checkBeginEnd, [502](#)
 - local ref checkSplitPunctuation, [537](#)
- \$in-stream
 - local ref Advance-Char, [637](#)
 - local ref next-line, [636](#)
 - local ref preparse1, [85](#)
- \$index, [75](#)
 - local def preparse1, [84](#)
 - local def preparseReadLine1, [90](#)
 - local ref preparse1, [84](#)
 - local ref preparseReadLine1, [90](#)
 - usedby initialize-preparse, [76](#)
 - usedby preparse, [80](#)
 - defvar, [75](#)
- \$initCapsuleErrorCount
 - local def compDefineCapsuleFunction, [152](#)
- \$initList
 - usedby compReduce1, [327](#)
- \$insideCapsuleFunctionIfTrue
 - local def compDefineCapsuleFunction, [152](#)
 - local ref CapsuleModemapFrame, [268](#)
 - local ref addEltModemap, [261](#)
 - local ref addModemap, [269](#)
 - local ref compile, [169](#)
 - local ref getDomainsInScope, [250](#)
 - local ref putDomainsInScope, [251](#)
 - local ref spadCompileOrSetq, [182](#)
 - usedby compDefine1, [141](#)
 - usedby s-process, [584](#)
- \$insideCategoryIfTrue
 - local def compDefineCategory2, [160](#)
 - usedby compCapsuleInner, [274](#)
 - usedby compColon, [291](#)
 - usedby compDefine1, [141](#)
 - usedby s-process, [584](#)
- \$insideCategoryPackageIfTrue
 - local ref getFormModemaps, [607](#)
 - usedby compCapsuleInner, [274](#)
 - usedby compDefineCategory1, [158](#)
 - usedby compDefineFunctor1, [146](#)
- \$insideCoerceInteractiveHardIfTrue
 - usedby s-process, [584](#)
- \$insideCompTypeOf
 - usedby comp3, [592](#)
 - usedby compTypeOf, [596](#)
- \$insideConstructIfTrue
 - local ref parseColon, [104](#)
 - usedby parseConstruct, [99](#)
- \$insideExpressionIfTrue

- local def compDefineCapsuleFunction, 152
- usedby compCapsule, 274
- usedby compColon, 291
- usedby compDefine1, 141, 142
- usedby compExpression, 135
- usedby compMakeDeclaration, 624
- usedby compSeq1, 332
- usedby compWhere, 351
- usedby s-process, 584
- \$insideFunctorIfTrue
 - local ref compileCases, 167
 - usedby compColon, 291
 - usedby compDefine1, 141
 - usedby compDefineCategory, 158
 - usedby compDefineFunctor1, 146
 - usedby getOperationAlist, 265
 - usedby s-process, 584
- \$insidePostCategoryIfTrue
 - usedby postCategory, 377
 - usedby postWith, 400
- \$insideSetqSingleIfTrue
 - usedby setqSingle, 340
- \$insideWhereIfTrue
 - usedby compDefine1, 142
 - usedby compWhere, 351
 - usedby s-process, 584
- \$is-eqlist, 408
 - defvar, 408
- \$is-gensymlist, 408
 - defvar, 408
- \$is-spill-list, 407
 - defvar, 407
- \$is-spill, 407
 - defvar, 407
- \$isOpPackageName
 - usedby compDefineFunctor1, 146
- \$keywords
 - local ref escape-keywords, 457
- \$killOptimizeIfTrue
 - usedby compTopLevel, 586
 - usedby compWithMappingMode1, 618
- \$kind
 - local def buildLibdbConEntry, 482
 - local def buildLibdb, 478
 - local ref buildLibAttr, 485
 - local ref buildLibOp, 484
 - local ref buildLibdbConEntry, 482
- \$leaveLevelStack
 - usedby compLeave, 323
 - usedby compRepeatOrCollect, 329
 - usedby s-process, 584
- \$leaveMode
 - usedby s-process, 584
- \$level
 - usedby compOrCroak1, 589
- \$lhsOfColon
 - local def evalAndSub, 265
 - usedby compColon, 291
 - usedby compSubsetCategory, 348
- \$lhs
 - usedby parseDEF, 105
 - usedby parseMDEF, 127
- \$libFile
 - local def compDefineLisplib, 164
 - local def initializeLisplib, 202
 - local ref compDefineCategory2, 160
 - local ref compilerDoitWithScreenedLisplib, 570
 - local ref finalizeLisplib, 203
 - local ref initializeLisplib, 202
 - local ref rwriteLispForm, 200
 - usedby compDefineFunctor1, 146
- \$line-handler
 - local ref next-line, 636
- \$linelist, 75
 - local def preparseReadLine1, 90
 - local ref preparse1, 84
 - local ref preparseReadLine1, 90
 - usedby initialize-preparse, 76
 - defvar, 75
- \$line
 - usedby Advance-Char, 637
 - usedby current-char, 463
 - usedby line-advance-char, 635
 - usedby line-at-end-p, 634
 - usedby line-clear, 634
 - usedby line-new-line, 636
 - usedby line-next-char, 635
 - usedby line-past-end-p, 635
 - usedby line-print, 634, 636
 - usedby match-advance-string, 455
 - usedby match-string, 454

- \$lispHash
 - local def checkRecordHash, 510
 - local ref checkRecordHash, 510
- \$lisplibAbbreviation
 - local def compDefineCategory2, 160
 - local def compDefineLisplib, 164
 - local def initializeLisplib, 202
 - local ref finalizeLisplib, 204
 - usedby compDefineFunctor1, 146
- \$lisplibAncestors
 - local def compDefineCategory2, 160
 - local def compDefineLisplib, 164
 - local def initializeLisplib, 202
 - local ref finalizeLisplib, 204
 - usedby compDefineFunctor1, 146
- \$lisplibAttributes
 - local ref finalizeLisplib, 203
- \$lisplibCategoriesExtended
 - local def compDefineLisplib, 163
 - usedby compDefineFunctor1, 146
- \$lisplibCategory
 - local def compDefineCategory2, 160
 - local def compDefineLisplib, 164
 - local def finalizeLisplib, 204
 - local ref compDefineCategory2, 160
 - local ref finalizeLisplib, 203
 - usedby compDefineCategory1, 158
 - usedby compDefineCategory, 158
 - usedby compDefineFunctor1, 146
- \$lisplibForm
 - local def compDefineCategory2, 160
 - local def compDefineLisplib, 163
 - local def initializeLisplib, 202
 - local ref finalizeDocumentation, 492
 - local ref finalizeLisplib, 203
 - usedby compDefineFunctor1, 146
- \$lisplibFunctionLocations
 - usedby compDefineFunctor1, 146
- \$lisplibItemsAlreadyThere
 - local ref compile, 169
- \$lisplibKind
 - local def compDefineCategory2, 160
 - local def compDefineLisplib, 164
 - local def initializeLisplib, 202
 - local ref compDefineLisplib, 163
 - local ref finalizeLisplib, 203
- usedby compDefineFunctor1, 146
- \$lisplibMissingFunctions
 - usedby compDefineFunctor1, 146
- \$lisplibModemapAlist
 - local def augLisplibModemapsFromCategory, 188
 - local def augmentLisplibModemapsFromFunctor, 212
 - local def compDefineLisplib, 164
 - local def initializeLisplib, 202
 - local ref augLisplibModemapsFromCategory, 188
 - local ref augmentLisplibModemapsFromFunctor, 212
 - local ref finalizeLisplib, 203
- \$lisplibModemap
 - local def compDefineCategory2, 160
 - local def compDefineLisplib, 164
 - local def initializeLisplib, 202
 - local ref finalizeLisplib, 203
 - usedby compDefineFunctor1, 146
- \$lisplibOpAlist
 - local def initializeLisplib, 202
- \$lisplibOperationAlist
 - local def compDefineLisplib, 164
 - local def initializeLisplib, 202
 - local ref getSlotFromFunctor, 208
 - usedby compDefineFunctor1, 146
- \$lisplibParents
 - local def compDefineCategory2, 160
 - local def compDefineLisplib, 163
 - local ref finalizeLisplib, 204
 - usedby compDefineFunctor1, 146
- \$lisplibPredicates
 - local def compDefineLisplib, 163
 - local ref finalizeLisplib, 204
- \$lisplibSignatureAlist
 - local def encodeFunctionName, 179
 - local def initializeLisplib, 202
 - local ref encodeFunctionName, 179
 - local ref finalizeLisplib, 203
- \$lisplibSlot1
 - local def compDefineLisplib, 164
 - local ref finalizeLisplib, 204
 - usedby compDefineFunctor1, 146
- \$lisplibSuperDomain

- local def compDefineLisplib, 164
- local def initializeLisplib, 202
- local ref finalizeLisplib, 203
- usedby compSubDomain1, 347
- \$lisplibVariableAlist
 - local def compDefineLisplib, 164
 - local def initializeLisplib, 202
 - local ref finalizeLisplib, 203
- \$lisplib
 - local def compDefineLisplib, 163
 - local ref compDefineCategory2, 160
 - local ref compile, 169
 - local ref encodeFunctionName, 179
 - local ref lisplibWrite, 209
 - local ref rwriteLispForm, 200
 - usedby /RQ,LIB, 572
 - usedby comp2, 591
 - usedby compDefineCategory, 158
 - usedby compDefineFunctor1, 145
 - usedby compDefineFunctor, 144
- \$lookupFunction
 - usedby compDefineFunctor1, 146
- \$macroIfTrue
 - local def compDefine, 140
 - local ref compile, 169
 - usedby compMacro, 323
- \$macroassoc
 - usedby s-process, 583
- \$maxSignatureLineNumber
 - local def recordDocumentation, 471
 - local ref recordHeaderDocumentation, 472
 - usedby postDef, 385
 - usedby preparse, 80
- \$mutableDomains
 - usedby compDefineFunctor1, 146
- \$mutableDomain
 - local ref compileConstructor1, 184
 - usedby compDefineFunctor1, 146
- \$mvl
 - usedby makeCategoryPredicates, 176
- \$myFunctorBody
 - local def compCapsuleItems, 275
 - usedby compDefineFunctor1, 146
- \$m
 - usedby compileSpad2Cmd, 566
- \$name
 - local def transformAndRecheckComments, 497
 - local ref checkRecordHash, 510
- \$new2OldRenameAssoc
 - local ref new2OldTran, 72
- \$newCompilerUnionFlag
 - usedby compColonInside, 596
 - usedby compPretend, 325
- \$newComp
 - usedby compileSpad2Cmd, 566
- \$newConlist, 557
 - local def compDefineLisplib, 164
 - local ref compDefineLisplib, 163
 - usedby compileSpad2Cmd, 567
 - usedby compiler, 563
 - defvar, 557
- \$newConstructorList
 - local def extendLocalLibdb, 477
 - local ref extendLocalLibdb, 477
- \$newspad
 - usedby s-process, 583
- \$noEnv
 - local ref setqMultiple, 337
 - usedby compColon, 291
- \$noParseCommands
 - local ref PARSE-SpecialCommand, 419
- \$noSubsumption
 - usedby spad, 574
- \$optimizableConstructorNames
 - local ref optCallSpecially, 231
- \$options
 - usedby compileSpad2Cmd, 566
 - usedby compileSpadLispCmd, 568
 - usedby compiler, 563
 - usedby mkCategoryPackage, 176
- \$op
 - local def compDefineCapsuleFunction, 152
 - local def compDefineCategory2, 160
 - local def compDefineLisplib, 163
 - local ref applyMapping, 594
 - local ref compApplication, 606
 - local ref compDefineCapsuleFunction, 152
 - local ref compDefineCategory2, 160
 - local ref finalizeDocumentation, 492
 - usedby compDefine1, 141
 - usedby compDefineFunctor1, 146

- usedby compSubDomain1, 347
 - usedby parseDollarGreaterEqual, 108
 - usedby parseDollarGreaterThan, 108
 - usedby parseDollarLessEqual, 109
 - usedby parseDollarNotEqual, 109
 - usedby parseGreaterEqual, 111
 - usedby parseGreaterThan, 112
 - usedby parseLessEqual, 125
 - usedby parseNotEqual, 129
 - usedby parseTran, 97
 - usedby reportOnFunctorCompilation, 215
- \$origin
 - local def transformAndRecheckComments, 497
 - local ref checkRecordHash, 510
- \$out-stream
 - local ref line-print, 634
 - local ref print-package, 549
- \$outStream
 - local def buildLibdb, 478
 - local def dbWriteLines, 481
 - local ref buildLibdb, 478
 - local ref checkDocError, 517
 - local ref dbWriteLines, 481
- \$packagesUsed
 - local def compDefine, 141
 - local def doIt, 277
 - local ref doIt, 277
 - usedby comp2, 591
 - usedby compAdd, 271
 - usedby compTopLevel, 586
- \$pairlis
 - local def finalizeLisplib, 204
 - local ref NRTgetLookupFunction, 210
 - usedby compDefineFunctor1, 146
- \$postStack
 - local ref displayPreCompilationErrors, 544
 - usedby postError, 370
 - usedby s-process, 584
- \$predAlist
 - usedby compDefWhereClause, 156
- \$predl
 - local ref doItIf, 281
 - local ref doIt, 277
- \$pred
 - local ref compCapsuleItems, 276
 - local ref compSingleCapsuleItem, 276
- \$prefix
 - local ref applyMapping, 594
 - local ref compApplication, 606
 - local ref compile, 170
 - usedby compDefine1, 141
 - usedby compDefineCategory2, 160
- \$preparse-last-line, 76
 - local def preparse1, 84
 - local def preparseReadLine1, 90
 - local ref preparse1, 84
 - usedby initialize-preparse, 76
 - usedby preparse, 80
 - defvar, 76
- \$preparseReportIfTrue
 - usedby preparse, 80
- \$previousTime
 - usedby s-process, 584
- \$profileAlist
 - usedby compDefineFunctor, 144
- \$profileCompiler
 - local ref compDefineCapsuleFunction, 152
 - local ref finalizeLisplib, 204
 - usedby compDefineFunctor, 144
 - usedby setqSingle, 341
- \$recheckingFlag
 - local def transformAndRecheckComments, 497
 - local ref checkDocError, 517
- \$reportExitModeStack
 - usedby modifyModeStack, 625
- \$reportOptimization
 - local ref optimizeFunctionDef, 224
- \$resolveTimeSum
 - usedby compTopLevel, 586
- \$returnMode
 - local def compDefineCapsuleFunction, 152
 - local ref compDefineCapsuleFunction, 152
 - usedby compReturn, 331
 - usedby s-process, 584
- \$savableItems
 - local def compile, 170
- \$saveableItems
 - local ref compilerDoitWithScreenedLisplib, 569
 - local ref compile, 169

- \$scanIfTrue
 - usedby compOrCroak1, 589
 - usedby compileSpad2Cmd, 566
- \$semanticErrorStack
 - local ref compDefineCapsuleFunction, 152
 - usedby reportOnFunctorCompilation, 215
 - usedby s-process, 584
- \$setOptions
 - local ref checkRecordHash, 510
- \$setelt
 - usedby compDefineFunctor1, 146
- \$sideEffectsList
 - usedby compReduce1, 327
- \$sigAlist
 - usedby compDefWhereClause, 156
- \$sigList
 - local def compCategory, 286
 - local ref compCategoryItem, 287
 - local ref compCategory, 286
- \$signatureOfForm
 - local def compCapsuleItems, 276
 - local def compDefineCapsuleFunction, 152
 - local ref compDefineCapsuleFunction, 152
 - local ref compile, 169, 170
 - local ref doIt, 277
- \$signature
 - usedby compCapsuleInner, 274
 - usedby compDefineFunctor1, 146
- \$skipme
 - local def preparse1, 84
 - usedby preparse, 80
- \$sourceFileTypes
 - usedby compileSpad2Cmd, 566
- \$spad-errors
 - usedby bumperrorcount, 545
- \$spadLibFT
 - local ref compDefineLisplib, 163
 - local ref compileDocumentation, 165
 - local ref finalizeLisplib, 204
 - local ref lisplibDoRename, 201
- \$spad
 - local def string2BootTree, 72
 - usedby quote-if-string, 456
 - usedby spad, 575
- \$specialCaseKeyList
 - local def compileCases, 167
- local ref optCallSpecially, 231
- \$splitUpItemsAlreadyThere
 - local ref compile, 169
- \$stack
 - usedby reduce-stack, 468
 - usedby stack-/empty, 93
 - usedby stack-clear, 93
 - usedby stack-load, 92
 - usedby stack-pop, 94
 - usedby stack-push, 93
- \$stringFauxNewline
 - local ref newWordFrom, 540
- \$suffix
 - local def compCapsuleItems, 276
 - local def compile, 170
 - local ref compile, 170
- \$sysHash
 - local def checkRecordHash, 510
 - local ref checkRecordHash, 510
- \$s
 - usedby compOrCroak1, 589
- \$template
 - usedby compDefineFunctor1, 146
- \$tokenCommands
 - local ref PARSE-SpecialCommand, 419
- \$token
 - usedby current-token, 95
 - usedby make-symbol-of, 460
 - usedby match-advance-string, 455
 - usedby next-token, 95
 - usedby prior-token, 94
 - usedby token-install, 96
 - usedby token-print, 96
 - usedby valid-tokens, 95
- \$top-level
 - local def compCapsuleItems, 275
 - local def compCategory, 286
 - local def compDefineCategory2, 160
 - usedby compDefineFunctor1, 145
 - usedby s-process, 584
- \$topOp
 - local ref displayPreCompilationErrors, 544
 - usedby s-process, 584
 - usedby setDefOp, 400
- \$tripleCache
 - local def compDefine, 140

- \$tripleHits
 - local def compDefine, 140
- \$true
 - local ref addArgumentConditions, 300
 - local ref optCONDtail, 226
 - local ref optIF2COND, 227
- \$ttl
 - usedby makeCategoryPredicates, 176
- \$uncondAlist
 - usedby compDefineFunctor1, 146
- \$until
 - usedby compReduce1, 327
 - usedby compRepeatOrCollect, 329
- \$viewNames
 - usedby compDefineFunctor1, 146
- \$vl, 408
 - defvar, 408
- \$warningStack
 - usedby reportOnFunctorCompilation, 215
 - usedby s-process, 584
- \$why
 - local def NRTgetLookupFunction, 210
 - local ref NRTgetLookupFunction, 210
- \$x
 - local def transDoc, 496
 - local def transformAndRecheckComments, 497
 - local ref checkDocMessage, 520
 - local ref checkTrim, 516
 - local ref transDoc, 496
- , 31
- abbreviation?
 - calledby checkIsValidType, 532
 - calledby checkNumOfArgs, 534
 - calledby parseHasRhs, 114
- abbreviationsSpad2Cmd
 - calledby mkCategoryPackage, 176
- action, 467
 - calledby PARSE-AnyId, 444
 - calledby PARSE-Category, 424
 - calledby PARSE-CommandTail, 421
 - calledby PARSE-Data, 442
 - calledby PARSE-FloatExponent, 437
 - calledby PARSE-GlyphTok, 444
 - calledby PARSE-Infix, 429
 - calledby PARSE-NBGlyphTok, 443
 - calledby PARSE-NewExpr, 417
 - calledby PARSE-OpenBrace, 446
 - calledby PARSE-OpenBracket, 446
 - calledby PARSE-Operation, 427
 - calledby PARSE-Prefix, 428
 - calledby PARSE-ReductionOp, 431
 - calledby PARSE-Sexpr1, 442
 - calledby PARSE-SpecialCommand, 419
 - calledby PARSE-SpecialKeyWord, 418
 - calledby PARSE-Suffix, 448
 - calledby PARSE-TokTail, 430
 - calledby PARSE-TokenCommandTail, 420
 - calledby PARSE-TokenList, 420
 - defun, 467
- add, 372
 - defplist, 372
- add-parens-and-semis-to-line, 88
 - calledby parsepiles, 87
 - calls addclose, 88
 - calls drop, 88
 - calls infixtok, 88
 - calls nonblankloc, 88
 - defun, 88
- addArgumentConditions, 300
 - calledby compDefineCapsuleFunction, 152
 - calls mkq, 300
 - calls systemErrorHere, 300
 - local def \$argumentConditionList, 300
 - local ref \$argumentConditionList, 300
 - local ref \$body, 300
 - local ref \$functionName, 300
 - local ref \$true, 300
 - defun, 300
- addBinding
 - calledby addModemap1, 270
 - calledby compDefineCategory2, 159
 - calledby getSuccessEnvironment, 315
 - calledby setqMultiple, 337
- addBinding[5]
 - called by setqSingle, 340
 - called by spad, 574
- addclose, 552
 - calledby add-parens-and-semis-to-line, 88
 - calls suffix, 552

- defun, 552
- addConstructorModemaps, 254
 - calledby augModemapsFromDomain1, 252
 - calls addModemap, 254
 - calls getl, 254
 - calls putDomainsInScope, 254
 - local def \$InteractiveMode, 254
 - defun, 254
- AddContour
 - calledby compApply, 595
- addContour
 - calledby compWhere, 351
- addDomain, 248
 - calledby comp2, 591
 - calledby comp3, 592
 - calledby compAtSign, 357
 - calledby compCapsule, 274
 - calledby compCase, 283
 - calledby compCoerce, 358
 - calledby compColonInside, 596
 - calledby compColon, 290
 - calledby compDefineCapsuleFunction, 151
 - calledby compElt, 306
 - calledby compForm1, 603
 - calledby compImport, 318
 - calledby compPretend, 324
 - calledby compSubDomain1, 346
 - calls addNewDomain, 248
 - calls constructor?, 248
 - calls domainMember, 248
 - calls getDomainsInScope, 248
 - calls getmode, 248
 - calls identp, 248
 - calls isCategoryForm, 248
 - calls isFunctor, 248
 - calls isLiteral, 248
 - calls member, 248
 - calls qslessp, 248
 - calls unknownTypeError, 248
 - defun, 248
- addEltModemap, 261
 - calledby addModemap0, 269
 - calls addModemap1, 261
 - calls makeLiteral, 261
 - calls systemErrorHere, 261
 - local def \$e, 261
- local ref \$insideCapsuleFunctionIfTrue, 261
- defun, 261
- addEmptyCapsuleIfNecessary, 173
 - calledby compDefine1, 141
 - calls kar, 173
 - uses \$SpecialDomainNames, 173
 - defun, 173
- addInformation
 - calledby compCapsuleInner, 274
- addModemap, 269
 - calledby addConstructorModemaps, 254
 - calledby augModemapsFromCategoryRep, 267
 - calledby genDomainOps, 220
 - calledby updateCategoryFrameForCategory, 117
 - calledby updateCategoryFrameForConstructor, 116
 - calls addModemap0, 269
 - calls knownInfo, 269
 - local def \$CapsuleModemapFrame, 269
 - local ref \$CapsuleModemapFrame, 269
 - local ref \$InteractiveMode, 269
 - local ref \$e, 269
 - local ref \$insideCapsuleFunctionIfTrue, 269
 - defun, 269
- addModemap0, 269
 - calledby addModemapKnown, 268
 - calledby addModemap, 269
 - calls addEltModemap, 269
 - calls addModemap1, 269
 - local ref \$functorForm, 269
 - defun, 269
- addModemap1, 270
 - calledby addEltModemap, 261
 - calledby addModemap0, 269
 - calls addBinding, 270
 - calls augProplist, 270
 - calls getProplist, 270
 - calls lassoc, 270
 - calls mkNewModemapList, 270
 - calls unErrorRef, 270
 - defun, 270
- addModemapKnown, 268

- calledby augModemapsFromCategory, 260
- calls addModemap0, 268
- local def \$CapsuleModemapFrame, 268
- local ref \$e, 268
- defun, 268
- addNewDomain, 251
 - calledby addDomain, 248
 - calledby augModemapsFromDomain, 252
 - calls augModemapsFromDomain, 251
 - defun, 251
- addoptions
 - calledby initializeLisplib, 202
- addStats
 - calledby compDefineCapsuleFunction, 152
 - calledby compile, 169
 - calledby reportOnFunctorCompilation, 215
- addSuffix, 299
 - calledby mkAbbrev, 298
 - defun, 299
- Advance-Char, 637
 - calls Line-Advance-Char, 637
 - calls Line-At-End-P, 637
 - calls current-char, 637
 - calls next-line, 637
 - local ref \$in-stream, 637
 - uses \$line, 637
 - defun, 637
- advance-char
 - calledby skip-blanks, 454
- advance-token, 462
 - calledby PARSE-AnyId, 444
 - calledby PARSE-FloatExponent, 437
 - calledby PARSE-GlyphTok, 444
 - calledby PARSE-Infix, 429
 - calledby PARSE-NBGlyphTok, 443
 - calledby PARSE-OpenBrace, 446
 - calledby PARSE-OpenBracket, 446
 - calledby PARSE-Prefix, 429
 - calledby PARSE-ReductionOp, 431
 - calledby PARSE-Suffix, 448
 - calledby PARSE-TokenList, 420
 - calledby parse-argument-designator, 548
 - calledby parse-identifier, 547
 - calledby parse-keyword, 548
 - calledby parse-number, 547
 - calledby parse-spadstring, 546
 - calledby parse-string, 546
 - calls copy-token, 462
 - calls current-token, 462
 - calls try-get-token, 462
 - uses current-token, 462
 - uses valid-tokens, 462
 - defun, 462
- alistSize, 299
 - calledby mkAbbrev, 298
 - defun, 299
- allConstructors[5]
 - called by buildLibdb, 478
- allLASSOCs, 213
 - calledby augmentLisplibModemapsFrom-Functor, 212
 - defun, 213
- and, 101
 - defplist, 101
- aplTran, 401
 - calledby postTransform, 365
 - calls aplTran1, 401
 - calls containsBang, 401
 - local def \$genno, 401
 - uses \$boot, 401
 - defun, 401
- aplTran1, 401
 - calledby aplTran1, 401
 - calledby aplTranList, 403
 - calledby aplTran, 401
 - calledby hasAplExtension, 403
 - calls aplTran1, 401
 - calls aplTranList, 401
 - calls hasAplExtension, 401
 - calls nreverse0, 401
 - local ref \$boot, 401
 - defun, 401
- aplTranList, 403
 - calledby aplTran1, 401
 - calledby aplTranList, 403
 - calls aplTran1, 403
 - calls aplTranList, 403
 - defun, 403
- applyMapping, 593
 - calledby comp3, 592
 - calls comp, 593
 - calls convert, 593

- calls encodeItem, 594
- calls getAbbreviation, 594
- calls get, 593
- calls isCategoryForm, 593
- calls member, 593
- calls sublis, 593
- local ref \$FormalMapVariableList, 594
- local ref \$formalArgList, 594
- local ref \$form, 594
- local ref \$op, 594
- local ref \$prefix, 594
- defun, 593
- argsToSig, 623
 - calledby compLambda, 321
 - defun, 623
- assignError, 342
 - calledby setqSingle, 340
 - calls stackMessage, 342
 - defun, 342
- assoc
 - calledby augModemapsFromCategoryRep, 267
 - calledby checkBalance, 501
 - calledby compColon, 290
 - calledby compDefineAddSignature, 143
 - calledby compForm2, 610
 - calledby mkCategoryPackage, 176
 - calledby mkNewModemapList, 262
 - calledby mkOpVec, 221
 - calledby stripOffSubdomainConditions, 301
 - calledby transformOperationAlist, 207
- AssocBarGensym, 222
 - calledby mkOpVec, 221
 - calls EqualBarGensym, 222
 - defun, 222
- assocleft
 - calledby compDefWhereClause, 156
 - calledby compileCases, 167
 - calledby finalizeDocumentation, 492
 - calledby mkAlistOfExplicitCategoryOps, 189
- assocright
 - calledby compDefWhereClause, 156
 - calledby compileCases, 167
 - calledby recordHeaderDocumentation, 472
- assq
 - calledby getAbbreviation, 298
 - calledby makeFunctorArgumentParameters, 217
 - calledby mkOpVec, 221
- assq[5]
 - called by freelist, 625
- atEndOfLine
 - calledby PARSE-TokenCommandTail, 420
- augLisplibModemapsFromCategory, 187
 - calledby compDefineCategory2, 160
 - calls interactiveModemapForm, 188
 - calls isCategoryForm, 188
 - calls lassoc, 188
 - calls member, 188
 - calls mkAlistOfExplicitCategoryOps, 188
 - calls mkpf, 188
 - calls sublis, 187
 - local def \$lisplibModemapAlist, 188
 - local ref \$EmptyEnvironment, 188
 - local ref \$PatternVariableList, 188
 - local ref \$domainShell, 188
 - local ref \$lisplibModemapAlist, 188
 - defun, 187
- augmentLisplibModemapsFromFunctor, 212
 - calledby compDefineFunctor1, 145
 - calls allLASSOCs, 212
 - calls formal2Pattern, 212
 - calls interactiveModemapForm, 212
 - calls listOfPatternIds, 212
 - calls member, 212
 - calls mkAlistOfExplicitCategoryOps, 212
 - calls mkDatabasePred, 212
 - calls mkpf, 212
 - local def \$e, 212
 - local def \$lisplibModemapAlist, 212
 - local ref \$PatternVariableList, 212
 - local ref \$e, 212
 - local ref \$lisplibModemapAlist, 212
 - defun, 212
- augModemapsFromCategory, 260
 - calledby augModemapsFromDomain1, 252
 - calledby compDefineFunctor1, 144
 - calledby genDomainView, 219
 - calls addModemapKnown, 260
 - calls compilerMessage, 260
 - calls evalAndSub, 260

- calls putDomainsInScope, 260
- local def \$base, 260
- defun, 260
- augModemapsFromCategoryRep, 267
 - calledby compDefineFunctor1, 144
 - calls addModemap, 267
 - calls assoc, 267
 - calls compilerMessage, 267
 - calls evalAndSub, 267
 - calls isCategory, 267
 - calls putDomainsInScope, 267
 - local def \$base, 267
 - defun, 267
- augModemapsFromDomain, 252
 - calledby addNewDomain, 251
 - calls addNewDomain, 252
 - calls augModemapsFromDomain1, 252
 - calls getDomainsInScope, 252
 - calls getdatabase, 252
 - calls kar, 252
 - calls listOrVectorElementNode, 252
 - calls member, 252
 - calls opOf, 252
 - calls stripUnionTags, 252
 - local ref \$Category, 252
 - local ref \$DummyFunctorNames, 252
 - defun, 252
- augModemapsFromDomain1, 252
 - calledby augModemapsFromDomain, 252
 - calledby compForm1, 603
 - calledby setqSingle, 340
 - calls addConstructorModemaps, 252
 - calls augModemapsFromCategory, 252
 - calls getl, 252
 - calls getmodeOrMapping, 253
 - calls getmode, 252
 - calls kar, 252
 - calls stackMessage, 253
 - calls substituteCategoryArguments, 253
 - defun, 252
- augProplist
 - calledby addModemap1, 270
- autoCoerceByModemap, 360
 - calledby coerceExtraHard, 355
 - calls getModemapList, 360
 - calls get, 360
 - calls member, 360
 - calls modeEqual, 360
 - calls stackMessage, 360
 - local ref \$fromCoerceable, 360
 - defun, 360
- awk
 - calledby whoOwns, 541
- Bang, 466
 - defmacro, 466
- bang
 - calledby PARSE-Category, 423
 - calledby PARSE-CommandTail, 421
 - calledby PARSE-Conditional, 451
 - calledby PARSE-Form, 431
 - calledby PARSE-Import, 425
 - calledby PARSE-IteratorTail, 447
 - calledby PARSE-Seg, 450
 - calledby PARSE-Sexpr1, 443
 - calledby PARSE-SpecialCommand, 419
 - calledby PARSE-TokenCommandTail, 419
- bfp-
 - calledby PARSE-FloatTok, 453
- blankp, 552
 - calledby nonblankloc, 555
 - defun, 552
- Block, 376
 - defplist, 376
- boot-line-stack
 - usedby spad, 575
 - usedby string2BootTree, 71
- bootStrapError, 215
 - calledby compCapsule, 274
 - calledby compFunctorBody, 168
 - calls mkDomainConstructor, 215
 - calls mkq, 215
 - calls namestring, 215
 - defun, 215
- bpiname
 - calledby compileTimeBindingOf, 233
 - calledby subrname, 228
- bright
 - calledby NRTgetLookupFunction, 210
 - calledby checkAndDeclare, 304
 - calledby compDefineLisplib, 163
 - calledby displayMissingFunctions, 216

- calledby doIt, 277
- calledby finalizeDocumentation, 492
- calledby hasSigInTargetCategory, 305
- calledby optimizeFunctionDef, 224
- calledby parseInBy, 122
- calledby postForm, 370
- calledby spadCompileOrSetq, 182
- browserAutoloadOnceTrigger
 - calledby compileSpad2Cmd, 566
- buildFunctor
 - calledby processFunctor, 275
- buildLibAttr, 485
 - calledby buildLibAttrs, 485
 - calls buildLibdbString, 485
 - calls checkCommentsForBraces, 485
 - calls concatWithBlanks, 485
 - calls form2LispString, 485
 - calls lassoc, 485
 - calls length, 485
 - calls stringimage, 485
 - calls sublis, 485
 - calls writedb, 485
 - local ref \$FormalMapVariableList, 485
 - local ref \$conform, 485
 - local ref \$conname, 486
 - local ref \$doc, 485
 - local ref \$exposed?, 485
 - local ref \$kind, 485
 - defun, 485
- buildLibAttrs, 485
 - calledby buildLibdb, 478
 - calls buildLibAttr, 485
 - defun, 485
- buildLibdb, 478
 - calledby extendLocalLibdb, 477
 - calls allConstructors[5], 478
 - calls buildLibAttrs, 478
 - calls buildLibOps, 478
 - calls buildLibdbConEntry, 478
 - calls buildLibdbString, 478
 - calls deleteFile[5], 478
 - calls deleteFile, 478
 - calls dsetq, 478
 - calls getConstructorExports, 478
 - calls ifcar, 478
 - calls make-outstream[5], 478
 - calls obey, 478
 - calls shut, 478
 - calls writedb, 478
 - local def \$AttrLst, 478
 - local def \$DefLst, 478
 - local def \$DomLst, 478
 - local def \$OpLst, 479
 - local def \$PakLst, 478
 - local def \$scatLst, 478
 - local def \$conform, 478
 - local def \$conname, 478
 - local def \$doc, 478
 - local def \$exposed?, 478
 - local def \$kind, 478
 - local def \$outStream, 478
 - local ref \$conform, 478
 - local ref \$outStream, 478
 - defun, 478
- buildLibdbConEntry, 482
 - calledby buildLibdb, 478
 - calls buildLibdbString, 482
 - calls concatWithBlanks, 482
 - calls dbMkForm, 482
 - calls downcase, 482
 - calls form2HtString, 482
 - calls getdatabase, 482
 - calls isExposedConstructor, 482
 - calls lassoc, 482
 - calls length, 482
 - calls libConstructorSig, 482
 - calls libdbTrim, 482
 - calls maxindex, 482
 - calls msubst, 482
 - calls pname, 482
 - calls strconc, 482
 - local def \$conname, 482
 - local def \$doc, 482
 - local def \$exposed?, 482
 - local def \$kind, 482
 - local ref \$conform, 482
 - local ref \$exposed?, 482
 - local ref \$kind, 482
 - defun, 482
- buildLibdbString, 480
 - calledby buildLibAttr, 485
 - calledby buildLibOp, 484

- calledby buildLibdbConEntry, 482
- calledby buildLibdb, 478
- calls strconc, 480
- calls stringimage, 480
- defun, 480
- buildLibOp, 484
 - calledby buildLibOps, 484
 - calls buildLibdbString, 484
 - calls checkCommentsForBraces, 484
 - calls concatWithBlanks, 484
 - calls form2LispString, 484
 - calls lassoc, 484
 - calls libdbTrim, 484
 - calls msubst, 484
 - calls strconc, 484
 - calls stringimage, 484
 - calls sublis, 484
 - calls writedb, 484
 - local ref \$conform, 484
 - local ref \$doc, 484
 - local ref \$exposed?, 484
 - local ref \$kind, 484
 - defun, 484
- buildLibOps, 484
 - calledby buildLibdb, 478
 - calls buildLibOp, 484
 - defun, 484
- bumperrorcount, 545
 - calledby postError, 370
 - uses \$InteractiveMode, 545
 - uses \$spad-errors, 545
 - defun, 545
- call, 229
 - defplist, 229
- canFuncall?
 - calledby loadLibIfNecessary, 115
- cannotDo
 - calledby doIt, 277
- canReturn, 312
 - calledby canReturn, 312
 - calledby compIf, 311
 - calls canReturn, 312
 - calls say, 312
 - calls systemErrorHere, 312
 - defun, 312
- capsule, 273
 - defplist, 273
- CapsuleModemapFrame
 - local ref \$insideCapsuleFunctionIfTrue, 268
- case, 283
 - defplist, 283
- catch, 240
 - defplist, 240
- catches
 - compOrCroak1, 589
 - compUniquely, 616
 - preparse1, 84
 - spad, 574, 575
- category, 102, 286, 377
 - defplist, 102, 286, 377
- char
 - calledby isCategoryPackageName, 210
- char-eq, 464
 - calledby PARSE-FloatBase, 436
 - calledby PARSE-TokTail, 430
 - defun, 464
- char-ne, 464
 - calledby PARSE-FloatBase, 436
 - calledby PARSE-Selector, 433
 - defun, 464
- charp
 - calledby checkSplitBrace, 525
 - calledby checkSplitOn, 536
 - calledby checkSplitPunctuation, 537
- charPosition
 - calledby checkAddSpaceSegments, 529
 - calledby checkExtract, 520
 - calledby checkGetArgs, 521
 - calledby checkGetLispFunctionName, 508
 - calledby checkSplitBackslash, 535
 - calledby checkSplitOn, 536
 - calledby checkSplitPunctuation, 537
 - calledby checkTrim, 516
 - calledby htcharPosition, 539
 - calledby removeBackslashes, 541
 - calledby screenLocalLine, 486
- chaseInferences
 - calledby compHas, 309
- checkAddBackSlashes, 527
 - calledby checkAddBackSlashes, 527

- calledby checkDecorate, 504
 - calls checkAddBackSlashes, 527
 - calls maxindex, 527
 - calls strconc, 527
 - local ref \$charBack, 528
 - local ref \$charEscapeList, 528
 - defun, 527
- checkAddIndented, 519
 - calledby checkRewrite, 499
 - calls checkAddSpaceSegments, 519
 - calls firstNonBlankPosition, 519
 - calls strconc, 519
 - calls stringimage, 519
 - defun, 519
- checkAddMacros, 528
 - calledby checkRewrite, 499
 - calls lassoc, 528
 - calls nreverse, 528
 - local ref \$HTmacs, 528
 - defun, 528
- checkAddPeriod, 529
 - calledby checkComments, 498
 - calls maxindex, 529
 - calls setelt, 529
 - defun, 529
- checkAddSpaces, 530
 - calledby checkComments, 498
 - calledby checkRewrite, 499
 - local ref \$charBlank, 530
 - local ref \$charFauxNewline, 530
 - defun, 530
- checkAddSpaceSegments, 529
 - calledby checkAddIndented, 519
 - calledby checkAddSpaceSegments, 529
 - calledby checkIndentedLines, 524
 - calls charPosition, 529
 - calls checkAddSpaceSegments, 529
 - calls maxindex, 529
 - calls strconc, 529
 - local ref \$charBlank, 530
 - defun, 529
- checkAlphabetic, 531
 - calledby checkSkipIdentifierToken, 525
 - calledby checkSkipOpToken, 525
 - local ref \$charIdentifierEndings, 531
 - defun, 531
- checkAndDeclare, 304
 - calledby compDefineCapsuleFunction, 151
 - calls bright, 304
 - calls getArgumentMode, 304
 - calls modeEqual, 304
 - calls put, 304
 - calls sayBrightly, 304
 - defun, 304
- checkArguments, 501
 - calledby checkComments, 498
 - calledby checkRewrite, 499
 - calls checkHTargs, 501
 - calls hget, 501
 - local ref \$htMacroTable, 501
 - defun, 501
- checkBalance, 501
 - calledby checkComments, 498
 - calls assoc, 501
 - calls checkBeginEnd, 501
 - calls checkDocError, 501
 - calls checkSayBracket, 501
 - calls nreverse, 501
 - calls rassoc, 501
 - local ref \$checkPrenAlist, 501
 - defun, 501
- checkBeginEnd, 502
 - calledby checkBalance, 501
 - calls checkDocError, 502
 - calls hget, 502
 - calls ifcar, 502
 - calls ifcdr, 502
 - calls length, 502
 - calls member, 502
 - calls substring?, 502
 - local ref \$beginEndList, 502
 - local ref \$charBack, 502
 - local ref \$charLbrace, 502
 - local ref \$charRbrace, 502
 - local ref \$htMacroTable, 502
 - defun, 502
- checkComments, 498
 - calledby transformAndRecheckComments, 497
 - calls checkAddPeriod, 498
 - calls checkAddSpaces, 498
 - calls checkArguments, 498

- calls checkBalance, 498
- calls checkDecorate, 498
- calls checkFixCommonProblems, 498
- calls checkGetArgs, 498
- calls checkGetMargin, 498
- calls checkIeEg, 498
- calls checkIndentedLines, 498
- calls checkSplit2Words, 498
- calls checkTransformFirsts, 498
- calls newString2Words, 498
- calls pp, 498
- calls strconc, 498
- local def \$argl, 498
- local def \$checkErrorFlag, 498
- local ref \$attribute?, 498
- local ref \$checkErrorFlag, 498
- defun, 498
- checkCommentsForBraces
 - calledby buildLibAttr, 485
 - calledby buildLibOp, 484
- checkDecorate, 504
 - calledby checkComments, 498
 - calls checkAddBackSlashes, 504
 - calls checkDocError, 504
 - calls hasNoVowels, 504
 - calls member, 504
 - local ref \$argl, 504
 - local ref \$charBack, 504
 - local ref \$charExclusions, 504
 - local ref \$charLbrace, 504
 - local ref \$charRbrace, 504
 - local ref \$checkingXmptex?, 504
 - defun, 504
- checkDecorateForHt, 506
 - calledby checkRewrite, 499
 - calls checkDocError, 506
 - calls member, 506
 - local ref \$charLbrace, 506
 - local ref \$charRbrace, 506
 - local ref \$checkingXmptex?, 506
 - defun, 506
- checkDocError, 517
 - calledby checkBalance, 501
 - calledby checkBeginEnd, 502
 - calledby checkDecorateForHt, 506
 - calledby checkDecorate, 504
 - calledby checkDocError1, 507
 - calledby checkFixCommonProblem, 508
 - calledby checkGetLispFunctionName, 508
 - calledby checkHTargs, 509
 - calledby checkRecordHash, 509
 - calledby checkTexht, 512
 - calledby checkTransformFirsts, 513
 - calledby checkTrim, 516
 - calledby transDocList, 495
 - calls checkDocMessage, 517
 - calls concat, 517
 - calls sayBrightly, 517
 - calls saybrightly1, 517
 - local def \$checkErrorFlag, 517
 - local def \$exposeFlagHeading, 517
 - local ref \$checkErrorFlag, 517
 - local ref \$constructorName, 517
 - local ref \$exposeFlagHeading, 517
 - local ref \$exposeFlag, 517
 - local ref \$outStream, 517
 - local ref \$recheckingFlag, 517
 - defun, 517
- checkDocError1, 507
 - calledby transDocList, 495
 - calledby transDoc, 496
 - calls checkDocError, 507
 - local ref \$compileDocumentation, 507
 - defun, 507
- checkDocMessage, 519
 - calledby checkDocError, 517
 - calls concat, 520
 - calls getdatabase, 519
 - calls whoOwns, 520
 - local ref \$constructorName, 520
 - local ref \$x, 520
 - defun, 519
- checkExtract, 520
 - calledby transDoc, 496
 - calls charPosition, 520
 - calls firstNonBlankPosition, 520
 - calls length, 520
 - calls substring?, 520
 - defun, 520
- checkFixCommonProblem, 508
 - calledby checkRewrite, 499
 - calls checkDocError, 508

- calls ifcar, 508
- calls ifcdr, 508
- calls member, 508
- local ref \$HTspadmacros, 508
- local ref \$charLbrace, 508
- defun, 508
- checkFixCommonProblems
 - calledby checkComments, 498
- checkGetArgs, 521
 - calledby checkComments, 498
 - calledby checkGetArgs, 521
 - calledby checkRewrite, 499
 - calls charPosition, 521
 - calls checkGetArgs, 521
 - calls firstNonBlankPosition, 521
 - calls getMatchingRightPren, 521
 - calls maxindex, 521
 - calls stringPrefix?, 521
 - calls trimString, 521
 - local ref \$charComma, 521
 - defun, 521
- checkGetLispFunctionName, 508
 - calledby checkRecordHash, 509
 - calls charPosition, 508
 - calls checkDocError, 508
 - defun, 508
- checkGetMargin, 522
 - calledby checkComments, 498
 - calls firstNonBlankPosition, 522
 - defun, 522
- checkGetParse, 522
 - calledby checkRecordHash, 509
 - calls ncParseFromString, 522
 - calls removeBackslashes, 522
 - defun, 522
- checkGetStringBeforeRightBrace, 523
 - calledby checkRecordHash, 509
 - local ref \$charRbrace, 523
 - defun, 523
- checkHTargs, 509
 - calledby checkArguments, 501
 - calledby checkHTargs, 509
 - calls checkDocError, 509
 - calls checkHTargs, 509
 - calls checkLookForLeftBrace, 509
 - calls checkLookForRightBrace, 509
- calls ifcdr, 509
- defun, 509
- checkIeEg, 523
 - calledby checkComments, 498
 - calls checkIeEgfun, 523
 - calls nreverse, 523
 - defun, 523
- checkIeEgFun
 - calledby checkIeEgfun, 531
- checkIeEgfun, 531
 - calledby checkIeEg, 523
 - calls checkIeEgFun, 531
 - calls maxindex, 531
 - local ref \$charPeriod, 531
 - defun, 531
- checkIndentedLines, 524
 - calledby checkComments, 498
 - calls checkAddSpaceSegments, 524
 - calls firstNonBlankPosition, 524
 - calls strconc, 524
 - local ref \$charFauxNewline, 524
 - defun, 524
- checkIsValidType, 532
 - calledby checkIsValidType, 532
 - calledby checkRecordHash, 510
 - calls abbreviation?, 532
 - calls checkIsValidType, 532
 - calls constructor?, 532
 - calls getdatabase, 532
 - calls length, 532
 - defun, 532
- checkLookForLeftBrace, 533
 - calledby checkHTargs, 509
 - calledby checkRecordHash, 509
 - local ref \$charBlank, 533
 - local ref \$charLbrace, 533
 - defun, 533
- checkLookForRightBrace, 533
 - calledby checkHTargs, 509
 - calledby checkRecordHash, 509
 - local ref \$charLbrace, 533
 - local ref \$charRbrace, 533
 - defun, 533
- checkNumOfArgs, 534
 - calledby checkRecordHash, 510
 - calls abbreviation?, 534

- calls constructor?, 534
- calls getdatabase, 534
- calls opOf, 534
- defun, 534
- checkRecordHash, 509
 - calledby checkRewrite, 499
 - calls checkDocError, 509
 - calls checkGetLispFunctionName, 509
 - calls checkGetParse, 509
 - calls checkGetStringBeforeRightBrace, 509
 - calls checkIsValidType, 510
 - calls checkLookForLeftBrace, 509
 - calls checkLookForRightBrace, 509
 - calls checkNumOfArgs, 510
 - calls form2HtString, 510
 - calls getl, 510
 - calls hget, 509
 - calls hput, 509
 - calls ifcdr, 509
 - calls intern, 509
 - calls member, 509
 - calls opOf, 509
 - calls spadSysChoose, 509
 - local def \$glossHash, 510
 - local def \$htHash, 510
 - local def \$lispHash, 510
 - local def \$sysHash, 510
 - local ref \$HTlinks, 510
 - local ref \$HTlisplinks, 510
 - local ref \$charBack, 510
 - local ref \$currentSysList, 510
 - local ref \$glossHash, 510
 - local ref \$htHash, 510
 - local ref \$lispHash, 510
 - local ref \$name, 510
 - local ref \$origin, 510
 - local ref \$setOptions, 510
 - local ref \$sysHash, 510
 - defun, 509
- checkRemoveComments, 518
 - calledby checkRewrite, 499
 - calls checkTrimCommented, 518
 - defun, 518
- checkRewrite, 499
 - calledby transformAndRecheckComments, 497
- calls checkAddIndented, 499
- calls checkAddMacros, 499
- calls checkAddSpaces, 499
- calls checkArguments, 499
- calls checkDecorateForHt, 499
- calls checkFixCommonProblem, 499
- calls checkGetArgs, 499
- calls checkRecordHash, 499
- calls checkRemoveComments, 499
- calls checkSplit2Words, 499
- calls checkTexht, 499
- calls newString2Words, 499
- local ref \$argl, 499
- local ref \$checkingErrorFlag, 499
- local ref \$checkingXmptex?, 500
- defun, 499
- checkSayBracket, 534
 - calledby checkBalance, 501
 - defun, 534
- checkSkipBlanks, 534
 - calledby checkTransformFirsts, 513
 - local ref \$charBlank, 534
 - defun, 534
- checkSkipIdentifierToken, 525
 - calledby checkSkipToken, 518
 - calls checkAlphabetic, 525
 - defun, 525
- checkSkipOpToken, 525
 - calledby checkSkipToken, 518
 - calls checkAlphabetic, 525
 - calls member, 525
 - local ref \$charDelimiters, 525
 - defun, 525
- checkSkipToken, 518
 - calledby checkTransformFirsts, 513
 - calls checkSkipIdentifierToken, 518
 - calls checkSkipOpToken, 518
 - defun, 518
- checkSplit2Words, 518
 - calledby checkComments, 498
 - calledby checkRewrite, 499
 - calls checkSplitBrace, 518
 - defun, 518
- checkSplitBackslash, 535
 - calledby checkSplitBackslash, 535
 - calledby checkSplitBrace, 526

- calls charPosition, 535
 - calls checkSplitBackslash, 535
 - calls maxindex, 535
 - local ref \$charBack, 535
 - defun, 535
- checkSplitBrace, 525
 - calledby checkSplit2Words, 518
 - calledby checkSplitBrace, 526
 - calls charp, 525
 - calls checkSplitBackslash, 526
 - calls checkSplitBrace, 526
 - calls checkSplitOn, 526
 - calls checkSplitPunctuation, 526
 - calls length, 526
 - defun, 525
- checkSplitOn, 536
 - calledby checkSplitBrace, 526
 - calledby checkSplitOn, 536
 - calls charPosition, 536
 - calls charp, 536
 - calls checkSplitOn, 536
 - calls maxindex, 536
 - local ref \$charBack, 536
 - local ref \$charSplitList, 536
 - defun, 536
- checkSplitPunctuation, 537
 - calledby checkSplitBrace, 526
 - calledby checkSplitPunctuation, 537
 - calls charPosition, 537
 - calls charp, 537
 - calls checkSplitPunctuation, 537
 - calls hget, 537
 - calls maxindex, 537
 - local ref \$charBack, 537
 - local ref \$charComma, 537
 - local ref \$charDash, 537
 - local ref \$charPeriod, 537
 - local ref \$charQuote, 537
 - local ref \$charSemiColon, 537
 - local ref \$htMacroTable, 537
 - defun, 537
- checkTexht, 512
 - calledby checkRewrite, 499
 - calls checkDocError, 512
 - calls ifcar, 512
 - calls ifcdr, 512
 - local ref \$charLbrace, 512
 - local ref \$charRbrace, 512
 - defun, 512
- checkTransformFirsts, 513
 - calledby checkComments, 498
 - calledby checkTransformFirsts, 513
 - calls checkDocError, 513
 - calls checkSkipBlanks, 513
 - calls checkSkipToken, 513
 - calls checkTransformFirsts, 513
 - calls fillerSpaces, 513
 - calls getMatchingRightPren, 513
 - calls getl, 513
 - calls lassoc, 513
 - calls leftTrim, 513
 - calls maxindex, 513
 - calls pname, 513
 - calls strconc, 513
 - local ref \$charBack, 513
 - local ref \$checkPrenAlist, 513
 - defun, 513
- checkTrim, 516
 - calledby transDoc, 496
 - calls charPosition, 516
 - calls checkDocError, 516
 - calls systemError, 516
 - local ref \$charBlank, 516
 - local ref \$charPlus, 516
 - local ref \$x, 516
 - defun, 516
- checkTrimCommented, 526
 - calledby checkRemoveComments, 518
 - calls htcharPosition, 526
 - calls length, 526
 - defun, 526
- checkWarning, 549
 - calledby postCapsule, 373
 - calls concat, 549
 - calls postError, 549
 - defun, 549
- clearClams
 - calledby compileConstructor, 183
- clearConstructorCache
 - calledby compileConstructor1, 184
- coerce, 351
 - calledby coerceExit, 357

- calledby coerceExtraHard, 355
- calledby coerceable, 356
- calledby compApplication, 605
- calledby compApplyModemap, 255
- calledby compAtSign, 357
- calledby compCase, 283
- calledby compCoerce1, 359
- calledby compCoerce, 358
- calledby compColonInside, 596
- calledby compForm1, 603
- calledby compHas, 309
- calledby compIf, 311
- calledby compIs, 318
- calledby convert, 600
- calls coerceEasy, 351
- calls coerceHard, 352
- calls coerceSubset, 352
- calls isSomeDomainVariable, 352
- calls keyedSystemError, 351
- calls rplac, 351
- calls stackMessage, 352
- local ref \$InteractiveMode, 352
- local ref \$Rep, 352
- local ref \$fromCoerceable, 352
- defun, 351
- coerceable, 356
 - calledby compFocompFormWithModemap, 613
 - calledby compForm1, 603
 - calls coerce, 356
 - calls pmatch, 356
 - calls sublis, 356
 - local ref \$fromCoerceable, 356
 - defun, 356
- coerceByModemap, 359
 - calledby compCoerce1, 359
 - calls genDeltaEntry, 359
 - calls isSubset, 359
 - calls modeEqual, 359
 - defun, 359
- coerceEasy, 352
 - calledby coerce, 351
 - calls modeEqualSubst, 352
 - local ref \$EmptyMode, 352
 - local ref \$Exit, 352
 - local ref \$NoValueMode, 352
- local ref \$Void, 352
- defun, 352
- coerceExit, 357
 - calledby compRepeatOrCollect, 329
 - calls coerce, 357
 - calls replaceExitEsc, 357
 - calls resolve, 357
 - local ref \$exitMode, 357
 - defun, 357
- coerceExtraHard, 355
 - calledby coerceHard, 354
 - calls autoCoerceByModemap, 355
 - calls coerce, 355
 - calls hasType, 355
 - calls isUnionMode, 355
 - calls member, 355
 - local ref \$Expression, 355
 - defun, 355
- coerceHard, 354
 - calledby coerce, 352
 - calls coerceExtraHard, 354
 - calls extendsCategoryForm, 354
 - calls getmode, 354
 - calls get, 354
 - calls isCategoryForm, 354
 - calls modeEqual, 354
 - local def \$e, 354
 - local ref \$String, 354
 - local ref \$bootStrapMode, 354
 - local ref \$e, 354
 - defun, 354
- coerceSubset, 353
 - calledby coerce, 352
 - calls eval, 353
 - calls get, 353
 - calls isSubset, 353
 - calls lassoc, 353
 - calls maxSuperType, 353
 - calls opOf, 353
 - defun, 353
- collect, 328, 379
 - calledby floatexpid, 465
 - defplist, 328, 379
- collectAndDeleteAssoc, 472
 - calledby collectComBlock, 471
 - local ref \$comblocklist, 472

- defun, [472](#)
- collectComBlock, [471](#)
 - calledby recordDocumentation, [471](#)
 - calls collectAndDeleteAssoc, [471](#)
 - local def \$comblocklist, [471](#)
 - defun, [471](#)
- comma2Tuple, [382](#)
 - calledby postComma, [382](#)
 - calledby postConstruct, [383](#)
 - calls postFlatten, [382](#)
 - defun, [382](#)
- comp, [590](#)
 - calledby applyMapping, [593](#)
 - calledby compAdd, [271](#)
 - calledby compApplication, [606](#)
 - calledby compApplyModemap, [255](#)
 - calledby compApply, [595](#)
 - calledby compArgumentsAndTryAgain, [616](#)
 - calledby compAtSign, [357](#)
 - calledby compBoolean, [314](#)
 - calledby compCase1, [284](#)
 - calledby compCoerce1, [359](#)
 - calledby compColonInside, [596](#)
 - calledby compCons1, [294](#)
 - calledby compDefWhereClause, [156](#)
 - calledby compDefineAddSignature, [143](#)
 - calledby compExit, [308](#)
 - calledby compExpressionList, [609](#)
 - calledby compForMode, [321](#)
 - calledby compForm1, [603](#)
 - calledby compFromIf, [312](#)
 - calledby compHasFormat, [309](#)
 - calledby compIs, [318](#)
 - calledby compLeave, [323](#)
 - calledby compList, [602](#)
 - calledby compOrCroak1, [589](#)
 - calledby compPretend, [325](#)
 - calledby compReduce1, [326](#)
 - calledby compRepeatOrCollect, [329](#)
 - calledby compReturn, [331](#)
 - calledby compSeqItem, [334](#)
 - calledby compSubsetCategory, [348](#)
 - calledby compSuchthat, [349](#)
 - calledby compUniquely, [616](#)
 - calledby compVector, [350](#)
 - calledby compWhere, [350](#)
 - calledby compWithMappingMode1, [617](#)
 - calledby compileConstructor1, [184](#)
 - calledby doItIf, [281](#)
 - calledby getSuccessEnvironment, [315](#)
 - calledby outputComp, [343](#)
 - calledby setqSetelt, [340](#)
 - calledby setqSingle, [340](#)
 - calledby spadCompileOrSetq, [182](#)
 - calls compNoStacking, [590](#)
 - local ref \$compStack, [590](#)
 - uses \$exitModeStack, [590](#)
 - defun, [590](#)
- comp-tran
 - calledby compWithMappingMode1, [618](#)
- comp2, [591](#)
 - calledby compNoStacking1, [591](#)
 - calledby compNoStacking, [590](#)
 - calls addDomain, [591](#)
 - calls comp3, [591](#)
 - calls insert, [591](#)
 - calls isDomainForm, [591](#)
 - calls isFunctor, [591](#)
 - calls opOf, [591](#)
 - uses \$bootStrapMode, [591](#)
 - uses \$lisplib, [591](#)
 - uses \$packagesUsed, [591](#)
 - defun, [591](#)
- comp3, [592](#)
 - calledby comp2, [591](#)
 - calledby compTypeOf, [596](#)
 - calls addDomain, [592](#)
 - calls applyMapping, [592](#)
 - calls compApply, [592](#)
 - calls compAtom, [592](#)
 - calls compCoerce, [592](#)
 - calls compColon, [592](#)
 - calls compExpression, [592](#)
 - calls compTypeOf, [592](#)
 - calls compWithMappingMode, [592](#)
 - calls getDomainsInScope, [592](#)
 - calls getmode, [592](#)
 - calls member[5], [592](#)
 - calls pname[5], [592](#)
 - calls stringPrefix?, [592](#)
 - uses \$e, [592](#)
 - uses \$insideCompTypeOf, [592](#)

- defun, 592
- compAdd, 271
 - calls NRTgetLocalIndex, 271
 - calls compCapsule, 271
 - calls compOrCroak, 271
 - calls compSubDomain1, 271
 - calls compTuple2Record, 271
 - calls comp, 271
 - calls nreverse0, 271
 - uses /editfile, 271
 - uses \$EmptyMode, 271
 - uses \$NRTaddForm, 271
 - uses \$addFormLhs, 271
 - uses \$addForm, 271
 - uses \$bootstrapMode, 272
 - uses \$functorForm, 272
 - uses \$packagesUsed, 271
 - defun, 271
- compAndDefine, 185
 - calledby compileConstructor1, 184
 - defun, 185
- compApplication, 605
 - calledby compToApply, 605
 - calls coerce, 605
 - calls comp, 606
 - calls eltForm, 605
 - calls encodeItem, 605
 - calls getAbbreviation, 605
 - calls isCategoryForm, 606
 - calls length, 606
 - calls member, 606
 - calls resolve, 605
 - calls strconc, 605
 - local ref \$Category, 606
 - local ref \$formatArgList, 606
 - local ref \$form, 606
 - local ref \$op, 606
 - local ref \$prefix, 606
 - defun, 605
- compApply, 595
 - calledby comp3, 592
 - calls AddContour, 595
 - calls Pair, 595
 - calls comp, 595
 - calls removeEnv, 595
 - calls resolve, 595
 - local ref \$EmptyMode, 595
 - defun, 595
- compApplyModemap, 255
 - calledby compFocompFormWithModemap, 613
 - calledby getModemap, 254
 - calls coerce, 255
 - calls compMapCond, 255
 - calls comp, 255
 - calls genDeltaEntry, 255
 - calls length, 255
 - calls member, 255
 - calls pmatchWithSl, 255
 - calls sublis, 255
 - local def \$bindings, 255
 - local def \$e, 255
 - local ref \$bindings, 255
 - local ref \$e, 255
 - defun, 255
- compareMode2Arg
 - calledby hasSigInTargetCategory, 305
- compArgumentConditions, 166
 - calledby compDefineCapsuleFunction, 151
 - calls compOrCroak, 166
 - local def \$argumentConditionList, 166
 - local ref \$Boolean, 166
 - local ref \$argumentConditionList, 166
 - defun, 166
- compArgumentsAndTryAgain, 616
 - calledby compForm, 602
 - calls compForm1, 616
 - calls comp, 616
 - uses \$EmptyMode, 616
 - defun, 616
- compAtom, 597
 - calledby comp3, 592
 - calls compAtomWithModemap, 597
 - calls compList, 597
 - calls compSymbol, 597
 - calls compVector, 597
 - calls convert, 597
 - calls get, 597
 - calls isSymbol, 597
 - calls modeIsAggregateOf, 597
 - calls primitiveType, 597
 - uses \$Expression, 597

- defun, 597
- compAtomWithModemap, 598
 - calledby compAtom, 597
 - calls convert, 598
 - calls modeEqual, 598
 - calls transImplementation, 598
 - local ref \$NoValueMode, 598
 - defun, 598
- compAtSign, 357
 - calledby compLambda, 321
 - calls addDomain, 357
 - calls coerce, 357
 - calls comp, 357
 - defun, 357
- compBoolean, 314
 - calledby compIf, 311
 - calls comp, 314
 - calls getInverseEnvironment, 314
 - calls getSuccessEnvironment, 314
 - defun, 314
- compCapsule, 274
 - calledby compAdd, 271
 - calledby compSubDomain, 346
 - calls addDomain, 274
 - calls bootStrapError, 274
 - calls compCapsuleInner, 274
 - uses \$bootStrapMode, 274
 - uses \$functorForm, 274
 - uses \$insideExpressionIfTrue, 274
 - uses editfile, 274
 - defun, 274
- compCapsuleInner, 274
 - calledby compCapsule, 274
 - calls addInformation, 274
 - calls compCapsuleItems, 274
 - calls mkpf, 274
 - calls processFunctor, 274
 - uses \$addForm, 274
 - uses \$form, 274
 - uses \$functorLocalParameters, 274
 - uses \$getDomainCode, 274
 - uses \$insideCategoryIfTrue, 274
 - uses \$insideCategoryPackageIfTrue, 274
 - uses \$signature, 274
 - defun, 274
- compCapsuleItems, 275
 - calledby compCapsuleInner, 274
 - calls compSingleCapsuleItem, 275
 - local def \$e, 276
 - local def \$myFunctorBody, 275
 - local def \$signatureOfForm, 276
 - local def \$suffix, 276
 - local def \$top-level, 275
 - local ref \$e, 276
 - local ref \$pred, 276
 - defun, 275
- compCase, 283
 - calls addDomain, 283
 - calls coerce, 283
 - calls compCase1, 283
 - defun, 283
- compCase1, 284
 - calledby compCase, 283
 - calls comp, 284
 - calls getModemapList, 284
 - calls modeEqual, 284
 - calls nreverse0, 284
 - uses \$Boolean, 284
 - uses \$EmptyMode, 284
 - defun, 284
- compCat, 285
 - calls get1, 285
 - defun, 285
- compCategory, 286
 - calls compCategoryItem, 286
 - calls mkExplicitCategoryFunction, 286
 - calls resolve, 286
 - calls systemErrorHere, 286
 - local def \$atList, 286
 - local def \$sigList, 286
 - local def \$top-level, 286
 - local ref \$atList, 286
 - local ref \$sigList, 286
 - defun, 286
- compCategoryItem, 287
 - calledby compCategoryItem, 287
 - calledby compCategory, 286
 - calls compCategoryItem, 287
 - calls mkpf, 287
 - local ref \$atList, 287
 - local ref \$sigList, 287
 - defun, 287

- compCoerce, 358
 - calledby comp3, 592
 - calls addDomain, 358
 - calls coerce, 358
 - calls compCoerce1, 358
 - calls getmode, 358
 - defun, 358
- compCoerce1, 359
 - calledby compCoerce, 358
 - calls coerceByModemap, 359
 - calls coerce, 359
 - calls comp, 359
 - calls mkq, 359
 - calls resolve, 359
 - defun, 359
- compColon, 290
 - calledby comp3, 592
 - calledby compColon, 290
 - calledby compMakeDeclaration, 624
 - calls addDomain, 290
 - calls assoc, 290
 - calls compColonInside, 290
 - calls compColon, 290
 - calls eqsubstlist, 290
 - calls genSomeVariable, 291
 - calls getDomainsInScope, 290
 - calls getmode, 290
 - calls isCategoryForm, 290
 - calls isDomainForm, 290
 - calls length, 290
 - calls makeCategoryForm, 290
 - calls member[5], 290
 - calls nreverse0, 290
 - calls put, 290
 - calls systemErrorHere, 290
 - calls take, 290
 - calls unknownTypeError, 290
 - uses \$FormalMapVariableList, 291
 - uses \$bootStrapMode, 291
 - uses \$insideCategoryIfTrue, 291
 - uses \$insideExpressionIfTrue, 291
 - uses \$insideFunctorIfTrue, 291
 - uses \$lhsOfColon, 291
 - uses \$noEnv, 291
 - defun, 290
- compColonInside, 596
 - calledby compColon, 290
 - calls addDomain, 596
 - calls coerce, 596
 - calls comp, 596
 - calls opOf, 596
 - calls stackSemanticError, 596
 - calls stackWarning, 596
 - uses \$EmptyMode, 596
 - uses \$newCompilerUnionFlag, 596
 - defun, 596
- compCons, 294
 - calls compCons1, 294
 - calls compForm, 294
 - defun, 294
- compCons1, 294
 - calledby compCons, 294
 - calls comp, 294
 - calls convert, 294
 - uses \$EmptyMode, 294
 - defun, 294
- compConstruct, 295
 - calls compForm, 295
 - calls compList, 295
 - calls compVector, 295
 - calls convert, 295
 - calls getDomainsInScope, 295
 - calls modeIsAggregateOf, 295
 - defun, 295
- compConstructorCategory, 297
 - calls resolve, 297
 - uses \$Category, 297
 - defun, 297
- compDefine, 140
 - calls compDefine1, 140
 - local def \$macroIfTrue, 140
 - local def \$packagesUsed, 141
 - local def \$tripleCache, 140
 - local def \$tripleHits, 140
 - defun, 140
- compDefine1, 141
 - calledby compDefine1, 141
 - calledby compDefineCategory1, 158
 - calledby compDefine, 140
 - calls addEmptyCapsuleIfNecessary, 141
 - calls compDefWhereClause, 141
 - calls compDefine1, 141

- calls compDefineAddSignature, [141](#)
- calls compDefineCapsuleFunction, [141](#)
- calls compDefineCategory, [141](#)
- calls compDefineFunctor, [141](#)
- calls compInternalFunction, [141](#)
- calls getAbbreviation, [141](#)
- calls getSignatureFromMode, [141](#)
- calls getTargetFromRhs, [141](#)
- calls giveFormalParametersValues, [141](#)
- calls isDomainForm, [141](#)
- calls isMacro, [141](#)
- calls length, [141](#)
- calls macroExpand, [141](#)
- calls stackAndThrow, [141](#)
- calls strconc, [141](#)
- uses \$Category, [141](#)
- uses \$ConstructorNames, [142](#)
- uses \$EmptyMode, [142](#)
- uses \$NoValueMode, [142](#)
- uses \$formalArgList, [141](#)
- uses \$form, [141](#)
- uses \$insideCapsuleFunctionIfTrue, [141](#)
- uses \$insideCategoryIfTrue, [141](#)
- uses \$insideExpressionIfTrue, [141](#), [142](#)
- uses \$insideFunctorIfTrue, [141](#)
- uses \$insideWhereIfTrue, [142](#)
- uses \$op, [141](#)
- uses \$prefix, [141](#)
- defun, [141](#)
- compDefineAddSignature, [143](#)
 - calledby compDefine1, [141](#)
 - calls assoc, [143](#)
 - calls comp, [143](#)
 - calls getPropList, [143](#)
 - calls hasFullSignature, [143](#)
 - calls lassoc, [143](#)
 - uses \$EmptyMode, [143](#)
 - defun, [143](#)
- compDefineCapsuleFunction, [151](#)
 - calledby compDefine1, [141](#)
 - calls NRTassignCapsuleFunctionSlot, [152](#)
 - calls addArgumentConditions, [152](#)
 - calls addDomain, [151](#)
 - calls addStats, [152](#)
 - calls checkAndDeclare, [151](#)
 - calls compArgumentConditions, [151](#)
 - calls compOrCroak, [152](#)
 - calls compileCases, [152](#)
 - calls formatUnabbreviated, [152](#)
 - calls getArgumentModeOrMoan, [151](#)
 - calls getSignature, [151](#)
 - calls getmode, [152](#)
 - calls get, [151](#)
 - calls giveFormalParametersValues, [151](#)
 - calls hasSigInTargetCategory, [151](#)
 - calls length, [151](#)
 - calls member, [152](#)
 - calls mkq, [152](#)
 - calls profileRecord, [151](#)
 - calls put, [151](#)
 - calls replaceExitEtc, [152](#)
 - calls resolve, [152](#)
 - calls sayBrightly, [152](#)
 - calls stripOffArgumentConditions, [151](#)
 - calls stripOffSubdomainConditions, [151](#)
 - local def \$CapsuleDomainsInScope, [152](#)
 - local def \$CapsuleModemapFrame, [152](#)
 - local def \$argumentConditionList, [152](#)
 - local def \$finalEnv, [152](#)
 - local def \$formalArgList, [152](#)
 - local def \$form, [152](#)
 - local def \$functionLocations, [152](#)
 - local def \$functionStats, [152](#)
 - local def \$initCapsuleErrorCount, [152](#)
 - local def \$insideCapsuleFunctionIfTrue, [152](#)
 - local def \$insideExpressionIfTrue, [152](#)
 - local def \$op, [152](#)
 - local def \$returnMode, [152](#)
 - local def \$signatureOffForm, [152](#)
 - local ref \$DomainsInScope, [152](#)
 - local ref \$compileOnlyCertainItems, [152](#)
 - local ref \$formalArgList, [152](#)
 - local ref \$functionLocations, [152](#)
 - local ref \$functionStats, [152](#)
 - local ref \$functorStats, [152](#)
 - local ref \$op, [152](#)
 - local ref \$profileCompiler, [152](#)
 - local ref \$returnMode, [152](#)
 - local ref \$semanticErrorStack, [152](#)
 - local ref \$signatureOffForm, [152](#)
 - defun, [151](#)

- compDefineCategory, 158
 - calledby compDefine1, 141
 - calls compDefineCategory1, 158
 - calls compDefineLisplib, 158
 - uses \$domainShell, 158
 - uses \$insideFunctorIfTrue, 158
 - uses \$lisplibCategory, 158
 - uses \$lisplib, 158
 - defun, 158
- compDefineCategory1, 158
 - calledby compDefineCategory, 158
 - calls compDefine1, 158
 - calls compDefineCategory2, 158
 - calls makeCategoryPredicates, 158
 - calls mkCategoryPackage, 158
 - uses \$EmptyMode, 158
 - uses \$bootStrapMode, 158
 - uses \$categoryPredicateList, 158
 - uses \$insideCategoryPackageIfTrue, 158
 - uses \$lisplibCategory, 158
 - defun, 158
- compDefineCategory2, 159
 - calledby compDefineCategory1, 158
 - calls addBinding, 159
 - calls augLisplibModemapsFromCategory, 160
 - calls compMakeDeclaration, 159
 - calls compOrCroak, 160
 - calls compile, 160
 - calls computeAncestorsOf, 160
 - calls constructor?, 160
 - calls evalAndRwriteLispForm, 160
 - calls eval, 160
 - calls getArgumentModeOrMoan, 159
 - calls getParentsFor, 160
 - calls giveFormalParametersValues, 159
 - calls lisplibWrite, 160
 - calls mkConstructor, 160
 - calls mkq, 160
 - calls opOf, 160
 - calls optFunctorBody, 160
 - calls removeZeroOne, 160
 - calls sublis, 159
 - calls take, 159
 - local def \$addForm, 160
 - local def \$definition, 160
 - local def \$domainShell, 160
 - local def \$extraParms, 160
 - local def \$formalArgList, 160
 - local def \$form, 160
 - local def \$frontier, 160
 - local def \$functionStats, 160
 - local def \$functorForm, 160
 - local def \$functorStats, 160
 - local def \$getDomainCode, 160
 - local def \$insideCategoryIfTrue, 160
 - local def \$lisplibAbbreviation, 160
 - local def \$lisplibAncestors, 160
 - local def \$lisplibCategory, 160
 - local def \$lisplibForm, 160
 - local def \$lisplibKind, 160
 - local def \$lisplibModemap, 160
 - local def \$lisplibParents, 160
 - local def \$op, 160
 - local def \$top-level, 160
 - local ref \$FormalMapVariableList, 160
 - local ref \$TriangleVariableList, 160
 - local ref \$definition, 160
 - local ref \$extraParms, 160
 - local ref \$formalArgList, 160
 - local ref \$form, 160
 - local ref \$libFile, 160
 - local ref \$lisplibCategory, 160
 - local ref \$lisplib, 160
 - local ref \$op, 160
 - uses \$prefix, 160
 - defun, 159
- compDefineFunctor, 144
 - calledby compDefine1, 141
 - calls compDefineFunctor1, 144
 - calls compDefineLisplib, 144
 - uses \$domainShell, 144
 - uses \$lisplib, 144
 - uses \$profileAlist, 144
 - uses \$profileCompiler, 144
 - defun, 144
- compDefineFunctor1, 144
 - calledby compDefineFunctor, 144
 - calls NRTgenInitialAttributeAlist, 144
 - calls NRTgetLocalIndex, 144
 - calls NRTgetLookupFunction, 145
 - calls NRTmakeSlot1Info, 145

- calls `augModemapsFromCategoryRep`, 144
- calls `augModemapsFromCategory`, 144
- calls `augmentLisplibModemapsFromFunc-`
`tor`, 145
- calls `compFuncorBody`, 145
- calls `compMakeCategoryObject`, 144
- calls `compMakeDeclaration`, 144
- calls `compile`, 145
- calls `computeAncestorsOf`, 145
- calls `constructor?`, 145
- calls `disallowNilAttribute`, 144
- calls `evalAndRwriteLispForm`, 145
- calls `getArgumentModeOrMoan`, 144
- calls `getModemap`, 144
- calls `getParentsFor`, 145
- calls `getdatabase`, 145
- calls `giveFormalParametersValues`, 144
- calls `isCategoryPackageName`, 144, 145
- calls `lisplibWrite`, 145
- calls `makeFuncorArgumentParameters`,
144
- calls `maxindex`, 144
- calls `mkq`, 145
- calls `pname`, 144
- calls `pp`, 144
- calls `remdup`, 144
- calls `removeZeroOne`, 145
- calls `reportOnFuncorCompilation`, 145
- calls `sayBrightly`, 144
- calls `simpBool`, 145
- calls `strconc`, 144
- calls `sublis`, 144
- uses `$CategoryFrame`, 145
- uses `$CheckVectorList`, 145
- uses `$FormalMapVariableList`, 145
- uses `$LocalDomainAlist`, 145
- uses `$NRTaddForm`, 145
- uses `$NRTaddList`, 145
- uses `$NRTattributeAlist`, 145
- uses `$NRTbase`, 145
- uses `$NRTdeltaLength`, 145
- uses `$NRTdeltaListComp`, 145
- uses `$NRTdeltaList`, 145
- uses `$NRTdomainFormList`, 145
- uses `$NRTloadTimeAlist`, 145
- uses `$NRTslot1Info`, 145
- uses `$NRTslot1PredicateList`, 145
- uses `$QuickCode`, 146
- uses `$Representation`, 145
- uses `$addForm`, 145
- uses `$attributesName`, 145
- uses `$bootStrapMode`, 145
- uses `$byteAddress`, 145
- uses `$byteVec`, 145
- uses `$compileOnlyCertainItems`, 145
- uses `$condAlist`, 145
- uses `$domainShell`, 145
- uses `$form`, 145
- uses `$functionLocations`, 145
- uses `$functionStats`, 145
- uses `$funcorForm`, 146
- uses `$funcorLocalParameters`, 146
- uses `$funcorSpecialCases`, 146
- uses `$funcorStats`, 146
- uses `$funcorTarget`, 146
- uses `$funcorsUsed`, 146
- uses `$genFVar`, 146
- uses `$genSDVar`, 146
- uses `$getDomainCode`, 146
- uses `$goGetList`, 146
- uses `$insideCategoryPackageIfTrue`, 146
- uses `$insideFuncorIfTrue`, 146
- uses `$isOpPackageName`, 146
- uses `$libFile`, 146
- uses `$lisplibAbbreviation`, 146
- uses `$lisplibAncestors`, 146
- uses `$lisplibCategoriesExtended`, 146
- uses `$lisplibCategory`, 146
- uses `$lisplibForm`, 146
- uses `$lisplibFunctionLocations`, 146
- uses `$lisplibKind`, 146
- uses `$lisplibMissingFunctions`, 146
- uses `$lisplibModemap`, 146
- uses `$lisplibOperationAlist`, 146
- uses `$lisplibParents`, 146
- uses `$lisplibSlot1`, 146
- uses `$lisplib`, 145
- uses `$lookupFunction`, 146
- uses `$mutableDomains`, 146
- uses `$mutableDomain`, 146
- uses `$myFuncorBody`, 146
- uses `$op`, 146

- uses \$pairlis, 146
- uses \$setelt, 146
- uses \$signature, 146
- uses \$template, 146
- uses \$top-level, 145
- uses \$uncondAlist, 146
- uses \$viewNames, 146
- defun, 144
- compDefineLisplib, 163
 - calledby compDefineCategory, 158
 - calledby compDefineFunctor, 144
 - calls bright, 163
 - calls compileDocumentation, 163
 - calls filep, 163
 - calls fillerSpaces, 163
 - calls finalizeLisplib, 163
 - calls getConstructorAbbreviation, 163
 - calls getdatabase, 163
 - calls lisplibDoRename, 163
 - calls localdatabase, 163
 - calls rpackfile, 163
 - calls rshut, 163
 - calls sayMSG, 163
 - calls unloadOneConstructor, 163
 - calls updateCategoryFrameForCategory, 163
 - calls updateCategoryFrameForConstructor, 163
 - local def \$libFile, 164
 - local def \$lisplibAbbreviation, 164
 - local def \$lisplibAncestors, 164
 - local def \$lisplibCategoriesExtended, 163
 - local def \$lisplibCategory, 164
 - local def \$lisplibForm, 163
 - local def \$lisplibKind, 164
 - local def \$lisplibModemapAlist, 164
 - local def \$lisplibModemap, 164
 - local def \$lisplibOperationAlist, 164
 - local def \$lisplibParents, 163
 - local def \$lisplibPredicates, 163
 - local def \$lisplibSlot1, 164
 - local def \$lisplibSuperDomain, 164
 - local def \$lisplibVariableAlist, 164
 - local def \$lisplib, 163
 - local def \$newConlist, 164
 - local def \$op, 163
 - local ref \$algebraOutputStream, 163
 - local ref \$compileDocumentation, 163
 - local ref \$filep, 163
 - local ref \$lisplibKind, 163
 - local ref \$newConlist, 163
 - local ref \$spadLibFT, 163
 - defun, 163
- compDefWhereClause, 155
 - calledby compDefine1, 141
 - calls assocleft, 156
 - calls assocright, 156
 - calls comp, 156
 - calls concat, 156
 - calls delete, 156
 - calls getmode, 155
 - calls lassoc, 156
 - calls listOfIdentifiersIn, 156
 - calls orderByDependency, 156
 - calls pairList, 156
 - calls union, 156
 - calls userError, 156
 - uses \$predAlist, 156
 - uses \$sigAlist, 156
 - defun, 155
- compElt, 306
 - calls addDomain, 306
 - calls compForm, 306
 - calls convert, 306
 - calls getDeltaEntry, 306
 - calls getModemapListFromDomain, 306
 - calls isDomainForm, 306
 - calls length, 306
 - calls opOf, 306
 - calls stackMessage, 306
 - calls stackWarning, 306
 - uses \$One, 306
 - uses \$Zero, 306
 - defun, 306
- compExit, 308
 - calls comp, 308
 - calls modifyModeStack, 308
 - calls stackMessageIfNone, 308
 - uses \$exitModeStack, 308
 - defun, 308
- compExpression, 135
 - calledby comp3, 592

- calls compForm, 135
 - calls getl, 135
 - uses \$insideExpressionIfTrue, 135
 - defun, 135
- compExpressionList, 609
 - calledby compForm1, 603
 - calls comp, 609
 - calls convert, 609
 - calls nreverse0, 609
 - local ref \$Expression, 609
 - defun, 609
- compFocompFormWithModemap, 613
 - calls coerceable, 613
 - calls compApplyModemap, 613
 - calls convert, 613
 - calls get, 613
 - calls identp, 613
 - calls isCategoryForm, 613
 - calls isFunctor, 613
 - calls last, 613
 - calls listOfSharpVars, 613
 - calls substituteIntoFunctorModemap, 613
 - local ref \$Category, 613
 - local ref \$FormalMapVariableList, 613
 - defun, 613
- compForm, 602
 - calledby compConstruct, 295
 - calledby compCons, 294
 - calledby compElt, 306
 - calledby compExpression, 135
 - calls compArgumentsAndTryAgain, 602
 - calls compForm1, 602
 - calls stackMessageIfNone, 602
 - defun, 602
- compForm1, 603
 - calledby compArgumentsAndTryAgain, 616
 - calledby compForm, 602
 - calls addDomain, 603
 - calls augModemapsFromDomain1, 603
 - calls coerceable, 603
 - calls coerce, 603
 - calls compExpressionList, 603
 - calls compForm2, 603
 - calls compOrCroak, 603
 - calls compToApply, 603
 - calls comp, 603
 - calls getFormModemaps, 603
 - calls length, 603
 - calls nreverse0, 603
 - calls outputComp, 603
 - uses \$EmptyMode, 603
 - uses \$Expression, 603
 - uses \$NumberOfArgsIfInteger, 603
 - defun, 603
- compForm2, 610
 - calledby compForm1, 603
 - calls PredImplies, 610
 - calls assoc, 610
 - calls compForm3, 610
 - calls compFormPartiallyBottomUp, 610
 - calls compUniquely, 610
 - calls isSimple, 610
 - calls length, 610
 - calls nreverse0, 610
 - calls sublis, 610
 - calls take, 610
 - uses \$EmptyMode, 610
 - uses \$TriangleVariableList, 610
 - defun, 610
- compForm3, 612
 - calledby compForm2, 610
 - calledby compFormPartiallyBottomUp, 615
 - calls compFormWithModemap, 612
 - local ref \$compUniquelyIfTrue, 612
 - defun, 612
 - throws, 612
- compFormMatch, 615
 - calledby compFormPartiallyBottomUp, 615
 - defun, 615
- compForMode, 321
 - calledby compJoin, 319
 - calls comp, 321
 - local def \$compForModeIfTrue, 321
 - defun, 321
- compFormPartiallyBottomUp, 615
 - calledby compForm2, 610
 - calls compForm3, 615
 - calls compFormMatch, 615
 - defun, 615
- compFormWithModemap
 - calledby compForm3, 612
- compFromIf, 312

- calledby compIf, 311
- calls comp, 312
- defun, 312
- compFunctorBody, 168
 - calledby compDefineFunctor1, 145
 - calls bootstrapError, 168
 - calls compOrCroak, 168
 - uses /editfile, 168
 - uses \$NRTaddForm, 168
 - uses \$bootstrapMode, 168
 - uses \$functorForm, 168
 - defun, 168
- compHas, 309
 - calls chaseInferences, 309
 - calls coerce, 309
 - calls compHasFormat, 309
 - local def \$e, 309
 - local ref \$Boolean, 309
 - local ref \$e, 309
 - defun, 309
- compHasFormat, 309
 - calledby compHas, 309
 - calls comp, 309
 - calls isDomainForm, 309
 - calls length, 309
 - calls mkDomainConstructor, 309
 - calls mkList, 309
 - calls sublis, 309
 - calls take, 309
 - local ref \$EmptyEnvironment, 309
 - local ref \$EmptyMode, 309
 - local ref \$FormalMapVariableList, 309
 - local ref \$e, 309
 - local ref \$form, 309
 - defun, 309
- compIf, 311
 - calls canReturn, 311
 - calls coerce, 311
 - calls compBoolean, 311
 - calls compFromIf, 311
 - calls intersectionEnvironment, 311
 - calls quotify, 311
 - calls resolve, 311
 - uses \$Boolean, 311
 - defun, 311
- compile, 169
 - calledby compDefineCategory2, 160
 - calledby compDefineFunctor1, 145
 - calledby compileCases, 167
 - calls addStats, 169
 - calls constructMacro, 169
 - calls elapsedTime, 169
 - calls encodeFunctionName, 169
 - calls encodeItem, 169
 - calls getmode, 169
 - calls get, 169
 - calls kar, 169
 - calls member, 169
 - calls modeEqual, 169
 - calls optimizeFunctionDef, 169
 - calls printStats, 169
 - calls putInLocalDomainReferences, 169
 - calls sayBrightly, 169
 - calls spadCompileOrSetq, 169
 - calls splitEncodedFunctionName, 169
 - calls strconc, 169
 - calls userError, 169
 - local def \$functionStats, 170
 - local def \$savableItems, 170
 - local def \$suffix, 170
 - local ref \$compileOnlyCertainItems, 169
 - local ref \$doNotCompileJustPrint, 169
 - local ref \$e, 170
 - local ref \$functionStats, 169
 - local ref \$functorForm, 169
 - local ref \$insideCapsuleFunctionIfTrue, 169
 - local ref \$lisplibItemsAlreadyThere, 169
 - local ref \$lisplib, 169
 - local ref \$macroIfTrue, 169
 - local ref \$prefix, 170
 - local ref \$saveableItems, 169
 - local ref \$signatureOfForm, 169, 170
 - local ref \$splitUpItemsAlreadyThere, 169
 - local ref \$suffix, 170
 - defun, 169
- compile-lib-file, 628
 - calledby recompile-lib-file-if-necessary, 627
 - defun, 628
- compileCases, 167
 - calledby compDefineCapsuleFunction, 152
 - calls assocleft, 167

- calls assocright, 167
- calls compile, 167
- calls eval, 167
- calls getSpecialCaseAssoc, 167
- calls get, 167
- calls mkpf, 167
- calls outerProduct, 167
- local def \$specialCaseKeyList, 167
- local ref \$getDomainCode, 167
- local ref \$insideFunctorIfTrue, 167
- defun, 167
- compileConstructor, 183
 - calledby spadCompileOrSetq, 182
 - calls clearClams, 183
 - calls compileConstructor1, 183
 - defun, 183
- compileConstructor1, 184
 - calledby compileConstructor, 183
 - calls clearConstructorCache, 184
 - calls compAndDefine, 184
 - calls comp, 184
 - calls getdatabase, 184
 - local def \$clamList, 184
 - local ref \$ConstructorCache, 184
 - local ref \$clamList, 184
 - local ref \$mutableDomain, 184
 - defun, 184
- compiled-function-p
 - calledby subrname, 228
- compileDocumentation, 165
 - calledby compDefineLisplib, 163
 - calls finalizeDocumentation, 165
 - calls lisplibWrite, 165
 - calls makeInputFilename, 165
 - calls rdefiostream, 165
 - calls replaceFile, 165
 - calls rpackfile, 165
 - calls rshut, 165
 - local ref \$EmptyMode, 166
 - local ref \$e, 166
 - local ref \$fcopy, 165
 - local ref \$spadLibFT, 165
 - defun, 165
- compileFileQuietly, 628
 - uses *standard-output*, 628
 - uses \$InteractiveMode, 628
- defun, 628
- compiler, 562
 - calls compileSpad2Cmd, 563
 - calls compileSpadLispCmd, 563
 - calls findfile, 563
 - calls helpSpad2Cmd[5], 563
 - calls mergePathnames[5], 563
 - calls namestring[5], 563
 - calls pathnameType[5], 563
 - calls pathname[5], 563
 - calls selectOptionLC[5], 563
 - calls throwKeyedMsg, 563
 - uses /editfile, 563
 - uses \$newConlist, 563
 - uses \$options, 563
 - defun, 562
- compilerDoit, 570
 - calledby compileSpad2Cmd, 566
 - calledby compilerDoitWithScreenedLisplib, 569
 - calls /RQ,LIB, 570
 - calls /rf[5], 570
 - calls /rq[5], 570
 - calls member[5], 570
 - calls opOf, 570
 - calls sayBrightly, 570
 - uses \$byConstructors, 570
 - uses \$constructorsSeen, 570
 - defun, 570
- compilerDoitWithScreenedLisplib
 - calledby compileSpad2Cmd, 566
 - calls compilerDoit, 569
 - calls embed, 569
 - calls rwrite, 569
 - calls unembed, 569
 - local ref \$libFile, 570
 - local ref \$saveableItems, 569
- compilerMessage
 - calledby augModemapsFromCategoryRep, 267
 - calledby augModemapsFromCategory, 260
- compileSpad2Cmd, 565
 - calledby compiler, 563
 - calls browserAutoloadOnceTrigger, 566
 - calls compilerDoitWithScreenedLisplib, 566
 - calls compilerDoit, 566

- calls error, 566
- calls extendLocalLibdb, 566
- calls namestring[5], 566
- calls object2String, 566
- calls pathnameType[5], 566
- calls pathname[5], 566
- calls sayKeyedMsg[5], 566
- calls selectOptionLC[5], 566
- calls spad2AsTranslatorAutoloadOnceTrigger, 566
- calls spadPrompt, 566
- calls strconc, 566
- calls terminateSystemCommand[5], 566
- calls throwKeyedMsg, 566
- calls updateSourceFiles[5], 566
- uses /editfile, 567
- uses \$InteractiveMode, 566
- uses \$QuickCode, 566
- uses \$QuickLet, 566
- uses \$compileOnlyCertainItems, 566
- uses \$f, 566
- uses \$m, 566
- uses \$newComp, 566
- uses \$newConlist, 567
- uses \$options, 566
- uses \$scanIfTrue, 566
- uses \$sourceFileTypes, 566
- defun, 565
- compileSpadLispCmd, 568
 - calledby compiler, 563
 - calls fnameMake[5], 568
 - calls fnameReadable?[5], 568
 - calls localdatabase[5], 568
 - calls namestring[5], 568
 - calls object2String, 568
 - calls pathnameDirectory[5], 568
 - calls pathnameName[5], 568
 - calls pathnameType[5], 568
 - calls pathname[5], 568
 - calls recompile-lib-file-if-necessary, 568
 - calls sayKeyedMsg[5], 568
 - calls selectOptionLC[5], 568
 - calls spadPrompt, 568
 - calls terminateSystemCommand[5], 568
 - calls throwKeyedMsg, 568
 - uses \$options, 568
- defun, 568
- compileTimeBindingOf, 233
 - calledby optSpecialCall, 232
 - calls bpiname, 233
 - calls keyedSystemError, 233
 - calls moan, 233
 - defun, 233
- compImport, 318
 - calls addDomain, 318
 - uses \$NoValueMode, 318
 - defun, 318
- compInternalFunction, 155
 - calledby compDefine1, 141
 - calls identp, 155
 - calls stackAndThrow, 155
 - defun, 155
- compls, 318
 - calls coerce, 318
 - calls comp, 318
 - uses \$Boolean, 319
 - uses \$EmptyMode, 319
 - defun, 318
- complerator
 - calledby compReduce1, 326
 - calledby compRepeatOrCollect, 329
- compJoin, 319
 - calls compForMode, 319
 - calls compJoin,getParms, 319
 - calls convert, 320
 - calls isCategoryForm, 319
 - calls nreverse0, 319
 - calls stackSemanticError, 319
 - calls union, 319
 - calls wrapDomainSub, 319
 - uses \$Category, 320
 - defun, 319
- compJoin,getParms
 - calledby compJoin, 319
- compLambda, 321
 - calledby compWithMappingMode1, 617
 - calls argsToSig, 321
 - calls compAtSign, 321
 - calls stackAndThrow, 321
 - defun, 321
- compLeave, 323
 - calls comp, 323

- calls modifyModeStack, 323
 - uses \$exitModeStack, 323
 - uses \$leaveLevelStack, 323
 - defun, 323
- compList, 602
 - calledby compAtom, 597
 - calledby compConstruct, 295
 - calls comp, 602
 - defun, 602
- compMacro, 323
 - calls formatUnabbreviated, 323
 - calls macroExpand, 323
 - calls put, 323
 - calls sayBrightly, 323
 - uses \$EmptyMode, 323
 - uses \$NoValueMode, 323
 - uses \$macroIfTrue, 323
 - defun, 323
- compMakeCategoryObject, 208
 - calledby compDefineFunctor1, 144
 - calledby getOperationAlist, 265
 - calledby getSlotFromFunctor, 208
 - calls isCategoryForm, 208
 - calls mkEvaluableCategoryForm, 209
 - local ref \$Category, 209
 - local ref \$e, 209
 - defun, 208
- compMakeDeclaration, 624
 - calledby compDefineCategory2, 159
 - calledby compDefineFunctor1, 144
 - calledby compSetq1, 336
 - calledby compSubDomain1, 346
 - calledby compWithMappingMode1, 617
 - calls compColon, 624
 - uses \$insideExpressionIfTrue, 624
 - defun, 624
- compMapCond, 256
 - calledby compApplyModemap, 255
 - calls compMapCond', 256
 - local ref \$bindings, 256
 - defun, 256
- compMapCond', 257
 - calledby compMapCond, 256
 - calls compMapCond", 257
 - calls compMapConfFun, 257
 - calls stackMessage, 257
 - defun, 257
- compMapCond", 257
 - calledby compMapCond", 257
 - calledby compMapCond', 257
 - calls compMapCond", 257
 - calls get, 257
 - calls knownInfo, 257
 - calls stackMessage, 257
 - local ref \$Information, 257
 - local ref \$e, 257
 - defun, 257
- compMapCondFun, 258
 - defun, 258
- compMapConfFun
 - calledby compMapCond', 257
- compNoStacking, 590
 - calledby compToApply, 605
 - calledby comp, 590
 - calls comp2, 590
 - calls compNoStacking1, 590
 - local ref \$compStack, 590
 - uses \$EmptyMode, 590
 - uses \$Representation, 590
 - defun, 590
- compNoStacking1, 591
 - calledby compNoStacking, 590
 - calls comp2, 591
 - calls get, 591
 - local ref \$compStack, 591
 - defun, 591
- compOrCroak, 588
 - calledby NRTgetLocalIndex, 211
 - calledby compAdd, 271
 - calledby compArgumentConditions, 166
 - calledby compDefineCapsuleFunction, 152
 - calledby compDefineCategory2, 160
 - calledby compForm1, 603
 - calledby compFunctorBody, 168
 - calledby compRepeatOrCollect, 329
 - calledby compSubDomain1, 346
 - calledby compTopLevel, 586
 - calledby doIt, 277
 - calledby getTargetFromRhs, 173
 - calledby makeCategoryForm, 293
 - calledby mkEvaluableCategoryForm, 178

- calledby substituteIntoFunctorModemap, 614
- calls compOrCroak1, 588
- defun, 588
- compOrCroak1, 589
 - calledby compOrCroak, 588
 - calls compOrCroak1, compactify, 589
 - calls comp, 589
 - calls displayComp, 589
 - calls displaySemanticErrors, 589
 - calls mkErrorExpr, 589
 - calls say, 589
 - calls stackSemanticError, 589
 - calls userError, 589
 - local def \$compStack, 589
 - uses \$compErrorMessageStack, 589
 - uses \$exitModeStack, 589
 - uses \$level, 589
 - uses \$scanIfTrue, 589
 - uses \$s, 589
 - catches, 589
 - defun, 589
- compOrCroak1, compactify, 626
 - calledby compOrCroak1, compactify, 626
 - calledby compOrCroak1, 589
 - calls compOrCroak1, compactify, 626
 - calls lassoc, 626
 - defun, 626
- compPretend, 324
 - calls addDomain, 324
 - calls comp, 325
 - calls opOf, 325
 - calls stackSemanticError, 325
 - calls stackWarning, 325
 - uses \$EmptyMode, 325
 - uses \$newCompilerUnionFlag, 325
 - defun, 324
- compQuote, 326
 - defun, 326
- compReduce, 326
 - calls compReduce1, 326
 - uses \$formalArgList, 326
 - defun, 326
- compReduce1, 326
 - calledby compReduce, 326
 - calls compIterator, 326
 - calls comp, 326
 - calls getIdentity, 327
 - calls nreverse0, 326
 - calls parseTran, 326
 - calls systemError, 326
 - uses \$Boolean, 327
 - uses \$endTestList, 327
 - uses \$e, 327
 - uses \$initList, 327
 - uses \$sideEffectsList, 327
 - uses \$until, 327
 - defun, 326
- compRepeatOrCollect, 329
 - calls coerceExit, 329
 - calls compIterator, 329
 - calls compOrCroak, 329
 - calls comp, 329
 - calls length, 329
 - calls modelsAggregateOf, 329
 - calls stackMessage, 329
 - calls , 329
 - uses \$Boolean, 329
 - uses \$NoValueMode, 329
 - uses \$exitModeStack, 329
 - uses \$formalArgList, 329
 - uses \$leaveLevelStack, 329
 - uses \$until, 329
 - defun, 329
- compReturn, 331
 - calls comp, 331
 - calls modifyModeStack, 331
 - calls resolve, 331
 - calls stackSemanticError, 331
 - calls userError, 331
 - uses \$exitModeStack, 331
 - uses \$returnMode, 331
 - defun, 331
- compSeq, 332
 - calls compSeq1, 332
 - uses \$exitModeStack, 332
 - defun, 332
- compSeq1, 332
 - calledby compSeq, 332
 - calls compSeqItem, 332
 - calls mkq, 332
 - calls nreverse0, 332

- calls `replaceExitEtc`, 332
 - uses `$NoValueMode`, 333
 - uses `$exitModeStack`, 332
 - uses `$finalEnv`, 333
 - uses `$insideExpressionIfTrue`, 332
 - defun, 332
- `compSeqItem`, 334
 - calledby `compSeq1`, 332
 - calls `comp`, 334
 - calls `macroExpand`, 334
 - defun, 334
- `compSetq`, 335
 - calledby `compSetq1`, 336
 - calls `compSetq1`, 335
 - defun, 335
- `compSetq1`, 335
 - calledby `compSetq`, 335
 - calledby `setqMultipleExplicit`, 339
 - calledby `setqMultiple`, 337
 - calls `compMakeDeclaration`, 336
 - calls `compSetq`, 336
 - calls `identp[5]`, 336
 - calls `setqMultiple`, 336
 - calls `setqSetelt`, 336
 - calls `setqSingle`, 335
 - uses `$EmptyMode`, 336
 - defun, 335
- `compSingleCapsuleItem`, 276
 - calledby `compCapsuleItems`, 275
 - calledby `doItIf`, 281
 - calledby `doIt`, 276
 - calls `doit`, 276
 - calls `macroExpandInPlace`, 276
 - local ref `$e`, 276
 - local ref `$pred`, 276
 - defun, 276
- `compString`, 345
 - calls `resolve`, 345
 - uses `$StringCategory`, 345
 - defun, 345
- `compSubDomain`, 346
 - calls `compCapsule`, 346
 - calls `compSubDomain1`, 346
 - uses `$NRTaddForm`, 346
 - uses `$addFormLhs`, 346
 - uses `$addForm`, 346
 - defun, 346
- `compSubDomain1`, 346
 - calledby `compAdd`, 271
 - calledby `compSubDomain`, 346
 - calls `addDomain`, 346
 - calls `compMakeDeclaration`, 346
 - calls `compOrCroak`, 346
 - calls `evalAndRwriteLispForm`, 347
 - calls `lispize`, 347
 - calls `stackSemanticError`, 346
 - uses `$Boolean`, 347
 - uses `$CategoryFrame`, 347
 - uses `$EmptyMode`, 347
 - uses `$lisplibSuperDomain`, 347
 - uses `$op`, 347
 - defun, 346
- `compSubsetCategory`, 348
 - calls `comp`, 348
 - calls `put`, 348
 - uses `$lhsOfColon`, 348
 - defun, 348
- `compSuchthat`, 349
 - calls `comp`, 349
 - calls `put`, 349
 - uses `$Boolean`, 349
 - defun, 349
- `compSymbol`, 600
 - calledby `compAtom`, 597
 - calls `NRTgetLocalIndex`, 601
 - calls `errorRef`, 601
 - calls `getmode`, 601
 - calls `get`, 601
 - calls `isFluid`, 600
 - calls `isFunction`, 601
 - calls `member[5]`, 601
 - calls `stackMessage`, 601
 - uses `$Boolean`, 601
 - uses `$Expression`, 601
 - uses `$FormalMapVariableList`, 601
 - uses `$NoValueMode`, 601
 - uses `$NoValue`, 601
 - uses `$Symbol`, 601
 - uses `$compForModeIfTrue`, 601
 - uses `$formalArgList`, 601
 - uses `$functorLocalParameters`, 601
 - defun, 600

- compToApply, 605
 - calledby compForm1, 603
 - calls compApplication, 605
 - calls compNoStacking, 605
 - local ref \$EmptyMode, 605
 - defun, 605
- compTopLevel, 586
 - calledby s-process, 583
 - calls compOrCroak, 586
 - uses \$NRTderivedTargetIfTrue, 586
 - uses \$compTimeSum, 586
 - uses \$envHashTable, 586
 - uses \$forceAdd, 586
 - uses \$skillOptimizeIfTrue, 586
 - uses \$packagesUsed, 586
 - uses \$resolveTimeSum, 586
 - defun, 586
- compTuple2Record, 273
 - calledby compAdd, 271
 - defun, 273
- compTypeOf, 596
 - calledby comp3, 592
 - calls comp3, 596
 - calls eqsubstlist, 596
 - calls get, 596
 - calls put, 596
 - uses \$FormalMapVariableList, 596
 - uses \$insideCompTypeOf, 596
 - defun, 596
- compUniquely, 616
 - calledby compForm2, 610
 - calls comp, 616
 - local def \$compUniquelyIfTrue, 616
 - catches, 616
 - defun, 616
- computeAncestorsOf
 - calledby compDefineCategory2, 160
 - calledby compDefineFunctor1, 145
- compVector, 349
 - calledby compAtom, 597
 - calledby compConstruct, 295
 - calls comp, 350
 - uses \$EmptyVector, 350
 - defun, 349
- compWhere, 350
 - calls addContour, 351
 - calls comp, 350
 - calls deltaContour, 351
 - calls macroExpand, 351
 - uses \$EmptyMode, 351
 - uses \$insideExpressionIfTrue, 351
 - uses \$insideWhereIfTrue, 351
 - defun, 350
- compWithMappingMode, 617
 - calledby comp3, 592
 - calls compWithMappingMode1, 617
 - uses \$formalArgList, 617
 - defun, 617
- compWithMappingMode1, 617
 - calledby compWithMappingMode, 617
 - calls comp-tran, 618
 - calls compLambda, 617
 - calls compMakeDeclaration, 617
 - calls comp, 617
 - calls extendsCategoryForm, 617
 - calls extractCodeAndConstructTriple, 617
 - calls freelist, 618
 - calls get, 617
 - calls hasFormalMapVariable, 617
 - calls isFunctor, 617
 - calls optimizeFunctionDef, 618
 - calls stackAndThrow, 617
 - calls take, 617
 - uses \$CategoryFrame, 618
 - uses \$EmptyMode, 618
 - uses \$FormalMapVariableList, 618
 - uses \$QuickCode, 618
 - uses \$formalArgList, 618
 - uses \$formatArgList, 618
 - uses \$funnameTail, 618
 - uses \$funname, 618
 - uses \$skillOptimizeIfTrue, 618
 - defun, 617
- concat
 - calledby checkDocError, 517
 - calledby checkDocMessage, 520
 - calledby checkWarning, 549
 - calledby compDefWhereClause, 156
- concatWithBlanks
 - calledby buildLibAttr, 485
 - calledby buildLibOp, 484
 - calledby buildLibdbConEntry, 482

- cond, [242](#)
 - defplist, [242](#)
- cons, [294](#)
 - defplist, [294](#)
- consProplistOf
 - calledby getSuccessEnvironment, [315](#)
 - calledby setqSingle, [340](#)
- construct, [99](#), [295](#), [383](#)
 - defplist, [99](#), [295](#), [383](#)
- constructMacro, [182](#)
 - calledby compile, [169](#)
 - calls identp, [182](#)
 - calls stackSemanticError, [182](#)
 - defun, [182](#)
- constructor?
 - calledby addDomain, [248](#)
 - calledby checkIsValidType, [532](#)
 - calledby checkNumOfArgs, [534](#)
 - calledby compDefineCategory2, [160](#)
 - calledby compDefineFunctor1, [145](#)
 - calledby getAbbreviation, [298](#)
 - calledby isFunctor, [249](#)
- contained
 - calledby evalAndSub, [265](#)
 - calledby parseCategory, [103](#)
 - calledby spadCompileOrSetq, [182](#)
 - calledby substVars, [198](#)
- containsBang, [404](#)
 - calledby aplTran, [401](#)
 - calledby containsBang, [404](#)
 - calls containsBang, [404](#)
 - defun, [404](#)
- convert, [600](#)
 - calledby applyMapping, [593](#)
 - calledby compAtomWithModemap, [598](#)
 - calledby compAtom, [597](#)
 - calledby compCons1, [294](#)
 - calledby compConstruct, [295](#)
 - calledby compElt, [306](#)
 - calledby compExpressionList, [609](#)
 - calledby compFocompFormWithModemap, [613](#)
 - calledby compJoin, [320](#)
 - calledby convertOrCroak, [334](#)
 - calledby setqMultiple, [337](#)
 - calledby setqSingle, [340](#)
 - calls coerce, [600](#)
 - calls resolve, [600](#)
 - defun, [600](#)
- convertOpAlist2compilerInfo, [116](#)
 - calledby updateCategoryFrameForConstructor, [116](#)
 - defun, [116](#)
- convertOrCroak, [334](#)
 - calledby replaceExitEtc, [333](#)
 - calls convert, [334](#)
 - calls userError, [334](#)
 - defun, [334](#)
- copy
 - calledby modifyModeStack, [625](#)
- copy-token
 - calledby PARSE-TokTail, [430](#)
 - calledby advance-token, [462](#)
- croak
 - calledby drop, [553](#)
- curoutstream
 - usedby s-process, [584](#)
 - usedby spad, [575](#)
- current-char, [463](#)
 - calledby Advance-Char, [637](#)
 - calledby PARSE-FloatBasePart, [437](#)
 - calledby PARSE-FloatBase, [436](#)
 - calledby PARSE-FloatExponent, [437](#)
 - calledby PARSE-Selector, [433](#)
 - calledby PARSE-TokTail, [430](#)
 - calledby match-string, [454](#)
 - calledby skip-blanks, [454](#)
 - uses \$line, [463](#)
 - uses current-line, [463](#)
 - defun, [463](#)
- current-fragment, [631](#)
 - defvar, [631](#)
- current-line, [634](#)
 - usedby PARSE-Category, [424](#)
 - usedby current-char, [463](#)
 - usedby next-char, [464](#)
 - defvar, [634](#)
- current-symbol, [460](#)
 - calledby PARSE-AnyId, [444](#)
 - calledby PARSE-ElseClause, [451](#)
 - calledby PARSE-FloatBase, [436](#)
 - calledby PARSE-FloatExponent, [437](#)

- calledby PARSE-Infix, [429](#)
- calledby PARSE-NewExpr, [417](#)
- calledby PARSE-OpenBrace, [446](#)
- calledby PARSE-OpenBracket, [446](#)
- calledby PARSE-Operation, [427](#)
- calledby PARSE-Prefix, [428](#)
- calledby PARSE-Primary1, [434](#)
- calledby PARSE-ReductionOp, [431](#)
- calledby PARSE-Selector, [433](#)
- calledby PARSE-SpecialCommand, [419](#)
- calledby PARSE-SpecialKeyWord, [418](#)
- calledby PARSE-Suffix, [448](#)
- calledby PARSE-TokTail, [430](#)
- calledby PARSE-TokenList, [420](#)
- calledby isTokenDelimiter, [457](#)
- calls current-token, [460](#)
- calls make-symbol-of, [460](#)
- defun, [460](#)
- current-token, [95](#), [461](#)
 - calledby PARSE-FloatBasePart, [437](#)
 - calledby PARSE-SpecialKeyWord, [418](#)
 - calledby advance-token, [462](#)
 - calledby current-symbol, [460](#)
 - calledby match-advance-string, [455](#)
 - calledby match-current-token, [459](#)
 - calledby next-token, [462](#)
 - calls try-get-token, [461](#)
 - usedby advance-token, [462](#)
 - usedby current-token, [461](#)
 - uses \$token, [95](#)
 - uses current-token, [461](#)
 - uses valid-tokens, [461](#)
- defun, [461](#)
- defvar, [95](#)
- curstrm
 - calledby s-process, [583](#)
- dbKind
 - calledby screenLocalLine, [486](#)
- dbMkForm
 - calledby buildLibdbConEntry, [482](#)
- dbName
 - calledby screenLocalLine, [486](#)
- dbPart
 - calledby screenLocalLine, [486](#)
- dbReadLines, [481](#)
 - calledby extendLocalLibdb, [477](#)
 - calls eofp, [481](#)
 - calls readline, [481](#)
 - defun, [481](#)
- dbWriteLines, [481](#)
 - calledby extendLocalLibdb, [477](#)
 - calls getTempPath, [481](#)
 - calls ifcar, [481](#)
 - calls make-outstream, [481](#)
 - calls shut, [481](#)
 - calls writedb, [481](#)
 - local def \$outStream, [481](#)
 - local ref \$outStream, [481](#)
 - defun, [481](#)
- dcq
 - calledby new2OldTran, [72](#)
- decodeScripts, [405](#)
 - calledby decodeScripts, [405](#)
 - calledby getScriptName, [404](#)
 - calls decodeScripts, [405](#)
 - calls strconc, [405](#)
 - defun, [405](#)
- deepestExpression, [404](#)
 - calledby deepestExpression, [404](#)
 - calledby hasAplExtension, [403](#)
 - calls deepestExpression, [404](#)
 - defun, [404](#)
- def, [105](#), [140](#)
 - defplist, [105](#), [140](#)
- def-process
 - calledby s-process, [583](#)
- def-rename, [587](#)
 - calledby def-rename, [587](#)
 - calledby s-process, [583](#)
 - calledby string2BootTree, [71](#)
 - calls def-rename, [587](#)
 - defun, [587](#)
- definition-name, [417](#)
 - usedby PARSE-NewExpr, [417](#)
 - defvar, [417](#)
- defmacro
 - Bang, [466](#)
 - line-clear, [634](#)
 - must, [466](#)
 - nth-stack, [551](#)
 - pop-stack-1, [550](#)

- pop-stack-2, 550
- pop-stack-3, 551
- pop-stack-4, 551
- reduce-stack-clear, 468
- stack-/empty, 93
- star, 467
- defplist, 101, 240, 271, 348, 357, 374
 - $+- >$, 321
 - $- >$, 390
 - $<=$, 125
 - $==>$, 391
 - $=>$, 386
 - $>$, 111
 - $>=$, 110, 111
 - $.,$ 382
 - $-$, 237
 - $/$, 397
 - $;$, 104, 290, 380
 - $::$, 103, 358, 381
 - $:BF;$, 375
 - $;$, 395
 - $==$, 384
 - add, 372
 - and, 101
 - Block, 376
 - call, 229
 - capsule, 273
 - case, 283
 - catch, 240
 - category, 102, 286, 377
 - collect, 328, 379
 - cond, 242
 - cons, 294
 - construct, 99, 295, 383
 - def, 105, 140
 - dollargreaterequal, 108
 - dollargreaterthan, 107
 - dollarnotequal, 109
 - elt, 306
 - eq, 235
 - eqv, 110
 - exit, 307
 - has, 112, 308
 - if, 117, 310, 387
 - implies, 120
 - import, 318
 - In, 389
 - in, 121, 388
 - inby, 122
 - is, 123, 318
 - isnt, 123
 - Join, 124, 319, 389
 - leave, 124, 322
 - lessp, 238
 - let, 126, 335
 - letd, 126
 - ListCategory, 296
 - Mapping, 285
 - mdef, 127, 323
 - minus, 236
 - mkRecord, 245
 - not, 128
 - notequal, 129
 - or, 129
 - pretend, 130, 324, 392
 - qsminus, 237
 - quote, 325, 392
 - Record, 284
 - RecordCategory, 296
 - recordcopy, 247
 - recordelt, 245
 - reduce, 326, 393
 - repeat, 328, 394
 - return, 131, 331
 - Scripts, 394
 - segment, 131, 132
 - seq, 234, 332
 - setq, 335
 - setrecordelt, 246
 - Signature, 396
 - spadcall, 239
 - String, 345
 - SubDomain, 346
 - SubsetCategory, 348
 - TupleCollect, 398
 - Union, 285
 - UnionCategory, 297
 - vcons, 132
 - vector, 349
 - VectorCategory, 297
 - where, 133, 350, 399
 - with, 399

- defstruct
 - line, 633
 - reduction, 96
 - stack, 92
 - token, 94
- defun
 - /RQ,LIB, 572
 - /rf, 571
 - /rf-1, 572
 - /rq, 571
 - action, 467
 - add-parens-and-semis-to-line, 88
 - addArgumentConditions, 300
 - addclose, 552
 - addConstructorModemaps, 254
 - addDomain, 248
 - addEltModemap, 261
 - addEmptyCapsuleIfNecessary, 173
 - addModemap, 269
 - addModemap0, 269
 - addModemap1, 270
 - addModemapKnown, 268
 - addNewDomain, 251
 - addSuffix, 299
 - Advance-Char, 637
 - advance-token, 462
 - alistSize, 299
 - allLASSOCs, 213
 - aplTran, 401
 - aplTran1, 401
 - aplTranList, 403
 - applyMapping, 593
 - argsToSig, 623
 - assignError, 342
 - AssocBarGensym, 222
 - augLisplibModemapsFromCategory, 187
 - augmentLisplibModemapsFromFunctor, 212
 - augModemapsFromCategory, 260
 - augModemapsFromCategoryRep, 267
 - augModemapsFromDomain, 252
 - augModemapsFromDomain1, 252
 - autoCoerceByModemap, 360
 - blankp, 552
 - bootStrapError, 215
 - buildLibAttr, 485
 - buildLibAttrs, 485
 - buildLibdb, 478
 - buildLibdbConEntry, 482
 - buildLibdbString, 480
 - buildLibOp, 484
 - buildLibOps, 484
 - bumperrorcount, 545
 - canReturn, 312
 - char-eq, 464
 - char-ne, 464
 - checkAddBackSlashes, 527
 - checkAddIndented, 519
 - checkAddMacros, 528
 - checkAddPeriod, 529
 - checkAddSpaces, 530
 - checkAddSpaceSegments, 529
 - checkAlphabetic, 531
 - checkAndDeclare, 304
 - checkArguments, 501
 - checkBalance, 501
 - checkBeginEnd, 502
 - checkComments, 498
 - checkDecorate, 504
 - checkDecorateForHt, 506
 - checkDocError, 517
 - checkDocError1, 507
 - checkDocMessage, 519
 - checkExtract, 520
 - checkFixCommonProblem, 508
 - checkGetArgs, 521
 - checkGetLispFunctionName, 508
 - checkGetMargin, 522
 - checkGetParse, 522
 - checkGetStringBeforeRightBrace, 523
 - checkHTargs, 509
 - checkIeEg, 523
 - checkIeEgfun, 531
 - checkIndentedLines, 524
 - checkIsValidType, 532
 - checkLookForLeftBrace, 533
 - checkLookForRightBrace, 533
 - checkNumOfArgs, 534
 - checkRecordHash, 509
 - checkRemoveComments, 518
 - checkRewrite, 499
 - checkSayBracket, 534
 - checkSkipBlanks, 534

- checkSkipIdentifierToken, 525
- checkSkipOpToken, 525
- checkSkipToken, 518
- checkSplit2Words, 518
- checkSplitBackslash, 535
- checkSplitBrace, 525
- checkSplitOn, 536
- checkSplitPunctuation, 537
- checkTexht, 512
- checkTransformFirsts, 513
- checkTrim, 516
- checkTrimCommented, 526
- checkWarning, 549
- coerce, 351
- coerceable, 356
- coerceByModemap, 359
- coerceEasy, 352
- coerceExit, 357
- coerceExtraHard, 355
- coerceHard, 354
- coerceSubset, 353
- collectAndDeleteAssoc, 472
- collectComBlock, 471
- comma2Tuple, 382
- comp, 590
- comp2, 591
- comp3, 592
- compAdd, 271
- compAndDefine, 185
- compApplication, 605
- compApply, 595
- compApplyModemap, 255
- compArgumentConditions, 166
- compArgumentsAndTryAgain, 616
- compAtom, 597
- compAtomWithModemap, 598
- compAtSign, 357
- compBoolean, 314
- compCapsule, 274
- compCapsuleInner, 274
- compCapsuleItems, 275
- compCase, 283
- compCase1, 284
- compCat, 285
- compCategory, 286
- compCategoryItem, 287
- compCoerce, 358
- compCoerce1, 359
- compColon, 290
- compColonInside, 596
- compCons, 294
- compCons1, 294
- compConstruct, 295
- compConstructorCategory, 297
- compDefine, 140
- compDefine1, 141
- compDefineAddSignature, 143
- compDefineCapsuleFunction, 151
- compDefineCategory, 158
- compDefineCategory1, 158
- compDefineCategory2, 159
- compDefineFunctor, 144
- compDefineFunctor1, 144
- compDefineLisplib, 163
- compDefWhereClause, 155
- compElt, 306
- compExit, 308
- compExpression, 135
- compExpressionList, 609
- compFocompFormWithModemap, 613
- compForm, 602
- compForm1, 603
- compForm2, 610
- compForm3, 612
- compFormMatch, 615
- compForMode, 321
- compFormPartiallyBottomUp, 615
- compFromIf, 312
- compFunctorBody, 168
- compHas, 309
- compHasFormat, 309
- compIf, 311
- compile, 169
- compile-lib-file, 628
- compileCases, 167
- compileConstructor, 183
- compileConstructor1, 184
- compileDocumentation, 165
- compileFileQuietly, 628
- compiler, 562
- compilerDoit, 570
- compileSpad2Cmd, 565

- compileSpadLispCmd, 568
- compileTimeBindingOf, 233
- compImport, 318
- compInternalFunction, 155
- compIs, 318
- compJoin, 319
- compLambda, 321
- compLeave, 323
- compList, 602
- compMacro, 323
- compMakeCategoryObject, 208
- compMakeDeclaration, 624
- compMapCond, 256
- compMapCond', 257
- compMapCond", 257
- compMapCondFun, 258
- compNoStacking, 590
- compNoStacking1, 591
- compOrCroak, 588
- compOrCroak1, 589
- compOrCroak1,compactify, 626
- compPretend, 324
- compQuote, 326
- compReduce, 326
- compReduce1, 326
- compRepeatOrCollect, 329
- compReturn, 331
- compSeq, 332
- compSeq1, 332
- compSeqItem, 334
- compSetq, 335
- compSetq1, 335
- compSingleCapsuleItem, 276
- compString, 345
- compSubDomain, 346
- compSubDomain1, 346
- compSubsetCategory, 348
- compSuchthat, 349
- compSymbol, 600
- compToApply, 605
- compTopLevel, 586
- compTuple2Record, 273
- compTypeOf, 596
- compUniquely, 616
- compVector, 349
- compWhere, 350
- compWithMappingMode, 617
- compWithMappingMode1, 617
- constructMacro, 182
- containsBang, 404
- convert, 600
- convertOpAlist2compilerInfo, 116
- convertOrCroak, 334
- current-char, 463
- current-symbol, 460
- current-token, 461
- dbReadLines, 481
- dbWriteLines, 481
- decodeScripts, 405
- deepestExpression, 404
- def-rename, 587
- disallowNilAttribute, 214
- displayMissingFunctions, 216
- displayPreCompilationErrors, 544
- doIt, 276
- doItIf, 281
- dollarTran, 465
- domainMember, 260
- drop, 553
- eltModemapFilter, 608
- encodeFunctionName, 179
- encodeItem, 181
- EqualBarGensym, 244
- errhuh, 409
- escape-keywords, 457
- escaped, 553
- evalAndRwriteLispForm, 200
- evalAndSub, 265
- expand-tabs, 91
- extendLocalLibdb, 477
- extractCodeAndConstructTriple, 622
- finalizeDocumentation, 492
- finalizeLisplib, 203
- fincomblock, 553
- firstNonBlankPosition, 538
- fixUpPredicate, 192
- flattenSignatureList, 190
- floatexpid, 465
- formal2Pattern, 214
- freelist, 625
- genDomainOps, 220
- genDomainView, 219

- genDomainViewList, 219
- genDomainViewList0, 219
- get-a-line, 638
- get-token, 463
- getAbbreviation, 298
- getArgumentMode, 305
- getArgumentModeOrMoan, 187
- getCaps, 181
- getCategoryOpsAndAtts, 205
- getConstructorExports, 475
- getConstructorOpsAndAtts, 205
- getDomainsInScope, 250
- getFormModemaps, 607
- getFunctorOpsAndAtts, 208
- getInverseEnvironment, 316
- getMatchingRightPren, 538
- getModemap, 254
- getModemapList, 259
- getModemapListFromDomain, 260
- getOperationAlist, 265
- getScriptName, 404
- getSignature, 302
- getSignatureFromMode, 299
- getSlotFromCategoryForm, 206
- getSlotFromFunctor, 208
- getSpecialCaseAssoc, 300
- getSuccessEnvironment, 314
- getTargetFromRhs, 173
- getToken, 458
- getUnionMode, 317
- getUniqueModemap, 259
- getUniqueSignature, 259
- giveFormalParametersValues, 174
- hackforis, 408
- hackforisl, 409
- hasAplExtension, 403
- hasFormalMapVariable, 623
- hasFullSignature, 172
- hasNoVowels, 539
- hasSigInTargetCategory, 304
- hasType, 356
- htcharPosition, 539
- indent-pos, 554
- infixtok, 555
- initial-substring, 637
- initial-substring-p, 456
- initialize-prepare, 76
- initializeLisplib, 202
- insertModemap, 263
- interactiveModemapForm, 191
- is-console, 555
- isCategoryPackageName, 210
- isDomainConstructorForm, 344
- isDomainForm, 344
- isDomainSubst, 196
- isFunctor, 249
- isListConstructor, 107
- isMacro, 282
- isSuperDomain, 251
- isTokenDelimiter, 457
- isUnionMode, 317
- killColons, 397
- line-advance-char, 635
- line-at-end-p, 634
- line-current-segment, 636
- line-new-line, 636
- line-next-char, 635
- line-past-end-p, 635
- line-print, 634
- lispize, 347
- lisplibDoRename, 201
- lisplibWrite, 209
- loadIfNecessary, 114
- loadLibIfNecessary, 115
- macroExpand, 174
- macroExpandInPlace, 174
- macroExpandList, 175
- make-symbol-of, 460
- makeCategoryForm, 293
- makeCategoryPredicates, 175
- makeFunctorArgumentParameters, 217
- makeSimplePredicateOrNil, 546
- match-advance-string, 455
- match-current-token, 459
- match-next-token, 460
- match-string, 454
- match-token, 460
- maxSuperType, 344
- mergeModemap, 263
- mergeSignatureAndLocalVarAlists, 209
- meta-syntax-error, 465
- mkAbbrev, 298

- mkAlistOfExplicitCategoryOps, [189](#)
- mkCategoryPackage, [176](#)
- mkConstructor, [201](#)
- mkDatabasePred, [214](#)
- mkEvalableCategoryForm, [178](#)
- mkExplicitCategoryFunction, [288](#)
- mkList, [310](#)
- mkNewModemapList, [262](#)
- mkOpVec, [221](#)
- mkRepetitionAssoc, [180](#)
- mkUnion, [362](#)
- modeEqual, [362](#)
- modeEqualSubst, [363](#)
- modemapPattern, [199](#)
- modifyModeStack, [625](#)
- moveORsOutside, [197](#)
- mustInstantiate, [289](#)
- ncINTERPFILE, [627](#)
- new2OldDefForm, [74](#)
- new2OldLisp, [72](#)
- new2OldTran, [72](#)
- newConstruct, [74](#)
- newDef2Def, [74](#)
- newIf2Cond, [73](#)
- newString2Words, [527](#)
- newWordFrom, [540](#)
- next-char, [463](#)
- next-line, [636](#)
- next-tab-loc, [555](#)
- next-token, [462](#)
- nonblankloc, [555](#)
- NRTassocIndex, [342](#)
- NRTgetLocalIndex, [211](#)
- NRTgetLookupFunction, [210](#)
- NRTputInHead, [186](#)
- NRTputInTail, [185](#)
- opt-, [238](#)
- optCall, [229](#)
- optCallEval, [233](#)
- optCallSpecially, [231](#)
- optCatch, [240](#)
- optCond, [242](#)
- optCONDtail, [226](#)
- optEQ, [236](#)
- optIF2COND, [227](#)
- optimize, [225](#)
- optimizeFunctionDef, [223](#)
- optional, [467](#)
- optLESSP, [238](#)
- optMINUS, [236](#)
- optMkRecord, [245](#)
- optPackageCall, [230](#)
- optPredicateIfTrue, [227](#)
- optQSMINUS, [237](#)
- optRECORDCOPY, [247](#)
- optRECORDELT, [245](#)
- optSEQ, [234](#)
- optSETRECORDELT, [246](#)
- optSPADCALL, [239](#)
- optSpecialCall, [232](#)
- optSuchthat, [240](#)
- optXLAMCond, [226](#)
- orderByDependency, [222](#)
- orderPredicateItems, [193](#)
- orderPredTran, [194](#)
- outputComp, [343](#)
- PARSE-AnyId, [444](#)
- PARSE-Application, [432](#)
- parse-argument-designator, [548](#)
- PARSE-Category, [423](#)
- PARSE-Command, [418](#)
- PARSE-CommandTail, [421](#)
- PARSE-Conditional, [451](#)
- PARSE-Data, [442](#)
- PARSE-ElseClause, [451](#)
- PARSE-Enclosure, [438](#)
- PARSE-Exit, [449](#)
- PARSE-Expr, [426](#)
- PARSE-Expression, [425](#)
- PARSE-Float, [435](#)
- PARSE-FloatBase, [436](#)
- PARSE-FloatBasePart, [436](#)
- PARSE-FloatExponent, [437](#)
- PARSE-FloatTok, [453](#)
- PARSE-Form, [431](#)
- PARSE-FormalParameter, [439](#)
- PARSE-FormalParameterTok, [439](#)
- PARSE-getSemanticForm, [428](#)
- PARSE-GlyphTok, [444](#)
- parse-identifier, [547](#)
- PARSE-Import, [425](#)
- PARSE-Infix, [429](#)

- PARSE-InfixWith, [423](#)
- PARSE-IntegerTok, [438](#)
- PARSE-Iterator, [447](#)
- PARSE-IteratorTail, [447](#)
- parse-keyword, [548](#)
- PARSE-Label, [433](#)
- PARSE-LabelExpr, [452](#)
- PARSE-Leave, [450](#)
- PARSE-LedPart, [426](#)
- PARSE-leftBindingPowerOf, [427](#)
- PARSE-Loop, [452](#)
- PARSE-Name, [441](#)
- PARSE-NBGlyphTok, [443](#)
- PARSE-NewExpr, [417](#)
- PARSE-NudPart, [426](#)
- parse-number, [547](#)
- PARSE-OpenBrace, [446](#)
- PARSE-OpenBracket, [446](#)
- PARSE-Operation, [427](#)
- PARSE-Option, [422](#)
- PARSE-Prefix, [428](#)
- PARSE-Primary, [434](#)
- PARSE-Primary1, [434](#)
- PARSE-PrimaryNoFloat, [434](#)
- PARSE-PrimaryOrQM, [421](#)
- PARSE-Quad, [439](#)
- PARSE-Qualification, [430](#)
- PARSE-Reduction, [431](#)
- PARSE-ReductionOp, [431](#)
- PARSE-Return, [449](#)
- PARSE-rightBindingPowerOf, [428](#)
- PARSE-ScriptItem, [441](#)
- PARSE-Scripts, [440](#)
- PARSE-Seg, [450](#)
- PARSE-Selector, [433](#)
- PARSE-SemiColon, [449](#)
- PARSE-Sequence, [445](#)
- PARSE-Sequence1, [445](#)
- PARSE-Sexpr, [442](#)
- PARSE-Sexpr1, [442](#)
- parse-spadstring, [546](#)
- PARSE-SpecialCommand, [419](#)
- PARSE-SpecialKeyWord, [418](#)
- PARSE-Statement, [422](#)
- PARSE-String, [439](#)
- parse-string, [546](#)
- PARSE-Suffix, [448](#)
- PARSE-TokenCommandTail, [419](#)
- PARSE-TokenList, [420](#)
- PARSE-TokenOption, [420](#)
- PARSE-TokTail, [430](#)
- PARSE-VarForm, [440](#)
- PARSE-With, [423](#)
- parseAnd, [101](#)
- parseAtom, [98](#)
- parseAtSign, [102](#)
- parseCategory, [103](#)
- parseCoerce, [104](#)
- parseColon, [104](#)
- parseConstruct, [99](#)
- parseDEF, [105](#)
- parseDollarGreaterEqual, [108](#)
- parseDollarGreaterThan, [108](#)
- parseDollarLessEqual, [109](#)
- parseDollarNotEqual, [109](#)
- parseDropAssertions, [103](#)
- parseEquivalence, [110](#)
- parseExit, [110](#)
- parseGreaterEqual, [111](#)
- parseGreaterThan, [112](#)
- parseHas, [112](#)
- parseHasRhs, [114](#)
- parseIf, [118](#)
- parseIf,ifTran, [118](#)
- parseImplies, [120](#)
- parseIn, [121](#)
- parseInBy, [122](#)
- parseIs, [123](#)
- parseIsnt, [123](#)
- parseJoin, [124](#)
- parseLeave, [125](#)
- parseLessEqual, [125](#)
- parseLET, [126](#)
- parseLETD, [127](#)
- parseLhs, [106](#)
- parseMDEF, [127](#)
- parseNot, [128](#)
- parseNotEqual, [129](#)
- parseOr, [129](#)
- parsepiles, [87](#)
- parsePretend, [130](#)
- parseprint, [556](#)

- parseReturn, 131
- parseSegment, 131
- parseSeq, 132
- parseTran, 97
- parseTranCheckForRecord, 545
- parseTranList, 99
- parseTransform, 97
- parseType, 102
- parseVCONS, 133
- parseWhere, 133
- Pop-Reduction, 552
- postAdd, 372
- postAtom, 367
- postAtSign, 374
- postBigFloat, 376
- postBlock, 376
- postBlockItem, 373
- postBlockItemList, 373
- postCapsule, 373
- postCategory, 377
- postcheck, 369
- postCollect, 379
- postCollect,finish, 377
- postColon, 381
- postColonColon, 381
- postComma, 382
- postConstruct, 383
- postDef, 384
- postDefArgs, 386
- postError, 370
- postExit, 387
- postFlatten, 382
- postFlattenLeft, 395
- postForm, 370
- postIf, 387
- postIn, 389
- postin, 388
- postInSeq, 388
- postIteratorList, 380
- postJoin, 390
- postMakeCons, 378
- postMapping, 390
- postMDef, 391
- postOp, 367
- postPretend, 392
- postQUOTE, 393
- postReduce, 393
- postRepeat, 394
- postScripts, 394
- postScriptsForm, 368
- postSemiColon, 395
- postSignature, 396
- postSlash, 397
- postTran, 366
- postTranList, 368
- postTranScripts, 368
- postTranSegment, 384
- postTransform, 365
- postTransformCheck, 369
- postTuple, 398
- postTupleCollect, 398
- postType, 375
- postWhere, 399
- postWith, 400
- preparse, 80
- preparse-echo, 92
- preparse1, 84
- preparseReadLine, 89
- preparseReadLine1, 90
- primitiveType, 600
- print-defun, 587
- print-package, 549
- processFunctor, 275
- purgeNewConstructorLines, 481
- push-reduction, 468
- putDomainsInScope, 250
- putInLocalDomainReferences, 185
- quote-if-string, 456
- read-a-line, 631
- recompile-lib-file-if-necessary, 627
- recordAttributeDocumentation, 470
- recordDocumentation, 471
- recordHeaderDocumentation, 472
- recordSignatureDocumentation, 470
- removeBackslashes, 541
- removeSuperfluousMapping, 396
- replaceExitEtc, 333
- replaceVars, 192
- reportOnFunctorCompilation, 215
- resolve, 361
- rwriteLispForm, 200
- s-process, 576

- screenLocalLine, 486
- setDefOp, 400
- seteltModemapFilter, 609
- setqMultiple, 337
- setqMultipleExplicit, 339
- setqSetelt, 340
- setqSingle, 340
- signatureTran, 193
- skip-blanks, 454
- skip-ifblock, 89
- skip-to-endif, 556
- spad, 573
- spad-fixed-arg, 627
- spadCompileOrSetq, 182
- splitEncodedFunctionName, 180
- stack-clear, 93
- stack-load, 92
- stack-pop, 94
- stack-push, 93
- storeblanks, 637
- string2BootTree, 71
- stripOffArgumentConditions, 302
- stripOffSubdomainConditions, 301
- subname, 228
- substituteCategoryArguments, 253
- substituteIntoFunctorModemap, 614
- substNames, 266
- substVars, 198
- token-install, 96
- token-lookahead-type, 455
- token-print, 96
- transDoc, 496
- transDocList, 495
- transformAndRecheckComments, 497
- transformOperationAlist, 206
- transImplementation, 599
- transIs, 106
- transIs1, 106
- translabel, 543
- translabel1, 543
- TruthP, 264
- try-get-token, 461
- tuple2List, 549
- uncons, 336
- underscore, 458
- unget-tokens, 458
- unknownTypeError, 249
- unloadOneConstructor, 201
- unTuple, 409
- updateCategoryFrameForCategory, 117
- updateCategoryFrameForConstructor, 116
- whoOwns, 541
- wrapDomainSub, 290
- writeLib1, 203
- defvar
 - \$BasicPredicates, 227
 - \$EmptyMode, 172
 - \$FormalMapVariableList, 266
 - \$NoValueMode, 172
 - \$byConstructors, 629
 - \$comblocklist, 553
 - \$constructorsSeen, 629
 - \$defstack, 407
 - \$echolinestack, 75
 - \$index, 75
 - \$is-eqlist, 408
 - \$is-gensymlist, 408
 - \$is-spill-list, 407
 - \$is-spill, 407
 - \$linelist, 75
 - \$newConlist, 557
 - \$preparse-last-line, 76
 - \$vl, 408
 - current-fragment, 631
 - current-line, 634
 - current-token, 95
 - definition-name, 417
 - initial-gensym, 408
 - lablasoc, 417
 - meta-error-handler, 464
 - next-token, 95
 - nonblank, 95
 - ParseMode, 417
 - prior-token, 94
 - reduce-stack, 468
 - tmptok, 416
 - tok, 416
 - valid-tokens, 95
 - XTokenReader, 463
- delete
 - calledby compDefWhereClause, 156
 - calledby getInverseEnvironment, 316

- calledby orderPredTran, 194
 - calledby putDomainsInScope, 250
- deleteFile
 - calledby buildLibdb, 478
- deleteFile[5]
 - called by buildLibdb, 478
 - called by extendLocalLibdb, 477
- deltaContour
 - calledby compWhere, 351
- digitp[5]
 - called by PARSE-FloatBasePart, 437
 - called by PARSE-FloatBase, 436
 - called by floatexpid, 465
- disallowNilAttribute, 214
 - calledby compDefineFunctor1, 144
 - defun, 214
- displayComp
 - calledby compOrCroak1, 589
- displayMissingFunctions, 216
 - calledby reportOnFunctorCompilation, 215
 - calls bright, 216
 - calls formatUnabbreviatedSig, 216
 - calls getmode, 216
 - calls member, 216
 - calls sayBrightly, 216
 - uses \$CheckVectorList, 216
 - uses \$env, 216
 - uses \$formalArgList, 216
 - defun, 216
- displayPreCompilationErrors, 544
 - calledby s-process, 583
 - calls length, 544
 - calls remdup, 544
 - calls sayBrightly, 544
 - calls sayMath, 544
 - local ref \$InteractiveMode, 544
 - local ref \$postStack, 544
 - local ref \$stopOp, 544
 - defun, 544
- displaySemanticErrors
 - calledby compOrCroak1, 589
 - calledby reportOnFunctorCompilation, 215
 - calledby s-process, 583
- displayWarnings
 - calledby reportOnFunctorCompilation, 215
- doIt, 276
 - calledby doIt, 277
 - calls NRTgetLocalIndex, 277
 - calls bright, 277
 - calls cannotDo, 277
 - calls compOrCroak, 277
 - calls compSingleCapsuleItem, 276
 - calls doItIf, 277
 - calls doIt, 277
 - calls formatUnabbreviated, 277
 - calls get, 277
 - calls insert, 277
 - calls isDomainForm, 276
 - calls isMacro, 277
 - calls kar, 277
 - calls lastnode, 276
 - calls member, 277
 - calls opOf, 277
 - calls put, 277
 - calls sayBrightly, 277
 - calls stackSemanticError, 277
 - calls stackWarning, 277
 - calls sublis, 277
 - local def \$LocalDomainAlist, 277
 - local def \$Representation, 277
 - local def \$e, 277
 - local def \$functorLocalParameters, 277
 - local def \$functorsUsed, 277
 - local def \$genno, 277
 - local def \$packagesUsed, 277
 - local ref \$EmptyMode, 277
 - local ref \$LocalDomainAlist, 277
 - local ref \$NRTopt, 277
 - local ref \$NonMentionableDomainNames, 277
 - local ref \$QuickCode, 277
 - local ref \$Representation, 277
 - local ref \$e, 277
 - local ref \$functorLocalParameters, 277
 - local ref \$functorsUsed, 277
 - local ref \$packagesUsed, 277
 - local ref \$predI, 277
 - local ref \$signatureOfForm, 277
 - defun, 276
- doIt
 - calledby compSingleCapsuleItem, 276
- doItIf, 281

- calledby doIt, 277
- calls compSingleCapsuleItem, 281
- calls comp, 281
- calls getSuccessEnvironment, 281
- calls localExtras, 281
- calls rplaca, 281
- calls rplacd, 281
- calls userError, 281
- local def \$e, 281
- local def \$functorLocalParameters, 281
- local ref \$Boolean, 281
- local ref \$e, 281
- local ref \$functorLocalParameters, 281
- local ref \$getDomainCode, 281
- local ref \$predl, 281
- defun, 281
- dollargreaterequal, 108
 - defplist, 108
- dollargreaterthan, 107
 - defplist, 107
- dollarnotequal, 109
 - defplist, 109
- dollarTran, 465
 - calledby PARSE-Qualification, 430
 - uses \$InteractiveMode, 465
 - defun, 465
- domainMember, 260
 - calledby addDomain, 248
 - calls modeEqual, 260
 - defun, 260
- doSystemCommand[5]
 - called by prepare1, 84
- downcase
 - calledby buildLibdbConEntry, 482
- drop, 553
 - calledby add-parens-and-semis-to-line, 88
 - calledby drop, 553
 - calls croak, 553
 - calls drop, 553
 - calls take, 553
 - defun, 553
- dsetq
 - calledby buildLibdb, 478
- Echo-Meta
 - usedby preparse-echo, 92
- echo-meta
 - usedby /rf-1, 573
 - usedby /rf, 571
 - usedby /rq, 571
 - usedby spad, 575
- echo-meta[5]
 - called by /RQ,LIB, 572
- editfile
 - usedby compCapsule, 274
- elapsedTime
 - calledby compile, 169
- elemn
 - calledby PARSE-Operation, 427
 - calledby PARSE-leftBindingPowerOf, 428
 - calledby PARSE-rightBindingPowerOf, 428
- elt, 306
 - defplist, 306
- eltForm
 - calledby compApplication, 605
- eltModemapFilter, 608
 - calledby getFormModemaps, 607
 - calls isConstantId, 608
 - calls stackMessage, 608
 - defun, 608
- embed
 - calledby compilerDoitWithScreenedLisplib, 569
- encodeFunctionName, 179
 - calledby compile, 169
 - calls encodeItem, 179
 - calls getAbbreviation, 179
 - calls internl, 179
 - calls length, 179
 - calls mkRepititionAssoc, 179
 - calls stringimage, 179
 - local def \$lisplibSignatureAlist, 179
 - local ref \$lisplibSignatureAlist, 179
 - local ref \$lisplib, 179
 - defun, 179
- encodeItem, 181
 - calledby applyMapping, 594
 - calledby compApplication, 605
 - calledby compile, 169
 - calledby encodeFunctionName, 179
 - calls getCaps, 181
 - calls identp, 181

- calls pname, [181](#)
- calls stringimage, [181](#)
- defun, [181](#)
- eofp
 - calledby dbReadLines, [481](#)
- eq, [235](#)
- defplist, [235](#)
- eqcar
 - calledby PARSE-OpenBrace, [446](#)
 - calledby PARSE-OpenBracket, [446](#)
 - calledby getToken, [458](#)
 - calledby hackforis1, [409](#)
- eqsubstlist
 - calledby compColon, [290](#)
 - calledby compTypeOf, [596](#)
 - calledby getSignatureFromMode, [299](#)
 - calledby isDomainConstructorForm, [345](#)
 - calledby substNames, [266](#)
 - calledby substituteIntoFunctorModemap, [614](#)
- EqualBarGensym, [244](#)
 - calledby AssocBarGensym, [222](#)
 - calledby optCond, [242](#)
 - calls gensymp, [244](#)
 - local def \$GensymAssoc, [244](#)
 - local ref \$GensymAssoc, [244](#)
 - defun, [244](#)
- eqv, [110](#)
 - defplist, [110](#)
- erase
 - calledby initializeLisplib, [202](#)
- errhuh, [409](#)
 - calls systemError, [409](#)
 - defun, [409](#)
- error
 - calledby compileSpad2Cmd, [566](#)
 - calledby processFunctor, [275](#)
- errorRef
 - calledby compSymbol, [601](#)
- errors
 - usedby initializeLisplib, [202](#)
- Escape-Character
 - usedby token-lookahead-type, [455](#)
- escape-keywords, [457](#)
 - calledby quote-if-string, [456](#)
 - local ref \$keywords, [457](#)
- defun, [457](#)
- escaped, [553](#)
 - calledby preparse1, [84](#)
 - defun, [553](#)
- eval
 - calledby coerceSubset, [353](#)
 - calledby compDefineCategory2, [160](#)
 - calledby compileCases, [167](#)
 - calledby evalAndRwriteLispForm, [200](#)
 - calledby getSlotFromCategoryForm, [206](#)
 - calledby optCallEval, [233](#)
- evalAndRwriteLispForm, [200](#)
 - calledby compDefineCategory2, [160](#)
 - calledby compDefineFunctor1, [145](#)
 - calledby compSubDomain1, [347](#)
 - calls eval, [200](#)
 - calls rwriteLispForm, [200](#)
 - defun, [200](#)
- evalAndSub, [265](#)
 - calledby augModemapsFromCategoryRep, [267](#)
 - calledby augModemapsFromCategory, [260](#)
 - calls contained, [265](#)
 - calls getOperationAlist, [265](#)
 - calls get, [265](#)
 - calls isCategory, [265](#)
 - calls put, [265](#)
 - calls substNames, [265](#)
 - local def \$lhsOfColon, [265](#)
 - defun, [265](#)
- exit, [307](#)
 - defplist, [307](#)
- expand-tabs, [91](#)
 - calledby preparseReadLine1, [90](#)
 - calls indent-pos, [91](#)
 - calls nonblankloc, [91](#)
 - defun, [91](#)
- extendLocalLibdb, [477](#)
 - calledby compileSpad2Cmd, [566](#)
 - calls buildLibdb, [477](#)
 - calls dbReadLines, [477](#)
 - calls dbWriteLines, [477](#)
 - calls deleteFile[5], [477](#)
 - calls msort, [477](#)
 - calls purgeNewConstructorLines, [477](#)
 - calls union, [477](#)

- local def \$newConstructorList, 477
 - local ref \$createLocalLibDb, 477
 - local ref \$newConstructorList, 477
 - defun, 477
- extendsCategoryForm
 - calledby coerceHard, 354
 - calledby compWithMappingModel, 617
- extractCodeAndConstructTriple, 622
 - calledby compWithMappingModel, 617
 - defun, 622
- FactoredForm
 - calledby optCallEval, 233
- File-Closed
 - usedby read-a-line, 631
- file-closed
 - usedby spad, 575
- filep
 - calledby compDefineLisplib, 163
- fillerSpaces
 - calledby checkTransformFirsts, 513
 - calledby compDefineLisplib, 163
- finalizeDocumentation, 492
 - calledby compileDocumentation, 165
 - calledby finalizeLisplib, 203
 - calls assocleft, 492
 - calls bright, 492
 - calls form2String, 492
 - calls formatOpSignature, 492
 - calls macroExpand, 492
 - calls remdup, 492
 - calls sayKeyedMsg, 492
 - calls sayMSG, 492
 - calls strconc, 492
 - calls stringimage, 492
 - calls sublislis, 492
 - calls transDocList, 492
 - local ref \$FormalMapVariableList, 493
 - local ref \$comblocklist, 493
 - local ref \$docList, 492
 - local ref \$e, 492
 - local ref \$lisplibForm, 492
 - local ref \$op, 492
 - defun, 492
- finalizeLisplib, 203
 - calledby compDefineLisplib, 163
- calls NRTgenInitialAttributeAlist, 203
 - calls finalizeDocumentation, 203
 - calls getConstructorOpsAndAtts, 203
 - calls lisplibWrite, 203
 - calls mergeSignatureAndLocalVarAlists, 203
 - calls namestring, 203
 - calls profileWrite, 203
 - calls removeZeroOne, 203
 - calls sayMSG, 203
 - local def \$NRTslot1PredicateList, 204
 - local def \$lisplibCategory, 204
 - local def \$pairlis, 204
 - local ref \$/editfile, 203
 - local ref \$FormalMapVariableList, 203
 - local ref \$libFile, 203
 - local ref \$lisplibAbbreviation, 204
 - local ref \$lisplibAncestors, 204
 - local ref \$lisplibAttributes, 203
 - local ref \$lisplibCategory, 203
 - local ref \$lisplibForm, 203
 - local ref \$lisplibKind, 203
 - local ref \$lisplibModemapAlist, 203
 - local ref \$lisplibModemap, 203
 - local ref \$lisplibParents, 204
 - local ref \$lisplibPredicates, 204
 - local ref \$lisplibSignatureAlist, 203
 - local ref \$lisplibSlot1, 204
 - local ref \$lisplibSuperDomain, 203
 - local ref \$lisplibVariableAlist, 203
 - local ref \$profileCompiler, 204
 - local ref \$spadLibFT, 204
 - defun, 203
- fincomblock, 553
 - calledby preparse1, 84
 - calls preparse-echo, 554
 - uses \$EchoLineStack, 554
 - uses \$comblocklist, 554
 - defun, 553
- findfile
 - calledby compiler, 563
- firstNonBlankPosition, 538
 - calledby checkAddIndented, 519
 - calledby checkExtract, 520
 - calledby checkGetArgs, 521
 - calledby checkGetMargin, 522

- calledby checkIndentedLines, 524
 - calls maxindex, 538
 - defun, 538
- fixUpPredicate, 192
 - calledby interactiveModemapForm, 191
 - calls length, 192
 - calls moveORsOutside, 192
 - calls orderPredicateItems, 192
 - defun, 192
- flattenSignatureList, 190
 - calledby flattenSignatureList, 190
 - calledby mkAListOfExplicitCategoryOps, 189
 - calls flattenSignatureList, 190
 - defun, 190
- floatexpid, 465
 - calledby PARSE-FloatExponent, 437
 - calls collect, 465
 - calls digitp[5], 465
 - calls identp[5], 465
 - calls maxindex, 465
 - calls pname[5], 465
 - calls spadreduce, 465
 - calls step, 465
 - defun, 465
- fnameMake[5]
 - called by compileSpadLispCmd, 568
- fnameReadable?[5]
 - called by compileSpadLispCmd, 568
- form2HtString
 - calledby buildLibdbConEntry, 482
 - calledby checkRecordHash, 510
- form2LispString
 - calledby buildLibAttr, 485
 - calledby buildLibOp, 484
- form2String
 - calledby NRTgetLookupFunction, 210
 - calledby finalizeDocumentation, 492
- formal2Pattern, 214
 - calledby augmentLisplibModemapsFrom-Functor, 212
 - calls pairList, 214
 - calls sublis, 214
 - local ref \$PatternVariableList, 214
 - defun, 214
- formatOpSignature
 - calledby finalizeDocumentation, 492
- formatUnabbreviated
 - calledby compDefineCapsuleFunction, 152
 - calledby compMacro, 323
 - calledby doIt, 277
- formatUnabbreviatedSig
 - calledby displayMissingFunctions, 216
- fp-output-stream
 - calledby is-console, 555
- freelist, 625
 - calledby compWithMappingModel, 618
 - calledby freelist, 625
 - calls assq[5], 625
 - calls freelist, 625
 - calls getmode, 625
 - calls identp[5], 625
 - calls unionq, 625
 - defun, 625
- function
 - calledby optSpecialCall, 232
- genDeltaEntry
 - calledby coerceByModemap, 359
 - calledby compApplyModemap, 255
 - calledby transImplementation, 599
- genDomainOps, 220
 - calledby genDomainView, 219
 - calls addModemap, 220
 - calls getOperationAlist, 220
 - calls mkDomainConstructor, 220
 - calls mkq, 220
 - calls substNames, 220
 - uses \$ConditionalOperators, 220
 - uses \$e, 220
 - uses \$getDomainCode, 220
 - defun, 220
- genDomainView, 219
 - calledby genDomainViewList, 219
 - calls augModemapsFromCategory, 219
 - calls genDomainOps, 219
 - calls member, 220
 - calls mkDomainConstructor, 220
 - uses \$e, 220
 - uses \$getDomainCode, 220
 - defun, 219
- genDomainViewList, 219

- calledby genDomainViewList, 219
- calls genDomainViewList, 219
- calls genDomainView, 219
- calls isCategoryForm, 219
- uses \$EmptyEnvironment, 219
- defun, 219
- genDomainViewList0, 219
 - calledby makeFunctorArgumentParameters, 217
 - calls getDomainViewList, 219
 - defun, 219
- genSomeVariable
 - calledby compColon, 291
 - calledby setqMultiple, 337
- gensymp
 - calledby EqualBarGensym, 244
- genvar
 - calledby hasAplExtension, 403
- genVariable
 - calledby setqMultipleExplicit, 339
 - calledby setqMultiple, 337
- get
 - calledby applyMapping, 593
 - calledby autoCoerceByModemap, 360
 - calledby coerceHard, 354
 - calledby coerceSubset, 353
 - calledby compAtom, 597
 - calledby compDefineCapsuleFunction, 151
 - calledby compFocompFormWithModemap, 613
 - calledby compMapCond", 257
 - calledby compNoStacking1, 591
 - calledby compSymbol, 601
 - calledby compTypeOf, 596
 - calledby compWithMappingModel, 617
 - calledby compileCases, 167
 - calledby compile, 169
 - calledby doIt, 277
 - calledby evalAndSub, 265
 - calledby getArgumentMode, 305
 - calledby getDomainsInScope, 250
 - calledby getFormModemaps, 607
 - calledby getInverseEnvironment, 316
 - calledby getModemapListFromDomain, 260
 - calledby getModemapList, 259
 - calledby getModemap, 254
 - calledby getSignature, 302
 - calledby getSuccessEnvironment, 315
 - calledby giveFormalParametersValues, 174
 - calledby hasFullSignature, 172
 - calledby hasType, 356
 - calledby isFunctor, 249
 - calledby isMacro, 282
 - calledby isSuperDomain, 251
 - calledby isUnionMode, 317
 - calledby maxSuperType, 344
 - calledby mkEvalableCategoryForm, 178
 - calledby optCallSpecially, 231
 - calledby outputComp, 343
 - calledby parseHasRhs, 114
 - calledby setqSingle, 340
- get-a-line, 638
 - calledby initialize-prepare, 76
 - calledby prepareReadLine1, 90
 - calls is-console, 638
 - calls mkprompt[5], 638
 - calls read-a-line, 638
 - defun, 638
- get-internal-run-time
 - calledby s-process, 583
- get-token, 463
 - calledby try-get-token, 461
 - calls XTokenReader, 463
 - uses XTokenReader, 463
 - defun, 463
- getAbbreviation, 298
 - calledby applyMapping, 594
 - calledby compApplication, 605
 - calledby compDefine1, 141
 - calledby encodeFunctionName, 179
 - calls assq, 298
 - calls constructor?, 298
 - calls mkAbbrev, 298
 - calls rplac, 298
 - local def \$abbreviationTable, 298
 - local ref \$abbreviationTable, 298
 - defun, 298
- getArgumentMode, 305
 - calledby checkAndDeclare, 304
 - calledby getArgumentModeOrMoan, 187
 - calledby hasSigInTargetCategory, 304
 - calls get, 305

- defun, 305
- getArgumentModeOrMoan, 187
 - calledby compDefineCapsuleFunction, 151
 - calledby compDefineCategory2, 159
 - calledby compDefineFunctor1, 144
 - calls getArgumentMode, 187
 - calls stackSemanticError, 187
 - defun, 187
- getCaps, 181
 - calledby encodeItem, 181
 - calls l-case, 181
 - calls maxindex, 181
 - calls strconc, 181
 - calls stringimage, 181
 - defun, 181
- getCategoryOpsAndAtts, 205
 - calledby getConstructorOpsAndAtts, 205
 - calls getSlotFromCategoryForm, 205
 - calls transformOperationAlist, 205
 - defun, 205
- getConstructorAbbreviation
 - calledby compDefineLisplib, 163
- getConstructorExports, 475
 - calledby buildLibdb, 478
 - defun, 475
- getConstructorOpsAndAtts, 205
 - calledby finalizeLisplib, 203
 - calls getCategoryOpsAndAtts, 205
 - calls getFunctorOpsAndAtts, 205
 - defun, 205
- getdatabase
 - calledby augModemapsFromDomain, 252
 - calledby buildLibdbConEntry, 482
 - calledby checkDocMessage, 519
 - calledby checkIsValidType, 532
 - calledby checkNumOfArgs, 534
 - calledby compDefineFunctor1, 145
 - calledby compDefineLisplib, 163
 - calledby compileConstructor1, 184
 - calledby getOperationAlist, 265
 - calledby isFunctor, 249
 - calledby loadLibIfNecessary, 115
 - calledby macroExpandList, 175
 - calledby mkCategoryPackage, 176
 - calledby mkEvaluableCategoryForm, 178
 - calledby parseHas, 112
 - calledby updateCategoryFrameForCategory, 117
 - calledby updateCategoryFrameForConstructor, 116
 - calledby whoOwns, 541
- getDeltaEntry
 - calledby compElt, 306
- getDomainsInScope, 250
 - calledby addDomain, 248
 - calledby augModemapsFromDomain, 252
 - calledby comp3, 592
 - calledby compColon, 290
 - calledby compConstruct, 295
 - calledby putDomainsInScope, 250
 - calls get, 250
 - local ref \$CapsuleDomainsInScope, 250
 - local ref \$insideCapsuleFunctionIfTrue, 250
 - defun, 250
- getDomainViewList
 - calledby genDomainViewList0, 219
- getExportCategory
 - calledby NRTgetLookupFunction, 210
- getFormModemaps, 607
 - calledby compForm1, 603
 - calledby getFormModemaps, 607
 - calls eltModemapFilter, 607
 - calls getFormModemaps, 607
 - calls get, 607
 - calls last, 607
 - calls length, 607
 - calls nreverse0, 607
 - calls stackMessage, 607
 - local ref \$insideCategoryPackageIfTrue, 607
 - defun, 607
- getFunctorOpsAndAtts, 208
 - calledby getConstructorOpsAndAtts, 205
 - calls getSlotFromFunctor, 208
 - calls transformOperationAlist, 208
 - defun, 208
- getIdentity
 - calledby compReduce1, 327
- getInverseEnvironment, 316
 - calledby compBoolean, 314
 - calls delete, 316

- calls getUnionMode, 316
- calls get, 316
- calls identp, 316
- calls isDomainForm, 316
- calls member, 316
- calls mkpf, 316
- calls put, 316
- local ref \$EmptyEnvironment, 316
- defun, 316
- getl
 - calledby PARSE-Operation, 427
 - calledby PARSE-ReductionOp, 431
 - calledby PARSE-leftBindingPowerOf, 427
 - calledby PARSE-rightBindingPowerOf, 428
 - calledby addConstructorModemaps, 254
 - calledby augModemapsFromDomain1, 252
 - calledby checkRecordHash, 510
 - calledby checkTransformFirsts, 513
 - calledby compCat, 285
 - calledby compExpression, 135
 - calledby loadLibIfNecessary, 115
 - calledby mustInstantiate, 289
 - calledby optSpecialCall, 232
 - calledby optimize, 225
 - calledby parseTran, 97
- getMatchingRightPren, 538
 - calledby checkGetArgs, 521
 - calledby checkTransformFirsts, 513
 - calls maxindex, 538
 - defun, 538
- getmode
 - calledby addDomain, 248
 - calledby augModemapsFromDomain1, 252
 - calledby coerceHard, 354
 - calledby comp3, 592
 - calledby compCoerce, 358
 - calledby compColon, 290
 - calledby compDefWhereClause, 155
 - calledby compDefineCapsuleFunction, 152
 - calledby compSymbol, 601
 - calledby compile, 169
 - calledby displayMissingFunctions, 216
 - calledby freelist, 625
 - calledby getSignatureFromMode, 299
 - calledby getSignature, 302
 - calledby getUnionMode, 317
 - calledby isUnionMode, 317
 - calledby setqSingle, 340
- getModemap, 254
 - calledby compDefineFunctor1, 144
 - calls compApplyModemap, 254
 - calls get, 254
 - calls sublis, 254
 - defun, 254
- getModemapList, 259
 - calledby autoCoerceByModemap, 360
 - calledby compCase1, 284
 - calledby getUniqueModemap, 259
 - calls getModemapListFromDomain, 259
 - calls get, 259
 - calls nreverse0, 259
 - defun, 259
- getModemapListFromDomain, 260
 - calledby compElt, 306
 - calledby getModemapList, 259
 - calls get, 260
 - defun, 260
- getmodeOrMapping
 - calledby augModemapsFromDomain1, 253
- getOperationAlist, 265
 - calledby evalAndSub, 265
 - calledby genDomainOps, 220
 - calls compMakeCategoryObject, 265
 - calls getdatabase, 265
 - calls isFunctor, 265
 - calls stackMessage, 265
 - calls systemError, 265
 - uses \$domainShell, 265
 - uses \$e, 265
 - uses \$functorForm, 265
 - uses \$insideFunctorIfTrue, 265
 - defun, 265
- getOperationAlistFromLisplib
 - calledby mkOpVec, 221
- getParentsFor
 - calledby compDefineCategory2, 160
 - calledby compDefineFunctor1, 145
- getPrincipalView
 - calledby mkOpVec, 221
- getProplist
 - calledby addModemap1, 270
 - calledby compDefineAddSignature, 143

- calledby getSuccessEnvironment, 314
- calledby loadLibIfNecessary, 115
- getProplist[5]
 - called by setqSingle, 340
- getScriptName, 404
 - calledby postScriptsForm, 368
 - calledby postScripts, 394
 - calls decodeScripts, 404
 - calls identp[5], 404
 - calls internl, 404
 - calls pname[5], 404
 - calls postError, 404
 - defun, 404
- getSignature, 302
 - calledby compDefineCapsuleFunction, 151
 - calls SourceLevelSubsume, 303
 - calls getmode, 302
 - calls get, 302
 - calls knownInfo, 302
 - calls length, 302
 - calls printSignature, 303
 - calls remdup, 302
 - calls say, 302
 - calls stackSemanticError, 303
 - local ref \$e, 303
 - defun, 302
- getSignatureFromMode, 299
 - calledby compDefine1, 141
 - calledby hasSigInTargetCategory, 304
 - calls eqsubstlist, 299
 - calls getmode, 299
 - calls length, 299
 - calls opOf, 299
 - calls stackAndThrow, 299
 - calls take, 299
 - local ref \$FormalMapVariableList, 300
 - defun, 299
- getSlotFromCategoryForm, 206
 - calledby getCategoryOpsAndAtts, 205
 - calls eval, 206
 - calls systemErrorHere, 206
 - calls take, 206
 - local ref \$FormalMapVariableList, 206
 - defun, 206
- getSlotFromFunctor, 208
 - calledby getFunctorOpsAndAtts, 208
- calls compMakeCategoryObject, 208
- calls systemErrorHere, 208
- local ref \$e, 208
- local ref \$isplibOperationAlist, 208
- defun, 208
- getSpecialCaseAssoc, 300
 - calledby compileCases, 167
 - local ref \$functorForm, 300
 - local ref \$functorSpecialCases, 300
 - defun, 300
- getSuccessEnvironment, 314
 - calledby compBoolean, 314
 - calledby doItIf, 281
 - calls addBinding, 315
 - calls comp, 315
 - calls consProplistOf, 315
 - calls getProplist, 314
 - calls get, 315
 - calls identp, 314
 - calls isDomainForm, 314
 - calls put, 314
 - calls removeEnv, 315
 - local ref \$EmptyEnvironment, 315
 - local ref \$EmptyMode, 315
 - defun, 314
- getTargetFromRhs, 173
 - calledby compDefine1, 141
 - calledby getTargetFromRhs, 173
 - calls compOrCroak, 173
 - calls getTargetFromRhs, 173
 - calls stackSemanticError, 173
 - defun, 173
- getTempPath
 - calledby dbWriteLines, 481
- getToken, 458
 - calledby PARSE-OpenBrace, 446
 - calledby PARSE-OpenBracket, 446
 - calls eqcar, 458
 - defun, 458
- getUnionMode, 317
 - calledby getInverseEnvironment, 316
 - calls getmode, 317
 - calls isUnionMode, 317
 - defun, 317
- getUniqueModemap, 259
 - calledby getUniqueSignature, 259

- calls getModemapList, 259
 - calls qslessp, 259
 - calls stackWarning, 259
 - defun, 259
- getUniqueSignature, 259
 - calls getUniqueModemap, 259
 - defun, 259
- giveFormalParametersValues, 174
 - calledby compDefine1, 141
 - calledby compDefineCapsuleFunction, 151
 - calledby compDefineCategory2, 159
 - calledby compDefineFunctor1, 144
 - calls get, 174
 - calls put, 174
 - defun, 174
- hackforis, 408
 - calls hackforis1, 408
 - defun, 408
- hackforis1, 409
 - calledby hackforis, 408
 - calls eqcar, 409
 - calls kar, 409
 - defun, 409
- has, 112, 308
 - defplist, 112, 308
- hasAplExtension, 403
 - calledby aplTran1, 401
 - calls aplTran1, 403
 - calls deepestExpression, 403
 - calls genvar, 403
 - calls nreverse0, 403
 - defun, 403
- hasFormalMapVariable, 623
 - calledby compWithMappingModel, 617
 - calls ScanOrPairVec[5], 623
 - local def \$formalMapVariables, 623
 - defun, 623
- hasFullSignature, 172
 - calledby compDefineAddSignature, 143
 - calls get, 172
 - defun, 172
- hasNoVowels, 539
 - calledby checkDecorate, 504
 - calls maxindex, 539
 - defun, 539
- hasSigInTargetCategory, 304
 - calledby compDefineCapsuleFunction, 151
 - calls bright, 305
 - calls compareMode2Arg, 305
 - calls getArgumentMode, 304
 - calls getSignatureFromMode, 304
 - calls length, 304
 - calls remdup, 304
 - calls stackWarning, 305
 - local ref \$domainShell, 305
 - defun, 304
- hasType, 356
 - calledby coerceExtraHard, 355
 - calls get, 356
 - defun, 356
- helpSpad2Cmd[5]
 - called by compiler, 563
- hget
 - calledby checkArguments, 501
 - calledby checkBeginEnd, 502
 - calledby checkRecordHash, 509
 - calledby checkSplitPunctuation, 537
- hput
 - calledby checkRecordHash, 509
- htcharPosition, 539
 - calledby checkTrimCommented, 526
 - calledby htcharPosition, 539
 - calls charPosition, 539
 - calls htcharPosition, 539
 - calls length, 539
 - local ref \$charBack, 540
 - defun, 539
- identp
 - calledby addDomain, 248
 - calledby compFocompFormWithModemap, 613
 - calledby compInternalFunction, 155
 - calledby constructMacro, 182
 - calledby encodeItem, 181
 - calledby getInverseEnvironment, 316
 - calledby getSuccessEnvironment, 314
 - calledby isFunctor, 249
 - calledby mkExplicitCategoryFunction, 288
 - calledby subname, 228
- identp[5]

- called by PARSE-FloatExponent, 437
- called by compSetq1, 336
- called by floatexpid, 465
- called by freelist, 625
- called by getScriptName, 404
- called by postTransform, 365
- called by setqSingle, 340
- if, 117, 310, 387
 - defplist, 117, 310, 387
- ifcar
 - calledby buildLibdb, 478
 - calledby checkBeginEnd, 502
 - calledby checkFixCommonProblem, 508
 - calledby checkTexht, 512
 - calledby dbWriteLines, 481
 - calledby preparse, 80
- ifcdr
 - calledby checkBeginEnd, 502
 - calledby checkFixCommonProblem, 508
 - calledby checkHTargs, 509
 - calledby checkRecordHash, 509
 - calledby checkTexht, 512
 - calledby recordAttributeDocumentation, 470
- implies, 120
 - defplist, 120
- import, 318
 - defplist, 318
- In, 389
 - defplist, 389
- in, 121, 388
 - defplist, 121, 388
- inby, 122
 - defplist, 122
- incExitLevel
 - calledby parseIf,ifTran, 118
- indent-pos, 554
 - calledby expand-tabs, 91
 - calledby preparse1, 84
 - calls next-tab-loc, 554
 - defun, 554
- infixtok, 555
 - calledby add-parens-and-semis-to-line, 88
 - calls string2id-n, 555
 - defun, 555
- init-boot/spad-reader[5]
 - called by spad, 574
- initial-gensym, 408
 - defvar, 408
- initial-substring, 637
 - calledby preparseReadLine, 89
 - calledby skip-ifblock, 89
 - calledby skip-to-endif, 556
 - defun, 637
- initial-substring-p, 456
 - calledby match-string, 454
 - calls string-not-greaterp, 456
 - defun, 456
- initialize-preparse, 76
 - calledby spad, 574
 - calls get-a-line, 76
 - uses \$echolinestack, 76
 - uses \$index, 76
 - uses \$linelist, 76
 - uses \$preparse-last-line, 76
 - defun, 76
- initializeLisplib, 202
 - calls LAM,FILEACTQ, 202
 - calls addoptions, 202
 - calls erase, 202
 - calls pathnameTypeId, 202
 - calls writeLib1, 202
 - local def \$libFile, 202
 - local def \$lisplibAbbreviation, 202
 - local def \$lisplibAncestors, 202
 - local def \$lisplibForm, 202
 - local def \$lisplibKind, 202
 - local def \$lisplibModemapAlist, 202
 - local def \$lisplibModemap, 202
 - local def \$lisplibOpAlist, 202
 - local def \$lisplibOperationAlist, 202
 - local def \$lisplibSignatureAlist, 202
 - local def \$lisplibSuperDomain, 202
 - local def \$lisplibVariableAlist, 202
 - local ref \$erase, 202
 - local ref \$libFile, 202
 - uses /editfile, 202
 - uses /major-version, 202
 - uses errors, 202
 - defun, 202
- insert
 - calledby comp2, 591

- calledby doIt, 277
- insertAlist
 - calledby transformOperationAlist, 207
- insertModemap, 263
 - calledby mkNewModemapList, 262
 - defun, 263
- insertWOC
 - calledby orderPredTran, 194
- Integer
 - calledby optCallEval, 233
- interactiveModemapForm, 191
 - calledby augLisplibModemapsFromCategory, 188
 - calledby augmentLisplibModemapsFromFunctor, 212
 - calls fixUpPredicate, 191
 - calls modemapPattern, 191
 - calls replaceVars, 191
 - calls substVars, 191
 - local ref \$FormalMapVariableList, 191
 - local ref \$PatternVariableList, 191
 - defun, 191
- intern
 - calledby checkRecordHash, 509
- internl
 - calledby encodeFunctionName, 179
 - calledby getScriptName, 404
 - calledby postForm, 370
 - calledby substituteCategoryArguments, 253
- intersection
 - calledby orderByDependency, 223
- intersectionEnvironment
 - calledby compIf, 311
 - calledby replaceExitEtc, 333
- intersectionq
 - calledby orderPredTran, 194
- ioclear
 - calledby spad, 574
- is, 123, 318
 - defplist, 123, 318
- is-console, 555
 - calledby get-a-line, 638
 - calledby preparse1, 84
 - calledby print-defun, 587
 - calls fp-output-stream, 555
 - uses *terminal-io*, 555
- defun, 555
- isAlmostSimple
 - calledby makeSimplePredicateOrNil, 546
- isCategory
 - calledby augModemapsFromCategoryRep, 267
 - calledby evalAndSub, 265
- isCategoryForm
 - calledby addDomain, 248
 - calledby applyMapping, 593
 - calledby augLisplibModemapsFromCategory, 188
 - calledby coerceHard, 354
 - calledby compApplication, 606
 - calledby compColon, 290
 - calledby compFocompFormWithModemap, 613
 - calledby compJoin, 319
 - calledby compMakeCategoryObject, 208
 - calledby genDomainViewList, 219
 - calledby isDomainConstructorForm, 344
 - calledby isDomainForm, 344
 - calledby makeCategoryForm, 293
 - calledby makeFunctorArgumentParameters, 217
 - calledby mkAlistOfExplicitCategoryOps, 189
 - calledby mkDatabasePred, 214
 - calledby signatureTran, 193
- isCategoryPackageName, 210
 - calledby compDefineFunctor1, 144, 145
 - calledby substNames, 266
 - calls char, 210
 - calls maxindex, 210
 - calls pname, 210
 - defun, 210
- isConstantId
 - calledby eltModemapFilter, 608
 - calledby seteltModemapFilter, 609
- isDomainConstructorForm, 344
 - calledby isDomainForm, 344
 - calls eqsubstlist, 345
 - calls isCategoryForm, 344
 - local ref \$FormalMapVariableList, 345
 - defun, 344
- isDomainForm, 344

- calledby comp2, 591
- calledby compColon, 290
- calledby compDefine1, 141
- calledby compElt, 306
- calledby compHasFormat, 309
- calledby doIt, 276
- calledby getInverseEnvironment, 316
- calledby getSuccessEnvironment, 314
- calledby setqSingle, 340
- calls isCategoryForm, 344
- calls isDomainConstructorForm, 344
- calls isFunctor, 344
- calls kar, 344
- local ref \$SpecialDomainNames, 344
- defun, 344
- isDomainInScope
 - calledby setqSingle, 340
- isDomainSubst, 196
 - calledby orderPredTran, 194
 - defun, 196
- isExposedConstructor
 - calledby buildLibdbConEntry, 482
- isFluid
 - calledby compSymbol, 600
- isFunction
 - calledby compSymbol, 601
- isFunctor, 249
 - calledby addDomain, 248
 - calledby comp2, 591
 - calledby compFocompFormWithModemap, 613
 - calledby compWithMappingMode1, 617
 - calledby getOperationAlist, 265
 - calledby isDomainForm, 344
 - calls constructor?, 249
 - calls getdatabase, 249
 - calls get, 249
 - calls identp, 249
 - calls opOf, 249
 - calls updateCategoryFrameForCategory, 249
 - calls updateCategoryFrameForConstructor, 249
 - local ref \$CategoryFrame, 249
 - local ref \$InteractiveMode, 249
 - defun, 249
- isListConstructor, 107
 - calledby transIs, 106
 - calls member, 107
 - defun, 107
- isLiteral
 - calledby addDomain, 248
- isMacro, 282
 - calledby compDefine1, 141
 - calledby doIt, 277
 - calls get, 282
 - defun, 282
- isnt, 123
 - defplist, 123
- isSimple
 - calledby compForm2, 610
 - calledby makeSimplePredicateOrNil, 546
- isSomeDomainVariable
 - calledby coerce, 352
- isSubset
 - calledby coerceByModemap, 359
 - calledby coerceSubset, 353
 - calledby isSuperDomain, 251
- isSuperDomain, 251
 - calledby mergeModemap, 263
 - calls get, 251
 - calls isSubset, 251
 - calls lassoc, 251
 - calls opOf, 251
 - defun, 251
- isTokenDelimiter, 457
 - calledby compAtom, 597
 - calledby PARSE-TokenList, 420
 - calls current-symbol, 457
 - defun, 457
- isUnionMode, 317
 - calledby coerceExtraHard, 355
 - calledby getUnionMode, 317
 - calls getmode, 317
 - calls get, 317
 - defun, 317
- Join, 124, 319, 389
 - defplist, 124, 319, 389
- JoinInner
 - calledby mkCategoryPackage, 176

- kar
 - calledby addEmptyCapsuleIfNecessary, 173
 - calledby augModemapsFromDomain1, 252
 - calledby augModemapsFromDomain, 252
 - calledby compile, 169
 - calledby doIt, 277
 - calledby hackforis1, 409
 - calledby isDomainForm, 344
 - calledby optCallSpecially, 231
- keyedSystemError
 - calledby NRTputInHead, 186
 - calledby coerce, 351
 - calledby compileTimeBindingOf, 233
 - calledby mkAlistOfExplicitCategoryOps, 189
 - calledby optRECORDELT, 245
 - calledby optSETRECORDELT, 246
 - calledby optSpecialCall, 232
 - calledby substituteIntoFunctorModemap, 614
 - calledby transformOperationAlist, 207
- killColons, 397
 - calledby killColons, 397
 - calledby postSignature, 396
 - calls killColons, 397
 - defun, 397
- knownInfo
 - calledby addModemap, 269
 - calledby compMapCond", 257
 - calledby getSignature, 302
- l-case
 - calledby getCaps, 181
- labasoc
 - usedby PARSE-Data, 442
- lablasoc, 417
 - defvar, 417
- LAM,EVALANDFILEACTQ
 - calledby spadCompileOrSetq, 182
- LAM,FILEACTQ
 - calledby initializeLisplib, 202
- lassoc
 - calledby NRTputInTail, 185
 - calledby addModemap1, 270
 - calledby augLisplibModemapsFromCategory, 188
 - calledby buildLibAttr, 485
 - calledby buildLibOp, 484
 - calledby buildLibdbConEntry, 482
 - calledby checkAddMacros, 528
 - calledby checkTransformFirsts, 513
 - calledby coerceSubset, 353
 - calledby compDefWhereClause, 156
 - calledby compDefineAddSignature, 143
 - calledby compOrCroak1,compactify, 626
 - calledby isSuperDomain, 251
 - calledby loadLibIfNecessary, 115
 - calledby mergeSignatureAndLocalVarAlists, 209
 - calledby optCallSpecially, 231
 - calledby translabel1, 543
- lassq
 - calledby transformOperationAlist, 207
- last
 - calledby compFocompFormWithModemap, 613
 - calledby getFormModemaps, 607
 - calledby parseSeq, 132
 - calledby setqMultipleExplicit, 339
- lastnode
 - calledby NRTputInHead, 186
 - calledby doIt, 276
- leave, 124, 322
 - defplist, 124, 322
- leftTrim
 - calledby checkTransformFirsts, 513
- length
 - calledby buildLibAttr, 485
 - calledby buildLibdbConEntry, 482
 - calledby checkBeginEnd, 502
 - calledby checkExtract, 520
 - calledby checkIsValidType, 532
 - calledby checkSplitBrace, 526
 - calledby checkTrimCommented, 526
 - calledby compApplication, 606
 - calledby compApplyModemap, 255
 - calledby compColon, 290
 - calledby compDefine1, 141
 - calledby compDefineCapsuleFunction, 151
 - calledby compElt, 306
 - calledby compForm1, 603
 - calledby compForm2, 610

- calledby compHasFormat, 309
- calledby compRepeatOrCollect, 329
- calledby displayPreCompilationErrors, 544
- calledby encodeFunctionName, 179
- calledby fixUpPredicate, 192
- calledby getFormModemaps, 607
- calledby getSignatureFromMode, 299
- calledby getSignature, 302
- calledby hasSigInTargetCategory, 304
- calledby htcharPosition, 539
- calledby mkOpVec, 221
- calledby modeEqualSubst, 363
- calledby optMkRecord, 245
- calledby postScriptsForm, 368
- calledby removeBackslashes, 541
- calledby setqMultiple, 337
- lessp, 238
 - defplist, 238
- let, 126, 335
 - defplist, 126, 335
- letd, 126
 - defplist, 126
- letError
 - calledby newDef2Def, 74
 - calledby newIf2Cond, 73
- libConstructorSig
 - calledby buildLibdbConEntry, 482
- libdbTrim
 - calledby buildLibOp, 484
 - calledby buildLibdbConEntry, 482
- line, 633
 - usedby match-string, 454
 - usedby spad, 575
 - defstruct, 633
- Line-Advance-Char
 - calledby Advance-Char, 637
- line-advance-char, 635
 - uses \$line, 635
 - defun, 635
- Line-At-End-P
 - calledby Advance-Char, 637
- line-at-end-p, 634
 - calledby next-char, 464
 - uses \$line, 634
 - defun, 634
- line-clear, 634
 - uses \$line, 634
 - defmacro, 634
- line-current-char
 - calledby match-advance-string, 455
- line-current-index
 - calledby match-advance-string, 455
- line-current-segment, 636
 - calledby unget-tokens, 458
 - defun, 636
- line-handler
 - usedby string2BootTree, 71
- Line-New-Line
 - calledby read-a-line, 631
- line-new-line, 636
 - calledby unget-tokens, 459
 - uses \$line, 636
 - defun, 636
- line-next-char, 635
 - calledby next-char, 464
 - uses \$line, 635
 - defun, 635
- line-number
 - calledby PARSE-Category, 424
 - calledby unget-tokens, 459
- line-past-end-p, 635
 - calledby match-advance-string, 455
 - calledby match-string, 454
 - uses \$line, 635
 - defun, 635
- line-print, 634
 - local ref \$out-stream, 634
 - uses \$line, 634, 636
 - defun, 634
- lispize, 347
 - calledby compSubDomain1, 347
 - calls optimize, 347
 - defun, 347
- lisplibDoRename, 201
 - calledby compDefineLisplib, 163
 - calls replaceFile, 201
 - local ref \$spadLibFT, 201
 - defun, 201
- lisplibWrite, 209
 - calledby compDefineCategory2, 160
 - calledby compDefineFunctor1, 145
 - calledby compileDocumentation, 165

- calledby finalizeLisplib, 203
 - calls rwrite128, 209
 - local ref \$lisplib, 209
 - defun, 209
- List
 - calledby optCallEval, 233
- ListCategory, 296
 - defplist, 296
- listOfIdentifiersIn
 - calledby compDefWhereClause, 156
- listOfPatternIds
 - calledby augmentLisplibModemapsFrom-Functor, 212
 - calledby orderPredTran, 194
- listOfSharpVars
 - calledby compFocompFormWithModemap, 613
- listOrVectorElementNode
 - calledby augModemapsFromDomain, 252
- loadIfNecessary, 114
 - calledby parseHasRhs, 114
 - calls loadLibIfNecessary, 114
 - defun, 114
- loadLib
 - calledby loadLibIfNecessary, 115
- loadLibIfNecessary, 115
 - calledby loadIfNecessary, 114
 - calledby loadLibIfNecessary, 115
 - calls canFuncall?, 115
 - calls getPropList, 115
 - calls getdatabase, 115
 - calls getl, 115
 - calls lassoc, 115
 - calls loadLibIfNecessary, 115
 - calls loadLib, 115
 - calls macrop, 115
 - calls throwKeyedMsg, 115
 - calls updateCategoryFrameForCategory, 115
 - calls updateCategoryFrameForConstructor, 115
 - local ref \$CategoryFrame, 115
 - local ref \$InteractiveMode, 115
 - defun, 115
- localdatabase
 - calledby compDefineLisplib, 163
- localdatabase[5]
 - called by compileSpadLispCmd, 568
- localExtras
 - calledby doItIf, 281
- lt
 - calledby PARSE-Operation, 427
- macroExpand, 174
 - calledby compDefine1, 141
 - calledby compMacro, 323
 - calledby compSeqItem, 334
 - calledby compWhere, 351
 - calledby finalizeDocumentation, 492
 - calledby macroExpandInPlace, 174
 - calledby macroExpandList, 175
 - calledby macroExpand, 174
 - calls macroExpandList, 175
 - calls macroExpand, 174
 - defun, 174
- macroExpandInPlace, 174
 - calledby compSingleCapsuleItem, 276
 - calls macroExpand, 174
 - defun, 174
- macroExpandList, 175
 - calledby macroExpand, 175
 - calls getdatabase, 175
 - calls macroExpand, 175
 - defun, 175
- macrop
 - calledby loadLibIfNecessary, 115
- make-float
 - calledby PARSE-Float, 435
- make-full-cvec
 - calledby preparse1, 84
- make-outstream
 - calledby dbWriteLines, 481
- make-outstream[5]
 - called by buildLibdb, 478
- make-reduction
 - calledby push-reduction, 468
- make-symbol-of, 460
 - calledby PARSE-Expression, 425
 - calledby current-symbol, 460
 - uses \$token, 460
 - defun, 460
- makeCategoryForm, 293

- calledby compColon, 290
- calls compOrCroak, 293
- calls isCategoryForm, 293
- local ref \$EmptyMode, 293
- defun, 293
- makeCategoryPredicates, 175
 - calledby compDefineCategory1, 158
 - uses \$FormalMapVariableList, 175
 - uses \$TriangleVariableList, 176
 - uses \$mvl, 176
 - uses \$stvl, 176
 - defun, 175
- makeFunctorArgumentParameters, 217
 - calledby compDefineFunctor1, 144
 - calls assq, 217
 - calls genDomainViewList0, 217
 - calls isCategoryForm, 217
 - calls union, 217
 - uses \$ConditionalOperators, 217
 - uses \$alternateViewList, 217
 - uses \$forceAdd, 217
 - defun, 217
- makeInitialModemapFrame[5]
 - called by spad, 574
- makeInputFilename
 - calledby compileDocumentation, 165
- makeInputFilename[5]
 - called by /rf-1, 572
- makeLiteral
 - calledby addEltModemap, 261
- makeNonAtomic
 - calledby parseHas, 112
- makeSimplePredicateOrNil, 546
 - calledby parseIf,ifTran, 118
 - calls isAlmostSimple, 546
 - calls isSimple, 546
 - calls wrapSEQExit, 546
 - defun, 546
- mapInto
 - calledby parseSeq, 132
 - calledby parseWhere, 133
- Mapping, 285
 - defplist, 285
- match-advance-string, 455
 - calledby PARSE-Category, 423
 - calledby PARSE-Command, 418
 - calledby PARSE-Conditional, 451
 - calledby PARSE-Enclosure, 438
 - calledby PARSE-Exit, 449
 - calledby PARSE-FloatBasePart, 436
 - calledby PARSE-FloatExponent, 437
 - calledby PARSE-Form, 431
 - calledby PARSE-Import, 425
 - calledby PARSE-IteratorTail, 447
 - calledby PARSE-Iterator, 447
 - calledby PARSE-Label, 433
 - calledby PARSE-Leave, 450
 - calledby PARSE-Loop, 452
 - calledby PARSE-Option, 422
 - calledby PARSE-Primary1, 435
 - calledby PARSE-PrimaryOrQM, 421
 - calledby PARSE-Quad, 439
 - calledby PARSE-Qualification, 430
 - calledby PARSE-Return, 449
 - calledby PARSE-ScriptItem, 441
 - calledby PARSE-Scripts, 440
 - calledby PARSE-Selector, 433
 - calledby PARSE-SemiColon, 449
 - calledby PARSE-Sequence, 445
 - calledby PARSE-Sexpr1, 442
 - calledby PARSE-SpecialCommand, 419
 - calledby PARSE-Statement, 422
 - calledby PARSE-TokenOption, 420
 - calledby PARSE-With, 423
 - calls current-token, 455
 - calls line-current-char, 455
 - calls line-current-index, 455
 - calls line-past-end-p, 455
 - calls match-string, 455
 - calls quote-if-string, 455
 - uses \$line, 455
 - uses \$token, 455
 - defun, 455
- match-current-token, 459
 - calledby PARSE-GlyphTok, 444
 - calledby PARSE-NBGlyphTok, 443
 - calledby PARSE-Operation, 427
 - calledby PARSE-SpecialKeyword, 418
 - calledby parse-argument-designator, 548
 - calledby parse-identifier, 547
 - calledby parse-keyword, 548
 - calledby parse-number, 547

- calledby parse-spadstring, 546
 - calledby parse-string, 546
 - calls current-token, 459
 - calls match-token, 459
 - defun, 459
- match-next-token, 460
 - calledby PARSE-ReductionOp, 431
 - calls match-token, 460
 - calls next-token, 460
 - defun, 460
- match-string, 454
 - calledby PARSE-AnyId, 444
 - calledby PARSE-NewExpr, 417
 - calledby PARSE-Primary1, 435
 - calledby match-advance-string, 455
 - calls current-char, 454
 - calls initial-substring-p, 454
 - calls line-past-end-p, 454
 - calls skip-blanks, 454
 - calls subseq, 454
 - calls unget-tokens, 454
 - uses \$line, 454
 - uses line, 454
 - defun, 454
- match-token, 460
 - calledby match-current-token, 459
 - calledby match-next-token, 460
 - calls token-symbol, 460
 - calls token-type, 460
 - defun, 460
- Matrix
 - calledby optCallEval, 233
- maxindex
 - calledby buildLibdbConEntry, 482
 - calledby checkAddBackSlashes, 527
 - calledby checkAddPeriod, 529
 - calledby checkAddSpaceSegments, 529
 - calledby checkGetArgs, 521
 - calledby checkIeEgfun, 531
 - calledby checkSplitBackslash, 535
 - calledby checkSplitOn, 536
 - calledby checkSplitPunctuation, 537
 - calledby checkTransformFirsts, 513
 - calledby compDefineFunctor1, 144
 - calledby firstNonBlankPosition, 538
 - calledby floatexpid, 465
 - calledby getCaps, 181
 - calledby getMatchingRightPren, 538
 - calledby hasNoVowels, 539
 - calledby isCategoryPackageName, 210
 - calledby prepare1, 84
 - calledby prepareReadLine1, 90
 - calledby translabel1, 543
- maxSuperType, 344
 - calledby coerceSubset, 353
 - calledby maxSuperType, 344
 - calledby setqSingle, 340
 - calls get, 344
 - calls maxSuperType, 344
 - defun, 344
- mbpip
 - calledby subname, 228
- mdef, 127, 323
 - defplist, 127, 323
- member
 - calledby addDomain, 248
 - calledby applyMapping, 593
 - calledby augLisplibModemapsFromCategory, 188
 - calledby augModemapsFromDomain, 252
 - calledby augmentLisplibModemapsFromFunctor, 212
 - calledby autoCoerceByModemap, 360
 - calledby checkBeginEnd, 502
 - calledby checkDecorateForHt, 506
 - calledby checkDecorate, 504
 - calledby checkFixCommonProblem, 508
 - calledby checkRecordHash, 509
 - calledby checkSkipOpToken, 525
 - calledby coerceExtraHard, 355
 - calledby compApplication, 606
 - calledby compApplyModemap, 255
 - calledby compDefineCapsuleFunction, 152
 - calledby compile, 169
 - calledby displayMissingFunctions, 216
 - calledby doIt, 277
 - calledby genDomainView, 220
 - calledby getInverseEnvironment, 316
 - calledby isListConstructor, 107
 - calledby mkNewModemapList, 262
 - calledby orderByDependency, 223
 - calledby orderPredTran, 194

- calledby parseHasRhs, 114
- calledby parseHas, 112
- calledby putDomainsInScope, 250
- calledby transformOperationAlist, 207
- member[5]
 - called by comp3, 592
 - called by compColon, 290
 - called by compSymbol, 601
 - called by compilerDoit, 570
- mergeModemap, 263
 - calledby mkNewModemapList, 262
 - calls TruthP, 263
 - calls isSuperDomain, 263
 - local ref \$forceAdd, 263
 - defun, 263
- mergePathnames[5]
 - called by compiler, 563
- mergeSignatureAndLocalVarAlists, 209
 - calledby finalizeLisplib, 203
 - calls lassoc, 209
 - defun, 209
- meta-error-handler, 464
 - calledby meta-syntax-error, 465
 - usedby meta-syntax-error, 465
 - defvar, 464
- meta-syntax-error, 465
 - calledby must, 466
 - calls meta-error-handler, 465
 - uses meta-error-handler, 465
 - defun, 465
- minus, 236
 - defplist, 236
- mkAbbrev, 298
 - calledby getAbbreviation, 298
 - calls addSuffix, 298
 - calls alistSize, 298
 - defun, 298
- mkAlistOfExplicitCategoryOps, 189
 - calledby augLisplibModemapsFromCategory, 188
 - calledby augmentLisplibModemapsFrom-Functor, 212
 - calledby mkAlistOfExplicitCategoryOps, 189
 - calls assocleft, 189
 - calls flattenSignatureList, 189
 - calls isCategoryForm, 189
 - calls keyedSystemError, 189
 - calls mkAlistOfExplicitCategoryOps, 189
 - calls nreverse0, 189
 - calls remdup, 189
 - calls union, 189
 - local ref \$e, 189
 - defun, 189
- mkAutoLoad
 - calledby unloadOneConstructor, 201
- mkCategoryPackage, 176
 - calledby compDefineCategory1, 158
 - calls JoinInner, 176
 - calls abbreviationsSpad2Cmd, 176
 - calls assoc, 176
 - calls getdatabase, 176
 - calls pname, 176
 - calls strconc, 176
 - calls sublislis, 176
 - uses \$FormalMapVariableList, 176
 - uses \$categoryPredicateList, 176
 - uses \$e, 176
 - uses \$options, 176
 - defun, 176
- mkConstructor, 201
 - calledby compDefineCategory2, 160
 - calledby mkConstructor, 201
 - calls mkConstructor, 201
 - defun, 201
- mkDatabasePred, 214
 - calledby augmentLisplibModemapsFrom-Functor, 212
 - calls isCategoryForm, 214
 - local ref \$e, 214
 - defun, 214
- mkDomainConstructor
 - calledby bootStrapError, 215
 - calledby compHasFormat, 309
 - calledby genDomainOps, 220
 - calledby genDomainView, 220
- mkErrorExpr
 - calledby compOrCroak1, 589
- mkEvalableCategoryForm, 178
 - calledby compMakeCategoryObject, 209
 - calledby mkEvalableCategoryForm, 178
 - calls compOrCroak, 178

- calls getdatabase, 178
- calls get, 178
- calls mkEvaluableCategoryForm, 178
- calls mkq, 178
- local def \$e, 178
- local ref \$CategoryFrame, 178
- local ref \$CategoryNames, 178
- local ref \$Category, 178
- local ref \$EmptyMode, 178
- local ref \$e, 178
- defun, 178
- mkExplicitCategoryFunction, 288
 - calledby compCategory, 286
 - calls identp, 288
 - calls mkq, 288
 - calls mustInstantiate, 288
 - calls remdup, 288
 - calls union, 288
 - calls wrapDomainSub, 288
 - defun, 288
- mkList, 310
 - calledby compHasFormat, 309
 - defun, 310
- mkNewModemapList, 262
 - calledby addModemap1, 270
 - calls assoc, 262
 - calls insertModemap, 262
 - calls member, 262
 - calls mergeModemap, 262
 - calls nreverse0, 262
 - local ref \$InteractiveMode, 262
 - local ref \$forceAdd, 262
 - defun, 262
- mkOpVec, 221
 - calls AssocBarGensym, 221
 - calls assoc, 221
 - calls assq, 221
 - calls getOperationAlistFromLisplib, 221
 - calls getPrincipalView, 221
 - calls length, 221
 - calls opOf, 221
 - calls sublis, 221
 - uses Undef, 221
 - uses \$FormalMapVariableList, 221
 - defun, 221
- mkpf
 - calledby augLisplibModemapsFromCategory, 188
 - calledby augmentLisplibModemapsFromFunctor, 212
 - calledby compCapsuleInner, 274
 - calledby compCategoryItem, 287
 - calledby compileCases, 167
 - calledby getInverseEnvironment, 316
 - calledby stripOffSubdomainConditions, 301
- mkprogn
 - calledby setqMultiple, 337
- mkprompt[5]
 - called by get-a-line, 638
- mkq
 - calledby addArgumentConditions, 300
 - calledby bootStrapError, 215
 - calledby compCoerce1, 359
 - calledby compDefineCapsuleFunction, 152
 - calledby compDefineCategory2, 160
 - calledby compDefineFunctor1, 145
 - calledby compSeq1, 332
 - calledby genDomainOps, 220
 - calledby mkEvaluableCategoryForm, 178
 - calledby mkExplicitCategoryFunction, 288
 - calledby optSpecialCall, 232
 - calledby spadCompileOrSetq, 182
- mkRecord, 245
 - defplist, 245
- mkRepfun
 - calledby mkRepititionAssoc, 180
- mkRepititionAssoc, 180
 - calledby encodeFunctionName, 179
 - calls mkRepfun, 180
 - defun, 180
- mkUnion, 362
 - calledby resolve, 361
 - calls union, 362
 - local ref \$Rep, 362
 - defun, 362
- moan
 - calledby compileTimeBindingOf, 233
 - calledby parseExit, 110
 - calledby parseReturn, 131
- modeEqual, 362
 - calledby autoCoerceByModemap, 360
 - calledby checkAndDeclare, 304

- calledby coerceByModemap, 359
- calledby coerceHard, 354
- calledby compAtomWithModemap, 598
- calledby compCase1, 284
- calledby compile, 169
- calledby domainMember, 260
- calledby modeEqualSubst, 363
- calledby resolve, 361
- defun, 362
- modeEqualSubst, 363
 - calledby coerceEasy, 352
 - calledby modeEqualSubst, 363
 - calls length, 363
 - calls modeEqualSubst, 363
 - calls modeEqual, 363
 - defun, 363
- modelsAggregateOf
 - calledby compAtom, 597
 - calledby compConstruct, 295
 - calledby compRepeatOrCollect, 329
- modemapPattern, 199
 - calledby interactiveModemapForm, 191
 - calls rassoc, 199
 - local ref \$PatternVariableList, 199
 - defun, 199
- modifyModeStack, 625
 - calledby compExit, 308
 - calledby compLeave, 323
 - calledby compReturn, 331
 - calls copy, 625
 - calls resolve, 625
 - calls say, 625
 - calls setelt, 625
 - uses \$exitModeStack, 625
 - uses \$reportExitModeStack, 625
 - defun, 625
- moveORsOutside, 197
 - calledby fixUpPredicate, 192
 - calledby moveORsOutside, 197
 - calls moveORsOutside, 197
 - defun, 197
- msort
 - calledby extendLocalLibdb, 477
- msubst
 - calledby buildLibOp, 484
 - calledby buildLibdbConEntry, 482
- must, 466
 - calledby PARSE-Category, 423
 - calledby PARSE-Command, 418
 - calledby PARSE-Conditional, 451
 - calledby PARSE-Enclosure, 438
 - calledby PARSE-Exit, 449
 - calledby PARSE-FloatBasePart, 437
 - calledby PARSE-FloatBase, 436
 - calledby PARSE-FloatExponent, 437
 - calledby PARSE-Float, 435
 - calledby PARSE-Form, 431
 - calledby PARSE-Import, 425
 - calledby PARSE-Infix, 429
 - calledby PARSE-Iterator, 447
 - calledby PARSE-LabelExpr, 452
 - calledby PARSE-Label, 433
 - calledby PARSE-Leave, 450
 - calledby PARSE-Loop, 452
 - calledby PARSE-NewExpr, 417
 - calledby PARSE-Option, 422
 - calledby PARSE-Prefix, 429
 - calledby PARSE-Primary1, 434
 - calledby PARSE-Qualification, 430
 - calledby PARSE-Reduction, 431
 - calledby PARSE-Return, 449
 - calledby PARSE-ScriptItem, 441
 - calledby PARSE-Scripts, 440
 - calledby PARSE-Selector, 433
 - calledby PARSE-SemiColon, 449
 - calledby PARSE-Sequence, 445
 - calledby PARSE-Sexpr1, 442
 - calledby PARSE-SpecialCommand, 419
 - calledby PARSE-Statement, 422
 - calledby PARSE-TokenOption, 420
 - calledby PARSE-With, 423
 - calls meta-syntax-error, 466
 - defmacro, 466
- mustInstantiate, 289
 - calledby mkExplicitCategoryFunction, 288
 - calls getl, 289
 - local ref \$DummyFunctorNames, 289
 - defun, 289
- namestring
 - calledby bootStrapError, 215
 - calledby finalizeLisplib, 203

- namestring[5]
 - called by compileSpad2Cmd, 566
 - called by compileSpadLispCmd, 568
 - called by compiler, 563
- ncINTERPFILE, 627
 - calledby /rf-1, 572
 - calls SpadInterpretStream[5], 627
 - uses \$EchoLines, 627
 - uses \$ReadingFile, 627
 - defun, 627
- ncParseFromString
 - calledby checkGetParse, 522
- new2OldDefForm, 74
 - calledby new2OldDefForm, 74
 - calledby newDef2Def, 74
 - calls new2OldDefForm, 74
 - calls new2OldTran, 74
 - defun, 74
- new2OldLisp, 72
 - calledby s-process, 583
 - calledby string2BootTree, 71
 - calls new2OldTran, 72
 - calls postTransform, 72
 - defun, 72
- new2OldTran, 72
 - calledby new2OldDefForm, 74
 - calledby new2OldLisp, 72
 - calledby new2OldTran, 72
 - calledby newDef2Def, 74
 - calledby newIf2Cond, 73
 - calls dcq, 72
 - calls new2OldTran, 72
 - calls newConstruct, 72
 - calls newDef2Def, 72
 - calls newIf2Cond, 72
 - local ref \$new2OldRenameAssoc, 72
 - defun, 72
- newConstruct, 74
 - calledby new2OldTran, 72
 - defun, 74
- newDef2Def, 74
 - calledby new2OldTran, 72
 - calls letError, 74
 - calls new2OldDefForm, 74
 - calls new2OldTran, 74
 - defun, 74
- newIf2Cond, 73
 - calledby new2OldTran, 72
 - calls letError, 73
 - calls new2OldTran, 73
 - defun, 73
- newString2Words, 527
 - calledby checkComments, 498
 - calledby checkRewrite, 499
 - calls newWordFrom, 527
 - calls nreverse0, 527
 - defun, 527
- newWordFrom, 540
 - calledby newString2Words, 527
 - local ref \$charBlank, 540
 - local ref \$charFauxNewline, 540
 - local ref \$stringFauxNewline, 540
 - defun, 540
- next-char, 463
 - calledby PARSE-FloatBase, 436
 - calls line-at-end-p, 464
 - calls line-next-char, 464
 - uses current-line, 464
 - defun, 463
- next-line, 636
 - calledby Advance-Char, 637
 - local ref \$in-stream, 636
 - local ref \$line-handler, 636
 - defun, 636
- next-tab-loc, 555
 - calledby indent-pos, 554
 - defun, 555
- next-token, 95, 462
 - calledby match-next-token, 460
 - calls current-token, 462
 - calls try-get-token, 462
 - usedby next-token, 462
 - uses \$token, 95
 - uses next-token, 462
 - uses valid-tokens, 462
 - defun, 462
 - defvar, 95
- nonblank, 95
 - defvar, 95
- nonblankloc, 555
 - calledby add-parens-and-semis-to-line, 88
 - calledby expand-tabs, 91

- calls blankp, 555
 - defun, 555
- normalizeStatAndStringify
 - calledby reportOnFunctorCompilation, 215
- not, 128
 - defplist, 128
- notequal, 129
 - defplist, 129
- nreverse
 - calledby checkAddMacros, 528
 - calledby checkBalance, 501
 - calledby checkIeEg, 523
 - calledby transDoc, 496
- nreverse0
 - calledby aplTran1, 401
 - calledby compAdd, 271
 - calledby compCase1, 284
 - calledby compColon, 290
 - calledby compExpressionList, 609
 - calledby compForm1, 603
 - calledby compForm2, 610
 - calledby compJoin, 319
 - calledby compReduce1, 326
 - calledby compSeq1, 332
 - calledby getFormModemaps, 607
 - calledby getModemapList, 259
 - calledby hasAplExtension, 403
 - calledby mkAlistOfExplicitCategoryOps, 189
 - calledby mkNewModemapList, 262
 - calledby newString2Words, 527
 - calledby outputComp, 343
 - calledby parseHas, 112
 - calledby postCategory, 377
 - calledby postDef, 385
 - calledby postIf, 387
 - calledby postMDef, 391
 - calledby setqMultiple, 337
 - calledby substNames, 266
 - calledby transIs1, 106
- NRTaddInner
 - calledby NRTgetLocalIndex, 211
- NRTassignCapsuleFunctionSlot
 - calledby compDefineCapsuleFunction, 152
- NRTassocIndex, 342
 - calledby NRTgetLocalIndex, 211
- calledby NRTputInHead, 186
 - calledby NRTputInTail, 185
 - calledby setqSingle, 340
 - local ref \$NRTaddForm, 342
 - local ref \$NRTbase, 342
 - local ref \$NRTdeltaLength, 342
 - local ref \$NRTdeltaList, 342
 - local ref \$found, 342
 - defun, 342
- NRTextendsCategory1
 - calledby NRTgetLookupFunction, 210
- NRTgenInitialAttributeAlist
 - calledby compDefineFunctor1, 144
 - calledby finalizeLisplib, 203
- NRTgetLocalIndex, 211
 - calledby compAdd, 271
 - calledby compDefineFunctor1, 144
 - calledby compSymbol, 601
 - calledby doIt, 277
 - calls NRTaddInner, 211
 - calls NRTassocIndex, 211
 - calls compOrCroak, 211
 - calls rplaca, 211
 - local def \$EmptyMode, 211
 - local def \$NRTbase, 211
 - local def \$e, 211
 - local ref \$NRTaddForm, 211
 - local ref \$NRTdeltaLength, 211
 - local ref \$NRTdeltaListComp, 211
 - local ref \$NRTdeltaList, 211
 - local ref \$formalArgList, 211
 - defun, 211
- NRTgetLookupFunction, 210
 - calledby compDefineFunctor1, 145
 - calls NRTextendsCategory1, 210
 - calls bright, 210
 - calls form2String, 210
 - calls getExportCategory, 210
 - calls sayBrightlyNT, 210
 - calls sayBrightly, 210
 - calls sublis, 210
 - local def \$why, 210
 - local ref \$pairlis, 210
 - local ref \$why, 210
 - defun, 210
- NRTmakeSlot1Info

- calledby compDefineFunctor1, 145
- NRTputInHead, 186
 - calledby NRTputInHead, 186
 - calledby NRTputInTail, 185
 - calls NRTassocIndex, 186
 - calls NRTputInHead, 186
 - calls NRTputInTail, 186
 - calls keyedSystemError, 186
 - calls lastnode, 186
 - local ref \$elt, 186
 - defun, 186
- NRTputInTail, 185
 - calledby NRTputInHead, 186
 - calledby putInLocalDomainReferences, 185
 - calls NRTassocIndex, 185
 - calls NRTputInHead, 185
 - calls lassoc, 185
 - calls rplaca, 185
 - local ref \$devaluateList, 185
 - local ref \$elt, 185
 - defun, 185
- nsbst
 - calledby substVars, 198
- nth-stack, 551
 - calledby PARSE-Category, 424
 - calledby PARSE-Sexpr1, 442
 - calls reduction-value, 551
 - calls stack-store, 551
 - defmacro, 551
- obey
 - calledby buildLibdb, 478
- object2String
 - calledby compileSpad2Cmd, 566
 - calledby compileSpadLispCmd, 568
- opFf
 - calledby parseDEF, 105
- opOf
 - calledby augModemapsFromDomain, 252
 - calledby checkNumOfArgs, 534
 - calledby checkRecordHash, 509
 - calledby coerceSubset, 353
 - calledby comp2, 591
 - calledby compColonInside, 596
 - calledby compDefineCategory2, 160
 - calledby compElt, 306
 - calledby compPretend, 325
 - calledby compilerDoit, 570
 - calledby doIt, 277
 - calledby getSignatureFromMode, 299
 - calledby isFunctor, 249
 - calledby isSuperDomain, 251
 - calledby mkOpVec, 221
 - calledby optCallSpecially, 231
 - calledby parseHas, 112
 - calledby parseLET, 126
 - calledby parseMDEF, 127
 - calledby recordAttributeDocumentation, 470
 - defun, 238
- opt-, 238
- defun, 238
- optCall, 229
 - calledby optSPADCALL, 239
 - calls optCallSpecially, 229
 - calls optPackageCall, 229
 - calls optimize, 229
 - calls rplac, 229
 - calls systemErrorHere, 229
 - local ref \$QuickCode, 229
 - local ref \$bootStrapMode, 229
 - defun, 229
- optCallEval, 233
 - calledby optSpecialCall, 232
 - calls FactoredForm, 233
 - calls Integer, 233
 - calls List, 233
 - calls Matrix, 233
 - calls PrimitivveArray, 233
 - calls Vector, 233
 - calls eval, 233
 - defun, 233
- optCallSpecially, 231
 - calledby optCall, 229
 - calls get, 231
 - calls kar, 231
 - calls lassoc, 231
 - calls opOf, 231
 - calls optSpecialCall, 231
 - local ref \$e, 231
 - local ref \$getDomainCode, 231
 - local ref \$optimizableConstructorNames, 231

- local ref \$specialCaseKeyList, 231
 - defun, 231
- optCatch, 240
 - calls optimize, 240
 - calls rplac, 240
 - local ref \$InteractiveMode, 240
 - defun, 240
- optCond, 242
 - calls EqualBarGensym, 242
 - calls TruthP, 242
 - calls rplacd, 242
 - calls rplac, 242
 - defun, 242
- optCONDtail, 226
 - calledby optCONDtail, 226
 - calledby optXLAMCond, 226
 - calls optCONDtail, 226
 - local ref \$true, 226
 - defun, 226
- optEQ, 236
 - defun, 236
- optFunctorBody
 - calledby compDefineCategory2, 160
- optIF2COND, 227
 - calledby optIF2COND, 227
 - calledby optimize, 225
 - calls optIF2COND, 227
 - local ref \$true, 227
 - defun, 227
- optimize, 225
 - calledby lispize, 347
 - calledby optCall, 229
 - calledby optCatch, 240
 - calledby optSpecialCall, 232
 - calledby optimizeFunctionDef, 224
 - calledby optimize, 225
 - calls getl, 225
 - calls optIF2COND, 225
 - calls optimize, 225
 - calls prettyprint, 225
 - calls rplac, 225
 - calls say, 225
 - calls subrname, 225
 - defun, 225
- optimizeFunctionDef, 223
 - calledby compWithMappingMode1, 618
 - calledby compile, 169
 - calls bright, 224
 - calls optimize, 224
 - calls pp, 224
 - calls rplac, 223
 - calls sayBrightlyI, 223
 - local ref \$reportOptimization, 224
 - defun, 223
- optional, 467
 - calledby PARSE-Application, 432
 - calledby PARSE-Category, 423
 - calledby PARSE-CommandTail, 421
 - calledby PARSE-Conditional, 451
 - calledby PARSE-Expr, 426
 - calledby PARSE-Form, 431
 - calledby PARSE-Import, 425
 - calledby PARSE-Infix, 429
 - calledby PARSE-IteratorTail, 447
 - calledby PARSE-Iterator, 447
 - calledby PARSE-Prefix, 429
 - calledby PARSE-Primary1, 434
 - calledby PARSE-PrimaryNoFloat, 434
 - calledby PARSE-ScriptItem, 441
 - calledby PARSE-Seg, 450
 - calledby PARSE-Sequence1, 445
 - calledby PARSE-Sexpr1, 442
 - calledby PARSE-SpecialCommand, 419
 - calledby PARSE-Statement, 422
 - calledby PARSE-Suffix, 448
 - calledby PARSE-TokenCommandTail, 420
 - calledby PARSE-VarForm, 440
 - defun, 467
- optionlist
 - usedby spad, 575
- optLESSP, 238
 - defun, 238
- optMINUS, 236
 - defun, 236
- optMkRecord, 245
 - calls length, 245
 - defun, 245
- optPackageCall, 230
 - calledby optCall, 229
 - calls rplaca, 230
 - calls rplacd, 230
 - defun, 230

- optPredicateIfTrue, 227
 - calledby optXLAMCond, 226
 - local ref \$BasicPredicates, 227
 - defun, 227
- optQSMINUS, 237
 - defun, 237
- optRECORDCOPY, 247
 - defun, 247
- optRECORDELT, 245
 - calls keyedSystemError, 245
 - defun, 245
- optSEQ, 234
 - defun, 234
- optSETRECORDELT, 246
 - calls keyedSystemError, 246
 - defun, 246
- optSPADCALL, 239
 - calls optCall, 239
 - local ref \$InteractiveMode, 239
 - defun, 239
- optSpecialCall, 232
 - calledby optCallSpecially, 231
 - calls compileTimeBindingOf, 232
 - calls function, 232
 - calls getl, 232
 - calls keyedSystemError, 232
 - calls mkq, 232
 - calls optCallEval, 232
 - calls optimize, 232
 - calls rplaca, 232
 - calls rplacw, 232
 - calls rplac, 232
 - local ref \$QuickCode, 232
 - local ref \$Undef, 232
 - defun, 232
- optSuchthat, 240
 - defun, 240
- optXLAMCond, 226
 - calledby optXLAMCond, 226
 - calls optCONDtail, 226
 - calls optPredicateIfTrue, 226
 - calls optXLAMCond, 226
 - calls rplac, 226
 - defun, 226
- or, 129
 - defplist, 129
- orderByDependency, 222
 - calledby compDefWhereClause, 156
 - calls intersection, 223
 - calls member, 223
 - calls remdup, 223
 - calls say, 222
 - calls userError, 223
 - defun, 222
- orderPredicateItems, 193
 - calledby fixUpPredicate, 192
 - calls orderPredTran, 193
 - calls signatureTran, 193
 - defun, 193
- orderPredTran, 194
 - calledby orderPredicateItems, 193
 - calls delete, 194
 - calls insertWOC, 194
 - calls intersectionq, 194
 - calls isDomainSubst, 194
 - calls listOfPatternIds, 194
 - calls member, 194
 - calls setdifference, 194
 - calls unionq, 194
 - defun, 194
- outerProduct
 - calledby compileCases, 167
- outputComp, 343
 - calledby compForm1, 603
 - calledby outputComp, 343
 - calledby setqSingle, 340
 - calls comp, 343
 - calls get, 343
 - calls nreverse0, 343
 - calls outputComp, 343
 - local ref \$Expression, 343
 - defun, 343
- pack
 - calledby quote-if-string, 456
- Pair
 - calledby compApply, 595
- pairList
 - calledby compDefWhereClause, 156
 - calledby formal2Pattern, 214
- PARSE-AnyId, 444
 - calledby PARSE-Sexpr1, 442

- calls action, [444](#)
- calls advance-token, [444](#)
- calls current-symbol, [444](#)
- calls match-string, [444](#)
- calls parse-identifier, [444](#)
- calls parse-keyword, [444](#)
- calls push-reduction, [444](#)
- defun, [444](#)
- PARSE-Application, [432](#)
 - calledby PARSE-Application, [432](#)
 - calledby PARSE-Category, [424](#)
 - calledby PARSE-Form, [432](#)
 - calls PARSE-Application, [432](#)
 - calls PARSE-Primary, [432](#)
 - calls PARSE-Selector, [432](#)
 - calls optional, [432](#)
 - calls pop-stack-1, [432](#)
 - calls pop-stack-2, [432](#)
 - calls push-reduction, [432](#)
 - calls star, [432](#)
 - defun, [432](#)
- parse-argument-designator, [548](#)
 - calledby PARSE-FormalParameterTok, [439](#)
 - calls advance-token, [548](#)
 - calls match-current-token, [548](#)
 - calls push-reduction, [548](#)
 - calls token-symbol, [548](#)
 - defun, [548](#)
- PARSE-Category, [423](#)
 - calledby PARSE-Category, [423](#)
 - calls PARSE-Application, [424](#)
 - calls PARSE-Category, [423](#)
 - calls PARSE-Expression, [423](#)
 - calls action, [424](#)
 - calls bang, [423](#)
 - calls line-number, [424](#)
 - calls match-advance-string, [423](#)
 - calls must, [423](#)
 - calls nth-stack, [424](#)
 - calls optional, [423](#)
 - calls pop-stack-1, [424](#)
 - calls pop-stack-2, [423](#)
 - calls pop-stack-3, [423](#)
 - calls push-reduction, [423](#)
 - calls recordAttributeDocumentation, [424](#)
 - calls recordSignatureDocumentation, [424](#)
 - calls star, [424](#)
 - uses current-line, [424](#)
 - defun, [423](#)
- PARSE-Command, [418](#)
 - calls PARSE-SpecialCommand, [418](#)
 - calls PARSE-SpecialKeyWord, [418](#)
 - calls match-advance-string, [418](#)
 - calls must, [418](#)
 - calls push-reduction, [418](#)
 - defun, [418](#)
- PARSE-CommandTail, [421](#)
 - calledby PARSE-CommandTail, [421](#)
 - calledby PARSE-SpecialCommand, [419](#)
 - calls PARSE-CommandTail, [421](#)
 - calls PARSE-Option, [421](#)
 - calls action, [421](#)
 - calls bang, [421](#)
 - calls optional, [421](#)
 - calls pop-stack-1, [421](#)
 - calls pop-stack-2, [421](#)
 - calls push-reduction, [421](#)
 - calls star, [421](#)
 - calls systemCommand[5], [421](#)
 - defun, [421](#)
- PARSE-Conditional, [451](#)
 - calledby PARSE-ElseClause, [451](#)
 - calls PARSE-ElseClause, [451](#)
 - calls PARSE-Expression, [451](#)
 - calls bang, [451](#)
 - calls match-advance-string, [451](#)
 - calls must, [451](#)
 - calls optional, [451](#)
 - calls pop-stack-1, [451](#)
 - calls pop-stack-2, [451](#)
 - calls pop-stack-3, [451](#)
 - calls push-reduction, [451](#)
 - defun, [451](#)
- PARSE-Data, [442](#)
 - calledby PARSE-Primary1, [435](#)
 - calls PARSE-Sexpr, [442](#)
 - calls action, [442](#)
 - calls pop-stack-1, [442](#)
 - calls push-reduction, [442](#)
 - calls translabel, [442](#)
 - uses labasoc, [442](#)
 - defun, [442](#)

- PARSE-ElseClause, [451](#)
 - calledby PARSE-Conditional, [451](#)
 - calls PARSE-Conditional, [451](#)
 - calls PARSE-Expression, [452](#)
 - calls current-symbol, [451](#)
 - defun, [451](#)
- PARSE-Enclosure, [438](#)
 - calledby PARSE-Primary1, [435](#)
 - calls PARSE-Expr, [438](#)
 - calls match-advance-string, [438](#)
 - calls must, [438](#)
 - calls pop-stack-1, [438](#)
 - calls push-reduction, [438](#)
 - defun, [438](#)
- PARSE-Exit, [449](#)
 - calls PARSE-Expression, [449](#)
 - calls match-advance-string, [449](#)
 - calls must, [449](#)
 - calls pop-stack-1, [450](#)
 - calls push-reduction, [449](#)
 - defun, [449](#)
- PARSE-Expr, [426](#)
 - calledby PARSE-Enclosure, [438](#)
 - calledby PARSE-Expression, [425](#)
 - calledby PARSE-Import, [425](#)
 - calledby PARSE-Iterator, [447](#)
 - calledby PARSE-LabelExpr, [452](#)
 - calledby PARSE-Loop, [452](#)
 - calledby PARSE-Primary1, [435](#)
 - calledby PARSE-Reduction, [431](#)
 - calledby PARSE-ScriptItem, [441](#)
 - calledby PARSE-SemiColon, [449](#)
 - calledby PARSE-Statement, [422](#)
 - calls PARSE-LedPart, [426](#)
 - calls PARSE-NudPart, [426](#)
 - calls optional, [426](#)
 - calls pop-stack-1, [426](#)
 - calls push-reduction, [426](#)
 - calls star, [426](#)
 - defun, [426](#)
- PARSE-Expression, [425](#)
 - calledby PARSE-Category, [423](#)
 - calledby PARSE-Conditional, [451](#)
 - calledby PARSE-ElseClause, [452](#)
 - calledby PARSE-Exit, [449](#)
 - calledby PARSE-Infix, [429](#)
 - calledby PARSE-Iterator, [447](#)
 - calledby PARSE-Leave, [450](#)
 - calledby PARSE-Prefix, [429](#)
 - calledby PARSE-Return, [449](#)
 - calledby PARSE-Seg, [450](#)
 - calledby PARSE-Sequence1, [445](#)
 - calledby PARSE-SpecialCommand, [419](#)
 - calls PARSE-Expr, [425](#)
 - calls PARSE-rightBindingPowerOf, [425](#)
 - calls make-symbol-of, [425](#)
 - calls pop-stack-1, [425](#)
 - calls push-reduction, [425](#)
 - uses ParseMode, [425](#)
 - uses prior-token, [425](#)
 - defun, [425](#)
- PARSE-Float, [435](#)
 - calledby PARSE-Primary, [434](#)
 - calledby PARSE-Selector, [433](#)
 - calls PARSE-FloatBase, [435](#)
 - calls PARSE-FloatExponent, [435](#)
 - calls make-float, [435](#)
 - calls must, [435](#)
 - calls pop-stack-1, [435](#)
 - calls pop-stack-2, [435](#)
 - calls pop-stack-3, [435](#)
 - calls pop-stack-4, [435](#)
 - calls push-reduction, [435](#)
 - defun, [435](#)
- PARSE-FloatBase, [436](#)
 - calledby PARSE-Float, [435](#)
 - calls PARSE-FloatBasePart, [436](#)
 - calls PARSE-IntegerTok, [436](#)
 - calls char-eq, [436](#)
 - calls char-ne, [436](#)
 - calls current-char, [436](#)
 - calls current-symbol, [436](#)
 - calls digitp[5], [436](#)
 - calls must, [436](#)
 - calls next-char, [436](#)
 - calls push-reduction, [436](#)
 - defun, [436](#)
- PARSE-FloatBasePart, [436](#)
 - calledby PARSE-FloatBase, [436](#)
 - calls PARSE-IntegerTok, [437](#)
 - calls current-char, [437](#)
 - calls current-token, [437](#)

- calls digitp[5], [437](#)
- calls match-advance-string, [436](#)
- calls must, [437](#)
- calls push-reduction, [437](#)
- calls token-nonblank, [437](#)
- defun, [436](#)
- PARSE-FloatExponent, [437](#)
 - calledby PARSE-Float, [435](#)
 - calls PARSE-IntegerTok, [437](#)
 - calls action, [437](#)
 - calls advance-token, [437](#)
 - calls current-char, [437](#)
 - calls current-symbol, [437](#)
 - calls floatexpid, [437](#)
 - calls identp[5], [437](#)
 - calls match-advance-string, [437](#)
 - calls must, [437](#)
 - calls push-reduction, [437](#)
 - defun, [437](#)
- PARSE-FloatTok, [453](#)
 - calls bfp-, [453](#)
 - calls parse-number, [453](#)
 - calls pop-stack-1, [453](#)
 - calls push-reduction, [453](#)
 - local ref \$boot, [453](#)
 - defun, [453](#)
- PARSE-Form, [431](#)
 - calledby PARSE-NudPart, [426](#)
 - calls PARSE-Application, [432](#)
 - calls bang, [431](#)
 - calls match-advance-string, [431](#)
 - calls must, [431](#)
 - calls optional, [431](#)
 - calls pop-stack-1, [432](#)
 - calls push-reduction, [431](#)
 - defun, [431](#)
- PARSE-FormalParameter, [439](#)
 - calledby PARSE-Primary1, [435](#)
 - calls PARSE-FormalParameterTok, [439](#)
 - defun, [439](#)
- PARSE-FormalParameterTok, [439](#)
 - calledby PARSE-FormalParameter, [439](#)
 - calls parse-argument-designator, [439](#)
 - defun, [439](#)
- PARSE-getSemanticForm, [428](#)
 - calledby PARSE-Operation, [427](#)
 - calls PARSE-Infix, [428](#)
 - calls PARSE-Prefix, [428](#)
 - defun, [428](#)
- PARSE-GlyphTok, [444](#)
 - calledby PARSE-Quad, [439](#)
 - calledby PARSE-Seg, [450](#)
 - calledby PARSE-Sexpr1, [443](#)
 - calls action, [444](#)
 - calls advance-token, [444](#)
 - calls match-current-token, [444](#)
 - uses tok, [444](#)
 - defun, [444](#)
- parse-identifier, [547](#)
 - calledby PARSE-AnyId, [444](#)
 - calledby PARSE-Name, [441](#)
 - calls advance-token, [547](#)
 - calls match-current-token, [547](#)
 - calls push-reduction, [547](#)
 - calls token-symbol, [547](#)
 - defun, [547](#)
- PARSE-Import, [425](#)
 - calls PARSE-Expr, [425](#)
 - calls bang, [425](#)
 - calls match-advance-string, [425](#)
 - calls must, [425](#)
 - calls optional, [425](#)
 - calls pop-stack-1, [425](#)
 - calls pop-stack-2, [425](#)
 - calls push-reduction, [425](#)
 - calls star, [425](#)
 - defun, [425](#)
- PARSE-Infix, [429](#)
 - calledby PARSE-getSemanticForm, [428](#)
 - calls PARSE-Expression, [429](#)
 - calls PARSE-TokTail, [429](#)
 - calls action, [429](#)
 - calls advance-token, [429](#)
 - calls current-symbol, [429](#)
 - calls must, [429](#)
 - calls optional, [429](#)
 - calls pop-stack-1, [429](#)
 - calls pop-stack-2, [429](#)
 - calls push-reduction, [429](#)
 - defun, [429](#)
- PARSE-InfixWith, [423](#)
 - calls PARSE-With, [423](#)

- calls pop-stack-1, [423](#)
- calls pop-stack-2, [423](#)
- calls push-reduction, [423](#)
- defun, [423](#)
- PARSE-IntegerTok, [438](#)
 - calledby PARSE-FloatBasePart, [437](#)
 - calledby PARSE-FloatBase, [436](#)
 - calledby PARSE-FloatExponent, [437](#)
 - calledby PARSE-Primary1, [435](#)
 - calledby PARSE-Sexpr1, [442](#)
 - calls parse-number, [438](#)
 - defun, [438](#)
- PARSE-Iterator, [447](#)
 - calledby PARSE-IteratorTail, [447](#)
 - calledby PARSE-Loop, [452](#)
 - calls PARSE-Expression, [447](#)
 - calls PARSE-Expr, [447](#)
 - calls PARSE-Primary, [447](#)
 - calls match-advance-string, [447](#)
 - calls must, [447](#)
 - calls optional, [447](#)
 - calls pop-stack-1, [447](#)
 - calls pop-stack-2, [447](#)
 - calls pop-stack-3, [447](#)
 - defun, [447](#)
- PARSE-IteratorTail, [447](#)
 - calledby PARSE-Sequence1, [445](#)
 - calls PARSE-Iterator, [447](#)
 - calls bang, [447](#)
 - calls match-advance-string, [447](#)
 - calls optional, [447](#)
 - calls star, [447](#)
 - defun, [447](#)
- parse-keyword, [548](#)
 - calledby PARSE-AnyId, [444](#)
 - calls advance-token, [548](#)
 - calls match-current-token, [548](#)
 - calls push-reduction, [548](#)
 - calls token-symbol, [548](#)
 - defun, [548](#)
- PARSE-Label, [433](#)
 - calledby PARSE-LabelExpr, [452](#)
 - calledby PARSE-Leave, [450](#)
 - calls PARSE-Name, [433](#)
 - calls match-advance-string, [433](#)
 - calls must, [433](#)
 - defun, [433](#)
- PARSE-LabelExpr, [452](#)
 - calls PARSE-Expr, [452](#)
 - calls PARSE-Label, [452](#)
 - calls must, [452](#)
 - calls pop-stack-1, [452](#)
 - calls pop-stack-2, [452](#)
 - calls push-reduction, [452](#)
 - defun, [452](#)
- PARSE-Leave, [450](#)
 - calls PARSE-Expression, [450](#)
 - calls PARSE-Label, [450](#)
 - calls match-advance-string, [450](#)
 - calls must, [450](#)
 - calls pop-stack-1, [450](#)
 - calls push-reduction, [450](#)
 - defun, [450](#)
- PARSE-LedPart, [426](#)
 - calledby PARSE-Expr, [426](#)
 - calls PARSE-Operation, [426](#)
 - calls pop-stack-1, [426](#)
 - calls push-reduction, [426](#)
 - defun, [426](#)
- PARSE-leftBindingPowerOf, [427](#)
 - calledby PARSE-Operation, [427](#)
 - calls elemn, [428](#)
 - calls getl, [427](#)
 - defun, [427](#)
- PARSE-Loop, [452](#)
 - calls PARSE-Expr, [452](#)
 - calls PARSE-Iterator, [452](#)
 - calls match-advance-string, [452](#)
 - calls must, [452](#)
 - calls pop-stack-1, [452](#)
 - calls pop-stack-2, [452](#)
 - calls push-reduction, [452](#)
 - calls star, [452](#)
 - defun, [452](#)
- PARSE-Name, [441](#)
 - calledby PARSE-Label, [433](#)
 - calledby PARSE-VarForm, [440](#)
 - calls parse-identifier, [441](#)
 - calls pop-stack-1, [441](#)
 - calls push-reduction, [441](#)
 - defun, [441](#)
- PARSE-NBGlyphTok, [443](#)

- calledby PARSE-Sexpr1, 442
- calls action, 443
- calls advance-token, 443
- calls match-current-token, 443
- uses tok, 443
- defun, 443
- PARSE-NewExpr, 417
 - calledby spad, 574
 - calls PARSE-Statement, 417
 - calls action, 417
 - calls current-symbol, 417
 - calls match-string, 417
 - calls must, 417
 - calls processSynonyms[5], 417
 - uses definition-name, 417
 - defun, 417
- PARSE-NudPart, 426
 - calledby PARSE-Expr, 426
 - calls PARSE-Form, 426
 - calls PARSE-Operation, 426
 - calls PARSE-Reduction, 426
 - calls pop-stack-1, 427
 - calls push-reduction, 426
 - uses rbp, 427
 - defun, 426
- parse-number, 547
 - calledby PARSE-FloatTok, 453
 - calledby PARSE-IntegerTok, 438
 - calls advance-token, 547
 - calls match-current-token, 547
 - calls push-reduction, 547
 - calls token-symbol, 547
 - defun, 547
- PARSE-OpenBrace, 446
 - calledby PARSE-Sequence, 445
 - calls action, 446
 - calls advance-token, 446
 - calls current-symbol, 446
 - calls eqcar, 446
 - calls getToken, 446
 - calls push-reduction, 446
 - defun, 446
- PARSE-OpenBracket, 446
 - calledby PARSE-Sequence, 445
 - calls action, 446
 - calls advance-token, 446
- calls current-symbol, 446
- calls eqcar, 446
- calls getToken, 446
- calls push-reduction, 446
- defun, 446
- PARSE-Operation, 427
 - calledby PARSE-LedPart, 426
 - calledby PARSE-NudPart, 426
 - calls PARSE-getSemanticForm, 427
 - calls PARSE-leftBindingPowerOf, 427
 - calls PARSE-rightBindingPowerOf, 427
 - calls action, 427
 - calls current-symbol, 427
 - calls elemn, 427
 - calls getl, 427
 - calls lt, 427
 - calls match-current-token, 427
 - uses ParseMode, 427
 - uses rbp, 427
 - uses tmptok, 427
 - defun, 427
- PARSE-Option, 422
 - calledby PARSE-CommandTail, 421
 - calls PARSE-PrimaryOrQM, 422
 - calls match-advance-string, 422
 - calls must, 422
 - calls star, 422
 - defun, 422
- PARSE-Prefix, 428
 - calledby PARSE-getSemanticForm, 428
 - calls PARSE-Expression, 429
 - calls PARSE-TokTail, 429
 - calls action, 428
 - calls advance-token, 429
 - calls current-symbol, 428
 - calls must, 429
 - calls optional, 429
 - calls pop-stack-1, 429
 - calls pop-stack-2, 429
 - calls push-reduction, 428, 429
 - defun, 428
- PARSE-Primary, 434
 - calledby PARSE-Application, 432
 - calledby PARSE-Iterator, 447
 - calledby PARSE-PrimaryOrQM, 421
 - calledby PARSE-Selector, 433

- calls PARSE-Float, [434](#)
- calls PARSE-PrimaryNoFloat, [434](#)
- defun, [434](#)
- PARSE-Primary1, [434](#)
 - calledby PARSE-Primary1, [434](#)
 - calledby PARSE-PrimaryNoFloat, [434](#)
 - calledby PARSE-Qualification, [430](#)
 - calls PARSE-Data, [435](#)
 - calls PARSE-Enclosure, [435](#)
 - calls PARSE-Expr, [435](#)
 - calls PARSE-FormalParameter, [435](#)
 - calls PARSE-IntegerTok, [435](#)
 - calls PARSE-Primary1, [434](#)
 - calls PARSE-Quad, [435](#)
 - calls PARSE-Sequence, [435](#)
 - calls PARSE-String, [435](#)
 - calls PARSE-VarForm, [434](#)
 - calls current-symbol, [434](#)
 - calls match-advance-string, [435](#)
 - calls match-string, [435](#)
 - calls must, [434](#)
 - calls optional, [434](#)
 - calls pop-stack-1, [434](#)
 - calls pop-stack-2, [434](#)
 - calls push-reduction, [434](#)
 - local ref \$boot, [435](#)
 - defun, [434](#)
- PARSE-PrimaryNoFloat, [434](#)
 - calledby PARSE-Primary, [434](#)
 - calledby PARSE-Selector, [433](#)
 - calls PARSE-Primary1, [434](#)
 - calls PARSE-TokTail, [434](#)
 - calls optional, [434](#)
 - defun, [434](#)
- PARSE-PrimaryOrQM, [421](#)
 - calledby PARSE-Option, [422](#)
 - calledby PARSE-PrimaryOrQM, [421](#)
 - calledby PARSE-SpecialCommand, [419](#)
 - calls PARSE-PrimaryOrQM, [421](#)
 - calls PARSE-Primary, [421](#)
 - calls match-advance-string, [421](#)
 - calls push-reduction, [421](#)
 - defun, [421](#)
- PARSE-Quad, [439](#)
 - calledby PARSE-Primary1, [435](#)
 - calls PARSE-GlyphTok, [439](#)
 - calls match-advance-string, [439](#)
 - calls push-reduction, [439](#)
 - uses \$boot, [439](#)
 - defun, [439](#)
- PARSE-Qualification, [430](#)
 - calledby PARSE-TokTail, [430](#)
 - calls PARSE-Primary1, [430](#)
 - calls dollarTran, [430](#)
 - calls match-advance-string, [430](#)
 - calls must, [430](#)
 - calls pop-stack-1, [430](#)
 - calls push-reduction, [430](#)
 - defun, [430](#)
- PARSE-Reduction, [431](#)
 - calledby PARSE-NudPart, [426](#)
 - calls PARSE-Expr, [431](#)
 - calls PARSE-ReductionOp, [431](#)
 - calls must, [431](#)
 - calls pop-stack-1, [431](#)
 - calls pop-stack-2, [431](#)
 - calls push-reduction, [431](#)
 - defun, [431](#)
- PARSE-ReductionOp, [431](#)
 - calledby PARSE-Reduction, [431](#)
 - calls action, [431](#)
 - calls advance-token, [431](#)
 - calls current-symbol, [431](#)
 - calls getl, [431](#)
 - calls match-next-token, [431](#)
 - defun, [431](#)
- PARSE-Return, [449](#)
 - calls PARSE-Expression, [449](#)
 - calls match-advance-string, [449](#)
 - calls must, [449](#)
 - calls pop-stack-1, [449](#)
 - calls push-reduction, [449](#)
 - defun, [449](#)
- PARSE-rightBindingPowerOf, [428](#)
 - calledby PARSE-Expression, [425](#)
 - calledby PARSE-Operation, [427](#)
 - calls elemn, [428](#)
 - calls getl, [428](#)
 - defun, [428](#)
- PARSE-ScriptItem, [441](#)
 - calledby PARSE-ScriptItem, [441](#)
 - calledby PARSE-Scripts, [440](#)

- calls PARSE-Expr, 441
- calls PARSE-ScriptItem, 441
- calls match-advance-string, 441
- calls must, 441
- calls optional, 441
- calls pop-stack-1, 441
- calls pop-stack-2, 441
- calls push-reduction, 441
- calls star, 441
- defun, 441
- PARSE-Scripts, 440
 - calledby PARSE-VarForm, 440
 - calls PARSE-ScriptItem, 440
 - calls match-advance-string, 440
 - calls must, 440
 - defun, 440
- PARSE-Seg, 450
 - calls PARSE-Expression, 450
 - calls PARSE-GlyphTok, 450
 - calls bang, 450
 - calls optional, 450
 - calls pop-stack-1, 451
 - calls pop-stack-2, 451
 - calls push-reduction, 450
 - defun, 450
- PARSE-Selector, 433
 - calledby PARSE-Application, 432
 - calls PARSE-Float, 433
 - calls PARSE-PrimaryNoFloat, 433
 - calls PARSE-Primary, 433
 - calls char-ne, 433
 - calls current-char, 433
 - calls current-symbol, 433
 - calls match-advance-string, 433
 - calls must, 433
 - calls pop-stack-1, 433
 - calls pop-stack-2, 433
 - calls push-reduction, 433
 - uses \$boot, 433
 - defun, 433
- PARSE-SemiColon, 449
 - calls PARSE-Expr, 449
 - calls match-advance-string, 449
 - calls must, 449
 - calls pop-stack-1, 449
 - calls pop-stack-2, 449
- calls push-reduction, 449
- defun, 449
- PARSE-Sequence, 445
 - calledby PARSE-Primary1, 435
 - calls PARSE-OpenBrace, 445
 - calls PARSE-OpenBracket, 445
 - calls PARSE-Sequence1, 445
 - calls match-advance-string, 445
 - calls must, 445
 - calls pop-stack-1, 445
 - calls push-reduction, 445
 - defun, 445
- PARSE-Sequence1, 445
 - calledby PARSE-Sequence, 445
 - calls PARSE-Expression, 445
 - calls PARSE-IteratorTail, 445
 - calls optional, 445
 - calls pop-stack-1, 445
 - calls pop-stack-2, 445
 - calls push-reduction, 445
 - defun, 445
- PARSE-Sexpr, 442
 - calledby PARSE-Data, 442
 - calls PARSE-Sexpr1, 442
 - defun, 442
- PARSE-Sexpr1, 442
 - calledby PARSE-Sexpr1, 442
 - calledby PARSE-Sexpr, 442
 - calls PARSE-AnyId, 442
 - calls PARSE-GlyphTok, 443
 - calls PARSE-IntegerTok, 442
 - calls PARSE-NBGlyphTok, 442
 - calls PARSE-Sexpr1, 442
 - calls PARSE-String, 443
 - calls action, 442
 - calls bang, 443
 - calls match-advance-string, 442
 - calls must, 442
 - calls nth-stack, 442
 - calls optional, 442
 - calls pop-stack-1, 443
 - calls pop-stack-2, 442
 - calls push-reduction, 442
 - calls star, 443
 - defun, 442
- parse-spadstring, 546

- calledby PARSE-String, 439
- calls advance-token, 546
- calls match-current-token, 546
- calls push-reduction, 546
- calls token-symbol, 546
- defun, 546
- PARSE-SpecialCommand, 419
 - calledby PARSE-Command, 418
 - calledby PARSE-SpecialCommand, 419
 - calls PARSE-CommandTail, 419
 - calls PARSE-Expression, 419
 - calls PARSE-PrimaryOrQM, 419
 - calls PARSE-SpecialCommand, 419
 - calls PARSE-TokenCommandTail, 419
 - calls PARSE-TokenList, 419
 - calls action, 419
 - calls bang, 419
 - calls current-symbol, 419
 - calls match-advance-string, 419
 - calls must, 419
 - calls optional, 419
 - calls pop-stack-1, 419
 - calls push-reduction, 419
 - calls star, 419
 - local ref \$noParseCommands, 419
 - local ref \$tokenCommands, 419
 - defun, 419
- PARSE-SpecialKeyWord, 418
 - calledby PARSE-Command, 418
 - calls action, 418
 - calls current-symbol, 418
 - calls current-token, 418
 - calls match-current-token, 418
 - calls token-symbol, 418
 - calls unAbbreviateKeyword[5], 418
 - defun, 418
- PARSE-Statement, 422
 - calledby PARSE-NewExpr, 417
 - calls PARSE-Expr, 422
 - calls match-advance-string, 422
 - calls must, 422
 - calls optional, 422
 - calls pop-stack-1, 422
 - calls pop-stack-2, 422
 - calls push-reduction, 422
 - calls star, 422
- defun, 422
- PARSE-String, 439
 - calledby PARSE-Primary1, 435
 - calledby PARSE-Sexpr1, 443
 - calls parse-spadstring, 439
 - defun, 439
- parse-string, 546
 - calls advance-token, 546
 - calls match-current-token, 546
 - calls push-reduction, 546
 - calls token-symbol, 546
 - defun, 546
- PARSE-Suffix, 448
 - calls PARSE-TokTail, 448
 - calls action, 448
 - calls advance-token, 448
 - calls current-symbol, 448
 - calls optional, 448
 - calls pop-stack-1, 448
 - calls push-reduction, 448
 - defun, 448
- PARSE-TokenCommandTail, 419
 - calledby PARSE-SpecialCommand, 419
 - calledby PARSE-TokenCommandTail, 420
 - calls PARSE-TokenCommandTail, 420
 - calls PARSE-TokenOption, 420
 - calls action, 420
 - calls atEndOfLine, 420
 - calls bang, 419
 - calls optional, 420
 - calls pop-stack-1, 420
 - calls pop-stack-2, 420
 - calls push-reduction, 420
 - calls star, 420
 - calls systemCommand[5], 420
 - defun, 419
- PARSE-TokenList, 420
 - calledby PARSE-SpecialCommand, 419
 - calledby PARSE-TokenOption, 420
 - calls action, 420
 - calls advance-token, 420
 - calls current-symbol, 420
 - calls isTokenDelimiter, 420
 - calls push-reduction, 420
 - calls star, 420
 - defun, 420

- PARSE-TokenOption, 420
 - calledby PARSE-TokenCommandTail, 420
 - calls PARSE-TokenList, 420
 - calls match-advance-string, 420
 - calls must, 420
 - defun, 420
- PARSE-TokTail, 430
 - calledby PARSE-Infix, 429
 - calledby PARSE-Prefix, 429
 - calledby PARSE-PrimaryNoFloat, 434
 - calledby PARSE-Suffix, 448
 - calls PARSE-Qualification, 430
 - calls action, 430
 - calls char-eq, 430
 - calls copy-token, 430
 - calls current-char, 430
 - calls current-symbol, 430
 - uses \$boot, 430
 - defun, 430
- PARSE-VarForm, 440
 - calledby PARSE-Primary1, 434
 - calls PARSE-Name, 440
 - calls PARSE-Scripts, 440
 - calls optional, 440
 - calls pop-stack-1, 440
 - calls pop-stack-2, 440
 - calls push-reduction, 440
 - defun, 440
- PARSE-With, 423
 - calledby PARSE-InfixWith, 423
 - calls match-advance-string, 423
 - calls must, 423
 - calls pop-stack-1, 423
 - calls push-reduction, 423
 - defun, 423
- parseAnd, 101
 - calledby parseAnd, 101
 - calls parseAnd, 101
 - calls parseIf, 101
 - calls parseTranList, 101
 - calls parseTran, 101
 - uses \$InteractiveMode, 101
 - defun, 101
- parseAtom, 98
 - calledby parseTran, 97
 - calls parseLeave, 98
 - uses \$NoValue, 98
 - defun, 98
- parseAtSign, 102
 - calls parseTran, 102
 - calls parseType, 102
 - uses \$InteractiveMode, 102
 - defun, 102
- parseCategory, 103
 - calls contained, 103
 - calls parseDropAssertions, 103
 - calls parseTranList, 103
 - defun, 103
- parseCoerce, 104
 - calls parseTran, 104
 - calls parseType, 104
 - uses \$InteractiveMode, 104
 - defun, 104
- parseColon, 104
 - calls parseTran, 104
 - calls parseType, 104
 - local ref \$insideConstructIfTrue, 104
 - uses \$InteractiveMode, 104
 - defun, 104
- parseConstruct, 99
 - calledby parseTran, 97
 - calls parseTranList, 99
 - uses \$insideConstructIfTrue, 99
 - defun, 99
- parseDEF, 105
 - calls opFf, 105
 - calls parseLhs, 105
 - calls parseTranCheckForRecord, 105
 - calls parseTranList, 105
 - calls setDefOp, 105
 - uses \$lhs, 105
 - defun, 105
- parseDollarGreaterEqual, 108
 - calls parseTran, 108
 - uses \$op, 108
 - defun, 108
- parseDollarGreaterThan, 108
 - calls parseTran, 108
 - uses \$op, 108
 - defun, 108
- parseDollarLessEqual, 109
 - calls parseTran, 109

- uses \$op, 109
 - defun, 109
- parseDollarNotEqual, 109
 - calls parseTran, 109
 - uses \$op, 109
 - defun, 109
- parseDropAssertions, 103
 - calledby parseCategory, 103
 - calledby parseDropAssertions, 103
 - calls parseDropAssertions, 103
 - defun, 103
- parseEquivalence, 110
 - calls parseIf, 110
 - defun, 110
- parseExit, 110
 - calls moan, 110
 - calls parseTran, 110
 - defun, 110
- parseGreaterEqual, 111
 - calls parseTran, 111
 - uses \$op, 111
 - defun, 111
- parseGreaterThan, 112
 - calls parseTran, 112
 - uses \$op, 112
 - defun, 112
- parseHas, 112
 - calls getdatabase, 112
 - calls makeNonAtomic, 112
 - calls member, 112
 - calls nreverse0, 112
 - calls opOf, 112
 - calls parseHasRhs, 112
 - calls parseType, 112
 - calls unabbrevAndLoad, 112
 - uses \$CategoryFrame, 112
 - uses \$InteractiveMode, 112
 - defun, 112
- parseHasRhs, 114
 - calledby parseHas, 112
 - calls abbreviation?, 114
 - calls get, 114
 - calls loadIfNecessary, 114
 - calls member, 114
 - calls unabbrevAndLoad, 114
 - uses \$CategoryFrame, 114
 - defun, 114
- parseIf, 118
 - calledby parseAnd, 101
 - calledby parseEquivalence, 110
 - calledby parseImplies, 120
 - calledby parseOr, 129
 - calls parseIf,ifTran, 118
 - calls parseTran, 118
 - defun, 118
- parseIf,ifTran, 118
 - calledby parseIf,ifTran, 118
 - calledby parseIf, 118
 - calls incExitLevel, 118
 - calls makeSimplePredicateOrNil, 118
 - calls parseIf,ifTran, 118
 - calls parseTran, 118
 - uses \$InteractiveMode, 118
 - defun, 118
- parseImplies, 120
 - calls parseIf, 120
 - defun, 120
- parseIn, 121
 - calledby parseInBy, 122
 - calls parseTran, 121
 - calls postError, 121
 - defun, 121
- parseInBy, 122
 - calls bright, 122
 - calls parseIn, 122
 - calls parseTran, 122
 - calls postError, 122
 - defun, 122
- parseIs, 123
 - calls parseTran, 123
 - calls transIs, 123
 - defun, 123
- parseIsnt, 123
 - calls parseTran, 123
 - calls transIs, 124
 - defun, 123
- parseJoin, 124
 - calls parseTranList, 124
 - defun, 124
- parseLeave, 125
 - calledby parseAtom, 98
 - calls parseTran, 125

- defun, 125
- parseLessEqual, 125
 - calls parseTran, 125
 - uses \$op, 125
 - defun, 125
- parseLET, 126
 - calls opOf, 126
 - calls parseTranCheckForRecord, 126
 - calls parseTran, 126
 - calls transIs, 126
 - defun, 126
- parseLETD, 127
 - calls parseTran, 127
 - calls parseType, 127
 - defun, 127
- parseLhs, 106
 - calledby parseDEF, 105
 - calls parseTran, 106
 - calls transIs, 106
 - defun, 106
- parseMDEF, 127
 - calls opOf, 127
 - calls parseTranCheckForRecord, 127
 - calls parseTranList, 127
 - calls parseTran, 127
 - uses \$lhs, 127
 - defun, 127
- ParseMode, 417
 - usedby PARSE-Expression, 425
 - usedby PARSE-Operation, 427
 - defvar, 417
- parseNot, 128
 - calls parseTran, 128
 - uses \$InteractiveMode, 128
 - defun, 128
- parseNotEqual, 129
 - calls parseTran, 129
 - uses \$op, 129
 - defun, 129
- parseOr, 129
 - calledby parseOr, 129
 - calls parseIf, 129
 - calls parseOr, 129
 - calls parseTranList, 129
 - calls parseTran, 129
 - defun, 129
- parsepiles, 87
 - calledby preparse1, 84
 - calls add-parens-and-semis-to-line, 87
 - defun, 87
- parsePretend, 130
 - calls parseTran, 130
 - calls parseType, 130
 - defun, 130
- parseprint, 556
 - calledby preparse, 80
 - defun, 556
- parseReturn, 131
 - calls moan, 131
 - calls parseTran, 131
 - defun, 131
- parseSegment, 131
 - calls parseTran, 131
 - defun, 131
- parseSeq, 132
 - calls last, 132
 - calls mapInto, 132
 - calls postError, 132
 - calls transSeq, 132
 - defun, 132
- parseTran, 97
 - calledby compReduce1, 326
 - calledby parseAnd, 101
 - calledby parseAtSign, 102
 - calledby parseCoerce, 104
 - calledby parseColon, 104
 - calledby parseDollarGreaterEqual, 108
 - calledby parseDollarGreaterThan, 108
 - calledby parseDollarLessEqual, 109
 - calledby parseDollarNotEqual, 109
 - calledby parseExit, 110
 - calledby parseGreaterEqual, 111
 - calledby parseGreaterThan, 112
 - calledby parseIf,ifTran, 118
 - calledby parseIf, 118
 - calledby parseInBy, 122
 - calledby parseIn, 121
 - calledby parseIsnt, 123
 - calledby parseIs, 123
 - calledby parseLETD, 127
 - calledby parseLET, 126
 - calledby parseLeave, 125

- calledby parseLessEqual, 125
- calledby parseLhs, 106
- calledby parseMDEF, 127
- calledby parseNotEqual, 129
- calledby parseNot, 128
- calledby parseOr, 129
- calledby parsePretend, 130
- calledby parseReturn, 131
- calledby parseSegment, 131
- calledby parseTranCheckForRecord, 545
- calledby parseTranList, 99
- calledby parseTransform, 97
- calledby parseTran, 97
- calledby parseType, 102
- calls getl, 97
- calls parseAtom, 97
- calls parseConstruct, 97
- calls parseTranList, 97
- calls parseTran, 97
- uses \$op, 97
- defun, 97
- parseTranCheckForRecord, 545
 - calledby parseDEF, 105
 - calledby parseLET, 126
 - calledby parseMDEF, 127
 - calls parseTran, 545
 - calls postError, 545
 - defun, 545
- parseTranList, 99
 - calledby parseAnd, 101
 - calledby parseCategory, 103
 - calledby parseConstruct, 99
 - calledby parseDEF, 105
 - calledby parseJoin, 124
 - calledby parseMDEF, 127
 - calledby parseOr, 129
 - calledby parseTranList, 99
 - calledby parseTran, 97
 - calledby parseVCONS, 133
 - calls parseTranList, 99
 - calls parseTran, 99
 - defun, 99
- parseTransform, 97
 - calledby s-process, 583
 - calls parseTran, 97
 - uses \$defOp, 97
- defun, 97
- parseType, 102
 - calledby parseAtSign, 102
 - calledby parseCoerce, 104
 - calledby parseColon, 104
 - calledby parseHas, 112
 - calledby parseLETD, 127
 - calledby parsePretend, 130
 - calls parseTran, 102
 - defun, 102
- parseVCONS, 133
 - calls parseTranList, 133
 - defun, 133
- parseWhere, 133
 - calls mapInto, 133
 - defun, 133
- pathname[5]
 - called by compileSpad2Cmd, 566
 - called by compileSpadLispCmd, 568
 - called by compiler, 563
- pathnameDirectory[5]
 - called by compileSpadLispCmd, 568
- pathnameName[5]
 - called by compileSpadLispCmd, 568
- pathnameType[5]
 - called by compileSpad2Cmd, 566
 - called by compileSpadLispCmd, 568
 - called by compiler, 563
- pathnameTypeId
 - calledby initializeLisplib, 202
- pmatch
 - calledby coerceable, 356
- pmatchWithSl
 - calledby compApplyModemap, 255
- pname
 - calledby buildLibdbConEntry, 482
 - calledby checkTransformFirsts, 513
 - calledby compDefineFunctor1, 144
 - calledby encodeItem, 181
 - calledby isCategoryPackageName, 210
 - calledby mkCategoryPackage, 176
 - calledby recordAttributeDocumentation, 470
- pname[5]
 - called by comp3, 592
 - called by floatexpid, 465

- called by getScriptName, [404](#)
- Pop-Reduction, [552](#)
 - calledby pop-stack-1, [550](#)
 - calledby pop-stack-2, [550](#)
 - calledby pop-stack-3, [551](#)
 - calledby pop-stack-4, [551](#)
 - calls stack-pop, [552](#)
 - defun, [552](#)
- pop-stack-1, [550](#)
 - calledby PARSE-Application, [432](#)
 - calledby PARSE-Category, [424](#)
 - calledby PARSE-CommandTail, [421](#)
 - calledby PARSE-Conditional, [451](#)
 - calledby PARSE-Data, [442](#)
 - calledby PARSE-Enclosure, [438](#)
 - calledby PARSE-Exit, [450](#)
 - calledby PARSE-Expression, [425](#)
 - calledby PARSE-Expr, [426](#)
 - calledby PARSE-FloatTok, [453](#)
 - calledby PARSE-Float, [435](#)
 - calledby PARSE-Form, [432](#)
 - calledby PARSE-Import, [425](#)
 - calledby PARSE-InfixWith, [423](#)
 - calledby PARSE-Infix, [429](#)
 - calledby PARSE-Iterator, [447](#)
 - calledby PARSE-LabelExpr, [452](#)
 - calledby PARSE-Leave, [450](#)
 - calledby PARSE-LedPart, [426](#)
 - calledby PARSE-Loop, [452](#)
 - calledby PARSE-Name, [441](#)
 - calledby PARSE-NudPart, [427](#)
 - calledby PARSE-Prefix, [429](#)
 - calledby PARSE-Primary1, [434](#)
 - calledby PARSE-Qualification, [430](#)
 - calledby PARSE-Reduction, [431](#)
 - calledby PARSE-Return, [449](#)
 - calledby PARSE-ScriptItem, [441](#)
 - calledby PARSE-Seg, [451](#)
 - calledby PARSE-Selector, [433](#)
 - calledby PARSE-SemiColon, [449](#)
 - calledby PARSE-Sequence1, [445](#)
 - calledby PARSE-Sequence, [445](#)
 - calledby PARSE-Sexpr1, [443](#)
 - calledby PARSE-SpecialCommand, [419](#)
 - calledby PARSE-Statement, [422](#)
 - calledby PARSE-Suffix, [448](#)
 - calledby PARSE-TokenCommandTail, [420](#)
 - calledby PARSE-VarForm, [440](#)
 - calls Pop-Reduction, [550](#)
 - calls reduction-value, [550](#)
 - defmacro, [550](#)
- pop-stack-2, [550](#)
 - calledby PARSE-Application, [432](#)
 - calledby PARSE-Category, [423](#)
 - calledby PARSE-CommandTail, [421](#)
 - calledby PARSE-Conditional, [451](#)
 - calledby PARSE-Float, [435](#)
 - calledby PARSE-Import, [425](#)
 - calledby PARSE-InfixWith, [423](#)
 - calledby PARSE-Infix, [429](#)
 - calledby PARSE-Iterator, [447](#)
 - calledby PARSE-LabelExpr, [452](#)
 - calledby PARSE-Loop, [452](#)
 - calledby PARSE-Prefix, [429](#)
 - calledby PARSE-Primary1, [434](#)
 - calledby PARSE-Reduction, [431](#)
 - calledby PARSE-ScriptItem, [441](#)
 - calledby PARSE-Seg, [451](#)
 - calledby PARSE-Selector, [433](#)
 - calledby PARSE-SemiColon, [449](#)
 - calledby PARSE-Sequence1, [445](#)
 - calledby PARSE-Sexpr1, [442](#)
 - calledby PARSE-Statement, [422](#)
 - calledby PARSE-TokenCommandTail, [420](#)
 - calledby PARSE-VarForm, [440](#)
 - calls Pop-Reduction, [550](#)
 - calls reduction-value, [550](#)
 - calls stack-push, [550](#)
 - defmacro, [550](#)
- pop-stack-3, [551](#)
 - calledby PARSE-Category, [423](#)
 - calledby PARSE-Conditional, [451](#)
 - calledby PARSE-Float, [435](#)
 - calledby PARSE-Iterator, [447](#)
 - calls Pop-Reduction, [551](#)
 - calls reduction-value, [551](#)
 - calls stack-push, [551](#)
 - defmacro, [551](#)
- pop-stack-4, [551](#)

- calledby PARSE-Float, 435
- calls Pop-Reduction, 551
- calls reduction-value, 551
- calls stack-push, 551
- defmacro, 551
- postAdd, 372
 - calls postCapsule, 372
 - calls postTran, 372
 - defun, 372
- postAtom, 367
 - calledby postTran, 366
 - local ref \$boot, 367
 - defun, 367
- postAtSign, 374
 - calls postTran, 374
 - calls postType, 374
 - defun, 374
- postBigFloat, 376
 - calls postTran, 376
 - uses \$InteractiveMode, 376
 - uses \$boot, 376
 - defun, 376
- postBlock, 376
 - calledby postSemiColon, 395
 - calls postBlockItemList, 376
 - calls postTran, 376
 - defun, 376
- postBlockItem, 373
 - calledby postBlockItemList, 373
 - calledby postCapsule, 373
 - calls postTran, 373
 - defun, 373
- postBlockItemList, 373
 - calledby postBlock, 376
 - calledby postCapsule, 373
 - calls postBlockItem, 373
 - defun, 373
- postCapsule, 373
 - calledby postAdd, 372
 - calls checkWarning, 373
 - calls postBlockItemList, 373
 - calls postBlockItem, 373
 - calls postFlatten, 373
 - defun, 373
- postCategory, 377
 - calls nreverse0, 377
 - calls postTran, 377
 - uses \$insidePostCategoryIfTrue, 377
 - defun, 377
- postcheck, 369
 - calledby postTransformCheck, 369
 - calledby postcheck, 369
 - calls postcheck, 369
 - calls setDefOp, 369
 - defun, 369
- postCollect, 379
 - calledby postCollect, 379
 - calledby postTupleCollect, 398
 - calls postCollect,finish, 379
 - calls postCollect, 379
 - calls postIteratorList, 379
 - calls postTran, 379
 - defun, 379
- postCollect,finish, 377
 - calledby postCollect, 379
 - calls postMakeCons, 377
 - calls postTranList, 378
 - calls tuple2List, 378
 - defun, 377
- postColon, 381
 - calls postTran, 381
 - calls postType, 381
 - defun, 381
- postColonColon, 381
 - calls postForm, 381
 - uses \$boot, 381
 - defun, 381
- postComma, 382
 - calls comma2Tuple, 382
 - calls postTuple, 382
 - defun, 382
- postConstruct, 383
 - calls comma2Tuple, 383
 - calls postMakeCons, 383
 - calls postTranList, 383
 - calls postTranSegment, 383
 - calls postTran, 383
 - calls tuple2List, 383
 - defun, 383
- postDef, 384
 - calls nreverse0, 385
 - calls postDefArgs, 385

- calls postMDef, 384
- calls postTran, 384
- calls recordHeaderDocumentation, 384
- uses \$InteractiveMode, 385
- uses \$boot, 385
- uses \$docList, 385
- uses \$headerDocumentation, 385
- uses \$maxSignatureLineNumber, 385
- defun, 384
- postDefArgs, 386
 - calledby postDefArgs, 386
 - calledby postDef, 385
 - calls postDefArgs, 386
 - calls postError, 386
 - defun, 386
- postError, 370
 - calledby checkWarning, 549
 - calledby getScriptName, 404
 - calledby parseInBy, 122
 - calledby parseIn, 121
 - calledby parseSeq, 132
 - calledby parseTranCheckForRecord, 545
 - calledby postDefArgs, 386
 - calledby postForm, 370
 - calls bumpererrorcount, 370
 - uses \$InteractiveMode, 370
 - uses \$defOp, 370
 - uses \$postStack, 370
 - defun, 370
- postExit, 387
 - calls postTran, 387
 - defun, 387
- postFlatten, 382
 - calledby comma2Tuple, 382
 - calledby postCapsule, 373
 - calledby postFlatten, 382
 - calls postFlatten, 382
 - defun, 382
- postFlattenLeft, 395
 - calledby postFlattenLeft, 395
 - calledby postSemiColon, 395
 - calls postFlattenLeft, 395
 - defun, 395
- postForm, 370
 - calledby postColonColon, 381
 - calledby postTran, 366
 - calls bright, 370
 - calls internl, 370
 - calls postError, 370
 - calls postTranList, 370
 - calls postTran, 370
 - uses \$boot, 370
 - defun, 370
- postIf, 387
 - calls nreverse0, 387
 - calls postTran, 387
 - uses \$boot, 387
 - defun, 387
- postIn, 389
 - calls postInSeq, 389
 - calls postTran, 389
 - calls systemErrorHere, 389
 - defun, 389
- postin, 388
 - calls postInSeq, 388
 - calls postTran, 388
 - calls systemErrorHere, 388
 - defun, 388
- postInSeq, 388
 - calledby postIn, 389
 - calledby postIteratorList, 380
 - calledby postin, 388
 - calls postTranSegment, 388
 - calls postTran, 388
 - calls tuple2List, 388
 - defun, 388
- postIteratorList, 380
 - calledby postCollect, 379
 - calledby postIteratorList, 380
 - calledby postRepeat, 394
 - calls postInSeq, 380
 - calls postIteratorList, 380
 - calls postTran, 380
 - defun, 380
- postJoin, 390
 - calls postTranList, 390
 - calls postTran, 390
 - defun, 390
- postMakeCons, 378
 - calledby postCollect, finish, 377
 - calledby postConstruct, 383
 - calledby postMakeCons, 378

- calls postMakeCons, 378
 - calls postTran, 378
 - defun, 378
- postMapping, 390
 - calls postTran, 390
 - calls unTuple, 390
 - defun, 390
- postMDef, 391
 - calledby postDef, 384
 - calls nreverse0, 391
 - calls postTran, 391
 - calls throwkeyedmsg, 391
 - uses \$InteractiveMode, 391
 - uses \$boot, 391
 - defun, 391
- postOp, 367
 - calledby postTran, 366
 - defun, 367
- postPretend, 392
 - calls postTran, 392
 - calls postType, 392
 - defun, 392
- postQUOTE, 393
 - defun, 393
- postReduce, 393
 - calledby postReduce, 393
 - calls postReduce, 393
 - calls postTran, 393
 - uses \$InteractiveMode, 393
 - defun, 393
- postRepeat, 394
 - calls postIteratorList, 394
 - calls postTran, 394
 - defun, 394
- postScripts, 394
 - calls getScriptName, 394
 - calls postTranScripts, 394
 - defun, 394
- postScriptsForm, 368
 - calledby postTran, 366
 - calls getScriptName, 368
 - calls length, 368
 - calls postTranScripts, 368
 - defun, 368
- postSemiColon, 395
 - calls postBlock, 395
 - calls postFlattenLeft, 395
 - defun, 395
- postSignature, 396
 - calls killColons, 396
 - calls postType, 396
 - calls removeSuperfluousMapping, 396
 - defun, 396
- postSlash, 397
 - calls postTran, 397
 - defun, 397
- postTran, 366
 - calledby postAdd, 372
 - calledby postAtSign, 374
 - calledby postBigFloat, 376
 - calledby postBlockItem, 373
 - calledby postBlock, 376
 - calledby postCategory, 377
 - calledby postCollect, 379
 - calledby postColon, 381
 - calledby postConstruct, 383
 - calledby postDef, 384
 - calledby postExit, 387
 - calledby postForm, 370
 - calledby postIf, 387
 - calledby postInSeq, 388
 - calledby postIn, 389
 - calledby postIteratorList, 380
 - calledby postJoin, 390
 - calledby postMDef, 391
 - calledby postMakeCons, 378
 - calledby postMapping, 390
 - calledby postPretend, 392
 - calledby postReduce, 393
 - calledby postRepeat, 394
 - calledby postSlash, 397
 - calledby postTranList, 368
 - calledby postTranScripts, 368
 - calledby postTranSegment, 384
 - calledby postTransform, 365
 - calledby postTran, 366
 - calledby postType, 375
 - calledby postWhere, 399
 - calledby postWith, 400
 - calledby postin, 388
 - calledby tuple2List, 549
 - calls postAtom, 366

- calls postForm, 366
- calls postOp, 366
- calls postScriptsForm, 366
- calls postTranList, 366
- calls postTran, 366
- calls unTuple, 366
- defun, 366
- postTranList, 368
 - calledby postCollect,finish, 378
 - calledby postConstruct, 383
 - calledby postForm, 370
 - calledby postJoin, 390
 - calledby postTran, 366
 - calledby postTuple, 398
 - calledby postWhere, 399
 - calls postTran, 368
 - defun, 368
- postTranScripts, 368
 - calledby postScriptsForm, 368
 - calledby postScripts, 394
 - calledby postTranScripts, 368
 - calls postTranScripts, 368
 - calls postTran, 368
 - defun, 368
- postTranSegment, 384
 - calledby postConstruct, 383
 - calledby postInSeq, 388
 - calledby tuple2List, 549
 - calls postTran, 384
 - defun, 384
- postTransform, 365
 - calledby new2OldLisp, 72
 - calledby recordAttributeDocumentation, 470
 - calledby recordSignatureDocumentation, 470
 - calledby s-process, 583
 - calls aplTran, 365
 - calls identp[5], 365
 - calls postTransformCheck, 365
 - calls postTran, 365
 - defun, 365
- postTransformCheck, 369
 - calledby postTransform, 365
 - calls postcheck, 369
 - uses \$defOp, 369
- defun, 369
- postTuple, 398
 - calledby postComma, 382
 - calls postTranList, 398
 - defun, 398
- postTupleCollect, 398
 - calls postCollect, 398
 - defun, 398
- postType, 375
 - calledby postAtSign, 374
 - calledby postColon, 381
 - calledby postPretend, 392
 - calledby postSignature, 396
 - calls postTran, 375
 - calls unTuple, 375
 - defun, 375
- postWhere, 399
 - calls postTranList, 399
 - calls postTran, 399
 - defun, 399
- postWith, 400
 - calls postTran, 400
 - uses \$insidePostCategoryIfTrue, 400
 - defun, 400
- pp
 - calledby checkComments, 498
 - calledby compDefineFunctor1, 144
 - calledby optimizeFunctionDef, 224
- PredImplies
 - calledby compForm2, 610
- preparse, 75, 80
 - calledby preparse, 80
 - calledby spad, 574
 - calls ifcar, 80
 - calls parseprint, 80
 - calls preparse1, 80
 - calls preparse, 80
 - uses \$comblocklist, 80
 - uses \$constructorLineNumber, 80
 - uses \$docList, 80
 - uses \$headerDocumentation, 80
 - uses \$index, 80
 - uses \$maxSignatureLineNumber, 80
 - uses \$preparse-last-line, 80
 - uses \$preparseReportIfTrue, 80
 - uses \$skipme, 80

- defun, 80
- preparse-echo, 92
 - calledby fincomblock, 554
 - calledby preparse1, 84
 - local ref \$EchoLineStack, 92
 - uses Echo-Meta, 92
 - defun, 92
- preparse1, 84
 - calledby preparse, 80
 - calls doSystemCommand[5], 84
 - calls escaped, 84
 - calls fincomblock, 84
 - calls indent-pos, 84
 - calls is-console, 84
 - calls make-full-cvec, 84
 - calls maxindex, 84
 - calls parsepiles, 84
 - calls preparse-echo, 84
 - calls preparseReadLine, 84
 - calls strposl[5], 84
 - local def \$index, 84
 - local def \$preparse-last-line, 84
 - local def \$skipme, 84
 - local ref \$byConstructors, 84
 - local ref \$constructorsSeen, 84
 - local ref \$echolinestack, 84
 - local ref \$in-stream, 85
 - local ref \$index, 84
 - local ref \$linelist, 84
 - local ref \$preparse-last-line, 84
 - catches, 84
 - defun, 84
- preparseReadLine, 89
 - calledby preparse1, 84
 - calledby preparseReadLine, 89
 - calledby skip-to-endif, 556
 - calls initial-substring, 89
 - calls preparseReadLine1, 89
 - calls preparseReadLine, 89
 - calls skip-to-endif, 89
 - calls storeblanks, 89
 - calls string2BootTree, 89
 - local ref \$*eof*, 89
 - defun, 89
- preparseReadLine1, 90
 - calledby preparseReadLine1, 90
 - calledby preparseReadLine, 89
 - calledby skip-ifblock, 89
 - calledby skip-to-endif, 556
 - calls expand-tabs, 90
 - calls get-a-line, 90
 - calls maxindex, 90
 - calls preparseReadLine1, 90
 - calls strconc, 90
 - local def \$EchoLineStack, 90
 - local def \$index, 90
 - local def \$linelist, 90
 - local def \$preparse-last-line, 90
 - local ref \$index, 90
 - local ref \$linelist, 90
 - defun, 90
- pretend, 130, 324, 392
 - defplist, 130, 324, 392
- prettyprint
 - calledby optimize, 225
 - calledby s-process, 583
- PrimitivteArray
 - calledby optCallEval, 233
- primitiveType, 600
 - calledby compAtom, 597
 - uses \$DoubleFloat, 600
 - uses \$EmptyMode, 600
 - uses \$NegativeInteger, 600
 - uses \$NonNegativeInteger, 600
 - uses \$PositiveInteger, 600
 - uses \$String, 600
 - defun, 600
- print-defun, 587
 - calls is-console, 587
 - calls print-full, 587
 - uses \$PrettyPrint, 587
 - uses vmlisp::optionlist, 587
 - defun, 587
- print-full
 - calledby print-defun, 587
- print-package, 549
 - local ref \$out-stream, 549
 - defun, 549
- printSignature
 - calledby getSignature, 303
- printStats
 - calledby compile, 169

- prior-token, [94](#)
 - usedby PARSE-Expression, [425](#)
 - uses \$token, [94](#)
 - defvar, [94](#)
- processFunctor, [275](#)
 - calledby compCapsuleInner, [274](#)
 - calls buildFunctor, [275](#)
 - calls error, [275](#)
 - defun, [275](#)
- processInteractive[5]
 - called by s-process, [583](#)
- processSynonyms[5]
 - called by PARSE-NewExpr, [417](#)
- profileRecord
 - calledby compDefineCapsuleFunction, [151](#)
 - calledby setqSingle, [340](#)
- profileWrite
 - calledby finalizeLisplib, [203](#)
- purgeNewConstructorLines, [481](#)
 - calledby extendLocalLibdb, [477](#)
 - calls screenLocalLine, [481](#)
 - defun, [481](#)
- push-reduction, [468](#)
 - calledby PARSE-AnyId, [444](#)
 - calledby PARSE-Application, [432](#)
 - calledby PARSE-Category, [423](#)
 - calledby PARSE-CommandTail, [421](#)
 - calledby PARSE-Command, [418](#)
 - calledby PARSE-Conditional, [451](#)
 - calledby PARSE-Data, [442](#)
 - calledby PARSE-Enclosure, [438](#)
 - calledby PARSE-Exit, [449](#)
 - calledby PARSE-Expression, [425](#)
 - calledby PARSE-Expr, [426](#)
 - calledby PARSE-FloatBasePart, [437](#)
 - calledby PARSE-FloatBase, [436](#)
 - calledby PARSE-FloatExponent, [437](#)
 - calledby PARSE-FloatTok, [453](#)
 - calledby PARSE-Float, [435](#)
 - calledby PARSE-Form, [431](#)
 - calledby PARSE-Import, [425](#)
 - calledby PARSE-InfixWith, [423](#)
 - calledby PARSE-Infix, [429](#)
 - calledby PARSE-LabelExpr, [452](#)
 - calledby PARSE-Leave, [450](#)
 - calledby PARSE-LedPart, [426](#)
 - calledby PARSE-Loop, [452](#)
 - calledby PARSE-Name, [441](#)
 - calledby PARSE-NudPart, [426](#)
 - calledby PARSE-OpenBrace, [446](#)
 - calledby PARSE-OpenBracket, [446](#)
 - calledby PARSE-Prefix, [428](#), [429](#)
 - calledby PARSE-Primary1, [434](#)
 - calledby PARSE-PrimaryOrQM, [421](#)
 - calledby PARSE-Quad, [439](#)
 - calledby PARSE-Qualification, [430](#)
 - calledby PARSE-Reduction, [431](#)
 - calledby PARSE-Return, [449](#)
 - calledby PARSE-ScriptItem, [441](#)
 - calledby PARSE-Seg, [450](#)
 - calledby PARSE-Selector, [433](#)
 - calledby PARSE-SemiColon, [449](#)
 - calledby PARSE-Sequence1, [445](#)
 - calledby PARSE-Sequence, [445](#)
 - calledby PARSE-Sexpr1, [442](#)
 - calledby PARSE-SpecialCommand, [419](#)
 - calledby PARSE-Statement, [422](#)
 - calledby PARSE-Suffix, [448](#)
 - calledby PARSE-TokenCommandTail, [420](#)
 - calledby PARSE-TokenList, [420](#)
 - calledby PARSE-VarForm, [440](#)
 - calledby PARSE-With, [423](#)
 - calledby parse-argument-designator, [548](#)
 - calledby parse-identifier, [547](#)
 - calledby parse-keyword, [548](#)
 - calledby parse-number, [547](#)
 - calledby parse-spadstring, [546](#)
 - calledby parse-string, [546](#)
 - calledby star, [467](#)
 - calls make-reduction, [468](#)
 - calls stack-push, [468](#)
 - uses reduce-stack, [468](#)
 - defun, [468](#)
- put
 - calledby checkAndDeclare, [304](#)
 - calledby compColon, [290](#)
 - calledby compDefineCapsuleFunction, [151](#)
 - calledby compMacro, [323](#)
 - calledby compSubsetCategory, [348](#)
 - calledby compSuchthat, [349](#)
 - calledby compTypeOf, [596](#)
 - calledby doIt, [277](#)

- calledby evalAndSub, 265
- calledby getInverseEnvironment, 316
- calledby getSuccessEnvironment, 314
- calledby giveFormalParametersValues, 174
- calledby putDomainsInScope, 250
- calledby setqMultiple, 337
- calledby updateCategoryFrameForCategory, 117
- calledby updateCategoryFrameForConstructor, 116
- putDomainsInScope, 250
 - calledby addConstructorModemaps, 254
 - calledby augModemapsFromCategoryRep, 267
 - calledby augModemapsFromCategory, 260
 - calls delete, 250
 - calls getDomainsInScope, 250
 - calls member, 250
 - calls put, 250
 - calls say, 250
 - local def \$CapsuleDomainsInScope, 251
 - local ref \$insideCapsuleFunctionIfTrue, 251
 - defun, 250
- putInLocalDomainReferences, 185
 - calledby compile, 169
 - calls NRTputInTail, 185
 - local def \$elt, 185
 - local ref \$QuickCode, 185
 - defun, 185
- qslessp
 - calledby addDomain, 248
 - calledby getUniqueModemap, 259
- qsminus, 237
 - defplist, 237
- quote, 325, 392
 - defplist, 325, 392
- quote-if-string, 456
 - calledby match-advance-string, 455
 - calledby unget-tokens, 458
 - calls escape-keywords, 456
 - calls pack, 456
 - calls strconc, 456
 - calls token-nonblank, 456
 - calls token-symbol, 456
- calls token-type, 456
- calls underscore, 456
- uses \$boot, 456
- uses \$spad, 456
- defun, 456
- quotify
 - calledby compIf, 311
- assoc
 - calledby checkBalance, 501
 - calledby modemapPattern, 199
- rbp
 - usedby PARSE-NudPart, 427
 - usedby PARSE-Operation, 427
- redefine-stream
 - calledby compileDocumentation, 165
 - calledby writeLib1, 203
- read-a-line, 631
 - calledby get-a-line, 638
 - calledby read-a-line, 631
 - calls Line-New-Line, 631
 - calls read-a-line, 631
 - calls subseq, 631
 - uses *eof*, 631
 - uses File-Closed, 631
 - defun, 631
- readline
 - calledby dbReadLines, 481
- recompile-lib-file-if-necessary, 627
 - calledby compileSpadLispCmd, 568
 - calls compile-lib-file, 627
 - uses *lisp-bin-filetype*, 627
 - defun, 627
- Record, 284
 - defplist, 284
- recordAttributeDocumentation, 470
 - calledby PARSE-Category, 424
 - calls ifcdr, 470
 - calls opOf, 470
 - calls pname, 470
 - calls postTransform, 470
 - calls recordDocumentation, 470
 - calls upper-case-p, 470
 - defun, 470
- RecordCategory, 296
 - defplist, 296

- recordcopy, 247
 - defplist, 247
- recordDocumentation, 471
 - calledby recordAttributeDocumentation, 470
 - calledby recordSignatureDocumentation, 470
 - calls collectComBlock, 471
 - calls recordHeaderDocumentation, 471
 - local def \$docList, 471
 - local def \$maxSignatureLineNumber, 471
 - defun, 471
- recordelt, 245
 - defplist, 245
- recordHeaderDocumentation, 472
 - calledby postDef, 384
 - calledby recordDocumentation, 471
 - calls assocright, 472
 - local def \$comblocklist, 472
 - local def \$headerDocumentation, 472
 - local ref \$comblocklist, 472
 - local ref \$headerDocumentation, 472
 - local ref \$maxSignatureLineNumber, 472
 - defun, 472
- recordSignatureDocumentation, 470
 - calledby PARSE-Category, 424
 - calls postTransform, 470
 - calls recordDocumentation, 470
 - defun, 470
- reduce, 326, 393
 - defplist, 326, 393
- reduce-stack, 468
 - usedby push-reduction, 468
 - uses \$stack, 468
 - defvar, 468
- reduce-stack-clear, 468
 - defmacro, 468
- reduction, 96
 - defstruct, 96
- reduction-value
 - calledby nth-stack, 551
 - calledby pop-stack-1, 550
 - calledby pop-stack-2, 550
 - calledby pop-stack-3, 551
 - calledby pop-stack-4, 551
- refvecp
 - calledby translabel1, 543
- remdup
 - calledby compDefineFunctor1, 144
 - calledby displayPreCompilationErrors, 544
 - calledby finalizeDocumentation, 492
 - calledby getSignature, 302
 - calledby hasSigInTargetCategory, 304
 - calledby mkAListOfExplicitCategoryOps, 189
 - calledby mkExplicitCategoryFunction, 288
 - calledby orderByDependency, 223
- removeBackslashes, 541
 - calledby checkGetParse, 522
 - calledby removeBackslashes, 541
 - calls charPosition, 541
 - calls length, 541
 - calls removeBackslashes, 541
 - calls strconc, 541
 - local ref \$charBack, 541
 - defun, 541
- removeEnv
 - calledby compApply, 595
 - calledby getSuccessEnvironment, 315
 - calledby setqSingle, 340
- removeSuperfluousMapping, 396
 - calledby postSignature, 396
 - defun, 396
- removeZeroOne
 - calledby compDefineCategory2, 160
 - calledby compDefineFunctor1, 145
 - calledby finalizeLisplib, 203
- remprop
 - calledby unloadOneConstructor, 201
- repeat, 328, 394
 - defplist, 328, 394
- replaceExitEsc
 - calledby coerceExit, 357
- replaceExitEtc, 333
 - calledby compDefineCapsuleFunction, 152
 - calledby compSeq1, 332
 - calledby replaceExitEtc, 333
 - calls convertOrCroak, 333
 - calls intersectionEnvironment, 333
 - calls replaceExitEtc, 333
 - calls rplac, 333
 - local def \$finalEnv, 333

- local ref \$finalEnv, 333
 - defun, 333
- replaceFile
 - calledby compileDocumentation, 165
 - calledby lisplibDoRename, 201
- replaceVars, 192
 - calledby interactiveModemapForm, 191
 - defun, 192
- reportOnFunctorCompilation, 215
 - calledby compDefineFunctor1, 145
 - calls addStats, 215
 - calls displayMissingFunctions, 215
 - calls displaySemanticErrors, 215
 - calls displayWarnings, 215
 - calls normalizeStatAndStringify, 215
 - calls sayBrightly, 215
 - uses \$functionStats, 215
 - uses \$functorStats, 215
 - uses \$op, 215
 - uses \$semanticErrorStack, 215
 - uses \$warningStack, 215
 - defun, 215
- resolve, 361
 - calledby coerceExit, 357
 - calledby compApplication, 605
 - calledby compApply, 595
 - calledby compCategory, 286
 - calledby compCoerce1, 359
 - calledby compConstructorCategory, 297
 - calledby compDefineCapsuleFunction, 152
 - calledby compIf, 311
 - calledby compReturn, 331
 - calledby compString, 345
 - calledby convert, 600
 - calledby modifyModeStack, 625
 - calls mkUnion, 361
 - calls modeEqual, 361
 - local ref \$EmptyMode, 361
 - local ref \$NoValueMode, 361
 - local ref \$String, 361
 - defun, 361
- return, 131, 331
 - defplist, 131, 331
- rpacfile
 - calledby compDefineLisplib, 163
 - calledby compileDocumentation, 165
- rplac
 - calledby coerce, 351
 - calledby getAbbreviation, 298
 - calledby optCall, 229
 - calledby optCatch, 240
 - calledby optCond, 242
 - calledby optSpecialCall, 232
 - calledby optXLAMCond, 226
 - calledby optimizeFunctionDef, 223
 - calledby optimize, 225
 - calledby replaceExitEtc, 333
- rplaca
 - calledby NRTgetLocalIndex, 211
 - calledby NRTputInTail, 185
 - calledby doItIf, 281
 - calledby optPackageCall, 230
 - calledby optSpecialCall, 232
- rplacd
 - calledby doItIf, 281
 - calledby optCond, 242
 - calledby optPackageCall, 230
- rplacw
 - calledby optSpecialCall, 232
- rshut
 - calledby compDefineLisplib, 163
 - calledby compileDocumentation, 165
- rwrite
 - calledby compilerDoitWithScreenedLisplib, 569
- rwrite128
 - calledby lisplibWrite, 209
- rwriteLispForm, 200
 - calledby evalAndRwriteLispForm, 200
 - local ref \$libFile, 200
 - local ref \$lisplib, 200
 - defun, 200
- s-process, 576
 - calledby spad, 574
 - calls compTopLevel, 583
 - calls curstrm, 583
 - calls def-process, 583
 - calls def-rename, 583
 - calls displayPreCompilationErrors, 583
 - calls displaySemanticErrors, 583
 - calls get-internal-run-time, 583

- calls new2OldLisp, 583
- calls parseTransform, 583
- calls postTransform, 583
- calls prettyprint, 583
- calls processInteractive[5], 583
- calls terpri, 583
- uses \$DomainFrame, 584
- uses \$EmptyEnvironment, 584
- uses \$EmptyMode, 583
- uses \$Index, 583
- uses \$InteractiveFrame, 584
- uses \$LocalFrame, 584
- uses \$PolyMode, 583
- uses \$PrintOnly, 584
- uses \$TranslateOnly, 584
- uses \$Translation, 584
- uses \$VariableCount, 584
- uses \$compUniquelyIfTrue, 583
- uses \$currentFunction, 583
- uses \$currentLine, 584
- uses \$exitModeStack, 584
- uses \$exitMode, 584
- uses \$e, 584
- uses \$form, 584
- uses \$genFVar, 584
- uses \$genSDVar, 584
- uses \$insideCapsuleFunctionIfTrue, 584
- uses \$insideCategoryIfTrue, 584
- uses \$insideCoerceInteractiveHardIfTrue, 584
- uses \$insideExpressionIfTrue, 584
- uses \$insideFunctorIfTrue, 584
- uses \$insideWhereIfTrue, 584
- uses \$leaveLevelStack, 584
- uses \$leaveMode, 584
- uses \$macroassoc, 583
- uses \$newspad, 583
- uses \$postStack, 584
- uses \$previousTime, 584
- uses \$returnMode, 584
- uses \$semanticErrorStack, 584
- uses \$stop-level, 584
- uses \$stopOp, 584
- uses \$warningStack, 584
- uses curoutstream, 584
- defun, 576
- say
 - calledby canReturn, 312
 - calledby compOrCroak1, 589
 - calledby getSignature, 302
 - calledby modifyModeStack, 625
 - calledby optimize, 225
 - calledby orderByDependency, 222
 - calledby putDomainsInScope, 250
- sayBrightly
 - calledby NRTgetLookupFunction, 210
 - calledby checkAndDeclare, 304
 - calledby checkDocError, 517
 - calledby compDefineCapsuleFunction, 152
 - calledby compDefineFunctor1, 144
 - calledby compMacro, 323
 - calledby compilerDoit, 570
 - calledby compile, 169
 - calledby displayMissingFunctions, 216
 - calledby displayPreCompilationErrors, 544
 - calledby doIt, 277
 - calledby reportOnFunctorCompilation, 215
 - calledby spadCompileOrSetq, 182
 - calledby transDocList, 495
 - calledby transformAndRecheckComments, 497
- saybrightly1
 - calledby checkDocError, 517
- sayBrightlyI
 - calledby optimizeFunctionDef, 223
- sayBrightlyNT
 - calledby NRTgetLookupFunction, 210
- sayKeyedMsg
 - calledby finalizeDocumentation, 492
- sayKeyedMsg[5]
 - called by compileSpad2Cmd, 566
 - called by compileSpadLispCmd, 568
- sayMath
 - calledby displayPreCompilationErrors, 544
- sayMSG
 - calledby compDefineLisplib, 163
 - calledby finalizeDocumentation, 492
 - calledby finalizeLisplib, 203
- ScanOrPairVec[5]
 - called by hasFormalMapVariable, 623
- screenLocalLine, 486
 - calledby purgeNewConstructorLines, 481

- calls charPosition, 486
 - calls dbKind, 486
 - calls dbName, 486
 - calls dbPart, 486
 - defun, 486
- Scripts, 394
 - defplist, 394
- segment, 131, 132
 - defplist, 131, 132
- selectOptionLC[5]
 - called by compileSpad2Cmd, 566
 - called by compileSpadLispCmd, 568
 - called by compiler, 563
- seq, 234, 332
 - defplist, 234, 332
- setDefOp, 400
 - calledby parseDEF, 105
 - calledby postcheck, 369
 - uses \$defOp, 400
 - uses \$stopOp, 400
 - defun, 400
- setdifference
 - calledby orderPredTran, 194
- setelt
 - calledby checkAddPeriod, 529
 - calledby modifyModeStack, 625
- seteltModemapFilter, 609
 - calls isConstantId, 609
 - calls stackMessage, 609
 - defun, 609
- setq, 335
 - defplist, 335
- setqMultiple, 337
 - calledby compSetq1, 336
 - calls addBinding, 337
 - calls compSetq1, 337
 - calls convert, 337
 - calls genSomeVariable, 337
 - calls genVariable, 337
 - calls length, 337
 - calls mkprogn, 337
 - calls nreverse0, 337
 - calls put, 337
 - calls setqMultipleExplicit, 337
 - calls stackMessage, 337
 - local ref \$EmptyMode, 337
 - local ref \$NoValueMode, 337
 - local ref \$noEnv, 337
 - defun, 337
- setqMultipleExplicit, 339
 - calledby setqMultiple, 337
 - calls compSetq1, 339
 - calls genVariable, 339
 - calls last, 339
 - calls stackMessage, 339
 - local ref \$EmptyMode, 339
 - local ref \$NoValueMode, 339
 - defun, 339
- setqSetelt, 340
 - calledby compSetq1, 336
 - calls comp, 340
 - defun, 340
- setqSingle, 340
 - calledby compSetq1, 335
 - calls NRTassocIndex, 340
 - calls addBinding[5], 340
 - calls assignError, 340
 - calls augModemapsFromDomain1, 340
 - calls comp, 340
 - calls consProplistOf, 340
 - calls convert, 340
 - calls getProplist[5], 340
 - calls getmode, 340
 - calls get, 340
 - calls identp[5], 340
 - calls isDomainForm, 340
 - calls isDomainInScope, 340
 - calls maxSuperType, 340
 - calls outputComp, 340
 - calls profileRecord, 340
 - calls removeEnv, 340
 - calls stackWarning, 340
 - uses \$EmptyMode, 341
 - uses \$NoValueMode, 341
 - uses \$QuickLet, 340
 - uses \$form, 341
 - uses \$insideSetqSingleIfTrue, 340
 - uses \$profileCompiler, 341
 - defun, 340
- setrecordelt, 246
 - defplist, 246
- shut

- calledby buildLibdb, 478
- calledby dbWriteLines, 481
- calledby whoOwns, 541
- shut[5]
 - called by spad, 574
- Signature, 396
 - defplist, 396
- signatureTran, 193
 - calledby orderPredicateItems, 193
 - calledby signatureTran, 193
 - calls isCategoryForm, 193
 - calls signatureTran, 193
 - local ref \$e, 193
 - defun, 193
- simpBool
 - calledby compDefineFunctor1, 145
- skip-blanks, 454
 - calledby match-string, 454
 - calls advance-char, 454
 - calls current-char, 454
 - calls token-lookahead-type, 454
 - defun, 454
- skip-ifblock, 89
 - calledby skip-ifblock, 89
 - calls initial-substring, 89
 - calls preparseReadLine1, 89
 - calls skip-ifblock, 89
 - calls storeblanks, 89
 - calls string2BootTree, 89
 - defun, 89
- skip-to-endif, 556
 - calledby preparseReadLine, 89
 - calledby skip-to-endif, 556
 - calls initial-substring, 556
 - calls preparseReadLine1, 556
 - calls preparseReadLine, 556
 - calls skip-to-endif, 556
 - defun, 556
- SourceLevelSubsume
 - calledby getSignature, 303
- spad, 573
 - calls PARSE-NewExpr, 574
 - calls addBinding[5], 574
 - calls init-boot/spad-reader[5], 574
 - calls initialize-preparse, 574
 - calls ioclear, 574
 - calls makeInitialModemapFrame[5], 574
 - calls pop-stack-1, 574
 - calls preparse, 574
 - calls s-process, 574
 - calls shut[5], 574
 - uses *comp370-apply*, 575
 - uses *eof*, 575
 - uses *fileactq-apply*, 575
 - uses /editfile, 575
 - uses \$InitialDomainsInScope, 575
 - uses \$InteractiveFrame, 574
 - uses \$InteractiveMode, 575
 - uses \$boot, 575
 - uses \$noSubsumption, 574
 - uses \$spad, 575
 - uses boot-line-stack, 575
 - uses curoutstream, 575
 - uses echo-meta, 575
 - uses file-closed, 575
 - uses line, 575
 - uses optionlist, 575
 - catches, 574, 575
 - defun, 573
- spad-fixed-arg, 627
 - defun, 627
- spad2AsTranslatorAutoloadOnceTrigger
 - calledby compileSpad2Cmd, 566
- spadcall, 239
 - defplist, 239
- spadCompileOrSetq, 182
 - calledby compile, 169
 - calls LAM,EVALANDFILEACTQ, 182
 - calls bright, 182
 - calls compileConstructor, 182
 - calls comp, 182
 - calls contained, 182
 - calls mkq, 182
 - calls sayBrightly, 182
 - local ref \$insideCapsuleFunctionIfTrue, 182
 - defun, 182
- SpadInterpretStream[5]
 - called by ncINTERPFILE, 627
- spadPrompt
 - calledby compileSpad2Cmd, 566
 - calledby compileSpadLispCmd, 568

- spadreduce
 - calledby floatexpid, 465
- spadSysChoose
 - calledby checkRecordHash, 509
- splitEncodedFunctionName, 180
 - calledby compile, 169
 - calls stringimage, 180
 - calls strpos, 180
 - defun, 180
- stack, 92
 - deconstruct, 92
- stack-/empty, 93
 - uses \$stack, 93
 - defmacro, 93
- stack-clear, 93
 - uses \$stack, 93
 - defun, 93
- stack-load, 92
 - uses \$stack, 92
 - defun, 92
- stack-pop, 94
 - calledby Pop-Reduction, 552
 - uses \$stack, 94
 - defun, 94
- stack-push, 93
 - calledby pop-stack-2, 550
 - calledby pop-stack-3, 551
 - calledby pop-stack-4, 551
 - calledby push-reduction, 468
 - uses \$stack, 93
 - defun, 93
- stack-size
 - calledby star, 467
- stack-store
 - calledby nth-stack, 551
- stackAndThrow
 - calledby compDefine1, 141
 - calledby compInternalFunction, 155
 - calledby compLambda, 321
 - calledby compWithMappingMode1, 617
 - calledby getSignatureFromMode, 299
- stackMessage
 - calledby assignError, 342
 - calledby augModemapsFromDomain1, 253
 - calledby autoCoerceByModemap, 360
 - calledby coerce, 352
 - calledby compElt, 306
 - calledby compMapCond", 257
 - calledby compMapCond', 257
 - calledby compRepeatOrCollect, 329
 - calledby compSymbol, 601
 - calledby eltModemapFilter, 608
 - calledby getFormModemaps, 607
 - calledby getOperationAlist, 265
 - calledby seteltModemapFilter, 609
 - calledby setqMultipleExplicit, 339
 - calledby setqMultiple, 337
- stackMessageIfNone
 - calledby compExit, 308
 - calledby compForm, 602
- stackSemanticError
 - calledby compColonInside, 596
 - calledby compJoin, 319
 - calledby compOrCroak1, 589
 - calledby compPretend, 325
 - calledby compReturn, 331
 - calledby compSubDomain1, 346
 - calledby constructMacro, 182
 - calledby doIt, 277
 - calledby getArgumentModeOrMoan, 187
 - calledby getSignature, 303
 - calledby getTargetFromRhs, 173
 - calledby unknownTypeError, 249
- stackWarning
 - calledby compColonInside, 596
 - calledby compElt, 306
 - calledby compPretend, 325
 - calledby doIt, 277
 - calledby getUniqueModemap, 259
 - calledby hasSigInTargetCategory, 305
 - calledby setqSingle, 340
- star, 467
 - calledby PARSE-Application, 432
 - calledby PARSE-Category, 424
 - calledby PARSE-CommandTail, 421
 - calledby PARSE-Expr, 426
 - calledby PARSE-Import, 425
 - calledby PARSE-IteratorTail, 447
 - calledby PARSE-Loop, 452
 - calledby PARSE-Option, 422
 - calledby PARSE-ScriptItem, 441
 - calledby PARSE-Sexpr1, 443

- calledby PARSE-SpecialCommand, 419
- calledby PARSE-Statement, 422
- calledby PARSE-TokenCommandTail, 420
- calledby PARSE-TokenList, 420
- calls pop-stack-1, 467
- calls push-reduction, 467
- calls stack-size, 467
- defmacro, 467
- step
 - calledby floatexpid, 465
- storeblanks, 637
 - calledby preparseReadLine, 89
 - calledby skip-ifblock, 89
 - defun, 637
- strconc
 - calledby buildLibOp, 484
 - calledby buildLibdbConEntry, 482
 - calledby buildLibdbString, 480
 - calledby checkAddBackSlashes, 527
 - calledby checkAddIndented, 519
 - calledby checkAddSpaceSegments, 529
 - calledby checkComments, 498
 - calledby checkIndentedLines, 524
 - calledby checkTransformFirst, 513
 - calledby compApplication, 605
 - calledby compDefine1, 141
 - calledby compDefineFunctor1, 144
 - calledby compileSpad2Cmd, 566
 - calledby compile, 169
 - calledby decodeScripts, 405
 - calledby finalizeDocumentation, 492
 - calledby getCaps, 181
 - calledby mkCategoryPackage, 176
 - calledby preparseReadLine1, 90
 - calledby quote-if-string, 456
 - calledby removeBackslashes, 541
 - calledby unget-tokens, 458
 - calledby whoOwns, 541
- String, 345
 - defplist, 345
- string-not-greaterp
 - calledby initial-substring-p, 456
- string2BootTree, 71
 - calledby preparseReadLine, 89
 - calledby skip-ifblock, 89
 - calls def-rename, 71
 - calls new2OldLisp, 71
 - local def \$boot, 72
 - local def \$spad, 72
 - uses boot-line-stack, 71
 - uses line-handler, 71
 - uses xtokenreader, 71
 - defun, 71
- string2id-n
 - calledby infixtok, 555
- stringimage
 - calledby buildLibAttr, 485
 - calledby buildLibOp, 484
 - calledby buildLibdbString, 480
 - calledby checkAddIndented, 519
 - calledby encodeFunctionName, 179
 - calledby encodeItem, 181
 - calledby finalizeDocumentation, 492
 - calledby getCaps, 181
 - calledby splitEncodedFunctionName, 180
 - calledby substituteCategoryArguments, 253
- stringPrefix?
 - calledby checkGetArgs, 521
 - calledby comp3, 592
- stripOffArgumentConditions, 302
 - calledby compDefineCapsuleFunction, 151
 - local def \$argumentConditionList, 302
 - local ref \$argumentConditionList, 302
 - defun, 302
- stripOffSubdomainConditions, 301
 - calledby compDefineCapsuleFunction, 151
 - calls assoc, 301
 - calls mkpf, 301
 - local def \$argumentConditionList, 301
 - local ref \$argumentConditionList, 301
 - defun, 301
- stripUnionTags
 - calledby augModemapsFromDomain, 252
- strpos
 - calledby splitEncodedFunctionName, 180
- strposl[5]
 - called by preparse1, 84
- SubDomain, 346
 - defplist, 346
- sublis
 - calledby NRTgetLookupFunction, 210
 - calledby applyMapping, 593

- calledby augLisplibModemapsFromCategory, 187
- calledby coerceable, 356
- calledby compApplyModemap, 255
- calledby compDefineCategory2, 159
- calledby compDefineFunctor1, 144
- calledby compForm2, 610
- calledby doIt, 277
- calledby formal2Pattern, 214
- calledby getModemap, 254
- calledby mkOpVec, 221
- calledby substituteCategoryArguments, 253
- calledby substituteIntoFunctorModemap, 614
- sublislis
 - calledby buildLibAttr, 485
 - calledby buildLibOp, 484
 - calledby compHasFormat, 309
 - calledby finalizeDocumentation, 492
 - calledby mkCategoryPackage, 176
- subrname, 228
 - calledby optimize, 225
 - calls bpiname, 228
 - calls compiled-function-p, 228
 - calls identp, 228
 - calls mbpip, 228
 - defun, 228
- subseq
 - calledby match-string, 454
 - calledby read-a-line, 631
- SubsetCategory, 348
 - defplist, 348
- substituteCategoryArguments, 253
 - calledby augModemapsFromDomain1, 253
 - calls internl, 253
 - calls stringimage, 253
 - calls sublis, 253
 - defun, 253
- substituteIntoFunctorModemap, 614
 - calledby compFocompFormWithModemap, take 613
 - calls compOrCroak, 614
 - calls eqsubstlist, 614
 - calls keyedSystemError, 614
 - calls sublis, 614
 - defun, 614
- substNames, 266
 - calledby evalAndSub, 265
 - calledby genDomainOps, 220
 - calls eqsubstlist, 266
 - calls isCategoryPackageName, 266
 - calls nreverse0, 266
 - uses \$FormalMapVariableList, 266
 - defun, 266
- substring?
 - calledby checkBeginEnd, 502
 - calledby checkExtract, 520
- substVars, 198
 - calledby interactiveModemapForm, 191
 - calls contained, 198
 - calls nsubst, 198
 - local ref \$FormalMapVariableList, 198
 - defun, 198
- suffix
 - calledby addclose, 552
- systemCommand[5]
 - called by PARSE-CommandTail, 421
 - called by PARSE-TokenCommandTail, 420
- systemError
 - calledby checkTrim, 516
 - calledby compReduce1, 326
 - calledby errhuh, 409
 - calledby getOperationAlist, 265
- systemErrorHere
 - calledby addArgumentConditions, 300
 - calledby addEltModemap, 261
 - calledby canReturn, 312
 - calledby compCategory, 286
 - calledby compColon, 290
 - calledby getSlotFromCategoryForm, 206
 - calledby getSlotFromFunctor, 208
 - calledby optCall, 229
 - calledby postIn, 389
 - calledby postin, 388
- calledby compColon, 290
- calledby compDefineCategory2, 159
- calledby compForm2, 610
- calledby compHasFormat, 309
- calledby compWithMappingModel, 617
- calledby drop, 553

- calledby getSignatureFromMode, 299
 - calledby getSlotFromCategoryForm, 206
- terminateSystemCommand[5]
 - called by compileSpad2Cmd, 566
 - called by compileSpadLispCmd, 568
- terpri
 - calledby s-process, 583
- throwKeyedMsg
 - calledby compileSpad2Cmd, 566
 - calledby compileSpadLispCmd, 568
 - calledby compiler, 563
 - calledby loadLibIfNecessary, 115
- throwkeyedmsg
 - calledby postMDef, 391
- throws
 - compForm3, 612
- tmptok, 416
 - usedby PARSE-Operation, 427
- defvar, 416
- tok, 416
 - usedby PARSE-GlyphTok, 444
 - usedby PARSE-NBGlyphTok, 443
- defvar, 416
- token, 94
 - destructure, 94
- token-install, 96
 - uses \$token, 96
- defun, 96
- token-lookahead-type, 455
 - calledby skip-blanks, 454
 - uses Escape-Character, 455
- defun, 455
- token-nonblank
 - calledby PARSE-FloatBasePart, 437
 - calledby quote-if-string, 456
 - calledby unget-tokens, 459
- token-print, 96
 - uses \$token, 96
- defun, 96
- token-symbol
 - calledby PARSE-SpecialKeyword, 418
 - calledby match-token, 460
 - calledby parse-argument-designator, 548
 - calledby parse-identifier, 547
 - calledby parse-keyword, 548
 - calledby parse-number, 547
 - calledby parse-spadstring, 546
 - calledby parse-string, 546
 - calledby quote-if-string, 456
- token-type
 - calledby match-token, 460
 - calledby quote-if-string, 456
- TPDHERE
 - Note that this function was missing without error, so may be junk, 519
 - See LocalAlgebra for an example call, 348
 - The use of and in spadreduce is undefined. rewrite this to loop, 465
 - This function is used but never defined. Remove it., 185
 - test with BASTYPE, 172
- transDoc, 496
 - calledby transDocList, 495
 - calls checkDocError1, 496
 - calls checkExtract, 496
 - calls checkTrim, 496
 - calls nreverse, 496
 - calls transformAndRecheckComments, 496
 - local def \$argl, 496
 - local def \$attribute?, 496
 - local def \$x, 496
 - local ref \$attribute?, 496
 - local ref \$x, 496
 - defun, 496
- transDocList, 495
 - calledby finalizeDocumentation, 492
 - calls checkDocError1, 495
 - calls checkDocError, 495
 - calls sayBrightly, 495
 - calls transDoc, 495
 - local ref \$constructorName, 495
 - defun, 495
- transformAndRecheckComments, 497
 - calledby transDoc, 496
 - calls checkComments, 497
 - calls checkRewrite, 497
 - calls sayBrightly, 497
 - local def \$checkingXmptex?, 497
 - local def \$exposeFlagHeading, 497
 - local def \$name, 497
 - local def \$origin, 497

- local def \$recheckingFlag, 497
 - local def \$x, 497
 - local ref \$exposeFlagHeading, 497
 - defun, 497
- transformOperationAlist, 206
 - calledby getCategoryOpsAndAtts, 205
 - calledby getFunctorOpsAndAtts, 208
 - calls assoc, 207
 - calls insertAlist, 207
 - calls keyedSystemError, 207
 - calls lassq, 207
 - calls member, 207
 - local ref \$functionLocations, 207
 - defun, 206
- transImplementation, 599
 - calledby compAtomWithModemap, 598
 - calls genDeltaEntry, 599
 - defun, 599
- transIs, 106
 - calledby parseIsnt, 124
 - calledby parseIs, 123
 - calledby parseLET, 126
 - calledby parseLhs, 106
 - calledby transIs1, 106
 - calls isListConstructor, 106
 - calls transIs1, 106
 - defun, 106
- transIs1, 106
 - calledby transIs1, 106
 - calledby transIs, 106
 - calls nreverse0, 106
 - calls transIs1, 106
 - calls transIs, 106
 - defun, 106
- translabel, 543
 - calledby PARSE-Data, 442
 - calls translabel1, 543
 - defun, 543
- translabel1, 543
 - calledby translabel1, 543
 - calledby translabel, 543
 - calls lassoc, 543
 - calls maxindex, 543
 - calls refvecp, 543
 - calls translabel1, 543
 - defun, 543
- transSeq
 - calledby parseSeq, 132
- trimString
 - calledby checkGetArgs, 521
- TruthP, 264
 - calledby mergeModemap, 263
 - calledby optCond, 242
 - defun, 264
- try-get-token, 461
 - calledby advance-token, 462
 - calledby current-token, 461
 - calledby next-token, 462
 - calls get-token, 461
 - uses valid-tokens, 461
 - defun, 461
- tuple2List, 549
 - calledby postCollect,finish, 378
 - calledby postConstruct, 383
 - calledby postInSeq, 388
 - calledby tuple2List, 549
 - calls postTranSegment, 549
 - calls postTran, 549
 - calls tuple2List, 549
 - uses \$InteractiveMode, 549
 - uses \$boot, 549
 - defun, 549
- TupleCollect, 398
 - defplist, 398
- unabbrevAndLoad
 - calledby parseHasRhs, 114
 - calledby parseHas, 112
- unAbbreviateKeyword[5]
 - called by PARSE-SpecialKeyWord, 418
- uncons, 336
 - calledby uncons, 336
 - calls uncons, 336
 - defun, 336
- Undef
 - usedby mkOpVec, 221
- underscore, 458
 - calledby quote-if-string, 456
 - calls vector-push, 458
 - defun, 458
- unembed

- calledby compilerDoitWithScreenedLisplibupdateCategoryFrameForCategory, 117
- 569
- unErrorRef
 - calledby addModemap1, 270
- unget-tokens, 458
 - calledby match-string, 454
 - calls line-current-segment, 458
 - calls line-new-line, 459
 - calls line-number, 459
 - calls quote-if-string, 458
 - calls strconc, 458
 - calls token-nonblank, 459
 - uses valid-tokens, 459
 - defun, 458
- Union, 285
 - defplist, 285
- union
 - calledby compDefWhereClause, 156
 - calledby compJoin, 319
 - calledby extendLocalLibdb, 477
 - calledby makeFunctorArgumentParameters, 217
 - calledby mkAlistOfExplicitCategoryOps, 189
 - calledby mkExplicitCategoryFunction, 288
 - calledby mkUnion, 362
- UnionCategory, 297
 - defplist, 297
- unionq
 - calledby freelist, 625
 - calledby orderPredTran, 194
- unknownTypeError, 249
 - calledby addDomain, 248
 - calledby compColon, 290
 - calls stackSemanticError, 249
 - defun, 249
- unloadOneConstructor, 201
 - calledby compDefineLisplib, 163
 - calls mkAutoLoad, 201
 - calls remprop, 201
 - defun, 201
- unTuple, 409
 - calledby postMapping, 390
 - calledby postTran, 366
 - calledby postType, 375
 - defun, 409
- calledby compDefineLisplib, 163
- calledby isFunctor, 249
- calledby loadLibIfNecessary, 115
- calls addModemap, 117
- calls getdatabase, 117
- calls put, 117
- local def \$CategoryFrame, 117
- local ref \$CategoryFrame, 117
- defun, 117
- updateCategoryFrameForConstructor, 116
 - calledby compDefineLisplib, 163
 - calledby isFunctor, 249
 - calledby loadLibIfNecessary, 115
 - calls addModemap, 116
 - calls convertOpAlist2compilerInfo, 116
 - calls getdatabase, 116
 - calls put, 116
 - local def \$CategoryFrame, 116
 - local ref \$CategoryFrame, 116
 - defun, 116
- updateSourceFiles[5]
 - called by compileSpad2Cmd, 566
- upper-case-p
 - calledby recordAttributeDocumentation, 470
- userError
 - calledby compDefWhereClause, 156
 - calledby compOrCroak1, 589
 - calledby compReturn, 331
 - calledby compile, 169
 - calledby convertOrCroak, 334
 - calledby doItIf, 281
 - calledby orderByDependency, 223
- valid-tokens, 95
 - usedby advance-token, 462
 - usedby current-token, 461
 - usedby next-token, 462
 - usedby try-get-token, 461
 - usedby unget-tokens, 459
 - uses \$token, 95
 - defvar, 95
- vcons, 132
 - defplist, 132
- Vector

- calledby optCallEval, 233
- vector, 349
 - defplist, 349
- vector-push
 - calledby underscore, 458
- VectorCategory, 297
 - defplist, 297
- vmisp::optionlist
 - usedby print-defun, 587
- where, 133, 350, 399
 - defplist, 133, 350, 399
- whoOwns, 541
 - calledby checkDocMessage, 520
 - calls awk, 541
 - calls getdatabase, 541
 - calls shut, 541
 - calls strconc, 541
 - local ref \$exposeFlag, 541
 - defun, 541
- with, 399
 - defplist, 399
- wrapDomainSub, 290
 - calledby compJoin, 319
 - calledby mkExplicitCategoryFunction, 288
 - defun, 290
- wrapSEQExit
 - calledby makeSimplePredicateOrNil, 546
- writedb
 - calledby buildLibAttr, 485
 - calledby buildLibOp, 484
 - calledby buildLibdb, 478
 - calledby dbWriteLines, 481
- writeLib1, 203
 - calledby initializeLisplib, 202
 - calls rdefiostream, 203
 - defun, 203
- XTokenReader, 463
 - calledby get-token, 463
 - usedby get-token, 463
 - defvar, 463
- xtokenreader
 - usedby string2BootTree, 71