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

Contents

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

all invoke `)compiler` on the file `/u/jones/as/mycode.as` if the current Axiom working directory is `/u/jones/as.` (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 chosen 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 old Axiom 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.

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

⟨*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.

⟨*defun compileSpad2Cmd*⟩≡

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

⟨*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|)))))
```

1.1.6 defun compileAsharpCmd

⟨*defun compileAsharpCmd*⟩≡

```
(defun |compileAsharpCmd| (args)
  (|compileAsharpCmd1| args)
  (|terminateSystemCommand|)
  (|spadPrompt|))
```

1.1.7 defun compileAsharpCmd1

⟨*defun compileAsharpCmd1*⟩≡

```
(defun |compileAsharpCmd1| (args)
 (let (path pathtype optlist optname optargs bequiet docompilelisp
       moreargs onlyargs dolibrary p tempargs s asharpargs command rc lsp)
  (declare (special |$options| |$asharpCmdlineFlags| |$newConlist|
            /editfile))
  (setq path (|pathname| args))
  (setq pathtype (|pathnameType| path))
  (cond
  ((and (nequal pathtype "as") (nequal pathtype "ao"))
    (|throwKeyedMsg| 's2iz0083 nil))
  ((null (probe-file path))
    (|throwKeyedMsg| 's2il0003 (cons (|namestring| args) nil)))
  (t
   (setq /editfile path)
   (|updateSourceFiles| path)
   (setq optlist
    '(|new| |old| |translate| |onlyargs| |moreargs| |quiet|
      |nolispcompile| |noquiet| |library| |nolibrary|))
   (setq bequiet nil)
   (setq dolibrary t)
   (setq docompilelisp t)
   (setq moreargs nil)
   (setq onlyargs nil)
   (dolist (opt |$options|)
     (setq optname (car opt))
     (setq optargs (cdr opt))
     (case (|selectOptionLC| optname optlist nil)
      (|new| nil)
      (|old| (|error| '|Internal error: compileAsharpCmd got )old|))
      (|translate|
       (|error| '|Internal error: compileAsharpCmd got )translate|))
      (|quiet|         (setq bequiet t))
      (|noquiet|       (setq bequiet nil))
      (|nolispcompile| (setq docompilelisp nil))
      (|moreargs|      (setq moreargs optargs))
      (|onlyargs|      (setq onlyargs optargs))
      (|library|       (setq dolibrary t))
      (|nolibrary|     (setq dolibrary nil))
      (t
       (|throwKeyedMsg| 's2iz0036
        (cons (strconc ")" (|object2String| optname)) nil)))))
   (setq tempargs
    (if (string= pathtype "ao")
```

```
    (if (setq p (strpos "-Fao" |$asharpCmdlineFlags| 0 nil))
      (if (eql p 0)
       (substring |$asharpCmdlineFlags| 5 nil)
       (strconc (substring |$asharpCmdlineFlags| 0 p)
          " " (substring |$asharpCmdlineFlags| (plus p 5) nil)))
      |$asharpCmdlineFlags|)
    |$asharpCmdlineFlags|))
(setq asharpargs
 (cond
  (onlyargs
   (setq s '||)
   (do ((t1 onlyargs (cdr t1)) (|a| nil))
       ((or (atom t1) (progn (setq |a| (car t1)) nil)) nil)
      (setq s (strconc s " " (|object2String| |a|))))
   s)
  (moreargs
   (setq s tempargs)
   (do ((t2 moreargs (cdr t2)) (|a| nil))
       ((or (atom t2) (progn (setq |a| (car t2)) nil)) nil)
      (setq s (strconc s " " (|object2String| |a|))))
   s)
  (t tempargs)))
(unless bequiet
 (|sayKeyedMsg| 's2iz0038a  (list (|namestring| args) asharpargs )))
(setq command
 (strconc
  (strconc (getenv "ALDORROOT") "/bin/")
  '|aldor | asharpargs " " (|namestring| args)))
(setq rc (obey command))
(cond
 ((and (eql rc 0) docompilelisp)
   (setq lsp (|fnameMake| "." (|pathnameName| args) "lsp"))
   (cond
    ((|fnameReadable?| lsp)
     (unless  bequiet
       (|sayKeyedMsg| 's2iz0089 (cons (|namestring| lsp) nil)))
     (|compileFileQuietly| lsp))
    (t (|sayKeyedMsg| 's2il0003 (cons (|namestring| lsp) nil))))))
(cond
 ((and (eql rc 0) dolibrary)
  (unless bequiet
     (|sayKeyedMsg| 's2iz0090 (cons (|pathnameName| path) nil)))
  (|withAsharpCmd| (cons (|pathnameName| path) nil)))
 ((null bequiet) (|sayKeyedMsg| 's2iz0084 nil))
 (t nil))
(|extendLocalLibdb| |$newConlist|)))))
```

1.1.8 defun compileAsharpArchiveCmd

⟨*defun compileAsharpArchiveCmd*⟩≡

```
(defun |compileAsharpArchiveCmd| (args)
 (let (path dir exists isdir curdir cmd rc asos)
  (declare (special $current-directory))
  (setq path (|pathname| args))
  (if (null (probe-file path))
   (|throwKeyedMsg| 's2il0003 (cons (|namestring| args) nil))
   (progn
    (setq dir (|fnameMake| "." (|pathnameName| path) "axldir"))
    (setq exists (probe-file dir))
    (setq isdir (|directoryp| (|namestring| dir)))
    (if (and exists (nequal isdir 1))
     (|throwKeyedMsg| 's2il0027 (list (|namestring| dir) (|namestring| args)))
     (progn
      (when (nequal isdir 1)
        (setq cmd (strconc "mkdir " (|namestring| dir)))
        (setq rc (obey cmd))
        (when (nequal rc 0)
          (|throwKeyedMsg| 's2il0027
           (list (|namestring| dir) (|namestring| args)))))
      (setq curdir $current-directory)
      (|cd| (cons (|object2Identifier| (|namestring| dir)) nil))
      (setq cmd (strconc "ar x " (|namestring| path)))
      (setq rc (obey cmd))
      (cond
       ((nequal rc 0)
        (|cd| (cons (|object2Identifier| (|namestring| curdir)) nil))
        (|throwKeyedMsg| 's2il0028
         (cons (|namestring| dir) (cons (|namestring| args) nil))))
       (t
        (setq asos (directory "*.ao"))
        (if (null asos)
         (progn
          (|cd| (cons (|object2Identifier| (|namestring| curdir)) nil))
          (|throwKeyedMsg| 's2il0029
           (cons (|namestring| dir) (cons (|namestring| args) nil))))
         (progn
          (dolist (aso asos)
             (|compileAsharpCmd1| (list (|namestring| aso))))
          (|cd| (list (|object2Identifier| (|namestring| curdir))))
          (|terminateSystemCommand|)
          (|spadPrompt|)))))))))))
```

1.1.9 defun compileAsharpLispCmd

⟨*defun compileAsharpLispCmd*⟩≡

```
(defun |compileAsharpLispCmd| (args)
 (let (path optlist optname optargs bequiet dolibrary lsp)
  (declare (special |$options|))
  (setq path (|pathname| args))
  (if (null (probe-file path))
   (|throwKeyedMsg| 's2il0003 (cons (|namestring| args) nil))
   (progn
    (setq optlist '(|quiet| |noquiet| |library| |nolibrary|))
    (setq bequiet nil)
    (setq dolibrary t)
    (dolist (opt |$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 (cons (|namestring| lsp) nil)))
      (|compileFileQuietly| lsp))
     (t (|sayKeyedMsg| 's2il0003 (cons (|namestring| lsp) nil))))
    (cond
     (dolibrary
      (unless  bequiet
        (|sayKeyedMsg| 's2iz0090 (cons (|pathnameName| path) nil)))
      (|withAsharpCmd| (cons (|pathnameName| path) nil)))
     ((null bequiet) (|sayKeyedMsg| 's2iz0084 nil))
     (t nil))
    (|terminateSystemCommand|)
    (|spadPrompt|)))))
```

1.1.10 defun withAsharpCmd

⟨*defun withAsharpCmd*⟩≡
```
(defun |withAsharpCmd| (args)
 (let (|$options|)
  (declare (special |$options|))
  (localdatabase args |$options|)))
```

1.1.11 defun compileFileQuietly

if `$InteractiveMode` then use a null outputstream

⟨*defun compileFileQuietly*⟩≡
```
(defun |compileFileQuietly| (fn)
  (let (
        (*standard-output*
         (if |$InteractiveMode| (make-broadcast-stream)
           *standard-output*)))
  (declare (special *standard-output* |$InteractiveMode|))
  (compile-file fn)))
```

1.1.12 defvar $byConstructors

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

1.1.13 defvar $constructorsSeen

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

1.1.14 defun compilerDoit

⟨*defun compilerDoit*⟩≡

```
(defun |compilerDoit| (constructor fun)
 (let (|$byConstructors| |$constructorsSeen|)
 (declare (special |$byConstructors| |$constructorsSeen|))
  (setq |$byConstructors| nil)
  (setq |$constructorsSeen| nil)
  (cond
   ((equal fun '(|rf| |lib|))   (|/RQ,LIB|))
   ((equal fun '(|rf| |nolib|)) (/rf))
   ((equal fun '(|rq| |lib|))   (|/RQ,LIB|))
   ((equal fun '(|rq| |nolib|)) (/rq))
   ((equal fun '(|c| |lib|))
    (setq |$byConstructors| (mapcar #'|opOf| constructor))
    (|/RQ,LIB|)
    (dolist (con |$byConstructors|)
     (unless (|member| con |$constructorsSeen|)
      (|sayBrightly| `(">>> Warning " |%b| ,con |%d| " was not found")))))))))
```

Chapter 2

Dangling references

2.1 shell variables

ALDORROOT

2.2 catch tags

2.3 throw tags

2.4 defined special variables

2.5 undefined special variables

|$asharpCmdlineFlags|
|$compileOnlyCertainItems|
$current-directory
/editfile
|$f|
|$InteractiveMode|
|$m|

|$ncConverse|
|$newComp|
|$newcompMode|
|$newConlist|
|$options|
|$QuickCode|
|$QuickLet|
|$scanIfTrue|
|$sourceFileTypes|
standard-output
$syscommands
|$systemCommands|

2.6 undefined functions

|browserAutoloadOnceTrigger|
|compilerDoit|
|compilerDoitWithScreenedLisplib|
|convertSpadToAsFile|
directory
|directoryp|
|error|
|extendLocalLibdb|
|fnameMake|
|fnameReadable?|
getenv
|member|
|namestring|
nequal
obey
|object2Identifier|
|object2String|
|oldParserAutoloadOnceTrigger|
|pathname|
|pathnameDirectory|
|pathnameName|
|pathnameType|
recompile-lib-file-if-necessary
|sayBrightly|
|sayKeyedMsg|
|spadPrompt|
|spad2AsTranslatorAutoloadOnceTrigger|
strconc
strpos
|throwKeyedMsg|
|terminateSystemCommand|
|updateSourceFiles|

Chapter 3

The Compiler

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

⟨*initvars*⟩

⟨*defun compileAsharpArchiveCmd*⟩
⟨*defun compileAsharpCmd*⟩
⟨*defun compileAsharpCmd1*⟩
⟨*defun compileAsharpLispCmd*⟩
⟨*defun compileFileQuietly*⟩
⟨*defun compilerDoit*⟩
⟨*defun compileSpad2Cmd*⟩
⟨*defun compileSpadLispCmd*⟩

⟨*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 4

Index

Index