axiom™

The 30 Year Horizon

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

Volume 9: Axiom Compiler

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

Cyril Alberga
George Andrews
Yurij Baransky
Gilbert Baumslag
Mark Botch
Martin Brock
William Burge
Bruce Char
Gregory V. Chudnovsky
Don Coppersmith
Gary Cornell
Timothy Daly Sr.
Jean Della Dora
Claire DiCrescendo
Martin Dunstan
Robert Edwards
Richard Fateman
Brian Ford
Constantine Frangos
Marc Gaetano
Patricia Gianni
Laureano Gonzalez-Vega
Matt Grayson
Oswald Gschnitzer
Vilya Harvey
Ralf Hemmecke
Pietro Iglio
Grant Keady
Klaus Kusche
Frederic Lehobey
Rudiger Loos
Camm Maguire
Ian Meikle
Gerard Milmeister
Michael Monagan
Mark Murray
John Nelder
Jinzhong Niu
Julian A. Padget
Susan Pelzel
Claude Quitte
Renaud Rioboo
Simon Robinson
Philip Santas
Gerhard Schneider
Fritz Schwarz
Elena Smirnova
Robert Sutor
James Thatcher
Dylan Thurston
Gregory Vanuxem
Juergen Weiss
James Wen
John M. Wiley
Stephen Wilson
Sandra Wityak

Roy Adler
Henry Baker
David R. Barton
Fred Blair
Alexandre Bouyer
Manuel Bronstein
Quentin Carpent
Cheekai Chin
Josh Cohen
George Corliss
Meino Cramer
Timothy Daly Jr.
Gabriel Dos Reis
Sam Dooley
Brian Dupee
Heow Eide-Goodman
Bertfried Fauser
Albrecht Fortenbacher
Timothy Freeman
Rudiger Gebauer
Holger Gollan
Stephen Gortler
James Griesmer
Jocelyn Guidry
Satoshi Hamaguchi
Henderson
Richard Jenks
Tony Kennedy
Bernhard Kutzler
Michel Levaud
Michael Lucks
Bob McElrath
David Mentre
Mohammed Mobarak
Marc Moreno-Maza
William Naylor
Godfrey Nolan
Michael O'Connor
Bill Page
Michel Petitot
Norman Ramsey
Jean Rivlin
Michael Rothstein
Alfred Scheerhorn
Martin Schoenert
Nick Simicich
Jonathan Steinbach
Moss E. Sweedler
Baldir Thomas
Barry Trager
Bernhard Wall
M. Weller
Thorsten Werther
Berhard Will
Shmuel Winograd
Waldemar Wiwianka

Richard Anderson
Stephen Balzac
Gerald Baumgartner
Vladimir Bondarenko
Peter A. Broadbery
Florian Bundschuh
Bob Caviness
David V. Chudnovsky
Christophe Conil
Robert Corless
Claire Di Crescenzo
James H. Davenport
Michael Dewar
Lionel Ducos
Dominique Duval
Lars Erickson
Stuart Feldman
George Frances
Korrinn Fu
Kathy Gerber
Teresa Gomez-Diaz
Johannes Grabmeier
Vladimir Grinberg
Steve Hague
Martin Hassner
Antoine Hersen
Kai Kaminski
Paul Kosinski
Larry Lambe
Howard Levy
Richard Luczak
Michael McGettrick
Victor S. Miller
H. Michael Moeller
Scott Morrison
C. Andrew Neff
Arthur Norman
Kostas Oikonomou
Jaap Weel
Didier Pinchon
Michael Richardson
Nicolas Robidoux
Martin Rubey
William Schelter
Marshall Schor
William Sit
Christine Sundaresan
Eugene Surowitz
Mike Thomas
Themos T. Tsikas
Stephen Watt
Mark Wegman
Michael Wester
Clifton J. Williamson
Robert Wisbauer
Knut Wolf

Contents

0.1 Makefile 1

1 Compiler top level **3**

1.1)compile 3

1.1.1 Spad compiler 6

1.1.2 Aldor compiler 8

1.1.3 The top level compiler command 9

1.1.4 The Spad compiler top level function 13

1.1.5 defun compilerDoit 16

1.1.6 defun /rf-1 17

1.1.7 defun spad 18

1.1.8 defun Interpreter interface to the compiler 20

1.1.9 defun compTopLevel 23

1.1.10 defun compOrCroak 24

1.1.11 defun compOrCroak1 25

1.1.12 defun comp 26

1.1.13 defun compNoStacking 27

1.1.14 defun compNoStacking1 27

1.1.15 defun comp2 28

1.1.16 defun comp3 29

1.1.17 defun compTypeOf 31

1.1.18 defun compColon 32

1.1.19 defun compColonInside 36

1.1.20 defun compAtom 37

1.1.21 defun convert 38

1.1.22 defun primitiveType 39

1.1.23 defun compSymbol 40

1.1.24 defun compList 41

1.1.25 defun compVector 42

1.1.26 defun compExpression 43

1.1.27 defun compForm 43

1.1.28 defun compForm1 44

1.1.29 defun compForm2 47

1.1.30 defun compArgumentsAndTryAgain 49

1.1.31 defun compWithMappingMode 49

1.1.32 defun compWithMappingMode1 50
1.1.33 defun extractCodeAndConstructTriple 58
1.1.34 defun hasFormalMapVariable 58
1.1.35 defun compLambda 59
1.1.36 defun compAtSign 60
1.1.37 defun argsToSig 61
1.1.38 defun compMakeDeclaration 62
1.1.39 defun Create a list of unbound symbols 63
1.1.40 defun compOrCroak1,compactify 64
1.1.41 defun Compiler/Interpreter interface 64
1.1.42 defun /RQ,LIB 64
1.1.43 defun compileSpadLispCmd 65
1.1.44 defun recompile-lib-file-if-necessary 66
1.1.45 defun spad-fixed-arg 66
1.1.46 defun compile-lib-file 67
1.1.47 defun compileAsharpCmd 67
1.1.48 defun compileAsharpCmd1 67
1.1.49 defun compileAsharpArchiveCmd 67
1.1.50 defun compileAsharpLispCmd 68
1.1.51 defun withAsharpCmd 68
1.1.52 defun compileFileQuietly 68
1.1.53 defvar $byConstructors 68
1.1.54 defvar $constructorsSeen 68

2 The Compiler **69**

3 Index **73**

New Foreword

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

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

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

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

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

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

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

0.1 Makefile

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

Chapter 1

Compiler top level

1.1)compile

This is the implementation of the)compile command.

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

User Level Required: compiler

Command Syntax:

`)compile`

`)compile` *fileName*

`)compile` *fileName*`.spad`

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

`)compile` *fileName* `)old`

`)compile` *fileName* `)translate`

`)compile` *fileName* `)quiet`

`)compile` *fileName* `)noquiet`

`)compile` *fileName* `)moreargs`

`)compile` *fileName* `)onlyargs`

`)compile` *fileName* `)break`

`)compile` *fileName* `)nobreak`

`)compile` *fileName* `)library`

`)compile` *fileName* `)nolibrary`

`)compile` *fileName* `)vartrace`

`)compile` *fileName* `)constructor` *nameOrAbbrev*

These command forms invoke the Aldor compiler.

`)compile` *fileName*`.as`

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

`)compile` *fileName*`.ao`

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

`)compile` *fileName*`.al`

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

`)compile` *fileName*`.lsp`

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

`)compile` *fileName* `)new`

Command Description:

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

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

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

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

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

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

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

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

This is frequently all you need to compile your file.

This simple command:

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

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

```
)compile mycode )nolibrary
```

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

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

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

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

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

```
)library )dir .
```

1.1.1 Spad compiler

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

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

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

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

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

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

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

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

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

The `)vartrace` option causes the compiler to generate extra code for the constructor to support conditional tracing of variable assignments. (see **??** on page **??**). Without this option, this code is suppressed and one cannot use the

`)vars` option for the trace command.

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

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

or

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

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

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

Given the top level command:

```
)co PR
```

The default call chain looks like:

```
  1> (|compiler| (PR))
    2> (|compileSpad2Cmd| ("/research/test/int/algebra/PR.spad"))
      3> (|compilerDoit| NIL (|rq| |lib|))
        4> (/RF-1 NIL)
          5> (SPAD "/research/test/int/algebra/PR.spad")
            6> (S-PROCESS (|where| ...
              7> (|compTopLevel| (|where| (DEF ...
                8> (|compOrCroak| (|where| (DEF ...
                  9> (|compOrCroak| (CATEGORY |domain| ...
                  <9 (|compOrCroak| ((|DomainSubstitutionMacro| # #) ...
...
                    10> (|compOrCroak| (|FreeModule| R E) |$EmptyMode| ((#)))
                    <10 (|compOrCroak| ((|FreeModule| R E) (|Join| # # #) (#)))
...
                    11> (|compOrCroak| (|construct| (|construct| # #)) $ ((#)))
                    <11 (|compOrCroak| ((LIST #) $ (#)))
...
                  <9 (|compOrCroak| ((PROGN ... $) #1=(|Join| # #) (#)))
                <8 (|compOrCroak| (|PolynomialRing| (|Mapping| ...
              <7 (|compTopLevel| (|PolynomialRing| ...
            <6 (S-PROCESS NIL)
          <5 (SPAD T)
        <4 (/RF-1 T)
      <3 (|compilerDoit| T)
(1) ->
```

1.1.2 Aldor compiler

This command compiles files with file extensions *.as, .ao* and *.al* with the Aldor compiler. It also can compile files with file extension *.lsp*. These are assumed to be Lisp files genererated by the Aldor compiler.

The general description of Aldor command line arguments is in the Aldor documentation. The default options used by the `)compile` command can be viewed and set using the `)set compiler args` Axiom system command. The current defaults are

```
-O -Fasy -Fao -Flsp -laxiom -Mno-AXL_W_WillObsolete -DAxiom
```

These options mean:

- `-O`: perform all optimizations,
- `-Fasy`: generate a `.asy` file,
- `-Fao`: generate a `.ao` file,
- `-Flsp`: generate a `.lsp` (Lisp) file,
- `-laxiom`: use the `axiom` library `libaxiom.al`,
- `-Mno-AXL_W_WillObsolete`: do not display messages about older generated files becoming obsolete, and
- `-DAxiom`: define the global assertion `Axiom` so that the Aldor libraries for generating stand-alone code are not accidentally used with Axiom.

To supplement these default arguments, use the `)moreargs` option on `)compile`. For example,

```
)compile mycode.as )moreargs "-v"
```

uses the default arguments and appends the `-v` (verbose) argument flag. The additional argument specification **must be enclosed in double quotes.**

To completely replace these default arguments for a particular use of `)compile`, use the `)onlyargs` option. For example,

```
)compile mycode.as )onlyargs "-v -O"
```

only uses the `-v` (verbose) and `-O` (optimize) arguments. The argument specification **must be enclosed in double quotes.** In this example, Lisp code is not produced and so the compilation output will not be available to Axiom.

To completely replace the default arguments for all calls to `)compile` within your Axiom session, use `)set compiler args`. For example, to use the above arguments for all compilations, issue

```
)set compiler args "-v -O"
```

Make sure you include the necessary `-l` and `-Y` arguments along with those needed for Lisp file creation. As above, **the argument specification must be enclosed in double quotes.**

The `)compile` command works with several file extensions. We saw above what happens when it is invoked on a file with extension `.as`. A `.ao` file is a portable binary compiled version of a `.as` file, and `)compile` simply passes the `.ao` file onto Aldor. The generated Lisp file is compiled and `)library` is automatically called, just as if you had specified a `.as` file.

A `.al` file is an archive file containing `.ao` files. The archive is created (on Unix systems) with the `ar` program. When `)compile` is given a `.al` file, it creates a directory whose name is based on that of the archive. For example, if you issue

```
)compile mylib.al
```

the directory `mylib.axldir` is created. All members of the archive are unarchived into the directory and `)compile` is called on each `.ao` file found. It is your responsibility to remove the directory and its contents, if you choose to do so.

A `.lsp` file is a Lisp source file, generated by Aldor when called with the `-Flsp` option. When `)compile` is used with a `.lsp` file, the Lisp file is compiled and `)library` is called. For Aldor, You must also have present a `.asy` generated from the same source file.

1.1.3 The top level compiler command

[helpSpad2Cmd(5) p**??**]
[selectOptionLC(5) p**??**]
[pathname(5) p**??**]
[mergePathnames(5) p**??**]
[pathnameType(5) p**??**]
[namestring(5) p**??**]
[throwKeyedMsg p**??**]
[findfile p**??**]
[compileSpad2Cmd p13]
[compileAsharpLispCmd p68]
[compileSpadLispCmd p65]
[compileAsharpCmd p67]
[compileAsharpArchiveCmd p67]
[$newConlist p**??**]
[$options p**??**]
[/editfile p**??**]

⟨*defun compiler*⟩≡

```
(defun |compiler| (args)
 "The top level compiler command"
 (let (|$newConlist| optlist optname optargs havenew haveold aft ef af af1)
  (declare (special |$newConlist| |$options| /editfile))
  (setq |$newConlist| nil)
  (cond
   ((and (null args) (null |$options|) (null /editfile))
     (|helpSpad2Cmd| '(|compiler|)))
   (t
    (cond ((null args) (setq args (cons /editfile nil))))
    (setq optlist '(|new| |old| |translate| |constructor|))
    (setq havenew nil)
    (setq haveold nil)
    (do ((t0 |$options| (cdr t0)) (opt nil))
        ((or (atom t0)
             (progn (setq opt (car t0)) nil)
             (null (null (and havenew haveold))))
          nil)
     (setq optname (car opt))
     (setq optargs (cdr opt))
     (case (|selectOptionLC| optname optlist nil)
      (|new|         (setq havenew t))
      (|translate|   (setq haveold t))
      (|constructor| (setq haveold t))
      (|old|         (setq haveold t))))
    (cond
     ((and havenew haveold) (|throwKeyedMsg| 's2iz0081 nil))
     (t
      (setq af (|pathname| args))
      (setq aft (|pathnameType| af))
      (cond
       ((or havenew (string= aft "as"))
        (if (null (setq af1 ($findfile af '(|as|))))
          (|throwKeyedMsg| 's2il0003 (cons (namestring af) nil))
          (|compileAsharpCmd| (cons af1 nil))))
       ((or haveold (string= aft "spad"))
        (if (null (setq af1 ($findfile af '(|spad|))))
           (|throwKeyedMsg| 's2il0003 (cons (namestring af) nil))
           (|compileSpad2Cmd| (cons af1 nil))))
       ((string= aft "lsp")
        (if (null (setq af1 ($findfile af '(|lsp|))))
          (|throwKeyedMsg| 's2il0003 (cons (namestring af) nil))
          (|compileAsharpLispCmd| (cons af1 nil))))
       ((string= aft "nrlib")
        (if (null (setq af1 ($findfile af '(|nrlib|))))
```

```
    (|throwKeyedMsg| 'S2IL0003 (cons (namestring af) nil))
    (|compileSpadLispCmd| (cons af1 nil))))
 ((string= aft "ao")
  (if (null (setq af1 ($findfile af '(|ao|))))
    (|throwKeyedMsg| 's2il0003 (cons (namestring af) nil))
   (|compileAsharpCmd| (cons af1 nil))))
 ((string= aft "al")
  (if (null (setq af1 ($findfile af '(|al|))))
    (|throwKeyedMsg| 's2il0003 (cons (namestring af) nil))
    (|compileAsharpArchiveCmd| (cons af1 nil))))
 (t
  (setq af1 ($findfile af '(|as| |spad| |ao| |asy|)))
  (cond
   ((and af1 (string= (|pathnameType| af1) "as"))
    (|compileAsharpCmd| (cons af1 nil)))
   ((and af1 (string= (|pathnameType| af1) "ao"))
    (|compileAsharpCmd| (cons af1 nil)))
   ((and af1 (string= (|pathnameType| af1) "spad"))
    (|compileSpad2Cmd| (cons af1 nil)))
   ((and af1 (string= (|pathnameType| af1) "asy"))
    (|compileAsharpArchiveCmd| (cons af1 nil)))
   (t
    (setq ef (|pathname| /editfile))
    (setq ef (|mergePathnames| af ef))
    (cond
     ((boot-equal ef af) (|throwKeyedMsg| 's2iz0039 nil))
     (t
      (setq af ef)
      (cond
       ((string= (|pathnameType| af) "as")
        (|compileAsharpCmd| args))
       ((string= (|pathnameType| af) "ao")
        (|compileAsharpCmd| args))
       ((string= (|pathnameType| af) "spad")
        (|compileSpad2Cmd| args))
       (t
        (setq af1 ($findfile af '(|as| |spad| |ao| |asy|)))
        (cond
         ((and af1 (string= (|pathnameType| af1) "as"))
           (|compileAsharpCmd| (cons af1 nil)))
         ((and af1 (string= (|pathnameType| af1) "ao"))
           (|compileAsharpCmd| (cons af1 nil)))
         ((and af1 (string= (|pathnameType| af1) "spad"))
           (|compileSpad2Cmd| (cons af1 nil)))
         ((and af1 (string= (|pathnameType| af1) "asy"))
          (|compileAsharpArchiveCmd| (cons af1 nil)))
```

```
(t (|throwKeyedMsg| 's2iz0039 nil)))))))))))))))))
```

1.1.4 The Spad compiler top level function

This is the old compiler. Assume we entered from the "compiler" function, so args is a file with file extension .spad.

The `$f` and `$m` are compiler variables, probably function and mode. [pathname(5) p??]
[pathnameType(5) p??]
[namestring(5) p??]
[updateSourceFiles(5) p??]
[selectOptionLC(5) p??]
[terminateSystemCommand(5) p??]
[nequal p??]
[throwKeyedMsg p??]
[sayKeyedMsg p??]
[error p??]
[strconc p??]
[object2String p??]
[oldParserAutoloadOnceTrigger p??]
[browserAutoloadOnceTrigger p??]
[spad2AsTranslatorAutoloadOnceTrigger p??]
[convertSpadToAsFile p??]
[compilerDoitWithScreenedLisplib p??]
[compilerDoit p16]
[extendLocalLibdb p??]
[spadPrompt p??]
[$newComp p??]
[$scanIfTrue p??]
[$compileOnlyCertainItems p??]
[$f p??]
[$m p??]
[$QuickLet p??]
[$QuickCode p??]
[$sourceFileTypes p??]
[$InteractiveMode p??]
[$options p??]
[$newConlist p??]
[/editfile p??]

⟨*defun compileSpad2Cmd*⟩≡
```
(defun |compileSpad2Cmd| (args)
 (let (|$newComp| |$scanIfTrue|
       |$compileOnlyCertainItems| |$f| |$m| |$QuickLet| |$QuickCode|
       |$sourceFileTypes| |$InteractiveMode| path optlist fun optname
       optargs fullopt translateoldtonew constructor)
  (declare (special |$newComp| |$scanIfTrue|
```

```
    |$compileOnlyCertainItems| |$f| |$m| |$QuickLet| |$QuickCode|
    |$sourceFileTypes| |$InteractiveMode| /editfile |$options|
    |$newConlist|))
(setq path (|pathname| args))
(cond
 ((nequal (|pathnameType| path) "spad") (|throwKeyedMsg| 's2iz0082 nil))
 ((null (probe-file path))
  (|throwKeyedMsg| 's2il0003 (cons (|namestring| args) nil)))
 (t
  (setq /editfile path)
  (|updateSourceFiles| path)
  (|sayKeyedMsg| 's2iz0038 (list (|namestring| args)))
  (setq optlist '(|break| |constructor| |functions| |library| |lisp|
      |new| |old| |nobreak| |nolibrary| |noquiet| |vartrace| |quiet|
      |translate|))
  (setq |$QuickLet| t)
  (setq |$QuickCode| t)
  (setq fun '(|rq| |lib|))
  (setq |$sourceFileTypes| '("SPAD"))
  (dolist (opt |$options|)
   (setq optname (car opt))
   (setq optargs (cdr opt))
   (setq fullopt (|selectOptionLC| optname optlist nil))
   (case fullopt
    (|new| (|error| '|Internal error: compileSpad2Cmd got )new|))
    (|old| nil)
    (|translate| (setq translateoldtonew t))
    (|library| (setelt fun 1 '|lib|))
    (|nolibrary| (setelt fun 1 '|nolib|))
    (|quiet| (when (nequal (elt fun 0) '|c|) (setelt fun 0 '|rq|)))
    (|noquiet| (when (nequal (elt fun 0) '|c|) (setelt fun 0 '|rf|)))
    (|nobreak| (setq |$scanIfTrue| t))
    (|break| (setq |$scanIfTrue| nil))
    (|vartrace| (setq |$QuickLet| nil))
    (|lisp| (|throwKeyedMsg| 's2iz0036 (list ")lisp")))
    (|functions|
     (if (null optargs)
      (|throwKeyedMsg| 's2iz0037 (list ")functions"))
      (setq |$compileOnlyCertainItems| optargs)))
    (|constructor|
     (if (null optargs)
      (|throwKeyedMsg| 's2iz0037 (list ")constructor"))
      (progn
       (setelt fun 0 '|c|)
       (setq constructor (mapcar #'|unabbrev| optargs)))))
    (t
```

```
    (|throwKeyedMsg| 's2iz0036
     (list (strconc ")" (|object2String| optname)))))))
(setq |$InteractiveMode| nil)
(cond
 (translateoldtonew
  (|oldParserAutoloadOnceTrigger|)
  (|browserAutoloadOnceTrigger|)
  (|spad2AsTranslatorAutoloadOnceTrigger|)
  (|sayKeyedMsg| 's2iz0085 nil)
  (|convertSpadToAsFile| path))
 (|$compileOnlyCertainItems|
  (if (null constructor)
   (|sayKeyedMsg| 's2iz0040 nil)
   (|compilerDoitWithScreenedLisplib| constructor fun)))
 (t (|compilerDoit| constructor fun)))
(|extendLocalLibdb| |$newConlist|)
(|terminateSystemCommand|)
(|spadPrompt|)))))
```

1.1.5 defun compilerDoit

[/rq(5) p**??**]
[/rf(5) p**??**]
[member(5) p**??**]
[sayBrightly p**??**]
[$byConstructors p68]
[$constructorsSeen p68]

⟨*defun compilerDoit*⟩≡

```
(defun |compilerDoit| (constructor fun)
 (let (|$byConstructors| |$constructorsSeen|)
 (declare (special |$byConstructors| |$constructorsSeen|))
  (cond
   ((equal fun '(|rf| |lib|))   (|/RQ,LIB|))   ; Ignore "noquiet"
   ((equal fun '(|rf| |nolib|)) (/rf))
   ((equal fun '(|rq| |lib|))   (|/RQ,LIB|))
   ((equal fun '(|rq| |nolib|)) (/rq))
   ((equal fun '(|c| |lib|))
    (setq |$byConstructors| (loop for x in constructor collect (|opOf| x)))
    (|/RQ,LIB|)
    (dolist (x |$byConstructors|)
     (unless (|member| x |$constructorsSeen|)
      (|sayBrightly| `(">>> Warning " |%b| ,x |%d| " was not found")))))))))
```

1.1.6 defun /rf-1

[makeInputFilename(5) p**??**]
[ncINTERPFILE p64]
[spad p18]
[/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)))))
```

1.1.7 defun spad

[spad-reader p??]
[addBinding p??]
[makeInitialModemapFrame p??]
[init-boot/spad-reader p??]
[initialize-preparse p??]
[preparse p??]
[PARSE-NewExpr p??]
[pop-stack-1 p??]
[s-process p20]
[ioclear p??]
[shut p??]
[$noSubsumption p??]
[$InteractiveFrame p??]
[$InitialDomainsInScope p??]
[$InteractiveMode p??]
[line p??]
[echo-meta p??]
[/editfile p??]
[*comp370-apply* p??]
[*eof* p??]
[file-closed p??]
[xcape p??]
[spad-reader p??]

⟨*defun spad*⟩≡

```
(defun spad (&optional (*spad-input-file* nil) (*spad-output-file* nil)
             &aux (*comp370-apply* #'print-defun)
                  (*fileactq-apply* #'print-defun)
                 ($spad t) ($boot nil) (xcape #\_) (optionlist nil) (*eof* nil)
                 (file-closed nil) (/editfile *spad-input-file*)
                (|$noSubsumption| |$noSubsumption|) in-stream out-stream)
  (declare (special echo-meta /editfile *comp370-apply* *eof*
                    file-closed xcape |$noSubsumption| |$InteractiveFrame|
                    |$InteractiveMode| |$InitialDomainsInScope|))
  ;; only rebind |$InteractiveFrame| if compiling
  (progv (if (not |$InteractiveMode|) '(|$InteractiveFrame|))
         (if (not |$InteractiveMode|)
             (list (|addBinding| '|$DomainsInScope|
                    `((fluid . |true|)
                      (|special| . ,(copy-tree |$InitialDomainsInScope|)))
                   (|addBinding| '|$Information| nil
                      (|makeInitialModemapFrame|)))))
  (init-boot/spad-reader)
```

```
(unwind-protect
  (progn
    (setq in-stream (if *spad-input-file*
                      (open *spad-input-file* :direction :input)
                       *standard-input*))
    (initialize-preparse in-stream)
    (setq out-stream (if *spad-output-file*
                      (open *spad-output-file* :direction :output)
                       *standard-output*))
    (when *spad-output-file*
       (format out-stream "~&;;; -*- Mode:Lisp; Package:Boot  -*-~%~%")
       (print-package "BOOT"))
    (setq curoutstream out-stream)
    (loop
     (if (or *eof* file-closed) (return nil))
     (catch 'spad_reader
       (if (setq boot-line-stack (preparse in-stream))
           (let ((line (cdar boot-line-stack)))
             (declare (special line))
             (|PARSE-NewExpr|)
             (let ((parseout (pop-stack-1)) )
               (when parseout
                     (let ((*standard-output* out-stream))
                       (s-process parseout))
                     (format out-stream "~&")))
             )))
    (ioclear in-stream out-stream)))
  (if *spad-input-file* (shut in-stream))
  (if *spad-output-file* (shut out-stream)))
t))
```

1.1.8 defun Interpreter interface to the compiler

[curstrm p??]
[def-rename p??]
[new2OldLisp p??]
[parseTransform p??]
[postTransform p??]
[displayPreCompilationErrors p??]
[prettyprint p??]
[processInteractive p??]
[compTopLevel p23]
[def-process p??]
[displaySemanticErrors p??]
[terpri p??]
[get-internal-run-time p??]
[$Index p??]
[$macroassoc p??]
[$newspad p??]
[$PolyMode p??]
[$EmptyMode p??]
[$compUniquelyIfTrue p??]
[$currentFunction p??]
[$postStack p??]
[$topOp p??]
[$semanticErrorStack p??]
[$warningStack p??]
[$exitMode p??]
[$exitModeStack p??]
[$returnMode p??]
[$leaveMode p??]
[$leaveLevelStack p??]
[$top-level p??]
[$insideFunctorIfTrue p??]
[$insideExpressionIfTrue p??]
[$insideCoerceInteractiveHardIfTrue p??]
[$insideWhereIfTrue p??]
[$insideCategoryIfTrue p??]
[$insideCapsuleFunctionIfTrue p??]
[$form p??]
[$DomainFrame p??]
[$e p??]
[$EmptyEnvironment p??]
[$genFVar p??]
[$genSDVar p??]
[$VariableCount p??]
[$previousTime p??]

[$LocalFrame p??]
[curoutstream p??]

⟨*defun s-process*⟩≡
```
(defun s-process (x)
 (prog ((|$Index| 0)
        ($macroassoc ())
        ($newspad t)
        (|$PolyMode| |$EmptyMode|)
        (|$compUniquelyIfTrue| nil)
        |$currentFunction|
        (|$postStack| nil)
        |$topOp|
        (|$semanticErrorStack| ())
        (|$warningStack| ())
        (|$exitMode| |$EmptyMode|)
        (|$exitModeStack| ())
        (|$returnMode| |$EmptyMode|)
        (|$leaveMode| |$EmptyMode|)
        (|$leaveLevelStack| ())
        $top_level |$insideFunctorIfTrue| |$insideExpressionIfTrue|
        |$insideCoerceInteractiveHardIfTrue| |$insideWhereIfTrue|
        |$insideCategoryIfTrue| |$insideCapsuleFunctionIfTrue| |$form|
        (|$DomainFrame| '((NIL)))
        (|$e| |$EmptyEnvironment|)
        (|$genFVar| 0)
        (|$genSDVar| 0)
        (|$VariableCount| 0)
        (|$previousTime| (get-internal-run-time))
        (|$LocalFrame| '((NIL)))
        (curstrm curoutstream) |$s| |$x| |$m| u)
  (declare (special |$Index| $macroassoc $newspad |$PolyMode| |$EmptyMode|
            |$compUniquelyIfTrue| |$currentFunction| |$postStack| |$topOp|
            |$semanticErrorStack| |$warningStack| |$exitMode| |$exitModeStack|
            |$returnMode| |$leaveMode| |$leaveLevelStack| $top_level
            |$insideFunctorIfTrue| |$insideExpressionIfTrue|
            |$insideCoerceInteractiveHardIfTrue| |$insideWhereIfTrue|
            |$insideCategoryIfTrue| |$insideCapsuleFunctionIfTrue| |$form|
            |$DomainFrame| |$e| |$EmptyEnvironment| |$genFVar| |$genSDVar|
            |$VariableCount| |$previousTime| |$LocalFrame|
            curstrm |$s| |$x| |$m| curoutstream $traceflag))
  (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)))
```

1.1.9 defun compTopLevel

[newComp p??]
[compOrCroak p24]
[$NRTderivedTargetIfTrue p??]
[$killOptimizeIfTrue p??]
[$forceAdd p??]
[$compTimeSum p??]
[$resolveTimeSum p??]
[$packagesUsed p??]
[$envHashTable p??]

⟨*defun compTopLevel*⟩≡

```
(defun |compTopLevel| (x m e)
 (let (|$NRTderivedTargetIfTrue| |$killOptimizeIfTrue| |$forceAdd|
       |$compTimeSum| |$resolveTimeSum| |$packagesUsed| |$envHashTable|
        t1 t2 t3 val mode)
 (declare (special |$NRTderivedTargetIfTrue| |$killOptimizeIfTrue|
                   |$forceAdd| |$compTimeSum| |$resolveTimeSum|
                   |$packagesUsed| |$envHashTable| ))
  (setq |$NRTderivedTargetIfTrue| nil)
  (setq |$killOptimizeIfTrue| nil)
  (setq |$forceAdd| nil)
  (setq |$compTimeSum| 0)
  (setq |$resolveTimeSum| 0)
  (setq |$packagesUsed| NIL)
  (setq |$envHashTable| (make-hashtable 'equal))
  (dolist (u (car (car e)))
   (dolist (v (cdr u))
    (hput |$envHashTable| (cons (car u) (cons (car v) nil)) t)))
  (cond
   ((or (and (pairp x) (eq (qcar x) 'def))
        (and (pairp x) (eq (qcar x) '|where|)
             (progn
               (setq t1 (qcdr x))
               (and (pairp t1)
               (progn
                (setq t2 (qcar t1))
                (and (pairp t2) (eq (qcar t2) 'def)))))))
    (setq t3 (|compOrCroak| x m e))
    (setq val (car t3))
    (setq mode (cadr t3))
    (cons val (cons mode (cons e nil))))
   (t (|compOrCroak| x m e)))))
```

Given:

```
CohenCategory(): Category == SetCategory with

  kind:(CExpr)->Boolean
  operand:(CExpr,Integer)->CExpr
  numberOfOperand:(CExpr)->Integer
  construct:(CExpr,CExpr)->CExpr
```

the resulting call looks like:

```
 (|compOrCroak|
       (DEF (|CohenCategory|)
        ((|Category|))
        (NIL)
        (|Join|
         (|SetCategory|)
         (CATEGORY |package|
          (SIGNATURE |kind| ((|Boolean|) |CExpr|))
          (SIGNATURE |operand| (|CExpr| |CExpr| (|Integer|)))
          (SIGNATURE |numberOfOperand| ((|Integer|) |CExpr|))
          (SIGNATURE |construct| (|CExpr| |CExpr| |CExpr|)))))
        |$EmptyMode|
        (((
           (|$DomainsInScope|
            (FLUID . |true|)
            (|special| |$EmptyMode| |$NoValueMode|))))))
```

This compiler call expects the first argument `x` to be a `DEF` form to compile, The second argument, `m`, is the mode. The third argument, `e`, is the environment.

1.1.10 defun compOrCroak

[compOrCroak1 p25]

⟨*defun compOrCroak*⟩≡

```
  (defun |compOrCroak| (x m e)
    (|compOrCroak1| x m e nil nil))
```

This results in a call to the inner function with

```
(|compOrCroak1|
          (DEF (|CohenCategory|)
           ((|Category|))
           (NIL)
           (|Join|
            (|SetCategory|)
            (CATEGORY |package|
             (SIGNATURE |kind| ((|Boolean|) |CExpr|))
             (SIGNATURE |operand| (|CExpr| |CExpr| (|Integer|)))
             (SIGNATURE |numberOfOperand| ((|Integer|) |CExpr|))
             (SIGNATURE |construct| (|CExpr| |CExpr| |CExpr|)))))
          |$EmptyMode|
          ((((
              |$DomainsInScope|
              (FLUID . |true|)
              (|special| |$EmptyMode| |$NoValueMode|))))
          NIL
          NIL
          |comp|)
```

The inner function augments the environment with information from the compiler stack `$compStack` and `$compErrorMessageStack`. Note that these variables are passed in the argument list so they get preserved on the call stack. The calling function gets called for every inner form so we use this implicit stacking to retain the information.

1.1.11 defun compOrCroak1

[comp p26]
[compOrCroak1,compactify p64]
[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 p24]

⟨*defun compOrCroak1*⟩≡

```
(defun |compOrCroak1| (x m e |$compStack| |$compErrorMessageStack|)
 (declare (special |$compStack| |$compErrorMessageStack|))
 (let (td errorMessage)
  (declare (special |$level| |$s| |$scanIfTrue| |$exitModeStack|))
  (cond
   ((setq td (catch '|compOrCroak| (|comp| x m e))) td)
   (t
     (setq |$compStack| (cons (list x m e |$exitModeStack|) |$compStack|))
     (setq |$s| (|compOrCroak1,compactify| |$compStack|))
     (setq |$level| (|#| |$s|))
     (setq errorMessage
       (if |$compErrorMessageStack|
         (car |$compErrorMessageStack|)
         '|unspecified error|))
     (cond
      (|$scanIfTrue|
        (|stackSemanticError| errorMessage (|mkErrorExpr| |$level|))
        (list '|failedCompilation| m e ))
      (t
       (|displaySemanticErrors|)
       (say "****** comp fails at level " |$level| " with expression: ******")
       (|displayComp| |$level|)
       (|userError| errorMessage)))))))
```

1.1.12 defun comp

[compNoStacking p27]
[$compStack p**??**]
[$exitModeStack p**??**]

⟨*defun comp*⟩≡

```
(defun |comp| (x m e)
 (let (td)
 (declare (special |$compStack| |$exitModeStack|))
  (if (setq td (|compNoStacking| x m e))
    (setq |$compStack| nil)
    (push (list x m e |$exitModeStack|) |$compStack|))
  td))
```

1.1.13 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 p28]
[compNoStacking1 p27]
[$compStack p**??**]
[$Representation p**??**]
[$EmptyMode p**??**]

⟨*defun compNoStacking*⟩≡
```
(defun |compNoStacking| (x m e)
 (let (td)
 (declare (special |$compStack| |$Representation| |$EmptyMode|))
  (if (setq td (|comp2| x m e))
    (if (and (equal m |$EmptyMode|) (equal (cadr td) |$Representation|))
      (list (car td) '$ (caddr td))
      td)
   (|compNoStacking1| x m e |$compStack|))))
```

1.1.14 defun compNoStacking1

[get p**??**]
[comp2 p28]
[$compStack p**??**]

⟨*defun compNoStacking1*⟩≡
```
(defun |compNoStacking1| (x m e |$compStack|)
 (declare (special |$compStack|))
 (let (u td)
  (if (setq u (|get| (if (eq m '$) '|Rep| m) '|value| e))
    (if (setq td (|comp2| x (car u) e))
      (list (car td) m (caddr td))
      nil)
    nil)))
```

1.1.15 defun comp2

[comp3 p29]
[isDomainForm p??]
[isFunctor p??]
[insert p??]
[opOf p??]
[nequal p??]
[addDomain p??]
[$bootStrapMode p??]
[$packagesUsed p??]
[$lisplib p??]

⟨*defun comp2*⟩≡

```
(defun |comp2| (x m e)
 (let (tmp1)
  (declare (special |$bootStrapMode| |$packagesUsed| $lisplib))
   (when (setq tmp1 (|comp3| x m e))
    (destructuring-bind (y mprime e) tmp1
     (when (and $lisplib (|isDomainForm| x e) (|isFunctor| x))
       (setq |$packagesUsed| (|insert| (list (|opOf| x)) |$packagesUsed|)))
     ; isDomainForm test needed to prevent error while compiling Ring
     ; $bootStrapMode-test necessary for compiling Ring in $bootStrapMode
     (if (and (nequal m mprime)
              (or |$bootStrapMode| (|isDomainForm| mprime e)))
       (list y mprime (|addDomain| mprime e))
       (list y mprime e))))))
```

1.1.16 defun comp3

```
;comp3(x,m,$e) ==
;  --returns a Triple or %else nil to signalcan't do'
;  $e:= addDomain(m,$e)
;  e:= $e --for debugging purposes
;  m is ["Mapping",:.] => compWithMappingMode(x,m,e)
;  m is ["QUOTE",a] => (x=a => [x,m,$e]; nil)
;  STRINGP m => (atom x => (m=x or m=STRINGIMAGE x => [m,m,e]; nil); nil)
;  ^x or atom x => compAtom(x,m,e)
;  op:= first x
;  getmode(op,e) is ["Mapping",:ml] and (u:= applyMapping(x,m,e,ml)) => u
;  op is ["KAPPA",sig,varlist,body] => compApply(sig,varlist,body,rest x,m,e)
;  op=":" => compColon(x,m,e)
;  op="::" => compCoerce(x,m,e)
;  not ($insideCompTypeOf=true) and stringPrefix?('"TypeOf",PNAME op) =>
;    compTypeOf(x,m,e)
;  t:= compExpression(x,m,e)
;  t is [x',m',e'] and not MEMBER(m',getDomainsInScope e') =>
;    [x',m',addDomain(m',e')]
;  t
```

[addDomain p**??**]
[compWithMappingMode p49]
[stringimage p**??**]
[compAtom p37]
[getmode p**??**]
[applyMapping p**??**]
[compApply p**??**]
[compColon p32]
[compCoerce p**??**]
[stringPrefix? p**??**]
[pname p**??**]
[compTypeOf p31]
[compExpression p43]
[member p**??**]
[getDomainsInScope p**??**]
[$e p**??**]
[$insideCompTypeOf p**??**]

⟨*defun comp3*⟩≡

```
(defun |comp3| (x m |$e|)
 (declare (special |$e|))
 (let (e a op ml u sig varlist tmp3 body tt xprime tmp1 mprime tmp2 eprime)
 (declare (special |$insideCompTypeOf|))
  (setq |$e| (|addDomain| m |$e|))
  (setq e |$e|)
```

```
(cond
 ((and (pairp m) (eq (qcar m) '|Mapping|)) (|compWithMappingMode| x m e))
 ((and (pairp m) (eq (qcar m) 'quote)
       (progn
        (setq tmp1 (qcdr m))
        (and (pairp tmp1) (eq (qcdr tmp1) nil)
        (progn (setq a (qcar tmp1)) t))))
  (when (equal x a) (list x m |$e|)))
 ((stringp m)
  (when (and (atom x) (or (equal m x) (equal m (stringimage x))))
   (list m m e )))
 ((or (null x) (atom x)) (|compAtom| x m e))
 (t
  (setq op (car x))
  (cond
   ((and (progn
          (setq tmp1 (|getmode| op e))
          (and (pairp tmp1)
               (eq (qcar tmp1) '|Mapping|)
               (progn (setq ml (qcdr tmp1)) t)))
          (setq u (|applyMapping| x m e ml)))
      u)
   ((and (pairp op) (eq (qcar op) 'kappa)
         (progn
          (setq tmp1 (qcdr op))
          (and (pairp tmp1)
               (progn
                (setq sig (qcar tmp1))
                 (setq tmp2 (qcdr tmp1))
                 (and (pairp tmp2)
                      (progn
                       (setq varlist (qcar tmp2))
                       (setq tmp3 (qcdr tmp2))
                       (and (pairp tmp3)
                            (eq (qcdr tmp3) nil)
                            (progn
                              (setq body (qcar tmp3))
                              t))))))))
     (|compApply| sig varlist body (cdr x) m e))
   ((eq op '|:|)  (|compColon| x m e))
   ((eq op '|::|) (|compCoerce| x m e))
   ((and (null (eq |$insideCompTypeOf| t))
         (|stringPrefix?| "TypeOf" (pname op)))
     (|compTypeOf| x m e))
    (t
     (setq tt (|compExpression| x m e))
```

```
        (cond
         ((and (pairp tt)
               (progn
                (setq xprime (qcar tt))
                (setq tmp1 (qcdr tt))
                (and (pairp tmp1)
                     (progn
                      (setq mprime (qcar tmp1))
                      (setq tmp2 (qcdr tmp1))
                      (and (pairp tmp2)
                           (eq (qcdr tmp2) nil)
                           (progn
                             (setq eprime (qcar tmp2))
                             t)))))
                (null (|member| mprime (|getDomainsInScope| eprime))))
          (list xprime mprime (|addDomain| mprime eprime)))
         (t tt))))))))
```

1.1.17 defun compTypeOf

[eqsubstlist p??]
[get p??]
[put p??]
[comp3 p29]
[$insideCompTypeOf p??]
[$FormalMapVariableList p??]

⟨*defun compTypeOf*⟩≡

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

1.1.18 defun compColon

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

[compColonInside p36]
[assoc p??]
[getDomainsInScope p??]
[isDomainForm p??]
[member p??]
[addDomain p??]
[isDomainForm p??]
[isCategoryForm p??]
[unknownTypeError p??]
[compColon p32]
[eqsubstlist p??]
[take p??]
[length p??]
[nreverse0 p??]
[getmode p??]
[systemErrorHere p??]

[put p??]
[makeCategoryForm p??]
[genSomeVariable p??]
[$lhsOfColon p??]
[$noEnv p??]
[$insideFunctorIfTrue p??]
[$bootStrapMode p??]
[$FormalMapVariableList p??]
[$insideCategoryIfTrue p??]
[$insideExpressionIfTrue p??]

⟨*defun compColon*⟩≡

```
(defun |compColon| (arg0 m e)
  (let (|$lhsOfColon| argf argt tprime mprime r l tmp1 td op argl newTarget a
        signature tmp2 catform tmp3 g2 g5)
  (declare (special |$lhsOfColon| |$noEnv| |$insideFunctorIfTrue|
                    |$bootStrapMode| |$FormalMapVariableList|
                    |$insideCategoryIfTrue| |$insideExpressionIfTrue|))
    (setq argf (cadr arg0))
    (setq argt (caddr arg0))
    (if |$insideExpressionIfTrue|
      (|compColonInside| argf m e argt)
      (progn
        (setq |$lhsOfColon| argf)
        (setq argt
         (cond
          ((and (atom argt)
                (setq tprime (|assoc| argt (|getDomainsInScope| e))))
            tprime)
          ((and (|isDomainForm| argt e) (null |$insideCategoryIfTrue|))
            (unless (|member| argt (|getDomainsInScope| e))
               (setq e (|addDomain| argt e)))
            argt)
          ((or (|isDomainForm| argt e) (|isCategoryForm| argt e))
            argt)
          ((and (pairp argt) (eq (qcar argt) '|Mapping|)
                (progn
                  (setq tmp2 (qcdr argt))
                  (and (pairp tmp2)
                       (progn
                        (setq mprime (qcar tmp2))
                        (setq r (qcdr tmp2))
                        t))))
            argt)
          (t
```

```
    (|unknownTypeError| argt)
    argt)))
(cond
 ((eq (car argf) 'listof)
   (dolist (x (cdr argf) td)
     (setq td (|compColon| (list '|:| x argt) m e))
     (setq e (caddr td))))
 (t
  (setq e
   (cond
    ((and (pairp argf)
          (progn
           (setq op (qcar argf))
           (setq argl (qcdr argf))
           t)
          (null (and (pairp argt) (eq (qcar argt) '|Mapping|))))
     (setq newTarget
      (eqsubstlist (take (|#| argl) |$FormalMapVariableList|)
      (dolist (x argl (nreverse0 g2))
        (setq g2
         (cons
          (cond
           ((and (pairp x) (eq (qcar x) '|:|)
                 (progn
                  (setq tmp2 (qcdr x))
                  (and (pairp tmp2)
                       (progn
                        (setq a (qcar tmp2))
                        (setq tmp3 (qcdr tmp2))
                        (and (pairp tmp3)
                             (eq (qcdr tmp3) nil)
                             (progn
                              (setq m (qcar tmp3))
                              t))))))
                a)
               (t x))
          g2)))
      argt))
     (setq signature
      (cons '|Mapping|
       (cons newTarget
        (dolist (x argl (nreverse0 g5))
          (setq g5
           (cons
            (cond
             ((and (pairp x) (eq (qcar x) '|:|)
```

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

1.1.19 defun compColonInside

[addDomain p**??**]
[comp p26]
[coerce p**??**]
[stackWarning p**??**]
[opOf p**??**]
[stackSemanticError p**??**]
[$newCompilerUnionFlag p**??**]
[$EmptyMode p**??**]

⟨*defun compColonInside*⟩≡

```
(defun |compColonInside| (x m e mprime)
 (let (mpp warningMessage td tprime failed)
 (declare (special |$newCompilerUnionFlag| |$EmptyMode|))
    (setq e (|addDomain| mprime e))
    (when (setq td (|comp| x |$EmptyMode| e))
     (cond
      ((equal (setq mpp (CADR td)) mprime)
       (setq warningMessage
        (list '|:| mprime '| -- should replace by @|))))
     (setq td (list (car td) mprime (caddr td)))
     (when (setq tprime (|coerce| td m))
      (cond
       (warningMessage (|stackWarning| warningMessage))
       ((and |$newCompilerUnionFlag| (eq (|opOf| mpp) '|Union|))
        (setq tprime
         (|stackSemanticError|
          (list '|cannot pretend | x '| of mode | mpp '| to mode | mprime )
          nil)))
       (t
        (|stackWarning|
         (list '|:| mprime '| -- should replace by pretend|))))
      tprime))))
```

1.1.20 defun compAtom

```
;compAtom(x,m,e) ==
;  T:= compAtomWithModemap(x,m,e,get(x,"modemap",e)) => T
;  x="nil" =>
;    T:=
;      modeIsAggregateOf('List,m,e) is [.,R]=> compList(x,['List,R],e)
;      modeIsAggregateOf('Vector,m,e) is [.,R]=> compVector(x,['Vector,R],e)
;    T => convert(T,m)
;  t:=
;    isSymbol x =>
;      compSymbol(x,m,e) or return nil
;    m = $Expression and primitiveType x => [x,m,e]
;    STRINGP x => [x,x,e]
;    [x,primitiveType x or return nil,e]
;  convert(t,m)
```

[compAtomWithModemap p**??**]
[get p**??**]
[modeIsAggregateOf p**??**]
[compList p41]
[compVector p42]
[convert p38]
[isSymbol p**??**]
[compSymbol p40]
[primitiveType p39]
[primitiveType p39]
[$Expression p**??**]

⟨*defun compAtom*⟩≡

```
(defun |compAtom| (x m e)
 (prog (tmp1 tmp2 r td tt)
  (declare (special |$Expression|))
   (return
   (cond
    ((setq td (|compAtomWithModemap| x m e (|get| x '|modemap| e))) td)
    ((eq x '|nil|)
     (setq td
      (cond
       ((progn
         (setq tmp1 (|modeIsAggregateOf| '|List| m e))
         (and (pairp tmp1)
              (progn
               (setq tmp2 (qcdr tmp1))
               (and (pairp tmp2)
                    (eq (qcdr tmp2) nil)
                    (progn
```

```
                    (setq r (qcar tmp2)) t)))))
        (|compList| x (list '|List| r) e))
       ((progn
         (setq tmp1 (|modeIsAggregateOf| '|Vector| m e))
         (and (pairp tmp1)
              (progn
               (setq tmp2 (qcdr tmp1))
               (and (pairp tmp2) (eq (qcdr tmp2) nil)
               (progn
                 (setq r (qcar tmp2)) t)))))
         (|compVector| x (list '|Vector| r) e))))
      (when td (|convert| td m)))
     (t
      (setq tt
       (cond
        ((|isSymbol| x) (or (|compSymbol| x m e) (return nil)))
        ((and (equal m |$Expression|) (|primitiveType| x)) (list x m e ))
        ((stringp x) (list x x e ))
        (t (list x (or (|primitiveType| x) (return nil)) e ))))
      (|convert| tt m))))))
```

1.1.21 defun convert

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

⟨*defun convert*⟩≡

```
(defun |convert| (td m)
 (let (res)
  (when (setq res (|resolve| (cadr td) m))
   (|coerce| td res))))
```

1.1.22 defun primitiveType

[$DoubleFloat p??]
[$NegativeInteger p??]
[$PositiveInteger p??]
[$NonNegativeInteger p??]
[$String p??]
[$EmptyMode p??]

⟨*defun primitiveType*⟩≡
```
(defun |primitiveType| (x)
  (declare (special |$DoubleFloat| |$NegativeInteger| |$PositiveInteger|
                    |$NonNegativeInteger| |$String| |$EmptyMode|))
  (cond
    ((null x) |$EmptyMode|)
    ((stringp x) |$String|)
    ((integerp x)
     (cond
       ((eql x 0) |$NonNegativeInteger|)
       ((> x 0) |$PositiveInteger|)
       (t |$NegativeInteger|)))
    ((floatp x) |$DoubleFloat|)
    (t nil)))
```

1.1.23 defun compSymbol

[getmode p??]
[get p??]
[NRTgetLocalIndex p??]
[member p??]
[isFunction p??]
[errorRef p??]
[stackMessage p??]
[$Symbol p??]
[$Expression p??]
[$FormalMapVariableList p??]
[$compForModeIfTrue p??]
[$formalArgList p??]
[$NoValueMode p??]
[$functorLocalParameters p??]
[$Boolean p??]
[$NoValue p??]

⟨*defun compSymbol*⟩≡

```
(defun |compSymbol| (s m e)
 (let (v mprime)
  (declare (special |$Symbol| |$Expression| |$FormalMapVariableList|
                    |$compForModeIfTrue| |$formalArgList| |$NoValueMode|
                    |$functorLocalParameters| |$Boolean| |$NoValue|))
   (cond
     ((eq s '|$NoValue|) (list '|$NoValue| |$NoValueMode| e ))
     ((|isFluid| s)
      (setq mode (|getmode| s e))
      (when mode (list s (|getmode| s e) e)))
     ((eq s '|true|) (list '(quote t) |$Boolean| e ))
     ((eq s '|false|) (list nil |$Boolean| e ))
     ((or (equal s m) (|get| s '|isLiteral| e)) (list (list 'quote s) s e))
     ((setq v (|get| s '|value| e))
      (cond
        ((member s |$functorLocalParameters|)
         ; s will be replaced by an ELT form in beforeCompile
         (|NRTgetLocalIndex| s)
         (list s (cadr v) e))
        (t
         ; s has been SETQd
         (list s (cadr v) e))))
     ((setq mprime (|getmode| s e))
      (cond
        ((and (null (|member| s |$formalArgList|))
```

```
                (null (member s |$FormalMapVariableList|))
                (null (|isFunction| s e))
                (null (eq |$compForModeIfTrue| t)))
           (|errorRef| s)))
         (list s mprime e ))
        ((member s |$FormalMapVariableList|)
         (|stackMessage| (list '|no mode found for| s )))
        ((or (equal m |$Expression|) (equal m |$Symbol|))
         (list (list 'quote s) m e ))
        ((null (|isFunction| s e)) (|errorRef| s)))))
```

1.1.24 defun compList

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

[comp p26]

⟨*defun compList*⟩≡

```
(defun |compList| (l m e)
 (let (tmp1 tmp2 t0 failed (mUnder (cadr m)))
  (if (null l)
   (list nil m e)
   (progn
    (setq t0
     (do ((t3 l (cdr t3)) (x nil))
         ((or (atom t3) failed) (unless failed (nreverse0 tmp2)))
       (setq x (car t3))
       (if (setq tmp1 (|comp| x mUnder e))
        (progn
         (setq mUnder (cadr tmp1))
         (setq e (caddr tmp1))
         (push tmp1 tmp2))
        (setq failed t))))
    (unless failed
     (cons
       (cons 'list (loop for texpr in t0 collect (car texpr)))
       (list (list '|List| mUnder) e)))))))
```

1.1.25 defun compVector

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

[comp p26]
[$EmptyVector p**??**]

⟨*defun compVector*⟩≡

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

1.1.26 defun compExpression

[getl p**??**]
[compForm p43]
[$insideExpressionIfTrue p**??**]

⟨*defun compExpression*⟩≡
```
(defun |compExpression| (x m e)
 (let (|$insideExpressionIfTrue| fn)
 (declare (special |$insideExpressionIfTrue|))
  (setq |$insideExpressionIfTrue| t)
  (if (and (atom (car x)) (setq fn (getl (car x) 'special)))
    (funcall fn x m e)
    (|compForm| x m e))))
```

1.1.27 defun compForm

[compForm1 p44]
[compArgumentsAndTryAgain p49]
[stackMessageIfNone p**??**]

⟨*defun compForm*⟩≡
```
(defun |compForm| (form m e)
 (cond
  ((|compForm1| form m e))
  ((|compArgumentsAndTryAgain| form m e))
  (t (|stackMessageIfNone| (list '|cannot compile| '|%b| form '|%d| )))))
```

1.1.28 defun compForm1

[length p??]
[outputComp p??]
[compOrCroak p24]
[compExpressionList p??]
[coerceable p??]
[comp p26]
[coerce p??]
[compForm2 p47]
[augModemapsFromDomain1 p??]
[getFormModemaps p??]
[nreverse0 p??]
[addDomain p??]
[compToApply p??]
[$NumberOfArgsIfInteger p??]
[$Expression p??]
[$EmptyMode p??]

⟨*defun compForm1*⟩≡

```
(defun |compForm1| (form m e)
 (let (|$NumberOfArgsIfInteger| op argl domain tmp1 opprime ans mmList td
       tmp2 tmp3 tmp4 tmp5 tmp6 tmp7)
 (declare (special |$NumberOfArgsIfInteger| |$Expression| |$EmptyMode|))
    (setq op (car form))
    (setq argl (cdr form))
    (setq |$NumberOfArgsIfInteger| (|#| argl))
    (cond
     ((eq op '|error|)
      (list
       (cons op
        (dolist (x argl (nreverse0 tmp4))
          (setq tmp2 (|outputComp| x e))
          (setq e (third tmp2))
          (push (car tmp2) tmp4)))
       m e))
     ((and (pairp op) (eq (qcar op) '|elt|)
           (progn
            (setq tmp3 (qcdr op))
            (and (pairp tmp3)
                 (progn
                  (setq domain (qcar tmp3))
                  (setq tmp1 (qcdr tmp3))
                  (and (pairp tmp1)
                       (eq (qcdr tmp1) nil)
```

```
                    (progn
                     (setq opprime (qcar tmp1))
                     t))))))
          (cond
           ((eq domain '|Lisp|)
             (list
              (cons opprime
               (dolist (x argl (nreverse tmp7))
                (setq tmp2 (|compOrCroak| x |$EmptyMode| e))
                (setq e (third tmp2))
                (push (car tmp2) tmp7)))
              m e))
           ((and (equal domain |$Expression|) (eq opprime '|construct|))
             (|compExpressionList| argl m e))
           ((and (eq opprime 'collect) (|coerceable| domain m e))
             (when (setq td (|comp| (cons opprime argl) domain e))
              (|coerce| td m)))
           ((and (pairp domain) (eq (qcar domain) '|Mapping|)
                 (setq ans
                  (|compForm2| (cons opprime argl) m
                   (setq e (|augModemapsFromDomain1| domain domain e))
                   (dolist (x (|getFormModemaps| (cons opprime argl) e)
                                                 (nreverse0 tmp6))
                    (when
                     (and (pairp x)
                          (and (pairp (qcar x)) (equal (qcar (qcar x)) domain)))
                       (push x tmp6))))))
             ans)
           ((setq ans
             (|compForm2| (cons opprime argl) m
              (setq e (|addDomain| domain e))
              (dolist (x (|getFormModemaps| (cons opprime argl) e)
                         (nreverse0 tmp5))
                (when
                  (and (pairp x)
                       (and (pairp (qcar x)) (equal (qcar (qcar x)) domain)))
                   (push x tmp5)))))
             ans)
           ((and (eq opprime '|construct|) (|coerceable| domain m e))
             (when (setq td (|comp| (cons opprime argl) domain e))
              (|coerce| td m)))
           (t nil)))
         (t
          (setq e (|addDomain| m e))
          (cond
           ((and (setq mmList (|getFormModemaps| form e))
```

```
        (setq td (|compForm2| form m e mmList)))
    td)
  (t
    (|compToApply| op argl m e)))))))
```

1.1.29 defun compForm2

[take p??]
[length p??]
[nreverse0 p??]
[sublis p??]
[assoc p??]
[PredImplies p??]
[isSimple p??]
[compUniquely p??]
[compFormPartiallyBottomUp p??]
[compForm3 p??]
[$EmptyMode p??]
[$TriangleVariableList p??]

⟨*defun compForm2*⟩≡

```
(defun |compForm2| (form m e modemapList)
 (let (op argl sargl aList dc cond nsig v ncond deleteList newList td tl
       partialModeList tmp1 tmp2 tmp3 tmp4 tmp5 tmp6 tmp7 tmp8 tmp9 tmpb tmpc)
 (declare (special |$EmptyMode| |$TriangleVariableList|))
  (setq op (car form))
  (setq argl (cdr form))
  (setq sargl (take (|#| argl) |$TriangleVariableList|))
  (setq aList (mapcar #'(lambda (x y) (cons x y)) sargl argl))
  (setq modemaplist (sublis aList modemapList))
  ; now delete any modemaps that are subsumed by something else, provided
  ; the conditions are right (i.e. subsumer true whenever subsumee true)
  (dolist (u modemapList)
   (cond
    ((and (pairp u)
          (progn
           (setq tmp6 (qcar u))
           (and (pairp tmp6) (progn (setq dc (qcar tmp6)) t)))
          (progn
           (setq tmp7 (qcdr u))
           (and (pairp tmp7) (eq (qcdr tmp7) nil)
                (progn
                 (setq tmp1 (qcar tmp7))
                 (and (pairp tmp1)
                      (progn
                       (setq cond (qcar tmp1))
                       (setq tmp2 (qcdr tmp1))
                       (and (pairp tmp2) (eq (qcdr tmp2) nil)
                            (progn
                             (setq tmp3 (qcar tmp2))
```

```
                                (and (pairp tmp3) (eq (qcar tmp3) '|Subsumed|)
                                     (progn
                                      (setq tmp4 (qcdr tmp3))
                                      (and (pairp tmp4)
                                           (progn
                                            (setq tmp5 (qcdr tmp4))
                                            (and (pairp tmp5)
                                                 (eq (qcdr tmp5) nil)
                                                 (progn
                                                  (setq nsig (qcar tmp5))
                                                  t)))))))))))
                        (setq v (|assoc| (cons dc nsig) modemapList))
                        (pairp v)
                        (progn
                         (setq tmp6 (qcdr v))
                         (and (pairp tmp6) (eq (qcdr tmp6) nil)
                              (progn
                               (setq tmp7 (qcar tmp6))
                               (and (pairp tmp7)
                                    (progn
                                     (setq ncond (qcar tmp7))
                                     t))))))
                    (setq deleteList (cons u deleteList))
                    (unless  (|PredImplies| ncond cond)
                       (setq newList (push `(,(car u) (,cond (elt ,dc nil))) newList))))))
                (when deleteList
                 (setq modemapList
                  (remove-if #'(lambda (x) (member x deletelist)) modemapList)))
                ; it is important that subsumed ops (newList) be considered last
                (when newList (setq modemapList (append modemapList newList)))
                (setq tl
                 (loop for x in argl
                       while (and (|isSimple| x)
                                  (setq td (|compUniquely| x |$EmptyMode| e)))
                       collect td
                       do (setq e (third td))))
                (cond
                 ((some #'identity tl)
                   (setq partialModeList (loop for x in tl collect (when x (second x))))
                   (or (|compFormPartiallyBottomUp| form m e modemapList partialModeList)
                       (|compForm3| form m e modemapList)))
                 (t (|compForm3| form m e modemapList)))))
```

1.1.30 defun compArgumentsAndTryAgain

[comp p26]
[compForm1 p44]
[$EmptyMode p**??**]

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

1.1.31 defun compWithMappingMode

[compWithMappingMode1 p50]
[$formalArgList p**??**]

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

1.1.32 defun compWithMappingMode1

```
;compWithMappingMode1(x,m is ["Mapping",m',:sl],oldE,$formalArgList) ==
;  $killOptimizeIfTrue: local:= true
;  e:= oldE
;  isFunctor x =>
;    if get(x,"modemap",$CategoryFrame) is {\tt{}[.,target,:argModeList],.],:.]\ and\nwnewline
;\ \ \ \ \ (and/[extendsCategoryForm("{\char36}",s,mode)\ for\ mode\ in\ argModeList\ for\ s\ in
;\ \ \ \ \ \ \ )\ and\ extendsCategoryForm("{\char36}",target,m')\ then\ return\ [x,m,e]\nwnewli
;\ \ if\ STRINGP\ x\ then\ x:=\ INTERN\ x\nwnewline
;\ \ ress:=nil\nwnewline
;\ \ old{\char95}style:=true\nwnewline
;\ \ if\ x\ is\ ["+->",vl,nx]\ then\nwnewline
;\ \ \ \ old{\char95}style:=false\nwnewline
;\ \ \ \ vl\ is\ [":",:.]\ =>\nwnewline
;\ \ \ \ \ \ ress:=compLambda(x,m,oldE)\nwnewline
;\ \ \ \ \ \ ress\nwnewline
;\ \ \ \ vl:=\nwnewline
;\ \ \ \ \ \ vl\ is\ ["Tuple",:vl1]\ =>\ vl1\nwnewline
;\ \ \ \ \ \ vl\nwnewline
;\ \ \ \ vl:=\nwnewline
;\ \ \ \ \ \ SYMBOLP(vl)\ =>\ [vl]\nwnewline
;\ \ \ \ \ \ LISTP(vl)\ and\ (and/[SYMBOLP(v)\ for\ v\ in\ vl])\ =>\ vl\nwnewline
;\ \ \ \ \ \ stackAndThrow\ ["bad\ +->\ arguments:",vl]\nwnewline
;\ \ \ \ {\char36}formatArgList:=[:vl,:{\char36}formalArgList]\nwnewline
;\ \ \ \ x:=nx\nwnewline
;\ \ else\nwnewline
;\ \ \ \ vl:=take({\char35}sl,{\char36}FormalMapVariableList)\nwnewline
;\ \ ress\ =>\ ress\nwnewline
;\ \ for\ m\ in\ sl\ for\ v\ in\ vl\ repeat\nwnewline
;\ \ \ \ [.,.,e]:=\ compMakeDeclaration([":",v,m],{\char36}EmptyMode,e)\nwnewline
;\ \ old{\char95}style\ and\ not\ null\ vl\ and\ not\ hasFormalMapVariable(x,\ vl)\ =>\ return\n
;\ \ \ \ [u,.,.]\ :=\ comp([x,:vl],m',e)\ or\ return\ nil\nwnewline
;\ \ \ \ extractCodeAndConstructTriple(u,\ m,\ oldE)\nwnewline
;\ \ null\ vl\ and\ (t\ :=\ comp([x],\ m',\ e))\ =>\ return\nwnewline
;\ \ \ \ [u,.,.]\ :=\ t\nwnewline
;\ \ \ \ extractCodeAndConstructTriple(u,\ m,\ oldE)\nwnewline
;\ \ [u,.,.]:=\ comp(x,m',e)\ or\ return\ nil\nwnewline
;\ \ uu:=optimizeFunctionDef\ [nil,['LAMBDA,vl,u}
;  --  At this point, we have a function that we would like to pass.
;  --  Unfortunately, it makes various free variable references outside
;  --  itself.  So we build a mini-vector that contains them all, and
;  --  pass this as the environment to our inner function.
;  $FUNNAME :local := nil
;  $FUNNAME__TAIL :local := [nil]
;  expandedFunction:=COMP_-TRAN CADR uu
;  frees:=freelist(expandedFunction,vl,nil,e)
;    where freelist(u,bound,free,e) ==
;      atom u =>
;        not IDENTP u => free
```

```
;        MEMQ(u,bound) => free
;        v:=ASSQ(u,free) =>
;          RPLACD(v,1+CDR v)
;          free
;        not getmode(u, e) => free
;        {\tt{}u,:1],:free]\nwnewline
;\ \ \ \ \ \ op:=CAR\ u\nwnewline
;\ \ \ \ \ \ MEMQ(op,\ '(QUOTE\ GO\ function))\ =>\ free\nwnewline
;\ \ \ \ \ \ EQ(op,'LAMBDA)\ =>\nwnewline
;\ \ \ \ \ \ \ \ bound:=UNIONQ(bound,CADR\ u)\nwnewline
;\ \ \ \ \ \ \ \ for\ v\ in\ CDDR\ u\ repeat\nwnewline
;\ \ \ \ \ \ \ \ \ \ free:=freelist(v,bound,free,e)\nwnewline
;\ \ \ \ \ \ \ \ free\nwnewline
;\ \ \ \ \ \ EQ(op,'PROG)\ =>\nwnewline
;\ \ \ \ \ \ \ \ bound:=UNIONQ(bound,CADR\ u)\nwnewline
;\ \ \ \ \ \ \ \ for\ v\ in\ CDDR\ u\ |\ NOT\ ATOM\ v\ repeat\nwnewline
;\ \ \ \ \ \ \ \ \ \ free:=freelist(v,bound,free,e)\nwnewline
;\ \ \ \ \ \ \ \ free\nwnewline
;\ \ \ \ \ \ EQ(op,'SEQ)\ =>\nwnewline
;\ \ \ \ \ \ \ \ for\ v\ in\ CDR\ u\ |\ NOT\ ATOM\ v\ repeat\nwnewline
;\ \ \ \ \ \ \ \ \ \ free:=freelist(v,bound,free,e)\nwnewline
;\ \ \ \ \ \ \ \ free\nwnewline
;\ \ \ \ \ \ EQ(op,'COND)\ =>\nwnewline
;\ \ \ \ \ \ \ \ for\ v\ in\ CDR\ u\ repeat\nwnewline
;\ \ \ \ \ \ \ \ \ \ for\ vv\ in\ v\ repeat\nwnewline
;\ \ \ \ \ \ \ \ \ \ \ \ free:=freelist(vv,bound,free,e)\nwnewline
;\ \ \ \ \ \ \ \ free\nwnewline
;\ \ \ \ \ \ if\ ATOM\ op\ then\ u:=CDR\ u\ \ --Atomic\ functions\ aren't\ descended\nwnewline
;\ \ \ \ \ \ for\ v\ in\ u\ repeat\nwnewline
;\ \ \ \ \ \ \ \ free:=freelist(v,bound,free,e)\nwnewline
;\ \ \ \ \ \ free\nwnewline
;\ \ expandedFunction\ :=\nwnewline
;\ \ \ \ \ \ \ \ \ \ --One\ free\ can\ go\ by\ itself,\ more\ than\ one\ needs\ a\ vector\nwnewline
;\ \ \ \ \ \ \ \ --An\ A-list\ name\ .\ number\ of\ times\ used\nwnewline
;\ \ \ \ {\char35}frees\ =\ 0\ =>\ ['LAMBDA,[:vl,"{\char36}{\char36}"],\ :CDDR\ expandedFunction]\nwnewline
;\ \ \ \ {\char35}frees\ =\ 1\ =>\nwnewline
;\ \ \ \ \ \ vec:=first\ first\ frees\nwnewline
;\ \ \ \ \ \ ['LAMBDA,[:vl,vec],\ :CDDR\ expandedFunction]\nwnewline
;\ \ \ \ scode:=nil\nwnewline
;\ \ \ \ vec:=nil\nwnewline
;\ \ \ \ locals:=nil\nwnewline
;\ \ \ \ i:=-1\nwnewline
;\ \ \ \ for\ v\ in\ frees\ repeat\nwnewline
;\ \ \ \ \ \ i:=i+1\nwnewline
;\ \ \ \ \ \ vec:=[first\ v,:vec]\nwnewline
;\ \ \ \ \ \ scode:=[['SETQ,first\ v,[({\char36}QuickCode\ =>\ 'QREFELT;'ELT),"{\char36}{\char36}",i},:scod
;      locals:=[first v,:locals]
;    body:=CDDR expandedFunction
;    if locals then
;      if body is {\tt{}'DECLARE,:.],:.]\ then\nwnewline
```

```
;\ \ \ \ \ \ \ \ body:=[CAR\ body,['PROG,locals,:scode,['RETURN,['PROGN,:CDR\ body]]}
;      else body:={\tt{}'PROG,locals,:scode,['RETURN,['PROGN,:body]]}
;    vec:=['VECTOR,:NREVERSE vec]
;    ['LAMBDA,[:vl,"$$"],:body]
;  fname:=['CLOSEDFN,expandedFunction]
;         --Like QUOTE, but gets compiled
;  uu:=
;    frees => ['CONS,fname,vec]
;    ['LIST,fname]
;  [uu,m,oldE]
```

[isFunctor p??]
[get p??]
[qcar p??]
[qcdr p??]
[extendsCategoryForm p??]
[compLambda p59]
[stackAndThrow p??]
[take p??]
[compMakeDeclaration p62]
[hasFormalMapVariable p58]
[comp p26]
[extractCodeAndConstructTriple p58]
[optimizeFunctionDef p??]
[comp-tran p??]
[freelist p63]
[$formalArgList p??]
[$killOptimizeIfTrue p??]
[$funname p??]
[$funnameTail p??]
[$QuickCode p??]
[$EmptyMode p??]
[$FormalMapVariableList p??]
[$CategoryFrame p??]

⟨*defun compWithMappingMode1*⟩≡

```
(defun |compWithMappingMode1| (x m oldE |$formalArgList|)
 (declare (special |$formalArgList|))
  (prog (|$killOptimizeIfTrue| $funname $funnameTail mprime sl tmp1 tmp2
         tmp3 tmp4 tmp5 tmp6 target argModeList nx oldstyle ress vl1 vl e tt
             u frees i scode locals body vec expandedFunction fname uu)
  (declare (special |$killOptimizeIfTrue| $funname $funnameTail
                    |$QuickCode| |$EmptyMode| |$FormalMapVariableList|
                    |$CategoryFrame|))
   (return
    (seq
```

```
(progn
 (setq mprime (cadr m))
 (setq sl (cddr m))
 (setq |$killOptimizeIfTrue| t)
 (setq e oldE)
 (cond
  ((|isFunctor| x)
   (cond
    ((and (progn
           (setq tmp1 (|get| x '|modemap| |$CategoryFrame|))
           (and (pairp tmp1)
                (progn
                 (setq tmp2 (qcar tmp1))
                 (and (pairp tmp2)
                      (progn
                       (setq tmp3 (qcar tmp2))
                       (and (pairp tmp3)
                            (progn
                             (setq tmp4 (qcdr tmp3))
                             (and (pairp tmp4)
                                  (progn
                                   (setq target (qcar tmp4))
                                   (setq argModeList (qcdr tmp4))
                                   t)))))
                      (progn
                       (setq tmp5 (qcdr tmp2))
                       (and (pairp tmp5) (eq (qcdr tmp5) nil)))))))))
          (prog (t1)
           (setq t1 t)
           (return
            (do ((t2 nil (null t1))
                 (t3 argModeList (cdr t3))
                 (mode nil)
                 (t4 sl (cdr t4))
                 (s nil))
                ((or t2 (atom t3)
                     (progn (setq mode (car t3)) nil)
                     (atom t4)
                     (progn (setq s (car t4)) nil))
                   t1)
              (seq (exit
                (setq t1
                 (and t1 (|extendsCategoryForm| '$ s mode))))))))
          (|extendsCategoryForm| '$ target mprime))
          (return (list x m e )))
    (t nil)))
```

```
(t
 (when (stringp x) (setq x (intern x)))
 (setq ress nil)
 (setq oldstyle t)
 (cond
  ((and (pairp x)
        (eq (qcar x) '+->)
        (progn
         (setq tmp1 (qcdr x))
         (and (pairp tmp1)
              (progn
               (setq vl (qcar tmp1))
               (setq tmp2 (qcdr tmp1))
               (and (pairp tmp2)
                    (eq (qcdr tmp2) nil)
                    (progn (setq nx (qcar tmp2)) t))))))
    (setq oldstyle nil)
    (cond
     ((and (pairp vl) (eq (qcar vl) '|:|))
       (setq ress (|compLambda| x m oldE))
       ress)
     (t
      (setq vl
       (cond
        ((and (pairp vl)
              (eq (qcar vl) '|@Tuple|)
              (progn (setq vl1 (qcdr vl)) t))
          vl1)
        (t vl)))
      (setq vl
       (cond
        ((symbolp vl) (cons vl nil))
        ((and
          (listp vl)
          (prog (t5)
           (setq t5 t)
           (return
            (do ((t7 nil (null t5))
                 (t6 vl (cdr t6))
                 (v nil))
                ((or t7 (atom t6) (progn (setq v (car t6)) nil)) t5)
              (seq
               (exit
                (setq t5 (and t5 (symbolp v))))))))
          vl)
         (t
```

```
                  (|stackAndThrow| (cons '|bad +-> arguments:| (list vl ))))))
              (setq |$formatArgList| (append vl |$formalArgList|))
              (setq x nx))))
           (t
            (setq vl (take (|#| sl) |$FormalMapVariableList|))))
          (cond
           (ress ress)
           (t
            (do ((t8 sl (cdr t8)) (m nil) (t9 vl (cdr t9)) (v nil))
                ((or (atom t8)
                     (progn (setq m (car t8)) nil)
                     (atom t9)
                     (progn (setq v (car t9)) nil))
                   nil)
                (seq (exit (progn
                 (setq tmp6
                  (|compMakeDeclaration| (list '|:| v m ) |$EmptyMode| e))
                 (setq e (caddr tmp6))
                 tmp6))))
              (cond
               ((and oldstyle
                     (null (null vl))
                     (null (|hasFormalMapVariable| x vl)))
                (return
                 (progn
                  (setq tmp6 (or (|comp| (cons x vl) mprime e) (return nil)))
                  (setq u (car tmp6))
                  (|extractCodeAndConstructTriple| u m oldE))))
               ((and (null vl) (setq tt (|comp| (cons x nil) mprime e)))
                (return
                 (progn
                   (setq u (car tt))
                   (|extractCodeAndConstructTriple| u m oldE))))
               (t
                (setq tmp6 (or (|comp| x mprime e) (return nil)))
                (setq u (car tmp6))
                (setq uu (|optimizeFunctionDef| `(nil (lambda ,vl ,u))))
;  --  At this point, we have a function that we would like to pass.
;  --  Unfortunately, it makes various free variable references outside
;  --  itself.  So we build a mini-vector that contains them all, and
;  --  pass this as the environment to our inner function.
                (setq $funname nil)
                (setq $funnameTail (list nil))
                (setq expandedFunction (comp-tran (cadr uu)))
                (setq frees (freelist expandedFunction vl nil e))
                (setq expandedFunction
```

```
(cond
 ((eql (|#| frees) 0)
  (cons 'lambda (cons (append vl (list '$$))
                      (cddr expandedFunction))))
 ((eql (|#| frees) 1)
  (setq vec (caar frees))
   (cons 'lambda (cons (append vl (list vec))
                      (cddr expandedFunction))))
 (t
  (setq scode nil)
  (setq vec nil)
  (setq locals nil)
  (setq i -1)
  (do ((t0 frees (cdr t0)) (v nil))
      ((or (atom t0) (progn (setq v (car t0)) nil)) nil)
    (seq
     (exit
      (progn
       (setq i (plus i 1))
       (setq vec (cons (car v) vec))
       (setq scode
        (cons
         (cons 'setq
          (cons (car v)
           (cons
            (cons
             (cond
              (|$QuickCode| 'qrefelt)
              (t 'elt))
             (cons '$$ (cons i nil)))
            nil)))
          scode))
       (setq locals (cons (car v) locals))))))
  (setq body (cddr expandedFunction))
  (cond
   (locals
    (cond
     ((and (pairp body)
           (progn
            (setq tmp1 (qcar body))
            (and (pairp tmp1)
                 (eq (qcar tmp1) 'declare))))
      (setq body
       (cons (car body)
        (cons
         (cons 'prog
```

```
                (cons locals
                 (append scode
                  (cons
                   (cons 'return
                    (cons
                     (cons 'progn
                      (cdr body))
                     nil))
                    nil))))
                nil))))
          (t
           (setq body
            (cons
             (cons 'prog
              (cons locals
               (append scode
                (cons
                 (cons 'return
                  (cons
                   (cons 'progn body)
                   nil))
                  nil))))
              nil))))))
     (setq vec (cons 'vector (nreverse vec)))
     (cons 'lambda (cons (append vl (list '$$)) body)))))
(setq fname (list 'closedfn expandedFunction))
(setq uu
 (cond
  (frees (list 'cons fname vec))
  (t (list 'list fname))))
(list uu m oldE))))))))))))
```

1.1.33 defun extractCodeAndConstructTriple

⟨*defun extractCodeAndConstructTriple*⟩≡

```
(defun |extractCodeAndConstructTriple| (u m oldE)
 (let (tmp1 a fn op env)
  (cond
   ((and (pairp u) (eq (qcar u) '|call|)
         (progn
          (setq tmp1 (qcdr u))
          (and (pairp tmp1)
               (progn (setq fn (qcar tmp1)) t))))
    (cond
     ((and (pairp fn) (eq (qcar fn) '|applyFun|)
           (progn
            (setq tmp1 (qcdr fn))
            (and (pairp tmp1) (eq (qcdr tmp1) nil)
                 (progn (setq a (qcar tmp1)) t))))
            (setq fn a)))
     (list fn m oldE))
    (t
     (setq op (car u))
     (setq env (car (reverse (cdr u))))
     (list (list 'cons  (list '|function| op) env) m oldE)))))
```

1.1.34 defun hasFormalMapVariable

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

⟨*defun hasFormalMapVariable*⟩≡

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

1.1.35 defun compLambda

[qcar p??]
[qcdr p??]
[argsToSig p61]
[compAtSign p60]
[stackAndThrow p??]

⟨*defun compLambda*⟩≡

```
(defun |compLambda| (x m e)
 (let (vl body tmp1 tmp2 tmp3 target a1 args arg1 sig1 ress)
  (setq vl (cadr x))
  (setq body (caddr x))
  (cond
   ((and (pairp vl) (eq (qcar vl) '|:|)
         (progn
          (setq tmp1 (qcdr vl))
          (and (pairp tmp1)
               (progn
                (setq args (qcar tmp1))
                (setq tmp2 (qcdr tmp1))
                (and (pairp tmp2)
                     (eq (qcdr tmp2) nil)
                     (progn
                      (setq target (qcar tmp2))
                      t))))))
    (when (and (pairp args) (eq (qcar args) '|@Tuple|))
     (setq args (qcdr args)))
    (cond
     ((listp args)
      (setq tmp3 (|argsToSig| args))
      (setq arg1 (CAR tmp3))
      (setq sig1 (CADR tmp3))
      (cond
       (sig1
        (setq ress
         (|compAtSign|
          (list '@
           (list '+-> arg1 body)
           (cons '|Mapping| (cons target sig1))) m e))
         ress)
       (t (|stackAndThrow| (list '|compLambda| x )))))
     (t (|stackAndThrow| (list '|compLambda| x )))))
   (t (|stackAndThrow| (list '|compLambda| x ))))))
```

1.1.36 defun compAtSign

[addDomain p**??**]
[comp p26]
[coerce p**??**]

⟨*defun compAtSign*⟩≡

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

1.1.37 defun argsToSig

⟨*defun argsToSig*⟩≡

```
(defun |argsToSig| (args)
 (let (tmp1 v tmp2 tt sig1 arg1 bad)
  (cond
   ((and (pairp args) (eq (qcar args) '|:|)
         (progn
          (setq tmp1 (qcdr args))
          (and (pairp tmp1)
               (progn
                (setq v (qcar tmp1))
                (setq tmp2 (qcdr tmp1))
                (and (pairp tmp2)
                     (eq (qcdr tmp2) nil)
                     (progn
                      (setq tt (qcar tmp2))
                      t))))))
    (list (list v) (list tt)))
   (t
    (setq sig1 nil)
    (setq arg1 nil)
    (setq bad nil)
    (dolist (arg args)
      (cond
       ((and (pairp arg) (eq (qcar arg) '|:|)
             (progn
              (setq tmp1 (qcdr arg))
              (and (pairp tmp1)
                   (progn
                    (setq v (qcar tmp1))
                    (setq tmp2 (qcdr tmp1))
                    (and (pairp tmp2) (eq (qcdr tmp2) nil)
                         (progn
                          (setq tt (qcar tmp2))
                          t))))))
        (setq sig1 (cons tt sig1))
        (setq arg1 (cons v arg1)))
       (t (setq bad t))))
    (cond
     (bad (list nil nil ))
     (t (list (reverse arg1) (reverse sig1)))))))))
```

1.1.38 defun compMakeDeclaration

[compColon p32]
[$insideExpressionIfTrue p**??**]

⟨*defun compMakeDeclaration*⟩≡

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

1.1.39 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 p63]
[assq p**??**]
[identp p**??**]
[getmode p**??**]
[unionq p**??**]

⟨*defun freelist*⟩≡
```
(defun freelist (u bound free e)
 (let (v op)
  (if (atom u)
   (cond
    ((null (identp u)) free)
    ((member u bound) free)
    ; more than 1 free becomes alist (name . number)
    ((setq v (assq u free)) (rplacd v (+ 1 (cdr v))) free)
    ((null (|getmode| u e)) free)
    (t (cons (cons u 1) free)))
   (progn
    (setq op (car u))
    (cond
     ((member op '(quote go |function|)) free)
     ((eq op 'lambda) ; lambdas bind symbols
      (setq bound (unionq bound (cadr u)))
      (dolist (v (cddr u))
       (setq free (freelist v bound free e))))
     ((eq op 'prog) ; progs bind symbols
      (setq bound (unionq bound (cadr 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))))
```

1.1.40 defun compOrCroak1,compactify

[compOrCroak1,compactify p64]
[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))))))
```

1.1.41 defun Compiler/Interpreter interface

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

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

1.1.42 defun /RQ,LIB

Compile a library quietly [/rf-1(5) p**??**]
[echo-meta(5) p**??**]
[$lisplib p**??**]

⟨*defun /RQ,LIB*⟩≡
```
(defun |/RQ,LIB| (&rest foo &aux (echo-meta nil) ($lisplib t))
 (declare (special echo-meta $lisplib))
  (/rf-1 nil))
```

1.1.43 defun compileSpadLispCmd

[pathname(5) p??]
[pathnameType(5) p??]
[selectOptionLC(5) p??]
[namestring(5) p??]
[terminateSystemCommand(5) p??]
[fnameMake(5) p??]
[pathnameDirectory(5) p??]
[pathnameName(5) p??]
[fnameReadable?(5) p??]
[localdatabase(5) p??]
[throwKeyedMsg p??]
[object2String p??]
[sayKeyedMsg p??]
[recompile-lib-file-if-necessary p66]
[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|)))))
```

1.1.44 defun recompile-lib-file-if-necessary

[compile-lib-file p67]
[*lisp-bin-filetype* p??]

⟨*defun recompile-lib-file-if-necessary*⟩≡

```
(defun recompile-lib-file-if-necessary (lfile)
 (let* ((bfile (make-pathname :type *lisp-bin-filetype* :defaults lfile))
        (bdate (and (probe-file bfile) (file-write-date bfile)))
        (ldate (and (probe-file lfile) (file-write-date lfile))))
  (unless (and ldate bdate (> bdate ldate))
   (compile-lib-file lfile)
   (list bfile))))
```

1.1.45 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)
```

1.1.46 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)))
```

1.1.47 defun compileAsharpCmd

⟨*defun compileAsharpCmd*⟩≡
```
(defun |compileAsharpCmd| (args)
 (error "compileAsharpCmd is no longer supported"))
```

1.1.48 defun compileAsharpCmd1

⟨*defun compileAsharpCmd1*⟩≡
```
(defun |compileAsharpCmd1| (args)
 (error "compileAsharpCmd1 is no longer supported"))
```

1.1.49 defun compileAsharpArchiveCmd

⟨*defun compileAsharpArchiveCmd*⟩≡
```
(defun |compileAsharpArchiveCmd| (args)
 (error "compileAsharpArchiveCmd is no longer supported"))
```

1.1.50 defun compileAsharpLispCmd

[error p**??**]

⟨*defun compileAsharpLispCmd*⟩≡
```
(defun |compileAsharpLispCmd| (args)
 (error "compileAsharpLispCmd is no longer supported"))
```

1.1.51 defun withAsharpCmd

⟨*defun withAsharpCmd*⟩≡
```
(defun |withAsharpCmd| (args)
 (error "withAsharpCmd is no longer supported"))
```

1.1.52 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)))
```

1.1.53 defvar $byConstructors

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

1.1.54 defvar $constructorsSeen

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

Chapter 2

The Compiler

⟨*Compiler*⟩≡
```
(in-package "BOOT")
```

⟨*initvars*⟩

⟨*defun argsToSig*⟩
⟨*defun comp*⟩
⟨*defun comp2*⟩
⟨*defun comp3*⟩
⟨*defun compArgumentsAndTryAgain*⟩
⟨*defun compAtom*⟩
⟨*defun compAtSign*⟩
⟨*defun compColon*⟩
⟨*defun compColonInside*⟩
⟨*defun compExpression*⟩
⟨*defun compForm*⟩
⟨*defun compForm1*⟩
⟨*defun compForm2*⟩
⟨*defun compLambda*⟩
⟨*defun compileAsharpArchiveCmd*⟩
⟨*defun compileAsharpCmd*⟩
⟨*defun compileAsharpCmd1*⟩
⟨*defun compileAsharpLispCmd*⟩
⟨*defun compileFileQuietly*⟩
⟨*defun compile-lib-file*⟩
⟨*defun compiler*⟩
⟨*defun compilerDoit*⟩
⟨*defun compileSpad2Cmd*⟩
⟨*defun compileSpadLispCmd*⟩
⟨*defun compList*⟩

⟨*defun compMakeDeclaration*⟩
⟨*defun compNoStacking*⟩
⟨*defun compNoStacking1*⟩
⟨*defun compOrCroak*⟩
⟨*defun compOrCroak1*⟩
⟨*defun compOrCroak1,compactify*⟩
⟨*defun compSymbol*⟩
⟨*defun compTopLevel*⟩
⟨*defun compTypeOf*⟩
⟨*defun compVector*⟩
⟨*defun compWithMappingMode*⟩
⟨*defun compWithMappingMode1*⟩
⟨*defun convert*⟩

⟨*defun extractCodeAndConstructTriple*⟩

⟨*defun freelist*⟩

⟨*defun hasFormalMapVariable*⟩

⟨*defun ncINTERPFILE*⟩

⟨*defun primitiveType*⟩

⟨*defun recompile-lib-file-if-necessary*⟩
⟨*defun /rf-1*⟩
⟨*defun /RQ,LIB*⟩

⟨*defun spad*⟩
⟨*defun spad-fixed-arg*⟩
⟨*defun s-process*⟩

⟨*defun withAsharpCmd*⟩

Bibliography

[1] Jenks, R.J. and Sutor, R.S. "Axiom – The Scientific Computation System" Springer-Verlag New York (1992) ISBN 0-387-97855-0

[2] Knuth, Donald E., "Literate Programming" Center for the Study of Language and Information ISBN 0-937073-81-4 Stanford CA (1992)

[3] Daly, Timothy, "The Axiom Wiki Website"
http://axiom.axiom-developer.org

[4] Watt, Stephen, "Aldor",
http://www.aldor.org

[5] Lamport, Leslie, "Latex – A Document Preparation System", Addison-Wesley, New York ISBN 0-201-52983-1

[6] Ramsey, Norman "Noweb – A Simple, Extensible Tool for Literate Programming"
http://www.eecs.harvard.edu/ ~nr/noweb

[7] Daly, Timothy, "The Axiom Literate Documentation"
http://axiom.axiom-developer.org/axiom-website/documentation.html

Chapter 3

Index

Index

comp370-apply
  usedby spad, 18
eof
  usedby spad, 18
lisp-bin-filetype
  usedby recompile-lib-file-if-necessary, 66
standard-output
  usedby compileFileQuietly, 68
/RQ,LIB, 64
  calls /rf-1(5), 64
  uses $lisplib, 64
  uses echo-meta(5), 64
  defun, 64
/editfile
  usedby /rf-1, 17
  usedby compileSpad2Cmd, 13
  usedby compiler, 9
  usedby spad, 18
/rf(5)
  calledby compilerDoit, 16
/rf-1, 17
  calls makeInputFilename(5), 17
  calls ncINTERPFILE, 17
  calls spad, 17
  uses /editfile, 17
  uses echo-meta, 17
  defun, 17
/rf-1(5)
  calledby /RQ,LIB, 64
/rq(5)
  calledby compilerDoit, 16
$Boolean
  usedby compSymbol, 40
$CategoryFrame
  usedby compWithMappingMode1, 52
$DomainFrame
  usedby s-process, 20
$DoubleFloat
  usedby primitiveType, 39
$EchoLines
  usedby ncINTERPFILE, 64
$EmptyEnvironment
  usedby s-process, 20
$EmptyMode
  usedby compArgumentsAndTryAgain, 49
  usedby compColonInside, 36
  usedby compForm1, 44
  usedby compForm2, 47
  usedby compNoStacking, 27
  usedby compWithMappingMode1, 52
  usedby primitiveType, 39
  usedby s-process, 20
$EmptyVector
  usedby compVector, 42
$Expression
  usedby compAtom, 37
  usedby compForm1, 44
  usedby compSymbol, 40
$FormalMapVariableList
  usedby compColon, 33
  usedby compSymbol, 40
  usedby compTypeOf, 31
  usedby compWithMappingMode1, 52
$Index
  usedby s-process, 20
$InitialDomainsInScope
  usedby spad, 18
$InteractiveFrame
  usedby spad, 18

$InteractiveMode
usedby compileFileQuietly, 68
usedby compileSpad2Cmd, 13
usedby spad, 18
$LocalFrame
usedby s-process, 21
$NRTderivedTargetIfTrue
usedby compTopLevel, 23
$NegativeInteger
usedby primitiveType, 39
$NoValueMode
usedby compSymbol, 40
$NoValue
usedby compSymbol, 40
$NonNegativeInteger
usedby primitiveType, 39
$NumberOfArgsIfInteger
usedby compForm1, 44
$PolyMode
usedby s-process, 20
$PositiveInteger
usedby primitiveType, 39
$QuickCode
usedby compWithMappingMode1, 52
usedby compileSpad2Cmd, 13
$QuickLet
usedby compileSpad2Cmd, 13
$ReadingFile
usedby ncINTERPFILE, 64
$Representation
usedby compNoStacking, 27
$String
usedby primitiveType, 39
$Symbol
usedby compSymbol, 40
$TriangleVariableList
usedby compForm2, 47
$VariableCount
usedby s-process, 20
$bootStrapMode
usedby comp2, 28
usedby compColon, 33
$byConstructors, 68
usedby compilerDoit, 16
defvar, 68
$compErrorMessageStack
usedby compOrCroak1, 25
$compForModeIfTrue
usedby compSymbol, 40
$compStack
usedby compNoStacking1, 27
usedby compNoStacking, 27
usedby compOrCroak1, 25
usedby comp, 26
$compTimeSum
usedby compTopLevel, 23
$compUniquelyIfTrue
usedby s-process, 20
$compileOnlyCertainItems
usedby compileSpad2Cmd, 13
$constructorsSeen, 68
usedby compilerDoit, 16
defvar, 68
$currentFunction
usedby s-process, 20
$envHashTable
usedby compTopLevel, 23
$exitModeStack
usedby compOrCroak1, 25
usedby comp, 26
usedby s-process, 20
$exitMode
usedby s-process, 20
$e
usedby comp3, 29
usedby s-process, 20
$forceAdd
usedby compTopLevel, 23
$formalArgList
usedby compSymbol, 40
usedby compWithMappingMode1, 52
usedby compWithMappingMode, 49
$formalMapVariables
usedby hasFormalMapVariable, 58
$form
usedby s-process, 20
$functorLocalParameters
usedby compSymbol, 40
$funnameTail

usedby compWithMappingMode1, 52
$funname
usedby compWithMappingMode1, 52
$f
usedby compileSpad2Cmd, 13
$genFVar
usedby s-process, 20
$genSDVar
usedby s-process, 20
$insideCapsuleFunctionIfTrue
usedby s-process, 20
$insideCategoryIfTrue
usedby compColon, 33
usedby s-process, 20
$insideCoerceInteractiveHardIfTrue
usedby s-process, 20
$insideCompTypeOf
usedby comp3, 29
usedby compTypeOf, 31
$insideExpressionIfTrue
usedby compColon, 33
usedby compExpression, 43
usedby compMakeDeclaration, 62
usedby s-process, 20
$insideFunctorIfTrue
usedby compColon, 33
usedby s-process, 20
$insideWhereIfTrue
usedby s-process, 20
$killOptimizeIfTrue
usedby compTopLevel, 23
usedby compWithMappingMode1, 52
$leaveLevelStack
usedby s-process, 20
$leaveMode
usedby s-process, 20
$level
usedby compOrCroak1, 25
$lhsOfColon
usedby compColon, 33
$lisplib
usedby /RQ,LIB, 64
usedby comp2, 28
$macroassoc
usedby s-process, 20
$m
usedby compileSpad2Cmd, 13
$newCompilerUnionFlag
usedby compColonInside, 36
$newComp
usedby compileSpad2Cmd, 13
$newConlist
usedby compileSpad2Cmd, 13
usedby compiler, 9
$newspad
usedby s-process, 20
$noEnv
usedby compColon, 33
$noSubsumption
usedby spad, 18
$options
usedby compileSpad2Cmd, 13
usedby compileSpadLispCmd, 65
usedby compiler, 9
$packagesUsed
usedby comp2, 28
usedby compTopLevel, 23
$postStack
usedby s-process, 20
$previousTime
usedby s-process, 21
$resolveTimeSum
usedby compTopLevel, 23
$returnMode
usedby s-process, 20
$scanIfTrue
usedby compOrCroak1, 25
usedby compileSpad2Cmd, 13
$semanticErrorStack
usedby s-process, 20
$sourceFileTypes
usedby compileSpad2Cmd, 13
$s
usedby compOrCroak1, 25
$top-level
usedby s-process, 20
$topOp
usedby s-process, 20
$warningStack

usedby s-process, 20

addBinding
calledby spad, 18
addDomain
calledby comp2, 28
calledby comp3, 29
calledby compAtSign, 60
calledby compColonInside, 36
calledby compColon, 32
calledby compForm1, 44
applyMapping
calledby comp3, 29
argsToSig, 61
calledby compLambda, 59
defun, 61
assoc
calledby compColon, 32
calledby compForm2, 47
assq
calledby freelist, 63
augModemapsFromDomain1
calledby compForm1, 44

browserAutoloadOnceTrigger
calledby compileSpad2Cmd, 13

catches
compOrCroak1, 25
spad, 18
coerce
calledby compAtSign, 60
calledby compColonInside, 36
calledby compForm1, 44
calledby convert, 38
coerceable
calledby compForm1, 44
comp, 26
calledby compArgumentsAndTryAgain, 49
calledby compAtSign, 60
calledby compColonInside, 36
calledby compForm1, 44
calledby compList, 41
calledby compOrCroak1, 25
calledby compVector, 42
calledby compWithMappingMode1, 52
calls compNoStacking, 26
uses $compStack, 26
uses $exitModeStack, 26
defun, 26
comp-tran
calledby compWithMappingMode1, 52
comp2, 28
calledby compNoStacking1, 27
calledby compNoStacking, 27
calls addDomain, 28
calls comp3, 28
calls insert, 28
calls isDomainForm, 28
calls isFunctor, 28
calls nequal, 28
calls opOf, 28
uses $bootStrapMode, 28
uses $lisplib, 28
uses $packagesUsed, 28
defun, 28
comp3, 29
calledby comp2, 28
calledby compTypeOf, 31
calls addDomain, 29
calls applyMapping, 29
calls compApply, 29
calls compAtom, 29
calls compCoerce, 29
calls compColon, 29
calls compExpression, 29
calls compTypeOf, 29
calls compWithMappingMode, 29
calls getDomainsInScope, 29
calls getmode, 29
calls member, 29
calls pname, 29
calls stringPrefix?, 29
calls stringimage, 29
uses $e, 29
uses $insideCompTypeOf, 29
defun, 29
compApply
calledby comp3, 29

compArgumentsAndTryAgain, 49
calledby compForm, 43
calls compForm1, 49
calls comp, 49
uses $EmptyMode, 49
defun, 49
compAtom, 37
calledby comp3, 29
calls compAtomWithModemap, 37
calls compList, 37
calls compSymbol, 37
calls compVector, 37
calls convert, 37
calls get, 37
calls isSymbol, 37
calls modeIsAggregateOf, 37
calls primitiveType, 37
uses $Expression, 37
defun, 37
compAtomWithModemap
calledby compAtom, 37
compAtSign, 60
calledby compLambda, 59
calls addDomain, 60
calls coerce, 60
calls comp, 60
defun, 60
compCoerce
calledby comp3, 29
compColon, 32
calledby comp3, 29
calledby compColon, 32
calledby compMakeDeclaration, 62
calls addDomain, 32
calls assoc, 32
calls compColonInside, 32
calls compColon, 32
calls eqsubstlist, 32
calls genSomeVariable, 33
calls getDomainsInScope, 32
calls getmode, 32
calls isCategoryForm, 32
calls isDomainForm, 32
calls length, 32
calls makeCategoryForm, 33
calls member, 32
calls nreverse0, 32
calls put, 33
calls systemErrorHere, 33
calls take, 32
calls unknownTypeError, 32
uses $FormalMapVariableList, 33
uses $bootStrapMode, 33
uses $insideCategoryIfTrue, 33
uses $insideExpressionIfTrue, 33
uses $insideFunctorIfTrue, 33
uses $lhsOfColon, 33
uses $noEnv, 33
defun, 32
compColonInside, 36
calledby compColon, 32
calls addDomain, 36
calls coerce, 36
calls comp, 36
calls opOf, 36
calls stackSemanticError, 36
calls stackWarning, 36
uses $EmptyMode, 36
uses $newCompilerUnionFlag, 36
defun, 36
compExpression, 43
calledby comp3, 29
calls compForm, 43
calls getl, 43
uses $insideExpressionIfTrue, 43
defun, 43
compExpressionList
calledby compForm1, 44
compForm, 43
calledby compExpression, 43
calls compArgumentsAndTryAgain, 43
calls compForm1, 43
calls stackMessageIfNone, 43
defun, 43
compForm1, 44
calledby compArgumentsAndTryAgain, 49
calledby compForm, 43
calls addDomain, 44
calls augModemapsFromDomain1, 44

calls coerceable, 44
calls coerce, 44
calls compExpressionList, 44
calls compForm2, 44
calls compOrCroak, 44
calls compToApply, 44
calls comp, 44
calls getFormModemaps, 44
calls length, 44
calls nreverse0, 44
calls outputComp, 44
uses $EmptyMode, 44
uses $Expression, 44
uses $NumberOfArgsIfInteger, 44
defun, 44
compForm2, 47
calledby compForm1, 44
calls PredImplies, 47
calls assoc, 47
calls compForm3, 47
calls compFormPartiallyBottomUp, 47
calls compUniquely, 47
calls isSimple, 47
calls length, 47
calls nreverse0, 47
calls sublis, 47
calls take, 47
uses $EmptyMode, 47
uses $TriangleVariableList, 47
defun, 47
compForm3
calledby compForm2, 47
compFormPartiallyBottomUp
calledby compForm2, 47
compile-lib-file, 67
calledby recompile-lib-file-if-necessary, 66
defun, 67
compileAsharpArchiveCmd, 67
calledby compiler, 9
defun, 67
compileAsharpCmd, 67
calledby compiler, 9
defun, 67
compileAsharpCmd1, 67
defun, 67
compileAsharpLispCmd, 68
calledby compiler, 9
calls error, 68
defun, 68
compileFileQuietly, 68
uses *standard-output*, 68
uses $InteractiveMode, 68
defun, 68
compiler, 9
calls compileAsharpArchiveCmd, 9
calls compileAsharpCmd, 9
calls compileAsharpLispCmd, 9
calls compileSpad2Cmd, 9
calls compileSpadLispCmd, 9
calls findfile, 9
calls helpSpad2Cmd(5), 9
calls mergePathnames(5), 9
calls namestring(5), 9
calls pathname(5), 9
calls pathnameType(5), 9
calls selectOptionLC(5), 9
calls throwKeyedMsg, 9
uses /editfile, 9
uses $newConlist, 9
uses $options, 9
defun, 9
compilerDoit, 16
calledby compileSpad2Cmd, 13
calls /rf(5), 16
calls /rq(5), 16
calls member(5), 16
calls sayBrightly, 16
uses $byConstructors, 16
uses $constructorsSeen, 16
defun, 16
compilerDoitWithScreenedLisplib
calledby compileSpad2Cmd, 13
compileSpad2Cmd, 13
calledby compiler, 9
calls browserAutoloadOnceTrigger, 13
calls compilerDoitWithScreenedLisplib, 13
calls compilerDoit, 13
calls convertSpadToAsFile, 13

calls error, 13
calls extendLocalLibdb, 13
calls namestring(5), 13
calls nequal, 13
calls object2String, 13
calls oldParserAutoloadOnceTrigger, 13
calls pathname(5), 13
calls pathnameType(5), 13
calls sayKeyedMsg, 13
calls selectOptionLC(5), 13
calls spad2AsTranslatorAutoloadOnceTrigger, 13
calls spadPrompt, 13
calls strconc, 13
calls terminateSystemCommand(5), 13
calls throwKeyedMsg, 13
calls updateSourceFiles(5), 13
uses /editfile, 13
uses $InteractiveMode, 13
uses $QuickCode, 13
uses $QuickLet, 13
uses $compileOnlyCertainItems, 13
uses $f, 13
uses $m, 13
uses $newComp, 13
uses $newConlist, 13
uses $options, 13
uses $scanIfTrue, 13
uses $sourceFileTypes, 13
defun, 13
compileSpadLispCmd, 65
calledby compiler, 9
calls fnameMake(5), 65
calls fnameReadable?(5), 65
calls localdatabase(5), 65
calls namestring(5), 65
calls object2String, 65
calls pathname(5), 65
calls pathnameDirectory(5), 65
calls pathnameName(5), 65
calls pathnameType(5), 65
calls recompile-lib-file-if-necessary, 65
calls sayKeyedMsg, 65
calls selectOptionLC(5), 65
calls spadPrompt, 65
calls terminateSystemCommand(5), 65
calls throwKeyedMsg, 65
uses $options, 65
defun, 65
compLambda, 59
calledby compWithMappingMode1, 52
calls argsToSig, 59
calls compAtSign, 59
calls qcar, 59
calls qcdr, 59
calls stackAndThrow, 59
defun, 59
compList, 41
calledby compAtom, 37
calls comp, 41
defun, 41
compMakeDeclaration, 62
calledby compWithMappingMode1, 52
calls compColon, 62
uses $insideExpressionIfTrue, 62
defun, 62
compNoStacking, 27
calledby comp, 26
calls comp2, 27
calls compNoStacking1, 27
uses $EmptyMode, 27
uses $Representation, 27
uses $compStack, 27
defun, 27
compNoStacking1, 27
calledby compNoStacking, 27
calls comp2, 27
calls get, 27
uses $compStack, 27
defun, 27
compOrCroak, 24
calledby compForm1, 44
calledby compTopLevel, 23
calls compOrCroak1, 24
defun, 24
compOrCroak1, 25

calledby compOrCroak, 24
calls compOrCroak1,compactify, 25
calls comp, 25
calls displayComp, 25
calls displaySemanticErrors, 25
calls mkErrorExpr, 25
calls say, 25
calls stackSemanticError, 25
calls userError, 25
uses $compErrorMessageStack, 25
uses $compStack, 25
uses $exitModeStack, 25
uses $level, 25
uses $scanIfTrue, 25
uses $s, 25
catches, 25
defun, 25
compOrCroak1,compactify, 64
calledby compOrCroak1,compactify, 64
calledby compOrCroak1, 25
calls compOrCroak1,compactify, 64
calls lassoc, 64
defun, 64
compSymbol, 40
calledby compAtom, 37
calls NRTgetLocalIndex, 40
calls errorRef, 40
calls getmode, 40
calls get, 40
calls isFunction, 40
calls member, 40
calls stackMessage, 40
uses $Boolean, 40
uses $Expression, 40
uses $FormalMapVariableList, 40
uses $NoValueMode, 40
uses $NoValue, 40
uses $Symbol, 40
uses $compForModeIfTrue, 40
uses $formalArgList, 40
uses $functorLocalParameters, 40
defun, 40
compToApply
calledby compForm1, 44
compTopLevel, 23
calledby s-process, 20
calls compOrCroak, 23
calls newComp, 23
uses $NRTderivedTargetIfTrue, 23
uses $compTimeSum, 23
uses $envHashTable, 23
uses $forceAdd, 23
uses $killOptimizeIfTrue, 23
uses $packagesUsed, 23
uses $resolveTimeSum, 23
defun, 23
compTypeOf, 31
calledby comp3, 29
calls comp3, 31
calls eqsubstlist, 31
calls get, 31
calls put, 31
uses $FormalMapVariableList, 31
uses $insideCompTypeOf, 31
defun, 31
compUniquely
calledby compForm2, 47
compVector, 42
calledby compAtom, 37
calls comp, 42
uses $EmptyVector, 42
defun, 42
compWithMappingMode, 49
calledby comp3, 29
calls compWithMappingMode1, 49
uses $formalArgList, 49
defun, 49
compWithMappingMode1, 50
calledby compWithMappingMode, 49
calls comp-tran, 52
calls compLambda, 52
calls compMakeDeclaration, 52
calls comp, 52
calls extendsCategoryForm, 52
calls extractCodeAndConstructTriple, 52
calls freelist, 52
calls get, 52
calls hasFormalMapVariable, 52
calls isFunctor, 52

calls optimizeFunctionDef, 52
calls qcar, 52
calls qcdr, 52
calls stackAndThrow, 52
calls take, 52
uses $CategoryFrame, 52
uses $EmptyMode, 52
uses $FormalMapVariableList, 52
uses $QuickCode, 52
uses $formalArgList, 52
uses $funnameTail, 52
uses $funname, 52
uses $killOptimizeIfTrue, 52
defun, 50
convert, 38
calledby compAtom, 37
calls coerce, 38
calls resolve, 38
defun, 38
convertSpadToAsFile
calledby compileSpad2Cmd, 13
curoutstream
usedby s-process, 21
curstrm
calledby s-process, 20

def-process
calledby s-process, 20
def-rename
calledby s-process, 20
defun
/RQ,LIB, 64
/rf-1, 17
argsToSig, 61
comp, 26
comp2, 28
comp3, 29
compArgumentsAndTryAgain, 49
compAtom, 37
compAtSign, 60
compColon, 32
compColonInside, 36
compExpression, 43
compForm, 43
compForm1, 44
compForm2, 47
compile-lib-file, 67
compileAsharpArchiveCmd, 67
compileAsharpCmd, 67
compileAsharpCmd1, 67
compileAsharpLispCmd, 68
compileFileQuietly, 68
compiler, 9
compilerDoit, 16
compileSpad2Cmd, 13
compileSpadLispCmd, 65
compLambda, 59
compList, 41
compMakeDeclaration, 62
compNoStacking, 27
compNoStacking1, 27
compOrCroak, 24
compOrCroak1, 25
compOrCroak1,compactify, 64
compSymbol, 40
compTopLevel, 23
compTypeOf, 31
compVector, 42
compWithMappingMode, 49
compWithMappingMode1, 50
convert, 38
extractCodeAndConstructTriple, 58
freelist, 63
hasFormalMapVariable, 58
ncINTERPFILE, 64
primitiveType, 39
recompile-lib-file-if-necessary, 66
s-process, 20
spad, 18
spad-fixed-arg, 66
withAsharpCmd, 68
defvar
$byConstructors, 68
$constructorsSeen, 68
displayComp
calledby compOrCroak1, 25
displayPreCompilationErrors
calledby s-process, 20
displaySemanticErrors
calledby compOrCroak1, 25
calledby s-process, 20

echo-meta
usedby /rf-1, 17
usedby spad, 18
echo-meta(5)
usedby /RQ,LIB, 64
eqsubstlist
calledby compColon, 32
calledby compTypeOf, 31
error
calledby compileAsharpLispCmd, 68
calledby compileSpad2Cmd, 13
errorRef
calledby compSymbol, 40
extendLocalLibdb
calledby compileSpad2Cmd, 13
extendsCategoryForm
calledby compWithMappingMode1, 52
extractCodeAndConstructTriple, 58
calledby compWithMappingMode1, 52
defun, 58

file-closed
usedby spad, 18
findfile
calledby compiler, 9
fnameMake(5)
calledby compileSpadLispCmd, 65
fnameReadable?(5)
calledby compileSpadLispCmd, 65
freelist, 63
calledby compWithMappingMode1, 52
calledby freelist, 63
calls assq, 63
calls freelist, 63
calls getmode, 63
calls identp, 63
calls unionq, 63
defun, 63

genSomeVariable
calledby compColon, 33
get
calledby compAtom, 37
calledby compNoStacking1, 27
calledby compSymbol, 40
calledby compTypeOf, 31
calledby compWithMappingMode1, 52
get-internal-run-time
calledby s-process, 20
getDomainsInScope
calledby comp3, 29
calledby compColon, 32
getFormModemaps
calledby compForm1, 44
getl
calledby compExpression, 43
getmode
calledby comp3, 29
calledby compColon, 32
calledby compSymbol, 40
calledby freelist, 63

hasFormalMapVariable, 58
calledby compWithMappingMode1, 52
calls ScanOrPairVec, 58
uses $formalMapVariables, 58
defun, 58
helpSpad2Cmd(5)
calledby compiler, 9

identp
calledby freelist, 63
init-boot/spad-reader
calledby spad, 18
initialize-preparse
calledby spad, 18
insert
calledby comp2, 28
ioclear
calledby spad, 18
isCategoryForm
calledby compColon, 32
isDomainForm
calledby comp2, 28
calledby compColon, 32
isFunction

calledby compSymbol, 40
isFunctor
calledby comp2, 28
calledby compWithMappingMode1, 52
isSimple
calledby compForm2, 47
isSymbol
calledby compAtom, 37

lassoc
calledby compOrCroak1,compactify, 64
length
calledby compColon, 32
calledby compForm1, 44
calledby compForm2, 47
line
usedby spad, 18
localdatabase(5)
calledby compileSpadLispCmd, 65

makeCategoryForm
calledby compColon, 33
makeInitialModemapFrame
calledby spad, 18
makeInputFilename(5)
calledby /rf-1, 17
member
calledby comp3, 29
calledby compColon, 32
calledby compSymbol, 40
member(5)
calledby compilerDoit, 16
mergePathnames(5)
calledby compiler, 9
mkErrorExpr
calledby compOrCroak1, 25
modeIsAggregateOf
calledby compAtom, 37

namestring(5)
calledby compileSpad2Cmd, 13
calledby compileSpadLispCmd, 65
calledby compiler, 9
ncINTERPFILE, 64
calledby /rf-1, 17
calls SpadInterpretStream(5), 64
uses $EchoLines, 64
uses $ReadingFile, 64
defun, 64
nequal
calledby comp2, 28
calledby compileSpad2Cmd, 13
new2OldLisp
calledby s-process, 20
newComp
calledby compTopLevel, 23
nreverse0
calledby compColon, 32
calledby compForm1, 44
calledby compForm2, 47
NRTgetLocalIndex
calledby compSymbol, 40

object2String
calledby compileSpad2Cmd, 13
calledby compileSpadLispCmd, 65
oldParserAutoloadOnceTrigger
calledby compileSpad2Cmd, 13
opOf
calledby comp2, 28
calledby compColonInside, 36
optimizeFunctionDef
calledby compWithMappingMode1, 52
outputComp
calledby compForm1, 44

PARSE-NewExpr
calledby spad, 18
parseTransform
calledby s-process, 20
pathname(5)
calledby compileSpad2Cmd, 13
calledby compileSpadLispCmd, 65
calledby compiler, 9
pathnameDirectory(5)
calledby compileSpadLispCmd, 65
pathnameName(5)
calledby compileSpadLispCmd, 65
pathnameType(5)

calledby compileSpad2Cmd, 13
calledby compileSpadLispCmd, 65
calledby compiler, 9
pname
calledby comp3, 29
pop-stack-1
calledby spad, 18
postTransform
calledby s-process, 20
PredImplies
calledby compForm2, 47
preparse
calledby spad, 18
prettyprint
calledby s-process, 20
primitiveType, 39
calledby compAtom, 37
uses $DoubleFloat, 39
uses $EmptyMode, 39
uses $NegativeInteger, 39
uses $NonNegativeInteger, 39
uses $PositiveInteger, 39
uses $String, 39
defun, 39
processInteractive
calledby s-process, 20
put
calledby compColon, 33
calledby compTypeOf, 31

qcar
calledby compLambda, 59
calledby compWithMappingMode1, 52
qcdr
calledby compLambda, 59
calledby compWithMappingMode1, 52

recompile-lib-file-if-necessary, 66
calledby compileSpadLispCmd, 65
calls compile-lib-file, 66
uses *lisp-bin-filetype*, 66
defun, 66
resolve
calledby convert, 38

s-process, 20
calledby spad, 18
calls compTopLevel, 20
calls curstrm, 20
calls def-process, 20
calls def-rename, 20
calls displayPreCompilationErrors, 20
calls displaySemanticErrors, 20
calls get-internal-run-time, 20
calls new2OldLisp, 20
calls parseTransform, 20
calls postTransform, 20
calls prettyprint, 20
calls processInteractive, 20
calls terpri, 20
uses $DomainFrame, 20
uses $EmptyEnvironment, 20
uses $EmptyMode, 20
uses $Index, 20
uses $LocalFrame, 21
uses $PolyMode, 20
uses $VariableCount, 20
uses $compUniquelyIfTrue, 20
uses $currentFunction, 20
uses $exitModeStack, 20
uses $exitMode, 20
uses $e, 20
uses $form, 20
uses $genFVar, 20
uses $genSDVar, 20
uses $insideCapsuleFunctionIfTrue, 20
uses $insideCategoryIfTrue, 20
uses $insideCoerceInteractiveHardIfTrue, 20
uses $insideExpressionIfTrue, 20
uses $insideFunctorIfTrue, 20
uses $insideWhereIfTrue, 20
uses $leaveLevelStack, 20
uses $leaveMode, 20
uses $macroassoc, 20
uses $newspad, 20
uses $postStack, 20
uses $previousTime, 21
uses $returnMode, 20

uses $semanticErrorStack, 20
uses $top-level, 20
uses $topOp, 20
uses $warningStack, 20
uses curoutstream, 21
defun, 20
say
calledby compOrCroak1, 25
sayBrightly
calledby compilerDoit, 16
sayKeyedMsg
calledby compileSpad2Cmd, 13
calledby compileSpadLispCmd, 65
ScanOrPairVec
calledby hasFormalMapVariable, 58
selectOptionLC(5)
calledby compileSpad2Cmd, 13
calledby compileSpadLispCmd, 65
calledby compiler, 9
shut
calledby spad, 18
spad, 18
calledby /rf-1, 17
calls PARSE-NewExpr, 18
calls addBinding, 18
calls init-boot/spad-reader, 18
calls initialize-preparse, 18
calls ioclear, 18
calls makeInitialModemapFrame, 18
calls pop-stack-1, 18
calls preparse, 18
calls s-process, 18
calls shut, 18
uses *comp370-apply*, 18
uses *eof*, 18
uses /editfile, 18
uses $InitialDomainsInScope, 18
uses $InteractiveFrame, 18
uses $InteractiveMode, 18
uses $noSubsumption, 18
uses echo-meta, 18
uses file-closed, 18
uses line, 18
uses xcape, 18
catches, 18
defun, 18
spad-fixed-arg, 66
defun, 66
spad2AsTranslatorAutoloadOnceTrigger
calledby compileSpad2Cmd, 13
SpadInterpretStream(5)
calledby ncINTERPFILE, 64
spadPrompt
calledby compileSpad2Cmd, 13
calledby compileSpadLispCmd, 65
stackAndThrow
calledby compLambda, 59
calledby compWithMappingMode1, 52
stackMessage
calledby compSymbol, 40
stackMessageIfNone
calledby compForm, 43
stackSemanticError
calledby compColonInside, 36
calledby compOrCroak1, 25
stackWarning
calledby compColonInside, 36
strconc
calledby compileSpad2Cmd, 13
stringimage
calledby comp3, 29
stringPrefix?
calledby comp3, 29
sublis
calledby compForm2, 47
systemErrorHere
calledby compColon, 33

take
calledby compColon, 32
calledby compForm2, 47
calledby compWithMappingMode1, 52
terminateSystemCommand(5)
calledby compileSpad2Cmd, 13
calledby compileSpadLispCmd, 65
terpri
calledby s-process, 20
throwKeyedMsg
calledby compileSpad2Cmd, 13
calledby compileSpadLispCmd, 65

calledby compiler, 9

unionq
calledby freelist, 63
unknownTypeError
calledby compColon, 32
updateSourceFiles(5)
calledby compileSpad2Cmd, 13
userError
calledby compOrCroak1, 25

withAsharpCmd, 68
defun, 68

xcape
usedby spad, 18