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 5: Axiom Interpreter

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On October 1, 2001 Axiom was withdrawn from the market and ended life as a commercial product.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))

Chapter 1

Credits

Axiom has a very long history and many people have contributed to the effort, some in large ways and some in small ways. Any and all effort deserves recognition. There is no other criteria than contribution of effort. We would like to acknowledge and thank the following people:

⟨*initvars*⟩≡

```
(defvar credits '(
"An alphabetical listing of contributors to AXIOM:"
"Cyril Alberga          Roy Adler              Christian Aistleitner"
"Richard Anderson       George Andrews         S.J. Atkins"
"Henry Baker            Stephen Balzac         Yurij Baransky"
"David R. Barton        Gerald Baumgartner     Gilbert Baumslag"
"Michael Becker         Jay Belanger           David Bindel"
"Fred Blair             Vladimir Bondarenko    Mark Botch"
"Alexandre Bouyer       Peter A. Broadbery     Martin Brock"
"Manuel Bronstein       Stephen Buchwald       Florian Bundschuh"
"Luanne Burns           William Burge"
"Quentin Carpent        Robert Caviness        Bruce Char"
"Ondrej Certik          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"
"David Cyganski"
"Timothy Daly Sr.       Timothy Daly Jr.       James H. Davenport"
"Didier Deshommes       Michael Dewar"
"Jean Della Dora        Gabriel Dos Reis       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 Samantha Goldrich Holger Gollan"
"Teresa Gomez-Diaz Laureano Gonzalez-Vega Stephen Gortler"
"Johannes Grabmeier Matt Grayson Klaus Ebbe Grue"
"James Griesmer Vladimir Grinberg Oswald Gschnitzer"
"Jocelyn Guidry"
"Steve Hague Satoshi Hamaguchi Mike Hansen"
"Richard Harke Vilya Harvey Martin Hassner"
"Arthur S. Hathaway Dan Hatton Waldek Hebisch"
"Karl Hegbloom Ralf Hemmecke Henderson"
"Antoine Hersen Gernot Hueber"
"Pietro Iglio"
"Alejandro Jakubi Richard Jenks"
"Kai Kaminski Grant Keady Tony Kennedy"
"Paul Kosinski Klaus Kusche Bernhard Kutzler"
"Tim Lahey Larry Lambe Franz Lehner"
"Frederic Lehobey Michel Levaud Howard Levy"
"Liu Xiaojun Rudiger Loos Michael Lucks"
"Richard Luczak"
"Camm Maguire Francois Maltey Alasdair McAndrew"
"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 Joel Moses"
"Mark Murray"
"William Naylor C. Andrew Neff John Nelder"
"Godfrey Nolan Arthur Norman Jinzhong Niu"
"Michael O'Connor Summat Oemrawsingh Kostas Oikonomou"
"Humberto Ortiz-Zuazaga"
"Julian A. Padget Bill Page Susan Pelzel"
"Michel Petitot Didier Pinchon Ayal Pinkus"
"Jose Alfredo Portes"
"Claude Quitte"
"Arthur C. Ralfs Norman Ramsey Anatoly Raportirenko"
"Michael Richardson Renaud Rioboo Jean Rivlin"
"Nicolas Robidoux Simon Robinson Raymond Rogers"
"Michael Rothstein Martin Rubey"
"Philip Santas Alfred Scheerhorn William Schelter"
"Gerhard Schneider Martin Schoenert Marshall Schor"
"Frithjof Schulze Fritz Schwarz Nick Simicich"
"William Sit Elena Smirnova Jonathan Steinbach"
"Fabio Stumbo Christine Sundaresan Robert Sutor"
"Moss E. Sweedler Eugene Surowitz"
"Max Tegmark James Thatcher Balbir Thomas"
"Mike Thomas Dylan Thurston Barry Trager"

```
"Themos T. Tsikas"
"Gregory Vanuxem"
"Bernhard Wall          Stephen Watt          Jaap Weel"
"Juergen Weiss          M. Weller             Mark Wegman"
"James Wen              Thorsten Werther      Michael Wester"
"John M. Wiley          Berhard Will          Clifton J. Williamson"
"Stephen Wilson         Shmuel Winograd       Robert Wisbauer"
"Sandra Wityak          Waldemar Wiwianka     Knut Wolf"
"Clifford Yapp          David Yun"
"Vadim Zhytnikov        Richard Zippel        Evelyn Zoernack"
"Bruno Zuercher         Dan Zwillinger"
))
```

Chapter 2

The Interpreter

The Axiom interpreter is a large common lisp program. It has several forms of interaction and run from terminal in a standalone fashion, run under the control of a session handler program, run as a web server, or run in a unix pipe.

Chapter 3

The Fundamental Data Structures

Chapter 4

Starting Axiom

Axiom starts by invoking a function value of the lisp symbol `*top-level-hook*`. The function invocation path to from this point until the prompt is approximates (skipping initializations):

```
lisp -> restart
     -> |spad|
     -> |runspad|
     -> |ncTopLevel|
     -> |ncIntLoop|
     -> |intloop|
     -> |SpadInterpretStream|
     -> |intloopReadConsole|
```

The —intloopReadConsole— function does tail-recursive calls to itself (don't break this) and never exits.

4.1 Variables Used

4.2 Data Structures

4.3 Functions

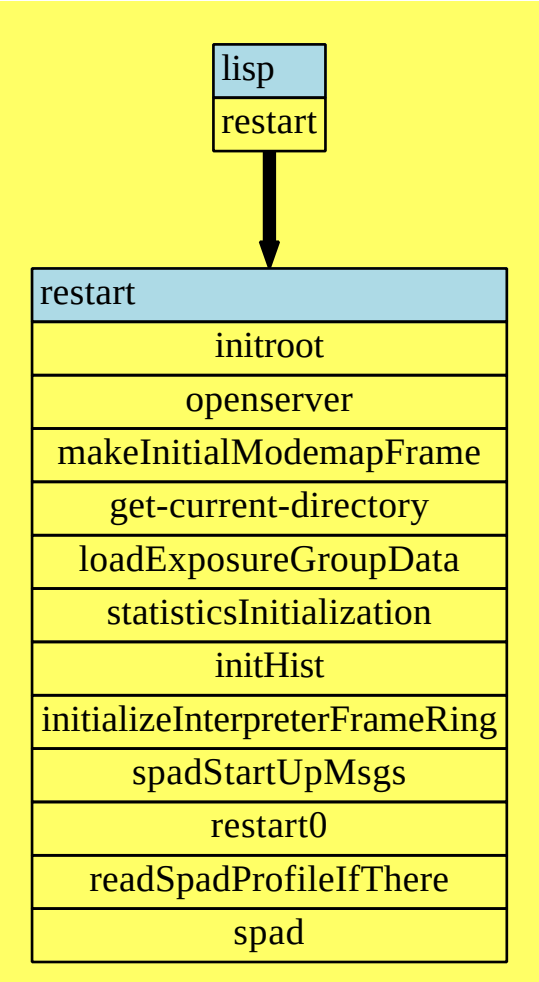
4.3.1 Set the restart hook

When a lisp image containing code is reloaded there is a hook to allow a function to be called. In our case it is the restart function which is the entry to the Axiom interpreter.

⟨*defun set-restart-hook*⟩≡

```
(defun set-restart-hook ()
  "Set the restart hook"
  #+KCL (setq system::*top-level-hook* 'restart)
  #+Lucid (setq boot::restart-hook 'restart)
  'restart
 )
```

4.3.2 restart (restart)



The restart function is the real root of the world. It sets up memory if we are working in a GCL/akcl version of the system.

The `compiler::*compile-verbose*` flag has been set to nil globally. We do not want to know about the microsteps of GCL's compile facility.

The `compiler::*suppress-compiler-warnings*` flag has been set to t. We do not care that certain generated variables are not used.

The `compiler::*suppress-compiler-notes*` flag has been set to t. We do not care that tail recursion occurs.

It sets the current package to be the "BOOT" package which is the standard package in which the interpreter runs.

The "initroot" (5.3.3 p 23) function sets global variables that depend on the

AXIOM shell variable. These are needed to find basic files like s2-us.msgs, which contains the error message text.

The "openserver" (46.0.17 p 493) function tried to set up the socket connection used for things like hyperdoc. The `$openServerIfTrue` variable starts true, which implies trying to start a server.

The `$IOindex` variable is the number associated with the input prompt. Every successful expression evaluated increments this number until a `)clear all` resets it. Here we set it to the initial value.

Axiom has multiple frames that contain independent information about a computation. There can be several frames at any one time and you can shift back and forth between the frames. By default, the system starts in "frame0" (try the `)frame names` command). See the Frame Mechanism chapter (**??** page **??**).

The `$InteractiveFrame` variable contains the state information related to the current frame, which includes things like the last value, the value of all of the variables, etc.

The "printLoadMsgs" (53.2.42 p 533) variable controls whether load messages will be output as library routines are loaded. We enable this by default. It can be changed by using `)set message autoload`.

The "current-directory" (**??** p **??**) variable is set to the current directory. This is used by the `)cd` function and some of the compile routines.

The "loadExposureGroupData" (48.0.19 p 497) function initializes several variables with the exposure groups. These lists limit the user visible names in order to keep the user and interpreter from being confused by names which are common but whose signatures are for internal algebra uses.

The "statisticsInitialization" (49.0.20 p 499) function initializes variables used to collect statistics. Currently, only the garbage collector information is initialized.

⟨*defun restart*⟩≡
```
(defun restart ()
  (declare (special $openServerIfTrue $SpadServerName |$SpadServer|
    |$IOindex| |$InteractiveFrame| |$printLoadMsgs| $current-directory
    |$displayStartMsgs| |$currentLine|))
#+:akcl
  (init-memory-config :cons 500 :fixnum 200 :symbol 500 :package 8
    :array 400 :string 500 :cfun 100 :cpages 3000 :rpages 1000 :hole 2000)
#+:akcl (setq compiler::*compile-verbose* nil)
#+:akcl (setq compiler::*suppress-compiler-warnings* t)
#+:akcl (setq compiler::*suppress-compiler-notes* t)
#+:akcl (setq si::*system-directory*
         (concatenate 'string (getenviron "AXIOM") "/bin"))
  (in-package "BOOT")
  (initroot)
#+:akcl
```

```
(when (and $openServerIfTrue (zerop (openserver $SpadServerName)))
  (setq $openServerIfTrue nil)
  (setq |$SpadServer| t))
(setq |$IOindex| 1)
(setq |$InteractiveFrame| (|makeInitialModemapFrame|))
(setq |$printLoadMsgs| t)
(setq $current-directory (get-current-directory))
(setq *default-pathname-defaults* (pathname $current-directory))
(|loadExposureGroupData|)
(|statisticsInitialization|)
(|initHist|)
(|initializeInterpreterFrameRing|)
(when |$displayStartMsgs| (|spadStartUpMsgs|))
(setq |$currentLine| nil)
(restart0)
(|readSpadProfileIfThere|)
(|spad|))
```

4.3.3 Starts the interpreter but do not read in profiles

⟨*defun spad*⟩≡

```
(defun |spad| ()
 "Starts the interpreter but do not read in profiles"
 (let (|$PrintCompilerMessageIfTrue| |$inLispVM|)
  (declare (special |$PrintCompilerMessageIfTrue| |$inLispVM|))
  (setq |$PrintCompilerMessageIfTrue| nil)
  (setq |$inLispVM| nil)
  (|setOutputAlgebra| '|%initialize%|)
  (|runspad|)
  '|EndOfSpad|))
```

4.3.4 defvar $quitTag

⟨*initvars*⟩+≡

```
(defvar |$quitTag| system::*quit-tag*)
```

4.3.5 defun runspad

⟨*defun runspad*⟩≡
```
(defun |runspad| ()
 (prog (mode)
 (declare (special |$quitTag|))
  (return
   (seq
    (progn
     (setq mode '|restart|)
     (do ()
         ((null (eq mode '|restart|)) nil)
      (seq
       (exit
        (progn
         (|resetStackLimits|)
         (catch |$quitTag|
          (catch '|coerceFailure|
           (setq mode (catch '|top_level| (|ncTopLevel|)))))))))))))
```

4.3.6 defun Reset the stack limits

⟨*defun resetStackLimits*⟩≡
```
(defun |resetStackLimits| ()
 "Reset the stack limits"
 (system:reset-stack-limits))
```

Chapter 5

Handling Input

5.1 Streams

5.1.1 defvar $curinstream

The curinstream variable is set to the value of the `*standard-input*` common lisp variable in ncIntLoop. While not using the "dollar" convention this variable is still "global".

⟨*initvars*⟩+≡
```
(defvar curinstream (make-synonym-stream '*standard-input*))
```

5.1.2 defvar $curoutstream

The curoutstream variable is set to the value of the `*standard-output*` common lisp variable in ncIntLoop. While not using the "dollar" convention this variable is still "global".

⟨*initvars*⟩+≡
```
(defvar curoutstream (make-synonym-stream '*standard-output*))
```

5.1.3 defvar $errorinstream

⟨*initvars*⟩+≡
```
(defvar errorinstream (make-synonym-stream '*terminal-io*))
```

5.1.4 defvar $erroroutstream

⟨*initvars*⟩+≡
```
(defvar erroroutstream (make-synonym-stream '*terminal-io*))
```

5.1.5 defvar $*eof*

⟨*initvars*⟩+≡
```
(defvar *eof* nil)
```

5.1.6 defvar $InteractiveMode

⟨*initvars*⟩+≡
```
(defvar |$InteractiveMode| t)
```

5.1.7 Top-level read-parse-eval-print loop

Top-level read-parse-eval-print loop for the interpreter. Uses the Bill Burge's parser.

⟨*defun ncTopLevel*⟩≡
```
(defun |ncTopLevel| ()
 "Top-level read-parse-eval-print loop"
 (let (|$e| $spad $newspad $boot |$InteractiveMode| *eof* in-stream)
  (declare (special |$e| $spad $newspad $boot |$InteractiveMode| *eof*
            in-stream |$InteractiveFrame|))
   (setq in-stream curinstream)
   (setq *eof* nil)
   (setq |$InteractiveMode| t)
   (setq $boot nil)
   (setq $newspad t)
   (setq $spad t)
   (setq |$e| |$InteractiveFrame|)
   (|ncIntLoop|)))
```

5.1.8 defun ncIntLoop

⟨*defun ncIntLoop*⟩≡
```
(defun |ncIntLoop| ()
  (let ((curinstream *standard-output*)
        (curoutstream *standard-input*))
    (declare (special curinstream curoutstream))
    (|intloop|)))
```

5.1.9 defun intloop

Note that the SpadInterpretStream function uses a list of three strings as an argument. The values in the list seem to have no use and can eventually be removed.

⟨*defun intloop*⟩≡
```
(defun |intloop| ()
 (prog (mode)
  (declare (special |$intTopLevel| |$intRestart|))
   (return
    (progn
     (setq mode |$intRestart|)
     ((lambda ()
       (loop
        (cond
         ((not (equal mode |$intRestart|))
           (return nil))
         (t
           (progn
             (|resetStackLimits|)
             (setq mode
               (catch |$intTopLevel|
                 (|SpadInterpretStream| 1
                   (list 'tim  'daly '?) t)))))))))))))
```

5.1.10 defun SpadInterpretStream

The SpadInterpretStream function takes three arguments

str This is passed as an argument to intloopReadConsole

source This is the name of a source file but appears not to be used. It is set to the list `(tim daly ?)`.

`interactive?` If this is false then various messages are suppressed and input does not use piles. If this is true then the library loading routines might output messages and piles are expected on input (as from a file).

The system commands are handled by the function kept in the "hook" variable `$systemCommandFunction` which has the default function `InterpExecuteSpadSystemCommand`. Thus, when a system command is entered this function is called.

The `$promptMsg` variable is set to the constant S2CTP023. This constant points to a message in src/doc/msgs/s2-us.msgs. This message does nothing but print the argument value.

5.1.11 defvar $promptMsg

⟨*initvars*⟩+≡

```
(defvar |$promptMsg| 'S2CTP023)
```

⟨*defun SpadInterpretStream*⟩≡

```
(defun |SpadInterpretStream| (str source interactive?)
 (let (|$promptMsg| |$systemCommandFunction|
       |$ncMsgList| |$erMsgToss| |$lastPos| |$inclAssertions|
       |$okToExecuteMachineCode| |$newcompErrorCount| |$newcompMode|
       |$libQuiet| |$fn|)
  (declare (special |$promptMsg|
            |$systemCommandFunction| |$ncMsgList| |$erMsgToss| |$lastPos|
            |$inclAssertions| |$okToExecuteMachineCode| |$newcompErrorCount|
            |$newcompMode| |$libQuiet| |$fn| |$nopos|))
  (setq |$fn| source)
  (setq |$libQuiet| (null interactive?))
  (setq |$newcompMode| nil)
  (setq |$newcompErrorCount| 0)
  (setq |$okToExecuteMachineCode| t)
  (setq |$inclAssertions| (list 'aix '|CommonLisp|))
  (setq |$lastPos| |$nopos|)
  (setq |$erMsgToss| nil)
  (setq |$ncMsgList| nil)
  (setq |$systemCommandFunction| #'|InterpExecuteSpadSystemCommand|)
  (setq |$promptMsg| 's2ctp023)
  (if interactive?
   (progn
     (princ (mkprompt))
     (|intloopReadConsole| "" str))
   (|intloopInclude| source 0))))
```

5.2 The Read-Eval-Print Loop

5.2.1 defun intloopReadConsole

Note that this function relies on the fact that lisp can do tail-recursion. The function recursively invokes itself.

The serverReadLine function is a special readline function that handles communication with the session manager code, which is a separate process running in parallel.

We read a line from standard input.

- If it is a null line then we exit Axiom.
- If it is a zero length line we prompt and recurse
- If $dalymode and open-paren we execute lisp code, prompt and recurse The $dalymode will interpret any input that begins with an open-paren as a lisp expression rather than Axiom input. This is useful for debugging purposes when most of the input lines will be lisp. Setting $dalymode non-nil will certainly break user expectations and is to be used with caution.
- If it is ")fi" or ")fin" we drop into lisp. Use the (restart) function to return to the interpreter loop.
- If it starts with ")" we process the command, prompt, and recurse
- If it is a command then we remember the current line, process the command, prompt, and recurse.
- If the input has a trailing underscore (Axiom line-continuation) then we cut off the continuation character and pass the truncated string to ourselves, prompt, and recurse
- otherwise we process the input, prompt, and recurse.

Notice that all but two paths (a null input or a ")fi" or a ")fin") will end up as a recursive call to ourselves.

⟨*defun intloopReadConsole*⟩≡

```
(defun |intloopReadConsole| (b n)
 (declare (special $dalymode))
 (let (c d pfx input)
   (setq input (|serverReadLine| *standard-input*))
   (when (null (stringp input)) (|leaveScratchpad|))
   (when (eql (length input) 0)
     (princ (mkprompt))
     (|intloopReadConsole| "" n))
   (when (and $dalymode (|intloopPrefix?| "(" input))
```

```
    (|intnplisp| input)
    (princ (mkprompt))
    (|intloopReadConsole| "" n))
  (setq pfx (|intloopPrefix?| ")fi" input))
  (when (and pfx (or (string= pfx ")fi") (string= pfx ")fin")))
    (throw '|top_level| nil))
  (when (and (equal b "") (setq d (|intloopPrefix?| ")" input)))
    (|setCurrentLine| d)
    (setq c (|ncloopCommand| d n))
    (princ (mkprompt))
    (|intloopReadConsole| "" c))
  (setq input (concat b input))
  (when (|ncloopEscaped| input)
    (|intloopReadConsole| (subseq input 0 (- (length input) 1)) n))
  (setq c (|intloopProcessString| input n))
  (princ (mkprompt))
  (|intloopReadConsole| "" c)))
```

5.3 Helper Functions

5.3.1 Get the value of an evironment variable

⟨*defun getenviron*⟩≡

```
(defun getenviron (var)
 "Get the value of an evironment variable"
 #+allegro (sys::getenv (string var))
 #+clisp (ext:getenv (string var))
 #+(or cmu scl)
  (cdr
   (assoc (string var) ext:*environment-list* :test #'equalp :key #'string))
 #+(or kcl akcl gcl) (si::getenv (string var))
 #+lispworks (lw:environment-variable (string var))
 #+lucid (lcl:environment-variable (string var))
 #+mcl (ccl::getenv var)
 #+sbcl (sb-ext:posix-getenv var)
 )
```

5.3.2 defun init-memory-config

Austin-Kyoto Common Lisp (AKCL), now known as Gnu Common Lisp (GCL) requires some changes to the default memory setup to run Axiom efficently. This function performs those setup commands.

⟨*defun init-memory-config*⟩≡

```
(defun init-memory-config (&key
                           (cons 500)
                           (fixnum 200)
                           (symbol 500)
                           (package 8)
                           (array 400)
                           (string 500)
                           (cfun 100)
                           (cpages 3000)
                           (rpages 1000)
                           (hole 2000) )
  ;; initialize AKCL memory allocation parameters
  #+:AKCL
  (progn
    (system:allocate 'cons cons)
    (system:allocate 'fixnum fixnum)
    (system:allocate 'symbol symbol)
    (system:allocate 'package package)
    (system:allocate 'array array)
    (system:allocate 'string string)
    (system:allocate 'cfun cfun)
    (system:allocate-contiguous-pages cpages)
    (system:allocate-relocatable-pages rpages)
    (system:set-hole-size hole))
  #-:AKCL
  nil)
```

5.3.3 Set spadroot to be the AXIOM shell variable

Sets up the system to use the **AXIOM** shell variable if we can and default to the **$spadroot** variable (which was the value of the **AXIOM** shell variable at build time) if we can't.

Called from "restart" (4.3.2 p 11).

⟨*defun initroot*⟩≡
```
(defun initroot (&optional (newroot (getenviron "AXIOM")))
 "Set spadroot to be the AXIOM shell variable"
  (declare (special $spadroot))
  (reroot (or newroot $spadroot (error "setenv AXIOM or (setq $spadroot)"))))
```

5.3.4 Does the string start with this prefix?

If the prefix string is the same as the whole string initial characters –R(ignoring spaces in the whole string) then we return the whole string minus any leading spaces.

⟨*defun intloopPrefix?*⟩≡
```
(defun |intloopPrefix?| (prefix whole)
 "Does the string start with this prefix?"
 (let ((newprefix (string-left-trim '(#\space) prefix))
       (newwhole  (string-left-trim '(#\space) whole)))
  (when (<= (length newprefix) (length newwhole))
   (when (string= newprefix newwhole :end2 (length prefix))
    newwhole))))
```

5.3.5 Get the current directory

⟨*defun get-current-directory*⟩≡
```
#+:cmu
(defun get-current-directory ()
 "Get the current directory"
  (namestring (extensions::default-directory)))

#+(or :akcl :gcl)
(defun get-current-directory ()
 "Get the current directory"
  (namestring (truename "")))
```

5.3.6 Prepend the absolute path to a filename

Prefix a filename with the **AXIOM** shell variable.

⟨*defun make-absolute-filename*⟩≡
```
(defun make-absolute-filename (name)
 "Prepend the absolute path to a filename"
 (declare (special $spadroot))
 (concatenate 'string $spadroot name))
```

5.3.7 Make the initial modemap frame

⟨*defun makeInitialModemapFrame*⟩≡
```
(defun |makeInitialModemapFrame| ()
  "Make the initial modemap frame"
  (declare (special |$InitialModemapFrame|))
  (copy |$InitialModemapFrame|))
```

5.3.8 defun ncloopEscaped

The ncloopEscaped function will return true if the last non-blank character of a line is an underscore, the Axiom line-continuation character. Otherwise, it returns nil.

⟨*defun ncloopEscaped*⟩≡
```
(defun |ncloopEscaped| (x)
 (let ((l (length x)))
  (dotimes (i l)
   (when (char= (char x (- l i 1)) #\_) (return t))
   (unless (char= (char x (- l i 1)) #\space) (return nil)))))
```

5.3.9 Call the garbage collector

Call the garbage collector on various platforms.

⟨*defun reclaim*⟩≡

```
#+abcl
(defun reclaim () "Call the garbage collector" (ext::gc))
#+:allegro
(defun reclaim () "Call the garbage collector" (excl::gc t))
#+:CCL
(defun reclaim () "Call the garbage collector" (gc))
#+clisp
(defun reclaim ()
 "Call the garbage collector"
 (#+lisp=cl ext::gc #-lisp=cl lisp::gc))
#+(or :cmulisp :cmu)
(defun reclaim () "Call the garbage collector" (ext:gc))
#+cormanlisp
(defun reclaim () "Call the garbage collector" (cl::gc))
#+(OR IBCL KCL GCL)
(defun reclaim () "Call the garbage collector" (si::gbc t))
#+lispworks
(defun reclaim () "Call the garbage collector" (hcl::normal-gc))
#+Lucid
(defun reclaim () "Call the garbage collector" (lcl::gc))
#+sbcl
(defun reclaim () "Call the garbage collector" (sb-ext::gc))
```

5.3.10 defun reroot

The reroot function is used to reset the important variables used by the system. In particular, these variables are sensitive to the **AXIOM** shell variable. That variable is renamed internally to be **$spadroot**. The **reroot** function will change the system to use a new root directory and will have the same effect as changing the **AXIOM** shell variable and rerunning the system from scratch. Note that we have changed from the NAG distribution back to the original form. If you need the NAG version you can push **:tpd** on the ***features*** variable before compiling this file. A correct call looks like:

```
(in-package "BOOT")
(reroot "/spad/mnt/${SYS}")
```

where the `${SYS}` variable is the same one set at build time.

⟨*defun reroot*⟩≡

```
(defun reroot (dir)
  (declare (special $spadroot $directory-list $relative-directory-list
     $library-directory-list $relative-library-directory-list
     |$defaultMsgDatabaseName| |$msgDatabaseName| $current-directory))
  (setq $spadroot dir)
  (setq $directory-list
   (mapcar #'make-absolute-filename $relative-directory-list))
  (setq $library-directory-list
   (mapcar #'make-absolute-filename $relative-library-directory-list))
  (setq |$defaultMsgDatabaseName|
        (pathname (make-absolute-filename "/doc/msgs/s2-us.msgs")))
  (setq |$msgDatabaseName| ())
  (setq $current-directory $spadroot))
```

5.3.11 defun setCurrentLine

Remember the current line. The cases are:

- If there is no $currentLine set it to the input
- Is the current line a string and the input a string? Make them into a list
- Is $currentLine not a cons cell? Make it one.
- Is the input a string? Cons it on the end of the list.
- Otherwise stick it on the end of the list

Note I suspect the last two cases do not occur in practice since they result in a dotted pair if the input is not a cons. However, this is what the current code does so I won't change it.

⟨*defun setCurrentLine*⟩≡

```
(defun |setCurrentLine| (s)
 (declare (special |$currentLine|))
 (cond
  ((null |$currentLine|) (setq |$currentLine| s))
  ((and (stringp |$currentLine|) (stringp s))
    (setq |$currentLine| (list |$currentLine| s)))
  ((not (consp |$currentLine|)) (setq |$currentLine| (cons |$currentLine| s)))
  ((stringp s) (rplacd (last |$currentLine|) (cons s nil)))
  (t (rplacd (last |$currentLine|) s)))
 |$currentLine|)
```

5.3.12 Show the Axiom prompt

⟨*defun mkprompt*⟩≡

```
(defun mkprompt ()
 "Show the Axiom prompt"
 (declare (special |$inputPromptType| |$IOindex| |$interpreterFrameName|))
 (case |$inputPromptType|
  (|none| "")
  (|plain| "-> ")
  (|step|  (strconc "(" (stringimage |$IOindex|) ") -> "))
  (|frame|
   (strconc (stringimage |$interpreterFrameName|) " ("
            (stringimage |$IOindex|) ") -> "))
  (t (strconc (stringimage |$interpreterFrameName|) " ["
              (substring (currenttime) 8 nil) "] ["
              (stringimage |$IOindex|) "] -> "))))
```

5.3.13 defvar $frameAlist

⟨*initvars*⟩+≡
```
(defvar |$frameAlist| nil)
```

5.3.14 defvar $frameNumber

⟨*initvars*⟩+≡
```
(defvar |$frameNumber| 0)
```

5.3.15 defvar $currentFrameNum

⟨*initvars*⟩+≡
```
(defvar |$currentFrameNum| 0)
```

5.3.16 defvar $EndServerSession

⟨*initvars*⟩+≡
```
(defvar |$EndServerSession| nil)
```

5.3.17 defvar $NeedToSignalSessionManager

⟨*initvars*⟩+≡
```
(defvar |$NeedToSignalSessionManager| nil)
```

5.3.18 defvar $sockBufferLength

⟨*initvars*⟩+≡
```
(defvar |$sockBufferLength| 9217)
```

5.3.19 READ-LINE in an Axiom server system

⟨*defun serverReadLine*⟩≡

```
(defun |serverReadLine| (stream)
 "used in place of READ-LINE in a Axiom server system."
 (let (in-stream *eof* l framename currentframe form stringbuf line action)
 (declare (special in-stream *eof* |$SpadServer| |$EndServerSession|
     |$NeedToSignalSessionManager| |$SessionManager| |$EndOfOutput|
     |$CallInterp| |$CreateFrame| |$frameAlist| |$frameNumber|
     |$currentFrameNum| |$CreateFrameAnswer| |$SwitchFrames| |$EndSession|
     |$EndServerSession| |$LispCommand| |$sockBufferLength| |$MenuServer|
     |$QuietSpadCommand| |$SpadCommand| |$NonSmanSession| |$KillLispSystem|))
  (force-output)
  (if (or (null |$SpadServer|) (null (is-console stream)))
   (|read-line| stream)
   (progn
    (setq in-stream stream)
    (setq *eof* nil)
    (setq line
     (do ()
         ((null (and (null |$EndServerSession|) (null *eof*))) nil)
         (when |$NeedToSignalSessionManager|
          (|sockSendInt| |$SessionManager| |$EndOfOutput|))
      (setq |$NeedToSignalSessionManager| nil)
      (setq action (|serverSwitch|))
      (cond
       ((= action |$CallInterp|)
        (setq l (|read-line| stream))
        (setq |$NeedToSignalSessionManager| t)
        (return l))
       ((= action |$CreateFrame|)
        (setq framename (gentemp "frame"))
        (|addNewInterpreterFrame| framename)
        (setq |$frameAlist|
         (cons (cons |$frameNumber| framename) |$frameAlist|))
        (setq |$currentFrameNum| |$frameNumber|)
        (|sockSendInt| |$SessionManager| |$CreateFrameAnswer|)
        (|sockSendInt| |$SessionManager| |$frameNumber|)
        (setq |$frameNumber| (plus |$frameNumber| 1))
        (|sockSendString| |$SessionManager| (mkprompt)))
       ((= action |$SwitchFrames|)
        (setq |$currentFrameNum| (|sockGetInt| |$SessionManager|))
        (setq currentframe (lassoc |$currentFrameNum| |$frameAlist|))
        (|changeToNamedInterpreterFrame| currentframe))
       ((= action |$EndSession|)
        (setq |$EndServerSession| t))
```

```
        ((= action |$LispCommand|)
         (setq |$NeedToSignalSessionManager| t)
         (setq stringbuf (make-string |$sockBufferLength|))
         (|sockGetString| |$MenuServer| stringbuf |$sockBufferLength|)
         (setq form
          (|unescapeStringsInForm| (read-from-string stringbuf)))
         (|protectedEVAL| form))
        ((= action |$QuietSpadCommand|)
         (setq |$NeedToSignalSessionManager| t)
         (|executeQuietCommand|))
        ((= action |$SpadCommand|)
         (setq |$NeedToSignalSessionManager| t)
         (setq stringbuf (make-string 512))
         (|sockGetString| |$MenuServer| stringbuf 512)
         (catch '|coerceFailure|
          (catch '|top_level|
           (catch 'spad_reader
            (|parseAndInterpret| stringbuf))))
         (princ (mkprompt))
         (finish-output))
        ((= action |$NonSmanSession|) (setq |$SpadServer| nil))
        ((= action |$KillLispSystem|) (bye))
        (t nil))))
    (cond
     (line line)
     (t '|||||||)))
```

5.3.20 Include a file into the stream

⟨*defun intloopInclude*⟩≡

```
(defun |intloopInclude| (name n)
  "Include a file into the stream"
  (with-open-file (st name) (|intloopInclude0| st name n)))
```

5.3.21 defun intloopInclude0

⟨*defun intloopInclude0*⟩≡
```
(defun |intloopInclude0| (|st| |name| |n|)
 (let (|$lines|)
 (declare (special |$lines|))
  (setq |$lines| (|incStream| |st| |name|))
  (|intloopProcess| |n| NIL
   (|next| (function |intloopEchoParse|)
    (|next| (function |insertpile|)
     (|next| (function |lineoftoks|)
      |$lines|))))))
```

5.3.22 defun ncloopInclude0

⟨*defun ncloopInclude0*⟩≡
```
(defun |ncloopInclude0| (st name n)
 (let (|$lines|)
 (declare (special |$lines|))
   (setq |$lines| (|incStream| st name))
   (|ncloopProcess| n nil
    (|next| (function |ncloopEchoParse|)
     (|next| (function |insertpile|)
      (|next| (function |lineoftoks|)
       |$lines|))))))
```

5.3.23 defun incStream

⟨*defun incStream*⟩≡
```
(defun |incStream| (st fn)
 (declare (special |Top|))
 (|incRenumber| (|incLude| 0 (|incRgen| st) 0 (list fn) (list |Top|))))
```

5.3.24 defun incLude

⟨*defun incLude*⟩≡
```
(defun |incLude| (eb ss ln ufos states)
 (|Delay| (function |incLude1|) (list eb ss ln ufos states)))
```

5.3.25 defmacro Rest

⟨*defmacro Rest*⟩≡
```
(defmacro |Rest| ()
 "used in incLude1 for parsing; s is not used."
 '(|incLude| eb (cdr ss) lno ufos states))
```

5.3.26 defvar $Top

⟨*initvars*⟩+≡
```
(defvar |Top| 1 "used in incLude1 for parsing")
```

5.3.27 defvar $IfSkipToEnd

⟨*initvars*⟩+≡
```
(defvar |IfSkipToEnd| 10 "used in incLude1 for parsing")
```

5.3.28 defvar $IfKeepPart

⟨*initvars*⟩+≡
```
(defvar |IfKeepPart| 11 "used in incLude1 for parsing")
```

5.3.29 defvar $IfSkipPart

⟨*initvars*⟩+≡
```
(defvar |IfSkipPart| 12 "used in incLude1 for parsing")
```

5.3.30 defvar $ElseifSkipToEnd

⟨*initvars*⟩+≡
```
(defvar |ElseifSkipToEnd| 20 "used in incLude1 for parsing")
```

5.3.31 defvar $ElseifKeepPart

⟨*initvars*⟩+≡
```
(defvar |ElseifKeepPart| 21 "used in incLude1 for parsing")
```

5.3.32 defvar $ElseifSkipPart

⟨*initvars*⟩+≡
```
(defvar |ElseifSkipPart| 22 "used in incLude1 for parsing")
```

5.3.33 defvar $ElseSkipToEnd

⟨*initvars*⟩+≡
```
(defvar |ElseSkipToEnd| 30 "used in incLude1 for parsing")
```

5.3.34 defvar $ElseKeepPart

⟨*initvars*⟩+≡
```
(defvar |ElseKeepPart| 31 "used in incLude1 for parsing")
```

5.3.35 defvar $Top?

⟨*defun Top?*⟩≡
```
(defun |Top?| (|st|)
 "used in incLude1 for parsing"
 (eql (quotient |st| 10) 0))
```

5.3.36 defvar $If?

⟨*defun If?*⟩≡
```
(defun |If?| (|st|)
 "used in incLude1 for parsing"
 (eql (quotient |st| 10) 1))
```

5.3.37 defvar $Elseif?

⟨*defun Elseif?*⟩≡
```
(defun |Elseif?| (|st|)
 "used in incLude1 for parsing"
 (eql (quotient |st| 10) 2))
```

5.3.38 defvar $Else?

⟨*defun Else?*⟩≡
```
(defun |Else?| (|st|)
 "used in incLude1 for parsing"
 (eql (quotient |st| 10) 3))
```

5.3.39 defvar $SkipEnd?

⟨*defun SkipEnd?*⟩≡
```
(defun |SkipEnd?| (|st|)
 "used in incLude1 for parsing"
 (eql (remainder |st| 10) 0))
```

5.3.40 defvar $KeepPart?

⟨*defun KeepPart?*⟩≡
```
(defun |KeepPart?| (|st|)
 "used in incLude1 for parsing"
 (eql (remainder |st| 10) 1))
```

5.3.41 defvar $SkipPart?

⟨*defun SkipPart?*⟩≡
```
(defun |SkipPart?| (|st|)
 "used in incLude1 for parsing"
 (eql (remainder |st| 10) 2))
```

5.3.42 defvar $Skipping?

⟨*defun Skipping?*⟩≡

```
(defun |Skipping?| (|st|)
 "used in incLude1 for parsing"
 (null (|KeepPart?| |st|)))
```

5.3.43 defun incLude1

⟨*defun incLude1*⟩≡

```
(defun |incLude1| (&rest z)
(let (pred s1 n tail head includee fn1 info str state lno states ufos ln ss eb)
  (setq eb (car z))
  (setq ss (cadr . (z)))
  (setq ln (caddr . (z)))
  (setq ufos (cadddr . (z)))
  (setq states (car (cddddr . (z))))
  (setq lno (+ ln 1))
  (setq state (elt states 0))
  (cond
   ((|StreamNull| ss)
     (cond
      ((null (|Top?| state))
       (cons (|xlPrematureEOF| eb ")--premature end" lno ufos)
             |StreamNil|))
      (t |StreamNil|)))
   (t
    (progn
     (setq str (expand-tabs (car ss)))
     (setq info (|incClassify| str))
     (cond
      ((null (elt info 0))
       (cond
        ((|Skipping?| state)
         (cons (|xlSkip| eb str lno (elt ufos 0)) (|Rest|)))
        (t
         (cons (|xlOK| eb str lno (elt ufos 0)) (|Rest|)))))
      ((equal (elt info 2) "other")
       (cond
        ((|Skipping?| state)
         (cons (|xlSkip| eb str lno (elt ufos 0)) (|Rest|)))
        (t
         (cons
          (|xlOK1| eb str (concat ")command" str) lno (elt ufos 0))
          (|Rest|)))))
      ((equal (elt info 2) "say")
       (cond
        ((|Skipping?| state)
         (cons (|xlSkip| eb str lno (elt ufos 0)) (|Rest|)))
        (t
         (progn
          (setq str (|incCommandTail| str info))
          (cons (|xlSay| eb str lno ufos str)
```

```
      (cons (|xlOK| eb str lno (ELT ufos 0)) (|Rest|)))))))
((equal (elt info 2) "include")
 (cond
  ((|Skipping?| state)
   (cons (|xlSkip| eb str lno (elt ufos 0)) (|Rest|)))
  (t
   (progn
    (setq fn1 (|inclFname| str info))
    (cond
     ((null fn1)
      (cons (|xlNoSuchFile| eb str lno ufos fn1) (|Rest|)))
     ((null (probe-file fn1))
      (cons (|xlCannotRead| eb str lno ufos fn1) (|Rest|)))
     ((|incActive?| fn1 ufos)
      (cons (|xlFileCycle| eb str lno ufos fn1) (|Rest|)))
     (t
      (progn
       (setq includee
        (|incLude| (+ eb (elt info 1))
                   (|incFileInput| fn1)
                   0
                   (cons fn1 ufos)
                   (cons |Top| states)))
       (cons (|xlOK| eb str lno (elt ufos 0))
             (|incAppend| includee (|Rest|)))))))))))
((equal (elt info 2) "console")
 (cond
  ((|Skipping?| state)
   (cons (|xlSkip| eb str lno (elt ufos 0)) (|Rest|)))
  (t
   (progn
    (setq head
     (|incLude| (+ eb (elt info 1))
                (|incConsoleInput|)
                0
                (cons "console" ufos)
                (cons |Top| states)))
    (setq tail (|Rest|))
    (setq n (|incNConsoles| ufos))
    (cond
     ((< 0 n)
      (setq head
       (cons (|xlConActive| eb str lno ufos n) head))
      (setq tail
       (cons (|xlConStill| eb str lno ufos n) tail))))
    (setq head (cons (|xlConsole| eb str lno ufos) head))
```

```
    (cons (|xlOK| eb str lno (elt ufos 0))
          (|incAppend| head tail))))))
((equal (elt info 2) "fin")
 (cond
  ((|Skipping?| state)
   (cons (|xlSkippingFin| eb str lno ufos) (|Rest|)))
  ((null (|Top?| state))
   (cons (|xlPrematureFin| eb str lno ufos) |StreamNil|))
  (t
   (cons (|xlOK| eb str lno (elt ufos 0)) |StreamNil|))))
((equal (elt info 2) "assert")
 (cond
  ((|Skipping?| state)
   (cons (|xlSkippingFin| eb str lno ufos) (|Rest|)))
  (t
   (progn
    (|assertCond| str info)
    (cons (|xlOK| eb str lno (elt ufos 0))
          (|incAppend| includee (|Rest|)))))))
((equal (elt info 2) "if")
 (progn
  (setq s1
   (cond
    ((|Skipping?| state) |IfSkipToEnd|)
    (t
     (cond
      ((|ifCond| str info) |IfKeepPart|)
      (t |IfSkipPart|)))))
  (cons (|xlOK| eb str lno (elt ufos 0))
        (|incLude| eb (cdr ss) lno ufos (cons s1 states)))))
((equal (elt info 2) "elseif")
 (cond
  ((and (null (|If?| state)) (null (|Elseif?| state)))
   (cons (|xlIfSyntax| eb str lno ufos info states)
         |StreamNil|))
  (t
   (cond
    ((or (|SkipEnd?| state)
         (|KeepPart?| state)
         (|SkipPart?| state))
     (setq s1
      (cond
       ((|SkipPart?| state)
        (setq pred (|ifCond| str info))
        (cond
         (pred |ElseifKeepPart|)
```

```
            (t |ElseifSkipPart|)))
          (t |ElseifSkipToEnd|)))
       (cons (|xlOK| eb str lno (elt ufos 0))
             (|incLude| eb (cdr ss) lno ufos (cons s1 (cdr states)))))
      (t
       (cons (|xlIfBug| eb str lno ufos) |StreamNil|))))))
((equal (elt info 2) "else")
 (cond
  ((and (null (|If?| state)) (null (|Elseif?| state)))
   (cons (|xlIfSyntax| eb str lno ufos info states)
         |StreamNil|))
  (t
   (cond
    ((or (|SkipEnd?| state)
         (|KeepPart?| state)
         (|SkipPart?| state))
     (setq s1
      (cond ((|SkipPart?| state) |ElseKeepPart|) (t |ElseSkipToEnd|)))
     (cons (|xlOK| eb str lno (elt ufos 0))
           (|incLude| eb (cdr ss) lno ufos (cons s1 (cdr states)))))
    (t
     (cons (|xlIfBug| eb str lno ufos) |StreamNil|))))))
((equal (elt info 2) "endif")
 (cond
  ((|Top?| state)
   (cons (|xlIfSyntax| eb str lno ufos info states)
         |StreamNil|))
  (t
   (cons (|xlOK| eb str lno (elt ufos 0))
         (|incLude| eb (cdr ss) lno ufos (cdr states))))))
(t (cons (|xlCmdBug| eb str lno ufos) |StreamNil|))))))))
```

5.3.44 defun incRgen

Note that incRgen1 recursively calls this function.

⟨*defun incRgen*⟩≡

```
(defun |incRgen| (s)
 (|Delay| (function |incRgen1|) (list s)))
```

5.3.45 defun Delay

⟨*defun Delay*⟩≡
```
(defun |Delay| (f x)
 (cons '|nonnullstream| (cons f x)))
```

⟨*initvars*⟩+≡
```
(defvar |StreamNil| (list '|nullstream|))
```

⟨*postvars*⟩≡
```
(eval-when (eval load)
  (setq |StreamNil| (list '|nullstream|)))
```

5.3.46 defun incRgen1

This function reads a line from the stream and then conses it up with a recursive call to incRgen. Note that incRgen recursively wraps this function in a delay list.

⟨*defun incRgen1*⟩≡
```
(defun |incRgen1| (&rest z)
 (let (a s)
 (declare (special |StreamNil|))
  (setq s (car z))
  (setq a (|shoeread-line| s))
  (if (null a)
   (progn
    (close s)
    |StreamNil|)
   (cons a (|incRgen| s)))))
```

Chapter 6

The Interpreter Syntax

6.1 syntax assignment

⟨*assignment.help*⟩≡

```
Immediate, Delayed, and Multiple Assignment

=======================================================================
Immediate Assignment
=======================================================================

A variable in Axiom refers to a value. A variable has a name beginning
with an uppercase or lowercase alphabetic character, "%", or "!".
Successive characters (if any) can be any of the above, digits, or "?".
Case is distinguished. The following are all examples of valid, distinct
variable names:

  a       tooBig?     a1B2c3%!?
  A       %j          numberOfPoints
  beta6   %J          numberofpoints

The ":=" operator is the immediate assignment operator. Use it to
associate a value with a variable. The syntax for immediate assignment
for a single variable is:

   variable := expression

The value returned by an immediate assignment is the value of expression.

  a := 1
```

```
    1
           Type: PositiveInteger
```

The right-hand side of the expression is evaluated, yielding 1. The value is then assigned to a.

```
  b := a
    1
           Type: PositiveInteger
```

The right-hand side of the expression is evaluated, yieldig 1. This value is then assigned to b. Thus a and b both have the value 1 after the sequence of assignments.

```
  a := 2
    2
           Type: PositiveInteger
```

What is the value of b if a is assigned the value 2?

```
  b
    1
           Type: PositiveInteger
```

The value of b is left unchanged.

This is what we mean when we say this kind of assignment is immediate. The variable b has no dependency on a after the initial assignment. This is the usual notion of assignment in programming languages such as C, Pascal, and Fortran.

```
========================================================================
Delayed Assignment
========================================================================
```

Axiom provides delayed assignment with "==". This implements a delayed evaluation of the right-hand side and dependency checking. The syntax for delayed assignment is

```
   variable == expression
```

The value returned by a delayed assignment is the unique value of Void.

```
  a == 1
           Type: Void
```

```
  b == a
          Type: Void
```

Using a and b as above, these are the corresponding delayed assignments.

```
  a
   Compiling body of rule a to compute value of type PositiveInteger
   1
          Type: PositiveInteger
```

The right-hand side of each delayed assignment is left unevaluated until the variables on the left-hand sides are evaluated.

```
  b
   Compiling body of rule b to compute value of type PositiveInteger
   1
          Type: PositiveInteger
```

This gives the same results as before. But if we change a to 2

```
  a == 2
   Compiled code for a has been cleared.
   Compiled code for b has been cleared.
   1 old definition(s) deleted for function or rule a
          Type: Void
```

Then a evaluates to 2, as expected

```
  a
   Compiling body of rule a to compute value of type PositiveInteger
   2
          Type: PositiveInteger
```

but the value of b reflects the change to a

```
  b
   Compiling body of rule b to compute value of type PositiveInteger
   2
          Type: PositiveInteger
```

```
========================================================================
Multiple Immediate Assignments
========================================================================
```

It is possible to set several variables at the same time by using a tuple of variables and a tuple of expressions. A tuple is a collection

of things separated by commas, often surrounded by parentheses. The syntax for multiple immediate assignment is

```
 ( var1, var2, ..., varN ) := ( expr1, expr2, ..., exprN )
```

The value returned by an immediate assignment is the value of exprN.

```
 ( x, y ) := ( 1, 2 )
   2
            Type: PositiveInteger
```

This sets x to 1 and y to 2. Multiple immediate assignments are parallel in the sense that the expressions on the right are all evaluated before any assignments on the left are made. However, the order of evaluation of these expressions is undefined.

```
 ( x, y ) := ( y, x )
   1
            Type: PositiveInteger

  x
   2
            Type: PositiveInteger
```

The variable x now has the previous value of y.

```
  y
   1
            Type: PositiveInteger
```

The variable y now has the previous value of x.

There is no syntactic form for multiple delayed assignments.

6.2 syntax blocks

⟨*blocks.help*⟩≡

```
==========================================================================
Blocks
==========================================================================

A block is a sequence of expressions evaluated in the order that they
appear, except as modified by control expressions such as leave, return,
iterate, and if-then-else constructions. The value of a block is the
value of the expression last evaluated in the block.

To leave a block early, use "=>". For example,

    i < 0 => x

The expression before the "=>" must evaluate to true or false. The
expression following the "=>" is the return value of the block.

A block can be constructed in two ways:

  1. the expressions can be separated by semicolons and the resulting
     expression surrounded by parentheses, and
  2. the expressions can be written on succeeding lines with each line
     indented the same number of spaces (which must be greater than zero).
     A block entered in this form is called a pile

Only the first form is available if you are entering expressions directly
to Axiom. Both forms are available in .input files. The syntax for a simple
block of expressions entered interactively is

  ( expression1 ; expression2 ; ... ; expressionN )

The value returned by a block is the value of an "=>" expression, or
expressionN if no "=>" is encountered.

In .input files, blocks can also be written in piles. The examples
given here are assumed to come from .input files.

  a :=
    i := gcd(234,672)
    i := 2*i**5 - i + 1
    1 / i

      1
    -----
```

```
    23323
                Type: Fraction Integer
```

In this example, we assign a rational number to a using a block consisting of three expressions. This block is written as a pile. Each expression in the pile has the same indentation, in this case two spaces to the right of the first line.

```
  a := ( i := gcd(234,672); i := 2*i**5 - i + 1; 1 / i )

      1
    -----
    23323
                Type: Fraction Integer
```

Here is the same block written on one line. This is how you are required to enter it at the input prompt.

```
  ( a := 1; b := 2; c := 3; [a,b,c] )
    [1,2,3]
                Type: List PositiveInteger
```

AAxiom gives you two ways of writing a block and the preferred way in an .input file is to use a pile. Roughly speaking, a pile is a block whose consituent expressions are indented the same amount. You begin a pile by starting a new line for the first expression, indenting it to the right of the previous line. You then enter the second expression on a new line, vertically aligning it with the first line. And so on. If you need to enter an inner pile, further indent its lines to the right of the outer pile. Axiom knows where a pile ends. It ends when a subsequent line is indented to the left of the pile or the end of the file.

Also See:
o)help if
o)help repeat
o)help while
o)help for
o)help suchthat
o)help parallel
o)help lists

[1]

6.3 system clef

⟨*clef.help*⟩≡

```
Entering printable keys generally inserts new text into the buffer (unless
in overwrite mode, see below).  Other special keys can be used to modify
the text in the buffer.  In the description of the keys below, ^n means
Control-n, or holding the CONTROL key down while pressing "n".  Errors
will ring the terminal bell.

^A/^E   : Move cursor to beginning/end of the line.
^F/^B   : Move cursor forward/backward one character.
^D      : Delete the character under the cursor.
^H, DEL : Delete the character to the left of the cursor.
^K      : Kill from the cursor to the end of line.
^L      : Redraw current line.
^O      : Toggle overwrite/insert mode. Initially in insert mode. Text
          added in overwrite mode (including yanks) overwrite
          existing text, while insert mode does not overwrite.
^P/^N   : Move to previous/next item on history list.
^R/^S   : Perform incremental reverse/forward search for string on
          the history list.  Typing normal characters adds to the current
          search string and searches for a match. Typing ^R/^S marks
          the start of a new search, and moves on to the next match.
          Typing ^H or DEL deletes the last character from the search
          string, and searches from the starting location of the last search.
          Therefore, repeated DEL's appear to unwind to the match nearest
          the point at which the last ^R or ^S was typed.  If DEL is
          repeated until the search string is empty the search location
          begins from the start of the history list.  Typing ESC or
          any other editing character accepts the current match and
          loads it into the buffer, terminating the search.
^T      : Toggle the characters under and to the left of the cursor.
^Y      : Yank previously killed text back at current location.  Note that
          this will overwrite or insert, depending on the current mode.
^U      : Show help (this text).
TAB     : Perform command completion based on word to the left of the cursor.
          Words are deemed to contain only the alphanumeric and the % ! ? _
          characters.
NL, CR  : returns current buffer to the program.
```

[1] "if" (6.6 p 56) "repeat" (6.10 p 64) "while" (6.13 p 71) "for" (6.5 p 51) "suchthat" (6.11 p 69) "parallel" (6.9 p 61) "lists" (**??** p **??**)

```
DOS and ANSI terminal arrow key sequences are recognized, and act like:

  up    : same as ^P
  down  : same as ^N
  left  : same as ^B
  right : same as ^F
```

6.4 syntax collection

⟨*collection.help*⟩≡

```
==========================================================================
Collection -- Creating Lists and Streams with Iterators
==========================================================================

All of the loop expressions which do not use the repeat leave or
iterate words can be used to create lists and streams. For example:

This creates a simple list of the integers from 1 to 10:

  list := [i for i in 1..10]
   [1,2,3,4,5,6,7,8,9,10]
                      Type: List PositiveInteger

Create a stream of the integers greater than or equal to 1:

  stream := [i for i in 1..]
   [1,2,3,4,5,6,7,...]
                      Type: Stream PositiveInteger

This is a list of the prime numbers between 1 and 10, inclusive:

  [i for i in 1..10 | prime? i]
   [2,3,5,7]
                      Type: List PositiveInteger

This is a stream of the prime integers greater than or equal to 1:

  [i for i in 1.. | prime? i]
   [2,3,5,7,11,13,17,...]
                      Type: Stream PositiveInteger

This is a list of the integers between 1 and 10, inclusive, whose
squares are less than 700:

  [i for i in 1..10 while i*i < 700]
   [1,2,3,4,5,6,7,8,9,10]
                      Type: List PositiveInteger

This is a stream of the integers greater than or equal to 1 whose
squares are less than 700:

  [i for i in 1.. while i*i < 700]
   [1,2,3,4,5,6,7,...]
```

```
                              Type: Stream PositiveInteger
```

The general syntax of a collection is

```
  [ collectExpression iterator1 iterator2 ... iteratorN ]
```

where each iterator is either a for or a while clause. The loop terminates immedidately when the end test of any iterator succeeds or when a return expression is evaluated in collectExpression. The value returned by the collection is either a list or a stream of elements, one for each iteration of the collectExpression.

Be careful when you use while to create a stream. By default Axiom tries to compute and display the first ten elements of a stream. If the while condition is not satisfied quickly, Axiom can spend a long (potentially infinite) time trying to compute the elements. Use

```
  )set streams calculate
```

to change the defaults to something else. This also affects the number of terms computed and displayed for power series. For the purposes of these examples we have use this system command to display fewer than ten terms.

6.5 syntax for

⟨*for.help*⟩≡
```
==========================================================================
for loops
==========================================================================

Axiom provide the for and in keywords in repeat loops, allowing you
to integrate across all elements of a list, or to have a variable take
on integral values from a lower bound to an upper bound. We shall refer
to these modifying clauses of repeat loops as for clauses. These clauses
can be present in addition to while clauses (See )help while). As with
all other types of repeat loops, leave (see )help leave) can be used to
prematurely terminate evaluation of the loop.

The syntax for a simple loop using for is

  for iterator repeat loopbody

The iterator has several forms. Each form has an end test which is
evaluted before loopbody is evaluated. A for loop terminates immediately
when the end test succeeds (evaluates to true) or when a leave or return
expression is evaluated in loopbody. The value returned by the loop is
the unique value of Void.

==========================================================================
for i in n..m repeat
==========================================================================

If for is followed by a variable name, the in keyword and then an integer
segment of the form n..m, the end test for this loop is the predicate
i > m. The body of the loop is evaluated m-n+1 times if this number is
greater than 0. If this number is less than or equal to 0, the loop body
is not evaluated at all.

The variable i has the value n, n+1, ..., m for successive iterations
of the loop body. The loop variable is a local variable within the loop
body. Its value is not available outside the loop body and its value and
type within the loop body completely mask any outer definition of a
variable with the same name.

  for i in 10..12 repeat output(i**3)
   1000
   1331
   1728
                        Type: Void
```

```
The loop prints the values of 10^3, 11^3, and 12^3.

  a := [1,2,3]
   [1,2,3]
                        Type: List PositiveInteger

  for i in 1..#a repeat output(a.i)
   1
   2
   3
                        Type: Void

Iterate across this list using "." to access the elements of a list
and the # operation to count its elements.

This type of iteration is applicable to anything that uses ".". You
can also use it with functions that use indices to extract elements.

   m := matrix [[1,2],[4,3],[9,0]]
    +-    -+
    | 1  2 |
    | 4  3 |
    | 9  0 |
    +-    -+
                        Type: Matrix Integer

Define m to be a matrix.

   for i in 1..nrows(m) repeat output row(m.i)
    [1,2]
    [4,3]
    [9,0]
                        Type: Void

Display the rows of m.

You can iterate with for-loops.

   for i in 1..5 repeat
     if odd?(i) then iterate
     output(i)
    2
    4
                        Type: Void
```

```
Display the even integers in a segment.

========================================================================
for i in n..m by s repeat
========================================================================

By default, the difference between values taken on by a variable in
loops such as

  for i in n..m repeat ...

is 1. It is possible to supply another, possibly negative, step value
by using the by keyword along with for and in. Like the upper and lower
bounds, the step value following the by keyword must be an integer. Note
that the loop

  for i in 1..2 by 0 repeat output(i)

will not terminate by itself, as the step value does not change the
index from its initial value of 1.

  for i in 1..5 by 2 repeat output(i)
   1
   3
   5
                      Type: Void

This expression displays the odd integers between two bounds.

  for i in 5..1 by -2 repeat output(i)
   5
   3
   1
                      Type: Void

Use this to display the numbers in reverse order.

========================================================================
for i in n.. repeat
========================================================================

If the value after the ".." is omitted, the loop has no end test. A
potentially infinite loop is thus created. The variable is given the
successive values n, n+1, n+2, ... and the loop is terminated only
if a leave or return expression is evaluated in the loop body. However,
you may also add some other modifying clause on the repeat, for example,
```

```
a while clause, to stop the loop.

  for i in 15.. while not prime?(i) repeat output(i)
   15
   16
                        Type: Void

This loop displays the integers greater than or equal to 15 and less
than the first prime number greater than 15.

========================================================================
for x in l repeat
========================================================================

Another variant of the for loop has the form:

  for x in list repeat loopbody

This form is used when you want to iterate directly over the elements
of a list. In this form of the for loop, the variable x takes on the
value of each successive element in l. The end test is most simply
stated in English: "are there no more x in l?"

  l := [0, -5, 3]
   [0, -5, 3]
                        Type: List Integer

  for x in l repeat output(x)
   0
   -5
   3
                        Type: Void

This displays all of the elements of the list l, one per line.

Since the list constructing expression

  expand [n..m]

creates the list

  [n, n+1, ..., m]

you might be tempted to think that the loops

  for i in n..m repeat output(i)
```

and

```
  for x in expand [n..m] repeat output(x)
```

are equivalent. The second form first creates the expanded list (no matter how large it might be) and then does the iteration. The first form potentially runs in much less space, as the index variable i is simply incremented once per loop and the list is not actually created. Using the first form is much more efficient.

Of course, sometimes you really want to iterate across a specific list. This displays each of the factors of 2400000:

```
  for f in factors(factor(2400000)) repeat output(f)
   [factor= 2, exponent= 8]
   [factor= 3, exponent= 1]
   [factor= 5, exponent= 5]
                      Type: Void
```

6.6 syntax if

⟨*if.help*⟩≡

```
==========================================================================
If-then-else
==========================================================================

Like many other programming languages, Axiom uses the three keywords
if, then, and else to form conditional expressions. The else part of
the conditional is optional. The expression between the if and then
keywords is a predicate: an expression that evaluates to or is
convertible to either true or false, that is, a Boolean.

The syntax for conditional expressions is

   if predicate then expression1 else expression2

where the "else expression2" part is optional. The value returned from
a conditional expression is expression1 if the predicate evaluates to
true and expression2 otherwise. If no else clause is given, the value
is always the unique value of Void.

An if-then-else expression always returns a value. If the else clause
is missing then the entire expression returns the unique value of Void.
If both clauses are present, the type of the value returned by if is
obtained by resolving the types of the values of the two clauses.

The predicate must evaluate to, or be convertible to, an object of type
Boolean: true or false. By default, the equal sign "=" creates an equation.

   x + 1 = y
    x + 1 = y
                Type: Equation Polynomial Integer

This is an equation, not a boolean condition. In particular, it is
an object of type Equation Polynomial Integer.

However, for predicates in if expressions, Axiom places a default
target type of Boolean on the predicate and equality testing is performed.
Thus you need not qualify the "=" in any way. In other contexts you may
need to tell Axiom that you want to test for equality rather than create
an equation. In these cases, use "@" and a target type of Boolean.

The compound symbol meaning "not equal" in Axiom is "~=". This can be
used directly without a package call or a target specification. The
expression "a ~= b" is directly translated to "not(a = b)".
```

Many other functions have return values of type Boolean. These include <, <=, >, >=, ~=, and member?. By convention, operations with names ending in "?" return Boolean values.

The usual rules for piles are suspended for conditional expressions. In .input files, the then and else keywords can begin in the same column as the corresponding if by may also appear to the right. Each of the following styles of writing if-then-else expressions is acceptable:

```
if i>0 then output("positive") else output("nonpositive")

if i>0 then output("positive")
  else output("nonpositive")

if i>0 then output("positive")
else output("nonpositive")

if i>0
then output("positive")
else output("nonpositive")

if i>0
  then output("positive")
  else output("nonpositive")
```

A block can follow the then or else keywords. In the following two assignments to a, the then and else clauses each are followed by two line piles. The value returned in each is the value of the second line.

```
a :=
  if i > 0 then
    j := sin(i * pi())
    exp(j + 1/j)
  else
    j := cos(i * 0.5 * pi())
    log(abs(j)**5 + i)


a :=
  if i > 0
    then
      j := sin(i * pi())
      exp(j + 1/j)
    else
      j := cos(i * 0.5 * pi())
```

```
          log(abs(j)**5 + i)
```

These are both equivalent to the following:

```
  a :=
    if i > 0 then (j := sin(i * pi()); exp(j + 1/j))
    else (j := cos(i * 0.5 * pi()); log(abs(j)**5 + i))
```

6.7 syntax iterate

⟨*iterate.help*⟩≡

```
==========================================================================
iterate in loops
==========================================================================

Axiom provides an iterate expression that skips over the remainder
of a loop body and starts the next loop execution. We first initialize
a counter.

  i := 0
   0
                        Type: NonNegativeInteger

Display the even integers from 2 to 5:

  repeat
    i := i + 1
    if i > 5 then leave
    if odd?(i) then iterate
    output(i)
   2
   4
                        Type: Void
```

6.8 syntax leave

⟨*leave.help*⟩≡

```
==========================================================================
leave in loops
==========================================================================

The leave keyword is often more useful in terminating a loop. A
leave causes control to transfer to the expression immediately following
the loop. As loops always return the unique value of Void, you cannot
return a value with leave. That is, leave takes no argument.

  f() ==
    i := 1
    repeat
      if factorial(i) > 1000 then leave
      i := i + 1
    i
                        Type: Void

This example is a modification of the last example in the previous
section. Instead of using return we'll use leave.

  f()
   7
                        Type: PositiveInteger

The loop terminates when factorial(i) gets big enough. The last line
of the function evaluates to the corresponding "good" value of i
and the function terminates, returning that value.

You can only use leave to terminate the evaluation of one loop. Lets
consider a loop within a loop, that is, a loop with a nested loop.
First, we initialize two counter variables.

  (i,j) := (1,1)
   1
                        Type: PositiveInteger

  repeat
    repeat
      if (i + j) > 10 then leave
      j := j + 1
    if (i + j) > 10 then leave
    i := i + 1
                        Type: Void
```

Nested loops must have multiple leave expressions at the appropriate nesting level. How would you rewrite this so (i + j) > 10 is only evaluated once?

```
=========================================================================
leave vs => in loop bodies
=========================================================================
```

Compare the following two loops:

```
  i := 1                        i := 1
  repeat                        repeat
    i := i + 1                    i := i + 1
    i > 3 => i                    if i > 3 then leave
    output(i)                     output(i)
```

In the example on the left, the values 2 and 3 for i are displayed but then the "=>" does not allow control to reach the call to output again. The loop will not terminate until you run out of space or interrupt the execution. The variable i will continue to be incremented because the "=>" only means to leave the block, not the loop.

In the example on the right, upon reaching 4, the leave will be executed, and both the block and the loop will terminate. This is one of the reasons why both "=>" and leave are provided. Using a while clase with the "=>" lets you simulate the action of leave.

6.9 syntax parallel

⟨*parallel.help*⟩≡

```
==========================================================================
parallel iteration
==========================================================================

Sometimes you want to iterate across two lists in parallel, or perhaps
you want to traverse a list while incrementing a variable.

The general syntax of a repeat loop is

 iterator1, iterator2, ..., iteratorN repeat loopbody

where each iterator is either a for or a while clause. The loop
terminates immediately when the end test of any iterator succeeds or
when a leave or return expression is evaluated in loopbody. The value
returned by the loop is the unique value of Void.

  l := [1,3,5,7]
   [1,3,5,7]
                      Type: List PositiveInteger

  m := [100,200]
   [100,200]
                      Type: List PositiveInteger

  sum := 0
   0
                      Type: NonNegativeInteger

Here we write a loop to iterate across two lists, computing the sum
of the pairwise product of the elements:

  for x in l for y in m repeat
    sum := sum + x*y
                      Type: Void

The last two elements of l are not used in the calculation because
m has two fewer elements than l.

  sum
   700
                      Type: NonNegativeInteger

This is the "dot product".
```

Next we write a loop to compute the sum of the products of the loop elements with their positions in the loop.

```
l := [2,3,5,7,11,13,17,19,23,29,31,37]
 [2,3,5,7,11,13,17,19,23,29,31,37]
                    Type: List PositiveInteger

sum := 0
 0
                    Type: NonNegativeInteger

for i in 0.. for x in l repeat sum := i * x
                    Type: Void
```

Here looping stops when the list l is exhaused, even though the for i in 0.. specifies no terminating condition.

```
sum
 407
                    Type: NonNegativeInteger
```

When "|" is used to qualify any of the for clauses in a parallel iteration, the variables in the predicates can be from an outer scope or from a for clause in or to the left of the modified clause.

This is correct:

```
for i in 1..10 repeat
  for j in 200..300 | ood? (i+j) repeat
    output [i,j]
```

But this is not correct. The variable j has not been defined outside the inner loop:

```
for i in 1..01 | odd? (i+j) repeat -- wrong, j not defined
  for j in 200..300 repeat
    output [i,j]
```

It is possible to mix several of repeat modifying clauses on a loop:

```
for i in 1..10
  for j in 151..160 | odd? j
    while i + j < 160 repeat
      output [i,j]
 [1,151]
```

```
[3,153]
                    Type: Void

Here are useful rules for composing loop expressions:

 1. while predicates can only refer to variables that are global (or
    in an outer scope) or that are defined in for clauses to the left
    of the predicate.
 2. A "such that" predicate (somthing following "|") must directly
    follow a for clause and can only refer to variables that are
    global (or in an outer scope) or defined in the modified for clause
    or any for clause to the left.
```

6.10 syntax repeat

⟨*repeat.help*⟩≡

```
==========================================================================
Repeat Loops
==========================================================================

A loop is an expression that contains another expression, called the loop
body, which is to be evaluated zero or more times. All loops contain the
repeat keyword and return the unique value of Void. Loops can contain
inner loops to any depth.

The most basic loop is of the form

  repeat loopbody

Unless loopbody contains a leave or return expression, the loop repeats
foreer. The value returned by the loop is the unique value of Void.

Axiom tries to determine completely the type of every object in a loop
and then to translate the loop body to Lisp or even to machine code. This
translation is called compilation.

If Axiom decides that it cannot compile the loop, it issues a message
stating the problem and then the following message:

  We will attemp to step through and interpret the code

It is still possible that Axiom can evalute the loop but in interpret-code
mode.

==========================================================================
Return in Loops
==========================================================================

A return expression is used to exit a function with a particular value.
In particular, if a return is in a loop within the function, the loop
is terminated whenever the return is evaluated.

  f() ==
    i := 1
    repeat
      if factorial(i) > 1000 then return i
      i := i + 1
                      Type: Void
```

```
  f()
                        Type: Void
```

When factorial(i) is big enough, control passes from inside the loop all the way outside the function, returning the value of i (so we think). What went wrong? Isn't it obvious that this function should return an integer? Well, Axiom makes no attempt to analyze the structure of a loop to determine if it always returns a value because, in general, this is impossible. So Axiom has this simple rule: the type of the function is determined by the type of its body, in this case a block. The normal value of a block is the value of its last expression, in this case, a loop. And the value of every loop is the unique value of Void. So the return type of f is Void.

There are two ways to fix this. The best way is for you to tell Axiom what the return type of f is. You do this by giving f a declaration

```
   f:() -> Integer
```

prior to calling for its value. This tells Axiom "trust me -- an integer is returned". Another way is to add a dummy expression as follows.

```
  f() ==
    i := 1
    repeat
      if factorial(i) > 1000 then return i
      i := i + 1
    0
                        Type: Void
```

Note that the dummy expression will never be evaluated but it is the last expression in the function and will determine the return type.

```
  f()
   7
                        Type: PositiveInteger
```

leave in loops

The leave keyword is often more useful in terminating a loop. A leave causes control to transfer to the expression immediately following the loop. As loops always return the unique value of Void, you cannot return a value with leave. That is, leave takes no argument.

```
f() ==
  i := 1
  repeat
    if factorial(i) > 1000 then leave
    i := i + 1
  i
                    Type: Void
```

This example is a modification of the last example in the previous section. Instead of using return we'll use leave.

```
f()
 7
                    Type: PositiveInteger
```

The loop terminates when factorial(i) gets big enough. The last line of the function evaluates to the corresponding "good" value of i and the function terminates, returning that value.

You can only use leave to terminate the evaluation of one loop. Lets consider a loop within a loop, that is, a loop with a nested loop. First, we initialize two counter variables.

```
(i,j) := (1,1)
 1
                    Type: PositiveInteger
```

```
repeat
  repeat
    if (i + j) > 10 then leave
    j := j + 1
  if (i + j) > 10 then leave
  i := i + 1
                    Type: Void
```

Nested loops must have multiple leave expressions at the appropriate nesting level. How would you rewrite this so (i + j) > 10 is only evaluated once?

```
========================================================================
leave vs => in loop bodies
========================================================================
```

Compare the following two loops:

```
i := 1                      i := 1
```

```
  repeat                        repeat
    i := i + 1                    i := i + 1
    i > 3 => i                    if i > 3 then leave
    output(i)                     output(i)
```

In the example on the left, the values 2 and 3 for i are displayed but then the "=>" does not allow control to reach the call to output again. The loop will not terminate until you run out of space or interrupt the execution. The variable i will continue to be incremented because the "=>" only means to leave the block, not the loop.

In the example on the right, upon reaching 4, the leave will be executed, and both the block and the loop will terminate. This is one of the reasons why both "=>" and leave are provided. Using a while clase with the "=>" lets you simulate the action of leave.

```
======================================================================
iterate in loops
======================================================================
```

Axiom provides an iterate expression that skips over the remainder of a loop body and starts the next loop execution. We first initialize a counter.

```
  i := 0
   0
                        Type: NonNegativeInteger
```

Display the even integers from 2 to 5:

```
  repeat
    i := i + 1
    if i > 5 then leave
    if odd?(i) then iterate
    output(i)
   2
   4
                        Type: Void
```

Also See:
o)help blocks
o)help if
o)help while
o)help for
o)help suchthat
o)help parallel

- `)help lists`

[2]

6.11 syntax suchthat

⟨*suchthat.help*⟩≡

```
==========================================================================
Such that predicates
==========================================================================

A for loop can be followed by a "|" and then a predicate. The predicate
qualifies the use of the values from the iterator that follows the for.
Think of the vertical bar "|" as the phrase "such that".

  for n in 0..4 | odd? n repeat output n
   1
   3
                       Type: Void

This loop expression prints out the integers n in the given segment
such that n is odd.

A for loop can also be written

  for iterator | predicate repeat loopbody

which is equivalent to:

  for iterator repeat if predicate then loopbody else iterate

The predicate need not refer only to the variable in the for clause.
Any variable in an outer scope can be part of the predicate.

  for i in 1..50 repeat
    for j in 1..50 | factorial(i+j) < 25 repeat
      output [i,j]
   [1,1]
   [1,2]
   [1,3]
   [2,1]
   [2,2]
   [3,1]
                       Type: Void
```

[2] "blocks" (6.2 p 45) "if" (6.6 p 56) "while" (6.13 p 71) "for" (6.5 p 51) "suchthat" (6.11 p 69) "parallel" (6.9 p 61) "lists" (**??** p **??**)

6.12 syntax syntax

⟨*syntax.help*⟩≡

```
The Axiom Interactive Language has the following features documented here.

More information is available by typing

  )help feature

where feature is one of:

  assignment -- Immediate and delayed assignments
  blocks     -- Blocks of expressions
  collection -- creating lists with iterators
  for        -- for loops
  if         -- If-then-else statements
  iterate    -- using iterate in loops
  leave      -- using leave in loops
  parallel   -- parallel iterations
  repeat     -- repeat loops
  suchthat   -- suchthat predicates
  while      -- while loops
```

6.13 syntax while

⟨*while.help*⟩≡

```
==========================================================================
while loops
==========================================================================

The repeat in a loop can be modified by adding one or more while
clauses. Each clause contains a predicate immediately following the
while keyword. The predicate is tested before the evaluation of the
body of the loop. The loop body is evaluated whenever the predicate
in a while clause is true.

The syntax for a simple loop using while is

  while predicate repeat loopbody

The predicate is evaluated before loopbody is evaluated. A while loop
terminates immediately when predicate evaluates to false or when a
leave or return expression is evaluted. See )help repeat for more
information on leave and return.

Here is a simple example of using while in a loop. We first initialize
the counter.

  i := 1
   1
                      Type: PositiveInteger

  while i < 1 repeat
    output "hello"
    i := i + 1
                      Type: Void

The steps involved in computing this example are
 (1) set i to 1
 (2) test the condition i < 1 and determine that it is not true
 (3) do not evaluate the loop body and therefore do not display "hello"

  (x, y) := (1, 1)
   1
                      Type: PositiveInteger

If you have multiple predicates to be tested use the logical and
operation to separate them. Axiom evaluates these predicates from
left to right.
```

```
while x < 4 and y < 10 repeat
  output [x,y]
  x := x + 1
  y := y + 2
 [1,1]
 [2,3]
 [3,5]
                    Type: Void
```

A leave expression can be included in a loop body to terminate a loop even if the predicate in any while clauses are not false.

```
(x, y) := (1, 1)
 1
                    Type: PositiveInteger

while x < 4 and y < 10 repeat
  if x + y > 7 then leave
  output [x,y]
  x := x + 1
  y := y + 2
 [1,1]
 [2,3]
                    Type: Void
```

Chapter 7

System Command Handling

7.0.1 defvar $systemCommands

The system commands are the top-level commands available in Axiom that can all be invoked by prefixing the symbol with a closed-paren. Thus, to see they copyright you type:

```
)copyright
```

New commands need to be added to this table. The command invoked will be the first entry of the pair and the "user level" of the command will be the second entry.

See:

- The "abbreviations" (8.2.1 p 87) command
- The "boot" (9 p 91) command
- The "browse" (10 p 93) command
- The "cd" (11 p 99) command
- The "clear" (12.3.1 p 103) command
- The "close" (13.2.2 p 113) command
- The "compiler" (14.2.1 p 121) command
- The "copyright" (15.2.1 p 131) command
- The "credits" (16.3.1 p 133) command
- The "display" (17.2.1 p 137) command

- The “edit” (18.2.1 p 146) command
- The “fin” (19 p 149) command
- The “frame” (20.5.16 p 167) command
- The “help” (21.2.1 p 174) command
- The “history” (22.4.7 p 183) command
- The “lisp” (25 p 223) command
- The “library” (24 p 221) command
- The “load” (26 p 225) command
- The “ltrace” (27 p 227) command
- The “pquit” (28.2.1 p 230) command
- The “quit” (29.2.1 p 234) command
- The “read” (30 p 237) command
- The “savesystem” (31 p 239) command
- The “set” (32.40.1 p 372) command
- The “show” (33 p 377) command
- The “spool” (34 p 379) command
- The “summary” (35.1.1 p 382) command
- The “synonym” (36 p 383) command
- The “system” (37 p 385) command
- The “trace” (38.1.2 p 392) command
- The “undo” (39.3.2 p 450) command
- The “what” (40.1.2 p 467) command
- The “with” (41.1.1 p 473) command
- The “workfiles” (42.1.1 p 475) command
- The “zsystemdevelopment” (43.1.1 p 479) command

7.1 Variables Used

7.1.1 defvar $systemCommands

⟨*initvars*⟩+≡
```
(defvar |$systemCommands| nil)
```

⟨*postvars*⟩+≡

```
(eval-when (eval load)
 (setq |$systemCommands|
 '(
   (|abbreviations|                  . |compiler|   )
   (|boot|                           . |development|)
   (|browse|                         . |development|)
   (|cd|                             . |interpreter|)
   (|clear|                          . |interpreter|)
   (|close|                          . |interpreter|)
   (|compiler|                       . |compiler|   )
   (|copyright|                      . |interpreter|)
   (|credits|                        . |interpreter|)
   (|display|                        . |interpreter|)
   (|edit|                           . |interpreter|)
   (|fin|                            . |development|)
   (|frame|                          . |interpreter|)
   (|help|                           . |interpreter|)
   (|history|                        . |interpreter|)
;; (|input|                          . |interpreter|)
   (|lisp|                           . |development|)
   (|library|                        . |interpreter|)
   (|load|                           . |interpreter|)
   (|ltrace|                         . |interpreter|)
   (|pquit|                          . |interpreter|)
   (|quit|                           . |interpreter|)
   (|read|                           . |interpreter|)
   (|savesystem|                     . |interpreter|)
   (|set|                            . |interpreter|)
   (|show|                           . |interpreter|)
   (|spool|                          . |interpreter|)
   (|summary|                        . |interpreter|)
   (|synonym|                        . |interpreter|)
   (|system|                         . |interpreter|)
   (|trace|                          . |interpreter|)
   (|undo|                           . |interpreter|)
   (|what|                           . |interpreter|)
   (|with|                           . |interpreter|)
   (|workfiles|                      . |development|)
   (|zsystemdevelopment| . |interpreter|)
 )))
```

7.1.2 defvar $syscommands

This table is used to look up a symbol to see if it might be a command.

⟨*initvars*⟩+≡
```
(defvar $syscommands nil)
```

⟨*postvars*⟩+≡
```
(eval-when (eval load)
 (setq $syscommands (mapcar #'car |$systemCommands|)))
```

7.1.3 defvar $noParseCommands

This is a list of the commands which have their arguments passed verbatim. Certain functions, such as the lisp function need to be able to handle all kinds of input that will not be acceptable to the interpreter.

⟨*initvars*⟩+≡
```
(defvar |$noParseCommands| nil)
```

⟨*postvars*⟩+≡
```
(eval-when (eval load)
 (setq |$noParseCommands|
 '(|boot| |copyright| |credits| |fin| |lisp| |pquit| |quit| |synonym| |system|
    )))
```

7.2 Functions

7.2.1 defun handleNoParseCommands

The system commands given by the global variable `$noParseCommands` require essentially no preprocessing/parsing of their arguments. Here we dispatch the functions which implement these commands.

There are four standard commands which receive arguments

- boot
- lisp
- synonym
- system

There are five standard commands which do not receive arguments –

- quit
- fin
- pquit
- credits
- copyright

As these commands do not necessarily exhaust those mentioned in `$noParseCommands`, we provide a generic dispatch based on two conventions: commands which do not require an argument name themselves, those which do have their names prefixed by "np". This makes it possible to dynamically define new system commands provided you handle the argument parsing.

⟨*defun handleNoParseCommands*⟩≡

```
(defun |handleNoParseCommands| (unab string)
 (let (spaceindex funname)
  (setq string (|stripSpaces| string))
  (setq spaceindex (search " " string))
  (cond
   ((eq unab '|lisp|)
    (if spaceindex
     (|nplisp| (|stripLisp| string))
     (|sayKeyedMsg| 's2iv0005 nil)))
   ((eq unab '|boot|)
    (if spaceindex
     (|npboot| (subseq string (1+ spaceindex)))
```

```
    (|sayKeyedMsg| 's2iv0005 nil)))
  ((eq unab '|system|)
   (if spaceindex
     (|npsystem| unab string)
     (|sayKeyedMsg| 's2iv0005 nil)))
  ((eq unab '|synonym|)
   (if spaceindex
    (|npsynonym| unab (subseq string (1+ spaceindex)))
    (|npsynonym| unab "")))
  ((null spaceindex)
   (funcall unab))
  ((|member| unab '(|quit| |fin| |pquit| |credits| |copyright|))
   (|sayKeyedMsg| 's2iv0005 nil))
  (t
   (setq funname (intern (concat "np" (string unab))))
   (funcall funname (subseq string (1+ spaceindex)))))))
```

7.2.2 defvar $tokenCommands

This is a list of the commands that expect the interpreter to parse their arguments. Thus the history command expects that Axiom will have tokenized and validated the input before calling the history function.

⟨*initvars*⟩+≡

```
(defvar |$tokenCommands| nil)
```

⟨*postvars*⟩+≡
```
(eval-when (eval load)
 (setq |$tokenCommands|
 '( |abbreviations|
    |cd|
    |clear|
    |close|
    |compiler|
    |depends|
    |display|
    |edit|
    |frame|
    |frame|
    |help|
    |history|
    |input|
    |library|
    |load|
    |ltrace|
    |read|
    |savesystem|
    |set|
    |spool|
    |undo|
    |what|
    |with|
    |workfiles|
    |zsystemdevelopment|
    )))
```

7.2.3 defvar $InitialCommandSynonymAlist

Axiom can create "synonyms" for commands. We create an initial table of synonyms which are in common use.

⟨*initvars*⟩+≡
```
(defvar |$InitialCommandSynonymAlist| nil)
```

⟨*postvars*⟩+≡

```
(eval-when (eval load)
 (setq |$InitialCommandSynonymAlist|
   '(
       (|?|          . "what commands")
       (|ap|         . "what things")
       (|apr|        . "what things")
       (|apropos|    . "what things")
       (|cache|      . "set functions cache")
       (|cl|         . "clear")
       (|cls|        . "zsystemdevelopment )cls")
       (|cms|        . "system")
       (|co|         . "compiler")
       (|d|          . "display")
       (|dep|        . "display dependents")
       (|dependents| . "display dependents")
       (|e|          . "edit")
       (|expose|     . "set expose add constructor")
       (|fc|         . "zsystemdevelopment )c")
       (|fd|         . "zsystemdevelopment )d")
       (|fdt|        . "zsystemdevelopment )dt")
       (|fct|        . "zsystemdevelopment )ct")
       (|fctl|       . "zsystemdevelopment )ctl")
       (|fe|         . "zsystemdevelopment )e")
       (|fec|        . "zsystemdevelopment )ec")
       (|fect|       . "zsystemdevelopment )ect")
       (|fns|        . "exec spadfn")
       (|fortran|    . "set output fortran")
       (|h|          . "help")
       (|hd|         . "system hypertex &")
       (|kclam|      . "boot clearClams ( )")
       (|killcaches| . "boot clearConstructorAndLisplibCaches ( )")
       (|patch|      . "zsystemdevelopment )patch")
       (|pause|      . "zsystemdevelopment )pause")
       (|prompt|     . "set message prompt")
       (|recurrence| . "set functions recurrence")
       (|restore|    . "history )restore")
       (|save|       . "history )save")
       (|startGraphics|    .  "system $AXIOM/lib/viewman &")
       (|startNAGLink|     .  "system $AXIOM/lib/nagman &")
       (|stopGraphics|     .  "lisp (|sockSendSignal| 2 15)")
       (|stopNAGLink|      .  "lisp (|sockSendSignal| 8 15)")
       (|time|       . "set message time")
       (|type|       . "set message type")
       (|unexpose|   . "set expose drop constructor")
       (|up|         . "zsystemdevelopment )update")
```

```
        (|version|    . "lisp *yearweek*")
        (|w|          . "what")
        (|wc|         . "what categories")
        (|wd|         . "what domains")
        (|who|        . "lisp (pprint credits)")
        (|wp|         . "what packages")
        (|ws|         . "what synonyms")
  )))
```

7.2.4 defvar $CommandSynonymAlist

The actual list of synonyms is initialized to be the same as the above initial list of synonyms. The user synonyms that are added during a session are pushed onto this list for later lookup.

⟨*initvars*⟩+≡

```
  (defvar |$CommandSynonymAlist| nil)
```

⟨*postvars*⟩+≡

```
  (eval-when (eval load)
   (setq |$CommandSynonymAlist| (copy-alist |$InitialCommandSynonymAlist|)))
```

7.2.5 defun ncloopCommand

The $systemCommandFunction is set in SpadInterpretStream to point to the function InterpExecuteSpadSystemCommand. The system commands are handled by the function kept in the "hook" variable `$systemCommandFunction` which has the default function `InterpExecuteSpadSystemCommand`. Thus, when a system command is entered this function is called.

The only exception is the `)include` function which inserts the contents of a file inline in the input stream. This is useful for processing `)read` of input files.

⟨*defun ncloopCommand*⟩≡

```
  (defun |ncloopCommand| (line n)
   (let (a)
   (declare (special |$systemCommandFunction|))
    (if (setq a (|ncloopPrefix?| ")include" line))
     (|ncloopInclude1| a n)
     (progn
      (funcall |$systemCommandFunction| line)
      n))))
```

7.2.6 defun ncloopPrefix?

If we find the prefix string in the whole string starting at position zero we return the remainder of the string without the leading prefix.

⟨*defun ncloopPrefix?*⟩≡

```
(defun |ncloopPrefix?| (prefix whole)
 (when (eql (search prefix whole) 0)
  (subseq whole (length prefix))))
```

7.2.7 defun selectOptionLC

⟨*defun selectOptionLC*⟩≡

```
(defun |selectOptionLC| (x l errorFunction)
 (|selectOption| (downcase (|object2Identifier| x)) l errorFunction))
```

7.2.8 defun selectOption

⟨*defun selectOption*⟩≡

```
(defun |selectOption| (x l errorfunction)
 (let (u y)
  (cond
   ((|member| x l) x)
   ((null (identp x))
    (cond
     (errorfunction (funcall errorfunction x u))
     (t nil)))
   (t
    (setq u
     (let (t0)
       (do ((t1 l (cdr t1)) (y nil))
           ((or (atom t1) (progn (setq y (car t1)) nil)) (nreverse0 t0))
          (if (|stringPrefix?| (pname x) (pname y))
             (setq t0 (cons y t0))))))
    (cond
     ((and (pairp u) (eq (qcdr u) nil) (progn (setq y (qcar u)) t)) y)
     (errorfunction (funcall errorfunction x u))
     (t nil))))))
```

Chapter 8

)abbreviations Command

8.1 abbreviations man page

⟨*abbreviations.help*⟩≡

```
==========================================================================
A.2.  )abbreviation
==========================================================================

User Level Required:  compiler

Command Syntax:

  -  )abbreviation query  [nameOrAbbrev]
  -  )abbreviation category  abbrev  fullname []quiet]
  -  )abbreviation domain  abbrev  fullname   []quiet]
  -  )abbreviation package  abbrev  fullname  []quiet]
  -  )abbreviation remove  nameOrAbbrev

Command Description:

This command is used to query, set and remove abbreviations for category,
domain and package constructors. Every constructor must have a unique
abbreviation. This abbreviation is part of the name of the subdirectory under
which the components of the compiled constructor are stored. Furthermore, by
issuing this command you let the system know what file to load automatically
if you use a new constructor. Abbreviations must start with a letter and then
be followed by up to seven letters or digits. Any letters appearing in the
abbreviation must be in uppercase.

When used with the query argument, this command may be used to list the name
```

associated with a particular abbreviation or the abbreviation for a constructor. If no abbreviation or name is given, the names and corresponding abbreviations for all constructors are listed.

The following shows the abbreviation for the constructor List:

```
)abbreviation query List
```

The following shows the constructor name corresponding to the abbreviation NNI:

```
)abbreviation query NNI
```

The following lists all constructor names and their abbreviations.

```
)abbreviation query
```

To add an abbreviation for a constructor, use this command with category, domain or package. The following add abbreviations to the system for a category, domain and package, respectively:

```
)abbreviation domain   SET Set
)abbreviation category COMPCAT  ComplexCategory
)abbreviation package  LIST2MAP ListToMap
```

If the)quiet option is used, no output is displayed from this command. You would normally only define an abbreviation in a library source file. If this command is issued for a constructor that has already been loaded, the constructor will be reloaded next time it is referenced. In particular, you can use this command to force the automatic reloading of constructors.

To remove an abbreviation, the remove argument is used. This is usually only used to correct a previous command that set an abbreviation for a constructor name. If, in fact, the abbreviation does exist, you are prompted for confirmation of the removal request. Either of the following commands will remove the abbreviation VECTOR2 and the constructor name VectorFunctions2 from the system:

```
)abbreviation remove VECTOR2
)abbreviation remove VectorFunctions2
```

Also See:
o)compile

1

8.2 Functions

8.2.1 defun abbreviations

⟨*defun abbreviations*⟩≡
```
(defun |abbreviations| (l)
 (|abbreviationsSpad2Cmd| l))
```

[1] "compile" (?? p ??)

8.2.2 defun abbreviationsSpad2Cmd

⟨*defun abbreviationsSpad2Cmd*⟩≡

```
(defun |abbreviationsSpad2Cmd| (arg)
 (let (abopts quiet opt key type constructor t2 a b al)
  (declare (special |$options|))
  (if (null arg)
   (|helpSpad2Cmd| '(|abbreviations|))
   (progn
    (setq abopts '(|query| |domain| |category| |package| |remove|))
    (setq quiet nil)
    (do ((t0 |$options| (cdr t0)) (t1 nil))
        ((or (atom t0)
             (progn (setq t1 (car t0)) nil)
             (progn (progn (setq opt (car t1)) t1) nil))
         nil)
      (setq opt (|selectOptionLC| opt '(|quiet|) '|optionError|))
      (when (eq opt '|quiet|) (setq quiet t)))
    (when
     (and (pairp arg)
          (progn
           (setq opt (qcar arg))
           (setq al (qcdr arg))
           t))
      (setq key (|opOf| (car al)))
      (setq type (|selectOptionLC| opt abopts '|optionError|))
      (cond
       ((eq type '|query|)
        (cond
         ((null al) (|listConstructorAbbreviations|))
         ((setq constructor (|abbreviation?| key))
           (|abbQuery| constructor))
         (t (|abbQuery| key))))
       ((eq type '|remove|)
        (deldatabase key 'abbreviation))
       ((oddp (size al))
        (|sayKeyedMsg| 's2iz0002 (list type)))
       (t
        (do () (nil nil)
         (seq
          (exit
           (cond
            ((null al) (return '|fromLoop|))
            (t
              (setq t2 al)
              (setq a (car t2))
```

```
            (setq b (cadr t2))
            (setq al (cddr t2))
            (|mkUserConstructorAbbreviation| b a type)
            (setdatabase b 'abbreviation a)
            (setdatabase b 'constructorkind type))))))
        (unless quiet
          (|sayKeyedMsg| 's2iz0001 (list a type (|opOf| b))))))))))))
```

8.2.3 defun listConstructorAbbreviations

⟨*defun listConstructorAbbreviations*⟩≡

```
(defun |listConstructorAbbreviations| ()
 (let (x)
  (setq x (upcase (|queryUserKeyedMsg| 's2iz0056 nil)))
  (if (memq (string2id-n x 1) '(Y YES))
   (progn
    (|whatSpad2Cmd| '(|categories|))
    (|whatSpad2Cmd| '(|domains|))
    (|whatSpad2Cmd| '(|packages|)))
   (|sayKeyedMsg| 's2iz0057 nil))))
```

Chapter 9

)boot Command

9.1 boot man page

⟨*boot.help*⟩≡

```
==========================================================================
A.3.  )boot
==========================================================================

User Level Required:  development

Command Syntax:

  -  )boot bootExpression

Command Description:

This command is used by AXIOM system developers to execute expressions
written in the BOOT language. For example,

)boot times3(x) == 3*x

creates and compiles the Lisp function ‘‘times3’’ obtained by translating the
BOOT code.

Also See:
o )fin
o )lisp
o )set
o )system
```

[1]

9.2 Functions

This command is in the list of `$noParseCommands` 7.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 7.2.1

[1] "fin" (19 p 149) "lisp" (25 p 223) "set" (32.40.1 p 372) "system" (37 p 385)

Chapter 10

)browse Command

10.1 browse man page

⟨*browse.help*⟩≡

```
User Level Required: development

Command Syntax:

  )browse

Command Description:

This command is used by Axiom system users to start the Axiom top level
loop listening for browser connections.
```

10.2 Overview

The Axiom book on the help browser is a complete rewrite of the hyperdoc mechanism. There are several components that were needed to make this function. Most of the web browser components are described in bookvol11.pamphlet. This portion describes some of the design issues needed to support the interface.

The axServer command takes a port (defaulting to 8085) and a program to handle the browser interaction (defaulting to multiServ). The axServer function opens the port, constructs the stream, and passes the stream to multiServ. The multiServ loop processes one interaction at a time.

So the basic process is that the Axiom ")browse" command opens a socket and listens for http requests. Based on the type of request (either 'GET' or 'POST') and the content of the request, which is one of:

- command - algebra request/response
- lispcall - a lisp s-expression to be evaluated
- showcall - an Axiom)show command

the multiServ function will call a handler function to evaluate the command line and construct a response. GET requests result in a new browser page. POST requests result in an inline result.

Most responses contain the fields:

- stepnum - this is the Axiom step number
- command - this is the original command from the browser
- algebra - this is the Axiom 2D algebra output
- mathml - this is the MathML version of the Axiom algebra
- type - this is the type of the Axiom result

10.3 Browsers, MathML, and Fonts

This work has the Firefox browser as its target. Firefox has built-in support for MathML, javascript, and XMLHttpRequest handling. More details are available in bookvol11.pamphlet but the very basic machinery for communication with the browser involves a dance between the browser and the multiServ function (see the axserver.spad.pamphlet).

In particular, a simple request is embedded in a web page as:

```
<ul>
 <li>
  <input type="submit" id="p3" class="subbut"
    onclick="makeRequest('p3');"
    value="sin(x)" />
  <div id="ansp3"><div></div></div>
 </li>
</ul>
```

which says that this is an html "input" field of type "submit". The CSS display class is "subbut" which is of a different color than the surrounding text to make it obvious that you can click on this field. Clickable fields that have no response text are of class "noresult".

The javascript call to "makeRequest" gives the "id" of this input field, which must be unique in the page, as an argument. In this case, the argument is 'p3'. The "value" field holds the display text which will be passed back to Axiom as a command.

When the result arrives the "showanswer" function will select out the mathml field of the response, construct the "id" of the html div to hold the response by concatenating the string "ans" (answer) to the "id" of the request resulting, in this case, as "ansp3". The "showanswer" function will find this div and replace it with a div containing the mathml result.

The "makeRequest" function is:

```
function makeRequest(arg) {
  http_request = new XMLHttpRequest();
  var command = commandline(arg);
  //alert(command);
  http_request.open('POST', '127.0.0.1:8085', true);
  http_request.onreadystatechange = handleResponse;
  http_request.setRequestHeader('Content-Type', 'text/plain');
  http_request.send("command="+command);
  return(false);
```

It contains a request to open a local server connection to Axiom, sets "handleResponse" as the function to call on reply, sets up the type of request, fills in the command field, and sends off the http request.

When a response is received, the "handleResponse" function checks for the correct reply state, strips out the important text, and calls "showanswer".

```
function handleResponse() {
 if (http_request.readyState == 4) {
  if (http_request.status == 200) {
   showanswer(http_request.responseText,'mathAns');
  } else
  {
```

```
    alert('There was a problem with the request.'+ http_request.statusText);
  }
 }
}
```

See bookvol11.pamphlet for further details.

10.4 The axServer/multiServ loop

The basic call to start an Axiom browser listener is:

```
)set message autoload off
)set output mathml on
axServer(8085,multiServ)$AXSERV
```

This call sets the port, opens a socket, attaches it to a stream, and then calls "multiServ" with that stream. The "multiServ" function loops serving web responses to that port.

10.5 The)browse command

In order to make the whole process cleaner the function ")browse" handles the details. This code creates the command-line function for)browse

The browse function does the internal equivalent of the following 3 command line statments:

```
)set message autoload off
)set output mathml on
axServer(8085,multiServ)$AXSERV
```

which causes Axiom to start serving web pages on port 8085

For those unfamiliar with calling algebra from lisp there are a few points to mention.

The loadLib needs to be called to load the algebra code into the image. Normally this is automatic but we are not using the interpreter so we need to do this "by hand".

Each algebra file contains a "constructor function" which builds the domain, which is a vector, and then caches the vector so that every call to the contructor returns an EQ vector, that is, the same vector. In this case, we call the constructor |AxiomServer|

The axServer function was mangled internally to |AXSERV;axServer;IMV;2|. The multiServ function was mangled to |AXSERV;multiServ;SeV;3| Note well

that if you change axserver.spad these names might change which will generate the error message along the lines of:

```
System error:
The function $\vert$AXSERV;axServer;IMV;2$\vert$ is undefined.
```

To fix this you need to look at int/algebra/AXSERV.nrlib/code.lsp and find the new mangled function name. A better solution would be to dynamically look up the surface names in the domain vector.

Each Axiom function expects the domain vector as the last argument. This is not obvious from the call as the interpreter supplies it. We must do that "by hand".

We don't call the multiServ function. We pass it as a parameter to the axServer function. When it does get called by the SPADCALL macro it needs to be a lisp pair whose car is the function and whose cdr is the domain vector. We construct that pair here as the second argument to axServer. The third, hidden, argument to axServer is the domain vector which we supply "by hand".

The socket can be supplied on the command line but defaults to 8085. Axiom supplies the arguments as a list.

10.6 Variables Used

10.7 Functions

⟨*defun browse*⟩≡

```
(defun |browse| (socket)
 (let (axserv browser)
  (if socket
    (setq socket (car socket))
    (setq socket 8085))
  (|set| '(|mes| |auto| |off|))
  (|set| '(|out| |mathml| |on|))
  (|loadLib| '|AxiomServer|)
  (setq axserv (|AxiomServer|))
  (setq browser
   (|AXSERV;axServer;IMV;2| socket
    (cons #'|AXSERV;multiServ;SeV;3| axserv) axserv))))
```

Now we have to bolt it into Axiom. This involves two lookups.

We create the lisp pair

```
(|browse| . |development|)
```

and cons it into the $systemCommands command table. This allows the command to be executed in development mode. This lookup decides if this command is allowed. It also has the side-effect of putting the command into the $SYSCOMMANDS variable which is used to determine if the token is a command.

10.8 The server support code

Chapter 11

)cd Command

11.1 cd man page

⟨*cd.help*⟩≡

```
==========================================================================
A.4.  )cd
==========================================================================

User Level Required:  interpreter

Command Syntax:

  -  )cd directory

Command Description:

This command sets the AXIOM working current directory. The current directory
is used for looking for input files (for )read), AXIOM library source files
(for )compile), saved history environment files (for )history )restore),
compiled AXIOM library files (for )library), and files to edit (for )edit).
It is also used for writing spool files (via )spool), writing history input
files (via )history )write) and history environment files (via )history
)save),and compiled AXIOM library files (via )compile).

If issued with no argument, this command sets the AXIOM current directory to
your home directory. If an argument is used, it must be a valid directory
name. Except for the ``)'' at the beginning of the command, this has the same
syntax as the operating system cd command.

Also See:
```

-)compile
-)edit
-)history
-)library
-)read
-)spool

[1]

11.2 Variables Used

11.3 Functions

[1] "compile" (?? p ??) "edit" (18.2.1 p 146) "history" (22.4.7 p 183) "library" (24 p 221) "read" (30 p 237) "spool" (34 p 379)

Chapter 12

)clear Command

12.1 clear man page

⟨*clear.help*⟩≡

```
  ========================================================================
  A.6.  )clear
  ========================================================================

  User Level Required:  interpreter

  Command Syntax:

    - )clear all
    - )clear completely
    - )clear properties all
    - )clear properties  obj1 [obj2 ...]
    - )clear value      all
    - )clear value     obj1 [obj2 ...]
    - )clear mode       all
    - )clear mode      obj1 [obj2 ...]

  Command Description:

  This command is used to remove function and variable declarations,
  definitions and values from the workspace. To empty the entire workspace and
  reset the step counter to 1, issue

  )clear all

  To remove everything in the workspace but not reset the step counter, issue
```

```
)clear properties all

To remove everything about the object x, issue

)clear properties x

To remove everything about the objects x, y and f, issue

)clear properties x y f

The word properties may be abbreviated to the single letter ‘‘p’’.

)clear p all
)clear p x
)clear p x y f

All definitions of functions and values of variables may be removed by either

)clear value all
)clear v all

This retains whatever declarations the objects had. To remove definitions and
values for the specific objects x, y and f, issue

)clear value x y f
)clear v x y f

To remove the declarations of everything while leaving the definitions and
values, issue

)clear mode  all
)clear m all

To remove declarations for the specific objects x, y and f, issue

)clear mode x y f
)clear m x y f

The )display names and )display properties commands may be used to see what
is currently in the workspace.

The command

)clear completely
```

```
does everything that )clear all does, and also clears the internal system
function and constructor caches.

Also See:
o )display
o )history
o )undo
```

[1]

12.2 Variables Used

12.2.1 defvar $clearOptions

⟨*initvars*⟩+≡

```
(defvar |$clearOptions| '(|modes| |operations| |properties| |types| |values|))
```

12.3 Functions

12.3.1 defun clear

⟨*defun clear*⟩≡

```
(defun |clear| (l)
 (|clearSpad2Cmd| l))
```

[1] "display" (17.2.1 p 137) "history" (22.4.7 p 183) "undo" (39.3.2 p 450)

12.3.2 defun clearSpad2Cmd

TPDHERE: Note that this function also seems to parse out)except)completely and)scaches which don't seem to be documented.

⟨*defun clearSpad2Cmd*⟩≡

```
(defun |clearSpad2Cmd| (l)
  (let (|$clearExcept| opt optlist arg)
  (declare (special |$clearExcept| |$options| |$clearOptions|))
  (cond
   (|$options|
    (setq |$clearExcept|
     (prog (t0)
      (setq t0 t)
      (return
       (do ((t1 nil (null t0))
            (t2 |$options| (cdr t2))
            (t3 nil))
           ((or t1
                (atom t2)
                (progn (setq t3 (car t2)) nil)
                (progn (progn (setq opt (car t3)) t3) nil))
            t0)
        (setq t0
         (and t0
              (eq
               (|selectOptionLC| opt '(|except|) '|optionError|)
               '|except|)))))))))
  (cond
   ((null l)
    (setq optlist
     (prog (t4)
      (setq t4 nil)
      (return
       (do ((t5 |$clearOptions| (cdr t5)) (x nil))
           ((or (atom t5) (progn (setq x (car t5)) nil)) t4)
        (setq t4 (append t4 `(|%l| "       " ,x)))))))
    (|sayKeyedMsg| 's2iz0010 (list optlist)))
   (t
    (setq arg
     (|selectOptionLC| (car l) '(|all| |completely| |scaches|) nil))
    (cond
     ((eq arg '|all|)        (|clearCmdAll|))
     ((eq arg '|completely|) (|clearCmdCompletely|))
     ((eq arg '|scaches|)    (|clearCmdSortedCaches|))
     (|$clearExcept|         (|clearCmdExcept| l))
```

```
        (t
         (|clearCmdParts| l)
         (|updateCurrentInterpreterFrame|)))))))
```

12.3.3 defun clearCmdSortedCaches

⟨*defun clearCmdSortedCaches*⟩≡

```
(defun |clearCmdSortedCaches| ()
 (let (|$lookupDefaults| domain pair)
  (declare (special |$lookupDefaults| |$Void| |$ConstructorCache|))
  (do ((t0 (hget |$ConstructorCache| '|SortedCache|) (cdr t0))
       (t1 nil))
      ((or (atom t0)
           (progn
            (setq t1 (car t0))
            (setq domain (cddr t1))
            nil))
         nil)
    (setq pair (|compiledLookupCheck| '|clearCache| (list |$Void|) domain))
    (spadcall pair))))
```

12.3.4 defun clearCmdCompletely

⟨*defun clearCmdCompletely*⟩≡

```
(defun |clearCmdCompletely| ()
  (declare (special |$localExposureData| |$xdatabase| |$CatOfCatDatabase|
    |$DomOfCatDatabase| |$JoinOfCatDatabase| |$JoinOfDomDatabase|
    |$attributeDb| |$functionTable| |$existingFiles|
    |$localExposureDataDefault|))
  (|clearCmdAll|)
  (setq |$localExposureData| (copy-seq |$localExposureDataDefault|))
  (setq |$xdatabase| nil)
  (setq |$CatOfCatDatabase| nil)
  (setq |$DomOfCatDatabase| nil)
  (setq |$JoinOfCatDatabase| nil)
  (setq |$JoinOfDomDatabase| nil)
  (setq |$attributeDb| nil)
  (setq |$functionTable| nil)
  (|sayKeyedMsg| 's2iz0013 nil)
  (|clearClams|)
  (|clearConstructorCaches|)
  (setq |$existingFiles| (make-hashtable 'uequal))
  (|sayKeyedMsg| 's2iz0014 nil)
  (reclaim)
  (|sayKeyedMsg| 's2iz0015 nil))
```

12.3.5 defun clearCmdAll

⟨*defun clearCmdAll*⟩≡

```
(defun |clearCmdAll| ()
  (declare (special |$frameRecord| |$previousBindings| |$variableNumberAlist|
     |$InteractiveFrame| |$useInternalHistoryTable| |$internalHistoryTable|
     |$frameMessages| |$interpreterFrameName| |$currentLine|))
  (|clearCmdSortedCaches|)
  (setq |$frameRecord| nil)
  (setq |$previousBindings| nil)
  (setq |$variableNumberAlist| nil)
  (|untraceMapSubNames| /tracenames)
  (setq |$InteractiveFrame| (list (list nil)))
  (|resetInCoreHist|)
  (when |$useInternalHistoryTable|
    (setq |$internalHistoryTable| nil)
   (|deleteFile| (|histFileName|)))
  (setq |$IOindex| 1)
  (|updateCurrentInterpreterFrame|)
  (setq |$currentLine| ")clear all")
  (|clearMacroTable|)
  (when |$frameMessages|
    (|sayKeyedMsg| 's2iz0011 (list |$interpreterFrameName|))
    (|sayKeyedMsg| 's2iz0012 nil)))
```

12.3.6 defun clearCmdExcept

Clear all the options except the argument.

⟨*defun clearCmdExcept*⟩≡

```
(defun |clearCmdExcept| (arg)
 (let ((opt (car arg)) (vl (cdr arg)))
 (declare (special |$clearOptions|))
  (dolist (option |$clearOptions|)
   (unless (|stringPrefix?| (|object2String| opt) (|object2String| option))
    (|clearCmdParts| (cons option vl))))))
```

12.3.7 defun clearCmdParts

⟨*defun clearCmdParts*⟩≡

```
(defun |clearCmdParts| (arg)
 (let (|$e| (opt (car arg)) option pmacs imacs (vl (cdr arg)) p1 lm prop p2)
 (declare (special |$e| |$InteractiveFrame| |$clearOptions|))
  (setq option (|selectOptionLC| opt |$clearOptions| '|optionError|))
  (setq option (intern (pname option)))
  (setq option
   (case option
    (|types| '|mode|)
    (|modes| '|mode|)
    (|values| '|value|)
    (t option)))
  (if (null vl)
   (|sayKeyedMsg| 's2iz0055 nil)
   (progn
    (setq pmacs (|getParserMacroNames|))
    (setq imacs (|getInterpMacroNames|))
    (cond
     ((boot-equal vl '(|all|))
      (setq vl (assocleft (caar |$InteractiveFrame|)))
      (setq vl (remdup (append vl pmacs)))))
    (setq |$e| |$InteractiveFrame|)
    (do ((t0 vl (cdr t0)) (x nil))
        ((or (atom t0) (progn (setq x (car t0)) nil)) nil)
      (|clearDependencies| x t)
      (when (and (eq option '|properties|) (|member| x pmacs))
        (|clearParserMacro| x))
      (when (and (eq option '|properties|)
                 (|member| x imacs)
                 (null (|member| x pmacs)))
        (|sayMessage| (cons
         "   You cannot clear the definition of the system-defined macro "
          (cons (|fixObjectForPrinting| x)
                (cons (intern "." "BOOT") nil)))))
      (cond
       ((setq p1 (|assoc| x (caar |$InteractiveFrame|)))
        (cond
         ((eq option '|properties|)
          (cond
           ((|isMap| x)
            (seq
             (cond
              ((setq lm
                (|get| x '|localModemap| |$InteractiveFrame|))
```

```
            (cond
             ((pairp lm)
              (exit (|untraceMapSubNames| (cons (cadar lm) nil))))))
           (t nil)))))
      (dolist (p2 (cdr p1))
        (setq prop (car p2))
        (|recordOldValue| x prop (cdr p2))
        (|recordNewValue| x prop nil))
      (setf (caar |$InteractiveFrame|)
       (|deleteAssoc| x (caar |$InteractiveFrame|))))
     ((setq p2 (|assoc| option (cdr p1)))
      (|recordOldValue| x option (cdr p2))
      (|recordNewValue| x option nil)
      (rplacd p2 nil))))))
nil))))
```

Chapter 13

)close Command

13.1 close man page

⟨*close.help*⟩≡

```
==========================================================================
A.5.  )close
==========================================================================

User Level Required:  interpreter

Command Syntax:

  - )close
  - )close )quietly

Command Description:

This command is used to close down interpreter client processes. Such
processes are started by HyperDoc to run AXIOM examples when you click on
their text. When you have finished examining or modifying the example and you
do not want the extra window around anymore, issue

)close

to the AXIOM prompt in the window.

If you try to close down the last remaining interpreter client process, AXIOM
will offer to close down the entire AXIOM session and return you to the
operating system by displaying something like
```

```
    This is the last AXIOM session. Do you want to kill AXIOM?

  Type "y" (followed by the Return key) if this is what you had in mind. Type
  "n" (followed by the Return key) to cancel the command.

  You can use the )quietly option to force AXIOM to close down the interpreter
  client process without closing down the entire AXIOM session.

  Also See:
  o )quit
  o )pquit
```

[1]

13.2 Functions

13.2.1 defun queryClients

Returns the number of active scratchpad clients

⟨*defun queryClients*⟩≡

```
(defun |queryClients| ()
  (declare (special |$SessionManager| |$QueryClients|))
  (|sockSendInt| |$SessionManager| |$QueryClients|)
  (|sockGetInt| |$SessionManager|))
```

[1] "quit" (29.2.1 p 234) "pquit" (28.2.1 p 230)

13.2.2 defun close

⟨*defun close*⟩≡

```
(defun |close| (args)
 (declare (ignore args))
 (let (numClients opt fullopt quiet x)
 (declare (special |$SpadServer| |$SessionManager| |$CloseClient|
     |$currentFrameNum| |$options|))
  (if (null |$SpadServer|)
   (|throwKeyedMsg| 's2iz0071 nil))
   (progn
    (setq numClients (|queryClients|))
    (cond
     ((> numClients 1)
       (|sockSendInt| |$SessionManager| |$CloseClient|)
       (|sockSendInt| |$SessionManager| |$currentFrameNum|)
       (|closeInterpreterFrame| nil))
     (t
      (do ((t0 |$options| (cdr t0)) (t1 nil))
          ((or (atom t0)
               (progn (setq t1 (car t0)) nil)
               (progn (progn (setq opt (car t1)) t1) nil))
             nil)
       (setq fullopt (|selectOptionLC| opt '(|quiet|) '|optionError|))
       (unless quiet (setq quiet (eq fullopt '|quiet|))))
      (cond
       (quiet
        (|sockSendInt| |$SessionManager| |$CloseClient|)
        (|sockSendInt| |$SessionManager| |$currentFrameNum|)
        (|closeInterpreterFrame| nil))
       (t
        (setq x (upcase (|queryUserKeyedMsg| 's2iz0072 nil)))
        (when (memq (string2id-n x 1) '(yes y)) (bye)))))))))
```

Chapter 14

)compiler Command

14.1 compiler man page

⟨*compiler.help*⟩≡

```
==========================================================================
A.7.  )compile
==========================================================================

User Level Required:  compiler

Command Syntax:

  -  )compile
  -  )compile fileName
  -  )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.spad
  -  )compile directory/fileName.spad
  -  )compile fileName )new
  -  )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

Command Description:

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 AXIOM-XL compiler. It is performing double-duty, acting as a front-end to both the AXIOM-XL 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 AXIOM-XL compiler is written in C and executed by the operating system when called from within AXIOM.)

The command compiles files with file extensions .as, .ao and .al with the AXIOM-XL compiler and files with file extension .spad with the old AXIOM system compiler. It also can compile files with file extension .lsp. These are assumed to be Lisp files genererated by the AXIOM-XL compiler. 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. Otherwise, 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)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 AXIOM-XL compiler and make any necessary corrections.

We now describe the options for the new AXIOM-XL compiler.

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:

  -  Invokes the AXIOM-XL compiler and produces Lisp output.
  -  Calls the Lisp compiler if the AXIOM-XL compilation was
  successful.
  -  Use 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.as )nolibrary

The general description of AXIOM-XL command line arguments is in the AXIOM-XL
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
  AXIOM-XL 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.

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.as )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/as/quantum
```

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

)library)dir .

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 AXIOM-XL. 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, presumably, in our context, generated by AXIOM-XL when called with the -Flsp option. When)compile is used with a .lsp file, the Lisp file is compiled and)library is called. You must also have present a .asy generated from the same source file.

The following are descriptions of options for the old system compiler.

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 (see description of command)cd) 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 list of commands serves as a table of contents for the file.

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 AXIOM-XL 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 description of command)trace). 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 old system 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.

Also See:
o)abbreviation
o)edit
o)library

1

[1] “abbreviation” (?? p ??) “edit” (18.2.1 p 146) “library” (24 p 221)

14.2 Functions

14.2.1 defun compiler

⟨*defun compiler*⟩≡

```
(defun |compiler| (args)
 (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))))))))))))))))
```

Chapter 15

)copyright Command

15.1 copyright man page

⟨*copyright.help*⟩≡

```
Axiom is distributed under terms of the Modified BSD license.
Axiom was released under this license as of September 3, 2002.
Source code is freely available at:
http://savannah.nongnu.org/projects/axiom
Copyrights remain with the original copyright holders.
Use of this material is by permission and/or license.
Individual files contain reference to these applicable copyrights.
The copyright and license statements are collected here for reference.

Portions Copyright (c) 2003- The Axiom Team

The Axiom Team is the collective name for the people who have
contributed to this project. Where no other copyright statement
is noted in a file this copyright will apply.

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

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

    - Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.

    - Redistributions in binary form must reproduce the above copyright
```

CCL Public License 1.0
==========

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

(1) Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

(2) Redistributions in binary form must reproduce the above copyright
notice,
this list of conditions and the following disclaimer in the documentation
and/or other materials provided with the distribution.

(3) Neither the name of Codemist nor the names of other contributors may
be used to endorse or promote products derived from this software without
specific prior written permission.

(4) If you distribute a modified form or either source or binary code
  (a) you must make the source form of these modification available
      to Codemist;
  (b) you grant Codemist a royalty-free license to use, modify
      or redistribute your modifications without limitation;
  (c) you represent that you are legally entitled to grant these rights
      and that you are not providing Codemist with any code that violates
      any law or breaches any contract.

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

Portions Copyright (C) 1995-1997 Eric Young (eay@mincom.oz.au)
All rights reserved.

This package is an SSL implementation written
by Eric Young (eay@mincom.oz.au).

```
The licence and distribution terms for any publically available version or
derivative of this code cannot be changed.  i.e. this code cannot simply be
copied and put under another distribution licence
[including the GNU Public Licence.]

Portions Copyright (C) 1988 by Leslie Lamport.

Portions Copyright (c) 1998 Free Software Foundation, Inc.

Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, distribute with modifications, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE ABOVE COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name(s) of the above copyright
holders shall not be used in advertising or otherwise to promote the
sale, use or other dealings in this Software without prior written
authorization.

Portions Copyright 1989-2000 by Norman Ramsey.  All rights reserved.

Noweb is protected by copyright.  It is not public-domain
software or shareware, and it is not protected by a ``copyleft''
agreement like the one used by the Free Software Foundation.

Noweb is available free for any use in any field of endeavor.  You may
redistribute noweb in whole or in part provided you acknowledge its
source and include this COPYRIGHT file.  You may modify noweb and
create derived works, provided you retain this copyright notice, but
the result may not be called noweb without my written consent.

You may sell noweb if you wish.  For example, you may sell a CD-ROM
```

15.2 Functions

15.2.1 defun copyright

⟨*defun copyright*⟩≡
```
(defun |copyright| ()
 (obey (strconc "cat " (getenviron "AXIOM") "/doc/spadhelp/spadhelp.help")))
```

This command is in the list of `$noParseCommands` 7.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 7.2.1

Chapter 16

)credits Command

16.1 credits man page

16.2 Variables Used

16.3 Functions

16.3.1 defun credits

⟨*defun credits*⟩≡
```
(defun |credits| ()
 (declare (special credits))
 (mapcar #'(lambda (x) (princ x) (terpri)) credits))
```

This command is in the list of `$noParseCommands` 7.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 7.2.1

Chapter 17

)display Command

17.1 display man page

⟨*display.help*⟩≡

```
==========================================================================
A.8.  )display
==========================================================================

User Level Required:  interpreter

Command Syntax:

  -  )display all
  -  )display properties
  -  )display properties all
  -  )display properties [obj1 [obj2 ...]]
  -  )display value all
  -  )display value [obj1 [obj2 ...]]
  -  )display mode all
  -  )display mode [obj1 [obj2 ...]]
  -  )display names
  -  )display operations opName

Command Description:

This command is used to display the contents of the workspace and signatures
of functions with a given name. (A signature gives the argument and return
types of a function.)

The command
```

```
)display names

lists the names of all user-defined objects in the workspace. This is useful
if you do not wish to see everything about the objects and need only be
reminded of their names.

The commands

)display all
)display properties
)display properties all

all do the same thing: show the values and types and declared modes of all
variables in the workspace. If you have defined functions, their signatures
and definitions will also be displayed.

To show all information about a particular variable or user functions, for
example, something named d, issue

)display properties d

To just show the value (and the type) of d, issue

)display value d

To just show the declared mode of d, issue

)display mode d

All modemaps for a given operation may be displayed by using )display
operations. A modemap is a collection of information about a particular
reference to an operation. This includes the types of the arguments and the
return value, the location of the implementation and any conditions on the
types. The modemap may contain patterns. The following displays the modemaps
for the operation FromcomplexComplexCategory:

)d op complex

Also See:
o )clear
o )history
o )set
o )show
o )what
```

[1]

17.1.1 defvar $displayOptions

The current value of $displayOptions is

⟨*initvars*⟩+≡

```
(defvar |$displayOptions|
 '(|abbreviations| |all| |macros| |modes| |names| |operations|
   |properties| |types| |values|))
```

17.2 Functions

17.2.1 defun display

This trivial function satisfies the standard pattern of making a user command match the name of the function which implements the command. That command immediatly invokes a "Spad2Cmd" version.

⟨*defun display*⟩≡

```
(defun |display| (l)
 (displaySpad2Cmd l))
```

[1] "clear" (12.3.1 p 103) "history" (22.4.7 p 183) "set" (32.40.1 p 372) "show" (33 p 377) "what" (40.1.2 p 467)

17.2.2 displaySpad2Cmd

We process the options to the command and call the appropriate display function. There are really only 4 display functions. All of the other options are just subcases.

There is a slight mismatch between the $displayOptions list of symbols and the options this command accepts so we have a cond branch to clean up the option variable. This allows for the options to be plural.

If we fall all the way thru we use the $displayOptions list to construct a list of strings for the sayMessage function and tell the user what options are available.

⟨*defun displaySpad2Cmd*⟩≡

```
(defun displaySpad2Cmd (l)
 (let ((|$e| |$EmptyEnvironment|) (opt (car l)) (vl (cdr l)) option)
  (declare (special |$e| |$EmptyEnvironment| |$displayOptions|))
  (if (and (pairp l) (not (eq opt '?)))
   (progn
    (setq option (|selectOptionLC| opt |$displayOptions| '|optionError|))
    (cond
     ((eq option '|all|)
       (setq l (list '|properties|))
       (setq option '|properties|))
     ((or (eq option '|modes|) (eq option '|types|))
       (setq l (cons '|type| vl))
       (setq option '|type|))
     ((eq option '|values|)
       (setq l (cons '|value| vl))
       (setq option '|value|)))
    (cond
     ((eq option '|abbreviations|)
      (if (null vl)
        (|listConstructorAbbreviations|)
        (dolist (v vl) (|abbQuery| (|opOf| v)))))
     ((eq option '|operations|) (|displayOperations| vl))
     ((eq option '|macros|) (|displayMacros| vl))
     ((eq option '|names|) (|displayWorkspaceNames|))
     (t (|displayProperties| option l))))
   (|sayMessage|
    (append
      '("  )display keyword arguments are")
      (mapcar #'(lambda (x) (format nil "~%     ~a" x)) |$displayOptions|)
      (format nil "~%  or abbreviations thereof"))))))
```

17.2.3 defun abbQuery

⟨*defun abbQuery*⟩≡
```
(defun |abbQuery| (x)
 (let (abb)
  (cond
   ((setq abb (getdatabase x 'abbreviation))
    (|sayKeyedMsg| 's2iz0001 (list abb (getdatabase x 'constructorkind) x)))
   ((setq abb (getdatabase x 'constructor))
    (|sayKeyedMsg| 's2iz0001 (list x (getdatabase abb 'constructorkind) abb)))
   (t
     (|sayKeyedMsg| 's2iz0003 (list x))))))
```

17.2.4 defun displayOperations

This function takes a list of operation names. If the list is null we query the user to see if they want all operations printed. Otherwise we print the information for the requested symbols.

⟨*defun displayOperations*⟩≡
```
(defun |displayOperations| (l)
 (if l
  (dolist (op l) (|reportOpSymbol| op))
  (if (yesanswer)
   (dolist (op (|allOperations|)) (|reportOpSymbol| op))
   (|sayKeyedMsg| 's2iz0059 nil))))
```

17.2.5 defun yesanswer

This is a trivial function to simplify the logic of displaySpad2Cmd. If the user didn't supply an argument to the)display op command we ask if they wish to have all information about all Axiom operations displayed. If the answer is either Y or YES we return true else nil.

⟨*defun yesanswer*⟩≡
```
(defun yesanswer ()
 (memq (string2id-n (upcase (|queryUserKeyedMsg| 's2iz0058 nil)) 1) '(y yes)))
```

17.2.6 defun displayMacros

⟨*defun displayMacros*⟩≡

```
(defun |displayMacros| (names)
 (let (imacs pmacs macros first)
  (setq imacs (|getInterpMacroNames|))
  (setq pmacs (|getParserMacroNames|))
  (if names
    (setq macros names)
    (setq macros (append imacs pmacs)))
  (setq macros (remdup macros))
  (cond
   ((null macros) (|sayBrightly| "   There are no Axiom macros."))
   (t
     (setq first t)
     (do ((t0 macros (cdr t0)) (macro nil))
         ((or (atom t0) (progn (setq macro (car t0)) nil)) nil)
      (seq
       (exit
        (cond
         ((|member| macro pmacs)
           (cond
            (first (|sayBrightly|
             (cons '|%l| (cons "User-defined macros:" nil))) (setq first nil)))
           (|displayParserMacro| macro))
         ((|member| macro imacs) '|iterate|)
         (t (|sayBrightly|
          (cons "   "
           (cons '|%b|
            (cons macro
             (cons '|%d| (cons " is not a known Axiom macro." nil)))))))))))
     (setq first t)
     (do ((t1 macros (cdr t1)) (macro nil))
         ((or (atom t1) (progn (setq macro (car t1)) nil)) nil)
      (seq
       (exit
        (cond
         ((|member| macro imacs)
           (cond
            ((|member| macro pmacs) '|iterate|)
            (t
             (cond
              (first
               (|sayBrightly|
                (cons '|%l|
                 (cons "System-defined macros:" nil))) (setq first nil)))
```

```
            (|displayMacro| macro))))
        ((|member| macro pmacs) '|iterate|)))))
    nil))))
```

17.2.7 defun sayExample

This function expects 2 arguments, the documentation string and the name of the operation. It searches the documentation string for ++X lines. These lines are examples lines for functions. They look like ordinary ++ comments and fit into the ordinary comment blocks. So, for example, in the plot.spad.pamphlet file we find the following function signature:

```
plot: (F -> F,R) -> %
  ++ plot(f,a..b) plots the function \spad{f(x)}
  ++ on the interval \spad{[a,b]}.
  ++
  ++X fp:=(t:DFLOAT):DFLOAT +-> sin(t)
  ++X plot(fp,-1.0..1.0)$PLOT
```

This function splits out and prints the lines that begin with ++X.

A minor complication of printing the examples is that the lines have been processed into internal compiler format. Thus the lines that read:

```
  ++X fp:=(t:DFLOAT):DFLOAT +-> sin(t)
  ++X plot(fp,-1.0..1.0)$PLOT
```

are actually stored as one long line containing the example lines

```
"\\indented{1}{plot(\\spad{f},{}a..\\spad{b}) plots the function
 \\spad{f(x)}} \\indented{1}{on the interval \\spad{[a,{}b]}.}
 \\blankline
 \\spad{X} fp:=(t:DFLOAT):DFLOAT +-> sin(\\spad{t})
 \\spad{X} plot(\\spad{fp},{}\\spad{-1}.0..1.0)\\$PLOT"
```

So when we have an example line starting with ++X, it gets converted to the compiler to \spad{X}. So each example line is delimited by \spad{X}.

The compiler also removes the newlines so if there is a subsequent \spad{X} in the docstring then it implies multiple example lines and we loop over them, splitting them up at the delimiter.

If there is only one then we clean it up and print it.

⟨*defun sayExample*⟩≡

```
(defun sayExample (docstring)
 (let (line point)
  (when (setq point (search "spad{X}" docstring))
   (setq line (subseq docstring (+ point 8)))
   (do ((mark (search "spad{X}" line) (search "spad{X}" line)))
     ((null mark))
    (princ (cleanupLine (subseq line 0 mark)))
    (|sayNewLine|)
    (setq line (subseq line (+ mark 8))))
```

```
    (princ (cleanupLine line))
    (|sayNewLine|)
    (|sayNewLine|))))
```

17.2.8 defun cleanupLine

This function expects example lines in internal format that has been partially processed to remove the prefix. Thus we get lines that look like:

```
fp:=(t:DFLOAT):DFLOAT +-> sin(\\spad{t})
plot(\\spad{fp},{}\\spad{-1}.0..1.0)\\$PLOT
```

It removes all instances of `{}`, and `\`, and unwraps the `spad{}` call, leaving only the argument.

We return lines that look like:

```
fp:=(t:DFLOAT):DFLOAT +-> sin(t)
plot(fp,-1.0..1.0)$PLOT
```

which is hopefully exactly what the user wrote.

The compiler inserts `{}` as a space so we remove it. We remove all of the `\` characters. We remove all of the `spad{...}` delimiters which will occur around other spad variables. Technically we should search recursively for the matching delimiter rather than the next brace but the problem does not arise in practice.

⟨*defun cleanupLine*⟩≡

```
(defun cleanupLine (line)
 (do ((mark (search "{}" line) (search "{}" line)))
  ((null mark))
  (setq line
   (concatenate 'string (subseq line 0 mark) (subseq line (+ mark 2)))))
 (do ((mark (search "\\" line) (search "\\" line)))
  ((null mark))
  (setq line
   (concatenate 'string (subseq line 0 mark) (subseq line (+ mark 1)))))
 (do ((mark (search "spad{" line) (search "spad{" line)))
  ((null mark))
  (let (left point mid right)
   (setq left (subseq line 0 mark))
   (setq point (search "}" line :start2 mark))
   (setq mid (subseq line (+ mark 5) point))
   (setq right (subseq line (+ point 1)))
   (setq line (concatenate 'string left mid right))))
 line)
```

Chapter 18

)edit Command

18.1 edit man page

⟨*edit.help*⟩≡

```
==========================================================================
A.9.  )edit
==========================================================================

User Level Required:  interpreter

Command Syntax:

  - )edit [filename]

Command Description:

This command is used to edit files. It works in conjunction with the )read
and )compile commands to remember the name of the file on which you are
working. By specifying the name fully, you can edit any file you wish. Thus

)edit /u/julius/matrix.input

will place you in an editor looking at the file /u/julius/matrix.input. By
default, the editor is vi, but if you have an EDITOR shell environment
variable defined, that editor will be used. When AXIOM is running under the X
Window System, it will try to open a separate xterm running your editor if it
thinks one is necessary. For example, under the Korn shell, if you issue

export EDITOR=emacs
```

```
then the emacs editor will be used by )edit.

If you do not specify a file name, the last file you edited, read or compiled
will be used. If there is no ``last file'' you will be placed in the editor
editing an empty unnamed file.

It is possible to use the )system command to edit a file directly. For
example,

)system emacs /etc/rc.tcpip

calls emacs to edit the file.

Also See:
o )system
o )compile
o )read
```

[1]

18.2 Functions

18.2.1 defun edit

⟨*defun edit*⟩≡

```
(defun |edit| (l) (|editSpad2Cmd| l))
```

[1] "system" (37 p 385) "compile" (?? p ??) "read" (30 p 237)

18.2.2 defun editSpad2Cmd

⟨*defun editSpad2Cmd*⟩≡

```
(defun |editSpad2Cmd| (l)
 (let (olddir filetypes ll rc)
 (declare (special /editfile))
    (setq l (cond ((null l) /editfile) (t (car l))))
    (setq l (|pathname| l))
    (setq olddir (|pathnameDirectory| l))
    (setq filetypes
     (cond
      ((|pathnameType| l) (list (|pathnameType| l)))
      ((eq |$UserLevel| '|interpreter|) '("input" "INPUT" "spad" "SPAD"))
      ((eq |$UserLevel| '|compiler|) '("input" "INPUT" "spad" "SPAD"))
      (t '("input" "INPUT" "spad" "SPAD" "boot" "BOOT"
           "lisp" "LISP" "meta" "META"))))
    (setq ll
     (cond
      ((string= olddir "")
        (|pathname| ($findfile (|pathnameName| l) filetypes)))
      (t l)))
    (setq l (|pathname| ll))
    (setq /editfile l)
    (setq rc (|editFile| l))
    (|updateSourceFiles| l)
    rc))
```

Chapter 19

)fin Command

19.1 fin man page

⟨*fin.help*⟩≡

```
==========================================================================
A.10.  )fin
==========================================================================

User Level Required:  development

Command Syntax:

  -  )fin

Command Description:

This command is used by AXIOM developers to leave the AXIOM system and return
to the underlying Lisp system. To return to AXIOM, issue the ``(|spad|)''
function call to Lisp.

Also See:
o )pquit
o )quit
```

1

19.2 Functions

This command is in the list of `$noParseCommands` 7.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 7.2.1

[1] "pquit" (28.2.1 p 230) "quit" (29.2.1 p 234)

Chapter 20

)frame Command

20.1 frame man page

⟨*frame.help*⟩≡

```
==========================================================================
A.11.  )frame
==========================================================================

User Level Required:  interpreter

Command Syntax:

  - )frame  new  frameName
  - )frame  drop  [frameName]
  - )frame  next
  - )frame  last
  - )frame  names
  - )frame  import frameName [objectName1 [objectName2 ...]]
  - )set message frame on | off
  - )set message prompt frame

Command Description:

A frame can be thought of as a logical session within the physical session
that you get when you start the system. You can have as many frames as you
want, within the limits of your computer's storage, paging space, and so on.
Each frame has its own step number, environment and history. You can have a
variable named a in one frame and it will have nothing to do with anything
that might be called a in any other frame.
```

Some frames are created by the HyperDoc program and these can have pretty strange names, since they are generated automatically. To find out the names of all frames, issue

```
)frame names
```

It will indicate the name of the current frame.

You create a new frame ‘‘quark’’ by issuing

```
)frame new quark
```

The history facility can be turned on by issuing either)set history on or)history)on. If the history facility is on and you are saving history information in a file rather than in the AXIOM environment then a history file with filename quark.axh will be created as you enter commands. If you wish to go back to what you were doing in the ‘‘initial’’ frame, use

```
)frame next
```

or

```
)frame last
```

to cycle through the ring of available frames to get back to ‘‘initial’’.

If you want to throw away a frame (say ‘‘quark’’), issue

```
)frame drop quark
```

If you omit the name, the current frame is dropped.

If you do use frames with the history facility on and writing to a file, you may want to delete some of the older history files. These are directories, so you may want to issue a command like rm -r quark.axh to the operating system.

You can bring things from another frame by using)frame import. For example, to bring the f and g from the frame ‘‘quark’’ to the current frame, issue

```
)frame import quark f g
```

If you want everything from the frame ‘‘quark’’, issue

```
)frame import quark
```

You will be asked to verify that you really want everything.

```
There are two )set flags to make it easier to tell where you are.

)set message frame on | off

will print more messages about frames when it is set on. By default, it is
off.

)set message prompt frame

will give a prompt that looks like

initial (1) ->

when you start up. In this case, the frame name and step make up the prompt.

Also See:
o )history
o )set
```

1

20.2 Variables Used

The frame mechanism uses several dollar variables.

20.2.1 Primary variables

Primary variables are those which exist solely to make the frame mechanism work.

The $interpreterFrameName contains a symbol which is the name of the current frame in use.

The $interpreterFrameRing contains a list of all of the existing frames. The first frame on the list is the "current" frame. When AXIOMsys is started directly there is only one frame named "initial".

If the system is started under sman (using the axiom shell script, for example), there are two frames, "initial" and "frame0". In this case, "frame0" is the current frame. This can cause subtle problems because functions defined in the axiom initialization file (.axiom.input) will be defined in frame "initial" but the current frame will be "frame0". They will appear to be undefined. However, if the user does ")frame next" they can switch to the "initial" frame and see the functions correctly defined.

The $frameMessages variable controls when frame messages will be displayed. The variable is initially NIL. It can be set on (T) or off (NIL) using the system command:

```
)set message frame on | off
```

Setting frame messages on will output a line detailing the current frame after every output is complete.

20.2.2 Used variables

The frame collects and uses a few top level variables. These are: $Interactive-Frame, $IOindex, $HiFiAccess, $HistList, $HistListLen, $HistListAct, $HistRecord, $internalHistoryTable, and $localExposureData.

These variables can also be changed by the frame mechanism when the user requests changing to a different frame.

[1] "history" (22.4.7 p 183) "set" (32.40.1 p 372)

20.3 Data Structures

20.3.1 Frames and the Interpreter Frame Ring

Axiom has the notion of "frames". A frame is a data structure which holds all the vital data from an Axiom session. There can be multiple frames and these live in a top-level variable called $interpreterFrameRing. This variable holds a circular list of frames. The parts of a frame and their initial, default values are:

```
$interpreterFrameName      a string, named on creation
$InteractiveFrame          (list (list nil))
$IOindex                   an integer, 1
$HiFiAccess                $HiFiAccess, see the variable description
$HistList                  $HistList, see the variable description
$HistListLen               $HistListLen, see the variable description
$HistListAct               $HistListAct, see the variable description
$HistRecord                $HistRecord, see the variable description
$internalHistoryTable      nil
$localExposureData         a copy of $localExposureData
```

20.4 Accessor Functions

These could be macros but we wish to export them to the API code in the algebra so we keep them as functions.

20.4.1 0th Frame Component – frameName

20.4.2 defun frameName

⟨*defun frameName*⟩≡

```
(defun frameName (frame)
 (car frame))
```

20.4.3 1st Frame Component – frameInteractive

⟨*defun frameInteractive*⟩≡

```
(defun frameInteractive (frame)
 (nth 1 frame))
```

20.4.4 2nd Frame Component – frameIOIndex

⟨*defun frameIOIndex*⟩≡
```
(defun frameIOIndex (frame)
 (nth 2 frame))
```

20.4.5 3rd Frame Component – frameHiFiAccess

⟨*defun frameHiFiAccess*⟩≡
```
(defun frameHiFiAccess (frame)
 (nth 3 frame))
```

20.4.6 4th Frame Component – frameHistList

⟨*defun frameHistList*⟩≡
```
(defun frameHistList (frame)
 (nth 4 frame))
```

20.4.7 5th Frame Component – frameHistListLen

⟨*defun frameHistListLen*⟩≡
```
(defun frameHistListLen (frame)
 (nth 5 frame))
```

20.4.8 6th Frame Component – frameHistListAct

⟨*defun frameHistListAct*⟩≡
```
(defun frameHistListAct (frame)
 (nth 6 frame))
```

20.4.9 7th Frame Component – frameHistRecord

⟨*defun frameHistRecord*⟩≡
```
(defun frameHistRecord (frame)
 (nth 7 frame))
```

20.4.10 8th Frame Component – frameHistoryTable

⟨*defun frameHistoryTable*⟩≡
```
(defun frameHistoryTable (frame)
 (nth 8 frame))
```

20.4.11 9th Frame Component – frameExposureData

⟨*defun frameExposureData*⟩≡
```
(defun frameExposureData (frame)
 (nth 9 frame))
```

20.5 Functions

20.5.1 Initializing the Interpreter Frame Ring

Now that we know what a frame looks like we need a function to initialize the list of frames. This function sets the initial frame name to "initial" and creates a list of frames containing an empty frame. This list is the interpreter frame ring and is not actually circular but is managed as a circular list.

As a final step we update the world from this frame. This has the side-effect of resetting all the important global variables to their initial values.

⟨*defun initializeInterpreterFrameRing*⟩≡
```
(defun |initializeInterpreterFrameRing| ()
 "Initializing the Interpreter Frame Ring"
 (declare (special |$interpreterFrameName| |$interpreterFrameRing|))
 (setq |$interpreterFrameName| '|initial|)
 (setq |$interpreterFrameRing|
   (list (|emptyInterpreterFrame| |$interpreterFrameName|)))
 (|updateFromCurrentInterpreterFrame|)
 nil)
```

20.5.2 Creating a List of all of the Frame Names

This function simply walks across the frame in the frame ring and returns a list of the name of each frame.

⟨*defun frameNames*⟩≡

```
(defun |frameNames| ()
 "Creating a List of all of the Frame Names"
 (declare (special |$interpreterFrameRing|))
 (mapcar #'frameName |$interpreterFrameRing|))
```

20.5.3 Get Named Frame Environment (aka Interactive)

If the frame is found we return the environment portion of the frame otherwise we construct an empty environment and return it. The initial values of an empty frame are created here. This function returns a single frame that will be placed in the frame ring.

⟨*defun frameEnvironment*⟩≡

```
(defun |frameEnvironment| (fname)
 "Get Named Frame Environment (aka Interactive)"
 (let ((frame (|findFrameInRing| fname)))
  (if frame
   (frameInteractive frame)
   (list (list nil)))))
```

20.5.4 Create a new, empty Interpreter Frame

⟨*defun emptyInterpreterFrame*⟩≡

```
(defun |emptyInterpreterFrame| (name)
 "Create a new, empty Interpreter Frame"
 (declare (special |$HiFiAccess| |$HistList| |$HistListLen| |$HistListAct|
   |$HistRecord| |$localExposureDataDefault|))
 (list name                                  ; frame name
   (list (list nil))                         ; environment
   1                                         ; $IOindex
   |$HiFiAccess|
   |$HistList|
   |$HistListLen|
   |$HistListAct|
   |$HistRecord|
   nil                                       ; $internalHistoryTable
   (copy-seq |$localExposureDataDefault|)))  ; $localExposureData
```

20.5.5 Collecting up the Environment into a Frame

We can collect up all the current environment information into one frame element with this call. It creates a list of the current values of the global variables and returns this as a frame element.

⟨*defun createCurrentInterpreterFrame*⟩≡

```
(defun |createCurrentInterpreterFrame| ()
 "Collecting up the Environment into a Frame"
 (declare (special |$interpreterFrameName| |$InteractiveFrame| |$IOindex|
   |$HiFiAccess| |$HistList| |$HistListLen| |$HistListAct| |$HistRecord|
   |$internalHistoryTable| |$localExposureData|))
 (list
   |$interpreterFrameName|
   |$InteractiveFrame|
   |$IOindex|
   |$HiFiAccess|
   |$HistList|
   |$HistListLen|
   |$HistListAct|
   |$HistRecord|
   |$internalHistoryTable|
   |$localExposureData|))
```

20.5.6 Update from the Current Frame

The frames are kept on a circular list. The first element on that list is known as "the current frame". This will initialize all of the interesting interpreter data structures from that frame.

⟨*defun updateFromCurrentInterpreterFrame*⟩≡

```
(defun |updateFromCurrentInterpreterFrame| ()
 "Update from the Current Frame"
 (let (tmp1)
  (declare (special |$interpreterFrameRing| |$interpreterFrameName|
    |$InteractiveFrame| |$IOindex| |$HiFiAccess| |$HistList| |$HistListLen|
    |$HistListAct| |$HistRecord| |$internalHistoryTable| |$localExposureData|
    |$frameMessages|))
  (setq tmp1 (first |$interpreterFrameRing|))
  (setq |$interpreterFrameName| (nth 0 tmp1))
  (setq |$InteractiveFrame|     (nth 1 tmp1))
  (setq |$IOindex|              (nth 2 tmp1))
  (setq |$HiFiAccess|           (nth 3 tmp1))
  (setq |$HistList|             (nth 4 tmp1))
  (setq |$HistListLen|          (nth 5 tmp1))
  (setq |$HistListAct|          (nth 6 tmp1))
  (setq |$HistRecord|           (nth 7 tmp1))
  (setq |$internalHistoryTable| (nth 8 tmp1))
  (setq |$localExposureData|    (nth 9 tmp1))
  (when |$frameMessages|
   (|sayMessage|
    `("   Current interpreter frame is called"
      ,#(|bright| |$interpreterFrameName|))))))
```

20.5.7 Find a Frame in the Frame Ring by Name

Each frame contains its name as the 0th element. We simply walk all the frames and if we find one we return it.

⟨*defun findFrameInRing*⟩≡

```
(defun |findFrameInRing| (name)
 "Find a Frame in the Frame Ring by Name"
 (let (result)
  (declare (special |$interpreterFrameRing|))
  (dolist (frame |$interpreterFrameRing|)
    (when (boot-equal (frameName frame) name)
      (setq result frame)))
  result))
```

20.5.8 Update the Current Interpreter Frame

This function collects the normal contents of the world into a frame object, places it first on the frame list, and then sets the current values of the world from the frame object.

⟨*defun updateCurrentInterpreterFrame*⟩≡

```
(defun |updateCurrentInterpreterFrame| ()
  "Update the Current Interpreter Frame"
  (declare (special |$interpreterFrameRing|))
  (rplaca |$interpreterFrameRing| (|createCurrentInterpreterFrame|))
  (|updateFromCurrentInterpreterFrame|))
```

20.5.9 Move to the next Interpreter Frame in Ring

This function updates the current frame to make sure all of the current information is recorded. If there are more frame elements in the list then this will destructively move the current frame to the end of the list, that is, assume the frame list reads (1 2 3) this function will destructively change it to (2 3 1). Note: the nconc2 function destructively inserts the second list at the end of the first.

⟨*defun nextInterpreterFrame*⟩≡

```
(defun |nextInterpreterFrame| ()
  "Move to the next Interpreter Frame in Ring"
  (declare (special |$interpreterFrameRing|))
  (when (cdr |$interpreterFrameRing|)
   (setq |$interpreterFrameRing|
    (nconc2 (cdr |$interpreterFrameRing|)
      (list (car |$interpreterFrameRing|))))
   (|updateFromCurrentInterpreterFrame|)))
```

20.5.10 Change to the Named Interpreter Frame

⟨*defun changeToNamedInterpreterFrame*⟩≡

```
(defun |changeToNamedInterpreterFrame| (name)
 "Change to the Named Interpreter Frame"
 (let (frame)
 (declare (special |$interpreterFrameRing|))
   (|updateCurrentInterpreterFrame|)
   (setq frame (|findFrameInRing| name))
   (when frame
     (setq |$interpreterFrameRing|
       (cons frame (nremove |$interpreterFrameRing| frame)))
     (|updateFromCurrentInterpreterFrame|))))
```

20.5.11 Move to the previous Interpreter Frame in Ring

⟨*defun previousInterpreterFrame*⟩≡

```
(defun |previousInterpreterFrame| ()
 "Move to the previous Interpreter Frame in Ring"
 (let (tmp1 l b)
 (declare (special |$interpreterFrameRing|))
  (|updateCurrentInterpreterFrame|)
  (when  (cdr |$interpreterFrameRing|)
    (setq tmp1 (reverse |$interpreterFrameRing|))
    (setq l (car tmp1))
    (setq b (nreverse (cdr tmp1)))
    (setq |$interpreterFrameRing| (nconc2 (cons l nil) b))
    (|updateFromCurrentInterpreterFrame|))))
```

20.5.12 Add a New Interpreter Frame

⟨*defun addNewInterpreterFrame*⟩≡

```
(defun |addNewInterpreterFrame| (name)
 "Add a New Interpreter Frame"
 (declare (special |$interpreterFrameRing|))
 (if (null name)
   (|throwKeyedMsg| 's2iz0018 nil)  ; you must provide a name for new frame
   (progn
     (|updateCurrentInterpreterFrame|)
     (dolist (f |$interpreterFrameRing|)
       (when (boot-equal name (frameName f)) ; existing frame with same name
         (|throwKeyedMsg| 's2iz0019 (list name))))
     (|initHistList|)
     (setq |$interpreterFrameRing|
       (cons (|emptyInterpreterFrame| name) |$interpreterFrameRing|))
     (|updateFromCurrentInterpreterFrame|)
     ($erase (|histFileName|)))))
```

20.5.13 Close an Interpreter Frame

⟨*defun closeInterpreterFrame*⟩≡

```
(defun |closeInterpreterFrame| (name)
 "Close an Interpreter Frame"
 (declare (special |$interpreterFrameRing| |$interpreterFrameName|))
 (let (ifr found)
    (if (null (cdr |$interpreterFrameRing|))
     (if (and name (nequal name |$interpreterFrameName|))
       (|throwKeyedMsg| 's2iz0020 ; 1 frame left. not the correct name.
          (cons |$interpreterFrameName| nil))
       (|throwKeyedMsg| 's2iz0021 nil)) ; only 1 frame left, not closed
     (progn
       (if (null name)
        (setq |$interpreterFrameRing| (cdr |$interpreterFrameRing|))
        (progn
          (setq found nil)
          (setq ifr nil)
          (dolist (f |$interpreterFrameRing|)
            (if (or found (nequal name (frameName f)))
              (setq ifr (cons f ifr)))
              (setq found t))
          (if (null found)
           (|throwKeyedMsg| 's2iz0022 (cons name nil))
           (progn
             ($erase (|makeHistFileName| name))
             (setq |$interpreterFrameRing| (nreverse ifr))))))
       (|updateFromCurrentInterpreterFrame|)))))
```

20.5.14 Display the Frame Names

⟨*defun displayFrameNames*⟩≡

```
(defun |displayFrameNames| ()
 "Display the Frame Names"
 (declare (special |$interpreterFrameRing|))
 (let (t1)
  (setq t1
   (mapcar #'(lambda (f) `(|%l| "     " ,@(|bright| (frameName f))))
   |$interpreterFrameRing|))
 (|sayKeyedMsg| 's2iz0024 (list (apply #'append t1)))))
```

20.5.15 Import items from another frame

⟨*defun importFromFrame*⟩≡

```
(defun |importFromFrame| (args)
 "Import items from another frame"
 (prog (temp1 fname fenv x v props vars plist prop val m)
 (declare (special |$interpreterFrameRing|))
  (when (and args (atom args)) (setq args (cons args nil)))
  (if (null args)
   (|throwKeyedMsg| 'S2IZ0073 nil) ; missing frame name
   (progn
     (setq temp1 args)
     (setq fname (car temp1))
     (setq args (cdr temp1))
     (cond
      ((null (|member| fname (|frameNames|)))
        (|throwKeyedMsg| 'S2IZ0074 (cons fname nil))) ; not frame name
      ((boot-equal fname (frameName (car |$interpreterFrameRing|)))
        (|throwKeyedMsg| 'S2IZ0075 NIL)) ; cannot import from curr frame
      (t
        (setq fenv (|frameEnvironment| fname))
        (cond
         ((null args)
           (setq x
             (upcase (|queryUserKeyedMsg| 'S2IZ0076 (cons fname nil))))
                                        ; import everything?
           (cond
            ((memq (string2id-n x 1) '(y yes))
              (setq vars nil)
              (do ((tmp0 (caar fenv) (cdr tmp0)) (tmp1 nil))
                  ((or (atom tmp0)
                       (progn (setq tmp1 (car tmp0)) nil)
                       (progn
                        (progn
                         (setq v (car tmp1))
                         (setq props (cdr tmp1))
                         tmp1)
                        nil))
                   nil)
               (cond
                ((eq v '|--macros|)
                  (do ((tmp2 props (cdr tmp2))
                       (tmp3 nil))
                      ((or (atom tmp2)
                           (progn (setq tmp3 (car tmp2)) nil)
                           (progn
```

```
                        (progn (setq m (car tmp3)) tmp3)
                        nil))
                    nil)
                (setq vars (cons m vars))))
          (t (setq vars (cons v vars)))))
       (|importFromFrame| (cons fname vars)))
     (t
       (|sayKeyedMsg| 'S2IZ0077 (cons fname nil)))))
(t
 (do ((tmp4 args (cdr tmp4)) (v nil))
     ((or (atom tmp4) (progn (setq v (car tmp4)) nil)) nil)
  (seq
   (exit
    (progn
     (setq plist (getalist (caar fenv) v))
     (cond
      (plist
       (|clearCmdParts| (cons '|propert| (cons v nil)))
       (do ((tmp5 plist (cdr tmp5)) (tmp6 nil))
           ((or (atom tmp5)
                (progn (setq tmp6 (car tmp5)) nil)
                (progn
                 (progn
                  (setq prop (car tmp6))
                  (setq val (cdr tmp6))
                  tmp6)
                 nil))
            nil)
         (seq
          (exit (|putHist| v prop val |$InteractiveFrame|)))))
      ((setq m (|get| '|--macros--| v fenv))
        (|putHist| '|--macros--| v m |$InteractiveFrame|))
      (t
        (|sayKeyedMsg| 'S2IZ0079 ; frame not found
          (cons v (cons fname nil)))))))))
 (|sayKeyedMsg| 'S2IZ0078 ; import complete
   (cons fname nil))))))))))
```

20.5.16 The top level frame command

⟨*defun frame*⟩≡
```
(defun |frame| (l)
 "The top level frame command"
 (|frameSpad2Cmd| l))
```

20.5.17 The top level frame command handler

⟨*defun frameSpad2Cmd*⟩≡

```
(defun |frameSpad2Cmd| (args)
 "The top level frame command handler"
 (let (frameArgs arg a)
  (declare (special |$options|))
  (setq frameArgs '(|drop| |import| |last| |names| |new| |next|))
  (cond
   (|$options|
    (|throwKeyedMsg| 'S2IZ0016 ; frame command does not take options
     (cons ")frame" nil)))
   ((null args) (|helpSpad2Cmd| (cons '|frame| nil)))
   (t
     (setq arg (|selectOptionLC| (car args) frameArgs '|optionError|))
     (setq args (cdr args))
     (when (and (pairp args)
            (eq (qcdr args) nil)
            (progn (setq a (qcar args)) t))
       (setq args a))
     (when (atom args) (setq args (|object2Identifier| args)))
     (case arg
      (|drop|
        (if (and args (pairp args))
          (|throwKeyedMsg| 'S2IZ0017 ; not a valid frame name
             (cons args nil))
          (|closeInterpreterFrame| args)))
      (|import| (|importFromFrame| args))
      (|last|   (|previousInterpreterFrame|))
      (|names|  (|displayFrameNames|))
      (|new|
        (if (and args (pairp args))
          (|throwKeyedMsg| 'S2IZ0017 ; not a valid frame name
             (cons args nil))
          (|addNewInterpreterFrame| args)))
      (|next| (|nextInterpreterFrame|))
      (t nil))))))
```

20.6 Frame File Messages

⟨*Frame File Messages*⟩≡

```
S2IZ0016
 The %1b system command takes arguments but no options.
S2IZ0017
 %1b is not a valid frame name
S2IZ0018
 You must provide a name for the new frame.
S2IZ0019
 You cannot use the name %1b for a new frame because an existing
 frame already has that name.
S2IZ0020
 There is only one frame active and therefore that cannot be closed.
 Furthermore, the frame name you gave is not the name of the current frame.
 The current frame is called %1b .
S2IZ0021
 The current frame is the only active one.  Issue %b )clear all %d to
 clear its contents.
S2IZ0022
 There is no frame called %1b and so your command cannot be
 processed.
S2IZ0024
 The names of the existing frames are: %1 %l
 The current frame is the first one listed.
S2IZ0073
 %b )frame import %d must be followed by the frame name. The names
 of objects in that frame can then optionally follow the frame name.
 For example,
 %ceon %b )frame import calculus %d %ceoff
 imports all objects in the %b calculus %d frame, and
 %ceon %b )frame import calculus epsilon delta %d %ceoff
 imports the objects named %b epsilon %d and %b delta %d from the
 frame %b calculus %d .
 Please note that if the current frame contained any information
 about objects with these names, then that information would be
 cleared before the import took place.
S2IZ0074
 You cannot import anything from the frame %1b because that is not
 the name of an existing frame.
S2IZ0075
 You cannot import from the current frame (nor is there a need!).
S2IZ0076
 User verification required:
 do you really want to import everything from the frame %1b ?
 If so, please enter %b y %d or %b yes %d :
```

```
S2IZ0077
 On your request, AXIOM will not import everything from frame %1b.
S2IZ0078
 Import from frame %1b is complete. Please issue %b )display all %d
 if you wish to see the contents of the current frame.
S2IZ0079
 AXIOM cannot import %1b from frame %2b because it cannot be found.
```

Chapter 21

)help Command

21.1 help man page

⟨*help.help*⟩≡

```
==========================================================================
A.12.  )help
==========================================================================

User Level Required:  interpreter

Command Syntax:

  - )help
  - )help commandName
  - )help syntax

Command Description:

This command displays help information about system commands. If you issue

)help

then this very text will be shown. You can also give the name or abbreviation
of a system command to display information about it. For example,

)help clear

will display the description of the )clear system command.

The command
```

```
)help syntax

will give further information about the Axiom language syntax.

All this material is available in the AXIOM User Guide and in HyperDoc. In
HyperDoc, choose the Commands item from the Reference menu.

========================================================================
A.1.  Introduction
========================================================================


System commands are used to perform AXIOM environment management. Among the
commands are those that display what has been defined or computed, set up
multiple logical AXIOM environments (frames), clear definitions, read files
of expressions and commands, show what functions are available, and terminate
AXIOM.

Some commands are restricted: the commands

)set userlevel interpreter
)set userlevel compiler
)set userlevel development

set the user-access level to the three possible choices. All commands are
available at development level and the fewest are available at interpreter
level. The default user-level is interpreter. In addition to the )set command
(discussed in description of command )set ) you can use the HyperDoc settings
facility to change the user-level. Click on [Settings] here to immediately go
to the settings facility.

Each command listing begins with one or more syntax pattern descriptions plus
examples of related commands. The syntax descriptions are intended to be easy
to read and do not necessarily represent the most compact way of specifying
all possible arguments and options; the descriptions may occasionally be
redundant.

All system commands begin with a right parenthesis which should be in the
first available column of the input line (that is, immediately after the
input prompt, if any). System commands may be issued directly to AXIOM or be
included in .input files.

A system command argument is a word that directly follows the command name
and is not followed or preceded by a right parenthesis. A system command
option follows the system command and is directly preceded by a right
```

parenthesis. Options may have arguments: they directly follow the option. This example may make it easier to remember what is an option and what is an argument:

```
)syscmd arg1 arg2 )opt1 opt1arg1 opt1arg2 )opt2 opt2arg1 ...
```

In the system command descriptions, optional arguments and options are enclosed in brackets (‘‘[’’ and ‘‘]’’). If an argument or option name is in italics, it is meant to be a variable and must have some actual value substituted for it when the system command call is made. For example, the syntax pattern description

```
)read fileName []quietly]
```

would imply that you must provide an actual file name for fileName but need not use the)quietly option. Thus

```
)read matrix.input
```

is a valid instance of the above pattern.

System command names and options may be abbreviated and may be in upper or lower case. The case of actual arguments may be significant, depending on the particular situation (such as in file names). System command names and options may be abbreviated to the minimum number of starting letters so that the name or option is unique. Thus

```
)s Integer
```

is not a valid abbreviation for the)set command, because both)set and)show begin with the letter ‘‘s’’. Typically, two or three letters are sufficient for disambiguating names. In our descriptions of the commands, we have used no abbreviations for either command names or options.

In some syntax descriptions we use a vertical line ‘‘|’’ to indicate that you must specify one of the listed choices. For example, in

```
)set output fortran on | off
```

only on and off are acceptable words for following boot. We also sometimes use ‘‘...’’ to indicate that additional arguments or options of the listed form are allowed. Finally, in the syntax descriptions we may also list the syntax of related commands.

```
========================================================================
Other help topics
```

```
==========================================================================
Available help topics are:

abbreviations assignment blocks     browse     boot       cd
clear         clef       close      collection compile    display
edit          fin        for        frame      help       history
if            iterate    leave      library    lisp       load
ltrace        parallel   pquit      quit       read       repeat
savesystem    set        show       spool      suchthat   synonym
system        syntax     trace      undo       what       while

Available algebra help topics are:
```

21.2 Functions

21.2.1 The top level help command

⟨*defun help*⟩≡

```
(defun |help| (l)
 "The top level help command"
 (|helpSpad2Cmd| l))
```

21.2.2 The top level help command handler

⟨*defun helpSpad2Cmd*⟩≡

```
(defun |helpSpad2Cmd| (|args|)
 "The top level help command handler"
 (unless (|newHelpSpad2Cmd| |args|)
  (|sayKeyedMsg| 's2iz0025 (cons |args| nil))))
```

21.2.3 defun newHelpSpad2Cmd

⟨*defun newHelpSpad2Cmd*⟩≡

```
(defun |newHelpSpad2Cmd| (args)
 (let (sarg arg narg helpfile filestream line)
 (declare (special $syscommands |$useFullScreenHelp|))
  (when (null args) (setq args (list '?)))
  (if (> (|#| args) 1)
   (|sayKeyedMsg| 's2iz0026 nil)
   (progn
    (setq sarg (pname (car args)))
    (cond
     ((string= sarg "?") (setq args (list '|help|)))
     ((string= sarg "%") (setq args (list '|history|)))
     ((string= sarg "%%") (setq args (list '|history|)))
     (t nil))
    (setq arg (|selectOptionLC| (car args) $syscommands nil))
    (cond ((null arg) (setq arg (car args))))
    (cond ((eq arg '|compiler|) (setq arg '|compile|)))
    (setq narg (pname arg))
    (cond
     ((null
       (setq helpfile
        (make-input-filename
         (cons narg (cons 'helpspad (cons '* nil))))))
       nil)
     (|$useFullScreenHelp|
      (obey (strconc "$AXIOM/lib/SPADEDIT " (|namestring| helpfile))) t)
     (t
      (setq filestream (make-instream helpfile))
      (do ((line (|read-line| filestream nil) (|read-line| filestream nil)))
          ((null line) (shut filestream))
       (say line))))))
 t))
```

Chapter 22

)history Command

22.1 history man page

⟨*history.help*⟩≡

```
==========================================================================
A.13.  )history
==========================================================================

User Level Required:  interpreter

Command Syntax:

  - )history )on
  - )history )off
  - )history )write historyInputFileName
  - )history )show [n] [both]
  - )history )save savedHistoryName
  - )history )restore [savedHistoryName]
  - )history )reset
  - )history )change n
  - )history )memory
  - )history )file
  - %
  - %%(n)
  - )set history on | off

Command Description:

The history facility within AXIOM allows you to restore your environment to
that of another session and recall previous computational results. Additional
```

commands allow you to review previous input lines and to create an .input file of the lines typed to AXIOM.

AXIOM saves your input and output if the history facility is turned on (which is the default). This information is saved if either of

```
)set history on
)history )on
```

has been issued. Issuing either

```
)set history off
)history )off
```

will discontinue the recording of information.

Whether the facility is disabled or not, the value of % in AXIOM always refers to the result of the last computation. If you have not yet entered anything, % evaluates to an object of type Variable('%). The function %% may be used to refer to other previous results if the history facility is enabled. In that case, %%(n) is the output from step n if n > 0. If n < 0, the step is computed relative to the current step. Thus %%(-1) is also the previous step, %%(-2), is the step before that, and so on. If an invalid step number is given, AXIOM will signal an error.

The environment information can either be saved in a file or entirely in memory (the default). Each frame (description of command)frame) has its own history database. When it is kept in a file, some of it may also be kept in memory for efficiency. When the information is saved in a file, the name of the file is of the form FRAME.axh where ``FRAME'' is the name of the current frame. The history file is placed in the current working directory (see description of command)cd). Note that these history database files are not text files (in fact, they are directories themselves), and so are not in human-readable format.

The options to the)history command are as follows:

)change n
  will set the number of steps that are saved in memory to n. This option only has effect when the history data is maintained in a file. If you have issued)history)memory (or not changed the default) there is no need to use)history)change.

)on
  will start the recording of information. If the workspace is not empty, you will be asked to confirm this request. If you do so, the workspace

```
  will be cleared and history data will begin being saved. You can also
  turn the facility on by issuing )set history on.

)off
  will stop the recording of information. The )history )show command will
  not work after issuing this command. Note that this command may be issued
  to save time, as there is some performance penalty paid for saving the
  environment data. You can also turn the facility off by issuing )set
  history off.

)file
  indicates that history data should be saved in an external file on disk.

)memory
  indicates that all history data should be kept in memory rather than
  saved in a file. Note that if you are computing with very large objects
  it may not be practical to kept this data in memory.

)reset
  will flush the internal list of the most recent workspace calculations so
  that the data structures may be garbage collected by the underlying Lisp
  system. Like )history )change, this option only has real effect when
  history data is being saved in a file.

)restore [savedHistoryName]
  completely clears the environment and restores it to a saved session, if
  possible. The )save option below allows you to save a session to a file
  with a given name. If you had issued )history )save jacobi the command
  )history )restore jacobi would clear the current workspace and load the
  contents of the named saved session. If no saved session name is
  specified, the system looks for a file called last.axh.

)save savedHistoryName
  is used to save a snapshot of the environment in a file. This file is
  placed in the current working directory (see description of command )cd
  ). Use )history )restore to restore the environment to the state
  preserved in the file. This option also creates an input file containing
  all the lines of input since you created the workspace frame (for
  example, by starting your AXIOM session) or last did a )clear all or
  )clear completely.

)show [n] [both]
  can show previous input lines and output results. )show will display up
  to twenty of the last input lines (fewer if you haven't typed in twenty
  lines). )show n will display up to n of the last input lines. )show both
  will display up to five of the last input lines and output results. )show
```

```
    n both will display up to n of the last input lines and output results.

  )write historyInputFile
    creates an .input file with the input lines typed since the start of the
    session/frame or the last )clear all or )clear completely. If
    historyInputFileName does not contain a period (``.'') in the filename,
    .input is appended to it. For example, )history )write chaos and )history
    )write chaos.input both write the input lines to a file called
    chaos.input in your current working directory. If you issued one or more
    )undo commands, )history )write eliminates all input lines backtracked
    over as a result of )undo. You can edit this file and then use )read to
    have AXIOM process the contents.

Also See:
o )frame
o )read
o )set
o )undo
```

[1]

History recording is done in two different ways:

- all changes in variable bindings (i.e. previous values) are written to `$HistList`, which is a circular list
- all new bindings (including the binding to `%`) are written to a file called histFileName() one older session is accessible via the file `$oldHistFileName()`

22.2 Initialized history variables

The following global variables are used:

`$HistList`, `$HistListLen` and `$HistListAct` which is the actual number of "undoable" steps)

`$HistRecord` collects the input line, all variable bindings and the output of a step, before it is written to the file histFileName().

`$HiFiAccess` is a flag, which is reset by)history)off

The result of step n can be accessed by `%n`, which is translated into a call of fetchOutput(n). The updateHist is called after every interpreter step. The putHist function records all changes in the environment to `$HistList` and `$HistRecord`.

[1] "frame" (20.5.16 p 167) "read" (30 p 237) "set" (32.40.1 p 372) "undo" (39.3.2 p 450)

22.2.1 defvar $oldHistoryFileName

⟨*initvars*⟩+≡
```
(defvar |$oldHistoryFileName| '|last| "vm/370 filename name component")
```

22.2.2 defvar $historyFileType

⟨*initvars*⟩+≡
```
(defvar |$historyFileType| '|axh|     "vm/370 filename type component")
```

22.2.3 defvar $historyDirectory

⟨*initvars*⟩+≡
```
(defvar |$historyDirectory| 'A        "vm/370 filename disk component")
```

22.2.4 defvar $useInternalHistoryTable

⟨*initvars*⟩+≡
```
(defvar |$useInternalHistoryTable| t  "t means keep history in core")
```

22.3 Data Structures

22.4 Functions

22.4.1 defun makeHistFileName

⟨*defun makeHistFileName*⟩≡
```
(defun |makeHistFileName| (fname)
 (|makePathname| fname |$historyFileType| |$historyDirectory|))
```

22.4.2 defun oldHistFileName

⟨*defun oldHistFileName*⟩≡
```
(defun |oldHistFileName| ()
 (declare (special |$oldHistoryFileName|))
 (|makeHistFileName| |$oldHistoryFileName|))
```

22.4.3 defun histFileName

⟨*defun histFileName*⟩≡
```
(defun |histFileName| ()
 (declare (special |$interpreterFrameName|))
 (|makeHistFileName| |$interpreterFrameName|))
```

22.4.4 defun histInputFileName

⟨*defun histInputFileName*⟩≡
```
(defun |histInputFileName| (fn)
 (declare (special |$interpreterFrameName| |$historyDirectory|))
 (if (null fn)
   (|makePathname| |$interpreterFrameName| 'input |$historyDirectory|)
   (|makePathname| fn 'input |$historyDirectory|)))
```

22.4.5 defun initHist

⟨*defun initHist*⟩≡
```
(defun |initHist| ()
 (let (oldFile newFile)
 (declare (special |$useInternalHistoryTable| |$HiFiAccess|))
   (if |$useInternalHistoryTable|
     (|initHistList|)
     (progn
       (setq oldFile (|oldHistFileName|))
       (setq newFile (|histFileName|))
       (|histFileErase| oldFile)
       (when (make-input-filename newFile) ($replace oldFile newFile))
       (setq |$HiFiAccess| t)
       (|initHistList|)))))
```

22.4.6 defun initHistList

⟨*defun initHistList*⟩≡
```
(defun |initHistList| ()
 (let (li)
 (declare (special |$HistListLen| |$HistList| |$HistListAct| |$HistRecord|))
   (setq |$HistListLen| 20)
   (setq |$HistList| (list nil))
   (setq li |$HistList|)
   (do ((|i| 1 (qsadd1 |i|)))
       ((qsgreaterp |i| |$HistListLen|) nil)
     (setq li (cons nil li)))
   (rplacd |$HistList| li)
   (setq |$HistListAct| 0)
   (setq |$HistRecord| nil)))
```

22.4.7 The top level history command

⟨*defun history*⟩≡
```
(defun |history| (l)
 "The top level history command"
 (declare (special |$options|))
 (if (or l (null |$options|))
  (|sayKeyedMsg| 's2ih0006 nil) ; syntax error
  (|historySpad2Cmd|)))
```

22.4.8 The top level history command handler

⟨*defun historySpad2Cmd*⟩≡

```
(defun |historySpad2Cmd| ()
 "The top level history command handler"
 (let (histOptions opts opt optargs x)
 (declare (special |$options| |$HiFiAccess| |$IOindex|))
  (setq histOptions
    '(|on| |off| |yes| |no| |change| |reset| |restore| |write|
      |save| |show| |file| |memory|))
  (setq opts
   (prog (tmp1)
     (setq tmp1 nil)
     (return
      (do ((tmp2 |$options| (cdr tmp2)) (tmp3 nil))
          ((or (atom tmp2)
               (progn
                 (setq tmp3 (car tmp2))
                 nil)
               (progn
                (progn
                 (setq opt (car tmp3))
                 (setq optargs (cdr tmp3))
                 tmp3)
                nil))
             (nreverse0 tmp1))
        (setq tmp1
         (cons
           (cons
             (|selectOptionLC| opt histOptions '|optionError|)
             optargs)
           tmp1))))))
  (do ((tmp4 opts (cdr tmp4)) (tmp5 nil))
      ((or (atom tmp4)
           (progn
             (setq tmp5 (car tmp4))
             nil)
           (progn
            (progn
             (setq opt (car tmp5))
             (setq optargs (cdr tmp5))
             tmp5)
            nil))
       nil)
    (seq
     (exit
```

```
      (cond
       ((|member| opt '(|on| |yes|))
          (cond
           (|$HiFiAccess|
             (|sayKeyedMsg| 'S2IH0007 nil)) ; history already on
           ((eql |$IOindex| 1)
             (setq |$HiFiAccess| t)
             (|initHistList|)
             (|sayKeyedMsg| 'S2IH0008 nil)) ; history now on
           (t
             (setq x ; really want to turn history on?
               (upcase (|queryUserKeyedMsg| 'S2IH0009 nil)))
             (cond
              ((memq (string2id-n x 1) '(Y YES))
                (|histFileErase| (|histFileName|))
                (setq |$HiFiAccess| t)
                (setq |$options| nil)
                (|clearSpad2Cmd| '(|all|))
                (|sayKeyedMsg| 'S2IH0008 nil) ; history now on
                (|initHistList|))
              (t
                (|sayKeyedMsg| 'S2IH0010 nil)))))) ; history still off
       ((|member| opt '(|off| |no|))
         (cond
          ((null |$HiFiAccess|)
            (|sayKeyedMsg| 'S2IH0011 nil)) ; history already off
          (t
            (setq |$HiFiAccess| nil)
            (|disableHist|)
            (|sayKeyedMsg| 'S2IH0012 nil)))) ; history now off
       ((eq opt '|file|)    (|setHistoryCore| nil))
       ((eq opt '|memory|)  (|setHistoryCore| t))
       ((eq opt '|reset|)   (|resetInCoreHist|))
       ((eq opt '|save|)    (|saveHistory| optargs))
       ((eq opt '|show|)    (|showHistory| optargs))
       ((eq opt '|change|)  (|changeHistListLen| (car optargs)))
       ((eq opt '|restore|) (|restoreHistory| optargs))
       ((eq opt '|write|)   (|writeInputLines| optargs 1))))))
  '|done|))
```

22.4.9 defun setHistoryCore

We case on the inCore argument value

> If history is already on and is kept in the same location as requested (file or memory) then complain.
>
> If history is not in use then start using the file or memory as requested. This is done by simply setting the `$useInternalHistoryTable` to the requested value, where T means use memory and NIL means use a file. We tell the user.
>
> If history should be in memory, that is inCore is not NIL, and the history file already contains information we read the information from the file, store it in memory, and erase the history file. We modify `$useInternalHistoryTable` to T to indicate that we're maintining the history in memory and tell the user.
>
> Otherwise history must be on and in memory. We erase any old history file and then write the in-memory history to a new file

⟨*defun setHistoryCore*⟩≡

```
(defun |setHistoryCore| (inCore)
 (let (l vec str n rec)
 (declare (special |$useInternalHistoryTable| |$internalHistoryTable|
     |$HiFiAccess| |$IOindex|))
  (cond
   ((boot-equal inCore |$useInternalHistoryTable|)
     (if inCore
        (|sayKeyedMsg| 's2ih0030 nil) ; memory history already in use
        (|sayKeyedMsg| 's2ih0029 nil))) ; file history already in use
   ((null |$HiFiAccess|)
     (setq |$useInternalHistoryTable| inCore)
     (if inCore
        (|sayKeyedMsg| 's2ih0032 nil) ; use memory history
        (|sayKeyedMsg| 's2ih0031 nil))) ; use file history
   (inCore
     (setq |$internalHistoryTable| nil)
     (cond
      ((nequal |$IOindex| 0)
        (setq l (length (rkeyids (|histFileName|))))
        (do ((|i| 1 (qsadd1 |i|)))
            ((qsgreaterp |i| l) nil)
          (setq vec (unwind-protect (|readHiFi| |i|) (|disableHist|)))
          (setq |$internalHistoryTable|
            (cons (cons |i| vec) |$internalHistoryTable|)))
        (|histFileErase| (|histFileName|))))
     (setq |$useInternalHistoryTable| t)
```

```
    (|sayKeyedMsg| 'S2IH0032 nil)) ; use memory history
  (t
    (setq |$HiFiAccess| nil)
    (|histFileErase| (|histFileName|))
    (setq str
     (rdefiostream
      (cons
        '(mode . output)
        (cons
          (cons 'file (|histFileName|))
           nil))))
    (do ((tmp0 (reverse |$internalHistoryTable|) (cdr tmp0))
         (tmp1 nil))
        ((or (atom tmp0)
             (progn
               (setq tmp1 (car tmp0))
               nil)
             (progn
              (progn
               (setq n (car tmp1))
               (setq rec (cdr tmp1))
               tmp1)
              nil))
             nil)
      (spadrwrite (|object2Identifier| n) rec str))
    (rshut str)
    (setq |$HiFiAccess| t)
    (setq |$internalHistoryTable| nil)
    (setq |$useInternalHistoryTable| nil)
    (|sayKeyedMsg| 's2ih0031 nil))))) ; use file history
```

22.4.10 defvar $underbar

Also used in the output routines.

⟨*initvars*⟩+≡

```
(defvar underbar "_")
```

22.4.11 defun writeInputLines

⟨*defun writeInputLines*⟩≡

```
(defun |writeInputLines| (fn initial)
 (let (maxn breakChars vecl k svec done n lineList file inp)
 (declare (special underbar |$HiFiAccess| |$IOindex|))
  (cond
   ((null |$HiFiAccess|) (|sayKeyedMsg| 's2ih0013 nil)) ; history is not on
   ((null fn) (|throwKeyedMsg| 's2ih0038 nil)) ; missing file name
   (t
     (setq maxn 72)
     (setq breakChars (cons '| | (cons '+ nil)))
     (do ((tmp0 (spaddifference |$IOindex| 1))
          (|i| initial (+ |i| 1)))
         ((> |i| tmp0) nil)
       (setq vecl (car (|readHiFi| |i|)))
       (when (stringp vecl) (setq vecl (cons vecl nil)))
       (dolist (vec vecl)
         (setq n (size vec))
         (do ()
             ((null (> n maxn)) nil)
             (setq done nil)
             (do ((|j| 1 (qsadd1 |j|)))
                 ((or (qsgreaterp |j| maxn) (null (null done))) nil)
               (setq k (spaddifference (plus 1 maxn) |j|))
               (when (memq (elt vec k) breakChars)
                 (setq svec (strconc (substring vec 0 (1+ k)) underbar))
                 (setq lineList (cons svec lineList))
                 (setq done t)
                 (setq vec (substring vec (1+ k) nil))
                 (setq n (size vec))))
             (when done (setq n 0)))
         (setq lineList (cons vec lineList))))
     (setq file (|histInputFileName| fn))
     (|histFileErase| file)
     (setq inp
      (defiostream
       (cons
        '(mode . output)
        (cons (cons 'file file) nil)) 255 0))
     (dolist (x (|removeUndoLines| (nreverse lineList)))
       (write-line x inp))
     (cond
      ((nequal fn '|redo|)
        (|sayKeyedMsg| 's2ih0014 ; edit this file to see input lines
          (list (|namestring| file)))))
```

```
      (shut inp)
      nil))))
```

22.4.12 defun resetInCoreHist

⟨*defun resetInCoreHist*⟩≡

```
(defun |resetInCoreHist| ()
 (declare (special |$HistListAct| |$HistListLen| |$HistList|))
 (setq |$HistListAct| 0)
 (do ((|i| 1 (qsadd1 |i|)))
     ((qsgreaterp |i| |$HistListLen|) nil)
   (setq |$HistList| (cdr |$HistList|))
   (rplaca |$HistList| nil)))
```

22.4.13 defun changeHistListLen

⟨*defun changeHistListLen*⟩≡

```
(defun |changeHistListLen| (n)
 (let (dif l)
 (declare (special |$HistListLen| |$HistList| |$HistListAct|))
 (if (null (integerp n))
   (|sayKeyedMsg| 's2ih0015 (list n)) ; only positive integers
   (progn
     (setq dif (spaddifference n |$HistListLen|))
     (setq |$HistListLen| n)
     (setq l (cdr |$HistList|))
     (cond
      ((> dif 0)
        (do ((|i| 1 (qsadd1 |i|)))
            ((qsgreaterp |i| dif) nil)
           (setq l (cons nil l))))
      ((minusp dif)
        (do ((tmp0 (spaddifference dif))
             (|i| 1 (qsadd1 |i|)))
            ((qsgreaterp |i| tmp0) nil)
           (setq l (cdr l)))
        (cond
         ((> |$HistListAct| n) (setq |$HistListAct| n))
         (t nil))))
     (rplacd |$HistList| l)
     '|done|))))
```

22.4.14 defun updateHist

⟨*defun updateHist*⟩≡
```
(defun |updateHist| ()
 (declare (special |$IOindex| |$HiFiAccess| |$HistRecord| |$mkTestInputStack|
   |$currentLine|))
 (when |$IOindex|
   (|startTimingProcess| '|history|)
   (|updateInCoreHist|)
   (when |$HiFiAccess|
      (unwind-protect (|writeHiFi|) (|disableHist|))
      (setq |$HistRecord| nil))
   (incf |$IOindex|)
   (|updateCurrentInterpreterFrame|)
   (setq |$mkTestInputStack| nil)
   (setq |$currentLine| nil)
   (|stopTimingProcess| '|history|)))
```

22.4.15 defun updateInCoreHist

⟨*defun updateInCoreHist*⟩≡
```
(defun |updateInCoreHist| ()
 (declare (special |$HistList| |$HistListLen| |$HistListAct|))
  (setq |$HistList| (cdr |$HistList|))
  (rplaca |$HistList| nil)
  (when (> |$HistListLen| |$HistListAct|)
    (setq |$HistListAct| (plus |$HistListAct| 1))))
```

22.4.16 defun putHist

⟨*defun putHist*⟩≡
```
(defun |putHist| (x prop val e)
  (declare (special |$HiFiAccess|))
  (when (null (eq x '%)) (|recordOldValue| x prop (|get| x prop e)))
  (when |$HiFiAccess| (|recordNewValue| x prop val))
  (|putIntSymTab| x prop val e))
```

22.4.17 defun recordNewValue

⟨*defun recordNewValue*⟩≡
```
(defun |recordNewValue| (x prop val)
  (|startTimingProcess| '|history|)
  (|recordNewValue0| x prop val)
  (|stopTimingProcess| '|history|))
```

22.4.18 defun recordNewValue0

⟨*defun recordNewValue0*⟩≡
```
(defun |recordNewValue0| (x prop val)
 (let (p1 p2 p)
 (declare (special |$HistRecord|))
   (if (setq p1 (assq x |$HistRecord|))
    (if (setq p2 (assq prop (cdr p1)))
      (rplacd p2 val)
      (rplacd p1 (cons (cons prop val) (cdr p1))))
    (progn
      (setq p (cons x (list (cons prop val))))
      (setq |$HistRecord| (cons p |$HistRecord|))))))
```

22.4.19 defun recordOldValue

⟨*defun recordOldValue*⟩≡
```
(defun |recordOldValue| (x prop val)
  (|startTimingProcess| '|history|)
  (|recordOldValue0| x prop val)
  (|stopTimingProcess| '|history|))
```

22.4.20 defun recordOldValue0

⟨*defun recordOldValue0*⟩≡
```
(defun |recordOldValue0| (x prop val)
 (let (p1 p)
 (declare (special |$HistList|))
 (when (setq p1 (assq x (car |$HistList|)))
   (when (null (assq prop (cdr p1)))
      (rplacd p1 (cons (cons prop val) (cdr p1)))))
 (setq p (cons x (list (cons prop val))))
 (rplaca |$HistList| (cons p (car |$HistList|)))))
```

22.4.21 defun undoInCore

⟨*defun undoInCore*⟩≡
```
(defun |undoInCore| (n)
 (let (li vec p p1 val)
 (declare (special |$HistList| |$HistListLen| |$IOindex| |$HiFiAccess|
    |$InteractiveFrame|))
  (setq li |$HistList|)
  (do ((i n (+ i 1)))
      ((> i |$HistListLen|) nil)
   (setq li (cdr li)))
  (|undoChanges| li)
  (setq n (spaddifference (spaddifference |$IOindex| n) 1))
  (and
    (> n 0)
    (if |$HiFiAccess|
     (progn
       (setq vec (cdr (unwind-protect (|readHiFi| n) (|disableHist|))))
       (setq val
        (and
         (setq p (assq '% vec))
         (setq p1 (assq '|value| (cdr p)))
         (cdr p1))))
      (|sayKeyedMsg| 's2ih0019 (cons n nil)))) ; no history file
  (setq |$InteractiveFrame| (|putHist| '% '|value| val |$InteractiveFrame|))
  (|updateHist|)))
```

22.4.22 defun undoChanges

⟨*defun undoChanges*⟩≡

```
(defun |undoChanges| (li)
 (let (x)
 (declare (special |$HistList| |$InteractiveFrame|))
  (when (null (boot-equal (cdr li) |$HistList|)) (|undoChanges| (cdr li)))
  (dolist (p1 (car li))
    (setq x (car p1))
    (dolist (p2 (cdr p1))
     (|putHist| x (car p2) (cdr p2) |$InteractiveFrame|)))))
```

22.4.23 defun undoFromFile

⟨*defun undoFromFile*⟩≡

```
(defun |undoFromFile| (n)
 (let (varl prop vec x p p1 val)
 (declare (special |$InteractiveFrame| |$HiFiAccess|))
  (do ((tmp0 (caar |$InteractiveFrame|) (cdr tmp0)) (tmp1 nil))
      ((or (atom tmp0)
           (progn (setq tmp1 (car tmp0)) nil)
           (progn
             (progn
              (setq x (car tmp1))
              (setq varl (cdr tmp1))
              tmp1)
              nil))
            nil)
   (seq
    (exit
     (do ((tmp2 varl (cdr tmp2)) (p nil))
         ((or (atom tmp2) (progn (setq p (car tmp2)) nil)) nil)
      (seq
       (exit
        (progn
         (setq prop (car p))
         (setq val (cdr p))
         (when val
           (progn
             (when (null (eq x '%))
               (|recordOldValue| x prop val))
             (when |$HiFiAccess|
               (|recordNewValue| x prop val))
             (rplacd p nil)))))))))
  (do ((|i| 1 (qsadd1 |i|)))
      ((qsgreaterp |i| n) nil)
    (setq vec
      (unwind-protect (cdr (|readHiFi| |i|)) (|disableHist|)))
    (do ((tmp3 vec (cdr tmp3)) (p1 nil))
        ((or (atom tmp3) (progn (setq p1 (car tmp3)) nil)) nil)
     (setq x (car p1))
     (do ((tmp4 (cdr p1) (cdr tmp4)) (p2 nil))
         ((or (atom tmp4) (progn (setq p2 (car tmp4)) nil)) nil)
      (setq |$InteractiveFrame|
         (|putHist| x (car p2) (CDR p2) |$InteractiveFrame|)))))
  (setq val
   (and
    (setq p (assq '% vec))
```

```
    (setq p1 (assq '|value| (cdr p)))
    (cdr p1)))
  (setq |$InteractiveFrame| (|putHist| '% '|value| val |$InteractiveFrame|))
  (|updateHist|)))
```

22.4.24 defun saveHistory

⟨*defun saveHistory*⟩≡

```
(defun |saveHistory| (fn)
 (let (|$seen| savefile inputfile saveStr n rec val)
 (declare (special |$seen| |$HiFiAccess| |$useInternalHistoryTable|
     |$internalHistoryTable|))
 (setq |$seen| (make-hashtable 'eq))
 (cond
  ((null |$HiFiAccess|)
    (|sayKeyedMsg| 's2ih0016 nil)) ; the history file is not on
  ((and (null |$useInternalHistoryTable|)
        (null (make-input-filename (|histFileName|))))
    (|sayKeyedMsg| 's2ih0022 nil)) ; no history saved yet
  ((null fn)
    (|throwKeyedMsg| 's2ih0037 nil)) ; need to specify a history filename
  (t
    (setq savefile (|makeHistFileName| fn))
    (setq inputfile (|histInputFileName| fn))
    (|writeInputLines| fn 1)
    (|histFileErase| savefile)
    (when |$useInternalHistoryTable|
      (setq saveStr
        (rdefiostream
         (cons '(mode . output)
           (cons (cons 'file savefile) nil))))
      (do ((tmp0 (reverse |$internalHistoryTable|) (cdr tmp0))
           (tmp1 nil))
          ((or (atom tmp0)
               (progn (setq tmp1 (car tmp0)) nil)
               (progn
                (progn
                 (setq n (car tmp1))
                 (setq rec (cdr tmp1))
                 tmp1)
                nil))
               nil)
        (setq val (spadrwrite0 (|object2Identifier| n) rec saveStr))
        (when (eq val '|writifyFailed|)
         (|sayKeyedMsg| 's2ih0035 ; can't save the value of step
          (list n inputfile))))
      (rshut saveStr))
    (|sayKeyedMsg| 's2ih0018 ; saved history file is
      (cons (|namestring| savefile) nil))
    nil))))
```

22.4.25 defun restoreHistory

⟨*defun restoreHistory*⟩≡

```
(defun |restoreHistory| (fn)
 (let (|$options| fnq restfile curfile l oldInternal vec line x a)
 (declare (special |$options| |$internalHistoryTable| |$HiFiAccess| |$e|
     |$useInternalHistoryTable| |$InteractiveFrame| |$oldHistoryFileName|))
 (cond
   ((null fn) (setq fnq |$oldHistoryFileName|))
   ((and (pairp fn)
         (eq (qcdr fn) nil)
         (progn
          (setq fnq (qcar fn))
          t)
         (identp fnq))
     (setq fnq fnq))
   (t (|throwKeyedMsg| 's2ih0023 (cons fnq nil)))) ; invalid filename
 (setq restfile (|makeHistFileName| fnq))
 (if (null (make-input-filename restfile))
  (|sayKeyedMsg| 's2ih0024  ; file does not exist
       (cons (|namestring| restfile) nil))
  (progn
    (setq |$options| nil)
    (|clearSpad2Cmd| '(|all|))
    (setq curfile (|histFileName|))
    (|histFileErase| curfile)
    ($fcopy restfile curfile)
    (setq l (length (rkeyids curfile)))
    (setq |$HiFiAccess| t)
    (setq oldInternal |$useInternalHistoryTable|)
    (setq |$useInternalHistoryTable| nil)
    (when oldInternal (setq |$internalHistoryTable| nil))
    (do ((|i| 1 (qsadd1 |i|)))
        ((qsgreaterp |i| l) nil)
      (setq vec (unwind-protect (|readHiFi| |i|) (|disableHist|)))
      (when oldInternal
        (setq |$internalHistoryTable|
         (cons (cons |i| vec) |$internalHistoryTable|)))
      (setq line (car vec))
      (dolist (p1 (cdr vec))
        (setq x (car p1))
        (do ((tmp1 (cdr p1) (cdr tmp1)) (p2 nil))
            ((or (atom tmp1) (progn (setq p2 (car tmp1)) nil)) nil)
         (setq |$InteractiveFrame|
           (|putHist| x
            (car p2) (cdr p2) |$InteractiveFrame|))))
```

```
  (|updateInCoreHist|))
(setq |$e| |$InteractiveFrame|)
(do ((tmp2 (caar |$InteractiveFrame|) (cdr tmp2)) (tmp3 nil))
    ((or (atom tmp2)
         (progn
           (setq tmp3 (car tmp2))
           nil)
         (progn
          (progn
            (setq a (car tmp3))
            tmp3)
            nil))
        nil)
   (when (|get| a '|localModemap| |$InteractiveFrame|)
      (|rempropI| a '|localModemap|)
      (|rempropI| a '|localVars|)
      (|rempropI| a '|mapBody|)))
(setq |$IOindex| (plus l 1))
(setq |$useInternalHistoryTable| oldInternal)
(|sayKeyedMsg| 'S2IH0025 ; workspace restored
  (cons (|namestring| restfile) nil))
(|clearCmdSortedCaches|)
 nil))))
```

22.4.26 defun setIOindex

⟨*defun setIOindex*⟩≡

```
(defun |setIOindex| (n)
  (declare (special |$IOindex|))
  (setq |$IOindex| n))
```

22.4.27 defun showInput

⟨*defun showInput*⟩≡
```
(defun |showInput| (mini maxi)
 (let (vec l)
  (do ((|ind| mini (+ |ind| 1)))
      ((> |ind| maxi) nil)
    (setq vec (unwind-protect (|readHiFi| |ind|) (|disableHist|)))
    (cond
     ((> 10 |ind|) (tab 2))
     ((> 100 |ind|) (tab 1))
     (t nil))
    (setq l (car vec))
    (if (stringp l)
      (|sayMSG| (list "   [" |ind| "] " (car vec)))
      (progn
       (|sayMSG| (list "   [" |ind| "] "))
       (do ((tmp0 l (cdr tmp0)) (ln nil))
           ((or (atom tmp0) (progn (setq ln (car tmp0)) nil)) nil)
         (|sayMSG| (list "      " ln))))))))
```

22.4.28 defun showInOut

⟨*defun showInOut*⟩≡
```
(defun |showInOut| (mini maxi)
 (let (vec Alist triple)
  (do ((ind mini (+ ind 1)))
      ((> ind maxi) nil)
   (setq vec (unwind-protect (|readHiFi| ind) (|disableHist|)))
   (|sayMSG| (cons (car vec) nil))
   (cond
    ((setq Alist (assq '% (cdr vec)))
      (setq triple (cdr (assq '|value| (cdr Alist))))
      (setq |$IOindex| ind)
      (|spadPrint| (|objValUnwrap| triple) (|objMode| triple)))))))
```

22.4.29 defun fetchOutput

⟨*defun fetchOutput*⟩≡

```
(defun |fetchOutput| (n)
 (let (vec Alist val)
  (cond
   ((and (boot-equal n (spaddifference 1)) (setq val (|getI| '% '|value|)))
     val)
   (|$HiFiAccess|
    (setq n
     (cond
      ((minusp n) (plus |$IOindex| n))
      (t n)))
    (cond
     ((>= n |$IOindex|)
       (|throwKeyedMsg| 'S2IH0001 (cons n nil))) ; no step n yet
     ((> 1 n)
       (|throwKeyedMsg| 's2ih0002 (cons n nil))) ; only nonzero steps
     (t
       (setq vec (unwind-protect (|readHiFi| n) (|disableHist|)))
       (cond
        ((setq Alist (assq '% (cdr vec)))
          (cond
            ((setq val (cdr (assq '|value| (cdr Alist))))
              val)
            (t
              (|throwKeyedMsg| 's2ih0003 (cons n nil))))) ; no step value
        (t (|throwKeyedMsg| 's2ih0003 (cons n nil))))))) ; no step value
   (t (|throwKeyedMsg| 's2ih0004 nil))))) ; history not on
```

22.4.30 Read the history file using index n

⟨*defun readHiFi*⟩≡

```
(defun |readHiFi| (n)
 "Read the history file using index n"
 (let (pair HiFi vec)
  (declare (special |$useInternalHistoryTable| |$internalHistoryTable|))
  (if |$useInternalHistoryTable|
   (progn
    (setq pair (|assoc| n |$internalHistoryTable|))
    (if (atom pair)
     (|keyedSystemError| 's2ih0034 nil) ; missing element
     (setq vec (qcdr pair))))
   (progn
     (setq HiFi
      (rdefiostream
       (cons
        '(mode . input)
         (cons
          (cons 'file (|histFileName|)) nil))))
     (setq vec (spadrread (|object2Identifier| n) HiFi))
     (rshut HiFi)))
  vec))
```

22.4.31 Writes information of the current step to history file

⟨*defun writeHiFi*⟩≡
```
(defun |writeHiFi| ()
 "Writes information of the current step to history file"
 (let (HiFi)
 (declare (special |$useInternalHistoryTable| |$internalHistoryTable|
     |$IOindex| |$HistRecord| |$currentLine|))
   (if |$useInternalHistoryTable|
     (setq |$internalHistoryTable|
      (cons
       (cons |$IOindex|
        (cons |$currentLine| |$HistRecord|))
        |$internalHistoryTable|))
    (progn
      (setq HiFi
        (rdefiostream
         (cons
           '(mode . output)
            (cons (cons 'file (|histFileName|)) nil))))
      (spadrwrite (|object2Identifier| |$IOindex|)
        (cons |$currentLine| |$HistRecord|) HiFi)
      (rshut HiFi)))))
```

22.4.32 Disable history if an error occurred

⟨*defun disableHist*⟩≡
```
(defun |disableHist| ()
 "Disable history if an error occurred"
 (declare (special |$HiFiAccess|))
 (cond
  ((null |$HiFiAccess|)
    (|histFileErase| (|histFileName|)))
  (t nil)))
```

22.4.33 defun writeHistModesAndValues

⟨*defun writeHistModesAndValues*⟩≡
```
(defun |writeHistModesAndValues| ()
 (let (a x)
 (declare (special |$InteractiveFrame|))
  (do ((tmp0 (caar |$InteractiveFrame|) (cdr tmp0)) (tmp1 nil))
      ((or (atom tmp0)
           (progn
             (setq tmp1 (car tmp0))
             nil)
           (progn
             (progn
               (setq a (car tmp1))
               tmp1)
           nil))
       nil)
      (cond
       ((setq x (|get| a '|value| |$InteractiveFrame|))
         (|putHist| a '|value| x |$InteractiveFrame|))
       ((setq x (|get| a '|mode| |$InteractiveFrame|))
         (|putHist| a '|mode| x |$InteractiveFrame|))))))
```

22.5 Lisplib output transformations

Lisplib output transformations

Some types of objects cannot be saved by LISP/VM in lisplibs. These functions transform an object to a writable form and back.

22.5.1 defun spadrwrite0

⟨*defun spadrwrite0*⟩≡
```
(defun spadrwrite0 (vec item stream)
 (let (val)
  (setq val (|safeWritify| item))
  (if (eq val '|writifyFailed|)
    val
    (progn
     (|rwrite| vec val stream)
     item))))
```

22.5.2 defun spadrwrite

⟨*defun spadrwrite*⟩≡
```
(defun spadrwrite (vec item stream)
 (let (val)
  (setq val (spadrwrite0 vec item stream))
  (if (eq val '|writifyFailed|)
   (|throwKeyedMsg| 's2ih0036 nil) ; cannot save value to file
   item)))
```

22.5.3 defun spadrread

⟨*defun spadrread*⟩≡
```
(defun spadrread (vec stream)
 (|dewritify| (|rread| vec stream nil)))
```

22.5.4 defun unwritable?

⟨*defun unwritable?*⟩≡
```
(defun |unwritable?| (ob)
 (cond
  ((or (pairp ob) (vecp ob)) nil)
  ((or (compiled-function-p ob) (hashtablep ob)) t)
  ((or (placep ob) (readtablep ob)) t)
  ((floatp ob) t)
  (t nil)))
```

22.5.5 defun writifyComplain

Create a full isomorphic object which can be saved in a lisplib. Note that `dewritify(writify(x))` preserves UEQUALity of hashtables. HASHTABLEs go both ways. READTABLEs cannot presently be transformed back.

⟨*defun writifyComplain*⟩≡
```
(defun |writifyComplain| (s)
 (declare (special |$writifyComplained|))
 (unless |$writifyComplained|
  (setq |$writifyComplained| t)
  (|sayKeyedMsg| 's2ih0027 (list s)))) ; cannot save value
```

22.5.6 defun safeWritify

⟨*defun safeWritify*⟩≡

```
(defun |safeWritify| (ob)
 (catch '|writifyTag| (|writify| ob)))
```

22.5.7 defun writify,writifyInner

⟨*defun writify,writifyInner*⟩≡

```
(defun |writify,writifyInner| (ob)
 (prog (e name tmp1 tmp2 tmp3 x qcar qcdr d n keys nob)
  (declare (special |$seen| |$NonNullStream| |$NullStream|))
  (return
   (seq
    (when (null ob) (exit nil))
    (when (setq e (hget |$seen| ob)) (exit e))
    (when (pairp ob)
     (exit
      (seq
       (setq qcar (qcar ob))
       (setq qcdr (qcdr ob))
       (when (setq name (|spadClosure?| ob))
        (exit
         (seq
          (setq d (|writify,writifyInner| (qcdr ob)))
          (setq nob
           (cons 'writified!!
            (cons 'spadclosure
             (cons d (cons name nil)))))
          (hput |$seen| ob nob)
          (hput |$seen| nob nob)
          (exit nob))))
       (when
        (and
         (and (pairp ob)
              (eq (qcar ob) 'lambda-closure)
              (progn
               (setq tmp1 (qcdr ob))
               (and (pairp tmp1)
                    (progn
                     (setq tmp2 (qcdr tmp1))
                     (and
                      (pairp tmp2)
                      (progn
                       (setq tmp3 (qcdr tmp2))
                       (and (pairp tmp3)
                            (progn
                             (setq x (qcar tmp3))
                             t))))))))) x)
        (exit
         (throw '|writifyTag| '|writifyFailed|)))
       (setq nob (cons qcar qcdr))
```

```
    (hput |$seen| ob nob)
    (hput |$seen| nob nob)
    (setq qcar (|writify,writifyInner| qcar))
    (setq qcdr (|writify,writifyInner| qcdr))
    (qrplaca nob qcar)
    (qrplacd nob qcdr)
    (exit nob))))
(when (vecp ob)
 (exit
  (seq
   (when (|isDomainOrPackage| ob)
     (setq d (|mkEvalable| (|devaluate| ob)))
     (setq nob (list 'writified!! 'devaluated (|writify,writifyInner| d)))
     (hput |$seen| ob nob)
     (hput |$seen| nob nob)
     (exit nob))
   (setq n (qvmaxindex ob))
   (setq nob (make-vec (plus n 1)))
   (hput |$seen| ob nob)
   (hput |$seen| nob nob)
   (do ((|i| 0 (qsadd1 |i|)))
       ((qsgreaterp |i| n) nil)
     (qsetvelt nob |i| (|writify,writifyInner| (qvelt ob |i|))))
   (exit nob))))
(when (eq ob 'writified!!)
 (exit
  (cons 'writified!! (cons 'self nil))))
(when (|constructor?| ob)
 (exit ob))
(when (compiled-function-p ob)
 (exit
  (throw '|writifyTag| '|writifyFailed|)))
(when (hashtablep ob)
   (setq nob (cons 'writified!! nil))
   (hput |$seen| ob nob)
   (hput |$seen| nob nob)
   (setq keys (hkeys ob))
   (qrplacd nob
    (cons
      'hashtable
      (cons
        (hashtable-class ob)
        (cons
          (|writify,writifyInner| keys)
          (cons
            (prog (tmp0)
```

```
                (setq tmp0 nil)
                (return
                 (do ((tmp1 keys (cdr tmp1)) (k nil))
                     ((or (atom tmp1)
                          (progn
                           (setq k (car tmp1))
                           nil))
                          (nreverse0 tmp0))
                   (setq tmp0
                    (cons (|writify,writifyInner| (hget ob k)) tmp0)))))
              nil)))))
   (exit nob))
(when (placep ob)
   (setq nob (cons 'writified!! (cons 'place nil)))
   (hput |$seen| ob nob)
   (hput |$seen| nob nob)
   (exit nob))
(when (readtablep ob)
 (exit
  (throw '|writifyTag| '|writifyFailed|)))
(when (stringp ob)
 (exit
  (seq
   (when (eq ob |$NullStream|)
    (exit
     (cons 'writified!! (cons 'nullstream nil))))
   (when (eq ob |$NonNullStream|)
    (exit
     (cons 'writified!! (cons 'nonnullstream nil))))
   (exit ob))))
(when (floatp ob)
 (exit
  (seq
   (when (boot-equal ob (read-from-string (stringimage ob)))
    (exit ob))
   (exit
    (cons 'writified!!
     (cons 'float
      (cons ob
       (multiple-value-list (integer-decode-float ob)))))))))
(exit ob)))))
```

22.5.8 defun writify

⟨*defun writify*⟩≡
```
(defun |writify| (ob)
 (let (|$seen| |$writifyComplained|)
 (declare (special |$seen| |$writifyComplained|))
  (if (null (|ScanOrPairVec| (|function| |unwritable?|) ob))
    ob
    (progn
     (setq |$seen| (make-hashtable 'eq))
     (setq |$writifyComplained| nil)
     (|writify,writifyInner| ob)))))
```

22.5.9 defun spadClosure?

⟨*defun spadClosure?*⟩≡
```
(defun |spadClosure?| (ob)
 (let (fun name vec)
  (setq fun (qcar ob))
  (if (null (setq name (bpiname fun)))
   nil
   (progn
     (setq vec (qcdr ob))
     (if (null (vecp vec))
      nil
      name)))))
```

22.5.10 defun dewritify,is?

⟨*defun dewritify,is?*⟩≡
```
(defun |dewritify,is?| (a)
 (eq a 'writified!!))
```

22.5.11 defun dewritify,dewritifyInner

⟨*defun dewritify,dewritifyInner*⟩≡

```
(defun |dewritify,dewritifyInner| (ob)
 (prog (e type oname f vec name tmp1 signif expon sign fval qcar qcdr n nob)
  (declare (special |$seen| |$NullStream| |$NonNullStream|))
  (return
   (seq
    (when (null ob)
      (exit nil))
    (when (setq e (hget |$seen| ob))
      (exit e))
    (when (and (pairp ob) (eq (car ob) 'writified!!))
     (exit
      (seq
       (setq type (elt ob 1))
       (when (eq type 'self)
        (exit 'writified!!))
       (when (eq type 'bpi)
        (exit
         (seq
          (setq oname (elt ob 2))
          (setq f
           (seq
            (when (intp oname) (exit (eval (gensymmer oname))))
            (exit (symbol-function oname))))
          (when (null (compiled-function-p f))
           (exit (|error| "A required BPI does not exist.")))
          (when (and (> (|#| ob) 3) (nequal (hasheq f) (elt ob 3)))
           (exit (|error| "A required BPI has been redefined.")))
          (hput |$seen| ob f)
          (exit f))))
       (when (eq type 'hashtable)
        (exit
         (seq
          (setq nob (make-hashtable (elt ob 2)))
          (hput |$seen| ob nob)
          (hput |$seen| nob nob)
          (do ((tmp0 (elt ob 3) (cdr tmp0))
               (k nil)
               (tmp1 (elt ob 4) (cdr tmp1))
               (e nil))
              ((or (atom tmp0)
                   (progn
                    (setq k (car tmp0))
                    nil)
```

```
            (atom tmp1)
            (progn
             (setq e (car tmp1))
             nil))
            nil)
      (seq
       (exit
        (hput nob (|dewritify,dewritifyInner| k)
         (|dewritify,dewritifyInner| e)))))
    (exit nob))))
(when (eq type 'devaluated)
 (exit
  (seq
   (setq nob (eval (|dewritify,dewritifyInner| (elt ob 2))))
   (hput |$seen| ob nob)
   (hput |$seen| nob nob)
   (exit nob))))
(when (eq type 'spadclosure)
 (exit
  (seq
   (setq vec (|dewritify,dewritifyInner| (elt ob 2)))
   (setq name (ELT ob 3))
   (when (null (fboundp name))
    (exit
     (|error|
      (strconc "undefined function: " (symbol-name name)))))
   (setq nob (cons (symbol-function name) vec))
   (hput |$seen| ob nob)
   (hput |$seen| nob nob)
   (exit nob))))
(when (eq type 'place)
 (exit
  (seq
   (setq nob (vmread (make-instream nil)))
   (hput |$seen| ob nob)
   (hput |$seen| nob nob)
   (exit nob))))
(when (eq type 'readtable)
 (exit (|error| "Cannot de-writify a read table.")))
(when (eq type 'nullstream)
 (exit |$NullStream|))
(when (eq type 'nonnullstream)
 (exit |$NonNullStream|))
(when (eq type 'float)
 (exit
  (seq
```

```
      (progn
       (setq tmp1 (cddr ob))
       (setq fval (car tmp1))
       (setq signif (cadr tmp1))
       (setq expon (caddr tmp1))
       (setq sign (cadddr tmp1))
       tmp1)
      (setq fval (scale-float (float signif fval) expon))
      (when (minusp sign)
       (exit (spaddifference fval)))
      (exit fval))))
   (exit (|error| "Unknown type to de-writify."))))
(when (pairp ob)
 (exit
  (seq
   (setq qcar (qcar ob))
   (setq qcdr (qcdr ob))
   (setq nob (cons qcar qcdr))
   (hput |$seen| ob nob)
   (hput |$seen| nob nob)
   (qrplaca nob (|dewritify,dewritifyInner| qcar))
   (qrplacd nob (|dewritify,dewritifyInner| qcdr))
   (exit nob))))
(when (vecp ob)
 (exit
  (seq
   (setq n (qvmaxindex ob))
   (setq nob (make-vec (plus n 1)))
   (hput |$seen| ob nob)
   (hput |$seen| nob nob)
   (do ((|i| 0 (qsadd1 |i|)))
       ((qsgreaterp |i| n) nil)
    (seq
     (exit
      (qsetvelt nob |i|
        (|dewritify,dewritifyInner| (qvelt ob |i|))))))
   (exit nob))))
(exit ob)))))
```

22.5.12 defun dewritify

⟨*defun dewritify*⟩≡
```
(defun |dewritify| (ob)
 (let (|$seen|)
 (declare (special |$seen|))
   (if (null (|ScanOrPairVec| (|function| |dewritify,is?|) ob))
    ob
    (progn
      (setq |$seen| (make-hashtable 'eq))
      (|dewritify,dewritifyInner| ob)))))
```

22.5.13 defun ScanOrPairVec,ScanOrInner

⟨*defun ScanOrPairVec,ScanOrInner*⟩≡
```
(defun |ScanOrPairVec,ScanOrInner| (f ob)
  (declare (special |$seen|))
  (when (hget |$seen| ob) nil)
  (when (pairp ob)
    (hput |$seen| ob t)
    (|ScanOrPairVec,ScanOrInner| f (qcar ob))
    (|ScanOrPairVec,ScanOrInner| f (qcdr ob)))
  (when (vecp ob)
     (hput |$seen| ob t)
     (do ((tmp0 (spaddifference (|#| ob) 1)) (|i| 0 (qsadd1 |i|)))
         ((qsgreaterp |i| tmp0) nil)
       (|ScanOrPairVec,ScanOrInner| f (elt ob |i|))))
  (when (funcall f ob) (throw '|ScanOrPairVecAnswer| t))
  nil)
```

22.5.14 defun ScanOrPairVec

⟨*defun ScanOrPairVec*⟩≡
```
(defun |ScanOrPairVec| (f ob)
 (let (|$seen|)
 (declare (special |$seen|))
  (setq |$seen| (make-hashtable 'eq))
  (catch '|ScanOrPairVecAnswer| (|ScanOrPairVec,ScanOrInner| f ob))))
```

22.5.15 defun gensymInt

⟨*defun gensymInt*⟩≡
```
(defun |gensymInt| (g)
 (let (p n)
  (if (null (gensymp g))
   (|error| "Need a GENSYM")
   (progn
     (setq p (pname g))
     (setq n 0)
     (do ((tmp0 (spaddifference (|#| p) 1)) (|i| 2 (qsadd1 |i|)))
         ((qsgreaterp |i| tmp0) nil)
       (setq n (plus (times 10 n) (|charDigitVal| (elt p |i|)))))
     n))))
```

22.5.16 defun charDigitVal

⟨*defun charDigitVal*⟩≡
```
(defun |charDigitVal| (c)
 (let (digits n)
  (setq digits "0123456789")
  (setq n (spaddifference 1))
  (do ((tmp0 (spaddifference (|#| digits) 1)) (|i| 0 (qsadd1 |i|)))
      ((or (qsgreaterp |i| tmp0) (null (minusp n))) nil)
   (if (char= c (elt digits |i|))
    (setq n |i|)
    nil))
  (if (minusp n)
   (|error| "Character is not a digit")
   n)))
```

22.5.17 defun histFileErase

⟨*defun histFileErase*⟩≡
```
(defun |histFileErase| (file)
 (when (probe-file file) (delete-file file)))
```

22.6 History File Messages

⟨*History File Messages*⟩≡

```
S2IH0001
 You have not reached step %1b yet, and so its value cannot be
 supplied.
S2IH0002
 Cannot supply value for step %1b because 1 is the first step.
S2IH0003
 Step %1b has no value.
S2IH0004
 The history facility is not on, so you cannot use %b %% %d .
S2IH0006
 You have not used the correct syntax for the %b history %d command.
 Issue %b )help history %d for more information.
S2IH0007
 The history facility is already on.
S2IH0008
 The history facility is now on.
S2IH0009
 Turning on the history facility will clear the contents of the
 workspace.
 Please enter %b y %d or %b yes %d if you really want to do this:
S2IH0010
 The history facility is still off.
S2IH0011
 The history facility is already off.
S2IH0012
 The history facility is now off.
S2IH0013
 The history facility is not on, so the .input file containing your user input
 cannot be created.
S2IH0014
 Edit %b %1 %d to see the saved input lines.
S2IH0015
 The argument %b n %d for %b )history )change n must be a nonnegative
 integer and your argument, %1b , is not one.
S2IH0016
 The history facility is not on, so no information can be saved.
S2IH0018
 The saved history file is %1b .
S2IH0019
 There is no history file, so value of step %1b is
 undefined.
S2IH0022
 No history information had been saved yet.
```

```
S2IH0023
 %1b is not a valid filename for the history file.
S2IH0024
 History information cannot be restored from %1b because the file does
 not exist.
S2IH0025
 The workspace has been successfully restored from the history file
 %1b .
S2IH0026
 The history facility command %1b cannot be performed because the
 history facility is not on.
S2IH0027
 A value containing a %1b is being saved in a history file or a
 compiled input file INLIB. This type
 is not yet usable in other history operations.  You might want to issue
 %b )history )off %d
S2IH0029
 History information is already being maintained in an external file
 (and not in memory).
S2IH0030
 History information is already being maintained in memory (and not
 in an external file).
S2IH0031
 When the history facility is active, history information will be
 maintained in a file (and not in an internal table).
S2IH0032
 When the history facility is active, history information will be
 maintained in memory (and not in an external file).
S2IH0034
 Missing element in internal history table.
S2IH0035
 Can't save the value of step number %1b.  You can re-generate this value
 by running the input file %2b.
S2IH0036
 The value specified cannot be saved to a file.
S2IH0037
 You must specify a file name to the history save command
S2IH0038
 You must specify a file name to the history write command
```

Chapter 23

)include Command

23.1 include man page

⟨*include.help*⟩≡

```
User Level Required: interpreter

Command Syntax:

 )include filename

Command Description:

The )include command can be used in .input files to place the contents
of another file inline with the current file.  The path can be an
absolute or relative pathname.
```

23.2 Functions

23.2.1 defun ncloopInclude1

⟨*defun ncloopInclude1*⟩≡

```
(defun |ncloopInclude1| (name n)
 (let (a)
  (if (setq a (|ncloopIncFileName| name))
    (|ncloopInclude| a n)
    n)))
```

23.2.2 Returns the first non-blank substring of the given string

⟨*defun ncloopIncFileName*⟩≡
```
(defun |ncloopIncFileName| (string)
 "Returns the first non-blank substring of the given string"
 (let (fn)
  (unless (setq fn (|incFileName| string))
   (write-line (concat string " not found")))
 fn))
```

23.2.3 Open the include file and read it in

The ncloopInclude0 function is part of the parser and lives in int-top.boot.

⟨*defun ncloopInclude*⟩≡
```
(defun |ncloopInclude| (name n)
 "Open the include file and read it in"
 (with-open-file (st name) (|ncloopInclude0| st name n)))
```

23.2.4 Return the include filename

Given a string we return the first token from the string which is the first non-blank substring.

⟨*defun incFileName*⟩≡
```
(defun |incFileName| (x)
 "Return the include filename"
  (car (|incBiteOff| x)))
```

23.2.5 Return the next token

Takes a sequence and returns the a list of the first token and the remaining string characters. If there are no remaining string characters the second string is of length 0. Effectively it "bites off" the first token in the string. If the string only 0 or more blanks it returns nil.

⟨*defun incBiteOff*⟩≡

```
(defun |incBiteOff| (x)
 "Return the next token"
 (let (blank nonblank)
  (setq x (string x))
  (when (setq nonblank (position #\space x :test-not #'char=))
    (setq blank (position #\space x :start nonblank))
    (if blank
     (list (subseq x nonblank blank) (subseq x blank))
     (list (subseq x nonblank) "")))))
```

Chapter 24

)library Command

24.1 library man page

⟨*library.help*⟩≡

```
==========================================================================
A.14.  )library
==========================================================================

User Level Required:  interpreter

Command Syntax:

  - )library libName1  [libName2 ...]
  - )library )dir dirName
  - )library )only objName1  [objlib2 ...]
  - )library )noexpose

Command Description:

This command replaces the )load system command that was available in AXIOM
releases before version 2.0. The )library command makes available to AXIOM
the compiled objects in the libraries listed.

For example, if you )compile dopler.as in your home directory, issue )library
dopler to have AXIOM look at the library, determine the category and domain
constructors present, update the internal database with various properties of
the constructors, and arrange for the constructors to be automatically loaded
when needed. If the )noexpose option has not been given, the constructors
will be exposed (that is, available) in the current frame.
```

If you compiled a file with the old system compiler, you will have an NRLIB present, for example, DOPLER.NRLIB, where DOPLER is a constructor abbreviation. The command)library DOPLER will then do the analysis and database updates as above.

To tell the system about all libraries in a directory, use)library)dir dirName where dirName is an explicit directory. You may specify ‘‘.’’ as the directory, which means the current directory from which you started the system or the one you set via the)cd command. The directory name is required.

You may only want to tell the system about particular constructors within a library. In this case, use the)only option. The command)library dopler)only Test1 will only cause the Test1 constructor to be analyzed, autoloaded, etc..

Finally, each constructor in a library are usually automatically exposed when the)library command is used. Use the)noexpose option if you not want them exposed. At a later time you can use)set expose add constructor to expose any hidden constructors.

Note for AXIOM beta testers: At various times this command was called)local and)with before the name)library became the official name.

Also See:
o)cd
o)compile
o)frame
o)set

1

[1] “cd” (11 p 99) “compile” (?? p ??) “frame” (20.5.16 p 167) “set” (32.40.1 p 372)

Chapter 25

)lisp Command

25.1 lisp man page

⟨*lisp.help*⟩≡

```
==========================================================================
A.15.  )lisp
==========================================================================

User Level Required:  development

Command Syntax:

  -  )lisp [lispExpression]

Command Description:

This command is used by AXIOM system developers to have single expressions
evaluated by the Lisp system on which AXIOM is built. The lispExpression is
read by the Lisp reader and evaluated. If this expression is not complete
(unbalanced parentheses, say), the reader will wait until a complete
expression is entered.

Since this command is only useful for evaluating single expressions, the )fin
command may be used to drop out of AXIOM into Lisp.

Also See:
o )system
o )boot
o )fin
```

[1]

25.2 Functions

This command is in the list of `$noParseCommands` 7.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 7.2.1

[1] "system" (37 p 385) "boot" (9 p 91) "fin" (19 p 149)

Chapter 26

)load Command

26.1 load man page

⟨*load.help*⟩≡

```
==========================================================================
A.16.  )load
==========================================================================

User Level Required:  interpreter

Command Description:

This command is obsolete. Use )library instead.
```

Chapter 27

)ltrace Command

27.1 ltrace man page

⟨*ltrace.help*⟩≡

```
==========================================================================
A.17.  )ltrace
==========================================================================

User Level Required:  development

Command Syntax:

This command has the same arguments as options as the )trace command.

Command Description:

This command is used by AXIOM system developers to trace Lisp or BOOT
functions. It is not supported for general use.

Also See:
o )boot
o )lisp
o )trace
```

[1]

27.2 Variables Used

27.3 Functions

[1] "boot" (9 p 91) "lisp" (25 p 223) "trace" (38.1.2 p 392)

Chapter 28

)pquit Command

28.1 pquit man page

⟨*pquit.help*⟩≡

```
==========================================================================
A.18.  )pquit
==========================================================================

User Level Required:  interpreter

Command Syntax:

  - )pquit

Command Description:

This command is used to terminate AXIOM and return to the operating system.
Other than by redoing all your computations or by using the )history )restore
command to try to restore your working environment, you cannot return to
AXIOM in the same state.

)pquit differs from the )quit in that it always asks for confirmation that
you want to terminate AXIOM (the ``p'' is for ``protected''). When you enter
the )pquit command, AXIOM responds

      Please enter y or yes if you really want to leave the interactive
                environment and return to the operating system:

If you respond with y or yes, you will see the message
```

```
            You are now leaving the AXIOM interactive environment.
     Issue the command axiom to the operating system to start a new session.

  and AXIOM will terminate and return you to the operating system (or the
  environment from which you invoked the system). If you responded with
  something other than y or yes, then the message

          You have chosen to remain in the AXIOM interactive environment.

  will be displayed and, indeed, AXIOM would still be running.

  Also See:
  o )fin
  o )history
  o )close
  o )quit
  o )system
```

[1]

28.2 Functions

28.2.1 The top level pquit command

⟨*defun pquit*⟩≡

```
(defun |pquit| ()
 "The top level pquit command"
 (|pquitSpad2Cmd|))
```

28.2.2 The top level pquit command handler

⟨*defun pquitSpad2Cmd*⟩≡

```
(defun |pquitSpad2Cmd| ()
 "The top level pquit command handler"
 (let ((|$quitCommandType| '|protected|))
  (declare (special |$quitCommandType|))
  (|quitSpad2Cmd|)))
```

[1] "fin" (19 p 149) "history" (22.4.7 p 183) "close" (13.2.2 p 113) "quit" (29.2.1 p 234) "system" (37 p 385)

This command is in the list of `$noParseCommands` 7.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 7.2.1

Chapter 29

)quit Command

29.1 quit man page

⟨*quit.help*⟩≡

```
==========================================================================
A.19.  )quit
==========================================================================

User Level Required:  interpreter

Command Syntax:

  - )quit
  - )set quit protected | unprotected

Command Description:

This command is used to terminate AXIOM and return to the operating system.
Other than by redoing all your computations or by using the )history )restore
command to try to restore your working environment, you cannot return to
AXIOM in the same state.

)quit differs from the )pquit in that it asks for confirmation only if the
command

)set quit protected

has been issued. Otherwise, )quit will make AXIOM terminate and return you to
the operating system (or the environment from which you invoked the system).
```

```
The default setting is )set quit protected so that )quit and )pquit behave in
the same way. If you do issue

)set quit unprotected

we suggest that you do not (somehow) assign )quit to be executed when you
press, say, a function key.

Also See:
o )fin
o )history
o )close
o )pquit
o )system
```

[1]

29.2 Functions

29.2.1 The top level quit command

⟨*defun quit*⟩≡

```
(defun |quit| ()
 "The top level quit command"
 (|quitSpad2Cmd|))
```

29.2.2 The top level quit command handler

⟨*defun quitSpad2Cmd*⟩≡

```
(defun |quitSpad2Cmd| ()
 "The top level quit command handler"
 (declare (special |$quitCommandType|))
 (if (eq |$quitCommandType| '|protected|)
  (let (x)
   (setq x (upcase (|queryUserKeyedMsg| 's2iz0031 nil)))
   (when (memq (string2id-n x 1) '(y yes)) (|leaveScratchpad|))
   (|sayKeyedMsg| 's2iz0032 nil)
   (tersyscommand))
   (|leaveScratchpad|)))
```

[1] "fin" (19 p 149) "history" (22.4.7 p 183) "close" (13.2.2 p 113) "pquit" (28.2.1 p 230) "system" (37 p 385)

29.2.3 Leave the Axiom interpreter

⟨*defun leaveScratchpad*⟩≡
```
(defun |leaveScratchpad| ()
 "Leave the Axiom interpreter"
 (bye))
```

This command is in the list of `$noParseCommands` 7.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 7.2.1

Chapter 30

)read Command

30.1 read man page

⟨*read.help*⟩≡

```
==========================================================================
A.20.  )read
==========================================================================

User Level Required:  interpreter

Command Syntax:

  -  )read [fileName]
  -  )read [fileName] []quiet] []ifthere]

Command Description:

This command is used to read .input files into AXIOM. The command

)read matrix.input

will read the contents of the file matrix.input into AXIOM. The ‘‘.input’’
file extension is optional. See the AXIOM User Guide index for more
information about .input files.

This command remembers the previous file you edited, read or compiled. If you
do not specify a file name, the previous file will be read.

The )ifthere option checks to see whether the .input file exists. If it does
not, the )read command does nothing. If you do not use this option and the
```

```
file does not exist, you are asked to give the name of an existing .input
file.

The )quiet option suppresses output while the file is being read.

Also See:
o )compile
o )edit
o )history
```

1

[1] "compile" (?? p ??) "edit" (18.2.1 p 146) "history" (22.4.7 p 183)

Chapter 31

)savesystem Command

31.1 savesystem man page

⟨*savesystem.help*⟩≡

```
==========================================================================
A.8.  )savesystem
==========================================================================

User Level Required:  interpreter


Command Syntax:

  - )savesystem filename

Command Description:

 This command is used to save an AXIOM image to disk.  This creates an
executable file which, when started, has everything loaded into it
that was there when the image was saved.  Thus, after executing commands
which cause the loading of some packages, the command:

)savesystem /tmp/savesys

will create an image that can be restarted  with the UNIX command:

axiom -ws /tmp/savesys

This new system will not need to reload the packages and domains that
were already loaded when the system was saved.
```

```
There is currently a restriction that only systems started with the
command "AXIOMsys" may be saved.
```

Chapter 32

)set Command

32.1 set man page

⟨*set.help*⟩≡

```
==========================================================================
A.21.  )set
==========================================================================

User Level Required:  interpreter

Command Syntax:

  -  )set
  -  )set label1 [... labelN]
  -  )set label1 [... labelN] newValue

Command Description:

The )set command is used to view or set system variables that control what
messages are displayed, the type of output desired, the status of the history
facility, the way AXIOM user functions are cached, and so on. Since this
collection is very large, we will not discuss them here. Rather, we will show
how the facility is used. We urge you to explore the )set options to
familiarize yourself with how you can modify your AXIOM working environment.
There is a HyperDoc version of this same facility available from the main
HyperDoc menu. Click [here] to go to it.

The )set command is command-driven with a menu display. It is
tree-structured. To see all top-level nodes, issue )set by itself.
```

```
)set

Variables with values have them displayed near the right margin. Subtrees of
selections have ``...'' displayed in the value field. For example, there are
many kinds of messages, so issue )set message to see the choices.

)set message

The current setting for the variable that displays whether computation times
are displayed is visible in the menu displayed by the last command. To see
more information, issue

)set message time

This shows that time printing is on now. To turn it off, issue

)set message time off

As noted above, not all settings have so many qualifiers. For example, to
change the )quit command to being unprotected (that is, you will not be
prompted for verification), you need only issue

)set quit unprotected

Also See:
o )quit
```

1

32.2 Overview

This section contains tree of information used to initialize the **)set** command in the interpreter. The current list is:

```
Variable     Description                                Current Value
-----------------------------------------------------------------
compiler   Library compiler options                     ...
breakmode  execute break processing on error            break
expose     control interpreter constructor exposure     ...
functions  some interpreter function options            ...
fortran    view and set options for FORTRAN output      ...
kernel     library functions built into the kernel for
           efficiency                                   ...
hyperdoc   options in using HyperDoc                    ...
help       view and set some help options               ...
history    save workspace values in a history file      on
messages   show messages for various system features    ...
naglink    options for NAGLink                          ...
output     view and set some output options             ...
quit       protected or unprotected quit                unprotected
streams    set some options for working with streams    ...
system     set some system development variables        ...
userlevel  operation access level of system user        development

Variables with current values of ... have further sub-options.
For example, issue )set  system to see what the options are
for system. For more information, issue )help set .
```

32.3 Variables Used

32.4 Functions

32.4.1 Initialize the set variables

The argument settree is initially the `$setOption` variable. The fourth element is a union-style switch symbol. The fifth element is usually a variable to set. The sixth element is a subtree to recurse for the TREE switch. The seventh

[1] "quit" (29.2.1 p 234)

element is usually the default value. For more detailed explanations see the list structure section 32.5.

⟨*defun initializeSetVariables*⟩≡

```
(defun |initializeSetVariables| (settree)
 "Initialize the set variables"
 (dolist (setdata settree)
  (case (fourth setdata)
   (FUNCTION
      (if (functionp (fifth setdata))
        (funcall (fifth setdata) '|%initialize%|)
        (|sayMSG| (concatenate 'string "   Function not implemented. "
         (package-name *package*) ":" (string (fifth setdata))))))
   (INTEGER (set (fifth setdata) (seventh setdata)))
   (STRING  (set (fifth setdata) (seventh setdata)))
   (LITERALS
       (set (fifth setdata) (|translateYesNo2TrueFalse| (seventh setdata))))
   (TREE  (|initializeSetVariables| (sixth setdata))))))
```

32.4.2 Reset the workspace variables

⟨*defun resetWorkspaceVariables*⟩≡

```
(defun |resetWorkspaceVariables| ()
 "Reset the workspace variables"
 (declare (special /countlist /editfile /sourcefiles |$sourceFiles| /pretty
    /spacelist /timerlist |$existingFiles| |$functionTable| $boot
    |$compileMapFlag| |$echoLineStack| |$operationNameList| |$slamFlag|
    |$CommandSynonymAlist| |$InitialCommandSynonymAlist|
    |$UserAbbreviationsAlist| |$msgAlist| |$msgDatabase| |$msgDatabaseName|
    |$dependeeClosureAlist| |$IOindex| |$coerceIntByMapCounter| |$e| |$env|
    |$setOptions|))
 (setq /countlist nil)
 (setq /editfile nil)
 (setq /sourcefiles nil)
 (setq |$sourceFiles| nil)
 (setq /pretty nil)
 (setq /spacelist nil)
 (setq /timerlist nil)
 (setq |$existingFiles| (make-hashtable 'uequal))
 (setq |$functionTable| nil)
 (setq $boot nil)
 (setq |$compileMapFlag| nil)
 (setq |$echoLineStack| nil)
 (setq |$operationNameList| nil)
 (setq |$slamFlag| nil)
 (setq |$CommandSynonymAlist| (copy |$InitialCommandSynonymAlist|))
 (setq |$UserAbbreviationsAlist| nil)
 (setq |$msgAlist| nil)
 (setq |$msgDatabase| nil)
 (setq |$msgDatabaseName| nil)
 (setq |$dependeeClosureAlist| nil)
 (setq |$IOindex| 1)
 (setq |$coerceIntByMapCounter| 0)
 (setq |$e| (cons (cons nil nil) nil))
 (setq |$env| (cons (cons nil nil) nil))
 (|initializeSetVariables| |$setOptions|))
```

32.4.3 Display the set option information

⟨*defun displaySetOptionInformation*⟩≡

```
(defun |displaySetOptionInformation| (arg setdata)
 "Display the set option information"
 (let (current)
 (declare (special $linelength))
  (cond
   ((eq (fourth setdata) 'tree)
    (|displaySetVariableSettings| (sixth setdata) (first setdata)))
   (t
    (|centerAndHighlight|
     (strconc "The " (|object2String| arg) " Option")
      $linelength (|specialChar| '|hbar|))
    (|sayBrightly|
     `(|%l| ,@(|bright| "Description:") ,(second setdata)))
    (case (fourth setdata)
     (FUNCTION
      (terpri)
      (if (functionp (fifth setdata))
        (funcall (fifth setdata) '|%describe%|)
        (|sayMSG| "   Function not implemented.")))
     (INTEGER
      (|sayMessage|
        `(" The" ,@(|bright| arg) "option"
          " may be followed by an integer in the range"
          ,@(|bright| (elt (sixth setdata) 0)) "to"
          |%l| ,@(|bright| (elt (sixth setdata) 1)) "inclusive."
          " The current setting is" ,@(|bright| (|eval| (fifth setdata))))))
     (STRING
      (|sayMessage|
        `(" The" ,@(|bright| arg) "option"
          " is followed by a string enclosed in double quote marks."
          '|%l| " The current setting is"
          ,@(|bright| (list '|"| (|eval| (fifth setdata)) '|"|)))))
     (LITERALS
       (|sayMessage|
        `(" The" ,@(|bright| arg) "option"
              " may be followed by any one of the following:"))
       (setq current
        (|translateTrueFalse2YesNo| (|eval| (fifth setdata))))
       (dolist (name (sixth setdata))
         (if (boot-equal name current)
           (|sayBrightly| `( " ->" ,@(|bright| (|object2String| name))))
           (|sayBrightly| (list "    " (|object2String| name)))))
       (|sayMessage| " The current setting is indicated."))))))
```

32.4.4 Display the set variable settings

⟨*defun displaySetVariableSettings*⟩≡

```
(defun |displaySetVariableSettings| (settree label)
 "Display the set variable settings"
 (let (setoption opt subtree subname)
 (declare (special $linelength))
  (if (eq label '||)
    (setq label ")set")
    (setq label (strconc "  " (|object2String| label) " ")))
  (|centerAndHighlight|
    (strconc "Current Values of" label " Variables") $linelength '| |)
  (terpri)
  (|sayBrightly|
   (list "Variable     " "Description                                    "
         "Current Value" ))
  (say (|fillerSpaces| $linelength (|specialChar| '|hbar|)))
  (setq subtree nil)
  (dolist (setdata settree)
   (when (|satisfiesUserLevel| (third setdata))
     (setq setoption (|object2String| (first setdata)))
     (setq setoption
      (strconc setoption
       (|fillerSpaces| (spaddifference 13 (|#| setoption)) " ")
       (second setdata)))
     (setq setoption
      (strconc setoption
       (|fillerSpaces| (spaddifference 55 (|#| setoption)) " ")))
     (case (fourth setdata)
      (FUNCTION
       (setq opt
        (if (functionp (fifth setdata))
         (funcall (fifth setdata) '|%display%|)
         "unimplemented"))
       (cond
        ((pairp opt)
         (setq opt
          (do ((t2 opt (cdr t2)) t1 (|o| nil))
              ((or (atom t2) (progn (setq |o| (car t2)) nil)) t1)
            (setq t1 (append t1 (cons |o| (cons " " nil))))))))
       (|sayBrightly| (|concat| setoption '|%b| opt '|%d|)))
      (STRING
       (setq opt (|object2String| (|eval| (fifth setdata))))
       (|sayBrightly| `(,setoption ,@(|bright| opt))))
      (INTEGER
       (setq opt (|object2String| (|eval| (fifth setdata))))
```

```
        (|sayBrightly| `(,setoption ,@(|bright| opt))))
       (LITERALS
        (setq opt (|object2String|
              (|translateTrueFalse2YesNo| (|eval| (fifth setdata)))))
        (|sayBrightly| `(,setoption ,@(|bright| opt))))
       (TREE
          (|sayBrightly| `(,setoption ,@(|bright| "...")))
          (setq subtree t)
          (setq subname (|object2String| (first setdata)))))))
   (terpri)
   (when subtree
      (|sayBrightly|
       `("Variables with current values of" ,@(|bright| "...")
             "have further sub-options. For example,"))
      (|sayBrightly|
       `("issue" ,@(|bright| ")set ") ,subname
         " to see what the options are for" ,@(|bright| subname) "."
         |%l| "For more information, issue" ,@(|bright| ")help set") ".")))))
```

32.4.5 Translate options values to t or nil

⟨*defun translateYesNo2TrueFalse*⟩≡

```
(defun |translateYesNo2TrueFalse| (x)
 "Translate options values to t or nil"
 (cond
  ((|member| x '(|yes| |on|)) t)
  ((|member| x '(|no| |off|)) nil)
  (t x)))
```

32.4.6 Translate t or nil to option values

⟨*defun translateTrueFalse2YesNo*⟩≡

```
(defun |translateTrueFalse2YesNo| (x)
 "Translate t or nil to option values"
 (cond
  ((eq x t) '|on|)
  ((null x) '|off|)
  (t x)))
```

32.5 The list structure

The structure of each list item consists of 7 items. Consider this example:

```
(userlevel
 "operation access level of system user"
 interpreter
 LITERALS
 $UserLevel
 (interpreter compiler development)
 development)
```

The list looks like (the names in bold are accessor names that can be found in **property.lisp.pamphlet**[**1**]. Look for "setName".):

1 *Name* the keyword the user will see. In this example the user would say "**)set output userlevel**".

2 *Label* the message the user will see. In this example the user would see "operation access level of system user".

3 *Level* the level where the command will be accepted. There are three levels: interpreter, compiler, development. These commands are restricted to keep the user from causing damage.

4 *Type* a symbol, one of **FUNCTION**, **INTEGER**, **STRING**, **LITERALS**, **FILENAME** or **TREE**.

5 *Var*

FUNCTION is the function to call

INTEGER is the variable holding the current user setting.

STRING is the variable holding the current user setting.

LITERALS variable which holds the current user setting.

FILENAME is the variable that holds the current user setting.

TREE

6 *Leaf*

FUNCTION is the list of all possible values

INTEGER is the range of possible values

STRING is a list of all possible values

LITERALS is a list of all of the possible values

FILENAME is the function to check the filename

TREE

7 *Def* is the default value

FUNCTION is the default setting

INTEGER is the default setting

STRING is the default setting

LITERALS is the default setting

FILENAME is the default value

TREE

32.6 breakmode

```
-------------------- The breakmode Option ----------------------

 Description: execute break processing on error

 The breakmode option may be followed by any one of the
 following:

    nobreak
 -> break
    query
    resume
    fastlinks

 The current setting is indicated.
```

32.6.1 defvar $BreakMode

⟨*initvars*⟩+≡

```
  (defvar |$BreakMode| '|nobreak| "execute break processing on error")
```

⟨*breakmode*⟩≡

```
    (|breakmode|
     "execute break processing on error"
     |interpreter|
     LITERALS
     |$BreakMode|
     (|nobreak| |break| |query| |resume| |fastlinks|)
     |nobreak|)         ; needed to avoid possible startup looping
```

32.7 debug

```
                Current Values of debug Variables

Variable     Description                              Current Value
-------------------------------------------------------------------
lambdatype   Show type information for #1 syntax      off
dalymode     Interpret leading open paren as lisp     off
```

⟨*debug*⟩≡
```
     (|debug|
      "debug options"
      |interpreter|
      TREE
      |novar|
      (
```
⟨*debuglambdatype*⟩
⟨*debugdalymode*⟩
```
      ))
```

32.8 debug lambda type

```
---------------------- The lambdatype Option ----------------------

 Description: Show type information for #1 syntax
```

32.8.1 defvar $lambdatype

⟨*initvars*⟩+≡
```
  (defvar $lambdatype nil "show type information for #1 syntax")
```

⟨*debuglambdatype*⟩≡
```
        (|lambdatype|
         "show type information for #1 syntax"
         |interpreter|
         LITERALS
         $lambdatype
         (|on| |off|)
         |off|)
```

32.9 debug dalymode

The `$dalymode` variable is used in a case statement in intloopReadConsole. This variable can be set to any non-nil value. When not nil the interpreter will send any line that begins with an "(" to be sent to the underlying lisp. This is useful for debugging Axiom. The normal value of this variable is NIL.

This variable was created as an alternative to prefixing every lisp command with)lisp. When doing a lot of debugging this is tedious and error prone. This variable was created to shortcut that process. Clearly it breaks some semantics of the language accepted by the interpreter as parens are used for grouping expressions.

```
---------------------- The dalymode Option ----------------------

 Description: Interpret leading open paren as lisp
```

32.9.1 defvar $dalymode

⟨*initvars*⟩+≡

```
(defvar $dalymode nil "Interpret leading open paren as lisp")
```

⟨*debugdalymode*⟩≡

```
        (|dalymode|
         "Interpret leading open paren as lisp"
         |interpreter|
         LITERALS
         $dalymode
         (|on| |off|)
         |off|)
```

32.10 compiler

```
                Current Values of  compiler  Variables

Variable     Description                              Current Value
-------------------------------------------------------------------
output       library in which to place compiled code
input        controls libraries from which to load compiled code
args         arguments for compiling AXIOM code
             -O -Fasy -Fao -Flsp -laxiom -Mno-AXL_W_WillObsolete
             -DAxiom -Y $AXIOM/algebra
```

⟨*compiler*⟩≡

```
     (|compiler|
      "Library compiler options"
      |interpreter|
      TREE
      |novar|
      (
  ⟨compileroutput⟩
  ⟨compilerinput⟩
  ⟨compilerargs⟩
      ))
```

32.11 compiler output

```
---------------------- The output Option ----------------------

 Description: library in which to place compiled code
```

⟨*compileroutput*⟩≡

```
        (|output|
         "library in which to place compiled code"
         |interpreter|
         FUNCTION
         |setOutputLibrary|
         NIL
         |htSetOutputLibrary|
          )
```

32.12 Variables Used

32.13 Functions

32.13.1 The set output command handler

⟨*defun setOutputLibrary*⟩≡

```
(defun |setOutputLibrary| (arg)
 "The set output command handler"
 (let (fn)
 (declare (special |$outputLibraryName|))
   (cond
    ((eq arg '|%initialize%|) (setq |$outputLibraryName| nil))
    ((eq arg '|%display%|) (or |$outputLibraryName| "user.lib"))
    ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?) (/= (|#| arg) 1))
     (|describeOutputLibraryArgs|))
    (t
     (when (filep (setq fn (stringimage (car arg))))
       (setq fn (truename fn)))
     (|openOutputLibrary| (setq |$outputLibraryName| fn))))))
```

32.13.2 Describe the set output library arguments

⟨*defun describeOutputLibraryArgs*⟩≡

```
(defun |describeOutputLibraryArgs| ()
 "Describe the set output library arguments"
 (|sayBrightly| (list
  '|%b| ")set compiler output library"
  '|%d| "is used to tell the compiler where to place"
  '|%l| "compiled code generated by the library compiler.  By default it goes"
  '|%l| "in a file called"
  '|%b| "user.lib"
  '|%d| "in the current directory.")))
```

32.13.3 Open the output library

The input-libraries and output-library are now truename based.

⟨*defun openOutputLibrary*⟩≡
```
(defun |openOutputLibrary| (lib)
 "Open the output library"
 (declare (special output-library input-libraries))
  (|dropInputLibrary| lib)
  (setq output-library (truename lib))
  (push output-library input-libraries))
```

32.14 compiler input

```
---------------------- The input Option -----------------------

 Description: controls libraries from which to load compiled code

 )set compiler input add library is used to tell AXIOM to add
  library to the front of the path which determines where
  compiled code is loaded from.
 )set compiler input drop library is used to tell AXIOM to remove
  library from this path.
```

⟨*compilerinput*⟩≡
```
        (|input|
         "controls libraries from which to load compiled code"
         |interpreter|
         FUNCTION
         |setInputLibrary|
         NIL
         |htSetInputLibrary|)
```

32.15 Variables Used

32.16 Functions

32.16.1 The set input library command handler

The input-libraries is now maintained as a list of truenames.

⟨*defun setInputLibrary*⟩≡

```
(defun |setInputLibrary| (arg)
 "The set input library command handler"
 (declare (special input-libraries))
 (let (tmp1 filename act)
  (cond
   ((eq arg '|%initialize%|) t)
   ((eq arg '|%display%|) (mapcar #'namestring input-libraries))
   ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
    (|describeInputLibraryArgs|))
   ((and (pairp arg)
         (progn
          (setq act (qcar arg))
          (setq tmp1 (qcdr arg))
          (and (pairp tmp1)
               (eq (qcdr tmp1) nil)
               (progn (setq filename (qcar tmp1)) t)))
         (setq act (|selectOptionLC| act '(|add| |drop|) nil)))
    (cond
     ((eq act '|add|)
      (|addInputLibrary| (truename (stringimage filename))))
     ((eq act '|drop|)
      (|dropInputLibrary| (truename (stringimage filename))))))
   (t (|setInputLibrary| nil)))))
```

32.16.2 Describe the set input library arguments

⟨*defun describeInputLibraryArgs*⟩≡
```
(defun |describeInputLibraryArgs| ()
 "Describe the set input library arguments"
 (|sayBrightly| (list
  '|%b| ")set compiler input add library"
  '|%d| "is used to tell AXIOM to add"
  '|%b| "library"
  '|%d| "to"
  '|%l| "the front of the path used to find compile code."
  '|%l|
  '|%b| ")set compiler input drop library"
  '|%d| "is used to tell AXIOM to remove"
  '|%b| "library"
  '|%d|
  '|%l| "from this path.")))
```

32.16.3 Add the input library to the list

The input-libraries variable is now maintained as a list of truenames.

⟨*defun addInputLibrary*⟩≡
```
(defun |addInputLibrary| (lib)
 "Add the input library to the list"
 (declare (special input-libraries))
  (|dropInputLibrary| lib)
  (push (truename lib) input-libraries))
```

32.16.4 Drop an input library from the list

⟨*defun dropInputLibrary*⟩≡
```
(defun |dropInputLibrary| (lib)
 "Drop an input library from the list"
 (declare (special input-libraries))
 (setq input-libraries (delete (truename lib) input-libraries :test #'equal)))
```

32.17 compiler args

```
---------------------- The args Option -----------------------

 Description: arguments for compiling AXIOM code

 )set compiler args  is used to tell AXIOM how to invoke the
  library compiler when compiling code for AXIOM.
  The args option is followed by a string enclosed in double
  quotes.

 The current setting is
 "-O -Fasy -Fao -Flsp -laxiom -Mno-AXL_W_WillObsolete
  -DAxiom -Y $AXIOM/algebra"
```

32.17.1 defvar $asharpCmdlineFlags

⟨*initvars*⟩+≡

```
(defvar |$asharpCmdlineFlags|
  "-O -Fasy -Fao -Flsp -laxiom -Mno-AXL__W__WillObsolete -DAxiom -Y $AXIOM/algebra"
  "arguments for compiling AXIOM code")
```

⟨*compilerargs*⟩≡

```
        (|args|
         "arguments for compiling AXIOM code"
         |interpreter|
         FUNCTION
         |setAsharpArgs|
         (("enter compiler options "
            STRING
            |$asharpCmdlineFlags|
            |chkDirectory|
            "-O -Fasy -Fao -Flsp -laxiom -Mno-AXL__W__WillObsolete -DAxiom -Y $AXIOM/algebra")
         NIL)
```

32.18 Variables Used

32.19 Functions

32.19.1 Handle the set compiler command arguments

⟨*defun setAsharpArgs*⟩≡
```
(defun |setAsharpArgs| (arg)
 "Handle the set compiler command arguments"
 (declare (special |$asharpCmdlineFlags|))
 (cond
  ((eq arg '|%initialize%|)
   (setq |$asharpCmdlineFlags|
    "-O -Fasy -Fao -Flsp -laxiom -Mno-AXL_W_WillObsolete -DAxiom -Y $AXIOM/algebr
  ((eq arg '|%display%|) |$asharpCmdlineFlags|)
  ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
   (|describeAsharpArgs|))
  (t (setq |$asharpCmdlineFlags| (car arg)))))
```

32.19.2 Describe the set compiler command arguments

⟨*defun describeAsharpArgs*⟩≡
```
(defun |describeAsharpArgs| ()
 "Describe the set compiler command arguments"
 (declare (special |$asharpCmdlineFlags|))
 (|sayBrightly| (list
  '|%b| ")set compiler args "
  '|%d| "is used to tell AXIOM how to invoke the library compiler "
  '|%l| " when compiling code for AXIOM."
  '|%l| " The args option is followed by a string enclosed in double quotes."
  '|%l|
  '|%l| " The current setting is"
  '|%l|
  '|%b| "\"" |$asharpCmdlineFlags| "\""
  '|%d|)))
```

32.20 expose

```
---------------------- The expose Option ----------------------

 Description: control interpreter constructor exposure

   The following groups are explicitly exposed in the current
   frame (called initial ):
                                    basic
                                categories
                                  naglink
                                   anna

   The following constructors are explicitly exposed in the
   current frame:
               there are no explicitly exposed constructors

   The following constructors are explicitly hidden in the
   current frame:
                there are no explicitly hidden constructors

   When )set expose is followed by no arguments, the information
   you now see is displayed. When followed by the initialize
   argument, the exposure group data in the file interp.exposed
   is read and is then available. The arguments add and drop are
   used to add or drop exposure groups or explicit constructors
   from the local frame exposure data. Issue
                  )set expose add    or    )set expose drop
   for more information.
```

⟨*expose*⟩≡

```
(|expose|
 "control interpreter constructor exposure"
 |interpreter|
 FUNCTION
 |setExpose|
 NIL
 |htSetExpose|)
```

32.21 Variables Used

32.22 Functions

32.22.1 The top level set expose command handler

⟨*defun setExpose*⟩≡

```
(defun |setExpose| (arg)
 "The top level set expose command handler"
 (let (fnargs fn)
   (cond
    ((eq arg '|%initialize%|) (|loadExposureGroupData|))
    ((eq arg '|%display%|) "...")
    ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
     (|displayExposedGroups|)
     (|sayMSG| " ")
     (|displayExposedConstructors|)
     (|sayMSG| " ")
     (|displayHiddenConstructors|)
     (|sayMSG| " ")
     (|sayKeyedMsg| 's2iz0049d
      (list (|namestring| (|pathname| (list '|interp| '|exposed|))))))
    ((and (pairp arg)
          (progn (setq fn (qcar arg)) (setq fnargs (qcdr arg)) t)
          (setq fn (|selectOptionLC| fn '(|add| |drop| |initialize|) nil)))
     (cond
      ((eq fn '|add|) (|setExposeAdd| fnargs))
      ((eq fn '|drop|) (|setExposeDrop| fnargs))
      ((eq fn '|initialize|) (|setExpose| '|%initialize%|))
      (t nil)))
    (t (|setExpose| nil)))))
```

32.22.2 The top level set expose add command handler

⟨*defun setExposeAdd*⟩≡

```
(defun |setExposeAdd| (arg)
 "The top level set expose add command handler"
 (declare (special $linelength))
 (let (fnargs fn)
   (cond
    ((null arg)
     (|centerAndHighlight|
      '|The add Option| $linelength (|specialChar| '|hbar|))
     (|displayExposedGroups|)
     (|sayMSG| " ")
     (|displayExposedConstructors|)
     (|sayMSG| " ")
     (|sayKeyedMsg| 's2iz0049e nil))
    ((and (pairp arg)
          (progn (setq fn (qcar arg)) (setq fnargs (qcdr arg)) t)
          (setq fn (|selectOptionLC| fn '(|group| |constructor|) nil)))
     (cond
      ((eq fn '|group|) (|setExposeAddGroup| fnargs))
      ((eq fn '|constructor|) (|setExposeAddConstr| fnargs))
      (t nil)))
    (t (|setExposeAdd| nil)))))
```

32.22.3 Expose a group

Note that `$localExposureData` is a vector of lists. It consists of [exposed groups,exposed constructors,hidden constructors]

⟨*defun setExposeAddGroup*⟩≡

```
(defun |setExposeAddGroup| (arg)
 "Expose a group"
 (declare (special |$globalExposureGroupAlist| |$localExposureData|
                   |$interpreterFrameName| $linelength))
 (if (null arg)
  (progn
   (|centerAndHighlight|
    '|The group Option| $linelength (|specialChar| '|hbar|))
   (|displayExposedGroups|)
   (|sayMSG| " ")
   (|sayKeyedMsg| 's2iz0049g
    (list (|namestring| (|pathname| (list '|interp| '|exposed| )))))
   (|sayMSG| " ")
   (|sayAsManyPerLineAsPossible|
    (mapcar #'(lambda (x) (|object2String| (first x)))
      |$globalExposureGroupAlist|)))
  (dolist (x arg)
    (when (pairp x) (setq x (qcar x)))
    (cond
     ((eq x '|all|)
      (setelt |$localExposureData| 0
        (mapcar #'first |$globalExposureGroupAlist|))
      (setelt |$localExposureData| 1 nil)
      (setelt |$localExposureData| 2 nil)
      (|displayExposedGroups|)
      (|sayMSG| " ")
      (|displayExposedConstructors|)
      (|sayMSG| " ")
      (|displayHiddenConstructors|)
      (|clearClams|))
     ((null (getalist |$globalExposureGroupAlist| x))
      (|sayKeyedMsg| 's2iz0049h (cons x nil)))
     ((|member| x (elt |$localExposureData| 0))
      (|sayKeyedMsg| 's2iz0049i (list x |$interpreterFrameName|)))
     (t
      (setelt |$localExposureData| 0
        (msort (cons x (elt |$localExposureData| 0))))
      (|sayKeyedMsg| 's2iz0049r (list x |$interpreterFrameName|))
      (|clearClams|))))))
```

32.22.4 The top level set expose add constructor handler

⟨*defun setExposeAddConstr*⟩≡

```
(defun |setExposeAddConstr| (arg)
 "The top level set expose add constructor handler"
 (declare (special $linelength |$localExposureData| |$interpreterFrameName|))
 (if (null arg)
   (progn
    (|centerAndHighlight|
     '|The constructor Option| $linelength (|specialChar| '|hbar|))
    (|displayExposedConstructors|))
   (dolist (x arg)
     (setq x (|unabbrev| x))
     (when (pairp x) (setq x (qcar x)))
     (cond
      ((null (getdatabase x 'constructorkind))
       (|sayKeyedMsg| 's2iz0049j (list x)))
      ((|member| x (elt |$localExposureData| 1))
       (|sayKeyedMsg| 's2iz0049k (list x |$interpreterFrameName| )))
      (t
       (when (|member| x (elt |$localExposureData| 2))
         (setelt |$localExposureData| 2
          (|delete| x (elt |$localExposureData| 2))))
       (setelt |$localExposureData| 1
        (msort (cons x (elt |$localExposureData| 1))))
       (|clearClams|)
       (|sayKeyedMsg| 's2iz0049p (list x |$interpreterFrameName| )))))))
```

32.22.5 The top level set expose drop handler

⟨*defun setExposeDrop*⟩≡

```
(defun |setExposeDrop| (arg)
 "The top level set expose drop handler"
 (declare (special $linelength))
 (let (fnargs fn)
   (cond
    ((null arg)
     (|centerAndHighlight|
      '|The drop Option| $linelength (|specialChar| '|hbar|))
     (|displayHiddenConstructors|)
     (|sayMSG| " ")
     (|sayKeyedMsg| 's2iz0049f nil))
    ((and (pairp arg)
          (progn (setq fn (qcar arg)) (setq fnargs (qcdr arg)) t)
          (setq fn (|selectOptionLC| fn '(|group| |constructor|) nil)))
     (cond
      ((eq fn '|group|) (|setExposeDropGroup| fnargs))
      ((eq fn '|constructor|) (|setExposeDropConstr| fnargs))
      (t nil)))
    (t (|setExposeDrop| nil)))))
```

32.22.6 The top level set expose drop group handler

⟨*defun setExposeDropGroup*⟩≡

```
(defun |setExposeDropGroup| (arg)
 "The top level set expose drop group handler"
 (declare (special $linelength |$localExposureData| |$interpreterFrameName|
                   |$globalExposureGroupAlist|))
 (if (null arg)
  (progn
   (|centerAndHighlight|
     '|The group Option| $linelength (|specialChar| '|hbar|))
   (|sayKeyedMsg| 's2iz0049l nil)
   (|sayMSG| " ")
   (|displayExposedGroups|))
  (dolist (x arg)
    (when (pairp x) (setq x (qcar x)))
    (cond
     ((eq x '|all|)
      (setelt |$localExposureData| 0 nil)
      (setelt |$localExposureData| 1 nil)
      (setelt |$localExposureData| 2 nil)
      (|displayExposedGroups|)
      (|sayMSG| " ")
      (|displayExposedConstructors|)
      (|sayMSG| " ")
      (|displayHiddenConstructors|)
      (|clearClams|))
     ((|member| x (elt |$localExposureData| 0))
      (setelt |$localExposureData| 0
       (|delete| x (elt |$localExposureData| 0)))
      (|clearClams|)
      (|sayKeyedMsg| 's2iz0049s (list x |$interpreterFrameName| )))
     ((getalist |$globalExposureGroupAlist| x)
      (|sayKeyedMsg| 's2iz0049i (list x |$interpreterFrameName| )))
     (t (|sayKeyedMsg| 's2iz0049h (list x )))))))
```

32.22.7 The top level set expose drop constructor handler

⟨*defun setExposeDropConstr*⟩≡

```
(defun |setExposeDropConstr| (arg)
 "The top level set expose drop constructor handler"
 (declare (special $linelength |$localExposureData| |$interpreterFrameName|))
 (if (null arg)
  (progn
   (|centerAndHighlight|
    '|The constructor Option| $linelength (|specialChar| '|hbar|))
   (|sayKeyedMsg| 's2iz0049n nil)
   (|sayMSG| " ")
   (|displayExposedConstructors|)
   (|sayMSG| " ")
   (|displayHiddenConstructors|))
  (dolist (x arg)
    (setq x (|unabbrev| x))
    (when (pairp x) (setq x (qcar x)))
    (cond
     ((null (getdatabase x 'constructorkind))
      (|sayKeyedMsg| 's2iz0049j (list x)))
     ((|member| x (elt |$localExposureData| 2))
      (|sayKeyedMsg| 's2iz0049o (list x |$interpreterFrameName|)))
     (t
      (when (|member| x (elt |$localExposureData| 1))
       (setelt |$localExposureData| 1
        (|delete| x (elt |$localExposureData| 1))))
      (setelt |$localExposureData| 2
        (msort (cons x (elt |$localExposureData| 2))))
      (|clearClams|)
      (|sayKeyedMsg| 's2iz0049q (list x |$interpreterFrameName|)))))))
```

32.22.8 Display exposed groups

⟨*defun displayExposedGroups*⟩≡

```
(defun |displayExposedGroups| ()
 "Display exposed groups"
 (declare (special |$interpreterFrameName| |$localExposureData|))
 (|sayKeyedMsg| 's2iz0049a (list |$interpreterFrameName|))
 (if  (null (elt |$localExposureData| 0))
  (|centerAndHighlight| "there are no exposed groups")
  (dolist (c (elt |$localExposureData| 0))
     (|centerAndHighlight| c))))
```

32.22.9 Display exposed constructors

⟨*defun displayExposedConstructors*⟩≡

```
(defun |displayExposedConstructors| ()
 "Display exposed constructors"
 (declare (special |$localExposureData|))
 (|sayKeyedMsg| 's2iz0049b nil)
 (if  (null (elt |$localExposureData| 1))
  (|centerAndHighlight| "there are no explicitly exposed constructors")
  (dolist (c (elt |$localExposureData| 1))
     (|centerAndHighlight| c))))
```

32.22.10 Display hidden constructors

⟨*defun displayHiddenConstructors*⟩≡

```
(defun |displayHiddenConstructors| ()
 "Display hidden constructors"
 (declare (special |$localExposureData|))
 (|sayKeyedMsg| 's2iz0049c nil)
 (if (null (elt |$localExposureData| 2))
   (|centerAndHighlight| "there are no explicitly hidden constructors")
   (dolist (c (elt |$localExposureData| 2))
     (|centerAndHighlight| c))))
```

32.23 functions

```
            Current Values of  functions  Variables

Variable     Description                              Current Value
-------------------------------------------------------------------
cache        number of function results to cache      0
compile      compile, don't just define function bodies off
recurrence   specially compile recurrence relations   on
```

⟨*functions*⟩≡

```
    (|functions|
     "some interpreter function options"
     |interpreter|
     TREE
     |novar|
     (
  ⟨functionscache⟩
  ⟨functionscompile⟩
  ⟨functionsrecurrence⟩
    ))
```

32.24 functions cache

```
---------------------- The cache Option -----------------------

 Description: number of function results to cache

 )set functions cache is used to tell AXIOM how many  values
  computed by interpreter functions should be saved.  This can
  save quite a bit of time in recursive functions, though one
  must consider that the cached values will take up (perhaps
  valuable) room in the workspace.

 The value given  after cache must either be the word all or
 a positive integer. This may be followed by any number of
 function names whose cache sizes you wish to so set.  If no
 functions are given, the default cache  size is set.

 Examples:   )set fun cache all
             )set fun cache 10 f g Legendre

 In general, functions will cache no returned values.
```

⟨*functionscache*⟩≡

```
       (|cache|
        "number of function results to cache"
        |interpreter|
        FUNCTION
        |setFunctionsCache|
        NIL
        |htSetCache|)
```

32.25 Variables Used

32.26 Functions

32.26.1 The top level set functions cache handler

```
\nwenddocs{}\nwbegincode{571}\moddef{defun setFunctionsCache}\endmoddef
(defun |setFunctionsCache| (arg)
 "The top level set functions cache handler"
 (let (|$options| n)
  (declare (special |$options| |$cacheCount| |$cacheAlist|))
  (cond
    ((eq arg '|%initialize%|)
      (setq |$cacheCount| 0)
      (setq |$cacheAlist| nil))
    ((eq arg '|%display%|)
      (if (null |$cacheAlist|)
       (|object2String| |$cacheCount|)
       "..."))
    ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
      (|describeSetFunctionsCache|)
      (terpri)
      (|sayAllCacheCounts|))
    (t
      (setq n (car arg))
      (cond
       ((and (nequal n '|all|) (or (null (fixp n)) (minusp n)))
        (|sayMessage|
         `("Your value of" ,@(|bright| n) "is invalid because ..."))
        (|describeSetFunctionsCache|)
        (|terminateSystemCommand|))
       (t
        (when (cdr arg) (list (cons '|vars| (cdr arg))))
        (|countCache| n)))))))

\nwendcode{}\nwbegindocs{572}\nwdocspar

\defunsec{countCache}{Display a particular cache count}
\nwenddocs{}\nwbegincode{573}\moddef{defun countCache}\endmoddef
(defun |countCache| (n)
 "Display a particular cache count"
 (let (tmp1 l cachecountname)
 (declare (special |$options| |$cacheAlist| |$cacheCount|))
 (cond
  (|$options|
   (cond
    ((and (pairp |$options|)
          (eq (qcdr |$options|) nil)
```

```
          (progn
           (spadlet tmp1 (qcar |$options|))
           (and (pairp tmp1)
                (eq (qcar tmp1) '|vars|)
                (progn (setq l (qcdr tmp1)) t))))
     (dolist (x l)
       (if (null (identp x))
         (|sayKeyedMsg| 's2if0007 (list x))
         (progn
          (setq |$cacheAlist| (|insertAlist| x n |$cacheAlist|))
          (setq cachecountname (internl x ";COUNT"))
          (set cachecountname n)
          (|sayCacheCount| x n)))))
    (t (|optionError| (caar |$options|) nil))))
  (t
   (|sayCacheCount| nil (setq |$cacheCount| n))))))
```

\nwendcode{}\nwbegindocs{574}\nwdocspar

\defunsec{describeSetFunctionsCache}{Describe the set functions cache}
\nwenddocs{}\nwbegincode{575}\moddef{defun describeSetFunctionsCache}\endmoddef

```
(defun |describeSetFunctionsCache| ()
 "Describe the set functions cache"
 (|sayBrightly| (list
  '|%b| ")set functions cache"
  '|%d| "is used to tell AXIOM how many"
  '|%l| " values computed by interpreter functions should be saved.  This"
  '|%l| " can save quite a bit of time in recursive functions, though one"
  '|%l| " must consider that the cached values will take up (perhaps"
  '|%l| " valuable) room in the workspace."
  '|%l|
  '|%l| " The value given after"
  '|%b| "cache"
  '|%d| "must either be the word"
  '|%b| "all"
  '|%d| "or a positive integer."
  '|%l| " This may be followed by any number of function names whose cache"
  '|%l| " sizes you wish to so set.  If no functions are given, the default"
  '|%l| " cache size is set."
  '|%l|
  '|%l|  " Examples:"
  '|%l|  "   )set fun cache all         )set fun cache 10 f g Legendre")))
```

\nwendcode{}\nwbegindocs{576}\nwdocspar

\defunsec{sayAllCacheCounts}{Display all cache counts}
\nwenddocs{}\nwbegincode{577}\moddef{defun sayAllCacheCounts}\endmoddef

```
(defun |sayAllCacheCounts| ()
 "Display all cache counts"
 (let (x n)
```

```
 (declare (special |$cacheCount| |$cacheAlist|))
   (|sayCacheCount| nil |$cacheCount|)
   (when |$cacheAlist|
     (do ((t0 |$cacheAlist| (cdr t0)) (t1 nil))
         ((or (atom t0)
              (progn (setq t1 (car t0)) nil)
              (progn
               (progn (setq x (car t1)) (setq n (cdr t1)) t1)
               nil))
             nil)
       (when (nequal n |$cacheCount|) (|sayCacheCount| x n))))))

\nwendcode{}\nwbegindocs{578}\nwdocspar

\defunsec{sayCacheCount}{Describe the cache counts}
\nwenddocs{}\nwbegincode{579}\moddef{defun sayCacheCount}\endmoddef
(defun |sayCacheCount| (fn n)
 "Describe the cache counts"
 (let (prefix phrase)
    (setq prefix
     (cond
      (fn (cons '|function| (|bright| (|linearFormatName| fn))))
      ((eql n 0) (list '|interpreter functions |))
      (t (list '|In general, interpreter functions |))))
    (cond
     ((eql n 0)
      (cond
       (fn
        (|sayBrightly|
         `("   Caching for " ,prefix "is turned off")))
       (t
        (|sayBrightly| " In general, functions will cache no returned values."
         ))))
     (t
      (setq phrase
       (cond
        ((eq n '|all|) `(,@(|bright| '|all|) |values.|))
        ((eql n 1) (list '| only the last value.|))
        (t `(| the last| ,@(|bright| n) |values.|))))
      (|sayBrightly|
       `("   " ,@prefix "will cache" ,@phrase))))))

\nwendcode{}\nwbegindocs{580}\nwdocspar

\section{functions compile}
\begin{verbatim}
--------------------- The compile Option ----------------------

 Description: compile, don't just define function bodies
```

```
The compile option may be followed by any one of the following:

-> on
   off

The current setting is indicated.
```

32.26.2 defvar $compileDontDefineFunctions

⟨*initvars*⟩+≡
```
(defvar |$compileDontDefineFunctions| t
   "compile, don't just define function bodies")
```

⟨*functionscompile*⟩≡
```
       (|compile|
        "compile, don't just define function bodies"
        |interpreter|
        LITERALS
        |$compileDontDefineFunctions|
        (|on| |off|)
        |on|)
```

32.27 functions recurrence

```
-------------------- The recurrence Option --------------------

Description: specially compile recurrence relations

The recurrence option may be followed by any one of the
following:

-> on
   off

The current setting is indicated.
```

32.27.1 defvar $compileRecurrence

⟨*initvars*⟩+≡
```
(defvar |$compileRecurrence| t "specially compile recurrence relations")
```

⟨*functionsrecurrence*⟩≡

```
(|recurrence|
 "specially compile recurrence relations"
 |interpreter|
 LITERALS
 |$compileRecurrence|
 (|on| |off|)
 |on|)
```

32.28 fortran

```
              Current Values of  fortran  Variables

Variable     Description                              Current Value
-----------------------------------------------------------------
ints2floats  where sensible, coerce integers to reals   on
fortindent   the number of characters indented          6
fortlength   the number of characters on a line         72
typedecs     print type and dimension lines             on
defaulttype  default generic type for FORTRAN object    REAL
precision    precision of generated FORTRAN objects     double
intrinsic    whether to use INTRINSIC FORTRAN functions off
explength    character limit for FORTRAN expressions    1320
segment      split long FORTRAN expressions             on
optlevel     FORTRAN optimisation level                 0
startindex   starting index for FORTRAN arrays          1
calling      options for external FORTRAN calls         ...

Variables with current values of ... have further sub-options.
For example, issue )set calling to see what the options are for
calling.
For more information, issue )help set .
```

⟨*fortran*⟩≡

```
    (|fortran|
     "view and set options for FORTRAN output"
     |interpreter|
     TREE
     |novar|
      (
  ⟨fortranints2floats⟩
  ⟨fortranfortindent⟩
  ⟨fortranfortlength⟩
  ⟨fortrantypedecs⟩
  ⟨fortrandefaulttype⟩
  ⟨fortranprecision⟩
  ⟨fortranintrinsic⟩
  ⟨fortranexplength⟩
  ⟨fortransegment⟩
  ⟨fortranoptlevel⟩
  ⟨fortranstartindex⟩
  ⟨fortrancalling⟩
    ))
```

32.28.1 ints2floats

```
------------------- The ints2floats Option --------------------

 Description: where sensible, coerce integers to reals

 The ints2floats option may be followed by any one of the
 following:

 -> on
    off

 The current setting is indicated.
```

32.28.2 defvar $fortInts2Floats

⟨*initvars*⟩+≡

```
  (defvar |$fortInts2Floats| t "where sensible, coerce integers to reals")
```

⟨*fortranints2floats*⟩≡

```
        (|ints2floats|
         "where sensible, coerce integers to reals"
         |interpreter|
         LITERALS
         |$fortInts2Floats|
         (|on| |off|)
         |on|)
```

32.28.3 fortindent

```
-------------------- The fortindent Option --------------------

 Description: the number of characters indented

 The fortindent option may be followed by an integer in the range
 0 to inclusive. The current setting is 6
```

32.28.4 defvar $fortIndent

⟨*initvars*⟩+≡

```
  (defvar |$fortIndent| 6 "the number of characters indented")
```

⟨*fortranfortindent*⟩≡
```
       (|fortindent|
        "the number of characters indented"
        |interpreter|
        INTEGER
        |$fortIndent|
        (0 NIL)
        6)
```

32.28.5 fortlength

```
-------------------- The fortlength Option --------------------

 Description: the number of characters on a line

 The fortlength option may be followed by an integer in the range
 1 to inclusive. The current setting is 72
```

32.28.6 defvar $fortLength

⟨*initvars*⟩+≡
```
  (defvar |$fortLength| 72 "the number of characters on a line")
```

⟨*fortranfortlength*⟩≡
```
       (|fortlength|
        "the number of characters on a line"
        |interpreter|
        INTEGER
        |$fortLength|
        (1 NIL)
        72)
```

32.28.7 typedecs

```
--------------------- The typedecs Option ---------------------

 Description: print type and dimension lines

 The typedecs option may be followed by any one of the
 following:

 -> on
    off

 The current setting is indicated.
```

32.28.8 defvar $printFortranDecs

⟨*initvars*⟩+≡

```
  (defvar |$printFortranDecs| t "print type and dimension lines")
```

⟨*fortrantypedecs*⟩≡

```
        (|typedecs|
         "print type and dimension lines"
         |interpreter|
         LITERALS
         |$printFortranDecs|
         (|on| |off|)
         |on|)
```

32.28.9 defaulttype

```
------------------- The defaulttype Option --------------------

 Description: default generic type for FORTRAN object

 The defaulttype option may be followed by any one of the
 following:

 -> REAL
    INTEGER
    COMPLEX
    LOGICAL
    CHARACTER

 The current setting is indicated.
```

32.28.10 defvar $defaultFortranType

⟨*initvars*⟩+≡
```
(defvar |$defaultFortranType| 'real "default generic type for FORTRAN object")
```

⟨*fortrandefaulttype*⟩≡
```
(|defaulttype|
 "default generic type for FORTRAN object"
 |interpreter|
 LITERALS
 |$defaultFortranType|
 (REAL INTEGER COMPLEX LOGICAL CHARACTER)
 REAL)
```

32.28.11 precision

```
-------------------- The precision Option ---------------------

 Description: precision of generated FORTRAN objects

 The precision option may be followed by any one of the
 following:

    single
 -> double

 The current setting is indicated.
```

32.28.12 defvar $fortranPrecision

⟨*initvars*⟩+≡
```
(defvar |$fortranPrecision| '|double| "precision of generated FORTRAN objects")
```

⟨*fortranprecision*⟩≡
```
(|precision|
"precision of generated FORTRAN objects"
 |interpreter|
 LITERALS
 |$fortranPrecision|
 (|single| |double|)
 |double|)
```

32.28.13 intrinsic

```
-------------------- The intrinsic Option --------------------

 Description: whether to use INTRINSIC FORTRAN functions

 The intrinsic option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.28.14 defvar $useIntrinsicFunctions

⟨*initvars*⟩+≡

```
  (defvar |$useIntrinsicFunctions| nil
     "whether to use INTRINSIC FORTRAN functions")
```

⟨*fortranintrinsic*⟩≡

```
        (|intrinsic|
         "whether to use INTRINSIC FORTRAN functions"
         |interpreter|
         LITERALS
         |$useIntrinsicFunctions|
         (|on| |off|)
         |off|)
```

32.28.15 explength

```
-------------------- The explength Option --------------------

 Description: character limit for FORTRAN expressions

 The explength option may be followed by an integer in the range
 0 to inclusive. The current setting is 1320
```

32.28.16 defvar $maximumFortranExpressionLength

⟨*initvars*⟩+≡

```
  (defvar |$maximumFortranExpressionLength| 1320
    "character limit for FORTRAN expressions")
```

⟨*fortranexplength*⟩≡

```
        (|explength|
         "character limit for FORTRAN expressions"
         |interpreter|
         INTEGER
         |$maximumFortranExpressionLength|
         (0 NIL)
         1320)
```

32.28.17 segment

```
--------------------- The segment Option ----------------------

 Description: split long FORTRAN expressions

 The segment option may be followed by any one of the following:

 -> on
    off

 The current setting is indicated.
```

32.28.18 defvar $fortranSegment

⟨*initvars*⟩+≡

```
  (defvar |$fortranSegment| t "split long FORTRAN expressions")
```

⟨*fortransegment*⟩≡
```
        (|segment|
         "split long FORTRAN expressions"
         |interpreter|
         LITERALS
         |$fortranSegment|
         (|on| |off|)
         |on|)
```

32.28.19 optlevel

```
-------------------- The optlevel Option --------------------

 Description: FORTRAN optimisation level

 The optlevel option may be followed by an integer in the range
 0 to 2 inclusive. The current setting is 0
```

32.28.20 defvar $fortranOptimizationLevel

⟨*initvars*⟩+≡
```
  (defvar |$fortranOptimizationLevel| 0 "FORTRAN optimisation level")
```

⟨*fortranoptlevel*⟩≡
```
        (|optlevel|
         "FORTRAN optimisation level"
         |interpreter|
         INTEGER
         |$fortranOptimizationLevel|
         (0 2)
         0)
```

32.28.21 startindex

```
-------------------- The startindex Option --------------------

 Description: starting index for FORTRAN arrays

 The startindex option may be followed by an integer in the range
 0 to 1 inclusive. The current setting is 1
```

32.28.22 defvar $fortranArrayStartingIndex

⟨*initvars*⟩+≡
```
(defvar |$fortranArrayStartingIndex| 1 "starting index for FORTRAN arrays")
```

⟨*fortranstartindex*⟩≡
```
        (|startindex|
         "starting index for FORTRAN arrays"
         |interpreter|
         INTEGER
         |$fortranArrayStartingIndex|
         (0 1)
         1)
```

32.28.23 calling

```
             Current Values of  calling  Variables

Variable     Description                              Current Value
-------------------------------------------------------------------
tempfile     set location of temporary data files       /tmp/
directory    set location of generated FORTRAN files    ./
linker       linker arguments (e.g. libraries to search) -lxlf
```

⟨*fortrancalling*⟩≡
```
        (|calling|
        "options for external FORTRAN calls"
        |interpreter|
        TREE
        |novar|
        (
```
⟨*callingtempfile*⟩
⟨*callingdirectory*⟩
⟨*callinglinker*⟩
```
         )
        )
```

tempfile

```
--------------------- The tempfile Option ---------------------

 Description: set location of temporary data files

 )set fortran calling tempfile  is used to tell AXIOM where
 to place intermediate FORTRAN data files . This must be the
 name of a valid existing directory to which you have permission
 to write (including the final slash).

 Syntax:
   )set fortran calling tempfile DIRECTORYNAME

 The current setting is /tmp/
```

32.28.24 defvar $fortranTmpDir

⟨*initvars*⟩+≡

```
  (defvar |$fortranTmpDir| "/tmp/" "set location of temporary data files")
```

⟨*callingtempfile*⟩≡

```
          (|tempfile|
           "set location of temporary data files"
           |interpreter|
           FUNCTION
           |setFortTmpDir|
           (("enter directory name for which you have write-permission"
             DIRECTORY
             |$fortranTmpDir|
             |chkDirectory|
             "/tmp/"))
           NIL)
```

32.28.25 The top level set fortran calling tempfile handler

⟨*defun setFortTmpDir*⟩≡
```
(defun |setFortTmpDir| (arg)
 "The top level set fortran calling tempfile handler"
 (let (mode)
   (declare (special |$fortranTmpDir|))
   (cond
    ((eq arg '|%initialize%|) (setq |$fortranTmpDir| "/tmp/"))
    ((eq arg '|%display%|)
     (if (stringp |$fortranTmpDir|)
      |$fortranTmpDir|
      (pname |$fortranTmpDir|)))
    ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
     (|describeSetFortTmpDir|))
    ((null (setq mode (|validateOutputDirectory| arg)))
     (|sayBrightly|
      `(" Sorry, but your argument(s)" ,@(|bright| arg)
            "is(are) not valid." |%l|))
      (|describeSetFortTmpDir|))
    (t (setq |$fortranTmpDir| mode)))))
```

32.28.26 Validate the output directory

⟨*defun validateOutputDirectory*⟩≡
```
(defun |validateOutputDirectory| (x)
 "Validate the output directory"
 (let ((dirname (car x)))
  (when (and (pathname-directory dirname) (null (probe-file dirname)))
   dirname)))
```

32.28.27 Describe the set fortran calling tempfile

⟨*defun describeSetFortTmpDir*⟩≡
```
(defun |describeSetFortTmpDir| ()
 "Describe the set fortran calling tempfile"
 (declare (special |$fortranTmpDir|))
 (|sayBrightly| (list
  '|%b| ")set fortran calling tempfile"
  '|%d| " is used to tell AXIOM where"
  '|%l| " to place intermediate FORTRAN data files . This must be the "
  '|%l| " name of a valid existing directory to which you have permission "
  '|%l| " to write (including the final slash)."
  '|%l|
  '|%l| " Syntax:"
  '|%l| "   )set fortran calling tempfile DIRECTORYNAME"
  '|%l|
  '|%l| " The current setting is"
  '|%b| |$fortranTmpDir|
  '|%d|)))
```

directory

```
-------------------- The directory Option ---------------------

 Description: set location of generated FORTRAN files

 )set fortran calling directory  is used to tell AXIOM where
 to place generated FORTRAN files. This must be the name
 of a valid existing directory to which you have permission
 to write (including the final slash).

 Syntax:
   )set fortran calling directory DIRECTORYNAME

 The current setting is ./
```

32.28.28 defvar $fortranDirectory

⟨*initvars*⟩+≡
```
(defvar |$fortranDirectory| "./" "set location of generated FORTRAN files")
```

⟨*callingdirectory*⟩≡
```
          (|directory|
           "set location of generated FORTRAN files"
           |interpreter|
           FUNCTION
           |setFortDir|
           (("enter directory name for which you have write-permission"
             DIRECTORY
             |$fortranDirectory|
             |chkDirectory|
             "./"))
           NIL)
```

32.28.29 defun setFortDir

⟨*defun setFortDir*⟩≡
```
(defun |setFortDir| (arg)
 (declare (special |$fortranDirectory|))
 (let (mode)
   (COND
    ((eq arg '|%initialize%|) (setq |$fortranDirectory| "./"))
    ((eq arg '|%display%|)
     (if (stringp |$fortranDirectory|)
      |$fortranDirectory|
      (pname |$fortranDirectory|)))
    ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
      (|describeSetFortDir|))
    ((null (setq mode (|validateOutputDirectory| arg)))
     (|sayBrightly|
      `(" Sorry, but your argument(s)" ,@(|bright| arg)
        "is(are) not valid." |%l|))
     (|describeSetFortDir|))
    (t (setq |$fortranDirectory| mode)))))
```

32.28.30 defun describeSetFortDir

⟨*defun describeSetFortDir*⟩≡
```
(defun |describeSetFortDir| ()
 (declare (special |$fortranDirectory|))
 (|sayBrightly| (list
  '|%b| ")set fortran calling directory"
  '|%d| " is used to tell AXIOM where"
  '|%l| " to place generated FORTRAN files. This must be the name "
  '|%l| " of a valid existing directory to which you have permission "
  '|%l| " to write (including the final slash)."
  '|%l|
  '|%l| " Syntax:"
  '|%l| "   )set fortran calling directory DIRECTORYNAME"
  '|%l|
  '|%l| " The current setting is"
  '|%b| |$fortranDirectory|
  '|%d|)))
```

linker

```
---------------------- The linker Option ----------------------

 Description: linker arguments (e.g. libraries to search)

 )set fortran calling linkerargs is used to pass arguments to
 the linker when using  mkFort  to create functions which call
 Fortran code. For example, it might give a list of libraries
 to be searched, and their locations.
 The string is passed verbatim, so must be the correct syntax for
 the particular linker being used.

 Example: )set fortran calling linker "-lxlf"

 The current setting is -lxlf
```

32.28.31 defvar $fortranLibraries

⟨*initvars*⟩+≡
```
(defvar |$fortranLibraries| "-lxlf"
  "linker arguments (e.g. libraries to search)")
```

⟨*callinglinker*⟩≡
```
          (|linker|
           "linker arguments (e.g. libraries to search)"
           |interpreter|
           FUNCTION
           |setLinkerArgs|
           (("enter linker arguments "
             STRING
             |$fortranLibraries|
             |chkDirectory|
             "-lxlf"))
           NIL
           )
```

32.28.32 defun setLinkerArgs

⟨*defun setLinkerArgs*⟩≡
```
  (defun |setLinkerArgs| (arg)
   (declare (special |$fortranLibraries|))
   (cond
    ((eq arg '|%initialize%|) (setq |$fortranLibraries| "-lxlf"))
    ((eq arg '|%display%|) (|object2String| |$fortranLibraries|))
    ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
      (|describeSetLinkerArgs|))
    ((and (listp arg) (stringp (car arg)))
      (setq |$fortranLibraries| (car arg)))
    (t (|describeSetLinkerArgs|))))
```

32.28.33 defun describeSetLinkerArgs

⟨*defun describeSetLinkerArgs*⟩≡
```
(defun |describeSetLinkerArgs| ()
 (declare (special |$fortranLibraries|))
 (|sayBrightly| (list
  '|%b| ")set fortran calling linkerargs"
  '|%d| " is used to pass arguments to the linker"
  '|%l| " when using "
  '|%b| "mkFort"
  '|%d| " to create functions which call Fortran code."
  '|%l| " For example, it might give a list of libraries to be searched,"
  '|%l| " and their locations."
  '|%l| " The string is passed verbatim, so must be the correct syntax for"
  '|%l| " the particular linker being used."
  '|%l|
  '|%l| " Example: )set fortran calling linker \"-lxlf\""
  '|%l|
  '|%l| " The current setting is"
  '|%b| |$fortranLibraries|
  '|%d|)))
```

32.29 kernel

```
             Current Values of  kernel  Variables

Variable     Description                            Current Value
-------------------------------------------------------------------
warn         warn when re-definition is attempted       off
protect      prevent re-definition of kernel functions  off
```

⟨*kernel*⟩≡
```
    (|kernel|
     "library functions built into the kernel for efficiency"
     |interpreter|
     TREE
     |novar|
     (
  ⟨kernelwarn⟩
  ⟨kernelprotect⟩
     )
    )
```

32.29.1 kernelwarn

```
---------------------- The warn Option ----------------------

 Description: warn when re-definition is attempted

Some AXIOM library functions are compiled into the kernel for
efficiency reasons. To prevent them being re-defined when loaded
from a library they are specially protected. If a user wishes to
know when an attempt is made to re-define such a function, he or
she should issue the command:
        )set kernel warn on
To restore the default behaviour, he or she should issue the
command:
        )set kernel warn off
```

⟨*kernelwarn*⟩≡

```
        (|warn|
         "warn when re-definition is attempted"
         |interpreter|
         FUNCTION
         |protectedSymbolsWarning|
         NIL
         |htSetKernelWarn|)
```

32.29.2 defun protectedSymbolsWarning

⟨*defun protectedSymbolsWarning*⟩≡

```
  (defun |protectedSymbolsWarning| (arg)
   (let (v)
     (cond
      ((eq arg '|%initialize%|) (protected-symbol-warn nil))
      ((eq arg '|%display%|)
       (setq v (protected-symbol-warn t))
       (protected-symbol-warn v)
       (if v "on" "off"))
      ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
       (|describeProtectedSymbolsWarning|))
      (t (protected-symbol-warn (|translateYesNo2TrueFalse| (car arg)))))))
```

32.29.3 defun describeProtectedSymbolsWarning

⟨*defun describeProtectedSymbolsWarning*⟩≡
```
(defun |describeProtectedSymbolsWarning| ()
 (|sayBrightly| (list
  "Some AXIOM library functions are compiled into the kernel for efficiency"
  '|%l| "reasons.  To prevent them being re-defined when loaded from a library"
  '|%l|
  "they are specially protected.  If a user wishes to know when an attempt"
  '|%l|
  "is made to re-define such a function, he or she should issue the command:"
  '|%l| "        )set kernel warn on"
  '|%l| "To restore the default behaviour, he or she should issue the command:"
  '|%l| "        )set kernel warn off")))
```

32.29.4 kernelprotect

```
---------------------- The protect Option ----------------------

 Description: prevent re-definition of kernel functions

Some AXIOM library functions are compiled into the kernel for
efficiency reasons. To prevent them being re-defined when loaded
from a library they are specially protected.  If a user wishes
to re-define these functions, he or she should issue the command:
        )set kernel protect off
To restore the default behaviour, he or she should issue the
command:
        )set kernel protect on
```

⟨*kernelprotect*⟩≡
```
        (|protect|
         "prevent re-definition of kernel functions"
         |interpreter|
         FUNCTION
         |protectSymbols|
         NIL
         |htSetKernelProtect|)
```

32.29.5 defun protectSymbols

⟨*defun protectSymbols*⟩≡
```
(defun |protectSymbols| (arg)
 (let (v)
   (cond
    ((eq arg '|%initialize%|) (protect-symbols  t))
    ((eq arg '|%display%|)
     (setq v (protect-symbols t))
     (protect-symbols v)
     (if v "on" "off"))
    ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
     (|describeProtectSymbols|))
    (t (protect-symbols (|translateYesNo2TrueFalse| (car arg)))))))
```

32.29.6 defun describeProtectSymbols

⟨*defun describeProtectSymbols*⟩≡
```
(defun |describeProtectSymbols| ()
 (|sayBrightly| (list
  "Some AXIOM library functions are compiled into the kernel for efficiency"
  '|%l|
  "reasons.  To prevent them being re-defined when loaded from a library"
  '|%l| "they are specially protected.  If a user wishes to re-define these"
  '|%l| "functions, he or she should issue the command:"
  '|%l| "        )set kernel protect off"
  '|%l|
  "To restore the default behaviour, he or she should issue the command:"
  '|%l| "        )set kernel protect on")))
```

32.30 hyperdoc

```
            Current Values of  hyperdoc  Variables

Variable     Description                            Current Value
-------------------------------------------------------------------
fullscreen   use full screen for this facility          off
mathwidth    screen width for history output            120
```

⟨*hyperdoc*⟩≡
```
    (|hyperdoc|
     "options in using HyperDoc"
     |interpreter|
     TREE
     |novar|
     (
  ⟨hyperdocfullscreen⟩
  ⟨hyperdocmathwidth⟩
     ))
```

32.30.1 fullscreen

```
-------------------- The fullscreen Option --------------------

 Description: use full screen for this facility

 The fullscreen option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.30.2 defvar $fullScreenSysVars

⟨*initvars*⟩+≡
```
  (defvar |$fullScreenSysVars| nil "use full screen for this facility")
```

⟨*hyperdocfullscreen*⟩≡
```
       (|fullscreen|
        "use full screen for this facility"
        |interpreter|
        LITERALS
        |$fullScreenSysVars|
        (|on| |off|)
        |off|)
```

32.30.3 mathwidth

```
-------------------- The mathwidth Option ---------------------

 Description: screen width for history output

 The mathwidth option may be followed by an integer in the range
 0 to inclusive. The current setting is 120
```

32.30.4 defvar $historyDisplayWidth

⟨*initvars*⟩+≡
```
  (defvar |$historyDisplayWidth| 120 "screen width for history output")
```

⟨*hyperdocmathwidth*⟩≡
```
       (|mathwidth|
        "screen width for history output"
        |interpreter|
        INTEGER
        |$historyDisplayWidth|
        (0 NIL)
        120)
```

32.31 help

```
              Current Values of  help  Variables

Variable     Description                            Current Value
-------------------------------------------------------------------
fullscreen   use fullscreen facility, if possible       off
```

⟨*help*⟩≡
```
    (|help|
     "view and set some help options"
     |interpreter|
     TREE
     |novar|
     (
  ⟨helpfullscreen⟩
     ))
```

32.31.1 fullscreen

```
-------------------- The fullscreen Option --------------------

 Description: use fullscreen facility, if possible

 The fullscreen option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.31.2 defvar $useFullScreenHelp

⟨*initvars*⟩+≡
```
  (defvar |$useFullScreenHelp| nil "use fullscreen facility, if possible")
```

⟨*helpfullscreen*⟩≡
```
      (|fullscreen|
       "use fullscreen facility, if possible"
       |interpreter|
       LITERALS
       |$useFullScreenHelp|
       (|on| |off|)
       |off|)
```

32.32 history

```
--------------------- The history Option ----------------------

 Description: save workspace values in a history file

 The history option may be followed by any one of the
 following:

 -> on
    off

 The current setting is indicated.
```

32.32.1 defvar $HiFiAccess

⟨*initvars*⟩+≡
```
  (defvar |$HiFiAccess| t "save workspace values in a history file")
```

⟨*history*⟩≡
```
    (|history|
     "save workspace values in a history file"
     |interpreter|
     LITERALS
     |$HiFiAccess|
     (|on| |off|)
     |on|)
```

32.33 messages

```
            Current Values of  messages  Variables

Variable     Description                              Current Value
-------------------------------------------------------------------
autoload     print file auto-load messages              off
bottomup     display bottom up modemap selection        off
coercion     display datatype coercion messages         off
dropmap      display old map defn when replaced         off
expose       warning for unexposed functions            off
file         print msgs also to SPADMSG LISTING         off
frame        display messages about frames              off
highlighting use highlighting in system messages        off
instant      present instantiation summary              off
insteach     present instantiation info                 off
interponly   say when function code is interpreted      on
number       display message number with message        off
prompt       set type of input prompt to display        step
selection    display function selection msgs            off
set          show )set setting after assignment         off
startup      display messages on start-up               off
summary      print statistics after computation         off
testing      print system testing header                off
time         print timings after computation            off
type         print type after computation               on
void         print Void value when it occurs            off
any          print the internal type of objects of domain Any on
naglink      show NAGLink messages                      on
```

⟨*messages*⟩≡

```
    (|messages|
     "show messages for various system features"
     |interpreter|
     TREE
     |novar|
     (
```

⟨*messagesany*⟩
⟨*messagesautoload*⟩
⟨*messagesbottomup*⟩
⟨*messagescoercion*⟩
⟨*messagesdropmap*⟩
⟨*messagesexpose*⟩
⟨*messagesfile*⟩
⟨*messagesframe*⟩
⟨*messageshighlighting*⟩
⟨*messagesinstant*⟩
⟨*messagesinsteach*⟩

⟨*messagesinterponly*⟩
⟨*messagesnaglink*⟩
⟨*messagesnumber*⟩
⟨*messagesprompt*⟩
⟨*messagesselection*⟩
⟨*messagesset*⟩
⟨*messagesstartup*⟩
⟨*messagessummary*⟩
⟨*messagestesting*⟩
⟨*messagestime*⟩
⟨*messagestype*⟩
⟨*messagesvoid*⟩
))

32.33.1 any

```
---------------------- The any Option ------------------------

 Description: print the internal type of objects of domain Any

 The any option may be followed by any one of the following:

 -> on
    off

 The current setting is indicated.
```

32.33.2 defvar $printAnyIfTrue

⟨*initvars*⟩+≡
```
(defvar |$printAnyIfTrue| t
   "print the internal type of objects of domain Any")
```

⟨*messagesany*⟩≡
```
       (|any|
        "print the internal type of objects of domain Any"
        |interpreter|
        LITERALS
        |$printAnyIfTrue|
        (|on| |off|)
        |on|)
```

32.33.3 autoload

```
--------------------- The autoload Option ---------------------

 Description: print file auto-load messages
```

32.33.4 defvar $printLoadMsgs

⟨*initvars*⟩+≡
```
  (defvar |$printLoadMsgs| t "print file auto-load messages")
```

⟨*messagesautoload*⟩≡
```
       (|autoload|
        "print file auto-load messages"
        |interpreter|
        LITERALS
        |$printLoadMsgs|
        (|on| |off|)
        |on|)
```

32.33.5 bottomup

```
--------------------- The bottomup Option ---------------------

 Description: display bottom up modemap selection

 The bottomup option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.33.6 defvar $reportBottomUpFlag

⟨*initvars*⟩+≡
```
  (defvar |$reportBottomUpFlag| nil "display bottom up modemap selection")
```

⟨*messagesbottomup*⟩≡
```
       (|bottomup|
        "display bottom up modemap selection"
        |development|
        LITERALS
        |$reportBottomUpFlag|
        (|on| |off|)
        |off|)
```

32.33.7 coercion

```
--------------------- The coercion Option ---------------------

 Description: display datatype coercion messages

 The coercion option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.33.8 defvar $reportCoerceIfTrue

⟨*initvars*⟩+≡
```
  (defvar |$reportCoerceIfTrue| nil "display datatype coercion messages")
```

⟨*messagescoercion*⟩≡
```
       (|coercion|
        "display datatype coercion messages"
        |development|
        LITERALS
        |$reportCoerceIfTrue|
        (|on| |off|)
        |off|)
```

32.33.9 dropmap

```
--------------------- The dropmap Option ---------------------

 Description: display old map defn when replaced

 The dropmap option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.33.10 defvar $displayDroppedMap

⟨*initvars*⟩+≡

```
  (defvar |$displayDroppedMap| nil "display old map defn when replaced")
```

⟨*messagesdropmap*⟩≡

```
        (|dropmap|
         "display old map defn when replaced"
         |interpreter|
         LITERALS
         |$displayDroppedMap|
         (|on| |off|)
         |off|)
```

32.33.11 expose

```
---------------------- The expose Option ----------------------

 Description: warning for unexposed functions

 The expose option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.33.12 defvar $giveExposureWarning

⟨*initvars*⟩+≡
```
  (defvar |$giveExposureWarning| nil "warning for unexposed functions")
```

⟨*messagesexpose*⟩≡
```
        (|expose|
         "warning for unexposed functions"
         |interpreter|
         LITERALS
         |$giveExposureWarning|
         (|on| |off|)
         |off|)
```

32.33.13 file

```
---------------------- The file Option ----------------------

 Description: print msgs also to SPADMSG LISTING

 The file option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.33.14 defvar $printMsgsToFile

⟨*initvars*⟩+≡

```
  (defvar |$printMsgsToFile| nil "print msgs also to SPADMSG LISTING")
```

⟨*messagesfile*⟩≡

```
       (|file|
        "print msgs also to SPADMSG LISTING"
        |development|
        LITERALS
        |$printMsgsToFile|
        (|on| |off|)
        |off|)
```

32.33.15 frame

```
---------------------- The frame Option -----------------------

 Description: display messages about frames

 The frame option may be followed by any one of the following:

    on
 -> off

 The current setting is indicated.
```

32.33.16 defvar $frameMessages

⟨*initvars*⟩+≡
```
  (defvar |$frameMessages| nil "display messages about frames")
```

⟨*messagesframe*⟩≡
```
        (|frame|
         "display messages about frames"
         |interpreter|
         LITERALS
         |$frameMessages|
         (|on| |off|)
         |off|)
```

32.33.17 highlighting

```
------------------- The highlighting Option -------------------

 Description: use highlighting in system messages

 The highlighting option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.33.18 defvar $highlightAllowed

⟨*initvars*⟩+≡

```
  (defvar |$highlightAllowed| nil "use highlighting in system messages")
```

⟨*messageshighlighting*⟩≡

```
        (|highlighting|
         "use highlighting in system messages"
         |interpreter|
         LITERALS
         |$highlightAllowed|
         (|on| |off|)
         |off|)
```

32.33.19 instant

```
--------------------- The instant Option ----------------------

 Description: present instantiation summary

 The instant option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.33.20 defvar $reportInstantiations

⟨*initvars*⟩+≡
```
  (defvar |$reportInstantiations| nil "present instantiation summary")
```

⟨*messagesinstant*⟩≡
```
        (|instant|
         "present instantiation summary"
         |development|
         LITERALS
         |$reportInstantiations|
         (|on| |off|)
         |off|)
```

32.33.21 insteach

```
---------------------- The insteach Option ----------------------

 Description: present instantiation info

 The insteach option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.33.22 defvar $reportEachInstantiation—

⟨*initvars*⟩+≡

```
  (defvar |$reportEachInstantiation| nil "present instantiation info")
```

⟨*messagesinsteach*⟩≡

```
        (|insteach|
         "present instantiation info"
         |development|
         LITERALS
         |$reportEachInstantiation|
         (|on| |off|)
         |off|)
```

32.33.23 interponly

```
-------------------- The interponly Option --------------------

 Description: say when function code is interpreted

 The interponly option may be followed by any one of the
 following:

 -> on
    off

 The current setting is indicated.
```

32.33.24 defvar $reportInterpOnly

⟨*initvars*⟩+≡

```
  (defvar |$reportInterpOnly| t "say when function code is interpreted")
```

⟨*messagesinterponly*⟩≡

```
        (|interponly|
         "say when function code is interpreted"
         |interpreter|
         LITERALS
         |$reportInterpOnly|
         (|on| |off|)
         |on|)
```

32.33.25 naglink

```
---------------------- The naglink Option ----------------------

 Description: show NAGLink messages

 The naglink option may be followed by any one of the
 following:

 -> on
    off

 The current setting is indicated.
```

32.33.26 defvar $nagMessages

⟨*initvars*⟩+≡

```
  (defvar |$nagMessages| t "show NAGLink messages")
```

⟨*messagesnaglink*⟩≡

```
        (|naglink|
         "show NAGLink messages"
         |interpreter|
         LITERALS
         |$nagMessages|
         (|on| |off|)
         |on|)
```

32.33.27 number

```
---------------------- The number Option ----------------------

 Description: display message number with message

 The number option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.33.28 defvar $displayMsgNumber

⟨*initvars*⟩+≡

```
  (defvar |$displayMsgNumber| nil "display message number with message")
```

⟨*messagesnumber*⟩≡

```
        (|number|
         "display message number with message"
         |interpreter|
         LITERALS
         |$displayMsgNumber|
         (|on| |off|)
         |off|)
```

32.33.29 prompt

```
---------------------- The prompt Option ----------------------

 Description: set type of input prompt to display

 The prompt option may be followed by any one of the following:

    none
    frame
    plain
 -> step
    verbose

 The current setting is indicated.
```

32.33.30 defvar $inputPromptType

⟨*initvars*⟩+≡
```
(defvar |$inputPromptType| '|step| "set type of input prompt to display")
```

⟨*messagesprompt*⟩≡
```
(|prompt|
 "set type of input prompt to display"
 |interpreter|
 LITERALS
 |$inputPromptType|
 (|none| |frame| |plain| |step| |verbose|)
 |step|)
```

32.33.31 selection

```
-------------------- The selection Option ---------------------

 Description: display function selection msgs

 The selection option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

TPDHERE: This is a duplicate of)set mes bot on because both use the `$reportBottomUpFlag` flag

⟨*messagesselection*⟩≡
```
(|selection|
 "display function selection msgs"
 |interpreter|
 LITERALS
 |$reportBottomUpFlag|
 (|on| |off|)
 |off|)
```

32.33.32 set

```
---------------------- The set Option -----------------------

 Description: show )set setting after assignment

 The set option may be followed by any one of the following:

    on
 -> off

 The current setting is indicated.
```

32.33.33 defvar $displaySetValue

⟨*initvars*⟩+≡
```
  (defvar |$displaySetValue| nil "show )set setting after assignment")
```

⟨*messagesset*⟩≡
```
        (|set|
         "show )set setting after assignment"
         |interpreter|
         LITERALS
         |$displaySetValue|
         (|on| |off|)
         |off|)
```

32.33.34 startup

```
--------------------- The startup Option ---------------------

 Description: display messages on start-up

 The startup option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.33.35 defvar $displayStartMsgs

⟨*initvars*⟩+≡

```
  (defvar |$displayStartMsgs| t "display messages on start-up")
```

⟨*messagesstartup*⟩≡

```
        (|startup|
         "display messages on start-up"
         |interpreter|
         LITERALS
         |$displayStartMsgs|
         (|on| |off|)
         |on|)
```

32.33.36 summary

```
--------------------- The summary Option ----------------------

 Description: print statistics after computation

 The summary option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.33.37 defvar $printStatisticsSummaryIfTrue

⟨*initvars*⟩+≡

```
  (defvar |$printStatisticsSummaryIfTrue| nil
       "print statistics after computation")
```

⟨*messagessummary*⟩≡

```
       (|summary|
        "print statistics after computation"
        |interpreter|
        LITERALS
        |$printStatisticsSummaryIfTrue|
        (|on| |off|)
        |off|)
```

32.33.38 testing

```
--------------------- The testing Option ----------------------

 Description: print system testing header

 The testing option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.33.39 defvar $testingSystem

⟨*initvars*⟩+≡
```
  (defvar |$testingSystem| nil "print system testing header")
```

⟨*messagestesting*⟩≡
```
       (|testing|
        "print system testing header"
        |development|
        LITERALS
        |$testingSystem|
        (|on| |off|)
        |off|)
```

32.33.40 time

```
---------------------- The time Option ----------------------

 Description: print timings after computation

 The time option may be followed by any one of the following:

    on
 -> off
    long

 The current setting is indicated.
```

32.33.41 defvar $printTimeIfTrue

⟨*initvars*⟩+≡
```
  (defvar |$printTimeIfTrue| nil "print timings after computation")
```

⟨*messagestime*⟩≡
```
       (|time|
        "print timings after computation"
        |interpreter|
        LITERALS
        |$printTimeIfTrue|
        (|on| |off| |long|)
        |off|)
```

32.33.42 type

```
---------------------- The type Option ----------------------

 Description: print type after computation

 The type option may be followed by any one of the following:

 -> on
    off

 The current setting is indicated.
```

32.33.43 defvar $printTypeIfTrue

⟨*initvars*⟩+≡
```
  (defvar |$printTypeIfTrue| t "print type after computation")
```

⟨*messagestype*⟩≡
```
        (|type|
         "print type after computation"
         |interpreter|
         LITERALS
         |$printTypeIfTrue|
         (|on| |off|)
         |on|)
```

32.33.44 void

```
---------------------- The void Option ----------------------

 Description: print Void value when it occurs

 The void option may be followed by any one of the following:

    on
 -> off

 The current setting is indicated.
```

32.33.45 defvar $printVoidIfTrue

⟨*initvars*⟩+≡

```
  (defvar |$printVoidIfTrue| nil "print Void value when it occurs")
```

⟨*messagesvoid*⟩≡

```
        (|void|
         "print Void value when it occurs"
         |interpreter|
         LITERALS
         |$printVoidIfTrue|
         (|on| |off|)
         |off|)
```

32.34 naglink

```
              Current Values of  naglink  Variables

Variable     Description                              Current Value
-------------------------------------------------------------------
host         internet address of host for NAGLink       localhost
persistence  number of (fortran) functions to remember  1
messages     show NAGLink messages                      on
double       enforce DOUBLE PRECISION ASPs              on
```

⟨*naglink*⟩≡

```
    (|naglink|
     "options for NAGLink"
     |interpreter|
     TREE
     |novar|
     (
  ⟨naglinkhost⟩
  ⟨naglinkpersistence⟩
  ⟨naglinkmessages⟩
  ⟨naglinkdouble⟩
     ))
```

32.34.1 host

```
---------------------- The host Option ----------------------

 Description: internet address of host for NAGLink

 )set naglink host is used to tell  AXIOM which  host to contact
 for a NAGLink request. An Internet address should be supplied.
 The host specified must be running the NAGLink daemon.

 The current setting is localhost
```

32.34.2 defvar $nagHost

⟨*initvars*⟩+≡

```
  (defvar |$nagHost| "localhost" "internet address of host for NAGLink")
```

⟨*naglinkhost*⟩≡
```
       (|host|
        "internet address of host for NAGLink"
        |interpreter|
        FUNCTION
        |setNagHost|
        (("enter host name"
          DIRECTORY
          |$nagHost|
          |chkDirectory|
          "localhost"))
        NIL)
```

32.34.3 defun setNagHost

⟨*defun setNagHost*⟩≡
```
  (defun |setNagHost| (|arg|)
   (declare (special |$nagHost|))
   (cond
    ((eq |arg| '|%initialize%|) (setq |$nagHost| "localhost"))
    ((eq |arg| '|%display%|) (|object2String| |$nagHost|))
    ((or (null |arg|) (eq |arg| '|%describe%|) (eq (car |arg|) '?))
     (|describeSetNagHost|))
    (t (setq |$nagHost| (|object2String| |arg|)))))
```

32.34.4 defun describeSetNagHost

⟨*defun describeSetNagHost*⟩≡
```
  (defun |describeSetNagHost| ()
   (declare (special |$nagHost|))
   (|sayBrightly| (list
    '|%b| ")set naglink host"
    '|%d| "is used to tell  AXIOM which  host to contact for"
    '|%l| " a NAGLink request. An Internet address should be supplied. The host"
    '|%l| " specified must be running the NAGLink daemon."
    '|%l|
    '|%l| " The current setting is"
    '|%b| |$nagHost|
    '|%d|)))
```

32.34.5 persistence

```
------------------- The persistence Option --------------------

 Description: number of (fortran) functions to remember

 )set naglink persistence is used to tell  the  nagd  daemon how
  many ASP source and object files to keep around in case you
  reuse them. This helps to avoid needless recompilations. The
  number specified should be a non-negative integer.

 The current setting is 1
```

32.34.6 defvar $fortPersistence

⟨*initvars*⟩+≡

```
  (defvar |$fortPersistence| 1 "number of (fortran) functions to remember")
```

⟨*naglinkpersistence*⟩≡

```
       (|persistence|
        "number of (fortran) functions to remember"
        |interpreter|
        FUNCTION
        |setFortPers|
        (("Requested remote storage (for asps):"
         INTEGER
         |$fortPersistence|
         (0 NIL)
         10))
        NIL)
```

32.34.7 defun setFortPers

⟨*defun setFortPers*⟩≡
```
(defun |setFortPers| (arg)
 (let (n)
 (declare (special |$fortPersistence|))
   (cond
    ((eq arg '|%initialize%|) (setq |$fortPersistence| 1))
    ((eq arg '|%display%|) |$fortPersistence|)
    ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
     (|describeFortPersistence|))
    (t
     (setq n (car arg))
     (cond
      ((or (null (fixp n)) (minusp n))
       (|sayMessage|
        `("Your value of" ,@(|bright| n) "is invalid because ..."))
        (|describeFortPersistence|)
        (|terminateSystemCommand|))
      (t (setq |$fortPersistence| (car arg))))))))
```

32.34.8 defun describeFortPersistence

⟨*defun describeFortPersistence*⟩≡
```
(defun |describeFortPersistence| ()
 (declare (special |$fortPersistence|))
 (|sayBrightly| (list
 '|%b| ")set naglink persistence"
 '|%d| "is used to tell  the "
 '|%b| '|nagd|
 '|%d| '| daemon how  many ASP|
 '|%l|
 " source and object files to keep around in case you reuse them. This helps"
 '|%l| " to avoid needless recompilations. The number specified should be a "
 '|%l| " non-negative integer."
 '|%l|
 '|%l| " The current setting is"
 '|%b| |$fortPersistence|
 '|%d|)))
```

32.34.9 messages

```
--------------------- The messages Option ---------------------

 Description: show NAGLink messages

 The messages option may be followed by any one of the
 following:

 -> on
    off

 The current setting is indicated.
```

TPDHERE: this is the same as)set nag mes on

⟨*naglinkmessages*⟩≡

```
(|messages|
 "show NAGLink messages"
 |interpreter|
 LITERALS
 |$nagMessages|
 (|on| |off|)
  |on|)
```

32.34.10 double

```
---------------------- The double Option ----------------------

 Description: enforce DOUBLE PRECISION ASPs

 The double option may be followed by any one of the following:

 -> on
    off

 The current setting is indicated.
```

32.34.11 defvar $nagEnforceDouble

⟨*initvars*⟩+≡

```
(defvar |$nagEnforceDouble| t "enforce DOUBLE PRECISION ASPs")
```

⟨*naglinkdouble*⟩≡

```
(|double|
 "enforce DOUBLE PRECISION ASPs"
 |interpreter|
 LITERALS
 |$nagEnforceDouble|
 (|on| |off|)
 |on|)
```

32.35 output

The result of the **)set output** command is:

```
Variable     Description                              Current Value
-------------------------------------------------------------------
abbreviate   abbreviate type names                      off
algebra      display output in algebraic form           On:CONSOLE
characters   choose special output character set        plain
fortran      create output in FORTRAN format            Off:CONSOLE
fraction     how fractions are formatted                vertical
length       line length of output displays             77
mathml       create output in MathML style              Off:CONSOLE
openmath     create output in OpenMath style            Off:CONSOLE
script       display output in SCRIPT formula format    Off:CONSOLE
scripts      show subscripts,... linearly               off
showeditor   view output of )show in editor             off
tex          create output in TeX style                 Off:CONSOLE
```

Since the output option has a bunch of sub-options each suboption is defined within the output structure.

⟨*output*⟩≡

```
    (|output|
     "view and set some output options"
     |interpreter|
     TREE
     |novar|
      (
```

⟨*outputabbreviate*⟩
⟨*outputalgebra*⟩
⟨*outputcharacters*⟩
⟨*outputfortran*⟩
⟨*outputfraction*⟩
⟨*outputlength*⟩
⟨*outputmathml*⟩
⟨*outputopenmath*⟩
⟨*outputscript*⟩
⟨*outputscripts*⟩
⟨*outputshoweditor*⟩
⟨*outputtex*⟩

```
     ))
```

32.35.1 abbreviate

```
-------------------- The abbreviate Option --------------------

 Description: abbreviate type names

 The abbreviate option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.35.2 defvar $abbreviateTypes

⟨*initvars*⟩+≡

```
  (defvar |$abbreviateTypes| nil "abbreviate type names")
```

⟨*outputabbreviate*⟩≡

```
        (|abbreviate|
         "abbreviate type names"
         |interpreter|
         LITERALS
         |$abbreviateTypes|
         (|on| |off|)
         |off|)
```

32.35.3 algebra

```
--------------------- The algebra Option ---------------------

 Description: display output in algebraic form

 )set output algebra is used to tell AXIOM to turn algebra-style
  output printing on and off, and where to place the output.  By
  default, the destination for the output is the screen but
  printing is turned off.

Syntax:   )set output algebra <arg>
    where arg can be one of
  on          turn algebra printing on (default state)
  off         turn algebra printing off
  console     send algebra output to screen (default state)
  fp<.fe>     send algebra output to file with file prefix fp
              and file extension .fe. If not given,
              .fe defaults to .spout.

If you wish to send the output to a file, you may need to issue
this command twice: once with on and once with the file name.
For example, to send algebra output to the file polymer.spout,
issue the two commands

  )set output algebra on
  )set output algebra polymer

The output is placed in the directory from which you invoked
AXIOM or the one you set with the )cd system command.
The current setting is:  On:CONSOLE
```

32.35.4 defvar $algebraFormat

⟨*initvars*⟩+≡

```
(defvar |$algebraFormat| t "display output in algebraic form")
```

32.35.5 defvar $algebraOutputFile

⟨*initvars*⟩+≡

```
(defvar |$algebraOutputFile| "CONSOLE"
  "where algebra printing goes (enter {\em console} or a pathname)?")
```

⟨*outputalgebra*⟩≡

```
(|algebra|
 "display output in algebraic form"
 |interpreter|
 FUNCTION
 |setOutputAlgebra|
 (("display output in algebraic form"
   LITERALS
   |$algebraFormat|
   (|off| |on|)
   |on|)
  (break $algebraFormat)
  ("where algebra printing goes (enter {\em console} or a pathname)?"
   FILENAME
   |$algebraOutputFile|
   |chkOutputFileName|
   "console"))
 NIL)
```

32.35.6 defun setOutputAlgebra

⟨*defun setOutputAlgebra*⟩≡

```
(defun |setOutputAlgebra| (arg)
 (let (label tmp1 tmp2 ptype fn ft fm filename teststream)
 (declare (special |$algebraOutputStream| |$algebraOutputFile|
  |$algebraFormat|))
   (cond
    ((eq arg '|%initialize%|)
     (setq |$algebraOutputStream|
      (defiostream '((mode . output) (device . console)) 255 0))
     (setq |$algebraOutputFile| "CONSOLE")
     (setq |$algebraFormat| t))
    ((eq arg '|%display%|)
     (if |$algebraFormat|
      (setq label "On:")
      (setq label "Off:"))
     (strconc label |$algebraOutputFile|))
    ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
     (|describeSetOutputAlgebra|))
    (t
     (cond
      ((and (pairp arg)
            (eq (qcdr arg) nil)
            (progn (setq fn (qcar arg)) t)
            (|member| fn '(y n ye yes no o on of off console
                    |y| |n| |ye| |yes| |no| |o| |on| |of| |off| |console|)))
       '|ok|)
      (t (setq arg (list fn '|spout|))))
     (cond
      ((and (pairp arg)
            (eq (qcdr arg) nil)
            (progn (setq fn (qcar arg)) t))
       (cond
        ((|member| (upcase fn) '(y n ye o of))
         (|sayKeyedMsg| 's2iv0002 '(|algebra| |algebra|)))
        ((|member| (upcase fn) '(no off)) (setq |$algebraFormat| nil))
        ((|member| (upcase fn) '(yes on)) (setq |$algebraFormat| t))
        ((eq (upcase fn) 'console)
         (shut |$algebraOutputStream|)
         (setq |$algebraOutputStream|
          (defiostream '((mode . output) (device . console)) 255 0))
         (setq |$algebraOutputFile| "CONSOLE"))))
      ((or
        (and (pairp arg)
             (progn
```

```
            (setq fn (qcar arg))
            (setq tmp1 (qcdr arg))
            (and (pairp tmp1)
                 (eq (qcdr tmp1) nil)
                 (progn (setq ft (qcar tmp1)) t))))
      (and (pairp arg)
           (progn (setq fn (qcar arg))
                  (setq tmp1 (qcdr arg))
                  (and (pairp tmp1)
                       (progn (setq ft (qcar tmp1))
                              (setq tmp2 (qcdr tmp1))
                              (and (pairp tmp2)
                                   (eq (qcdr tmp2) nil)
                                   (progn
                                    (setq fm (qcar tmp2))
                                    t)))))))
     (when (setq ptype (|pathnameType| fn))
      (setq fn (strconc (|pathnameDirectory| fn) (|pathnameName| fn)))
      (setq ft ptype))
     (unless fm (setq fm 'a))
     (setq filename ($filep fn ft fm))
     (cond
      ((null filename)
       (|sayKeyedMsg| 's2iv0003 (list fn ft fm)))
      ((setq teststream (make-outstream filename 255 0))
       (shut |$algebraOutputStream|)
       (setq |$algebraOutputStream| teststream)
       (setq |$algebraOutputFile| (|object2String| filename))
       (|sayKeyedMsg| 's2iv0004 (list "Algebra" |$algebraOutputFile|)))
      (t (|sayKeyedMsg| 's2iv0003 (list fn ft fm)))))
    (t
     (|sayKeyedMsg| 's2iv0005 nil)
     (|describeSetOutputAlgebra|)))))))
```

32.35.7 defun describeSetOutputAlgebra

⟨*defun describeSetOutputAlgebra*⟩≡

```
(defun |describeSetOutputAlgebra| ()
 (|sayBrightly| (list
 '|%b| ")set output algebra"
 '|%d| "is used to tell AXIOM to turn algebra-style output"
 '|%l| "printing on and off, and where to place the output.  By default, the"
 '|%l| "destination for the output is the screen but printing is turned off."
 '|%l|
 '|%l| "Syntax:   )set output algebra <arg>"
 '|%l| "    where arg can be one of"
 '|%l| "  on          turn algebra printing on (default state)"
 '|%l| "  off         turn algebra printing off"
 '|%l| "  console     send algebra output to screen (default state)"
 '|%l| "  fp<.fe>     send algebra output to file with file prefix fp"
 '|%l|
  "              and file extension .fe. If not given, .fe defaults to .spout."
 '|%l|
 '|%l|
 "If you wish to send the output to a file, you may need to issue this command"
 '|%l| "twice: once with"
 '|%b| "on"
 '|%d| "and once with the file name. For example, to send"
 '|%l| "algebra output to the file"
 '|%b| "polymer.spout,"
 '|%d| "issue the two commands"
 '|%l|
 '|%l| "  )set output algebra on"
 '|%l| "  )set output algebra polymer"
 '|%l|
 '|%l| "The output is placed in the directory from which you invoked AXIOM or"
 '|%l| "the one you set with the )cd system command."
 '|%l| "The current setting is: "
 '|%b| (|setOutputAlgebra| '|%display%|)
 '|%d|)))
```

32.35.8 characters

```
-------------------- The characters Option --------------------

 Description: choose special output character set


 The characters option may be followed by any one of the
 following:

    default
 -> plain

 The current setting is indicated.  This
 option  determines the special characters used for algebraic
 output. This is what the current choice of special characters
 looks like:
   ulc is shown as +          urc is shown as +
   llc is shown as +          lrc is shown as +
   vbar is shown as |         hbar is shown as -
   quad is shown as ?         lbrk is shown as [
   rbrk is shown as ]         lbrc is shown as {
   rbrc is shown as }         ttee is shown as +
   btee is shown as +         rtee is shown as +
   ltee is shown as +         ctee is shown as +
   bslash is shown as \
```

⟨*outputcharacters*⟩≡

```
(|characters|
 "choose special output character set"
 |interpreter|
 FUNCTION
 |setOutputCharacters|
 NIL
 |htSetOutputCharacters|)
```

32.35.9 defun setOutputCharacters

⟨*defun setOutputCharacters*⟩≡

```
(defun |setOutputCharacters| (arg)
 (let (current char s l fn)
 (declare (special |$specialCharacters| |$plainRTspecialCharacters|
   |$RTspecialCharacters| |$specialCharacterAlist|))
    (if (eq arg '|%initialize%|)
     (setq |$specialCharacters| |$plainRTspecialCharacters|)
     (progn
      (setq current
       (cond
        ((eq |$specialCharacters| |$RTspecialCharacters|) "default")
        ((eq |$specialCharacters| |$plainRTspecialCharacters|) "plain")
        (t "unknown")))
      (cond
       ((eq arg '|%display%|) current)
       ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
        (|sayMessage|
         `(" The" ,@(|bright| "characters")
           "option may be followed by any one of the following:"))
        (dolist (name '("default" "plain"))
         (if (string= (string current) name)
          (|sayBrightly| `(" ->" ,@(|bright| name)))
          (|sayBrightly|  (list "    " name))))
        (terpri)
        (|sayBrightly|
" The current setting is indicated within the list.  This option determines ")
        (|sayBrightly|
        " the special characters used for algebraic output.  This is what the")
        (|sayBrightly|
         " current choice of special characters looks like:")
        (do ((t1 |$specialCharacterAlist| (CDR t1)) (t2 nil))
           ((or (atom t1)
                (progn (setq t2 (car t1)) nil)
                (progn (progn (setq char (car t2)) t2) nil)) nil)
         (setq s
          (strconc "   " (pname char) " is shown as "
           (pname (|specialChar| char))))
          (setq l (cons s l)))
        (|sayAsManyPerLineAsPossible| (reverse l)))
       ((and (pairp arg)
             (eq (qcdr arg) NIL)
             (progn (spadlet fn (qcar arg)) t)
             (setq fn (downcase fn)))
        (cond
```

```
          ((eq fn '|default|)
           (setq |$specialCharacters| |$RTspecialCharacters|))
          ((eq fn '|plain|)
           (setq |$specialCharacters| |$plainRTspecialCharacters|))
          (t (|setOutputCharacters| nil))))
        (t (|setOutputCharacters| nil)))))))
```

32.35.10 fortran

```
---------------------- The fortran Option ----------------------

 Description: create output in FORTRAN format

 )set output fortran is used to tell AXIOM to turn FORTRAN-style
  output printing on and off, and where to place the output.  By
  default, the destination for the output is the screen but
  printing is turned off.

Also See: )set fortran

Syntax:   )set output fortran <arg>
    where arg can be one of
  on          turn FORTRAN printing on
  off         turn FORTRAN printing off (default state)
  console     send FORTRAN output to screen (default state)
  fp<.fe>     send FORTRAN output to file with file prefix
              fp and file extension .fe. If not given,
              .fe defaults to .sfort.

If you wish to send the output to a file, you must issue
this command twice: once with on and once with the file name.
For example, to send FORTRAN output to the file polymer.sfort,
 issue the two commands

  )set output fortran on
  )set output fortran polymer

The output is placed in the directory from which you invoked
AXIOM or the one you set with the )cd system command.
The current setting is:  Off:CONSOLE
```

32.35.11 defvar $fortranFormat

⟨*initvars*⟩+≡

```
  (defvar |$fortranFormat| nil "create output in FORTRAN format")
```

32.35.12 defvar $HiFiAccess

⟨*initvars*⟩+≡

```
(defvar |$fortranOutputFile| "CONSOLE"
   "where FORTRAN output goes (enter {\em console} or a a pathname)")
```

⟨*outputfortran*⟩≡

```
       (|fortran|
        "create output in FORTRAN format"
        |interpreter|
        FUNCTION
        |setOutputFortran|
        (("create output in FORTRAN format"
          LITERALS
          |$fortranFormat|
          (|off| |on|)
          |off|)
         (|break| |$fortranFormat|)
         ("where FORTRAN output goes (enter {\em console} or a a pathname)"
          FILENAME
          |$fortranOutputFile|
          |chkOutputFileName|
          "console"))
        NIL)
```

32.35.13 defun setOutputFortran

⟨*defun setOutputFortran*⟩≡

```
(defun |setOutputFortran| (arg)
 (let (label APPEND quiet tmp1 tmp2 ptype fn ft fm filename teststream)
 (declare (special |$fortranOutputStream| |$fortranOutputFile|
    |$fortranFormat|))
  (cond
   ((eq arg '|%initialize%|)
    (setq |$fortranOutputStream|
     (defiostream '((mode . output) (device . console)) 255 0))
    (setq |$fortranOutputFile| "CONSOLE")
    (setq |$fortranFormat| nil))
   ((eq arg '|%display%|)
    (if |$fortranFormat|
     (setq label "On:")
     (setq label "Off:"))
    (strconc label |$fortranOutputFile|))
   ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
    (|describeSetOutputFortran|))
   (t
    (DO ()
        ((null (and (listp arg)
                    (|member| (upcase (car arg)) '(append quiet))))
      nil)
     (cond
      ((eq (upcase (car arg)) 'append) (setq append t))
      ((eq (upcase (car arg)) 'quiet) (setq quiet t))
      (t nil))
     (setq arg (cdr arg)))
    (cond
     ((and (pairp arg)
           (eq (qcdr arg) nil)
           (progn (setq fn (qcar arg)) t)
           (|member| fn '(Y N YE YES NO O ON OF OFF CONSOLE
             |y| |n| |ye| |yes| |no| |o| |on| |of| |off| |console|)))
      '|ok|)
     (t (setq arg (list fn '|sfort|))))
    (cond
     ((and (pairp arg) (eq (qcdr arg) nil) (progn (setq fn (qcar arg)) t))
      (cond
       ((|member| (upcase fn) '(y n ye o of))
        (|sayKeyedMsg| 's2iv0002 '(fortran |fortran|)))
       ((|member| (upcase fn) '(no off)) (setq |$fortranFormat| nil))
       ((|member| (upcase fn) '(yes on)) (setq |$fortranFormat| t))
       ((eq (upcase fn) 'console)
```

```
    (shut |$fortranOutputStream|)
    (setq |$fortranOutputStream|
     (defiostream '((mode . output) (device . console)) 255 0))
    (setq |$fortranOutputFile| "CONSOLE"))))
((or
  (and (pairp arg)
       (progn
        (setq fn (qcar arg))
        (setq tmp1 (qcdr arg))
        (and (pairp tmp1)
             (eq (qcdr tmp1) nil)
             (progn (setq ft (qcar tmp1)) t))))
  (and (pairp arg)
       (progn
        (setq fn (qcar arg))
        (setq tmp1 (qcdr arg))
        (and (pairp tmp1)
             (progn
              (setq ft (qcar tmp1))
              (setq tmp2 (qcdr tmp1))
              (and (pairp tmp2)
                   (eq (qcdr tmp2) nil)
                   (progn (setq fm (qcar tmp2)) t)))))))
 (when (setq ptype (|pathnameType| fn))
   (setq fn (strconc (|pathnameDirectory| fn) (|pathnameName| fn)))
   (setq ft ptype))
 (unless fm (setq fm 'a))
 (setq filename ($filep fn ft fm))
 (cond
  ((null filename)
   (|sayKeyedMsg| 'S2IV0003 (list fn ft fm)))
  ((setq teststream (|makeStream| append filename 255 0))
   (SHUT |$fortranOutputStream|)
   (setq |$fortranOutputStream| teststream)
   (setq |$fortranOutputFile| (|object2String| filename))
   (unless quiet
     (|sayKeyedMsg| 'S2IV0004 (list 'fortran |$fortranOutputFile|))))
  ((null quiet)
   (|sayKeyedMsg| 'S2IV0003 (list fn ft fm)))
  (t nil)))
(t
 (unless quiet (|sayKeyedMsg| 'S2IV0005 nil))
 (|describeSetOutputFortran|)))))))
```

32.35.14 defun describeSetOutputFortran

⟨*defun describeSetOutputFortran*⟩≡

```
(defun |describeSetOutputFortran| ()
 (|sayBrightly| (list
  '|%b| ")set output fortran"
  '|%d| "is used to tell AXIOM to turn FORTRAN-style output"
  '|%l| "printing on and off, and where to place the output.  By default, the"
  '|%l| "destination for the output is the screen but printing is turned off."
  '|%l|
  '|%l| "Also See: )set fortran"
  '|%l|
  '|%l| "Syntax:   )set output fortran <arg>"
  '|%l| "    where arg can be one of"
  '|%l| "  on          turn FORTRAN printing on"
  '|%l| "  off         turn FORTRAN printing off (default state)"
  '|%l| "  console     send FORTRAN output to screen (default state)"
  '|%l|
  "  fp<.fe>     send FORTRAN output to file with file prefix fp and file"
  '|%l| "              extension .fe. If not given, .fe defaults to .sfort."
  '|%l|
  '|%l| "If you wish to send the output to a file, you must issue this command"
  '|%l| "twice: once with"
  '|%b| "on"
  '|%d| "and once with the file name. For example, to send"
  '|%l| "FORTRAN output to the file"
  '|%b| "polymer.sfort,"
  '|%d| "issue the two commands"
  '|%l|
  '|%l| "  )set output fortran on"
  '|%l| "  )set output fortran polymer"
  '|%l|
  '|%l| "The output is placed in the directory from which you invoked AXIOM or"
  '|%l| "the one you set with the )cd system command."
  '|%l| "The current setting is: "
  '|%b| (|setOutputFortran| '|%display%|)
  '|%d|)))
```

32.35.15 fraction

```
--------------------- The fraction Option ---------------------

 Description: how fractions are formatted

 The fraction option may be followed by any one of the following:

 -> vertical
    horizontal

 The current setting is indicated.
```

32.35.16 defvar $HiFiAccess

⟨*initvars*⟩+≡

```
  (defvar |$fractionDisplayType| '|vertical| "how fractions are formatted")
```

⟨*outputfraction*⟩≡

```
        (|fraction|
         "how fractions are formatted"
         |interpreter|
         LITERALS
         |$fractionDisplayType|
         (|vertical| |horizontal|)
         |vertical|)
```

32.35.17 length

```
---------------------- The length Option ----------------------

 Description: line length of output displays

 The length option may be followed by an integer in the range
 10 to 245 inclusive. The current setting is 77
```

32.35.18 defvar $linelength

⟨*initvars*⟩+≡

```
  (defvar $linelength 77 "line length of output displays")
```

⟨*outputlength*⟩≡
```
        (|length|
         "line length of output displays"
         |interpreter|
         INTEGER
         $LINELENGTH
         (10 245)
         77)
```

32.35.19 mathml

```
---------------------- The mathml Option -----------------------

 Description: create output in MathML style

 )set output mathml is used to tell AXIOM to turn MathML-style output
printing on and off, and where to place the output.  By default,
the destination for the output is the screen but printing is
turned off.

Syntax:   )set output mathml <arg>
    where arg can be one of
  on          turn MathML printing on
  off         turn MathML printing off (default state)
  console     send MathML output to screen (default state)
  fp<.fe>     send MathML output to file with file prefix fp
              and file extension .fe. If not given,
              .fe defaults to .smml.

If you wish to send the output to a file, you must issue
this command twice: once with on and once with the file name.
For example, to send MathML output to the file polymer.smml,
issue the two commands

  )set output mathml on
  )set output mathml polymer

The output is placed in the directory from which you invoked
AXIOM or the one you set with the )cd system command.
The current setting is:  Off:CONSOLE
```

32.35.20 defvar $mathmlFormat

⟨*initvars*⟩+≡
```
  (defvar |$mathmlFormat| nil "create output in MathML format")
```

32.35.21 defvar $mathmlOutputFile

⟨*initvars*⟩+≡

```
(defvar |$mathmlOutputFile| "CONSOLE"
   "where MathML output goes (enter {\em console} or a pathname)")
```

⟨*outputmathml*⟩≡

```
       (|mathml|
        "create output in MathML style"
        |interpreter|
        FUNCTION
        |setOutputMathml|
        (("create output in MathML format"
          LITERALS
          |$mathmlFormat|
          (|off| |on|)
          |off|)
         (|break| |$mathmlFormat|)
         ("where MathML output goes (enter {\em console} or a pathname)"
          FILENAME
          |$mathmlOutputFile|
          |chkOutputFileName|
          "console"))
        NIL)
```

32.35.22 defun setOutputMathml

⟨*defun setOutputMathml*⟩≡

```
(defun |setOutputMathml| (arg)
 (let (label tmp1 tmp2 ptype fn ft fm filename teststream)
 (declare (special |$mathmlOutputStream| |$mathmlOutputFile| |$mathmlFormat|))
   (cond
    ((eq arg '|%initialize%|)
     (setq |$mathmlOutputStream|
      (defiostream '((mode . output) (device . console)) 255 0))
     (setq |$mathmlOutputFile| "CONSOLE")
     (setq |$mathmlFormat| nil))
    ((eq arg '|%display%|)
     (if |$mathmlFormat|
      (setq label "On:")
      (setq label "Off:"))
     (strconc label |$mathmlOutputFile|))
    ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
     (|describeSetOutputMathml|))
    (t
     (cond
      ((and (pairp arg)
            (eq (qcdr arg) nil)
            (progn (setq fn (qcar arg)) t)
            (|member| fn '(y n ye yes no o on of off console
                   |y| |n| |ye| |yes| |no| |o| |on| |of| |off| |console|)))
       '|ok|)
      (t (setq arg (list fn '|smml|))))
     (cond
      ((and (pairp arg)
            (eq (qcdr arg) nil)
            (progn (setq fn (qcar arg)) t))
       (cond
        ((|member| (upcase fn) '(y n ye o of))
         (|sayKeyedMsg| 's2iv0002 '(|MathML| |mathml|)))
        ((|member| (upcase fn) '(no off)) (setq |$mathmlFormat| nil))
        ((|member| (upcase fn) '(yes on)) (setq |$mathmlFormat| t))
        ((eq (upcase fn) 'console)
         (shut |$mathmlOutputStream|)
         (setq |$mathmlOutputStream|
          (defiostream '((mode . output) (device . console)) 255 0))
         (setq |$mathmlOutputFile| "CONSOLE"))))
       ((or
         (and (pairp arg)
              (progn
               (setq fn (qcar arg))
```

```
          (setq tmp1 (qcdr arg))
          (and (pairp tmp1)
               (eq (qcdr tmp1) nil)
               (progn (setq ft (qcar tmp1)) t))))
   (and (pairp arg)
        (progn (setq fn (qcar arg))
               (setq tmp1 (qcdr arg))
               (and (pairp tmp1)
                    (progn
                     (setq ft (qcar tmp1))
                     (setq tmp2 (qcdr tmp1))
                     (and (pairp tmp2)
                          (eq (qcdr tmp2) nil)
                          (progn
                           (setq fm (qcar tmp2))
                           t)))))))
     (when (setq ptype (|pathnameType| fn))
      (setq fn
       (strconc (|pathnameDirectory| fn) (|pathnameName| fn)))
      (setq ft ptype))
     (unless fm (setq fm 'a))
     (setq filename ($filep fn ft fm))
     (cond
      ((null filename) (|sayKeyedMsg| 's2iv0003 (list fn ft fm)))
      ((setq teststream (make-outstream filename 255 0))
       (shut |$mathmlOutputStream|)
       (setq |$mathmlOutputStream| teststream)
       (setq |$mathmlOutputFile| (|object2String| filename))
       (|sayKeyedMsg| 's2iv0004 (list "MathML" |$mathmlOutputFile|)))
      (t (|sayKeyedMsg| 's2iv0003 (list fn ft fm)))))
  (t
   (|sayKeyedMsg| 's2iv0005 nil)
   (|describeSetOutputMathml|)))))))
```

32.35.23 defun describeSetOutputMathml

⟨*defun describeSetOutputMathml*⟩≡

```
(defun |describeSetOutputMathml| ()
 (|sayBrightly| (LIST
 '|%b| ")set output mathml"
 '|%d| "is used to tell AXIOM to turn MathML-style output"
 '|%l| "printing on and off, and where to place the output.  By default, the"
 '|%l| "destination for the output is the screen but printing is turned off."
 '|%l|
 '|%l| "Syntax:   )set output mathml <arg>"
 '|%l| "    where arg can be one of"
 '|%l| "  on          turn MathML printing on"
 '|%l| "  off         turn MathML printing off (default state)"
 '|%l| "  console     send MathML output to screen (default state)"
 '|%l| "  fp<.fe>     send MathML output to file with file prefix fp and file"
 '|%l| "              extension .fe. If not given, .fe defaults to .stex."
 '|%l|
 '|%l| "If you wish to send the output to a file, you must issue this command"
 '|%l| "twice: once with"
 '|%b| "on"
 '|%d| "and once with the file name. For example, to send"
 '|%l| "MathML output to the file"
 '|%b| "polymer.smml,"
 '|%d| "issue the two commands"
 '|%l|
 '|%l| "  )set output mathml on"
 '|%l| "  )set output mathml polymer"
 '|%l|
 '|%l| "The output is placed in the directory from which you invoked AXIOM or"
 '|%l| "the one you set with the )cd system command."
 '|%l| "The current setting is: "
 '|%b| (|setOutputMathml| '|%display%|)
 '|%d|)))
```

32.35.24 openmath

```
---------------------- The openmath Option -----------------------

 Description: create output in OpenMath style

 )set output tex is used to tell AXIOM to turn OpenMath output
printing on and off, and where to place the output.  By default,
the destination for the output is the screen but printing is
turned off.

Syntax:   )set output tex <arg>
    where arg can be one of
  on          turn OpenMath printing on
  off         turn OpenMath printing off (default state)
  console     send OpenMath output to screen (default state)
  fp<.fe>     send OpenMath output to file with file prefix fp
              and file extension .fe. If not given,
              .fe defaults to .sopen.

If you wish to send the output to a file, you must issue
this command twice: once with on and once with the file name.
For example, to send OpenMath output to the file polymer.sopen,
issue the two commands

  )set output openmath on
  )set output openmath polymer

The output is placed in the directory from which you invoked
AXIOM or the one you set with the )cd system command.
The current setting is:  Off:CONSOLE
```

32.35.25 defvar $openMathFormat

⟨*initvars*⟩+≡

```
(defvar |$openMathFormat| nil "create output in OpenMath format")
```

32.35.26 defvar $openMathOutputFile

⟨*initvars*⟩+≡

```
(defvar |$openMathOutputFile| "CONSOLE"
  "where TeX output goes (enter {\em console} or a pathname)")
```

⟨*outputopenmath*⟩≡

```
(|openmath|
 "create output in OpenMath style"
 |interpreter|
 FUNCTION
 |setOutputOpenMath|
 (("create output in OpenMath format"
   LITERALS
   |$openMathFormat|
   (|off| |on|)
   |off|)
  (|break| |$openMathFormat|)
  ("where TeX output goes (enter {\em console} or a pathname)"
   FILENAME
   |$openMathOutputFile|
   |chkOutputFileName|
   "console"))
 NIL)
```

32.35.27 defun setOutputOpenMath

⟨*defun setOutputOpenMath*⟩≡

```
(defun |setOutputOpenMath| (arg)
 (let (label tmp1 tmp2 ptype fn ft fm filename teststream)
 (declare (special |$openMathOutputStream| |$openMathFormat|
     |$openMathOutputFile|))
   (cond
    ((eq arg '|%initialize%|)
     (setq |$openMathOutputStream|
      (defiostream  '((mode . output) (device . console)) 255 0))
     (setq |$openMathOutputFile| "CONSOLE")
     (setq |$openMathFormat| NIL))
    ((eq arg '|%display%|)
     (if |$openMathFormat|
      (setq label "On:")
      (setq label "Off:"))
     (strconc label |$openMathOutputFile|))
    ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
     (|describeSetOutputOpenMath|))
    (t
     (cond
      ((and (pairp arg)
            (eq (qcdr arg) nil)
            (progn (setq fn (qcar arg)) t)
            (|member| fn '(y n ye yes no o on of off console
                   |y| |n| |ye| |yes| |no| |o| |on| |of| |off| |console|)))
        '|ok|)
      (t (setq arg (list fn '|som|))))
     (cond
      ((and (pairp arg)
            (eq (qcdr arg) nil)
            (progn (setq fn (qcar arg)) t))
       (cond
        ((|member| (upcase fn) '(y n ye o of))
         (|sayKeyedMsg| 's2iv0002 '(|OpenMath| |openmath|)))
        ((|member| (upcase fn) '(no off)) (setq |$openMathFormat| nil))
        ((|member| (upcase fn) '(yes on)) (setq |$openMathFormat| t))
        ((eq (upcase fn) 'console)
         (shut |$openMathOutputStream|)
         (setq |$openMathOutputStream|
          (defiostream '((mode . output) (device . console)) 255 0))
         (setq |$openMathOutputFile| "CONSOLE"))))
      ((or
        (and (pairp arg)
             (progn (setq fn (qcar arg))
```

```
                (setq tmp1 (qcdr arg))
                (and (pairp tmp1)
                     (eq (qcdr tmp1) nil)
                     (progn (setq ft (qcar tmp1)) t))))
  (and (pairp arg)
       (progn
        (setq fn (qcar arg))
        (setq tmp1 (qcdr arg))
        (and (pairp tmp1)
             (progn (setq ft (qcar tmp1))
                    (setq tmp2 (qcdr tmp1))
                    (and (pairp tmp2)
                         (eq (qcdr tmp2) nil)
                         (progn (setq fm (qcar tmp2)) t)))))))
  (when (setq ptype (|pathnameType| fn))
   (setq fn (strconc (|pathnameDirectory| fn) (|pathnameName| fn)))
   (setq ft ptype))
  (unless fm (setq fm 'a))
  (setq filename ($filep fn ft fm))
  (cond
   ((null filename)
    (|sayKeyedMsg| 's2iv0003 (list fn ft fm)))
   ((setq teststream (make-outstream filename 255 0))
    (shut |$openMathOutputStream|)
    (setq |$openMathOutputStream| teststream)
    (setq |$openMathOutputFile| (|object2String| filename))
    (|sayKeyedMsg| 's2iv0004 (list "OpenMath" |$openMathOutputFile|)))
   (t
    (|sayKeyedMsg| 's2iv0003 (list fn ft fm)))))
(t
 (|sayKeyedMsg| 's2iv0005 nil)
 (|describeSetOutputOpenMath|)))))))
```

32.35.28 defun describeSetOutputOpenMath

⟨*defun describeSetOutputOpenMath*⟩≡

```
(defun |describeSetOutputOpenMath| ()
 (|sayBrightly| (list
 '|%b| ")set output openmath"
 '|%d| "is used to tell AXIOM to turn OpenMath output"
 '|%l| "printing on and off, and where to place the output.  By default, the"
 '|%l| "destination for the output is the screen but printing is turned off."
 '|%l|
 '|%l| "Syntax:   )set output openmath <arg>"
 '|%l| "    where arg can be one of"
 '|%l| "  on          turn OpenMath printing on"
 '|%l| "  off         turn OpenMath printing off (default state)"
 '|%l| "  console     send OpenMath output to screen (default state)"
 '|%l|
 "  fp<.fe>     send OpenMath output to file with file prefix fp and file"
 '|%l| "              extension .fe. If not given, .fe defaults to .som."
 '|%l|
 '|%l| "If you wish to send the output to a file, you must issue this command"
 '|%l| "twice: once with"
 '|%b| "on"
 '|%d| "and once with the file name. For example, to send"
 '|%l| "OpenMath output to the file"
 '|%b| "polymer.som,"
 '|%d| "issue the two commands"
 '|%l|
 '|%l| "  )set output openmath on"
 '|%l| "  )set output openmath polymer"
 '|%l|
 '|%l| "The output is placed in the directory from which you invoked AXIOM or"
 '|%l| "the one you set with the )cd system command."
 '|%l| "The current setting is: "
 '|%b| (|setOutputOpenMath| '|%display%|)
 '|%d|)))
```

32.35.29 script

```
---------------------- The script Option ----------------------

 Description: display output in SCRIPT formula format

 )set output script is used to tell AXIOM to turn IBM Script
 formula-style output printing on and off, and where to place
 the output.  By default, the destination for the output is the
 screen but printing is turned off.

Syntax:   )set output script <arg>
    where arg can be one of
  on      turn IBM Script formula printing on
  off     turn IBM Script formula printing off
          (default state)
  console send IBM Script formula output to screen
          (default state)
  fp<.fe> send IBM Script formula output to file with file
          prefix fp and file extension .fe. If not given,
          .fe defaults to .sform.

If you wish to send the output to a file, you must issue
this command twice: once with on and once with the file
name. For example, to send IBM Script formula output to
the file polymer.sform, issue the two commands

  )set output script on
  )set output script polymer

The output is placed in the directory from which you
invoked AXIOM or the one you set with the )cd system command.
The current setting is:  Off:CONSOLE
```

32.35.30 defvar $formulaFormat

⟨*initvars*⟩+≡

```
(defvar |$formulaFormat| nil "display output in SCRIPT format")
```

32.35.31 defvar $formulaOutputFile

⟨*initvars*⟩+≡

```
(defvar |$formulaOutputFile| "CONSOLE"
    "where script output goes (enter {\em console} or a a pathname)")
```

⟨*outputscript*⟩≡

```
(|script|
 "display output in SCRIPT formula format"
 |interpreter|
 FUNCTION
 |setOutputFormula|
 (("display output in SCRIPT format"
   LITERALS
   |$formulaFormat|
   (|off| |on|)
   |off|)
  (|break| |$formulaFormat|)
  ("where script output goes (enter {\em console} or a a pathname)"
   FILENAME
   |$formulaOutputFile|
   |chkOutputFileName|
   "console"))
 NIL)
```

32.35.32 defun setOutputFormula

⟨*defun setOutputFormula*⟩≡

```
(defun |setOutputFormula| (arg)
 (let (label tmp1 tmp2 ptype fn ft fm filename teststream)
 (declare (special |$formulaOutputStream| |$formulaOutputFile|
    |$formulaFormat|))
   (cond
    ((eq arg '|%initialize%|)
     (setq |$formulaOutputStream|
      (defiostream '((mode . output) (device . console)) 255 0))
     (setq |$formulaOutputFile| "CONSOLE")
     (setq |$formulaFormat| nil))
    ((eq arg '|%display%|)
     (if |$formulaFormat|
      (setq label "On:")
      (setq label "Off:"))
     (strconc label |$formulaOutputFile|))
    ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
     (|describeSetOutputFormula|))
    (t
     (cond
      ((and (pairp arg)
            (eq (qcdr arg) nil)
            (progn (setq fn (qcar arg)) t)
            (|member| fn '(y n ye yes no o on of off console
                      |y| |n| |ye| |yes| |no| |o| |on| |of| |off| |console|)))
        '|ok|)
      (t (setq arg (list fn '|sform|))))
     (cond
      ((and (pairp arg)
            (eq (qcdr arg) nil)
            (progn (setq fn (qcar arg)) t))
       (cond
        ((|member| (upcase fn) '(y n ye o of))
         (|sayKeyedMsg| 's2iv0002 '(|script| |script|)))
        ((|member| (upcase fn) '(no off)) (setq |$formulaFormat| nil))
        ((|member| (upcase fn) '(yes on)) (setq |$formulaFormat| t))
        ((eq (upcase fn) 'console)
         (SHUT |$formulaOutputStream|)
         (setq |$formulaOutputStream|
          (defiostream '((mode . output) (device . console)) 255 0))
         (setq |$formulaOutputFile| "CONSOLE"))))
       ((or
         (and (pairp arg)
              (progn (setq fn (qcar arg))
```

```
                (setq tmp1 (qcdr arg))
                (and (pairp tmp1)
                     (eq (qcdr tmp1) nil)
                     (progn (setq ft (qcar tmp1)) t))))
  (and (pairp arg)
       (progn (setq fn (qcar arg))
              (setq tmp1 (qcdr arg))
              (and (pairp tmp1)
                   (progn (setq ft (qcar tmp1))
                          (setq tmp2 (qcdr tmp1))
                          (and (pairp tmp2)
                               (eq (qcdr tmp2) nil)
                               (progn
                                (setq fm (qcar tmp2)) t)))))))
  (if (setq ptype (|pathnameType| fn))
    (setq fn (strconc (|pathnameDirectory| fn) (|pathnameName| fn)))
    (setq ft ptype))
  (unless fm (setq fm 'a))
  (setq filename ($filep fn ft fm))
  (cond
   ((null filename) (|sayKeyedMsg| 's2iv0003 (list fn ft fm)))
   ((setq teststream (make-outstream filename 255 0))
    (shut |$formulaOutputStream|)
    (setq |$formulaOutputStream| teststream)
    (setq |$formulaOutputFile| (|object2String| filename))
    (|sayKeyedMsg| 's2iv0004
     (list "IBM Script formula" |$formulaOutputFile| )))
   (t
    (|sayKeyedMsg| 's2iv0003 (list fn ft fm)))))
(t
 (|sayKeyedMsg| 's2iv0005 nil)
 (|describeSetOutputFormula|)))))))
```

32.35.33 defun describeSetOutputFormula

⟨*defun describeSetOutputFormula*⟩≡

```
(defun |describeSetOutputFormula| ()
 (|sayBrightly| (list
 '|%b| ")set output script"
 '|%d| "is used to tell AXIOM to turn IBM Script formula-style"
 '|%l|
 "output printing on and off, and where to place the output.  By default, the"
 '|%l| "destination for the output is the screen but printing is turned off."
 '|%l|
 '|%l| "Syntax:   )set output script <arg>"
 '|%l| "    where arg can be one of"
 '|%l| "  on          turn IBM Script formula printing on"
 '|%l| "  off         turn IBM Script formula printing off (default state)"
 '|%l| "  console     send IBM Script formula output to screen (default state)"
 '|%l|
 "  fp<.fe>     send IBM Script formula output to file with file prefix fp"
 '|%l|
 "              and file extension .fe. If not given, .fe defaults to .sform."
 '|%l|
 '|%l| "If you wish to send the output to a file, you must issue this command"
 '|%l| "twice: once with"
 '|%b| "on"
 '|%d| "and once with the file name. For example, to send"
 '|%l| "IBM Script formula output to the file"
 '|%b| "polymer.sform,"
 '|%d| "issue the two commands"
 '|%l|
 '|%l| "  )set output script on"
 '|%l| "  )set output script polymer"
 '|%l|
 '|%l| "The output is placed in the directory from which you invoked AXIOM or"
 '|%l| "the one you set with the )cd system command."
 '|%l| "The current setting is: "
 '|%b| (|setOutputFormula| '|%display%|)
 '|%d|)))
```

32.35.34 scripts

```
--------------------- The scripts Option ----------------------

 Description: show subscripts,... linearly

 The scripts option may be followed by any one of the following:

    yes
    no

 The current setting is indicated.
```

32.35.35 defvar $linearFormatScripts

⟨*initvars*⟩+≡
```
  (defvar |$linearFormatScripts| nil "show subscripts,... linearly")
```

⟨*outputscripts*⟩≡
```
        (|scripts|
         "show subscripts,... linearly"
         |interpreter|
         LITERALS
         |$linearFormatScripts|
         (|on| |off|)
         |off|)
```

32.35.36 showeditor

```
-------------------- The showeditor Option --------------------

 Description: view output of )show in editor

 The showeditor option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.35.37 defvar $useEditorForShowOutput

⟨*initvars*⟩+≡

```
(defvar |$useEditorForShowOutput| nil "view output of )show in editor")
```

⟨*outputshoweditor*⟩≡

```
(|showeditor|
 "view output of )show in editor"
 |interpreter|
 LITERALS
 |$useEditorForShowOutput|
 (|on| |off|)
 |off|)
```

32.35.38 tex

```
---------------------- The tex Option -----------------------

 Description: create output in TeX style

 )set output tex is used to tell AXIOM to turn TeX-style output
printing on and off, and where to place the output.  By default,
the destination for the output is the screen but printing is
turned off.

Syntax:   )set output tex <arg>
    where arg can be one of
  on          turn TeX printing on
  off         turn TeX printing off (default state)
  console     send TeX output to screen (default state)
  fp<.fe>     send TeX output to file with file prefix fp
              and file extension .fe. If not given,
              .fe defaults to .stex.

If you wish to send the output to a file, you must issue
this command twice: once with on and once with the file name.
For example, to send TeX output to the file polymer.stex,
issue the two commands

  )set output tex on
  )set output tex polymer

The output is placed in the directory from which you invoked
AXIOM or the one you set with the )cd system command.
The current setting is:  Off:CONSOLE
```

32.35.39 defvar $texFormat

⟨*initvars*⟩+≡

```
(defvar |$texFormat| nil "create output in TeX format")
```

32.35.40 defvar $texOutputFile

⟨*initvars*⟩+≡

```
(defvar |$texOutputFile| "CONSOLE"
    "where TeX output goes (enter {\em console} or a pathname)")
```

⟨*outputtex*⟩≡

```
(|tex|
 "create output in TeX style"
 |interpreter|
 FUNCTION
 |setOutputTex|
 (("create output in TeX format"
   LITERALS
   |$texFormat|
   (|off| |on|)
   |off|)
  (|break| |$texFormat|)
  ("where TeX output goes (enter {\em console} or a pathname)"
   FILENAME
   |$texOutputFile|
   |chkOutputFileName|
   "console"))
 NIL)
```

32.35.41 defun setOutputTex

⟨*defun setOutputTex*⟩≡

```
(defun |setOutputTex| (arg)
 (let (label tmp1 tmp2 ptype fn ft fm filename teststream)
  (declare (special |$texOutputStream| |$texOutputFile| |$texFormat|))
  (cond
   ((eq arg '|%initialize%|)
    (setq |$texOutputStream|
     (defiostream '((mode . output) (device . console)) 255 0))
    (setq |$texOutputFile| "CONSOLE")
    (setq |$texFormat| nil))
   ((eq arg '|%display%|)
    (if |$texFormat|
     (setq label "On:")
     (setq label "Off:"))
    (strconc label |$texOutputFile|))
   ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
    (|describeSetOutputTex|))
   (t
    (cond
     ((and (pairp arg)
           (eq (qcdr arg) nil)
           (progn (setq fn (qcar arg)) t)
           (|member| fn '(y n ye yes no o on of off console
                 |y| |n| |ye| |yes| |no| |o| |on| |of| |off| |console|)))
      '|ok|)
     (t (setq arg (list fn '|stex| nil))))
    (cond
     ((and (pairp arg)
           (eq (qcdr arg) nil)
           (progn (setq fn (qcar arg)) t))
      (cond
       ((|member| (upcase fn) '(y n ye o of))
        (|sayKeyedMsg| 's2iv0002 '(|TeX| |tex|)))
       ((|member| (upcase fn) '(no off)) (setq |$texFormat| nil))
       ((|member| (upcase fn) '(yes on)) (setq |$texFormat| t))
       ((eq (upcase fn) 'console)
        (shut |$texOutputStream|)
        (setq |$texOutputStream|
          (defiostream '((mode . output) (device . console)) 255 0))
        (setq |$texOutputFile| "CONSOLE"))))
     ((or
       (and (pairp arg)
            (progn (setq fn (qcar arg))
                   (setq tmp1 (qcdr arg))
```

```
              (and (pairp tmp1)
                   (eq (qcdr tmp1) nil)
                   (progn (setq ft (qcar tmp1)) t))))
 (and (pairp arg)
      (progn (setq fn (qcar arg))
             (setq tmp1 (qcdr arg))
             (and (pairp tmp1)
             (progn (setq ft (qcar tmp1))
                    (setq tmp2 (qcdr tmp1))
                    (and (pairp tmp2)
                         (eq (qcdr tmp2) nil)
                         (progn (setq fm (qcar tmp2)) t)))))))
 (when (setq ptype (|pathnameType| fn))
  (setq fn (strconc (|pathnameDirectory| fn) (|pathnameName| fn)))
  (setq ft ptype))
 (unless fm (setq fm 'A))
 (setq filename ($filep fn ft fm))
 (cond
  ((null filename) (|sayKeyedMsg| 's2iv0003 (list fn ft fm )))
  ((setq teststream (make-outstream filename 255 0))
   (shut |$texOutputStream|)
   (setq |$texOutputStream| teststream)
   (setq |$texOutputFile| (|object2String| filename))
   (|sayKeyedMsg| 's2iv0004 (list "TeX" |$texOutputFile|)))
  (t (|sayKeyedMsg| 'S2IV0003 (list fn ft fm ))))
(t
 (|sayKeyedMsg| 's2iv0005 nil)
 (|describeSetOutputTex|)))))))
```

32.35.42 defun describeSetOutputTex

⟨*defun describeSetOutputTex*⟩≡

```
(defun |describeSetOutputTex| ()
 (|sayBrightly| (list
 '|%b| ")set output tex"
 '|%d| "is used to tell AXIOM to turn TeX-style output"
 '|%l| "printing on and off, and where to place the output.  By default, the"
 '|%l| "destination for the output is the screen but printing is turned off."
 '|%l|
 '|%l| "Syntax:   )set output tex <arg>"
 '|%l| "    where arg can be one of"
 '|%l| "  on          turn TeX printing on"
 '|%l| "  off         turn TeX printing off (default state)"
 '|%l| "  console     send TeX output to screen (default state)"
 '|%l| "  fp<.fe>     send TeX output to file with file prefix fp and file"
 '|%l| "              extension .fe. If not given, .fe defaults to .stex."
 '|%l|
 '|%l| "If you wish to send the output to a file, you must issue this command"
 '|%l| "twice: once with"
 '|%b| "on"
 '|%d| "and once with the file name. For example, to send"
 '|%l| "TeX output to the file"
 '|%b| "polymer.stex,"
 '|%d| "issue the two commands"
 '|%l|
 '|%l| "  )set output tex on"
 '|%l| "  )set output tex polymer"
 '|%l|
 '|%l| "The output is placed in the directory from which you invoked AXIOM or"
 '|%l| "the one you set with the )cd system command."
 '|%l| "The current setting is: "
 '|%b| (|setOutputTex| '|%display%|)
 '|%d|)))
```

32.36 quit

```
---------------------- The quit Option -----------------------

 Description: protected or unprotected quit

 The quit option may be followed by any one of the following:

    protected
 -> unprotected

 The current setting is indicated.
```

32.36.1 defvar $quitCommandType

⟨*initvars*⟩+≡
```
  (defvar |$quitCommandType| '|protected| "protected or unprotected quit")
```

⟨*quit*⟩≡
```
    (|quit|
     "protected or unprotected quit"
     |interpreter|
     LITERALS
     |$quitCommandType|
     (|protected| |unprotected|)
     |protected|)
```

32.37 streams

```
             Current Values of  streams  Variables

Variable     Description                              Current Value
------------------------------------------------------------------
calculate    specify number of elements to calculate    10
showall      display all stream elements computed       off
```

⟨*streams*⟩≡
```
    (|streams|
     "set some options for working with streams"
     |interpreter|
     TREE
     |novar|
     (
  ⟨streamscalculate⟩
  ⟨streamsshowall⟩
    ))
```

32.37.1 calculate

```
-------------------- The calculate Option ---------------------

 Description: specify number of elements to calculate

   )set streams calculate is used to tell AXIOM how many elements
    of a stream to calculate when a computation uses the stream.
    The value given after calculate must either be the word all
    or a positive integer.

      The current setting is 10 .
```

32.37.2 defvar $streamCount

⟨*initvars*⟩+≡
```
  (defvar |$streamCount| 10
      "number of initial stream elements you want calculated")
```

⟨*streamscalculate*⟩≡
```
       (|calculate|
        "specify number of elements to calculate"
        |interpreter|
        FUNCTION
        |setStreamsCalculate|
        (("number of initial stream elements you want calculated"
         INTEGER
         |$streamCount|
         (0 NIL)
         10))
        NIL)
```

32.37.3 defun setStreamsCalculate

⟨*defun setStreamsCalculate*⟩≡
```
  (defun |setStreamsCalculate| (arg)
   (let (n)
    (declare (special |$streamCount|))
    (cond
     ((eq arg '|%initialize%|) (setq |$streamCount| 10))
     ((eq arg '|%display%|) (|object2String| |$streamCount|))
     ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
       (|describeSetStreamsCalculate|))
     (t
       (setq n (car arg))
       (cond
        ((and (nequal n '|all|) (or (null (fixp n)) (minusp n)))
          (|sayMessage|
           `("Your value of" ,@(|bright| n) "is invalid because ..."))
          (|describeSetStreamsCalculate|)
          (|terminateSystemCommand|))
        (t (setq |$streamCount| n)))))))
```

32.37.4 defun describeSetStreamsCalculate

⟨*defun describeSetStreamsCalculate*⟩≡
```
  (defun |describeSetStreamsCalculate| ()
   (declare (special |$streamCount|))
   (|sayKeyedMsg| 's2iv0001 (list |$streamCount|)))
```

32.37.5 showall

```
--------------------- The showall Option ----------------------

 Description: display all stream elements computed

 The showall option may be followed by any one of the following:

    on
 -> off

 The current setting is indicated.
```

32.37.6 defvar $streamsShowAll

⟨*initvars*⟩+≡
```
  (defvar |$streamsShowAll| nil "display all stream elements computed")
```

⟨*streamsshowall*⟩≡
```
       (|showall|
        "display all stream elements computed"
        |interpreter|
        LITERALS
        |$streamsShowAll|
        (|on| |off|)
        |off|)
```

32.38 system

```
              Current Values of  system  Variables

Variable     Description                              Current Value
-------------------------------------------------------------------
functioncode show gen. LISP for functions when compiled off
optimization show optimized LISP code                 off
prettyprint  prettyprint BOOT func's as they compile  off
```

⟨*system*⟩≡
```
    (|system|
     "set some system development variables"
     |development|
     TREE
     |novar|
     (
```
⟨*systemfunctioncode*⟩
⟨*systemoptimization*⟩
⟨*systemprettyprint*⟩
```
     ))
```

32.38.1 functioncode

```
------------------- The functioncode Option --------------------

 Description: show gen. LISP for functions when compiled

 The functioncode option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.38.2 defvar $reportCompilation

⟨*initvars*⟩+≡
```
  (defvar |$reportCompilation| nil "show gen. LISP for functions when compiled")
```

⟨*systemfunctioncode*⟩≡
```
       (|functioncode|
        "show gen. LISP for functions when compiled"
        |development|
        LITERALS
        |$reportCompilation|
        (|on| |off|)
        |off|)
```

32.38.3 optimization

```
------------------- The optimization Option -------------------

 Description: show optimized LISP code

 The optimization option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.38.4 defvar $reportOptimization

⟨*initvars*⟩+≡
```
  (defvar |$reportOptimization| nil "show optimized LISP code")
```

⟨*systemoptimization*⟩≡
```
       (|optimization|
        "show optimized LISP code"
        |development|
        LITERALS
        |$reportOptimization|
        (|on| |off|)
        |off|)
```

32.38.5 prettyprint

```
------------------- The prettyprint Option --------------------

 Description: prettyprint BOOT func's as they compile

 The prettyprint option may be followed by any one of the
 following:

    on
 -> off

 The current setting is indicated.
```

32.38.6 defvar $prettyprint

⟨*initvars*⟩+≡

```
  (defvar $prettyprint t "prettyprint BOOT func's as they compile")
```

⟨*systemprettyprint*⟩≡

```
       (|prettyprint|
        "prettyprint BOOT func's as they compile"
        |development|
        LITERALS
        $prettyprint
        (|on| |off|)
        |on|)
```

32.39 userlevel

```
-------------------- The userlevel Option ---------------------

 Description: operation access level of system user

 The userlevel option may be followed by any one of the
 following:

    interpreter
    compiler
 -> development

 The current setting is indicated.
```

32.39.1 defvar $UserLevel

⟨*initvars*⟩+≡
```
(defvar |$UserLevel| '|development| "operation access level of system user")
```

⟨*userlevel*⟩≡
```
    (|userlevel|
     "operation access level of system user"
     |interpreter|
     LITERALS
     |$UserLevel|
     (|interpreter| |compiler| |development|)
     |development|)
```

⟨*initvars*⟩+≡
```
(defvar |$setOptions| '(
```
⟨*breakmode*⟩
⟨*compiler*⟩
⟨*debug*⟩
⟨*expose*⟩
⟨*functions*⟩
⟨*fortran*⟩
⟨*kernel*⟩
⟨*hyperdoc*⟩
⟨*help*⟩
⟨*history*⟩
⟨*messages*⟩
⟨*naglink*⟩
⟨*output*⟩
⟨*quit*⟩
⟨*streams*⟩
⟨*system*⟩
⟨*userlevel*⟩
```
 ))
```

⟨*initvars*⟩+≡
```
(defvar |$setOptionNames| (mapcar #'car |$setOptions|))
```

⟨*postvars*⟩+≡
```
(eval-when (eval load)
 (|initializeSetVariables| |$setOptions|))
```

32.40 Set code

32.40.1 defun set

⟨*defun set*⟩≡
```
(defun |set| (l)
 (declare (special |$setOptions|))
 (|set1| l |$setOptions|))
```

32.40.2 defun set1

This function will be called with the top level arguments to)set. For instance, given the command

```
)set break break
```

this function gets

```
(set1 (|break| |break|) ....)
```

and given the command

```
)set mes auto off
```

this function gets

```
(set1 (|mes| |auto| |off|) ....)
```

which, because "message" is a TREE, generates the recursive call:

```
(set1 (|auto| |off|) <the message subtree>)
```

The "autoload" subtree is a FUNCTION (printLoadMessages), which gets called with `%describe%`

⟨*defun set1*⟩≡

```
(defun |set1| (l settree)
 (let (|$setOptionNames| arg setdata st setfunarg num upperlimit arg2)
  (declare (special |$setOptionNames| |$UserLevel| |$displaySetValue|))
  (cond
   ((null l) (|displaySetVariableSettings| settree '||))
   (t
    (setq |$setOptionNames|
      (do ((t1 settree (cdr t1)) t0 (x nil))
          ((or (atom t1) (progn (setq x (car t1)) nil)) (nreverse0 t0))
       (seq
        (exit
         (setq t0 (cons (elt x 0) t0))))))
    (setq arg
     (|selectOption| (downcase (car l)) |$setOptionNames| '|optionError|))
    (setq setdata (cons arg (lassoc arg settree)))
    (cond
     ((null (|satisfiesUserLevel| (third setdata)))
      (|sayKeyedMsg| 's2iz0007 (list |$UserLevel| "set option" nil)))
     ((eql 1 (|#| l)) (|displaySetOptionInformation| arg setdata))
     (t
```

```
(setq st (fourth setdata))
(case (fourth setdata)
 (FUNCTION
  (setq setfunarg
   (if (eq (elt l 1) 'default)
    '|%initialize%|
    (kdr l)))
  (if (functionp (fifth setdata))
    (funcall (fifth setdata) setfunarg)
    (|sayMSG| (concatenate 'string "   Function not implemented. "
     (string (fifth setdata)))))
  (when |$displaySetValue|
   (|displaySetOptionInformation| arg setdata))
 NIL)
 (STRING
  (setq arg2 (elt l 1))
  (cond
   ((eq arg2 'default) (set (fifth setdata) (seventh setdata)))
   (arg2 (set (fifth setdata) arg2))
   (t nil))
  (when (or |$displaySetValue| (null arg2))
   (|displaySetOptionInformation| arg setdata))
 NIL)
 (INTEGER
  (setq arg2
   (progn
    (setq num (elt l 1))
    (cond
     ((and (fixp num)
           (>= num (elt (sixth setdata) 0))
           (or (null (setq upperlimit (elt (sixth setdata) 1)))
               (<= num upperlimit)))
      num)
     (t
      (|selectOption|
       (elt l 1)
       (cons '|default| (sixth setdata)) nil)))))
  (cond
   ((eq arg2 'default) (set (fifth setdata) (seventh setdata)))
   (arg2 (set (fifth setdata) arg2))
   (t nil))
  (cond
   ((or |$displaySetValue| (null arg2))
    (|displaySetOptionInformation| arg setdata)))
  (cond
   ((null arg2)
```

```
    (|sayMessage|
     `(" Your value" ,@(|bright| (|object2String| (elt l 1)))
       "is not among the valid choices.")))
   (t nil)))
 (LITERALS
  (cond
   ((setq arg2
     (|selectOption| (elt l 1)
      (cons '|default| (sixth setdata)) nil))
    (cond
     ((eq arg2 'default)
      (set (fifth setdata)
        (|translateYesNo2TrueFalse| (seventh setdata))))
     (t
      (cond ((eq arg2 '|nobreak|) (use-fast-links t)))
      (cond
       ((eq arg2 '|fastlinks|)
        (use-fast-links nil)
        (setq arg2 '|break|)))
      (set (fifth setdata) (|translateYesNo2TrueFalse| arg2))))))
  (when (or |$displaySetValue| (null arg2))
    (|displaySetOptionInformation| arg setdata))
  (cond
   ((null arg2)
    (|sayMessage|
     (cons " Your value"
      (append (|bright| (|object2String| (elt l 1)))
       (cons "is not among the valid choices." nil)))))
   (t nil)))
 (TREE (|set1| (kdr l) (sixth setdata)) nil)
 (t
  (|sayMessage|
   `("Cannot handle set tree node type" ,@(|bright| st) |yet|))
  nil))))))))
```

Chapter 33

)show Command

33.1 show man page

⟨*show.help*⟩≡

```
==========================================================================
A.22.  )show
==========================================================================

User Level Required:  interpreter

Command Syntax:

  - )show nameOrAbbrev
  - )show nameOrAbbrev )operations
  - )show nameOrAbbrev )attributes

Command Description:
This command displays information about AXIOM domain, package and category
constructors. If no options are given, the )operations option is assumed. For
example,

)show POLY
)show POLY )operations
)show Polynomial
)show Polynomial )operations

each display basic information about the Polynomial domain constructor and
then provide a listing of operations. Since Polynomial requires a Ring (for
example, Integer) as argument, the above commands all refer to a unspecified
ring R. In the list of operations, $ means Polynomial(R).
```

```
The basic information displayed includes the signature of the constructor
(the name and arguments), the constructor abbreviation, the exposure status
of the constructor, and the name of the library source file for the
constructor.

If operation information about a specific domain is wanted, the full or
abbreviated domain name may be used. For example,

)show POLY INT
)show POLY INT )operations
)show Polynomial Integer
)show Polynomial Integer )operations

are among the combinations that will display the operations exported by the
domain Polynomial(Integer) (as opposed to the general domain constructor
Polynomial). Attributes may be listed by using the )attributes option.

Also See:
o )display
o )set
o )what
```

1

[1] "display" (17.2.1 p 137) "set" (32.40.1 p 372) "what" (40.1.2 p 467)

Chapter 34

)spool Command

34.1 spool man page

⟨*spool.help*⟩≡

```
==========================================================================
A.23.  )spool
==========================================================================

User Level Required:  interpreter

Command Syntax:

  - )spool [fileName]
  - )spool

Command Description:

This command is used to save (spool) all AXIOM input and output into a file,
called a spool file. You can only have one spool file active at a time. To
start spool, issue this command with a filename. For example,

)spool integrate.out

To stop spooling, issue )spool with no filename.

If the filename is qualified with a directory, then the output will be placed
in that directory. If no directory information is given, the spool file will
be placed in the current directory. The current directory is the directory
from which you started AXIOM or is the directory you specified using the )cd
command.
```

```
Also See:
o )cd
```

1

[1] "cd" (11 p 99)

Chapter 35

)summary Command

35.1 summary man page

⟨*summary.help*⟩≡

```
)credits      : list the people who have contributed to Axiom

)help <command> gives more information
)quit         : exit AXIOM

)abbreviation : query, set and remove abbreviations for constructors
)cd           : set working directory
)clear        : remove declarations, definitions or values
)close        : throw away an interpreter client and workspace
)compile      : invoke constructor compiler
)display      : display Library operations and objects in your workspace
)edit         : edit a file
)frame        : manage interpreter workspaces
)history      : manage aspects of interactive session
)library      : introduce new constructors
)lisp         : evaluate a LISP expression
)read         : execute AXIOM commands from a file
)savesystem   : save LISP image to a file
)set          : view and set system variables
)show         : show constructor information
)spool        : log input and output to a file
)synonym      : define an abbreviation for system commands
)system       : issue shell commands
)trace        : trace execution of functions
)undo         : restore workspace to earlier state
)what         : search for various things by name
```

35.1.1 defun summary

⟨*defun summary*⟩≡

```
(defun |summary| (l)
 (declare (ignore l))
 (obey (strconc "cat " (getenviron "AXIOM") "/doc/spadhelp/summary.help")))
```

Chapter 36

)synonym Command

36.1 synonym man page

⟨*synonym.help*⟩≡

```
  ========================================================================
  A.24.  )synonym
  ========================================================================

  User Level Required:  interpreter

  Command Syntax:

    - )synonym
    - )synonym synonym fullCommand
    - )what synonyms

  Command Description:

  This command is used to create short synonyms for system command expressions.
  For example, the following synonyms might simplify commands you often use.

  )synonym save         history )save
  )synonym restore      history )restore
  )synonym mail         system mail
  )synonym ls           system ls
  )synonym fortran      set output fortran

  Once defined, synonyms can be used in place of the longer command
  expressions. Thus
```

```
)fortran on

is the same as the longer

)set fortran output on

To list all defined synonyms, issue either of

)synonyms
)what synonyms

To list, say, all synonyms that contain the substring ‘‘ap’’, issue

)what synonyms ap

Also See:
o )set
o )what
```

[1]

This command is in the list of `$noParseCommands` 7.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 7.2.1

[1] “set” (32.40.1 p 372) “what” (40.1.2 p 467)

Chapter 37

)system Command

37.1 system man page

⟨*system.help*⟩≡

```
==========================================================================
A.25.  )system
==========================================================================

User Level Required:  interpreter

Command Syntax:

  - )system cmdExpression

Command Description:

This command may be used to issue commands to the operating system while
remaining in AXIOM. The cmdExpression is passed to the operating system for
execution.

To get an operating system shell, issue, for example, )system sh. When you
enter the key combination, Ctrl-D (pressing and holding the Ctrl key and then
pressing the D key) the shell will terminate and you will return to AXIOM. We
do not recommend this way of creating a shell because Lisp may field some
interrupts instead of the shell. If possible, use a shell running in another
window.

If you execute programs that misbehave you may not be able to return to
AXIOM. If this happens, you may have no other choice than to restart AXIOM
and restore the environment via )history )restore, if possible.
```

```
Also See:
o )boot
o )fin
o )lisp
o )pquit
o )quit
```

1

This command is in the list of `$noParseCommands` 7.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 7.2.1

[1] "boot" (9 p 91) "fin" (19 p 149) "lisp" (25 p 223) "pquit" (28.2.1 p 230) "quit" (29.2.1 p 234)

Chapter 38

)trace Command

38.1 trace man page

⟨*trace.help*⟩≡

```
==========================================================================
A.26.  )trace
==========================================================================

User Level Required:  interpreter

Command Syntax:

  - )trace
  - )trace )off

  - )trace function [options]
  - )trace constructor [options]
  - )trace domainOrPackage [options]

where options can be one or more of

  - )after S-expression
  - )before S-expression
  - )break after
  - )break before
  - )cond S-expression
  - )count
  - )count n
  - )depth n
  - )local op1 [... opN]
```

-)nonquietly
-)nt
-)off
-)only listOfDataToDisplay
-)ops
-)ops op1 [... opN]
-)restore
-)stats
-)stats reset
-)timer
-)varbreak
-)varbreak var1 [... varN]
-)vars
-)vars var1 [... varN]
-)within executingFunction

Command Description:

This command is used to trace the execution of functions that make up the AXIOM system, functions defined by users, and functions from the system library. Almost all options are available for each type of function but exceptions will be noted below.

To list all functions, constructors, domains and packages that are traced, simply issue

)trace

To untrace everything that is traced, issue

)trace)off

When a function is traced, the default system action is to display the arguments to the function and the return value when the function is exited. Note that if a function is left via an action such as a THROW, no return value will be displayed. Also, optimization of tail recursion may decrease the number of times a function is actually invoked and so may cause less trace information to be displayed. Other information can be displayed or collected when a function is traced and this is controlled by the various options. Most options will be of interest only to AXIOM system developers. If a domain or package is traced, the default action is to trace all functions exported.

Individual interpreter, lisp or boot functions can be traced by listing their names after)trace. Any options that are present must follow the functions to be traced.

```
)trace f

traces the function f. To untrace f, issue

)trace f )off

Note that if a function name contains a special character, it will be
necessary to escape the character with an underscore

)trace _/D_,1

To trace all domains or packages that are or will be created from a
particular constructor, give the constructor name or abbreviation after
)trace.

)trace MATRIX
)trace List Integer

The first command traces all domains currently instantiated with Matrix. If
additional domains are instantiated with this constructor (for example, if
you have used Matrix(Integer) and Matrix(Float)), they will be automatically
traced. The second command traces List(Integer). It is possible to trace
individual functions in a domain or package. See the )ops option below.

The following are the general options for the )trace command.

  )break after
    causes a Lisp break loop to be entered after exiting the traced function.

  )break before
    causes a Lisp break loop to be entered before entering the traced
    function.

  )break
    is the same as )break before.

  )count
    causes the system to keep a count of the number of times the traced
    function is entered. The total can be displayed with )trace )stats and
    cleared with )trace )stats reset.

  )count n
    causes information about the traced function to be displayed for the
    first n executions. After the nth execution, the function is untraced.
```

```
)depth n
  causes trace information to be shown for only n levels of recursion of
  the traced function. The command

  )trace fib )depth 10

  will cause the display of only 10 levels of trace information for the
  recursive execution of a user function fib.

)math
  causes the function arguments and return value to be displayed in the
  AXIOM monospace two-dimensional math format.

)nonquietly
  causes the display of additional messages when a function is traced.

)nt
  This suppresses all normal trace information. This option is useful if
  the )count or )timer options are used and you are interested in the
  statistics but not the function calling information.

)off
  causes untracing of all or specific functions. Without an argument, all
  functions, constructors, domains and packages are untraced. Otherwise,
  the given functions and other objects are untraced. To immediately
  retrace the untraced functions, issue )trace )restore.

)only listOfDataToDisplay
  causes only specific trace information to be shown. The items are listed
  by using the following abbreviations:

  a        display all arguments
  v        display return value
  1        display first argument
  2        display second argument
  15       display the 15th argument, and so on

)restore
  causes the last untraced functions to be retraced. If additional options
  are present, they are added to those previously in effect.

)stats
  causes the display of statistics collected by the use of the )count and
  )timer options.

)stats reset
```

```
    resets to 0 the statistics collected by the use of the )count and )timer
    options.

  )timer
    causes the system to keep a count of execution times for the traced
    function. The total can be displayed with )trace )stats and cleared with
    )trace )stats reset.

  )varbreak var1 [... varN]
    causes a Lisp break loop to be entered after the assignment to any of the
    listed variables in the traced function.

  )vars
    causes the display of the value of any variable after it is assigned in
    the traced function. Note that library code must have been compiled (see
    description of command )compile ) using the )vartrace option in order to
    support this option.

  )vars var1 [... varN]
    causes the display of the value of any of the specified variables after
    they are assigned in the traced function. Note that library code must
    have been compiled (see description of command )compile ) using the
    )vartrace option in order to support this option.

  )within executingFunction
    causes the display of trace information only if the traced function is
    called when the given executingFunction is running.

The following are the options for tracing constructors, domains and packages.

  )local [op1 [... opN]]
    causes local functions of the constructor to be traced. Note that to
    untrace an individual local function, you must use the fully qualified
    internal name, using the escape character _ before the semicolon.

    )trace FRAC )local
    )trace FRAC_;cancelGcd )off

  )ops op1 [... opN]
    By default, all operations from a domain or package are traced when the
    domain or package is traced. This option allows you to specify that only
    particular operations should be traced. The command

    )trace Integer )ops min max _+ _-

    traces four operations from the domain Integer. Since + and - are special
```

```
      characters, it is necessary to escape them with an underscore.

  Also See:
  o )boot
  o )lisp
  o )ltrace
```

[1]

38.1.1 The trace global variables

This decides when to give trace and untrace messages.

⟨*initvars*⟩+≡
```
  (defvar |$traceNoisely| nil)
```

This reports the traced functions

⟨*initvars*⟩+≡
```
  (defvar |$reportSpadTrace| nil)
```

⟨*initvars*⟩+≡
```
  (defvar |$optionAlist| nil)
```

⟨*initvars*⟩+≡
```
  (defvar |$tracedMapSignatures| nil)
```

⟨*initvars*⟩+≡
```
  (defvar |$traceOptionList|
   '(|after| |before| |break| |cond| |count| |depth| |local| |mathprint|
     |nonquietly| |nt| |of| |only| |ops| |restore| |timer| |varbreak|
     |vars| |within|))
```

38.1.2 defun trace

⟨*defun trace*⟩≡
```
  (defun |trace| (l)
   (|traceSpad2Cmd| l))
```

[1] "boot" (9 p 91) "lisp" (25 p 223) "ltrace" (27 p 227)

38.1.3 defun traceSpad2Cmd

⟨*defun traceSpad2Cmd*⟩≡

```
(defun |traceSpad2Cmd| (l)
 (let (tmp1 l1)
 (declare (special |$mapSubNameAlist|))
  (cond
   ((and (pairp l)
         (eq (qcar l) '|Tuple|)
         (progn
          (setq tmp1 (qcdr l))
          (and (pairp tmp1)
               (eq (qcdr tmp1) nil)
               (progn
                (setq l1 (qcar tmp1))
                t))))
     (setq l l1)))
    (setq |$mapSubNameAlist| (|getMapSubNames| l))
    (|trace1| (|augmentTraceNames| l))
    (|traceReply|)))
```

38.1.4 defun trace1

⟨*defun trace1*⟩≡

```
(defun |trace1| (arg)
 (prog (|$traceNoisely| constructor ops lops temp1 opt a
        oldl newoptions domain tracelist optionlist domainlist
        oplist y varlist argument)
  (declare (special |$traceNoisely| |$options| |$lastUntraced|
     |$optionAlist|))
  (return
   (seq
    (progn
     (setq |$traceNoisely| nil)
     (cond
      ((|hasOption| |$options| '|nonquietly|)
        (setq |$traceNoisely| t)))
     (cond
      ((|hasOption| |$options| '|off|)
       (cond
        ((or (setq ops (|hasOption| |$options| 'ops))
             (setq lops (|hasOption| |$options| '|local|)))
          (cond
           ((null arg) (|throwKeyedMsg| 's2it0019 nil))
           (t
             (setq constructor
              (|unabbrev|
               (cond
                ((atom arg) arg)
                ((null (cdr arg))
                  (cond
                   ((atom (car arg)) (car arg))
                   (t (car (car arg)))))
                (t nil))))
             (cond
              ((null (|isFunctor| constructor))
                (|throwKeyedMsg| 's2it0020 nil))
              (t
                (cond (ops (setq ops (|getTraceOption| ops)) nil))
                (cond
                 (lops
                  (setq lops (cdr (|getTraceOption| lops)))
                  (|untraceDomainLocalOps|))
                 (t nil)))))))
        ((and (qslessp 1 (|#| |$options|))
              (null (|hasOption| |$options| '|nonquietly|)))
          (|throwKeyedMsg| 's2it0021 nil))
```

```
    (t (|untrace| arg))))
  ((|hasOption| |$options| '|stats|)
    (cond
     ((qslessp 1 (|#| |$options|))
      (|throwKeyedMsg| 's2it0001 (cons ")trace ... )stats" nil)))
     (t
       (setq temp1 (car |$options|))
       (setq opt (cdr temp1))
       (cond
        ((null opt)
          (|centerAndHighlight| "Traced function execution times" 78 '-)
          (|ptimers|)
          (say " ")
          (|centerAndHighlight| "Traced function execution counts" 78 '-)
          (|pcounters|))
        (t
          (|selectOptionLC| (car opt) '(|reset|) '|optionError|)
          (|resetSpacers|)
          (|resetTimers|)
          (|resetCounters|)
          (|throwKeyedMsg| 's2it0002 nil))))))
  ((setq a (|hasOption| |$options| '|restore|))
    (unless (setq oldl |$lastUntraced|)
     (setq newoptions (|delete| a |$options|))
     (if (null arg)
      (|trace1| oldl)
      (progn
       (dolist (x arg)
        (if (and (pairp x)
                 (progn
                  (setq domain (qcar x))
                  (setq oplist (qcdr x))
                  t)
                 (vecp domain))
         (|sayKeyedMsg| 's2it0003 (cons (|devaluate| domain) nil))
         (progn
           (setq |$options| (append newoptions (lassoc x |$optionAlist|)))
           (|trace1| (list x)))))))))
  ((null arg) nil)
  ((and (pairp arg) (eq (qcdr arg) nil) (eq (qcar arg) '?)) (|?t|))
  (t
   (setq tracelist
    (or
     (prog (t1)
      (setq t1 nil)
      (return
```

```
    (do ((t2 arg (cdr t2)) (x nil))
        ((or (atom t2)
             (progn (setq x (car t2)) nil))
         (nreverse0 t1))
     (seq
      (exit
       (setq t1 (cons (|transTraceItem| x) t1)))))))
  (return nil)))
(do ((t3 tracelist (cdr t3)) (x nil))
    ((or (atom t3) (progn (setq x (car t3)) nil)) nil)
  (seq
   (exit
    (setq |$optionAlist| (addassoc x |$options| |$optionAlist|)))))
(setq optionlist (|getTraceOptions| |$options|))
(setq argument
 (cond
  ((setq domainlist (lassoc '|of| optionlist))
    (cond
     ((lassoc 'ops optionlist)
       (|throwKeyedMsg| 's2it0004 nil))
     (t
       (setq oplist
        (cond
         (tracelist (list (cons 'ops tracelist)))
         (t nil)))
       (setq varlist
        (cond
         ((setq y (lassoc '|vars| optionlist))
           (list (cons '|vars| y)))
         (t nil)))
       (append domainlist (append oplist varlist)))))
  (optionlist (append tracelist optionlist))
  (t tracelist)))
(|/TRACE,0|
 (prog (t4)
  (setq t4 nil)
  (return
   (do ((t5 argument (cdr t5)) (|funName| nil))
       ((or (atom t5)
            (progn (setq |funName| (car t5)) nil))
        (nreverse0 t4))
    (seq
     (exit
      (setq t4 (cons |funName| t4)))))))
(|saveMapSig|
 (prog (t6)
```

```
          (setq t6 nil)
          (return
           (do ((t7 argument (cdr t7)) (|funName| nil))
               ((or (atom t7)
                    (progn (setq |funName| (car t7)) nil))
                 (nreverse0 t6))
            (seq
             (exit
              (setq t6 (cons |funName| t6))))))))))))))))
```

38.1.5 defun getTraceOptions

⟨*defun getTraceOptions*⟩≡

```
(defun |getTraceOptions| (|options|)
 (prog (|$traceErrorStack| optionlist temp1 key |parms|)
  (declare (special |$traceErrorStack|))
  (return
   (seq
    (progn
     (setq |$traceErrorStack| nil)
     (setq optionlist
      (prog (t0)
       (setq t0 nil)
       (return
        (do ((t1 |options| (cdr t1)) (x nil))
            ((or (atom t1) (progn (setq x (car t1)) nil)) (nreverse0 t0))
         (seq
          (exit
           (setq t0 (cons (|getTraceOption| x) t0))))))))))
    (cond
     (|$traceErrorStack|
      (cond
       ((null (cdr |$traceErrorStack|))
        (setq temp1 (car |$traceErrorStack|))
        (setq key (car temp1))
        (setq |parms| (cadr temp1))
        (|throwKeyedMsg| key (cons "" |parms|)))
       (t
        (|throwListOfKeyedMsgs| 's2it0017
         (cons (|#| |$traceErrorStack|) nil)
         (nreverse |$traceErrorStack|)))))
     (t optionlist)))))))
```

38.1.6 defun saveMapSig

⟨*defun saveMapSig*⟩≡
```
(defun |saveMapSig| (funnames)
 (let (map)
 (declare (special |$tracedMapSignatures| |$mapSubNameAlist|))
  (dolist (name funnames)
   (when (setq map (|rassoc| name |$mapSubNameAlist|))
     (setq |$tracedMapSignatures|
       (addassoc name (|getMapSig| map name) |$tracedMapSignatures|))))))
```

38.1.7 defun getMapSig

⟨*defun getMapSig*⟩≡
```
(defun |getMapSig| (mapname subname)
 (let (lmms sig)
 (declare (special |$InteractiveFrame|))
  (when (setq lmms (|get| mapname '|localModemap| |$InteractiveFrame|))
    (do ((t0 lmms (cdr t0)) (|mm| nil) (t1 nil sig))
        ((or (atom t0) (progn (setq |mm| (car t0)) nil) t1) nil)
      (when (boot-equal (cadr |mm|) subname) (setq sig (cdar |mm|))))
    sig)))
```

38.1.8 defun getTraceOption

⟨*defun getTraceOption,hn*⟩≡
```
(defun |getTraceOption,hn| (x)
 (prog (g)
  (return
   (seq
    (if (and (atom x) (null (upper-case-p (elt (stringimage x) 0))))
     (exit
      (seq
       (if (|isDomainOrPackage| (eval x)) (exit x))
       (exit
        (|stackTraceOptionError|
         (cons 's2it0013 (cons (cons x nil) nil)))))))
    (if (setq g (|domainToGenvar| x)) (exit g))
    (exit
     (|stackTraceOptionError| (cons 's2it0013 (cons (cons x nil) nil))))))))
```

⟨*defun getTraceOption*⟩≡

```
(defun |getTraceOption| (arg)
 (prog (l |opts| key a |n|)
 (declare (special |$traceOptionList|))
  (return
   (seq
    (progn
     (setq key (car arg))
     (setq l (cdr arg))
     (setq key
       (|selectOptionLC| key |$traceOptionList| '|traceOptionError|))
     (setq arg (cons key l))
     (cond
      ((memq key '(|nonquietly| |timer| |nt|)) arg)
      ((eq key '|break|)
       (cond
        ((null l) (cons '|break| (cons '|before| nil)))
        (t
          (setq |opts|
           (prog (t0)
            (setq t0 nil)
            (return
             (do ((t1 l (cdr t1)) (y nil))
                 ((or (atom t1)
                      (progn (setq y (car t1)) nil))
                    (nreverse0 t0))
              (seq
               (exit
                (setq t0
                 (cons
                  (|selectOptionLC| y '(|before| |after|) nil) t0))))))))
          (cond
           ((prog (t2)
             (setq t2 t)
             (return
              (do ((t3 nil (null t2)) (t4 |opts| (cdr t4)) (y nil))
                  ((or t3 (atom t4) (progn (setq y (car t4)) nil)) t2)
                 (seq
                  (exit
                   (setq t2 (and t2 (identp y))))))))
            (cons '|break| |opts|))
           (t
            (|stackTraceOptionError| (cons 's2it0008 (cons nil nil))))))))
      ((eq key '|restore|)
       (cond
        ((null l) arg)
```

```
  (t
   (|stackTraceOptionError|
    (cons 's2it0009
     (cons (cons (strconc ")" (|object2String| key)) nil) nil))))))
((eq key '|only|) (cons '|only| (|transOnlyOption| l)))
((eq key '|within|)
  (cond
   ((and (pairp l)
         (eq (qcdr l) nil)
         (progn (setq a (qcar l)) t)
         (identp a))
     arg)
   (t
     (|stackTraceOptionError|
      (cons 's2it0010 (cons (cons ")within" nil) nil))))))
((memq key '(|cond| |before| |after|))
  (setq key
   (cond
    ((eq key '|cond|) '|when|)
    (t key)))
  (cond
   ((and (pairp l)
         (eq (qcdr l) nil)
         (progn (setq a (qcar l)) t))
      (cons key l))
   (t
    (|stackTraceOptionError|
     (cons 's2it0011
      (cons
       (cons (strconc ")"
        (|object2String| key)) nil) nil))))))
((eq key '|depth|)
  (cond
   ((and (pairp l)
         (eq (qcdr l) nil)
         (progn (setq |n| (qcar l)) t)
         (fixp |n|))
     arg)
   (t
    (|stackTraceOptionError|
     (cons 's2it0012 (cons (cons ")depth" nil) nil))))))
((eq key '|count|)
  (cond
   ((or (null l)
        (and (pairp l)
             (eq (qcdr l) nil)
```

```
                (progn (setq |n| (qcar l)) t)
                (fixp |n|)))
        arg)
     (t
      (|stackTraceOptionError|
        (cons 's2it0012 (cons (cons ")count" nil) nil))))))
((eq key '|of|)
  (cons '|of|
   (prog (t5)
    (setq t5 nil)
    (return
     (do ((t6 l (cdr t6)) (y nil))
         ((or (atom t6) (progn (setq y (car t6)) nil)) (nreverse0 t5))
       (seq
        (exit
         (setq t5 (cons (|getTraceOption,hn| y) t5)))))))))
((memq key '(|local| ops |vars|))
  (cond
   ((or (null l)
        (and (pairp l) (eq (qcdr l) nil) (eq (qcar l) '|all|)))
     (cons key '|all|))
   ((|isListOfIdentifiersOrStrings| l) arg)
   (t
     (|stackTraceOptionError|
      (cons 's2it0015
       (cons
        (cons (strconc ")" (|object2String| key)) nil) nil))))))
((eq key '|varbreak|)
  (cond
   ((or (null l)
        (and (pairp l) (eq (qcdr l) nil) (eq (qcar l) '|all|)))
     (cons '|varbreak| '|all|))
   ((|isListOfIdentifiers| l) arg)
   (t
     (|stackTraceOptionError|
      (cons 's2it0016
       (cons
        (cons (strconc ")" (|object2String| key)) nil) nil))))))
((eq key '|mathprint|)
  (cond
   ((null l) arg)
   (t
     (|stackTraceOptionError|
      (cons 's2it0009
       (cons
        (cons (strconc ")" (|object2String| key)) nil) nil))))))
```

```
        (key (|throwKeyedMsg| 's2it0005 (cons key nil)))))))))
```

38.1.9 defun traceOptionError

⟨*defun traceOptionError*⟩≡

```
(defun |traceOptionError| (opt keys)
 (if (null keys)
  (|stackTraceOptionError| (cons 's2it0007 (cons (cons opt nil) nil)))
  (|commandAmbiguityError| '|trace option| opt keys)))
```

38.1.10 defun resetTimers

⟨*defun resetTimers*⟩≡

```
(defun |resetTimers| ()
 (declare (special /timerlist))
 (dolist (timer /timerlist)
  (set (intern (strconc timer ",TIMER")) 0)))
```

38.1.11 defun resetSpacers

⟨*defun resetSpacers*⟩≡

```
(defun |resetSpacers| ()
 (declare (special /spacelist))
 (dolist (spacer /spacelist)
   (set (intern (strconc spacer ",SPACE")) 0)))
```

38.1.12 defun resetCounters

⟨*defun resetCounters*⟩≡

```
(defun |resetCounters| ()
 (declare (special /countlist))
 (dolist (k /countlist)
  (set (intern (strconc k ",COUNT")) 0)))
```

38.1.13 defun ptimers

⟨*defun ptimers*⟩≡
```
(defun |ptimers| ()
  (declare (special /timerlist |$timerTicksPerSecond|))
  (if (null /timerlist)
   (|sayBrightly| "   no functions are timed")
   (dolist (timer /timerlist)
     (|sayBrightly|
      `("  " ,@(|bright| timer) |:| " "
         ,(quotient (eval (intern (strconc timer ",TIMER")))
            (|float| |$timerTicksPerSecond|)) " sec.")))))
```

38.1.14 defun pspacers

⟨*defun pspacers*⟩≡
```
(defun |pspacers| ()
 (declare (special /spacelist))
 (if (null /spacelist)
   (|sayBrightly| "   no functions have space monitored")
   (dolist (spacer /spacelist)
     (|sayBrightly|
      `("  " ,@(|bright| spacer) |: |
        ,(eval (intern (strconc spacer ",SPACE"))) " bytes")))))
```

38.1.15 defun pcounters

⟨*defun pcounters*⟩≡
```
(defun |pcounters| ()
 (declare (special /countlist))
 (if (null /countlist)
   (|sayBrightly| "   no functions are being counted")
   (dolist (k /countlist)
     (|sayBrightly|
      `("  " ,@(|bright| k) |:| " " ,(eval (intern (strconc k ",COUNT")))
        " times")))))
```

38.1.16 defun transOnlyOption

⟨*defun transOnlyOption*⟩≡
```
(defun |transOnlyOption| (arg)
 (let (y n)
  (when (and (pairp arg) (progn (setq n (qcar arg)) (setq y (qcdr arg)) t))
    (cond
     ((fixp n) (cons n (|transOnlyOption| y)))
     ((memq (setq n (upcase n)) '(v a c)) (cons n (|transOnlyOption| y)))
     (t
        (|stackTraceOptionError| (cons 's2it0006 (list (list n))))
        (|transOnlyOption| y))))))
```

38.1.17 defun stackTraceOptionError

⟨*defun stackTraceOptionError*⟩≡
```
(defun |stackTraceOptionError| (x)
 (declare (special |$traceErrorStack|))
 (push x |$traceErrorStack|)
 nil)
```

38.1.18 defun removeOption

⟨*defun removeOption*⟩≡
```
(defun |removeOption| (op options)
 (let (opt t0)
  (do ((t1 options (cdr t1)) (optentry nil))
      ((or (atom t1)
           (progn (setq optentry (car t1)) nil)
           (progn (progn (setq opt (car optentry)) optentry) nil))
    (nreverse0 t0))
  (when (nequal opt op) (setq t0 (cons optentry t0))))))
```

38.1.19 defun domainToGenvar

⟨*defun domainToGenvar*⟩≡
```
(defun |domainToGenvar| (arg)
 (let (|$doNotAddEmptyModeIfTrue| y g)
 (declare (special |$doNotAddEmptyModeIfTrue|))
  (setq |$doNotAddEmptyModeIfTrue| t)
  (when
   (and (setq y (|unabbrevAndLoad| arg))
        (eq (getdatabase (|opOf| y) 'constructorkind) '|domain|))
     (setq g (|genDomainTraceName| y))
     (set g (|evalDomain| y))
     g)))
```

38.1.20 defun genDomainTraceName

⟨*defun genDomainTraceName*⟩≡
```
(defun |genDomainTraceName| (y)
 (let (u g)
 (declare (special |$domainTraceNameAssoc|))
   (if (setq u (lassoc y |$domainTraceNameAssoc|))
    u
    (progn
     (setq g (genvar))
     (setq |$domainTraceNameAssoc| (cons (cons y g) |$domainTraceNameAssoc|))
     g))))
```

38.1.21 defun untrace

⟨*defun untrace*⟩≡

```
(defun |untrace| (arg)
 (let (untracelist)
 (declare (special |$lastUntraced| /tracenames |$mapSubNameAlist|))
  (if arg
   (setq |$lastUntraced| arg)
   (setq |$lastUntraced| (copy /tracenames)))
  (setq untracelist
   (do ((t1 arg (cdr t1)) (x nil) (t0 nil))
       ((or (atom t1) (progn (setq x (car t1)) nil))
      (nreverse0 t0))
     (push (|transTraceItem| x) t0)))
  (|/UNTRACE,0|
   (do ((t3 untracelist (cdr t3)) (|funName| nil) (t2 nil))
       ((or (atom t3) (progn (setq |funName| (car t3)) nil))
     (nreverse0 t2))
    (push (|lassocSub| |funName| |$mapSubNameAlist|) t2)))
  (|removeTracedMapSigs| untracelist)))
```

38.1.22 defun transTraceItem

⟨*defun transTraceItem*⟩≡

```
(defun |transTraceItem| (x)
 (prog (|$doNotAddEmptyModeIfTrue| |value| y)
  (declare (special |$doNotAddEmptyModeIfTrue|))
  (return
   (progn
    (setq |$doNotAddEmptyModeIfTrue| t)
    (cond
     ((atom x)
       (cond
        ((and (setq |value| (|get| x '|value| |$InteractiveFrame|))
              (|member| (|objMode| |value|)
                '((|Mode|) (|Domain|) (|SubDomain| (|Domain|)))))
          (setq x (|objVal| |value|))
          (cond
           ((setq y (|domainToGenvar| x)) y)
           (t x)))
        ((upper-case-p (elt (stringimage x) 0))
          (setq y (|unabbrev| x))
          (cond
           ((|constructor?| y) y)
           ((and (pairp y) (|constructor?| (car y))) (car y))
           ((setq y (|domainToGenvar| x)) y)
           (t x)))
        (t x)))
      ((vecp (car x)) (|transTraceItem| (|devaluate| (car x))))
      ((setq y (|domainToGenvar| x)) y)
      (t (|throwKeyedMsg| 's2it0018 (cons x nil))))))))
```

38.1.23 defun removeTracedMapSigs

⟨*defun removeTracedMapSigs*⟩≡

```
(defun |removeTracedMapSigs| (untraceList)
 (declare (special |$tracedMapSignatures|))
 (dolist (name untraceList)
   (remprop name |$tracedMapSignatures|)))
```

38.1.24 defun coerceTraceArgs2E

⟨*defun coerceTraceArgs2E*⟩≡

```
(defun |coerceTraceArgs2E| (tracename subname args)
 (declare (ignore tracename))
 (let (name)
  (declare (special |$OutputForm| |$mathTraceList| |$tracedMapSignatures|))
  (cond
   ((memq (setq name subname) |$mathTraceList|)
     (if (spadsysnamep (pname name))
      (|coerceSpadArgs2E| (reverse (cdr (reverse args))))
      (do ((t1 '(|arg1| |arg2| |arg3| |arg4| |arg5| |arg6| |arg7| |arg8|
                 |arg9| |arg10| |arg11| |arg12| |arg13| |arg14| |arg15|
                 |arg16| |arg17| |arg18| |arg19|) (cdr t1))
           (name nil)
           (t2 args (cdr t2))
           (arg nil)
           (t3 (cdr (lassoc subname |$tracedMapSignatures|)) (cdr t3))
           (type nil)
           (t0 nil))
          ((or (atom t1)
               (progn (setq name (car t1)) nil)
               (atom t2)
               (progn (setq arg (car t2)) nil)
               (atom t3)
               (progn (setq type (car t3)) nil))
            (nreverse0 t0))
          (setq t0
           (cons
            (list '= name
             (|objValUnwrap|
              (|coerceInteractive|
               (|objNewWrap| arg type) |$OutputForm|))) t0)))))
  ((spadsysnamep (pname name)) (reverse (cdr (reverse args))))
  (t args))))
```

38.1.25 defun coerceSpadArgs2E

⟨*defun coerceSpadArgs2E*⟩≡

```
(defun |coerceSpadArgs2E| (args)
 (let ((|$streamCount| 0))
  (declare (special |$streamCount| |$OutputForm| |$tracedSpadModemap|))
  (do ((t1 '(|arg1| |arg2| |arg3| |arg4| |arg5| |arg6| |arg7| |arg8|
             |arg9| |arg10| |arg11| |arg12| |arg13| |arg14| |arg15|
             |arg16| |arg17| |arg18| |arg19|) (cdr t1))
       (name nil)
       (t2 args (cdr t2))
       (arg nil)
       (t3 (cdr |$tracedSpadModemap|) (cdr t3))
       (type nil)
       (t0 nil))
      ((or (atom t1)
           (progn (setq name (car t1)) nil)
           (atom t2)
           (progn (setq arg (car t2)) nil)
           (atom t3)
           (progn (setq type (car t3)) nil))
        (nreverse0 t0))
    (seq
     (exit
      (setq t0
       (cons
        (cons '=
         (cons name
          (cons (|objValUnwrap|
            (|coerceInteractive|
             (|objNewWrap| arg type)
             |$OutputForm|)) nil)))
         t0)))))))
```

38.1.26 defun subTypes

⟨*defun subTypes*⟩≡
```
(defun |subTypes| (|mm| |sublist|)
 (prog (s)
  (return
   (seq
    (cond
     ((atom |mm|)
       (cond ((setq s (lassoc |mm| |sublist|)) s) (t |mm|)))
     (t
      (prog (t0)
       (setq t0 nil)
       (return
        (do ((t1 |mm| (cdr t1)) (|m| nil))
            ((or (atom t1) (progn (setq |m| (car t1)) nil)) (nreverse0 t0))
         (seq
          (exit
           (setq t0 (cons (|subTypes| |m| |sublist|) t0))))))))))))
```

38.1.27 defun coerceTraceFunValue2E

⟨*defun coerceTraceFunValue2E*⟩≡
```
(defun |coerceTraceFunValue2E| (tracename subname |value|)
 (let (name u)
 (declare (special |$tracedMapSignatures| |$OutputForm| |$mathTraceList|))
   (if (memq (setq name subname) |$mathTraceList|)
    (cond
     ((spadsysnamep (pname tracename)) (|coerceSpadFunValue2E| |value|))
     ((setq u (lassoc subname |$tracedMapSignatures|))
       (|objValUnwrap|
        (|coerceInteractive| (|objNewWrap| |value| (car u)) |$OutputForm|)))
     (t |value|))
    |value|)))
```

38.1.28 defun coerceSpadFunValue2E

⟨*defun coerceSpadFunValue2E*⟩≡
```
(defun |coerceSpadFunValue2E| (|value|)
 (let (|$streamCount|)
  (declare (special |$streamCount| |$tracedSpadModemap| |$OutputForm|))
  (setq |$streamCount| 0)
  (|objValUnwrap|
   (|coerceInteractive|
    (|objNewWrap| |value| (car |$tracedSpadModemap|))
    |$OutputForm|))))
```

38.1.29 defun isListOfIdentifiers

⟨*defun isListOfIdentifiers*⟩≡
```
(defun |isListOfIdentifiers| (arg)
 (prog ()
  (return
   (seq
    (prog (t0)
     (setq t0 t)
     (return
      (do ((t1 nil (null t0)) (t2 arg (cdr t2)) (x nil))
          ((or t1 (atom t2) (progn (setq x (car t2)) nil)) t0)
       (seq
        (exit
         (setq t0 (and t0 (identp x))))))))))))
```

38.1.30 defun isListOfIdentifiersOrStrings

⟨*defun isListOfIdentifiersOrStrings*⟩≡

```
(defun |isListOfIdentifiersOrStrings| (arg)
 (prog ()
  (return
   (seq
    (prog (t0)
     (setq t0 t)
     (return
      (do ((t1 nil (null t0)) (t2 arg (cdr t2)) (x nil))
          ((or t1 (atom t2) (progn (setq x (car t2)) nil)) t0)
       (seq
        (exit
         (setq t0 (and t0 (or (identp x) (stringp x))))))))))))
```

38.1.31 defun getMapSubNames

⟨*defun getMapSubNames*⟩≡

```
(defun |getMapSubNames| (arg)
 (let (lmm subs)
 (declare (special /tracenames |$lastUntraced| |$InteractiveFrame|))
  (setq subs nil)
  (dolist (mapname arg)
   (when (setq lmm (|get| mapname '|localModemap| |$InteractiveFrame|))
     (setq subs
      (append
        (do ((t2 lmm (cdr t2)) (t1 nil) (|mm| nil))
            ((or (atom t2)
                 (progn (setq |mm| (CAR t2)) nil)) (nreverse0 t1))
           (setq t1 (cons (cons mapname (cadr |mm|)) t1)))
       subs))))
  (|union| subs
    (|getPreviousMapSubNames| (unionq /tracenames |$lastUntraced|)))))
```

38.1.32 defun getPreviousMapSubNames

⟨*defun getPreviousMapSubNames*⟩≡

```
(defun |getPreviousMapSubNames| (|traceNames|)
 (prog (lmm subs)
  (return
   (seq
    (progn
     (setq subs nil)
     (seq
      (do ((t0 (assocleft (caar |$InteractiveFrame|)) (cdr t0))
           (mapname nil))
          ((or (atom t0) (progn (setq mapname (car t0)) nil)) nil)
       (seq
        (exit
         (cond
          ((setq lmm
             (|get| mapname '|localModemap| |$InteractiveFrame|))
            (exit
             (cond
              ((memq (cadar lmm) |traceNames|)
               (exit
                (do ((t1 lmm (cdr t1)) (|mm| nil))
                    ((or (atom t1) (progn (setq |mm| (car t1)) nil)) nil)
                 (seq
                  (exit
                   (setq subs
                    (cons (cons mapname (cadr |mm|)) subs)))))))))))))
      (exit subs)))))))
```

38.1.33 defun lassocSub

⟨*defun lassocSub*⟩≡

```
(defun |lassocSub| (x subs)
 (let (y)
  (if (setq y (lassq x subs))
    y
    x)))
```

38.1.34 defun rassocSub

⟨*defun rassocSub*⟩≡
```
(defun |rassocSub| (x subs)
 (let (y)
   (if (setq y (|rassoc| x subs))
    y
    x)))
```

38.1.35 defun isUncompiledMap

⟨*defun isUncompiledMap*⟩≡
```
(defun |isUncompiledMap| (x)
 (let (y)
 (declare (special |$InteractiveFrame|))
  (when (setq y (|get| x '|value| |$InteractiveFrame|))
    (and
      (eq (caar y) 'map)
      (null (|get| x '|localModemap| |$InteractiveFrame|))))))
```

38.1.36 defun isInterpOnlyMap

⟨*defun isInterpOnlyMap*⟩≡
```
(defun |isInterpOnlyMap| (map)
 (let (x)
 (declare (special |$InteractiveFrame|))
  (when (setq x (|get| map '|localModemap| |$InteractiveFrame|))
    (eq (caaar x) '|interpOnly|))))
```

38.1.37 defun augmentTraceNames

⟨*defun augmentTraceNames*⟩≡
```
(defun |augmentTraceNames| (arg)
 (let (mml res)
 (declare (special |$InteractiveFrame|))
 (dolist (tracename arg)
  (if (setq mml (|get| tracename '|localModemap| |$InteractiveFrame|))
   (setq res
    (append
     (prog (t1)
      (setq t1 nil)
      (return
       (do ((t2 mml (cdr t2)) (|mm| nil))
           ((or (atom t2)
                (progn (setq |mm| (CAR t2)) nil))
             (nreverse0 t1))
          (setq t1 (cons (cadr |mm|) t1)))))
     res))
   (setq res (cons tracename res))))
 res))
```

38.1.38 defun isSubForRedundantMapName

⟨*defun isSubForRedundantMapName*⟩≡
```
(defun |isSubForRedundantMapName| (subname)
 (let (mapname tail)
 (declare (special |$mapSubNameAlist|))
  (when (setq mapname (|rassocSub| subname |$mapSubNameAlist|))
    (when (setq tail (|member| (cons mapname subname) |$mapSubNameAlist|))
      (memq mapname (cdr (assocleft tail)))))))
```

38.1.39 defun untraceMapSubNames

⟨*defun untraceMapSubNames*⟩≡
```
(defun |untraceMapSubNames| (|traceNames|)
 (let (|$mapSubNameAlist| subs)
 (declare (special |$mapSubNameAlist| |$lastUntraced|))
  (if
   (null (setq |$mapSubNameAlist| (|getPreviousMapSubNames| |traceNames|)))
   nil
   (dolist (name (setq subs (assocright |$mapSubNameAlist|)))
     (when (memq name /tracenames)
       (|/UNTRACE,2| name nil)
       (setq |$lastUntraced| (setdifference |$lastUntraced| subs)))))))
```

38.1.40 defmacro funfind

⟨*defun funfind,LAM*⟩≡
```
(defun |funfind,LAM| (functor opname)
 (prog (ops tmp1)
  (return
   (seq
    (progn
     (setq ops (|isFunctor| functor))
     (prog (t0)
      (setq t0 nil)
      (return
       (do ((t1 ops (cdr t1)) (u nil))
           ((or (atom t1) (progn (setq u (car t1)) nil)) (nreverse0 t0))
        (seq
         (exit
          (cond
           ((and (pairp u)
                 (progn
                  (setq tmp1 (qcar u))
                  (and (pairp tmp1) (equal (qcar tmp1) opname))))
             (setq t0 (cons u t0))))))))))))))
```

⟨*defmacro funfind*⟩≡
```
(defmacro |funfind| (&whole t0 &rest notused &aux t1)
 (declare (ignore notused))
 (dsetq t1 t0)
 (cons '|funfind,LAM| (vmlisp::wrap (cdr t1) '(quote quote))))
```

38.1.41 defun isDomainOrPackage

⟨*defun isDomainOrPackage*⟩≡
```
(defun |isDomainOrPackage| (dom)
 (and
  (refvecp dom)
  (> (|#| dom) 0)
  (|isFunctor| (|opOf| (elt dom 0)))))
```

38.1.42 defun isTraceGensym

⟨*defun isTraceGensym*⟩≡
```
(defun |isTraceGensym| (x)
 (gensymp x))
```

38.1.43 defun spadTrace,g

⟨*defun spadTrace,g*⟩≡
```
(defun |spadTrace,g| (x)
  (if (stringp x) (intern x) x))
```

38.1.44 defun spadTrace,isTraceable

⟨*defun spadTrace,isTraceable*⟩≡

```
(defun |spadTrace,isTraceable| (x |domain|)
 (prog (n |functionSlot|)
  (return
   (seq
    (progn
     (setq n (caddr x))
     x
     (seq
      (if (atom (elt |domain| n)) (exit nil))
      (setq |functionSlot| (car (elt |domain| n)))
      (if (gensymp |functionSlot|)
       (exit (seq (|reportSpadTrace| '|Already Traced| x) (exit nil))))
      (if (null (bpiname |functionSlot|))
       (exit
        (seq
         (|reportSpadTrace| '|No function for| x)
         (exit nil))))
      (exit t)))))))
```

38.1.45 defun spadTrace

⟨*defun spadTrace*⟩≡

```
(defun |spadTrace| (domain options)
 (let (|$tracedModemap| listofoperations listofvariables
       listofbreakvars anyiftrue domainid currententry
       currentalist opstructurelist sig kind triple fn op
       mm n alias tracename sigslotnumberalist)
 (declare (special |$tracedModemap| /tracenames |$fromSpadTrace| |$letAssoc|
    |$reportSpadTrace| |$traceNoisely|))
 (setq |$fromSpadTrace| t)
 (setq |$tracedModemap| nil)
 (cond
  ((and (pairp domain)
        (refvecp (car domain))
        (eql (elt (car domain) 0) 0))
    (|aldorTrace| domain options))
  ((null (|isDomainOrPackage| domain))
    (|userError| "bad argument to trace"))
  (t
   (setq listofoperations
    (prog (t0)
     (setq t0 nil)
     (return
      (do ((t1 (|getOption| 'ops options) (cdr t1)) (x nil))
          ((or (atom t1) (progn (setq x (car t1)) nil)) (nreverse0 t0))
       (seq
        (exit
         (setq t0 (cons (|spadTrace,g| x) t0))))))))
   (cond
    ((setq listofvariables (|getOption| 'vars options))
      (setq options (|removeOption| 'vars options))))
   (cond
    ((setq listofbreakvars (|getOption| 'varbreak options))
      (setq options (|removeOption| 'varbreak options))))
   (setq anyiftrue (null listofoperations))
   (setq domainid (|opOf| (elt domain 0)))
   (setq currententry (|assoc| domain /tracenames))
   (setq currentalist (kdr currententry))
   (setq opstructurelist
    (|flattenOperationAlist| (|getOperationAlistFromLisplib| domainid)))
   (setq sigslotnumberalist
    (prog (t2)
     (setq t2 nil)
     (return
      (do ((t3 opstructurelist (cdr t3)) (t4 nil))
```

```
          ((or (atom t3)
               (progn (setq t4 (CAR t3)) nil)
               (progn
                (progn
                 (setq op (car t4))
                 (setq sig (cadr t4))
                 (setq n (caddr t4))
                 (setq kind (car (cddddr t4))) t4)
                nil))
             (nreverse0 t2))
      (seq
       (exit
        (cond
         ((and (eq kind 'elt)
               (or anyiftrue (memq op listofoperations))
               (fixp n)
               (|spadTrace,isTraceable|
                (setq triple
                 (cons op (cons sig (cons n nil)))) domain))
            (setq t2 (cons triple t2)))))))))
(cond
 (listofvariables
  (do ((t5 sigslotnumberalist (cdr t5)) (t6 nil))
      ((or (atom t5)
           (progn (setq t6 (car t5)) nil)
           (progn (progn (setq n (caddr t6)) t6) nil))
       nil)
    (seq
     (exit
      (progn
       (setq fn (car (elt domain n)))
       (setq |$letAssoc|
        (as-insert (bpiname fn) listofvariables |$letAssoc|))))))))
(cond
 (listofbreakvars
  (do ((t7 sigslotnumberalist (cdr t7)) (t8 nil))
      ((or (atom t7)
           (progn (setq t8 (car t7)) nil)
           (progn (progn (setq n (caddr t8)) t8) nil))
          nil)
    (seq
     (exit
      (progn
       (setq fn (car (elt domain n)))
       (setq |$letAssoc|
        (as-insert (bpiname fn)
```

```
        (cons (cons 'break listofbreakvars) nil) |$letAssoc|)))))))))
(do ((t9 sigslotnumberalist (cdr t9)) (|pair| nil))
    ((or (atom t9)
         (progn (setq |pair| (car t9)) nil)
         (progn
          (progn
           (setq op (car |pair|))
           (setq mm (cadr |pair|))
           (setq n (caddr |pair|))
           |pair|)
          nil))
        nil)
 (seq
  (exit
   (progn
    (setq alias (|spadTraceAlias| domainid op n))
    (setq |$tracedModemap|
      (|subTypes| mm (|constructSubst| (elt domain 0))))
    (setq tracename
      (bpitrace (car (elt domain n)) alias options))
    (nconc |pair|
      (cons listofvariables
       (cons (car (elt domain n))
        (cons tracename (cons alias nil)))))
    (rplac (car (elt domain n)) tracename)))))
(setq sigslotnumberalist
 (prog (t10)
  (setq t10 nil)
  (return
   (do ((t11 sigslotnumberalist (cdr t11)) (x nil))
       ((or (atom t11) (progn (setq x (car t11)) nil)) (nreverse0 t10))
    (seq
     (exit
      (cond ((cdddr x) (setq t10 (cons x t10)))))))))))
(cond
 (|$reportSpadTrace|
  (cond (|$traceNoisely| (|printDashedLine|)))
  (do ((t12 (|orderBySlotNumber| sigslotnumberalist) (cdr t12))
       (x nil))
      ((or (atom t12)
           (progn (setq x (car t12)) nil))
          nil)
       (seq (exit (|reportSpadTrace| 'tracing x))))))
(cond (|$letAssoc| (setletprintflag t)))
(cond
 (currententry
```

```
       (rplac (cdr currententry)
         (append sigslotnumberalist currentalist)))
      (t
       (setq /tracenames
        (cons (cons domain sigslotnumberalist) /tracenames))
        (|spadReply|)))))))
```

38.1.46 defun traceDomainLocalOps

⟨*defun traceDomainLocalOps*⟩≡

```
(defun |traceDomainLocalOps| ()
  (|sayMSG| '("  The )local option has been withdrawn"))
  (|sayMSG| '("  Use )ltr to trace local functions.")))
```

38.1.47 defun untraceDomainLocalOps

⟨*defun untraceDomainLocalOps*⟩≡

```
(defun |untraceDomainLocalOps| ()
  (|sayMSG| '("  The )local option has been withdrawn"))
  (|sayMSG| '("  Use )ltr to trace local functions.")))
```

38.1.48 defun traceDomainConstructor

⟨*defun traceDomainConstructor*⟩≡

```
(defun |traceDomainConstructor| (|domainConstructor| options)
 (prog (|listOfLocalOps| |argl| |domain| |innerDomainConstructor|)
 (declare (special |$ConstructorCache|))
  (return
   (seq
    (progn
     (|loadFunctor| |domainConstructor|)
     (setq |listOfLocalOps| (|getOption| 'local options))
     (when |listOfLocalOps| (|traceDomainLocalOps|))
     (cond
      ((and |listOfLocalOps| (null (|getOption| 'ops options))) nil)
      (t
       (do ((t2 (hget |$ConstructorCache| |domainConstructor|) (cdr t2))
            (t3 nil))
           ((or (atom t2)
                (progn (setq t3 (car t2)) nil)
                (progn
                 (progn
                  (setq |argl| (car t3))
                  (setq |domain| (cddr t3)) t3)
                 nil))
               nil)
         (seq
          (exit
           (|spadTrace| |domain| options))))
       (setq /tracenames (cons |domainConstructor| /tracenames))
       (setq |innerDomainConstructor|
         (intern (strconc |domainConstructor| ";")))
       (cond
        ((fboundp |innerDomainConstructor|)
          (setq |domainConstructor| |innerDomainConstructor|)))
       (embed |domainConstructor|
        (cons 'lambda
         (cons
          (cons '&rest
           (cons 'args nil))
          (cons
           (cons 'prog
            (cons
             (cons '|domain| nil)
             (cons
              (cons 'setq
               (cons '|domain|
```

```
                (cons
                 (cons 'apply (cons |domainConstructor|
                  (cons 'args nil))) nil)))
               (cons
                (cons '|spadTrace|
                 (cons '|domain|
                  (cons (mkq options) nil)))
                (cons (cons 'return (cons '|domain| nil)) nil)))))
          nil)))))))))))
```

38.1.49 defun untraceDomainConstructor

⟨*defun untraceDomainConstructor,keepTraced?*⟩≡

```
(defun |untraceDomainConstructor,keepTraced?| (df |domainConstructor|)
 (prog (dc)
  (return
   (seq
    (if (and
         (and
          (and (pairp df) (progn (setq dc (qcar df)) t))
           (|isDomainOrPackage| dc))
          (boot-equal (kar (|devaluate| dc)) |domainConstructor|))
     (exit (seq (|/UNTRACE,0| (cons dc nil)) (exit nil))))
    (exit t)))))
```

⟨*defun untraceDomainConstructor*⟩≡

```
(defun |untraceDomainConstructor| (|domainConstructor|)
 (prog (|innerDomainConstructor|)
 (declare (special /tracenames))
  (return
   (seq
    (progn
     (setq /tracenames
      (prog (t0)
       (setq t0 nil)
       (return
        (do ((t1 /tracenames (cdr t1)) (df nil))
            ((or (atom t1) (progn (setq df (car t1)) nil)) (nreverse0 t0))
         (seq
          (exit
           (cond ((|untraceDomainConstructor,keepTraced?|
                    df |domainConstructor|)
             (setq t0 (cons df t0)))))))))
     (setq |innerDomainConstructor|
      (intern (strconc |domainConstructor| ";")))
     (cond
      ((fboundp |innerDomainConstructor|) (unembed |innerDomainConstructor|))
      (t (unembed |domainConstructor|)))
     (setq /tracenames (|delete| |domainConstructor| /tracenames)))))))
```

38.1.50 defun flattenOperationAlist

⟨*defun flattenOperationAlist*⟩≡

```
(defun |flattenOperationAlist| (|opAlist|)
 (prog (op |mmList| |res|)
  (return
   (seq
    (progn
     (setq |res| nil)
     (do ((t0 |opAlist| (cdr t0)) (t1 nil))
         ((or (atom t0)
              (progn (setq t1 (car t0)) nil)
              (progn
               (progn (setq op (car t1)) (setq |mmList| (cdr t1)) t1)
               nil))
             nil)
      (seq
       (exit
        (setq |res|
         (append |res|
          (prog (t2)
           (setq t2 nil)
           (return
            (do ((t3 |mmList| (cdr t3)) (mm nil))
                ((or (atom t3)
                     (progn (setq mm (car t3)) nil)) (nreverse0 t2))
             (seq
              (exit
               (setq t2 (cons (cons op mm) t2))))))))))
     |res|)))))
```

38.1.51 defun mapLetPrint

⟨*defun mapLetPrint*⟩≡

```
(defun |mapLetPrint| (x val currentFunction)
  (setq x (|getAliasIfTracedMapParameter| x currentFunction))
  (setq currentFunction (|getBpiNameIfTracedMap| currentFunction))
  (|letPrint| x val currentFunction))
```

38.1.52 defun letPrint

⟨*defun letPrint*⟩≡

```
(defun |letPrint| (x |val| |currentFunction|)
 (prog (y)
 (declare (special |$letAssoc|))
  (return
   (progn
    (cond ((and |$letAssoc|
                (or
                 (setq y (lassoc |currentFunction| |$letAssoc|))
                 (setq y (lassoc '|all| |$letAssoc|))))
     (cond
      ((and (or (eq y '|all|)
                 (memq x y))
            (null
             (or (is_genvar x) (|isSharpVarWithNum| x) (gensymp x))))
       (|sayBrightlyNT| (append (|bright| x) (cons '|: | nil)))
       (prin0 (|shortenForPrinting| |val|))
       (terpri)))
     (cond
      ((and (setq y (|hasPair| 'break y))
            (or (eq y '|all|)
                (and (memq x y)
                     (null (memq (elt (pname x) 0) '($ |#|)))
                     (null (gensymp x)))))
       (|break|
        (append
         (|bright| |currentFunction|)
         (cons "breaks after"
          (append
           (|bright| x)
           (cons ":= " (cons (|shortenForPrinting| |val|) nil)))))))
       (t nil))))
     |val|))))
```

38.1.53 defun letPrint2

⟨*defun letPrint2*⟩≡

```
(defun |letPrint2| (x |printform| |currentFunction|)
 (prog (|$BreakMode| |flag| y)
  (declare (special |$BreakMode| |$letAssoc| ))
  (return
   (progn
    (setq |$BreakMode| nil)
    (cond
     ((and |$letAssoc|
           (or (setq y (lassoc |currentFunction| |$letAssoc|))
               (setq y (lassoc '|all| |$letAssoc|))))
      (cond
       ((and
          (or (eq y '|all|) (memq x y))
          (null (or (is_genvar x) (|isSharpVarWithNum| x) (gensymp x))))
        (setq |$BreakMode| '|letPrint2|)
        (setq |flag| nil)
        (catch '|letPrint2|
         (|mathprint| (cons '= (cons x (cons |printform| nil)))) |flag|)
        (cond
         ((eq |flag| '|letPrint2|) (|print| |printform|))
         (t nil))))
      (cond
       ((and
         (setq y (|hasPair| 'break y))
         (or (eq y '|all|)
             (and
              (memq x y)
              (null (memq (elt (pname x) 0) '($ |#|)))
              (null (gensymp x)))))
        (|break|
         (append
          (|bright| |currentFunction|)
          (cons "breaks after"
           (append (|bright| x) (cons '|:= | (cons |printform| nil)))))))
       (t nil))))
     x))))
```

38.1.54 defun letPrint3

This is the version for use when we have our hands on a function to convert the data into type "Expression"

⟨*defun letPrint3*⟩≡

```
(defun |letPrint3| (x |xval| |printfn| |currentFunction|)
 (prog (|$BreakMode| |flag| y)
  (declare (special |$BreakMode| |$letAssoc| ))
  (return
   (progn
    (setq |$BreakMode| nil)
    (cond
     ((and |$letAssoc|
       (or (setq y (lassoc |currentFunction| |$letAssoc|))
           (setq y (lassoc '|all| |$letAssoc|))))
      (cond
       ((and
          (or (eq y '|all|) (memq x y))
          (null (or (is_genvar x) (|isSharpVarWithNum| x) (gensymp x))))
        (setq |$BreakMode| '|letPrint2|)
        (setq |flag| nil)
        (catch '|letPrint2|
         (|mathprint|
          (cons '= (cons x (cons (spadcall |xval| |printfn|) nil))))
         |flag|)
        (cond
         ((eq |flag| '|letPrint2|) (|print| |xval|))
         (t nil))))
      (cond
       ((and
         (setq y (|hasPair| 'break y))
         (or
          (eq y '|all|)
          (and
           (memq x y)
           (null (memq (elt (pname x) 0) '($ |#|)))
           (null (gensymp x)))))
        (|break|
         (append
          (|bright| |currentFunction|)
          (cons "breaks after"
           (append (|bright| x) (cons ":= " (cons |xval| nil)))))))
       (t nil))))
    x))))
```

38.1.55 defun getAliasIfTracedMapParameter

⟨*defun getAliasIfTracedMapParameter*⟩≡
```
(defun |getAliasIfTracedMapParameter| (x |currentFunction|)
 (prog (|aliasList|)
 (declare (special |$InteractiveFrame|))
  (return
   (seq
    (cond
     ((|isSharpVarWithNum| x)
       (cond
        ((setq |aliasList|
           (|get| |currentFunction| 'alias |$InteractiveFrame|))
         (exit
          (elt |aliasList|
           (spaddifference
            (string2pint-n (substring (pname x) 1 nil) 1) 1))))))
     (t x))))))
```

38.1.56 defun getBpiNameIfTracedMap

⟨*defun getBpiNameIfTracedMap*⟩≡
```
(defun |getBpiNameIfTracedMap| (name)
 (prog (lmm |bpiName|)
 (declare (special |$InteractiveFrame| /tracenames))
  (return
   (seq
    (cond
     ((setq lmm (|get| name '|localModemap| |$InteractiveFrame|))
       (cond
        ((memq (setq |bpiName| (cadar lmm)) /tracenames)
           (exit |bpiName|))))
     (t name))))))
```

38.1.57 defun hasPair

⟨*defun hasPair*⟩≡
```
(defun |hasPair| (key arg)
 (prog (tmp1 a)
  (return
   (cond
    ((atom arg) nil)
    ((and (pairp arg)
          (progn
           (setq tmp1 (qcar arg))
           (and (pairp tmp1)
                (equal (qcar tmp1) key)
                (progn (setq a (qcdr tmp1)) t))))
       a)
    (t (|hasPair| key (cdr arg)))))))
```

38.1.58 defun shortenForPrinting

⟨*defun shortenForPrinting*⟩≡
```
(defun |shortenForPrinting| (|val|)
 (if (|isDomainOrPackage| |val|)
  (|devaluate| |val|)
  |val|))
```

38.1.59 defun spadTraceAlias

⟨*defun spadTraceAlias*⟩≡
```
(defun |spadTraceAlias| (domainid op n)
 (internl domainid (intern "." "boot") op '|,| (stringimage n)))
```

38.1.60 defun getOption

⟨*defun getOption*⟩≡
```
(defun |getOption| (opt l)
 (let (y)
  (when (setq y (|assoc| opt l))  (cdr y))))
```

38.1.61 defun reportSpadTrace

⟨*defun reportSpadTrace*⟩≡

```
(defun |reportSpadTrace| (|header| t0)
 (prog (op sig n |t| |msg| |namePart| y |tracePart|)
 (declare (special |$traceNoisely|))
  (return
   (progn
    (setq op (car t0))
    (setq sig (cadr t0))
    (setq n (caddr t0))
    (setq |t| (cdddr t0))
    (cond
     ((null |$traceNoisely|) nil)
     (t
      (setq |msg|
       (cons |header|
        (cons '|%b|
         (cons op
          (cons '|:|
           (cons '|%d|
            (cons (CDR sig)
             (cons '| -> |
              (cons (car sig)
               (cons '| in slot |
                (cons n nil)))))))))))
      (setq |namePart| nil)
      (setq |tracePart|
       (cond
        ((and (pairp |t|) (progn (setq y (qcar |t|)) t) (null (null y)))
          (cond
           ((eq y '|all|)
             (cons '|%b| (cons '|all| (cons '|%d| (cons '|vars| nil)))))
           (t (cons '| vars: | (cons y nil)))))
        (t nil)))
      (|sayBrightly| (append |msg| (append |namePart| |tracePart|)))))))))
```

38.1.62 defun orderBySlotNumber

⟨*defun orderBySlotNumber*⟩≡

```
(defun |orderBySlotNumber| (arg)
 (prog (n)
  (return
   (seq
    (assocright
     (|orderList|
      (prog (t0)
       (setq t0 nil)
       (return
        (do ((t1 arg (cdr t1)) (x nil))
            ((or (atom t1)
                 (progn (setq x (car t1)) nil)
                 (progn (progn (setq n (caddr x)) x) nil))
               (nreverse0 t0))
         (seq
          (exit
           (setq t0 (cons (cons n x) t0)))))))))))))
```

38.1.63 defun /tracereply

⟨*defun /tracereply*⟩≡
```
(defun /tracereply ()
 (prog (|d| domainlist |functionList|)
 (declare (special /tracenames))
  (return
   (seq
    (cond
     ((null /tracenames) "   Nothing is traced.")
     (t
       (do ((t0 /tracenames (cdr t0)) (x nil))
           ((or (atom t0) (progn (setq x (car t0)) nil)) nil)
        (seq
         (exit
          (cond
           ((and (pairp x)
                 (progn (setq |d| (qcar x)) t)
                 (|isDomainOrPackage| |d|))
              (setq domainlist (cons (|devaluate| |d|) domainlist)))
           (t
            (setq |functionList| (cons x |functionList|)))))))
       (append |functionList|
        (append domainlist (cons '|traced| nil)))))))))
```

38.1.64 defun spadReply

⟨*defun spadReply,printName*⟩≡
```
(defun |spadReply,printName| (x)
 (prog (|d|)
  (return
   (seq
    (if (and (and (pairp x) (progn (setq |d| (qcar x)) t))
             (|isDomainOrPackage| |d|))
       (exit (|devaluate| |d|)))
     (exit x)))))
```

⟨*defun spadReply*⟩≡

```
(defun |spadReply| ()
 (prog ()
 (declare (special /tracenames))
  (return
   (seq
    (prog (t0)
     (setq t0 nil)
     (return
       (do ((t1 /tracenames (cdr t1)) (x nil))
           ((or (atom t1) (progn (setq x (car t1)) nil)) (nreverse0 t0))
        (seq
         (exit
          (setq t0 (cons (|spadReply,printName| x) t0)))))))))))
```

38.1.65 defun spadUntrace

⟨*defun spadUntrace*⟩≡

```
(defun |spadUntrace| (|domain| options)
 (prog (anyiftrue listofoperations domainid |pair| sigslotnumberalist
        op sig n |lv| |bpiPointer| tracename alias |assocPair|
        |newSigSlotNumberAlist|)
 (declare (special |$letAssoc| /tracenames))
  (return
   (seq
    (cond
     ((null (|isDomainOrPackage| |domain|))
      (|userError| "bad argument to untrace"))
     (t
      (setq anyiftrue (null options))
      (setq listofoperations (|getOption| '|ops:| options))
      (setq domainid (|devaluate| |domain|))
      (cond
       ((null (setq |pair| (|assoc| |domain| /tracenames)))
        (|sayMSG|
         (cons "   No functions in"
          (append
           (|bright| (|prefix2String| domainid))
           (cons "are now traced." nil)))))
       (t
        (setq sigslotnumberalist (cdr |pair|))
        (do ((t0 sigslotnumberalist (cdr t0)) (|pair| nil))
            ((or (atom t0)
                 (progn (setq |pair| (car t0)) nil)
                 (progn
                  (progn
                   (setq op (car |pair|))
                   (setq sig (cadr |pair|))
                   (setq n (caddr |pair|))
                   (setq |lv| (cadddr |pair|))
                   (setq |bpiPointer| (car (cddddr |pair|)))
                   (setq tracename (cadr (cddddr |pair|)))
                   (setq alias (caddr (cddddr |pair|)))
                   |pair|)
                  nil))
                 nil)
          (seq
           (exit
            (cond
             ((or anyiftrue (memq op listofoperations))
              (progn
```

```
                (bpiuntrace tracename alias)
                (rplac (car (elt |domain| n)) |bpiPointer|)
                (rplac (cdddr |pair|) nil)
                (cond
                 ((setq |assocPair|
                     (|assoc| (bpiname |bpiPointer|) |$letAssoc|))
                   (setq |$letAssoc| (remover |$letAssoc| |assocPair|))
                   (cond
                    ((null |$letAssoc|) (setletprintflag nil))
                    (t nil)))
                 (t nil))))))))
          (setq |newSigSlotNumberAlist|
           (prog (t1)
            (setq t1 nil)
            (return
             (do ((t2 sigslotnumberalist (cdr t2)) (x nil))
                 ((or (atom t2) (progn (setq x (car t2)) nil)) (nreverse0 t1))
               (seq
                (exit
                 (cond ((cdddr x) (setq t1 (cons x t1)))))))))))
          (cond
           (|newSigSlotNumberAlist|
            (rplac (cdr |pair|) |newSigSlotNumberAlist|))
           (t
            (setq /tracenames (delasc |domain| /tracenames))
            (|spadReply|))))))))))))
```

38.1.66 defun prTraceNames,fn

⟨*defun prTraceNames,fn*⟩≡

```
(defun |prTraceNames,fn| (x)
 (prog (|d| |t|)
  (return
   (seq
    (if (and (and (pairp x)
                  (progn (setq |d| (qcar x)) (setq |t| (qcdr x)) t))
             (|isDomainOrPackage| |d|))
      (exit (cons (|devaluate| |d|) |t|)))
    (exit x)))))
```

38.1.67 defun prTraceNames

⟨*defun prTraceNames*⟩≡

```
(defun |prTraceNames| ()
 (declare (special /tracenames))
 (seq
  (progn
   (do ((t0 /tracenames (cdr t0)) (x nil))
       ((or (atom t0) (progn (setq x (car t0)) nil)) nil)
    (seq
     (exit
      (print (|prTraceNames,fn| x))))) nil)))
```

38.1.68 defun traceReply

⟨*defun traceReply*⟩≡

```
(defun |traceReply| ()
 (prog (|$domains| |$packages| |$constructors| |d| |functionList|
        |displayList|)
  (declare (special |$domains| |$packages| |$constructors| /tracenames
     $linelength))
  (return
   (seq
    (progn
     (setq |$domains| nil)
     (setq |$packages| nil)
     (setq |$constructors| nil)
     (cond
      ((null /tracenames) (|sayMessage| "   Nothing is traced now."))
      (t
        (|sayBrightly| " ")
        (do ((t0 /tracenames (cdr t0)) (x nil))
            ((or (atom t0) (progn (setq x (car t0)) nil)) nil)
         (seq
          (exit
           (cond
            ((and (pairp x)
                  (progn (setq |d| (qcar x)) t) (|isDomainOrPackage| |d|))
              (|addTraceItem| |d|))
            ((atom x)
               (cond
                ((|isFunctor| x) (|addTraceItem| x))
                ((is_genvar x) (|addTraceItem| (EVAL x)))
                (t (setq |functionList| (cons x |functionList|)))))
            (t (|userError| "bad argument to trace"))))))
        (setq |functionList|
          (prog (t1)
           (setq t1 nil)
           (return
            (do ((t2 |functionList| (cdr t2)) (x nil))
                ((or (atom t2) (progn (setq x (car t2)) nil)) t1)
              (seq
               (exit
                (cond
                 ((null (|isSubForRedundantMapName| x))
                   (setq t1
                    (append t1
                     (cons (|rassocSub| x |$mapSubNameAlist|)
                           (cons " " nil))))))))))))))
```

```
(cond
 (|functionList|
  (cond
   ((eql 2 (|#| |functionList|))
      (|sayMSG| (cons '|   Function traced: | |functionList|)))
   ((<= (PLUS 22 (|sayBrightlyLength| |functionList|)) $linelength)
      (|sayMSG| (cons '|   Functions traced: | |functionList|)))
   (t
      (|sayBrightly| "   Functions traced:")
      (|sayBrightly|
       (|flowSegmentedMsg| |functionList| $linelength 6))))))
(cond
 (|$domains|
  (setq |displayList|
   (|concat|
    (|prefix2String| (CAR |$domains|))
    (prog (t3)
     (setq t3 nil)
     (return
      (do ((t4 (cdr |$domains|) (cdr t4)) (x nil))
          ((or (atom t4) (progn (setq x (car t4)) nil)) t3)
       (seq
        (exit
         (setq t3
          (append t3 (|concat| "," " " (|prefix2String| x))))))))))))
  (cond
   ((atom |displayList|)
    (setq |displayList| (cons |displayList| nil))))
  (|sayBrightly| "   Domains traced: ")
  (|sayBrightly| (|flowSegmentedMsg| |displayList| $linelength 6))))
(cond
 (|$packages|
   (setq |displayList|
    (|concat|
     (|prefix2String| (CAR |$packages|))
     (prog (t5)
      (setq t5 nil)
      (return
       (do ((t6 (cdr |$packages|) (cdr t6)) (x nil))
           ((or (atom t6) (progn (setq x (car t6)) nil)) t5)
        (seq
         (exit
          (setq t5
           (append t5 (|concat| '|, | (|prefix2String| x))))))))))))
   (cond ((atom |displayList|)
       (setq |displayList| (cons |displayList| nil))))
```

```
      (|sayBrightly| "   Packages traced: ")
      (|sayBrightly| (|flowSegmentedMsg| |displayList| $linelength 6))))
    (cond
     (|$constructors|
      (setq |displayList|
       (|concat|
        (|abbreviate| (CAR |$constructors|))
        (prog (t7)
         (setq t7 nil)
         (return
          (do ((t8 (cdr |$constructors|) (cdr t8)) (x nil))
              ((or (atom t8) (progn (setq x (car t8)) nil)) t7)
           (seq
            (exit
             (setq t7
             (append t7 (|concat| '|, | (|abbreviate| x))))))))))))
      (cond ((atom |displayList|)
         (setq |displayList| (cons |displayList| nil))))
      (|sayBrightly| "   Parameterized constructors traced:")
      (|sayBrightly| (|flowSegmentedMsg| |displayList| $linelength 6)))
     (t nil)))))))))
```

38.1.69 defun addTraceItem

⟨*defun addTraceItem*⟩≡

```
(defun |addTraceItem| (|d|)
 (declare (special |$constructors| |$domains| |$packages|))
 (cond
  ((|constructor?| |d|)
    (setq |$constructors| (cons |d| |$constructors|)))
  ((|isDomain| |d|)
    (setq |$domains| (cons (|devaluate| |d|) |$domains|)))
  ((|isDomainOrPackage| |d|)
    (setq |$packages| (cons (|devaluate| |d|) |$packages|)))))
```

38.1.70 defun ?t

⟨*defun ?t*⟩≡
```
(defun |?t| ()
 (let (llm d suffix l)
  (declare (special /tracenames |$InteractiveFrame| |$mapSubNameAlist|))
  (if (null /tracenames)
   (|sayMSG| (|bright| "nothing is traced"))
   (progn
    (dolist (x /tracenames)
     (cond
      ((and (atom x) (null (is_genvar x)))
       (progn
        (cond
         ((setq llm (|get| x '|localModemap| |$InteractiveFrame|))
           (setq x (list (cadar llm)))))
        (|sayMSG|
         `("Function" ,@(|bright| (|rassocSub| x |$mapSubNameAlist|))
           "traced"))))))
    (dolist (x /tracenames)
     (cond
      ((and (pairp x)
            (progn (setq d (qcar x)) (setq l (qcdr x)) t)
            (|isDomainOrPackage| d))
       (progn
        (setq suffix (cond ((|isDomain| d) "domain") (t "package")))
        (|sayBrightly|
         `("   Functions traced in " ,suffix |%b| ,(|devaluate| d) |%d| ":"))
        (dolist (x (|orderBySlotNumber| l))
          (|reportSpadTrace| '|   | (TAKE 4 x)))
        (terpri)))))))))
```

38.1.71 defun tracelet

⟨*defun tracelet*⟩≡

```
(defun |tracelet| (fn |vars|)
 (prog ($traceletflag |$QuickLet| l)
  (declare (special $traceletflag |$QuickLet| |$letAssoc|
     |$traceletFunctions|))
  (return
   (progn
    (cond
     ((and (gensymp fn) (|stupidIsSpadFunction| (eval fn)))
       (setq fn (eval fn))
       (cond
        ((compiled-function-p fn) (setq fn (bpiname fn)))
        (t nil))))
     (cond
      ((eq fn '|Undef|) nil)
      (t
        (setq |vars|
          (cond
           ((eq |vars| '|all|) '|all|)
           ((setq l (lassoc fn |$letAssoc|)) (|union| |vars| l))
           (t |vars|)))
        (setq |$letAssoc| (cons (cons fn |vars|) |$letAssoc|))
        (cond (|$letAssoc| (setletprintflag t)))
        (setq $traceletflag t)
        (setq |$QuickLet| nil)
        (cond
         ((and (null (memq fn |$traceletFunctions|))
               (null (is_genvar fn))
               (compiled-function-p (symbol-function fn))
               (null (|stupidIsSpadFunction| fn))
               (null (gensymp fn)))
           (progn
            (setq |$traceletFunctions| (cons fn |$traceletFunctions|))
            (|compileBoot| fn)
            (setq |$traceletFunctions|
              (|delete| fn |$traceletFunctions|)))))))))))
```

38.1.72 defun breaklet

⟨*defun breaklet*⟩≡

```
(defun |breaklet| (fn |vars|)
 (prog (|$QuickLet| |fnEntry| |pair|)
  (declare (special |$QuickLet| |$letAssoc| |$traceletFunctions|))
  (return
   (progn
    (cond
     ((and (gensymp fn) (|stupidIsSpadFunction| (eval fn)))
       (setq fn (eval fn))
       (cond
        ((compiled-function-p fn) (setq fn (bpiname fn)))
        (t nil))))
     (cond
      ((eq fn '|Undef|) nil)
      (t
       (setq |fnEntry| (lassoc fn |$letAssoc|))
       (setq |vars|
        (cond
         ((setq |pair| (|assoc| 'break |fnEntry|))
           (|union| |vars| (cdr |pair|)))
         (t |vars|)))
       (setq |$letAssoc|
        (cond
         ((null |fnEntry|)
           (cons (cons fn (list (cons 'break |vars|))) |$letAssoc|))
         (|pair| (rplacd |pair| |vars|) |$letAssoc|)))
       (cond (|$letAssoc| (setletprintflag t)))
       (setq |$QuickLet| nil)
       (cond
        ((and (null (memq fn |$traceletFunctions|))
              (null (|stupidIsSpadFunction| fn))
              (null (gensymp fn)))
          (progn
           (setq |$traceletFunctions| (cons fn |$traceletFunctions|))
           (|compileBoot| fn)
           (setq |$traceletFunctions|
            (|delete| fn |$traceletFunctions|)))))))))))
```

38.1.73 defun stupidIsSpadFunction

⟨*defun stupidIsSpadFunction*⟩≡
```
(defun |stupidIsSpadFunction| (fn)
 (strpos ";" (pname fn) 0 nil))
```

38.1.74 defun break

⟨*defun break*⟩≡
```
(defun |break| (msg)
 (prog (condition)
  (declare (special /breakcondition))
  (return
   (progn
    (setq condition (|MONITOR,EVALTRAN| /breakcondition nil))
    (enable-backtrace nil)
    (when (eval condition)
      (|sayBrightly| msg)
      (interrupt))))))
```

38.1.75 defun compileBoot

⟨*defun compileBoot*⟩≡
```
(defun |compileBoot| (fn)
 (|/D,1| (list fn) '(/comp) nil nil))
```

Chapter 39

)undo Command

39.1 undo man page

⟨*undo.help*⟩≡

```
==========================================================================
A.27.  )undo
==========================================================================

User Level Required:  interpreter

Command Syntax:

  - )undo
  - )undo integer
  - )undo integer [option]
  - )undo )redo

where option is one of

  - )after
  - )before

Command Description:

This command is used to restore the state of the user environment to an
earlier point in the interactive session. The argument of an )undo is an
integer which must designate some step number in the interactive session.

)undo n
)undo n )after
```

These commands return the state of the interactive environment to that immediately after step n. If n is a positive number, then n refers to step nummber n. If n is a negative number, it refers to the nth previous command (that is, undoes the effects of the last -n commands).

A)clear all resets the)undo facility. Otherwise, an)undo undoes the effect of)clear with options properties, value, and mode, and that of a previous undo. If any such system commands are given between steps n and n + 1 (n > 0), their effect is undone for)undo m for any 0 < m <= n .

The command)undo is equivalent to)undo -1 (it undoes the effect of the previous user expression). The command)undo 0 undoes any of the above system commands issued since the last user expression.

)undo n)before

This command returns the state of the interactive environment to that immediately before step n. Any)undo or)clear system commands given before step n will not be undone.

)undo)redo

This command reads the file redo.input. created by the last)undo command. This file consists of all user input lines, excluding those backtracked over due to a previous)undo.

The command)history)write will eliminate the ‘‘undone’’ command lines of your program.

Also See:
o)history

[1]

39.2 Data Structures

`$frameRecord = [delta1, delta2,... ]` where delta(i) contains changes in the "backwards" direction. Each delta(i) has the form `((var . proplist)...)` where proplist denotes an ordinary proplist. For example, an entry of the form `((x (value) (mode (Integer)))...)` indicates that to undo 1 step, x's value is cleared and its mode should be set to (Integer).

A delta(i) of the form (systemCommand . delta) is a special delta indicating changes due to system commands executed between the last command and the current command. By recording these deltas separately, it is possible to undo to either BEFORE or AFTER the command. These special delta(i)s are given ONLY when a a system command is given which alters the environment.

Note: recordFrame('system) is called before a command is executed, and record-Frame('normal) is called after (see processInteractive1). If no changes are found for former, no special entry is given.

The `$previousBindings` is a copy of the `CAAR $InteractiveFrame`. This is used to compute the delta(i)s stored in `$frameRecord`.

39.3 Functions

39.3.1 Initial Undo Variables

```
$undoFlag := true     --Default setting for undo is "on"
$frameRecord  := nil  --Initial setting for frame record
$previousBindings := nil
```

⟨*initvars*⟩+≡

```
(defvar |$undoFlag| t "t means we record undo information")
(defvar |$frameRecord| nil "a list of value changes")
(defvar |$previousBindings| nil "a copy of Interactive Frame info for undo")
(defvar |$reportUndo| nil "t means we report the steps undo takes")
```

[1] "history" (22.4.7 p 183)

39.3.2 defun undo

⟨*defun undo*⟩≡

```
(defun |undo| (l)
 (let (tmp1 key s undoWhen n)
 (declare (special |$options| |$InteractiveFrame|))
  (setq undoWhen '|after|)
  (when
    (and (pairp |$options|)
         (eq (qcdr |$options|) nil)
         (progn
          (setq tmp1 (qcar |$options|))
          (and (pairp tmp1)
               (eq (qcdr tmp1) nil)
               (progn (setq key (qcar tmp1)) t)))
     (cond
      ((|stringPrefix?| (setq s (pname key)) "redo")
        (setq |$options| nil)
        (|read| '(|redo.input|)))
      ((null (|stringPrefix?| s "before"))
        (|userError| "only option to undo is \")redo\""))
      (t
        (setq undoWhen '|before|)))))
 (if (null l)
   (setq n (spaddifference 1))
   (setq n (car l)))
 (when (identp n)
   (setq n (parse-integer (pname n)))
   (unless (fixp n)
     (|userError| "undo argument must be an integer")))
 (setq |$InteractiveFrame| (|undoSteps| (|undoCount| n) undoWhen))
 nil))
```

39.3.3 defun recordFrame

⟨*defun recordFrame*⟩≡

```
(defun |recordFrame| (systemNormal)
 (prog (currentAlist delta)
 (declare (special |$undoFlag| |$frameRecord| |$InteractiveFrame|
    |$previousBindings|))
  (return
   (seq
    (cond
     ((null |$undoFlag|) nil)
     (t
       (setq currentAlist (kar |$frameRecord|))
       (setq delta
         (|diffAlist| (caar |$InteractiveFrame|) |$previousBindings|))
       (cond
        ((eq systemNormal '|system|)
          (cond
           ((null delta)
             (return nil))
           (t
             (setq delta (cons '|systemCommand| delta))))))
       (setq |$frameRecord| (cons delta |$frameRecord|))
       (setq |$previousBindings|
        (prog (tmp0)
         (setq tmp0 nil)
         (return
          (do ((tmp1 (caar |$InteractiveFrame|) (cdr tmp1)) (x nil))
              ((or (atom tmp1)
                   (progn (setq x (car tmp1)) nil))
                (nreverse0 tmp0))
           (seq
            (exit
             (setq tmp0
              (cons
               (cons
                (car x)
                (prog (tmp2)
                 (setq tmp2 nil)
                 (return
                  (do ((tmp3 (cdr x) (cdr tmp3)) (y nil))
                      ((or (atom tmp3)
                           (progn (setq y (car tmp3)) nil))
                        (nreverse0 tmp2))
                    (seq
                     (exit
```

```
                    (setq tmp2 (cons (cons (car y) (cdr y)) tmp2))))))))
               tmp0))))))))
      (car |$frameRecord|))))))))
```

39.3.4 defun diffAlist

```
diffAlist(new,old) ==
--record only those properties which are different
  for (pair := [name,:proplist]) in new repeat
    -- name has an entry both in new and old world
    -- (1) if the old world had no proplist for that variable, then
    --     record NIL as the value of each new property
    -- (2) if the old world does have a proplist for that variable, then
    --     a) for each property with a value: give the old value
    --     b) for each property missing:      give NIL as the old value
    oldPair := ASSQ(name,old) =>
      null (oldProplist := CDR oldPair) =>
      --record old values of new properties as NIL
        acc := {\tt{}name,:[[prop]\ for\ [prop,:.]\ in\ proplist},:acc]
      deltas := nil
      for (propval := [prop,:val]) in proplist repeat
        null (oldPropval := ASSOC(prop,oldProplist)) => --missing property
          deltas := {\tt{}prop],:deltas]\nwnewline
\ \ \ \ \ \ \ \ EQ(CDR\ oldPropval,val)\ =>\ 'skip\nwnewline
\ \ \ \ \ \ \ \ deltas\ :=\ [oldPropval,:deltas]\nwnewline
\ \ \ \ \ \ deltas\ =>\ acc\ :=\ [[name,:NREVERSE\ deltas],:acc]\nwnewline
\ \ \ \ acc\ :=\ [[name,:[[prop]\ for\ [prop,:.]\ in\ proplist},:acc]
--record properties absent on new list (say, from a )cl all)
  for (oldPair := [name,:r]) in old repeat
    r and null LASSQ(name,new) =>
      acc := [oldPair,:acc]
    -- name has an entry both in new and old world
    -- (1) if the new world has no proplist for that variable
    --     (a) if the old world does, record the old proplist
    --     (b) if the old world does not, record nothing
    -- (2) if the new world has a proplist for that variable, it has
    --     been handled by the first loop.
  res := NREVERSE acc
  if BOUNDP '$reportUndo and $reportUndo then reportUndo res
  res
```

⟨*defun diffAlist*⟩≡

```
(defun |diffAlist| (new old)
 (prog (proplist oldPair oldProplist val oldPropval deltas prop name r acc res)
  (return
   (seq
    (progn
     (do ((tmp0 new (cdr tmp0)) (pair nil))
         ((or (atom tmp0)
              (progn (setq pair (car tmp0)) nil)
              (progn
               (progn
                (setq name (car pair))
```

```
            (setq proplist (cdr pair))
            pair)
           nil))
         nil)
(seq
 (exit
  (cond
   ((setq oldPair (assq name old))
    (cond
     ((null (setq oldProplist (cdr oldPair)))
       (setq acc
        (cons
         (cons
           name
           (prog (tmp1)
            (setq tmp1 nil)
            (return
             (do ((tmp2 proplist (cdr tmp2)) (tmp3 nil))
                 ((or (atom tmp2)
                      (progn (setq tmp3 (car tmp2)) nil)
                      (progn
                       (progn (setq prop (car tmp3)) tmp3)
                       nil))
                    (nreverse0 tmp1))
              (seq
               (exit
                (setq tmp1 (cons (cons prop nil) tmp1))))))))
         acc)))
     (t
       (setq deltas nil)
       (do ((tmp4 proplist (cdr tmp4)) (|propval| nil))
           ((or (atom tmp4)
                (progn (setq |propval| (car tmp4)) nil)
                (progn
                 (progn
                  (setq prop (car |propval|))
                  (setq val (cdr |propval|))
                  |propval|)
                 nil))
             nil)
        (seq
         (exit
          (cond
           ((null (setq oldPropval (|assoc| prop oldProplist)))
             (setq deltas (cons (cons prop nil) deltas)))
           ((eq (cdr oldPropval) val) '|skip|)
```

```
                (t (setq deltas (cons oldPropval deltas)))))))
            (when deltas
              (setq acc
               (cons (cons name (nreverse deltas)) acc))))))
       (t
         (setq acc
          (cons
           (cons
            name
            (prog (tmp5)
             (setq tmp5 nil)
             (return
              (do ((tmp6 proplist (cdr tmp6)) (tmp7 nil))
                  ((or (atom tmp6)
                       (progn (setq tmp7 (CAR tmp6)) nil)
                       (progn
                        (progn (setq prop (CAR tmp7)) tmp7)
                        nil))
                   (nreverse0 tmp5))
                (seq
                 (exit
                  (setq tmp5 (cons (cons prop nil) tmp5)))))))
           acc)))))))
(seq
 (do ((tmp8 old (cdr tmp8)) (oldPair nil))
     ((or (atom tmp8)
          (progn (setq oldPair (car tmp8)) nil)
          (progn
           (progn
            (setq name (car oldPair))
            (setq r (cdr oldPair))
            oldPair)
           nil))
         nil)
  (seq
   (exit
    (cond
     ((and r (null (lassq name new)))
       (exit
        (setq acc (cons oldPair acc))))))))
 (setq res (nreverse acc))
 (cond
  ((and (boundp '|$reportUndo|) |$reportUndo|)
    (|reportUndo| res)))
 (exit res)))))))
```

39.3.5 defun reportUndo

This function is enabled by setting `$reportUndo` to a non-nil value. An example of the output generated is:

```
r := binary(22/7)



          ---
   (1)  11.001
                                                        Type: BinaryExpansion
Properties of % ::
  value was: NIL
  value is:  ((|BinaryExpansion|) WRAPPED . #(1 (1 1) NIL (0 0 1)))
Properties of r ::
  value was: NIL
  value is:  ((|BinaryExpansion|) WRAPPED . #(1 (1 1) NIL (0 0 1)))
```

⟨*defun reportUndo*⟩≡

```
(defun |reportUndo| (acc)
 (prog (name proplist curproplist prop value)
 (declare (special |$InteractiveFrame|))
  (return
   (seq
    (do ((tmp0 acc (cdr tmp0)) (tmp1 nil))
        ((or (atom tmp0)
             (progn (setq tmp1 (car tmp0)) nil)
             (progn
              (progn
               (setq name (car tmp1))
               (setq proplist (cdr tmp1))
               tmp1)
              nil))
            nil)
     (seq
      (exit
       (progn
        (|sayBrightly|
          (strconc '|Properties of | (pname name) " ::"))
        (setq curproplist (lassoc name (caar |$InteractiveFrame|)))
        (do ((tmp2 proplist (cdr tmp2)) (tmp3 nil))
            ((or (atom tmp2)
                 (progn (setq tmp3 (car tmp2)) nil)
                 (progn
                  (progn
                   (setq prop (car tmp3))
                   (setq value (cdr tmp3))
```

```
                tmp3)
               nil))
             nil)
        (seq
         (exit
          (progn
           (|sayBrightlyNT|
            (cons "  " (cons prop (cons " was: " nil))))
           (|pp| value)
           (|sayBrightlyNT|
            (cons "  " (cons prop (cons " is:  " nil))))
           (|pp| (lassoc prop curproplist)))))))))))))))
```

39.3.6 defun clearFrame

⟨*defun clearFrame*⟩≡

```
(defun |clearFrame| ()
 (declare (special |$frameRecord| |$previousBindings|))
  (|clearCmdAll|)
  (setq |$frameRecord| nil)
  (setq |$previousBindings| nil))
```

39.3.7 Undo previous n commands

⟨*defun undoCount*⟩≡

```
(defun |undoCount| (n)
 "Undo previous n commands"
 (prog (m)
 (declare (special |$IOindex|))
  (return
   (progn
    (setq m
     (cond
       ((>= n 0) (spaddifference (spaddifference |$IOindex| n) 1))
       (t (spaddifference n))))
    (cond
     ((>= m |$IOindex|)
       (|userError|
        (strconc "Magnitude of undo argument must be less than step number ("
           (stringimage |$IOindex|) ").")))
     (t m))))))
```

39.3.8 defun undoSteps

```
-- undoes m previous commands; if )before option, then undo one extra at end
--Example: if $IOindex now is 6 and m = 2 then general layout of $frameRecord,
--  after the call to recordFrame below will be:
--  (<change for systemcommands>
--  (<change for #5> <change for system commands>
--  (<change for #4> <change for system commands>
--  (<change for #3> <change for system commands>
--   <change for #2> <change for system commands>
--   <change for #1> <change for system commands>) where system
--  command entries are optional and identified by (systemCommand . change).
--  For a ")undo 3 )after", m = 2 and undoStep swill restore the environment
--  up to, but not including <change for #3>.
--  An "undo 3 )before" will additionally restore <change for #3>.
--  Thus, the later requires one extra undo at the end.
```

⟨*defun undoSteps*⟩≡

```
(defun |undoSteps| (m beforeOrAfter)
 (let (tmp1 tmp2 systemDelta lastTailSeen env)
 (declare (special |$IOindex| |$InteractiveFrame| |$frameRecord|))
  (|writeInputLines| '|redo| (spaddifference |$IOindex| m))
  (|recordFrame| '|normal|)
  (setq env (copy (caar |$InteractiveFrame|)))
  (do ((|i| 0 (qsadd1 |i|)) (framelist |$frameRecord| (cdr framelist)))
      ((or (qsgreaterp |i| m) (atom framelist)) nil)
    (setq env (|undoSingleStep| (CAR framelist) env))
    (if (and (pairp framelist)
             (progn
              (setq tmp1 (qcdr framelist))
              (and (pairp tmp1)
                   (progn
                    (setq tmp2 (qcar tmp1))
                    (and (pairp tmp2)
                         (eq (qcar tmp2) '|systemCommand|)
                         (progn
                          (setq systemDelta (qcdr tmp2))
                          t))))))
        (progn
          (setq framelist (cdr framelist))
          (setq env (|undoSingleStep| systemDelta env)))
        (setq lastTailSeen framelist)))
  (cond
   ((eq beforeOrAfter '|before|)
     (setq env (|undoSingleStep| (car (cdr lastTailSeen)) env))))
  (setq |$frameRecord| (cdr |$frameRecord|))
  (setq |$InteractiveFrame| (list (list env)))))
```

39.3.9 defun undoSingleStep

```
undoSingleStep(changes,env) ==
--Each change is a name-proplist pair. For each change:
--  (1) if there exists a proplist in env, then for each prop-value change:
--      (a) if the prop exists in env, RPLAC in the change value
--      (b) otherwise, CONS it onto the front of prop-values for that name
--  (2) add change to the front of env
--  pp '"----Undoing 1 step--------"
--  pp changes
```

⟨*defun undoSingleStep*⟩≡

```
(defun |undoSingleStep| (changes env)
 (prog (name changeList pairlist proplist prop value node)
  (return
   (seq
    (progn
     (do ((tmp0 changes (cdr tmp0)) (|change| nil))
         ((or (atom tmp0)
              (progn (setq |change| (car tmp0)) nil)
              (progn
               (progn
                (setq name (car |change|))
                (setq changeList (cdr |change|))
                |change|)
               nil))
          nil)
      (seq
       (exit
        (progn
         (when (lassoc '|localModemap| changeList)
           (setq changeList (|undoLocalModemapHack| changeList)))
         (cond
          ((setq pairlist (assq name env))
           (cond
            ((setq proplist (cdr pairlist))
             (do ((tmp1 changeList (cdr tmp1)) (pair nil))
                 ((or (atom tmp1)
                      (progn (setq pair (car tmp1)) nil)
                      (progn
                       (progn
                        (setq prop (car pair))
                        (setq value (cdr pair))
                        pair)
                       nil))
                  nil)
              (seq
```

```
                (exit
                 (cond
                  ((setq node (assq prop proplist))
                    (rplacd node value))
                  (t
                    (rplacd proplist
                      (cons (car proplist) (cdr proplist)))
                    (rplaca proplist pair)))))))
           (t (rplacd pairlist changeList))))
       (t
         (setq env (cons |change| env))))))))
env)))))
```

39.3.10 defun undoLocalModemapHack

⟨*defun undoLocalModemapHack*⟩≡

```
(defun |undoLocalModemapHack| (changeList)
 (prog (name value)
  (return
   (seq
    (prog (tmp0)
     (setq tmp0 nil)
     (return
      (do ((tmp1 changeList (cdr tmp1)) (pair nil))
          ((or (atom tmp1)
               (progn (setq pair (car tmp1)) nil)
               (progn
                (progn
                 (setq name (car pair))
                 (setq value (cdr pair))
                 pair)
                nil))
              (nreverse0 tmp0))
       (seq
        (exit
         (cond
          ((cond
             ((eq name '|localModemap|) (cons name nil))
             (t pair))
           (setq tmp0
            (cons
             (cond
              ((eq name '|localModemap|) (cons name nil))
              (t pair)) tmp0)))))))))))))
```

39.3.11 Remove undo lines from history write

Removing undo lines from `)hist )write linelist`

⟨*defun removeUndoLines*⟩≡

```
(defun |removeUndoLines| (u)
 "Remove undo lines from history write"
 (prog (xtra savedIOindex s s1 m s2 x code c n acc)
 (declare (special |$currentLine| |$IOindex|))
  (return
   (seq
    (progn
     (setq xtra
      (cond
       ((stringp |$currentLine|) (cons |$currentLine| nil))
       (t (reverse |$currentLine|))))
     (setq xtra
      (prog (tmp0)
       (setq tmp0 nil)
       (return
        (do ((tmp1 xtra (cdr tmp1)) (x nil))
            ((or (atom tmp1)
                 (progn (setq x (car tmp1)) nil))
               (nreverse0 tmp0))
         (seq
          (exit
           (cond
            ((null (|stringPrefix?| ")history" x))
              (setq tmp0 (cons x tmp0)))))))))))
     (setq u (append u xtra))
     (cond
      ((null
        (prog (tmp2)
         (setq tmp2 nil)
         (return
          (do ((tmp3 nil tmp2) (tmp4 u (cdr tmp4)) (x nil))
              ((or tmp3 (atom tmp4) (progn (setq x (car tmp4)) nil)) tmp2)
           (seq
            (exit
             (setq tmp2
               (or tmp2 (|stringPrefix?| ")undo" x))))))))) u)
      (t
        (setq savedIOindex |$IOindex|)
        (setq |$IOindex| 1)
        (do ((y u (cdr y)))
            ((atom y) nil)
```

```
(seq
 (exit
  (cond
   ((eql (elt (setq x (car y)) 0) #\) )
     (cond
      ((|stringPrefix?| ")undo"
                        (setq s (|trimString| x)))
       (setq s1 (|trimString| (substring s 5 nil)))
       (cond
         ((nequal s1 ")redo")
           (setq m (|charPosition| #\) s1 0))
           (setq code
            (cond
              ((> (maxindex s1) m) (elt s1 (plus m 1)))
              (t #\a)))
           (setq s2 (|trimString| (substring s1 0 m)))))
       (setq n
        (cond
         ((string= s1 ")redo")
            0)
         ((nequal s2 "")
           (|undoCount| (parse-integer s2)))
         (t (spaddifference 1))))
       (rplaca y
         (concat ">" code (stringimage n))))
      (t nil)))
   (t (setq |$IOindex| (plus |$IOindex| 1)))))))
(setq acc nil)
(do ((y (nreverse u) (cdr y)))
    ((atom y) nil)
 (seq
  (exit
   (cond
    ((eql (elt (setq x (car y)) 0) #\>)
      (setq code (elt x 1))
      (setq n (parse-integer (substring x 2 nil)))
      (setq y (cdr y))
      (do ()
          ((null y) nil)
       (seq
        (exit
         (progn
          (setq c (car y))
          (cond
           ((or (eql (elt c 0) #\))
                (eql (elt c 0) #\>))
```

```
                (setq y (cdr y)))
               ((eql n 0)
                (return nil))
               (t
                (setq n (spaddifference n 1))
                (setq y (cdr y))))))))
        (cond
         ((and y (nequal code #\b))
           (setq acc (cons c acc)))))
     (t (setq acc (cons x acc)))))))
(setq |$IOindex| savedIOindex)
acc)))))))
```

Chapter 40

)what Command

40.1 what man page

⟨*what.help*⟩≡

```
==========================================================================
A.28.  )what
==========================================================================

User Level Required:  interpreter

Command Syntax:

  - )what categories pattern1 [pattern2 ...]
  - )what commands   pattern1 [pattern2 ...]
  - )what domains    pattern1 [pattern2 ...]
  - )what operations pattern1 [pattern2 ...]
  - )what packages   pattern1 [pattern2 ...]
  - )what synonym    pattern1 [pattern2 ...]
  - )what things     pattern1 [pattern2 ...]
  - )apropos         pattern1 [pattern2 ...]

Command Description:

This command is used to display lists of things in the system. The patterns
are all strings and, if present, restrict the contents of the lists. Only
those items that contain one or more of the strings as substrings are
displayed. For example,

)what synonym
```

```
displays all command synonyms,

)what synonym ver

displays all command synonyms containing the substring ‘‘ver’’,

)what synonym ver pr

displays all command synonyms containing the substring ‘‘ver’’ or the
substring ‘‘pr’’. Output similar to the following will be displayed

---------------- System Command Synonyms -----------------


user-defined synonyms satisfying patterns:
      ver pr


  )apr ........................... )what things
  )apropos ....................... )what things
  )prompt ........................ )set message prompt
  )version ....................... )lisp *yearweek*

Several other things can be listed with the )what command:

  categories displays a list of category constructors.
  commands  displays a list of  system commands available  at your
    user-level. Your user-level is set via the )set userlevel command. To get
    a description of a particular command, such as ‘‘)what’’, issue )help
    what.
  domains   displays a list of domain constructors.
  operations displays a list of operations in  the system library.
    It is recommended that you qualify this command with one or more
    patterns, as there are thousands of operations available. For example,
    say you are looking for functions that involve computation of
    eigenvalues. To find their names, try )what operations eig. A rather
    large list of operations is loaded into the workspace when this command
    is first issued. This list will be deleted when you clear the workspace
    via )clear all or )clear completely. It will be re-created if it is
    needed again.
  packages  displays a list of package constructors.
  synonym  lists system command synonyms.
  things    displays all  of the  above types for  items containing
    the pattern strings as substrings. The command synonym )apropos is
    equivalent to )what things.
```

```
Also See:
o )display
o )set
o )show
```

[1]

40.1.1 defvar $whatOptions

⟨*initvars*⟩+≡

```
(defvar |$whatOptions| '(|operations| |categories| |domains| |packages|
                         |commands| |synonyms| |things|))
```

40.1.2 defun what

⟨*defun what*⟩≡

```
(defun |what| (l)
 (|whatSpad2Cmd| l))
```

40.1.3 defun whatSpad2Cmd,fixpat

⟨*defun whatSpad2Cmd,fixpat*⟩≡

```
(defun |whatSpad2Cmd,fixpat| (x)
 (prog (|x'|)
  (return
   (seq
    (if (and (pairp x) (progn (setq |x'| (qcar x)) t))
      (exit (downcase |x'|)))
    (exit (downcase x))))))
```

[1] "display" (17.2.1 p 137) "set" (32.40.1 p 372) "show" (33 p 377)

40.1.4 defun whatSpad2Cmd

⟨*defun whatSpad2Cmd*⟩≡

```
(defun |whatSpad2Cmd| (arg)
 (prog (|$e| |key0| key args)
  (declare (special |$e| |$whatOptions|))
  (return
   (seq
    (progn
     (setq |$e| |$EmptyEnvironment|)
     (cond
      ((null arg) (|reportWhatOptions|))
      (t
        (setq |key0| (car arg))
        (setq args (cdr arg))
        (setq key (|selectOptionLC| |key0| |$whatOptions| nil))
        (cond
         ((null key) (|sayKeyedMsg| 's2iz0043 nil))
         (t
          (setq args
           (prog (t0)
            (setq t0 nil)
            (return
              (do ((t1 args (cdr t1)) (p nil))
                  ((or (atom t1)
                       (progn (setq p (car t1)) nil))
                    (nreverse0 t0))
                 (seq
                  (exit
                   (setq t0 (cons (|whatSpad2Cmd,fixpat| p) t0))))))))
          (seq
           (cond
            ((eq key '|things|)
              (do ((t2 |$whatOptions| (cdr t2)) (opt nil))
                  ((or (atom t2) (progn (setq opt (CAR t2)) nil)) nil)
                 (seq
                  (exit
                   (cond
                    ((null (memq opt '(|things|)))
                      (exit (|whatSpad2Cmd| (cons opt args))))))))
            ((eq key '|categories|)
              (|filterAndFormatConstructors| '|category| "Categories" args))
            ((eq key '|commands|) (|whatCommands| args))
            ((eq key '|domains|)
              (|filterAndFormatConstructors| '|domain| "Domains" args))
            ((eq key '|operations|)
```

```
            (|apropos| args))
           ((eq key '|packages|)
            (|filterAndFormatConstructors| '|package| "Packages" args))
           (t
            (cond ((eq key '|synonyms|)
                   (|printSynonyms| args))))))))))))))
```

40.1.5 defun filterAndFormatConstructors

⟨*defun filterAndFormatConstructors*⟩≡

```
(defun |filterAndFormatConstructors| (|constrType| label |patterns|)
 (prog (l)
 (declare (special $linelength ))
  (return
   (progn (|centerAndHighlight| label $linelength (|specialChar| '|hbar|))
    (setq l
     (|filterListOfStringsWithFn| |patterns|
       (|whatConstructors| |constrType|)
       (|function| cdr)))
    (cond (|patterns|
     (cond
      ((null l)
       (|sayMessage|
        (cons "   No "
         (cons label
          (cons " with names matching patterns:"
           (cons '|%l|
            (cons "   "
             (cons '|%b|
              (append (|blankList| |patterns|)
               (cons '|%d| nil)))))))))
      (t
        (|sayMessage|
         (cons label
          (cons " with names matching patterns:"
           (cons '|%l|
            (cons "   "
             (cons '|%b|
              (append (|blankList| |patterns|)
               (cons '|%d| nil))))))))))))
    (cond (l (|pp2Cols| l)))))))
```

40.1.6 defun whatConstructors

⟨*defun whatConstructors*⟩≡

```
(defun |whatConstructors| (|constrType|)
 (prog nil
  (return
   (seq
    (msort
     (prog (t0)
      (setq t0 nil)
      (return
       (do ((t1 (|allConstructors|) (cdr t1)) (|con| nil))
           ((or (atom t1) (progn (setq |con| (car t1)) nil)) (nreverse0 t0))
        (seq
         (exit
          (cond
           ((boot-equal (getdatabase |con| 'constructorkind)
                        |constrType|)
             (setq t0
              (cons
               (cons
                (getdatabase |con| 'abbreviation)
                (string |con|))
               t0)))))))))))))
```

40.1.7 Display all operation names containing the fragment

Argument l is a list of operation name fragments. This displays all operation names containing these fragments

⟨*defun apropos*⟩≡

```
(defun |apropos| (arg)
 "Display all operation names containing the fragment"
 (prog (ops)
  (return
   (seq
    (progn
     (setq ops
      (cond
       ((null arg) (|allOperations|))
       (t
         (|filterListOfStrings|
          (prog (t0)
           (setq t0 nil)
           (return
            (do ((t1 arg (cdr t1)) (p nil))
                ((or (atom t1) (progn (setq p (car t1)) nil))
                  (nreverse0 t0))
             (seq (exit (setq t0 (cons (downcase (stringimage p)) t0)))))))
          (|allOperations|)))))
    (cond
     (ops
      (|sayMessage| "Operations whose names satisfy the above pattern(s):")
      (|sayAsManyPerLineAsPossible| (msort ops))
      (|sayKeyedMsg| 's2if0011 (cons (car ops) nil)))
     (t
      (|sayMessage| "   There are no operations containing those patterns")
      nil)))))))
```

Chapter 41

)with Command

41.1 with man page

⟨*with.help*⟩≡

```
This command is obsolete.
This has been renamed )library.

See also:
o )library
```

[1]

41.1.1 defun with

⟨*defun with*⟩≡

```
(defun |with| (args)
 (|library| args))
```

[1] "library" (24 p 221)

Chapter 42

)workfiles Command

42.1 workfiles man page

42.1.1 defun workfiles

⟨*defun workfiles*⟩≡
```
(defun |workfiles| (l)
 (|workfilesSpad2Cmd| l))
```

42.1.2 defun workfilesSpad2Cmd

⟨*defun workfilesSpad2Cmd*⟩≡

```
(defun |workfilesSpad2Cmd| (args)
 (let (deleteflag type flist type1 fl)
  (declare (special |$options| |$sourceFiles| $linelength))
  (cond
   (args (|throwKeyedMsg| 's2iz0047 nil))
   (t
     (setq deleteflag nil)
     (do ((t0 |$options| (cdr t0)) (t1 nil))
         ((or (atom t0)
              (progn (setq t1 (car t0)) nil)
              (progn (progn (setq type (car t1)) t1) nil))
          nil)
       (setq type1
          (|selectOptionLC| type '(|boot| |lisp| |meta| |delete|) nil))
       (cond
        ((null type1) (|throwKeyedMsg| 's2iz0048 (cons type nil)))
        ((eq type1 '|delete|) (setq deleteflag t))))
     (do ((t2 |$options| (cdr t2)) (t3 nil))
         ((or (atom t2)
          (progn (setq t3 (CAR t2)) nil)
          (progn
           (progn
            (setq type (car t3))
            (setq flist (cdr t3)) t3)
           nil))
          nil)
      (setq type1 (|selectOptionLC| type '(|boot| |lisp| |meta| |delete|) nil))
      (unless (eq type1 '|delete|)
        (dolist (file flist)
         (setq fl (|pathname| (list file type1 "*")))
         (cond
          (deleteflag
           (setq |$sourceFiles| (|delete| fl |$sourceFiles|)))
          ((null (make-input-filename fl))
           (|sayKeyedMsg| 's2iz0035 (list (|namestring| fl))))
          (t (|updateSourceFiles| fl))))))
     (say " ")
     (|centerAndHighlight|
       '| User-specified work files |
       $linelength
       (|specialChar| '|hbar|))
     (say " ")
     (if (null |$sourceFiles|)
```

```
(say "   no files specified")
(progn
  (setq |$sourceFiles| (sortby '|pathnameType| |$sourceFiles|))
  (do ((t5 |$sourceFiles| (cdr t5)) (fl nil))
      ((or (atom t5) (progn (setq fl (car t5)) nil)) nil)
     (|sayBrightly| (list "   " (|namestring| fl)))))))))))
```

Chapter 43

)zsystemdevelopment Command

43.1 zsystemdevelopment man page

43.1.1 defun zsystemdevelopment

⟨*defun zsystemdevelopment*⟩≡
```
(defun |zsystemdevelopment| (arg)
 (|zsystemDevelopmentSpad2Cmd| arg))
```

43.1.2 defun zsystemDevelopmentSpad2Cmd

⟨*defun zsystemDevelopmentSpad2Cmd*⟩≡
```
(defun |zsystemDevelopmentSpad2Cmd| (arg)
 (declare (special |$InteractiveMode|))
 (|zsystemdevelopment1| arg |$InteractiveMode|))
```

43.1.3 defun zsystemdevelopment1

⟨*defun zsystemdevelopment1*⟩≡

```
(defun |zsystemdevelopment1| (arg im)
 (let (|$InteractiveMode| fromopt opt optargs newopt opt1 constream upf fun)
 (declare (special |$InteractiveMode| /wsname /version |$options|))
 (setq |$InteractiveMode| im)
 (setq fromopt nil)
 (do ((t0 |$options| (cdr t0)) (t1 nil))
     ((or (atom t0)
          (progn (setq t1 (car t0)) nil)
          (progn
           (progn
            (setq opt (CAR t1))
            (setq optargs (CDR t1))
            t1)
           nil))
      nil)
  (setq opt1 (|selectOptionLC| opt '(|from|) nil))
  (when (eq opt1 '|from|) (setq fromopt (cons (cons 'from optargs) nil))))
 (do ((t2 |$options| (cdr t2)) (t3 nil))
     ((or (atom t2)
          (progn (setq t3 (car t2)) nil)
          (progn
           (progn
            (setq opt (car t3))
            (setq optargs (cdr t3))
            t3)
           nil))
      nil)
  (unless optargs (setq optargs arg))
  (setq newopt (append optargs fromopt))
  (setq opt1 (|selectOptionLC| opt '(|from|) nil))
  (cond
   ((eq opt1 '|from|)   nil)
   ((eq opt '|c|)       (|/D,1| newopt (/COMP) nil nil))
   ((eq opt '|d|)       (|/D,1| newopt 'define nil nil))
   ((eq opt '|dt|)      (|/D,1| newopt 'define nil t))
   ((eq opt '|ct|)      (|/D,1| newopt (/COMP) nil t))
   ((eq opt '|ctl|)     (|/D,1| newopt (/COMP) nil 'tracelet))
   ((eq opt '|ec|)      (|/D,1| newopt (/COMP) t nil))
   ((eq opt '|ect|)     (|/D,1| newopt (/COMP) t t))
   ((eq opt '|e|)       (|/D,1| newopt nil t nil))
   ((eq opt '|version|) (|version|))
   ((eq opt '|pause|)
     (setq constream
```

```
    (defiostream '((device . console) (qual . v)) 120 0))
  (next constream)
  (shut constream))
((or
    (eq opt '|update|)
    (eq opt '|patch|))
  (setq |$InteractiveMode| nil)
  (setq upf
   (cons
    (or (kar optargs) /version)
    (cons
     (or (kadr optargs) /wsname)
     (cons (or (kaddr optargs) '*) nil))))
  (setq fun
   (cond
    ((eq opt '|patch|) '/update-lib-1)
    (t '/update-1)))
  (catch 'filenam (funcall fun upf))
  (|sayMessage| "   Update/patch is completed."))
((null optargs)
 (|sayBrightly| `("   An argument is required for" ,@(|bright| opt))))
(t
 (|sayMessage|
  `("   Unknown option:" ,@(|bright| opt)
    |%l| "   Available options are"
    ,@(|bright|
       "c ct e ec ect cls pause update patch compare record")))))))
```

Chapter 44

Handling output

44.1 Special Character Tables

44.1.1 defvar $defaultSpecialCharacters

⟨*initvars*⟩+≡

```
(defvar |$defaultSpecialCharacters| (list
 (int-char 28)    ; upper left corner
 (int-char 27)    ; upper right corner
 (int-char 30)    ; lower left corner
 (int-char 31)    ; lower right corner
 (int-char 79)    ; vertical bar
 (int-char 45)    ; horizontal bar
 (int-char 144)   ; APL quad
 (int-char 173)   ; left bracket
 (int-char 189)   ; right bracket
 (int-char 192)   ; left brace
 (int-char 208)   ; right brace
 (int-char 59)    ; top    box tee
 (int-char 62)    ; bottom box tee
 (int-char 63)    ; right  box tee
 (int-char 61)    ; left   box tee
 (int-char 44)    ; center box tee
 (int-char 224))) ; back slash
```

44.1.2 defvar $plainSpecialCharacters0

⟨*initvars*⟩+≡

```
(defvar |$plainSpecialCharacters0| (list
 (int-char 78)     ; upper left corner   (+)
 (int-char 78)     ; upper right corner  (+)
 (int-char 78)     ; lower left corner   (+)
 (int-char 78)     ; lower right corner  (+)
 (int-char 79)     ; vertical bar
 (int-char 96)     ; horizontal bar      (-)
 (int-char 111)    ; APL quad            (?)
 (int-char 173)    ; left bracket
 (int-char 189)    ; right bracket
 (int-char 192)    ; left brace
 (int-char 208)    ; right brace
 (int-char 78)     ; top    box tee      (+)
 (int-char 78)     ; bottom box tee      (+)
 (int-char 78)     ; right  box tee      (+)
 (int-char 78)     ; left   box tee      (+)
 (int-char 78)     ; center box tee      (+)
 (int-char 224)))  ; back slash
```

44.1.3 defvar $plainSpecialCharacters1

⟨*initvars*⟩+≡

```
(defvar |$plainSpecialCharacters1| (list
 (int-char 107)    ; upper left corner   (,)
 (int-char 107)    ; upper right corner  (,)
 (int-char 125)    ; lower left corner   (')
 (int-char 125)    ; lower right corner  (')
 (int-char 79)     ; vertical bar
 (int-char 96)     ; horizontal bar      (-)
 (int-char 111)    ; APL quad            (?)
 (int-char 173)    ; left bracket
 (int-char 189)    ; right bracket
 (int-char 192)    ; left brace
 (int-char 208)    ; right brace
 (int-char 78)     ; top    box tee      (+)
 (int-char 78)     ; bottom box tee      (+)
 (int-char 78)     ; right  box tee      (+)
 (int-char 78)     ; left   box tee      (+)
 (int-char 78)     ; center box tee      (+)
 (int-char 224)))  ; back slash
```

44.1.4 defvar $plainSpecialCharacters2

⟨*initvars*⟩+≡

```
(defvar |$plainSpecialCharacters2| (list
 (int-char 79)     ; upper left corner   (|)
 (int-char 79)     ; upper right corner  (|)
 (int-char 79)     ; lower left corner   (|)
 (int-char 79)     ; lower right corner  (|)
 (int-char 79)     ; vertical bar
 (int-char 96)     ; horizontal bar      (-)
 (int-char 111)    ; APL quad            (?)
 (int-char 173)    ; left bracket
 (int-char 189)    ; right bracket
 (int-char 192)    ; left brace
 (int-char 208)    ; right brace
 (int-char 78)     ; top    box tee      (+)
 (int-char 78)     ; bottom box tee      (+)
 (int-char 78)     ; right  box tee      (+)
 (int-char 78)     ; left   box tee      (+)
 (int-char 78)     ; center box tee      (+)
 (int-char 224)))  ; back slash
```

44.1.5 defvar $plainSpecialCharacters3

⟨*initvars*⟩+≡

```
(defvar |$plainSpecialCharacters3| (list
 (int-char 96)     ; upper left corner   (-)
 (int-char 96)     ; upper right corner  (-)
 (int-char 96)     ; lower left corner   (-)
 (int-char 96)     ; lower right corner  (-)
 (int-char 79)     ; vertical bar
 (int-char 96)     ; horizontal bar      (-)
 (int-char 111)    ; APL quad            (?)
 (int-char 173)    ; left bracket
 (int-char 189)    ; right bracket
 (int-char 192)    ; left brace
 (int-char 208)    ; right brace
 (int-char 78)     ; top    box tee      (+)
 (int-char 78)     ; bottom box tee      (+)
 (int-char 78)     ; right  box tee      (+)
 (int-char 78)     ; left   box tee      (+)
 (int-char 78)     ; center box tee      (+)
 (int-char 224)))  ; back slash
```

44.1.6 defvar $plainRTspecialCharacters

⟨*initvars*⟩+≡

```
(defvar |$plainRTspecialCharacters| (list
 (QUOTE +)       ; upper left corner   (+)
 (QUOTE +)       ; upper right corner  (+)
 (QUOTE +)       ; lower left corner   (+)
 (QUOTE +)       ; lower right corner  (+)
 (QUOTE |\||)    ; vertical bar
 (QUOTE -)       ; horizontal bar      (-)
 (QUOTE ?)       ; APL quad            (?)
 (QUOTE [)       ; left bracket
 (QUOTE ])       ; right bracket
 (QUOTE {)       ; left brace
 (QUOTE })       ; right brace
 (QUOTE +)       ; top    box tee      (+)
 (QUOTE +)       ; bottom box tee      (+)
 (QUOTE +)       ; right  box tee      (+)
 (QUOTE +)       ; left   box tee      (+)
 (QUOTE +)       ; center box tee      (+)
 (QUOTE |\\|)))  ; back slash
```

44.1.7 defvar $RTspecialCharacters

⟨*initvars*⟩+≡

```
(defvar |$RTspecialCharacters| (list
 (intern (string (code-char 218))) ;-- upper left corner   (+)
 (intern (string (code-char 191))) ;-- upper right corner  (+)
 (intern (string (code-char 192))) ;-- lower left corner   (+)
 (intern (string (code-char 217))) ;-- lower right corner  (+)
 (intern (string (code-char 179))) ;-- vertical bar
 (intern (string (code-char 196))) ;-- horizontal bar      (-)
 (list (code-char #x1d) (code-char #xe2))
                                   ;-- APL quad           (?)
 (QUOTE [)                         ;-- left bracket
 (QUOTE ])                         ;-- right bracket
 (QUOTE {)                         ;-- left brace
 (QUOTE })                         ;-- right brace
 (intern (string (code-char 194))) ;-- top    box tee      (+)
 (intern (string (code-char 193))) ;-- bottom box tee      (+)
 (intern (string (code-char 180))) ;-- right  box tee      (+)
 (intern (string (code-char 195))) ;-- left   box tee      (+)
 (intern (string (code-char 197))) ;-- center box tee      (+)
 (QUOTE |\\|)))                    ;-- back slash
```

44.1.8 defvar $specialCharacters

⟨*initvars*⟩+≡

```
(defvar |$specialCharacters| |$RTspecialCharacters|)
```

44.1.9 defvar $specialCharacterAlist

⟨*initvars*⟩+≡

```
(defvar |$specialCharacterAlist|
 '((|ulc| . 0)
   (|urc| . 1)
   (|llc| . 2)
   (|lrc| . 3)
   (|vbar| . 4)
   (|hbar| . 5)
   (|quad| . 6)
   (|lbrk| . 7)
   (|rbrk| . 8)
   (|lbrc| . 9)
   (|rbrc| . 10)
   (|ttee| . 11)
   (|btee| . 12)
   (|rtee| . 13)
   (|ltee| . 14)
   (|ctee| . 15)
   (|bslash| . 16)))
```

Chapter 45

Stream Handling

45.0.10 defun make-instream

⟨*defun make-instream*⟩≡
```
(defun make-instream (filespec &optional (recnum 0))
 (declare (ignore recnum))
   (cond ((numberp filespec) (make-synonym-stream '*terminal-io*))
         ((null filespec) (error "not handled yet"))
         (t (open (make-input-filename filespec)
                  :direction :input :if-does-not-exist nil))))
```

45.0.11 defun make-outstream

⟨*defun make-outstream*⟩≡
```
(defun make-outstream (filespec &optional (width nil) (recnum 0))
 (declare (ignore width) (ignore recnum))
   (cond ((numberp filespec) (make-synonym-stream '*terminal-io*))
         ((null filespec) (error "not handled yet"))
         (t (open (make-filename filespec) :direction :output))))
```

45.0.12 defun make-appendstream

⟨*defun make-appendstream*⟩≡

```
(defun make-appendstream (filespec &optional (width nil) (recnum 0))
 "fortran support"
 (declare (ignore width) (ignore recnum))
 (cond
  ((numberp filespec) (make-synonym-stream '*terminal-io*))
  ((null filespec) (error "make-appendstream: not handled yet"))
  ('else (open (make-filename filespec) :direction :output
          :if-exists :append :if-does-not-exist :create))))
```

45.0.13 defun defiostream

⟨*defun defiostream*⟩≡

```
(defun defiostream (stream-alist buffer-size char-position)
 (declare (ignore buffer-size))
   (let ((mode (or (cdr (assoc 'mode stream-alist)) 'input))
         (filename (cdr (assoc 'file stream-alist)))
         (dev (cdr (assoc 'device stream-alist))))
      (if (eq dev 'console) (make-synonym-stream '*terminal-io*)
       (let ((strm (case mode
                          ((output o) (open (make-filename filename)
                                            :direction :output))
                          ((input i) (open (make-input-filename filename)
                                           :direction :input)))))
         (if (and (numberp char-position) (> char-position 0))
           (file-position strm char-position))
           strm))))
```

45.0.14 defun shut

⟨*defun shut*⟩≡

```
(defun shut (st)
  (if (is-console st)
   st
   (if (streamp st) (close st) -1)))
```

45.0.15 defun eofp

⟨*defun eofp*⟩≡
```
(defun eofp (stream) (null (peek-char nil stream nil nil)))
```

45.0.16 defun makeStream

⟨*defun makeStream*⟩≡
```
(defun |makeStream| (append filename i j)
 (if append
  (make-appendstream filename i j)
  (make-outstream filename i j)))
```

Chapter 46

The Spad Server Mechanism

⟨*initvars*⟩+≡
```
(defvar $openServerIfTrue nil "t means try starting an open server")
```

⟨*initvars*⟩+≡
```
(defconstant $SpadServerName "/tmp/.d" "the name of the spad server socket")
```

⟨*initvars*⟩+≡
```
(defvar |$SpadServer| nil "t means Scratchpad acts as a remote server")
```

46.0.17 openserver (openserver)

This is a cover function for the C code used for communication interface.

⟨*defun openserver*⟩≡
```
(defun openserver (name)
  (open_server name))
```

Chapter 47

Axiom Build-time Functions

47.0.18 defun spad-save

The **spad-save** function is just a cover function for more lisp system specific save functions. There is no standard name for saving a lisp image so we make one and conditionalize it at compile time.

This function is passed the name of an image that will be saved. The saved image contains all of the loaded functions.

This is used in the src/interp/Makefile.pamphlet in three places:

creating depsys, an image for compiling axiom.

Some of the Common Lisp code we compile uses macros which are assumed to be available at compile time. The **DEPSYS** image is created to contain the compile time environment and saved. We pipe compile commands into this environment to compile from Common Lisp to machine dependent code.

```
DEPSYS=${OBJ}/${SYS}/bin/depsys
```

creating savesys, an image for running axiom.

Once we've compile all of the Common Lisp files we fire up a clean lisp image called **LOADSYS**, load all of the final executable code and save it out as **SAVESYS**. The **SAVESYS** image is copied to the `${MNT}/${SYS}/bin` subdirectory and becomes the axiom executable image.

```
LOADSYS= ${OBJ}/${SYS}/bin/lisp
SAVESYS= ${OBJ}/${SYS}/bin/interpsys
AXIOMSYS= ${MNT}/${SYS}/bin/AXIOMsys
```

creating debugsys, an image with all interpreted functions loaded.

Occasionally we need to really get into the system internals. The best way to do this is to run almost all of the lisp code interpreted rather than compiled (note that cfuns.lisp and sockio.lisp still need to be loaded in compiled form as they depend on the loader to link with lisp internals). This image is nothing more than a load of the file src/interp/debugsys.lisp.pamphlet. If you need to make test modifications you can add code to that file and it will show up here.

```
DEBUGSYS=${OBJ}/${SYS}/bin/debugsys
```

⟨*defun spad-save*⟩≡

```
(defun user::spad-save (save-file)
  (declare (special |$SpadServer| $openServerIfTrue))
  (setq |$SpadServer| nil)
  (setq $openServerIfTrue t)
#+:AKCL
  (system::save-system save-file)
#+:allegro
  (if (fboundp 'boot::restart)
   (excl::dumplisp :name save-file :restart-function #'boot::restart)
   (excl::dumplisp :name save-file))
#+Lucid
  (if (fboundp 'boot::restart)
   (sys::disksave save-file :restart-function #'boot::restart)
   (sys::disksave save-file))
#+:CCL
  (preserve)
)
```

Chapter 48

Exposure Groups

Exposure groups are a way of controlling the namespace available to the user. Certain algebra files are only useful for internal purposes but they contain functions have common names (like "map". In order to separate the user visible functions from the internal functions the algebra files are collected into "exposure groups". These large groups are grouped into sets in the file `exposed.lsp` which lives in the algebra subdirectory.

Exposure group information is kept in the local frame. For more information "The Frame Mechanism" **??** on page **??**.

48.0.19 loadExposureGroupData (loadExposureGroupData)

This function is called from "restart" (4.3.2 p 11) at system startup time to load the file `exposed.lsp` to set up the exposure group information.

⟨*defun loadExposureGroupData*⟩≡

```
(defun |loadExposureGroupData| ()
 (cond
  ((load "./exposed" :verbose nil :if-does-not-exist nil)
    '|done|)
  ((load (concat (getenviron "AXIOM") "/algebra/exposed")
     :verbose nil :if-does-not-exist nil)
   '|done|)
  (t '|failed|) ))
```

Chapter 49

System Statistics

49.0.20 statisticsInitialization (statisticsInitialization)

⟨*defun statisticsInitialization*⟩≡

```
(defun |statisticsInitialization| ()
 "initialize the garbage collection timer"
 #+:akcl (system:gbc-time 0)
 nil)
```

Chapter 50

Special Lisp Functions

50.0.21 defmacro identp

⟨*defmacro identp*⟩≡
```
(defmacro identp (x)
 (if (atom x)
  `(and ,x (symbolp ,x))
  (let ((xx (gensym)))
   `(let ((,xx ,x))
     (and ,xx (symbolp ,xx))))))
```

50.0.22 defun concat

⟨*defun concat*⟩≡
```
(defun concat (a b &rest l)
 (if (bit-vector-p a)
   (if l
     (apply #'concatenate 'bit-vector a b l)
     (concatenate 'bit-vector a b))
   (if l
     (apply #'system:string-concatenate a b l)
     (system:string-concatenate a b))))
```

50.0.23 defun functionp

⟨*defun functionp*⟩≡
```
(defun |functionp| (fn)
 (if (identp fn)
  (and (fboundp fn) (not (macro-function fn)))
  (functionp fn)))
```

;; ————————¿ NEW DEFINITION (override in msgdb.boot.pamphlet)

50.0.24 defun brightprint

⟨*defun brightprint*⟩≡
```
(defun brightprint (x)
 (messageprint x))
```

;; ————————¿ NEW DEFINITION (override in msgdb.boot.pamphlet)

50.0.25 defun brightprint-0

⟨*defun brightprint-0*⟩≡
```
(defun brightprint-0 (x)
 (messageprint-1 x))
```

50.0.26 defun member

⟨*defun member*⟩≡
```
(defun |member| (item sequence)
 (cond
   ((symbolp item) (member item sequence :test #'eq))
   ((stringp item) (member item sequence :test #'equal))
   ((and (atom item) (not (arrayp item))) (member item sequence))
   (t (member item sequence :test #'equalp))))
```

50.0.27 defun messageprint

⟨*defun messageprint*⟩≡
```
(defun messageprint (x)
 (mapc #'messageprint-1 x))
```

50.0.28 defun messageprint-1

⟨*defun messageprint-1*⟩≡
```
(defun messageprint-1 (x)
 (cond
  ((or (eq x '|%l|) (equal x "%l")) (terpri))
  ((stringp x) (princ x))
  ((identp x) (princ x))
  ((atom x) (princ x))
  ((princ "(")
   (messageprint-1 (car x))
   (messageprint-2 (cdr x))
   (princ ")"))))
```

50.0.29 defun messageprint-2

⟨*defun messageprint-2*⟩≡
```
(defun messageprint-2 (x)
  (if (atom x)
   (unless x (progn (princ " . ") (messageprint-1 x)))
   (progn (princ " ") (messageprint-1 (car x)) (messageprint-2 (cdr x)))))
```

50.0.30 defun sayBrightly1

⟨*defun sayBrightly1*⟩≡
```
(defun sayBrightly1 (x *standard-output*)
  (if (atom x)
   (progn (brightprint-0 x) (terpri) (force-output))
   (progn (brightprint   x) (terpri) (force-output))))
```

50.0.31 defvar $algebraOutputStream

⟨*initvars*⟩+≡
```
(defvar |$algebraOutputStream| *standard-output*)
```

50.0.32 defun sayMSG

⟨*defun sayMSG*⟩≡

```
(defun |sayMSG| (x)
 (declare (special |$algebraOutputStream|))
 (when x (sayBrightly1 x |$algebraOutputStream|)))
```

Chapter 51

Dangling references

51.1 shell variables

AXIOM

51.2 catch tags

|coerceFailure|
filenam
|$intTopLevel|
|letPrint2|
|$quitTag|
|ScanOrPairVecAnswer|
|top_level|
|writifyTag|

51.3 catch tags

|ScanOrPairVecAnswer|
|top_level|
|writifyTag|

51.4 defined special variables

|$abbreviateTypes|
|$algebraFormat|
|$algebraOutputFile|
|$algebraOutputStream|

```
|$asharpCmdlineFlags|
|$BreakMode|
|$clearExcept|
|$clearOptions|
|$CommandSynonymAlist|
|$compileDontDefineFunctions|
|$compileRecurrence|
compiler::*compile-verbose*
credits
|$defaultFortranType|
*default-pathname-defaults*
|$defaultSpecialCharacters|
|$displayDroppedMap|
|$displayMsgNumber|
|$displayOptions|
|$displaySetValue|
|$displayStartMsgs|
|$formulaFormat|
|$formulaOutputFile|
|$fortIndent|
|$fortInts2Floats|
|$fortLength|
|$fortranArrayStartingIndex|
|$fortranDirectory|
|$fortranFormat|
|$fortranLibraries|
|$fortranOptimizationLevel|
|$fortranOutputFile|
|$fortranPrecision|
|$fortranSegment|
|$fortranTmpDir|
|$fortPersistence|
|$fractionDisplayType|
|$frameMessages|
|$fullScreenSysVars|
|$giveExposureWarning|
|$HiFiAccess|
|$highlightAllowed|
|$historyDirectory|
|$historyDisplayWidth|
|$historyFileType|
|$InitialCommandSynonymAlist|
|$inputPromptType|
|$linearFormatScripts|
$linelength
|$mapSubNameAlist|
|$mathmlFormat|
|$mathmlOutputFile|
|$maximumFortranExpressionLength|
|$nagEnforceDouble|
```

```
|$nagHost|
|$nagMessages|
|$noParseCommands|
|$oldHistoryFileName|
|$openMathFormat|
|$openMathOutputFile|
$openServerIfTrue
|$optionAlist|
|$options|
|$plainRTspecialCharacters|
|$plainSpecialCharacters0|
|$plainSpecialCharacters1|
|$plainSpecialCharacters2|
|$plainSpecialCharacters3|
$prettyprint
|$printAnyIfTrue|
|$printFortranDecs|
|$printLoadMsgs|
|$printMsgsToFile|
|$printStatisticsSummaryIfTrue|
|$printTimeIfTrue|
|$printTypeIfTrue|
|$printVoidIfTrue|
|$quitCommandType|
|$reportBottomUpFlag|
|$reportCoerceIfTrue|
|$reportCompilation|
|$reportEachInstantiation|
|$reportInstantiations|
|$reportInterpOnly|
|$reportOptimization|
|$reportSpadTrace|
|$RTspecialCharacters|
*standard-input*
*standard-output*
|$SpadServer|
$SpadServerName
|$specialCharacterAlist|
|$specialCharacters|
|$streamCount|
|$streamsShowAll|
compiler::*suppress-compiler-notes*
compiler::*suppress-compiler-warnings*
|$systemCommandFunction|
$syscommands
|$systemCommands|
*terminal-io*
|$testingSystem|
|$texFormat|
|$texOutputFile|
```

```
|$tokenCommands|
system::*top-level-hook*
|$tracedMapSignatures|
|$traceNoisely|
|$traceOptionList|
underbar
|$useEditorForShowOutput|
|$useFullScreenHelp|
|$useInternalHistoryTable|
|$useIntrinsicFunctions|
|$UserLevel|
|$whatOptions|
```

51.5 undefined special variables

```
|$attributeDb|
$boot
|$cacheAlist|
|$cacheCount|
|$CatOfCatDatabase|
|$CloseClient|
|$coerceIntByMapCounter|
|$compileMapFlag|
|$ConstructorCache|
|$constructors|
/countlist
curinstream
curoutstream
$current-directory
|$currentFrameNum|
|$currentLine|
$dalymode
|$defaultMsgDatabaseName|
|$dependeeClosureAlist|
$directory-list
|$displayStartMsgs|
|$domains|
|$DomOfCatDatabase|
|$domainTraceNameAssoc|
|$doNotAddEmptyModeIfTrue|
|$e|
|$echoLineStack|
/editfile
|$EmptyEnvironment|
|$env|
*eof*
|$erMsgToss|
|$existingFiles|
```

```
|$fn|
|$formulaOutputStream|
|$fortranOutputStream|
|$frameMessages|
|$frameRecord|
|$fromSpadTrace|
|$functionTable|
|$globalExposureGroupAlist|
|$HistList|
|$HistListAct|
|$HistListLen|
|$HistRecord|
|$inLispVM|
|$inclAssertions|
|$InitialModemapFrame|))
in-stream
|$InteractiveFrame|
|$internalHistoryTable|
|$interpreterFrameName|
|$interpreterFrameRing|
|$intRestart|
|$intTopLevel|
|$IOindex|
|$JoinOfCatDatabase|
|$JoinOfDomDatabase|
|$lastPos|
|$lastUntraced|
|$letAssoc|
|$libQuiet|
$library-directory-list
|$lines|
|$localExposureData|
|$localExposureDataDefault|
|$lookupDefaults|
|$mathmlOutputStream|
|$mathTraceList|
|$mkTestInputStack|
|$msgAlist|
|$msgDatabase|
|$msgDatabaseName|
|$ncMsgList|
|$newConlist|
|$NonNullStream|
|$nopos|
|$newcompErrorCount|
|$newcompMode|
$newspad
|$NullStream|
|$okToExecuteMachineCode|
|$openMathOutputStream|
```

```
|$operationNameList|
|$outputLibraryName|
|$OutputForm|
|$packages|
/pretty
|$previousBindings|
|$PrintCompilerMessageIfTrue|
|$printLoadMsgs|
|$promptMsg|
|$QuickLet
|$quitTag|
$relative-directory-list
$relative-library-directory-list
|$seen|
|$SessionManager|
|$setOptions|
|$slamFlag|
/sourcefiles
|$sourceFiles|
/spacelist
$spad
$spadroot
|$texOutputStream|
/timerlist
|$timerTicksPerSecond|
|Top|
|$tracedMapSignatures|
|$tracedModemap|
|$tracedSpadModemap|
|$traceErrorStack|
$traceletflag
|$traceletFunctions|
|$undoFlag|
|$useFullScreenHelp|
|$UserAbbreviationsAlist|
|$variableNumberAlist|
|$Void|
|$writifyComplained|
/wsname
|$xdatabase|
```

51.6 undefined functions

```
currenttime
error
|incRenumber|
|incRgen1|
|insertpile|
```

```
|intloopEchoParse|
|intloopProcess|
|intloopProcessString|
|intnplisp|
|lineoftoks|
|ncloopEchoParse|
|ncloopProcess|
|resetStackLimits|
|shoeread-line|
stringimage
```

Chapter 52

The Interpreter

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

⟨*defmacro funfind*⟩
⟨*defmacro identp*⟩
⟨*defmacro Rest*⟩

⟨*defun abbQuery*⟩
⟨*defun abbreviations*⟩
⟨*defun abbreviationsSpad2Cmd*⟩
⟨*defun addInputLibrary*⟩
⟨*defun addNewInterpreterFrame*⟩
⟨*defun addTraceItem*⟩
⟨*defun apropos*⟩
⟨*defun augmentTraceNames*⟩

⟨*defun break*⟩
⟨*defun breaklet*⟩
⟨*defun brightprint*⟩
⟨*defun brightprint-0*⟩
⟨*defun browse*⟩

⟨*defun changeHistListLen*⟩
⟨*defun changeToNamedInterpreterFrame*⟩
⟨*defun charDigitVal*⟩
⟨*defun cleanupLine*⟩
⟨*defun clear*⟩
⟨*defun clearCmdAll*⟩
⟨*defun clearCmdCompletely*⟩

⟨*defun clearCmdExcept*⟩
⟨*defun clearCmdParts*⟩
⟨*defun clearCmdSortedCaches*⟩
⟨*defun clearFrame*⟩
⟨*defun clearSpad2Cmd*⟩
⟨*defun close*⟩
⟨*defun closeInterpreterFrame*⟩
⟨*defun coerceSpadArgs2E*⟩
⟨*defun coerceSpadFunValue2E*⟩
⟨*defun coerceTraceArgs2E*⟩
⟨*defun coerceTraceFunValue2E*⟩
⟨*defun compileBoot*⟩
⟨*defun compiler*⟩
⟨*defun concat*⟩
⟨*defun copyright*⟩
⟨*defun countCache*⟩
⟨*defun createCurrentInterpreterFrame*⟩
⟨*defun credits*⟩

⟨*defun defiostream*⟩
⟨*defun Delay*⟩
⟨*defun describeAsharpArgs*⟩
⟨*defun describeFortPersistence*⟩
⟨*defun describeInputLibraryArgs*⟩
⟨*defun describeOutputLibraryArgs*⟩
⟨*defun describeProtectedSymbolsWarning*⟩
⟨*defun describeProtectSymbols*⟩
⟨*defun describeSetFortDir*⟩
⟨*defun describeSetFortTmpDir*⟩
⟨*defun describeSetFunctionsCache*⟩
⟨*defun describeSetLinkerArgs*⟩
⟨*defun describeSetNagHost*⟩
⟨*defun describeSetOutputAlgebra*⟩
⟨*defun describeSetOutputFormula*⟩
⟨*defun describeSetOutputFortran*⟩
⟨*defun describeSetOutputMathml*⟩
⟨*defun describeSetOutputOpenMath*⟩
⟨*defun describeSetOutputTex*⟩
⟨*defun describeSetStreamsCalculate*⟩
⟨*defun dewritify*⟩
⟨*defun dewritify,dewritifyInner*⟩
⟨*defun dewritify,is?*⟩
⟨*defun diffAlist*⟩
⟨*defun disableHist*⟩
⟨*defun display*⟩
⟨*defun displayExposedConstructors*⟩

⟨*defun displayExposedGroups*⟩
⟨*defun displayFrameNames*⟩
⟨*defun displayHiddenConstructors*⟩
⟨*defun displayMacros*⟩
⟨*defun displayOperations*⟩
⟨*defun displaySetOptionInformation*⟩
⟨*defun displaySetVariableSettings*⟩
⟨*defun displaySpad2Cmd*⟩
⟨*defun domainToGenvar*⟩
⟨*defun dropInputLibrary*⟩

⟨*defun edit*⟩
⟨*defun editSpad2Cmd*⟩
⟨*defun Else?*⟩
⟨*defun Elseif?*⟩
⟨*defun emptyInterpreterFrame*⟩
⟨*defun eofp*⟩

⟨*defun fetchOutput*⟩
⟨*defun filterAndFormatConstructors*⟩
⟨*defun findFrameInRing*⟩
⟨*defun flattenOperationAlist*⟩
⟨*defun frame*⟩
⟨*defun frameEnvironment*⟩
⟨*defun frameExposureData*⟩
⟨*defun frameHiFiAccess*⟩
⟨*defun frameHistList*⟩
⟨*defun frameHistListAct*⟩
⟨*defun frameHistListLen*⟩
⟨*defun frameHistoryTable*⟩
⟨*defun frameHistRecord*⟩
⟨*defun frameInteractive*⟩
⟨*defun frameIOIndex*⟩
⟨*defun frameName*⟩
⟨*defun frameNames*⟩
⟨*defun frameSpad2Cmd*⟩
⟨*defun functionp*⟩
⟨*defun funfind,LAM*⟩

⟨*defun genDomainTraceName*⟩
⟨*defun gensymInt*⟩
⟨*defun getAliasIfTracedMapParameter*⟩
⟨*defun getBpiNameIfTracedMap*⟩
⟨*defun get-current-directory*⟩
⟨*defun getenviron*⟩
⟨*defun getMapSig*⟩

⟨*defun getMapSubNames*⟩
⟨*defun getOption*⟩
⟨*defun getPreviousMapSubNames*⟩
⟨*defun getTraceOption*⟩
⟨*defun getTraceOption,hn*⟩
⟨*defun getTraceOptions*⟩

⟨*defun handleNoParseCommands*⟩
⟨*defun hasPair*⟩
⟨*defun help*⟩
⟨*defun helpSpad2Cmd*⟩
⟨*defun histFileErase*⟩
⟨*defun histFileName*⟩
⟨*defun histInputFileName*⟩
⟨*defun history*⟩
⟨*defun historySpad2Cmd*⟩

⟨*defun If?*⟩
⟨*defun importFromFrame*⟩
⟨*defun incBiteOff*⟩
⟨*defun incLude*⟩
⟨*defun incLude1*⟩
⟨*defun incFileName*⟩
⟨*defun incRgen*⟩
⟨*defun incRgen1*⟩
⟨*defun incStream*⟩
⟨*defun initHist*⟩
⟨*defun initHistList*⟩
⟨*defun initializeInterpreterFrameRing*⟩
⟨*defun initializeSetVariables*⟩
⟨*defun init-memory-config*⟩
⟨*defun initroot*⟩
⟨*defun intloop*⟩
⟨*defun intloopInclude*⟩
⟨*defun intloopInclude0*⟩
⟨*defun intloopPrefix?*⟩
⟨*defun intloopReadConsole*⟩
⟨*defun isDomainOrPackage*⟩
⟨*defun isInterpOnlyMap*⟩
⟨*defun isListOfIdentifiers*⟩
⟨*defun isListOfIdentifiersOrStrings*⟩
⟨*defun isSubForRedundantMapName*⟩
⟨*defun isTraceGensym*⟩
⟨*defun isUncompiledMap*⟩

⟨*defun KeepPart?*⟩

⟨*defun lassocSub*⟩
⟨*defun leaveScratchpad*⟩
⟨*defun letPrint*⟩
⟨*defun letPrint2*⟩
⟨*defun letPrint3*⟩
⟨*defun listConstructorAbbreviations*⟩
⟨*defun loadExposureGroupData*⟩

⟨*defun make-absolute-filename*⟩
⟨*defun make-appendstream*⟩
⟨*defun makeHistFileName*⟩
⟨*defun makeInitialModemapFrame*⟩
⟨*defun make-instream*⟩
⟨*defun make-outstream*⟩
⟨*defun makeStream*⟩
⟨*defun mapLetPrint*⟩
⟨*defun member*⟩
⟨*defun messageprint*⟩
⟨*defun messageprint-1*⟩
⟨*defun messageprint-2*⟩
⟨*defun mkprompt*⟩

⟨*defun ncIntLoop*⟩
⟨*defun ncloopCommand*⟩
⟨*defun ncloopEscaped*⟩
⟨*defun ncloopIncFileName*⟩
⟨*defun ncloopInclude*⟩
⟨*defun ncloopInclude0*⟩
⟨*defun ncloopInclude1*⟩
⟨*defun ncloopPrefix?*⟩
⟨*defun ncTopLevel*⟩
⟨*defun newHelpSpad2Cmd*⟩
⟨*defun nextInterpreterFrame*⟩

⟨*defun oldHistFileName*⟩
⟨*defun openOutputLibrary*⟩
⟨*defun openserver*⟩
⟨*defun orderBySlotNumber*⟩

⟨*defun pcounters*⟩
⟨*defun pquit*⟩
⟨*defun pquitSpad2Cmd*⟩
⟨*defun previousInterpreterFrame*⟩
⟨*defun protectedSymbolsWarning*⟩
⟨*defun protectSymbols*⟩

⟨*defun prTraceNames*⟩
⟨*defun prTraceNames,fn*⟩
⟨*defun pspacers*⟩
⟨*defun ptimers*⟩
⟨*defun putHist*⟩

⟨*defun queryClients*⟩
⟨*defun quit*⟩
⟨*defun quitSpad2Cmd*⟩

⟨*defun rassocSub*⟩
⟨*defun readHiFi*⟩
⟨*defun reclaim*⟩
⟨*defun recordFrame*⟩
⟨*defun recordNewValue*⟩
⟨*defun recordNewValue0*⟩
⟨*defun recordOldValue*⟩
⟨*defun recordOldValue0*⟩
⟨*defun removeOption*⟩
⟨*defun removeTracedMapSigs*⟩
⟨*defun removeUndoLines*⟩
⟨*defun reportSpadTrace*⟩
⟨*defun reportUndo*⟩
⟨*defun reroot*⟩
⟨*defun resetCounters*⟩
⟨*defun resetInCoreHist*⟩
⟨*defun resetSpacers*⟩
⟨*defun resetStackLimits*⟩
⟨*defun resetTimers*⟩
⟨*defun resetWorkspaceVariables*⟩
⟨*defun restart*⟩
⟨*defun restoreHistory*⟩
⟨*defun runspad*⟩

⟨*defun safeWritify*⟩
⟨*defun saveHistory*⟩
⟨*defun saveMapSig*⟩
⟨*defun sayAllCacheCounts*⟩
⟨*defun sayBrightly1*⟩
⟨*defun sayCacheCount*⟩
⟨*defun sayExample*⟩
⟨*defun sayMSG*⟩
⟨*defun ScanOrPairVec*⟩
⟨*defun ScanOrPairVec,ScanOrInner*⟩
⟨*defun selectOption*⟩
⟨*defun selectOptionLC*⟩

⟨*defun serverReadLine*⟩
⟨*defun set*⟩
⟨*defun set1*⟩
⟨*defun setAsharpArgs*⟩
⟨*defun setCurrentLine*⟩
⟨*defun setExpose*⟩
⟨*defun setExposeAdd*⟩
⟨*defun setExposeAddConstr*⟩
⟨*defun setExposeAddGroup*⟩
⟨*defun setExposeDrop*⟩
⟨*defun setExposeDropConstr*⟩
⟨*defun setExposeDropGroup*⟩
⟨*defun setFortDir*⟩
⟨*defun setFortPers*⟩
⟨*defun setFortTmpDir*⟩
⟨*defun setFunctionsCache*⟩
⟨*defun setHistoryCore*⟩
⟨*defun setInputLibrary*⟩
⟨*defun setIOindex*⟩
⟨*defun setLinkerArgs*⟩
⟨*defun setNagHost*⟩
⟨*defun setOutputAlgebra*⟩
⟨*defun setOutputCharacters*⟩
⟨*defun setOutputFormula*⟩
⟨*defun setOutputFortran*⟩
⟨*defun setOutputLibrary*⟩
⟨*defun setOutputMathml*⟩
⟨*defun setOutputOpenMath*⟩
⟨*defun setOutputTex*⟩
⟨*defun set-restart-hook*⟩
⟨*defun setStreamsCalculate*⟩
⟨*defun shortenForPrinting*⟩
⟨*defun showInOut*⟩
⟨*defun showInput*⟩
⟨*defun shut*⟩
⟨*defun SkipEnd?*⟩
⟨*defun SkipPart?*⟩
⟨*defun Skipping?*⟩
⟨*defun spad*⟩
⟨*defun spadClosure?*⟩
⟨*defun SpadInterpretStream*⟩
⟨*defun spadReply*⟩
⟨*defun spadReply,printName*⟩
⟨*defun spadrread*⟩
⟨*defun spadrwrite*⟩
⟨*defun spadrwrite0*⟩

⟨*defun spad-save*⟩
⟨*defun spadTrace*⟩
⟨*defun spadTraceAlias*⟩
⟨*defun spadTrace,g*⟩
⟨*defun spadTrace,isTraceable*⟩
⟨*defun spadUntrace*⟩
⟨*defun stackTraceOptionError*⟩
⟨*defun statisticsInitialization*⟩
⟨*defun stupidIsSpadFunction*⟩
⟨*defun subTypes*⟩
⟨*defun summary*⟩

⟨*defun ?t*⟩
⟨*defun Top?*⟩
⟨*defun trace*⟩
⟨*defun trace1*⟩
⟨*defun traceDomainConstructor*⟩
⟨*defun traceDomainLocalOps*⟩
⟨*defun tracelet*⟩
⟨*defun traceOptionError*⟩
⟨*defun /tracereply*⟩
⟨*defun traceReply*⟩
⟨*defun traceSpad2Cmd*⟩
⟨*defun translateTrueFalse2YesNo*⟩
⟨*defun translateYesNo2TrueFalse*⟩
⟨*defun transOnlyOption*⟩
⟨*defun transTraceItem*⟩

⟨*defun undo*⟩
⟨*defun undoChanges*⟩
⟨*defun undoCount*⟩
⟨*defun undoFromFile*⟩
⟨*defun undoInCore*⟩
⟨*defun undoLocalModemapHack*⟩
⟨*defun undoSingleStep*⟩
⟨*defun undoSteps*⟩
⟨*defun untrace*⟩
⟨*defun untraceDomainConstructor*⟩
⟨*defun untraceDomainConstructor,keepTraced?*⟩
⟨*defun untraceDomainLocalOps*⟩
⟨*defun untraceMapSubNames*⟩
⟨*defun unwritable?*⟩
⟨*defun updateCurrentInterpreterFrame*⟩
⟨*defun updateFromCurrentInterpreterFrame*⟩
⟨*defun updateHist*⟩
⟨*defun updateInCoreHist*⟩

⟨*defun validateOutputDirectory*⟩

⟨*defun what*⟩
⟨*defun whatConstructors*⟩
⟨*defun whatSpad2Cmd*⟩
⟨*defun whatSpad2Cmd,fixpat*⟩
⟨*defun with*⟩
⟨*defun workfiles*⟩
⟨*defun workfilesSpad2Cmd*⟩
⟨*defun writeHiFi*⟩
⟨*defun writeHistModesAndValues*⟩
⟨*defun writeInputLines*⟩
⟨*defun writify*⟩
⟨*defun writifyComplain*⟩
⟨*defun writify,writifyInner*⟩

⟨*defun yesanswer*⟩

⟨*defun zsystemdevelopment*⟩
⟨*defun zsystemdevelopment1*⟩
⟨*defun zsystemDevelopmentSpad2Cmd*⟩

⟨*postvars*⟩

Chapter 53

The Global Variables

53.1 Star Global Variables

NAME	SET	USE
eof*	ncTopLevel	
features*		restart
package*		restart
standard-input*		ncIntLoop
standard-output*		ncIntLoop
top-level-hook*	set-restart-hook	

53.1.1 *eof*

The `*eof*` variable is set to NIL in ncTopLevel.

53.1.2 *features*

The `*features*` variable from common lisp is tested for the presence of the `:unix` keyword. Apparently this controls the use of Saturn, a previous Axiom frontend. The Saturn frontend was never released as open source and so this test and the associated variables are probably not used.

53.1.3 *package*

The `*package*` variable, from common lisp, is set in restart to the BOOT package where the intepreter lives.

53.1.4 *standard-input*

The `*standard-input*` common lisp variable is used to set the curinstream variable in ncIntLoop.

This variable is an argument to serverReadLine in the intloopReadConsole function.

53.1.5 *standard-output*

The `*standard-output*` common lisp variable is used to set the curoutstream variable in ncIntLoop.

53.1.6 *top-level-hook*

The `*top-level-hook*` common lisp variable contains the name of a function to invoke when an image is started. In our case it is called restart. This is the entry point to the Axiom interpreter.

53.2 Dollar Global Variables

NAME	SET	USE
$boot	ncTopLevel	
coerceFailure		runspad
curinstream	ncIntLoop	
curoutstream	ncIntLoop	
$current-directory	restart	
	reroot	
$currentLine	restart	removeUndoLines
$dalymode		intloopReadConsole
$defaultMsgDatabaseName	reroot	
$directory-list	reroot	
$displayStartMsgs		restart
$e	ncTopLevel	
$erMsgToss	SpadInterpretStream	
$fn	SpadInterpretStream	
$frameRecord	initvars	
	clearFrame	
	undoSteps	undoSteps
	recordFrame	recordFrame
$HiFiAccess	initHist	historySpad2Cmd
	historySpad2Cmd	
		setHistoryCore
$HistList	initHist	
$HistListAct	initHist	
$HistListLen	initHistList	
$HistRecord	initHistList	
$historyDirectory		makeHistFileName
		makeHistFileName
$historyFileType	initvars	histInputFileName
$inclAssertions	SpadInterpretStream	
$inLispVM	spad	
$InteractiveFrame	restart	ncTopLevel
	undo	recordFrame
	undoSteps	undoSteps
		reportUndo
$internalHistoryTable	initvars	
$interpreterFrameName	initializeInterpreterFrameRing	
$interpreterFrameRing	initializeInterpreterFrameRing	
$InitialModemapFrame		makeInitialModemapFrame
$intRestart		intloop
$intTopLevel	intloop	
$IOindex	restart	historySpad2Cmd
	removeUndoLines	undoCount
$genValue	bookvol5	i-toplev
		i-analy
		i-syscmd
		i-spec1
		i-spec2
		i-map
$lastPos	SpadInterpretStream	
$libQuiet	SpadInterpretStream	
$library-directory-list	reroot	
$msgDatabaseName	reroot *	
$ncMsgList	SpadInterpretStream	
$newcompErrorCount	SpadInterpretStream	
$newcompMode	SpadInterpretStream	
$newspad	ncTopLevel	
$nopos		SpadInterpretStream

53.2.1 $boot

The `$boot` variable is set to NIL in ncTopLevel.

53.2.2 coerceFailure

The coerceFailure symbol is a catch tag used in runspad to catch an exit from ncTopLevel.

53.2.3 $current-directory

This is set to the value returned by the `get-current-directory` function in "restart" (4.3.2 p 11). It is set to the argument of the `reroot` function.

So during execute both `$current-directory` and `$spadroot` reflect the value of the AXIOM shell variable.

53.2.4 $currentLine

The `$currentLine` line is set to NIL in restart. It is used in removeUndoLines in the undo mechanism.

53.2.5 $defaultMsgDatabaseName

The `$defaultMsgDatabaseName` is the absolute path to the s2-us.msgs file which contains all of the english language messages output by the system.

53.2.6 $directory-list

The `$directory-list` is a list of absolute directory names. These names are made absolute by mapping the make-absolute-filename over the variable `$relative-directory-list`.

53.2.7 $displayStartMsgs

The `$displayStartMsgs` variable is used in restart but is not set so this is likely a bug.

53.2.8 $e

The `$e` variable is set to the value of `$InteractiveFrame` which is set in restart to the value of the call to the makeInitialModemapFrame function. This function simply returns a copy of the variable `$InitialModemapFrame`.

Thus `$e` is a copy of the variable `$InitialModemapFrame`.

This variable is used in the undo mechanism.

53.2.9 $erMsgToss

The `$erMsgToss` variable is set to NIL in SpadInterpretStream.

53.2.10 $fn

The `$fn` variable is set in SpadInterpretStream. It is set to the second argument which is a list. It appears that this list has the same structure as an argument to the LispVM rdefiostream function.

53.2.11 $frameRecord

`$frameRecord = [delta1, delta2,... ]` where delta(i) contains changes in the "backwards" direction. Each delta(i) has the form ((var . proplist)...) where proplist denotes an ordinary proplist. For example, an entry of the form ((x (value) (mode (Integer)))...) indicates that to undo 1 step, x's value is cleared and its mode should be set to (Integer).

A delta(i) of the form (systemCommand . delta) is a special delta indicating changes due to system commands executed between the last command and the current command. By recording these deltas separately, it is possible to undo to either BEFORE or AFTER the command. These special delta(i)s are given ONLY when a a system command is given which alters the environment.

Note: recordFrame('system) is called before a command is executed, and recordFrame('normal) is called after (see processInteractive1). If no changes are found for former, no special entry is given.

This is part of the undo mechanism.

53.2.12 $genValue

If the `$genValue` variable is true then evaluate generated code, otherwise leave code unevaluated. If `$genValue` is false then we are compiling. This variable is only defined and used locally.

⟨*initvars*⟩+≡

```
(defvar |$genValue| nil "evaluate generated code if true")
```

53.2.13 $HiFiAccess

The `$HiFiAccess` is set by initHist to T. It is a flag used by the history mechanism to record whether the history function is currently on. It can be reset by using the axiom command

```
)history off
```

It appears that the name means "History File Access".

The `$HiFiAccess` variable is used by historySpad2Cmd to check whether history is turned on. T means it is, NIL means it is not.

53.2.14 $HistList

Thie `$HistList` variable is set by initHistList to an initial value of NIL elements. The last element of the list is smashed to point to the first element to make the list circular. This is a circular list of length `$HistListLen`.

53.2.15 $HistListAct

The `$HistListAct` variable is set by initHistList to 0. This variable holds the actual number of elements in the history list. This is the number of "undoable" steps.

53.2.16 $HistListLen

The `$HistListLen` variable is set by initHistList to 20. This is the length of a circular list maintained in the variable `$HistList`.

53.2.17 $HistRecord

The `$HistRecord` variable is set by initHistList to NIL. `$HistRecord` collects the input line, all variable bindings and the output of a step, before it is written to the file named by the function histFileName.

53.2.18 $historyFileType

The `$historyFileType` is set at load time by a call to initvars to a value of "axh". It appears that this is intended to be used as a filetype extension. It is part of the history mechanism. It is used in makeHistFileName as part of the history file name.

53.2.19 $inclAssertions

The `$inclAssertions` is set in the function SpadInterpretStream to the list (aix —CommonLisp—)

53.2.20 $internalHistoryTable

The `$internalHistoryTable` variable is set at load time by a call to initvars to a value of NIL. It is part of the history mechanism.

53.2.21 $interpreterFrameName

The `$interpreterFrameName` variable, set in initializeInterpreterFrameRing to the constant initial to indicate that this is the initial (default) frame.

Frames are structures that capture all of the variables defined in a session. There can be multiple frames and the user can freely switch between them. Frames are kept in a ring data structure so you can move around the ring.

53.2.22 $interpreterFrameRing

The `$interpreterFrameRing` is set to a pair whose car is set to the result of emptyInterpreterFrame

53.2.23 $InitialModemapFrame

This variable is copied and returned by the function `makeInitialModemapFrame`. There is no initial value so this is probably a bug.

53.2.24 $inLispVM

The `$inLispVM` is set to NIL in spad. LispVM is a non-common lisp that runs on IBM/370 mainframes. This is probably dead code. It appears that this list has the same structure as an argument to the LispVM rdefiostream function.

53.2.25 $InteractiveFrame

The `$InteractiveFrame` is set in restart to the value of the call to the makeInitialModemapFrame function. This function simply returns a copy of the variable `$InitialModemapFrame`

53.2.26 $intRestart

The `$intRestart` variable is used in intloop but has no value. This is probably a bug. While the variable's value is unchanged the system will continually reenter the SpadInterpretStream function.

53.2.27 $intTopLevel

The `$intTopLevel` is a catch tag. Throwing to this tags which is caught in the intloop will restart the SpadInterpretStream function.

53.2.28 $IOindex

The `$IOindex` index variable is set to 1 in restart. This variable is used in the historySpad2Cmd function in the history mechanism. It is set in the removeUndoLines function in the undo mechanism.

This is used in the undo mechanism in function undoCount to compute the number of undos. You can't undo more actions then have already happened.

53.2.29 $lastPos

The `$lastPos` variable is set in SpadInterpretStream to the value of the `$nopos` variable. Since `$nopos` appears to have no value this is likely a bug.

53.2.30 $libQuiet

The `$libQuiet` variable is set to the third argument of the SpadInterpretStream function. This is passed from intloop with the value of T. This variable appears to be intended to control the printing of library loading messages which would need to be suppressed if input was coming from a file.

53.2.31 $library-directory-list

The `$library-directory-list` variable is set by reroot by mapping the function make-absolute-filename across the `$relative-library-directory-list` variable which is not yet set so this is probably a bug.

53.2.32 $msgDatabaseName

The `$msgDatabaseName` is set to NIL in reroot.

53.2.33 $ncMsgList

The `$ncMsgList` is set to NIL in SpadInterpretStream.

53.2.34 $newcompErrorCount

The `$newcompErrorCount` is set to 0 in SpadInterpretStream.

53.2.35 $newcompMode

The `$newcompMode` is set to NIL in SpadInterpretStream.

53.2.36 $newspad

The `$newspad` is set to T in ncTopLevel.

53.2.37 $nopos

The `$nopos` variable is used in SpadInterpretStream but does not appear to have a value and is likely a bug.

53.2.38 $oldHistoryFileName

The `$oldHistoryFileName` is set at load time by a call to initvars to a value of "last". It is part of the history mechanism. It is used in the function oldHistFileName and restoreHistory.

53.2.39 $okToExecuteMachineCode

The `$okToExecuteMachineCode` is set to T in SpadInterpretStream.

53.2.40 $options

The `$options` variable is tested by the history function. If it is NIL then output the message

```
You have not used the correct syntax for the history command.
Issue )help history for more information.
```

The `$options` variable is tested in the historySpad2Cmd function. It appears to record the options that were given to a spad command on the input line. The function selectOptionLC appears to take a list off options to scan.

This variable is not yet set and is probably a bug.

53.2.41 $previousBindings

The `$previousBindings` is a copy of the `CAAR $InteractiveFrame`. This is used to compute the delta(i)s stored in `$frameRecord`. This is part of the undo mechanism.

53.2.42 printLoadMsgs (printLoadMsgs)

The `$printLoadMsgs` variable is set to T in restart.

53.2.43 $PrintCompilerMessageIfTrue

The `$PrintCompilerMessageIfTrue` variable is set to NIL in spad.

53.2.44 $openServerIfTrue

The `$openServerIfTrue` is tested in restart before it has been set (and is thus a bug). It appears to control whether the interpreter will be used as an open server, probably for OpenMath use.

If an open server is not requested then this variable to NIL

53.2.45 $relative-directory-list

The `$relative-directory-list` is used in reroot to create `$directory-list` which is a list of absolute directory names. It is not yet set and is probably a bug.

53.2.46 $relative-library-directory-list

The `$relative-library-directory-list` is used in reroot to create a list of absolute directory names from `$library-directory-list` (which is It is not yet set and is probably a bug).

53.2.47 $reportUndo

The `$reportUndo` variable is used in diffAlist. It was not normally bound but has been set to T in initvars. If the variable is set to T then we call reportUndo.

It is part of the undo mechanism.

53.2.48 $spadroot

The `$spadroot` variable is the internal name for the AXIOM shell variable.

The `$spadroot` variable is set in reroot to the value of the argument. The argument is expected to be a directory name.

The `$spadroot` variable is tested in initroot.

The `$spadroot` variable is used by the function make-absolute-filename. It concatenates this variable to the front of a relative pathname to make it absolute.

53.2.49 $spad

The `$spad` variable is set to T in ncTopLevel.

53.2.50 $SpadServer

If an open server is not requested then this variable to T. It has no value before this time (and is thus a bug).

53.2.51 $SpadServerName

The `$SpadServerName` is passed to the openServer function, if the function exists.

53.2.52 $systemCommandFunction

The `$systemCommandFunction` is set in SpadInterpretStream to point to the function InterpExecuteSpadSystemCommand.

53.2.53 top_level

The top_level symbol is a catch tag used in runspad to catch an exit from ncTopLevel.

53.2.54 $quitTag

The `$quitTag` is used as a variable in a catch block. It appears that it can be thrown somewhere below ncTopLevel.

53.2.55 $useInternalHistoryTable

The `$useInternalHistoryTable` variable is set at load time by a call to initvars to a value of NIL. It is part of the history mechanism.

53.2.56 $undoFlag

The `$undoFlag` is used in recordFrame to decide whether to do undo recording. It is initially set to T in initvars. This is part of the undo mechanism.

Bibliography

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

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

Chapter 54

Index

Index