$SPAD/src/emacs/axiommode.el
Version 16-10-2007

Jay Belanger, François Maltey, Martin Rubey and Cliff Yapp

February 25, 2018

Abstract

Emacs is one of the most popular environments for developing and using software. Among its many abilities it can interface with external programs and provide intelligent syntax highlighting for a wide variety of languages. This work is an attempt to make use of a variety of popular Emacs packages and define an Emacs environment tuned specifically to assist Axiom users and authors.

Contents

1 Copyright and License **3**

2 Introduction - Emacs and Axiom **4**

3 Tutorial **4**

4 Structuring the File **6**

5 How can we know that Axiom has finished a computation? **7**

6 Utility Definitions **8**

7 Filtering the output **12**

8 Starting Axiom **12**

9 Terminal Mode **13**

9.1 Axiom-Mode 13
9.2 Saving a session 16
9.3 Moving in Axiom Mode 19
9.3.1 Restricting Movements and Deletions 19
9.4 Reset 23
9.5 Selecting Text for Copying 25
9.6 Modifying old input 26
9.7 Annotating output 28
9.8 Coloring output 29
9.9 Overwriting Old Evaluation Blocks 32
9.10 Command completion 37
9.11 Inserting definitions from the kill ring 41
9.12 Restarting and Re-evaluating - Kill and Restart Axiom without Erasing the Document 42
9.13 Axiom Specific Menus 42
9.14 Assembling the Pieces 42

1 Copyright and License

Unlike the vast majority of Axiom, which is licensed under the Modified BSD license, this Emacs mode is licensed under the GPL. This is primarily due to the very large collection of additional Emacs modes that are used to make this environment as powerful as it is.

⟨*copyright*⟩≡

```
;; Copyright (C) 1995, 2006, 2007 by Jay Belanger, Francois Maltey, Martin
;; Rubey and Cliff Yapp
```

⟨*license*⟩≡

```
;; This program is free software; you can redistribute it and/or
;; modify it under the terms of the GNU General Public License as
;; published by the Free Software Foundation; either version 2 of
;; the License, or (at your option) any later version.
;;
;; This program is distributed in the hope that it will be
;; useful, but WITHOUT ANY WARRANTY; without even the implied
;; warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
;; PURPOSE.  See the GNU General Public License for more details.
;;
;; You should have received a copy of the GNU General Public
;; License along with this program; if not, write to the Free
;; Software Foundation, Inc., 59 Temple Place, Suite 330, Boston,
;; MA 02111-1307 USA
```

2 Introduction - Emacs and Axiom

There are a wide variety of Emacs modes available on the World Wide Web, which provide a vast array of features for virtually any task imaginable. Among those are several which can be applied to the challenges of working with Axiom:

- **mmm-mode** - This mode allows the use of different Emacs "major-modes" in a single text file. Basically, it allows Emacs to provide tailored environments for different regions of code in a single document.
- **AucTeX** - AucTeX is the state of the art LaTeX editing environment for LaTeX documents.
- **EMaxima** - This is a highly sophisticated Emacs environment created for the Maxima computer algebra system.

There are also some existing Emacs modes for Axiom, providing a number of abilities which will serve as the foundation of this document.

Looking at these modes and the needs of Axiom, we can draw up a feature list of abilities we would like to see in an Axiom Emacs mode:

1. Console Axiom sessions in an Emacs buffer. A basic ability, but a good fallback to have for situations which don't allow for elaborate formatting.
2. LaTeX document editing with Axiom IO sessions as formatted parts of the document. EMaxima provides this functionality for Maxima.
3. Intelligent handling of code environments found in Axiom, including in particular Lisp, SPAD, and Aldor code.
4. Supporting any of the above in a pamphlet aware environment, including intelligent syntax highlighting on a per-chunk basis.

3 Tutorial

In this section we describe very briefly how to use this package. The following emacs commands are provided:

axiom starts an axiom session or returns to an already started one. We really should provide functionality to have several sessions in parallel, but I don't know how to do this.

M-up, M-down moves the cursor to the previous or next input.

C-up/M-p, C-down/M-n fetches the previous or next input.

M-k copies the current input-output combination into the kill-ring.

S-up, S-down, S-left, S-right turns the cursor into a paint-brush.

axiom-paint-face changes the face of the paint-brush.

return evaluates input. If point is on a previous input, it overwrites old output "nicely".

M-return evaluates input and appends output at the end of the buffer.

C-return writes the front item of the kill ring into a temporary file, and then `)read`s that file. With an argument, it reads the file [)quiet]ly.

tab completes file names after a system command like `)read`.

C-x s saves the worksheet. If you give it extension `.axiom` and put

```
(setq auto-mode-alist (cons '("\\.axiom$" . axiom-mode) auto-mode-alist))
```

in your `.emacs`, files will be automatically restored when you visit them.

There remain a few things to do:

- most importantly, overwriting old output and even [axiom-reset] are still not robust enough. `axiom-yank] sometimes complains that the file does not exist. [[kill-line` should not kill the newline at the end of an input region.
- I would like to inhibit the read only feature at times, but in a controlled way, for example, only for the current output region.
- clean up the initialisation code following the elisp reference manual `Major Mode Conventions`.
- enable `)undo`, binding it to `C-_`, possibly as follows:
  - locally, undo input,
  - if at the last prompt, and there is no input, `C-_` should issue an undo and delete the last input output combination. This probably means that we need to track the prompt, doesn't it?
- I should rethink usage of text-properties and overlays. Unfortunately, overlays always take precedence over text-properties, so painting will not work if output were marked with an overlay - as would be logical. I could, of course, make painting modify at the same time the overlay as well as the text-property.
- I'd really like to have an xemacs port for Ralf. This would also make the code more robust, I guess.
- I should look into [comint-redirect-*].

4 Structuring the File

$\langle * \rangle \equiv$

$\langle$*copyright*$\rangle$
$\langle$*license*$\rangle$
$\langle$*requires*$\rangle$
$\langle$*AxiomStartupVariables*$\rangle$
$\langle$*UtilityVariables*$\rangle$
$\langle$*UtilityFunctions*$\rangle$
$\langle$*StartingAxiom*$\rangle$
$\langle$*TerminalMode*$\rangle$

5 How can we know that Axiom has finished a computation?

Contrary to shell mode or ansi-term, axiom mode has to know when axiom is ready for input, since we want to make output and prompt read-only, and have some cursor-positioning features. (Although, really, this assumption should be checked again and again!)

To guess whether Axiom is ready, there are several strategies which we probably need to combine:

1. Wait a little, namely `axiom-after-output-wait` miliseconds.
2. Look for the prompt, which satisfies the regular expression `axiom-prompt`.
3. Track axiom system commands, which may require user input, and which may produce output satisfying `axiom-prompt`, in the variable `axiom-system-command`.
4. Track the position of `process-mark`.

Here are two tests that illustrate what can go wrong and what can't:

```
for i in 1..5 repeat (output "(1) -> "; [j for j in 1..10000])
```

Although this computation takes some time, it does not produce output that matches `axiom-prompt`, since `OutputPackage` indents all the output it produces. Of course, we should keep in mind that other packages may not do this: `print("(2) -> ")$Lisp`.

Furthermore, one can always fall back to lisp:

```
)lisp (dotimes (i 5) (sleep 1) (format t "(1) -> ~%"))
```

Finally, we currently assume that only system commands like

```
)di op 1
```

require input without producing a prompt matching `axiom-prompt`. Unfortunately, this is not quite true:

```
)set break break
```

will make axiom enter the lisp debugger on error, which has a different prompt. We really should track this, too.

To minimze the likelihood that the mode gets fooled into believing Axiom is ready for new input, we can watch the `process-mark`. This is properly moved as long as Axiom outputs something. Thus, we really should allow input only if

- we are answering a system command, or
- the `process-mark` is just after a prompt.

Furthermore, repositioning the `process-mark` using `M-p`, `M-up`, [M-down]] or the like, should only be allowed when it is after a prompt.

6 Utility Definitions

The Axiom process will be run in a buffer using Emacs's comint (command-interpreter) mode.

⟨*requires*⟩≡
```
(require 'comint)
```

As in any program, a variety of variables and functions are defined for basic but necessary jobs. First, we will define the name of the axiom program and the arguments to be provided:

⟨*AxiomStartupVariables*⟩≡
```
;; Variables used when starting Axiom
(defvar axiom-command "")  ;; There is a function to determine this value, but it need
                           ;; to be defined before being set.
(defvar axiom-args "-noclef")  ;; Ignored when AXIOMsys is used.
(defvar axiom-prompt "^(\\([0-9]+\\)) -> ")
(defvar axiom-after-output-wait 100) ;; time to wait for axiom to repond prompt
```

⟨*axiom-mode-map*⟩≡
```
(defvar axiom-mode-map (copy-keymap comint-mode-map)
  "local key map for Axiom terminal mode")
(define-key axiom-mode-map [(meta k)] 'axiom-copy-to-clipboard)
(define-key axiom-mode-map [(ctrl return)] 'axiom-yank)
(define-key axiom-mode-map [(meta up)] 'axiom-previous-input)
(define-key axiom-mode-map (kbd "ESC <up>") 'axiom-previous-input)
(define-key axiom-mode-map [(meta down)] 'axiom-next-input)
(define-key axiom-mode-map (kbd "ESC <down>") 'axiom-next-input)
(define-key axiom-mode-map [left] 'axiom-backward-char)
(define-key axiom-mode-map [right] 'axiom-forward-char)
(define-key axiom-mode-map [(shift up)] 'axiom-paint-previous-line)
(define-key axiom-mode-map [(shift down)] 'axiom-paint-next-line)
(define-key axiom-mode-map [(shift left)] 'axiom-paint-previous-char)
(define-key axiom-mode-map [(shift right)] 'axiom-paint-next-char)
(define-key axiom-mode-map "\C-c\C-t" 'axiom-toggle-annotate)
;; doesn't work in terminal
;; (define-key axiom-mode-map [return] 'axiom-eval)
(define-key axiom-mode-map "\C-m" 'axiom-eval)
;; doesn't work in terminal
;;(define-key axiom-mode-map [(meta return)] 'axiom-eval-append)
(define-key axiom-mode-map "\M-\C-m" 'axiom-eval-append)
(define-key axiom-mode-map "\t" 'axiom-dynamic-complete)
```

I had `(define-key axiom-mode-map [(tab)] 'axiom-dynamic-complete)` here before, but that did not work entirely: hitting tab twice inserted a tab!

In cases where we are overriding comint key bindings, we apparently need to do this after `use-local-map`, which is issued in `axiom-run`. However, I do not quite understand why we do that, in fact.

These settings will be universal no matter which mode Axiom is in. The first line define the name of the command to run Axiom, The second line provides the options to disable the graphical interface and clef intelligent command line utility. The third line supplies the REGEXP syntax to identify the Axiom prompts.

We also need to initialize some variables for later use:

⟨*UtilityVariables*⟩≡

```
;; Utility variables
(defcustom axiom-mode-hook '()
  "*Hook for customising Axiom mode."
  :type 'hook
  :group 'axiom)

(defvar axiom-system-command nil)
(defvar axiom-end-of-input   0)
(defvar axiom-current-input  "")
(defvar axiom-paint-face     'axiom-paint-lightblue)
```

and define convenience functions:

⟨*UtilityFunctions*⟩≡

```
;;due to William G. Dubuque 1994
(defun looking-backward-at (regexp &optional bound)
  (let ((point (point)))
    (save-restriction
      (narrow-to-region (or bound (point-min)) point)
      ;; Recall that \' matches end of buffer in a regexp.
      ;; Todo: bug: regexp compiler does not appear to optimize this well
      ;; (seems like test (eobp) is done after each match instead of anchoring
      ;; at eob), so use a bound to improve efficiency.
      (prog1 (re-search-backward (concat regexp "\\'") bound t)
             (goto-char point)))))

; from emacs 22
(defun make-temp-file (prefix &optional dir-flag suffix)
  "Create a temporary file.
The returned file name (created by appending some random characters at the end
of PREFIX, and expanding against `temporary-file-directory' if necessary),
is guaranteed to point to a newly created empty file.
You can then use `write-region' to write new data into the file.

If DIR-FLAG is non-nil, create a new empty directory instead of a file.

If SUFFIX is non-nil, add that at the end of the file name."
  (let ((umask (default-file-modes))
        file)
    (unwind-protect
        (progn
          ;; Create temp files with strict access rights.  It's easy to
          ;; loosen them later, whereas it's impossible to close the
          ;; time-window of loose permissions otherwise.
          (set-default-file-modes ?\700)
          (while (condition-case ()
                     (progn
                       (setq file
                             (make-temp-name
                              (expand-file-name prefix temporary-file-directory)))
                       (if suffix
                           (setq file (concat file suffix)))
                       (if dir-flag
                           (make-directory file)
                         (write-region "" nil file nil 'silent nil 'excl))
                       nil)
                   (file-already-exists t))
```

```
            ;; the file was somehow created by someone else between
            ;; `make-temp-name' and `write-region', let's try again.
            nil)
          file)
      ;; Reset the umask.
      (set-default-file-modes umask))))

(defun axiom-output? (position)
  "Non-nil if position is in the output region"
  (get-text-property position 'axiom-output))

(defun axiom-prompt? (position)
  "Non-nil if position is in the prompt region"
  (get-text-property position 'axiom-prompt))
```

We use the text-properties `axiom-prompt` and `axiom-output` to mark prompt and output. `make-output` is called by `axiom-output-filter` which in turn is hooked to `comint-output-filter-functions` and thus is called whenever we receive output from axiom.

⟨*UtilityFunctions*⟩+≡

```
(defun axiom-make-prompt (begin end)
  "Makes the region a prompt.  In particular, it gets the field property, the
right face, and is made read-only"
  (let ((inhibit-read-only t))
    (put-text-property begin end 'axiom-prompt t)
    (remove-text-properties begin end '(axiom-output nil))

    (put-text-property begin end 'face 'comint-highlight-prompt)
    (put-text-property begin end 'field t)
    (put-text-property begin end 'front-sticky t)
    (put-text-property begin end 'rear-nonsticky t)
    (put-text-property begin end 'read-only t)))

(defun axiom-make-output (begin end)
  (let ((inhibit-read-only t))
    (put-text-property begin end 'axiom-output t)
    (put-text-property begin end 'face 'axiom-output)
    (put-text-property begin end 'front-sticky t)  ; otherwise I can insert text
    (put-text-property begin end 'rear-nonsticky t); otherwise I cannot append text
    (put-text-property begin end 'read-only t)))
```

7 Filtering the output

⟨*UtilityFunctions*⟩+≡
```
(defun axiom-output-filter (str)
  "Look for a new input prompt."
  (let ((comint-last-output-end
         (process-mark (get-buffer-process (current-buffer))))
        (prompt-start
         (string-match axiom-prompt str)))
    (axiom-make-output (1+ axiom-end-of-input)
                       (if prompt-start
                           (+ comint-last-output-start
                              prompt-start)
                         comint-last-output-end))
    (when prompt-start
      (axiom-make-prompt (+ comint-last-output-start
                            prompt-start)
                         comint-last-output-end))))

(defun axiom-wait-for-output ()
  "Wait for output from the Axiom process."
  (sit-for 0 axiom-after-output-wait))
```

8 Starting Axiom

Axiom can be started using either of two commands - "axiom", which enables extra functionality like graphing, and "AXIOMsys" which is a basic Axiom session only. On Windows (as of 5/3/2006) only AXIOMsys is available, so we want to be able to automatically spot situations where we only have AXIOMsys. Note that we don't need to know the exact path, but we do need to know if "axiom" or "AXIOMsys" is to be used. To that end, we define the following function:

⟨*StartingAxiom*⟩≡
```
(defun axiom-get-command ()
   "Searches the local system PATH variable for the axiom binary"
   (let ((axiom-localpaths exec-path))
     (while axiom-localpaths
       (when (file-executable-p (concat (car axiom-localpaths) "/axiom"))
         (setq axiom-localpaths nil)
         (setq axiom-command "axiom"))
       (setq axiom-localpaths (cdr axiom-localpaths)))
     ;; If we come up empty, default to AXIOMsys
     (unless (equal axiom-command "axiom")
       (setq axiom-command "AXIOMsys"))))
```

I really would like to have a behaviour like that of shell mode, i.e., `M-x axiom` should switch to the buffer `*axiom*`, if it exists and if it is an "axiom buffer". (I guess that should be a buffer in `axiom-mode`, with a running axiom process.)

If there is no such buffer, it should create a new buffer and a new axiom process.

`(add-hook 'axiom-mode-hook (lambda () (rename-uniquely)))` should rename the buffer. I do not know what else one could use `shell-mode-hook` (and therefore `axiom-mode-hook`) for.

9 Terminal Mode

The most basic mode of interaction for any CAS is the command prompt, a.k.a terminal mode. We will call this mode "axiom-mode" here, since this is likely to be the most common interaction mode for casual use. Even though this environment is text based, we would still like to provide a more sophisticated environment for the user than simply showing the Axiom text IO in a window. Among the features we would like to target for this mode:

- We would prefer to only allow the cursor to move to valid input positions. This does a number of things, not the least of which is prevent any damaging of the text structure Emacs will be using to provide intelligent navigation and IO positioning.
- "Notebook" style evaluation of inputs - that is, moving to a previous input line and overwriting the old entry with a new one - are also useful from a visual flow standpoint.
- In the various input positions, we would like a history of previous inputs to be available, and to have Emacs evaluate them in place. This would result in overwriting any old answer that was there and the shifting any subsequent answers up or down as the case may be.
- In terms of saving sessions, we would like to have available the ability to save as text, LaTeX, and possibly selective saves such as inputs only. Ideally we could also have a save mode which would save a session file in such a way as to allow us to reload, re-evaluate, and carry on at some later date.

9.1 Axiom-Mode

This function is run only when issuing `M-x axiom`. Very likely, as soon as I allow visiting saved history, this needs to change.

⟨*StartingAxiom*⟩+≡
```
(defun axiom-run()
  "Run Axiom in a buffer."
```

```
  ;; Get the command to use
  (axiom-get-command)
  ;; Run that command and switch to the new buffer
  (switch-to-buffer (make-comint "axiom" axiom-command nil axiom-args)))
```

⟨*StartingAxiom*⟩+≡

```
(defun axiom-wait-start (proc before-banner delete-banner)
  "Wait until the prompts have appeared.  Find the end of the banner after
 before-banner. If delete-banner is nil, make the banner write-protected,
 otherwise delete it."
  (comint-goto-process-mark)
  ;; wait until the prompt(s) have appeared
  (cond ((equal axiom-command "axiom")
         (while (not (re-search-backward "(1) -> [^ ](1) ->" before-banner t))
           (accept-process-output proc)))
        ((equal axiom-command "AXIOMsys")
         (while (not (re-search-backward "(1) -> " before-banner t))
           (accept-process-output proc))))

  (if delete-banner
      (delete-region before-banner (point))
    (progn
      (remove-text-properties before-banner (point) '(face nil rear-nonsticky nil))
      (put-text-property before-banner (point) 'front-sticky t)
      (put-text-property before-banner (point) 'read-only t)))

  (goto-char before-banner)
  (search-forward "(1) ->")
  (search-backward "(1) ->")

  (let ((inhibit-read-only t)
        (point-after-last-message nil)
        (point-before-current-marker nil))

    (setq point-after-last-message (point-marker))
    (goto-char (point-max))
    (re-search-backward "([0-9]+) -> ")
    (setq point-before-current-marker (point-marker))
    (unless (eq point-after-last-message point-before-current-marker)
      (delete-region point-after-last-message point-before-current-marker))
    (goto-char (point-max))
  ;; We need to explicitly write protect the first prompt, since it
  ;; is outside the normal write protect mode used for subsequent
  ;; output:
    (axiom-make-prompt (- (point) 7) (point))))
```

⟨*defineaxiom-mode*⟩≡

```
(put 'axiom-mode 'mode-class 'special)

(define-derived-mode axiom-mode comint-mode "AXIOM"
  ""
  (let ((proc (get-buffer-process (current-buffer))))
    (if proc
        (axiom-wait-start proc (point-min) nil)
      (let ((before-banner (point-max)))
        ;; if proc is nil,  we assume that we have visited a file.
        ;; I guess I should make sure that axiom is running otherwise, but I do
        ;; not know how.
        (axiom-parse-region)
        (axiom-get-command)
        ;; inserst a newline so that the banner is easier to find
        (goto-char before-banner)
        (insert "\n")
        ;; Run that command
        (make-comint-in-buffer "axiom"
                               (current-buffer)
                               axiom-command nil axiom-args)
        (axiom-wait-start (get-buffer-process (current-buffer))
                          (1+ before-banner) t)))

    (make-local-variable 'axiom-system-command)
    (setq axiom-system-command nil)
    (make-local-variable 'axiom-end-of-input)
    (setq axiom-end-of-input 0)
    (make-local-variable 'axiom-current-input)
    (setq axiom-current-input "")
    (make-local-variable 'axiom-paint-face)
    (setq axiom-paint-face 'axiom-paint-lightblue)

    ;; we want this hook to be local lest we conflict with other buffers in comint
    ;; or shell mode
    (add-hook 'comint-output-filter-functions 'axiom-output-filter nil t)

    (setq buffer-offer-save t)
    (add-hook 'after-save-hook 'axiom-save-history nil t)
    (add-hook 'kill-buffer-query-functions 'axiom-query-kill)

    ;; Next, we turn on some key bindings for our new mode:
    (use-local-map axiom-mode-map)
    (substitute-key-definition 'comint-previous-input
                               'axiom-scroll-previous-input axiom-mode-map)
    (substitute-key-definition 'comint-next-input
```

```
                                'axiom-scroll-next-input axiom-mode-map)
    ;; HyperDoc currently sends loading messages to the buffer.  Because of the
    ;; current setup, they are going to be read-only, and they are not followed
    ;; by a prompt.  Thus, lest we cannot append any further input, we have to
    ;; mute them.  Currently this is only possible via )set messages autoload off
    (insert ")se me au of")
    (axiom-normal-eval-wait)
    (unless proc
      (axiom-wait-for-output)
      (insert (concat ")history )restore " (buffer-file-name)))
      (axiom-normal-eval-wait))
    (set-buffer-modified-p nil)))

(defun axiom ()
   "Run axiom in a terminal environment"
  (interactive)
  (if (not (comint-check-proc "*axiom*"))
      (progn (axiom-run)
             (axiom-mode))
    (pop-to-buffer "*axiom*")))

(provide 'axiom)
```

9.2 Saving a session

Axiom provides `)history )save` and `)history )restore` to save and restore a session. Additionally, we save the text of the buffer, so after reloading it looks and behaves really the same.

⟨*axiom-maybe-save*⟩≡

```
(defun axiom-query-kill ()
  (if (eq major-mode 'axiom-mode)
      (or (not (buffer-modified-p))
          (yes-or-no-p (format "Buffer %s modified; kill anyway? "
                               (buffer-name))))
    t))
```

This is going to be called just before the buffer is saved into the filename supplied by the user. Thus, the following files will be created:

- `foo`
- `foo.input`
- `foo.axh/index.KAF`

I'd rather have everything in one directory, but maybe that's not worth the effort. Visiting `foo` should make emacs restore the history. Maybe I should rather call it `foo.axiom`, to enable auto-detection of the major-mode.

⟨*axiom-maybe-save*⟩+≡
```
(defun axiom-save-history ()
  "If necessary, removes previous saved histories, since it seems that axiom
does not store the %% facility correctly.  Then issues )history )save."
  (comint-goto-process-mark)
  (when (file-exists-p (concat (buffer-file-name) ".axh"))
    (delete-file (concat (buffer-file-name) ".axh/index.KAF"))
    (delete-directory (concat (buffer-file-name) ".axh")))

  (insert (concat ")history )save " (buffer-file-name)))
  (axiom-normal-eval-wait)
  (set-buffer-modified-p nil))

(defun axiom-make-input-parse (beg end)
  "Makes the region between beg and end an input region, as if it would have
been sent with comint-send-input."

  (let ((over (make-overlay beg end nil nil t)))
    (overlay-put over 'field 'input)
    (when comint-highlight-input
      (overlay-put over 'face 'comint-highlight-input)
      (overlay-put over 'mouse-face 'highlight)
      (overlay-put over
                   'help-echo
                   "mouse-2: insert after prompt as new input")
      (overlay-put over 'evaporate t)))
  (let ((over (make-overlay end (1+ end) nil t nil)))
    (overlay-put over 'field 'boundary)
    (overlay-put over 'inhibit-line-move-field-capture t)
    (overlay-put over 'evaporate t))
  (axiom-make-input (1- beg) (1+ end)))

(defun axiom-parse-region ()
  "Parses the saved file and puts appropriate text properties.  There remains a
glitch at the two commands in the middle:
```

```
)se me au off
)history )restore filename
which have incorrect field boundaries."

  (let ((inhibit-read-only t)
        (pos (point-min))
        prompt-begin prompt-end)
    (goto-char pos)
    (while (and (< pos (point-max))
                (setq prompt-end (re-search-forward axiom-prompt (point-max) "goto-end
      (setq prompt-begin (match-beginning 0))
      (axiom-make-output (1+ pos) prompt-begin)
      (axiom-make-prompt prompt-begin prompt-end)
      (axiom-make-input-parse prompt-end (setq pos (axiom-end-of-input))))))
```

9.3 Moving in Axiom Mode

We will need to restrict the legal positions of the cursor to input lines. Practically, this means the following will have to be accomplished:

- Restrict the left and right arrow keys to motion between the first whitespace after the `->` input arrow and the carriage return at the end of the line. Some systems allow multiple line input on the command prompt, but this requires the use of an "end of line" character and Axiom does not use this method. It is conceivable that an intelligent way to allow this could be coded but unless there is demand for it these limits will work.
- Restrict the up and down arrow keys to moving from one input line to the next, or no movement if no further prompt can be identified.
- Prevent the backspace and delete commands from removing either input prompts or carrage returns, since these will be the determining characters for beginning and end of input.

9.3.1 Restricting Movements and Deletions

Some care has to be taken to find the preceding prompt and the next, since, for example `)lisp (format t "(1) -> ")` produces output that matches `axiom-prompt`.

This function actually moves the cursor to a previous input line.

⟨*axiom-previous-input*⟩≡

```
(defun axiom-previous-prompt ()
  "Put point just after the previous prompt and return this position.  If
there is no previous prompt, point stays where it is end we return nil.  We use
the regular expression to find a prompt, rather than the text-property, for
axiom-reset. ???It seems that this is not quite true???"
  (let ((found?))
    (while (and (setq found? (re-search-backward axiom-prompt nil t))
                (not (axiom-prompt? (point)))))
    (when found?
; originally I used (end-of-line) to move to the end of the prompt, relying on
; the field property of the prompt.  But it seems that this does not always
; work.
      (re-search-forward axiom-prompt nil t)
      (point))))

(defun axiom-previous-input ()
  "If in input, puts point just after the prompt before the previous prompt and
return this position. If in output, or at the first input line, puts point just
after the previous prompt. Otherwise, the behaviour is undefined."
  (interactive)
  (let ((found?))
    (if (or (axiom-output? (point))
```

```
          (axiom-prompt? (point)))
      (axiom-previous-prompt)
    (while (and (re-search-backward axiom-prompt nil t)
                (not (axiom-prompt? (point)))))
    (while (and (setq found? (re-search-backward axiom-prompt nil t))
                (not (axiom-prompt? (point)))))
    (end-of-line)
    (when found? (point)))))
```

This function will move the cursor to the next identified input line.

⟨*axiom-next-input*⟩≡

```
(defun axiom-next-input ()
  "Puts point just after the next prompt and return this position.  If there
is no next prompt, point stays where it is end we return nil."
  (interactive)
  (let ((found?))
    (while (and (setq found? (re-search-forward axiom-prompt nil t))
                (not (axiom-prompt? (1- (point))))))
    found?))

(defun axiom-next-prompt ()
  "Puts point just before the next prompt and return this position.  If there
is no next prompt, point stays where it is end we return nil."
  (when (axiom-next-input) (re-search-backward axiom-prompt nil t)))
```

Fetch the previous input.

⟨*axiom-scroll-input*⟩≡
```
(defun axiom-end-of-input ()
  "Puts point at the end of the last input line and returns that position."
  (while (and (not (axiom-output? (point))) (looking-at ".*_ *$"))
    (next-line 1)
    (beginning-of-line))
  (end-of-line)
  (point))

(defun axiom-scroll-previous-input (&optional arg)
  "Fetch the previous input, if in input region."
  (interactive "p")
  (unless (or (axiom-prompt? (point))
              (axiom-output? (point)))
    (axiom-previous-prompt)
    (comint-set-process-mark)
    (comint-previous-input arg)
    ;; delete the rest
    (delete-region (point) (axiom-end-of-input))))

(defun axiom-scroll-next-input (&optional arg)
  "Move to the next input line, if in input region."
  (interactive "p")
  (unless (or (axiom-prompt? (point))
              (axiom-output? (point)))
    (axiom-previous-prompt)
    (comint-set-process-mark)
    (comint-next-input arg)
    ;; delete the rest
    (delete-region (point) (axiom-end-of-input))))
```

This function will permit the cursor to move one character back if it does not result in the cursor entering an input prompt.

⟨*axiom-backward-char*⟩≡
```
(defun axiom-backward-char (&optional arg)
  "Move left only if we stay in the input region."
  (interactive "p")
  (dotimes (i arg)
    (if (axiom-prompt? (1- (point)))
        (when (axiom-previous-input)
          (axiom-end-of-input))
      (backward-char))))
```

This function will permit the cursor to move one character to the right only if it does not result in the cursor going inside a prompt - otherwise, it will skip ahead to the input location.

⟨*axiom-forward-char*⟩≡
```
(defun axiom-forward-char (&optional arg)
  "Move right depending on the region we are in."
  (interactive "p")
  (dotimes (i arg)
    (cond ((eobp)) ; at the end of the buffer we signal an error
          ((looking-at axiom-prompt)
           (re-search-forward axiom-prompt))
          ((and (not (axiom-output? (point)))
                (axiom-output? (1+ (point))))
           (axiom-next-input))
          (t (forward-char)))))
```

9.4 Reset

Unfortunately, at times other processes happen to write into the axiom buffer, without appending a prompt. Then the space behind the last prompt becomes read-only, and the buffer becomes awkward to use. The following function provides a workaround.

⟨*axiom-reset*⟩≡

```
(defun axiom-resync-directory ()
  "Go to the process mark. Send )sys pwd to the axiom process, parse the
 result, cd to it and clean up output.  FIXME: Doesn't seem to work nicely
 after undo..."
  (interactive)
  (comint-goto-process-mark)
  (let ((inhibit-read-only t)
        (begin (point))
        end dir)
    (process-send-string (get-buffer-process (current-buffer)) ")sys pwd\n")
    (axiom-wait-for-output)
    (forward-line -1)
    (end-of-line)
    (setq end (point))
    (cd (buffer-substring-no-properties begin end))
    (forward-line 0)
    (delete-region (1- (point)) end)
    (comint-goto-process-mark)))


;  (let* ((inhibit-read-only t)
;         (begin (progn (forward-line 0) (point)))
;         (end (progn (end-of-line) (point)))
;         (dir))
;    (delete-region begin end)
;    (process-send-string (get-buffer-process (current-buffer)) ")sys pwd\n")
;    (axiom-wait-for-output)
;    (forward-line -1)
;    (end-of-line)
;    (sit-for 0)
;    (setq end (point))
;    (cd (buffer-substring-no-properties begin end))
;    (delete-region (1- begin) end)
;    (forward-line 1)
;    (end-of-line)))

;  (let ((inhibit-read-only t)
;        (inhibit-field-text-motion t)
;        dir begin end)
```

```
;    (delete-region (progn (forward-line 0) (point))
;                   (progn (end-of-line) (point)))
;    (process-send-string (get-buffer-process (current-buffer)) ")sys pwd\n")
;    (axiom-wait-for-output)
;    (forward-line -1)
;    (setq begin (point))
;    (end-of-line)
;    (sit-for 0)
;    (setq end (point))
;    (cd (buffer-substring-no-properties begin end))
;    (delete-region (1- begin) end)
;    (forward-line 1)
;    (end-of-line)))


(defun axiom-reset ()
   "Remove read-only properties from everything after the last prompt. Set
process-mark so we can continue.  Resync directory."
   (interactive)
   (let ((inhibit-read-only t))
     (goto-char (point-max))
     (delete-region (axiom-previous-prompt) (point-max))
     (comint-set-process-mark)
     (axiom-make-prompt (re-search-backward axiom-prompt nil t) (point))
     (comint-goto-process-mark)
     ; do not use copy-marker here, because we don't want this position to
     ; move.  Otherwise axiom-resync-directory will fail in axiom-filter-output
     (setq axiom-end-of-input (point)))
   (axiom-resync-directory))
```

9.5 Selecting Text for Copying

The following provides a possibility for copying the current input-output pair into the kill-ring. Note that color is preserved, if we yank it into a non-`font-lock-mode` buffer. Using `enriched-mode` we can even save it to a file.

This function will copy the current input-output combination, if the cursor is within a complete input-output pair, or copy just the current input if there is no output yet.

It should actually take an argument, specifying how many pairs should be copied.

⟨*axiom-text-selection*⟩≡

```
(defun axiom-copy-to-clipboard (&optional arg)
  "Copy the arg previous input-output combinations into the kill-ring."
  (interactive "p")
  (save-excursion
    (let* ((end (or (axiom-next-prompt) (point-max)))
           (n arg)
           (begin (progn (while (< 0 n)
                           (axiom-previous-prompt)
                           (if (re-search-backward axiom-prompt nil t)
                               (setq n (1- n))
                             (setq n 0)
                             (goto-char (point-min))))
                         (point))))
      (clipboard-kill-ring-save begin end))))
```

9.6 Modifying old input

Because we allow the user to enter previous expressions and change them, we need a way to visually flag a changed-but-unevaluated historical input in order to avoid any confusion. There are a number of possible visual cues that can be used for this process - the important thing to start with is determing how we will know when a line has changed, when it has been evaluated, and how to handle both situations.

Emacs provides a text property called modification-hooks which allows the programmer to call one or more functions on alteration of that text. This property will be used in two situations - once after an evaluation to "flag" an input and assign the proper input text properties, and again if that input is changed subsequently. In that case, all pre-existing overlays are removed and a flagging overlay is added - red text color, for example.

First, we will define our formatting for an altered input:

⟨*axiom-changed-prompt-face*⟩≡

```
(defface axiom-changed-input '((t (:foreground "red")))
  "Face to use to highlight modified input."
  :group 'axiom)
```

Next we create a convenience function to clear old overlays. This is to avoid unanticipated conflicts between old overlays and new ones.

⟨*axiom-clear-text-properties*⟩≡

```
(defun axiom-clear-overlays (pos)
   "Clears all text properties at pos"
   (while (overlays-at pos)
     (delete-overlay (car (overlays-at pos)))))
```

When a change actually occurs, we just change the face of the existing overlay to `axiom-changed-input`, and remove the modification hooks call to this function to prevent recursion.

⟨*axiom-changed-function*⟩≡

```
(defun axiom-flag-as-changed (overlay after-change begin end &optional len)
  "Handles updating the input prompt in question after a change.  We assume
that point is between begin and end."
  (let ((new-input (buffer-substring begin end)))
    (if after-change
        (unless (string= axiom-current-input new-input)
          ;; Change the look of the text
          (overlay-put overlay 'face 'axiom-changed-input)
          ;; And remove this function from the modification hooks list
          (overlay-put overlay 'modification-hooks '())
          ;; Mark all prompts below to indicate that they have to be taken with
          ;; care
          (let ((position end)
                (max (point-max))
                (inhibit-read-only t))
            (while (not (= position max))
              (if (axiom-prompt? position)
                  (let ((begin position))
                    (while (and (axiom-prompt? position)
                                (< position max))
                      (setq position (1+ position)))
                    (let ((overlay (make-overlay begin (- position 4))))
                      (overlay-put overlay 'face 'axiom-changed-input)))
                (setq position (1+ position))))
            (sit-for 0)))
      (setq axiom-current-input new-input))))
```

⟨*axiom-flag-prompt*⟩≡

```
(defun axiom-make-input (beg end)
  "Call this when a new evaluation begins. It sets the 'modification-hooks
 property for the input after the current prompt.  It sets point at the end of
 the current input.

  begin should be the position just before input starts, end the position just
 after input."
  (let ((over (make-overlay beg end nil nil t)))
    (overlay-put over 'modification-hooks '(axiom-flag-as-changed))))
```

9.7 Annotating output

Usually, output is read-only. However, sometimes we would like to add some annotations, ideally only temporarily and only within the current output region.

⟨*axiom-annotate*⟩≡

```
(defun axiom-toggle-annotate ()
  (interactive)
  (setq inhibit-read-only (not inhibit-read-only)))
```

9.8 Coloring output

When demonstrating calculations to other people or when trying to focus oneselves attention to certain bits of the output, it may come useful to permanently mark those bits. In this section we provide rudimentary support. We use the shift key as modifier to turn the cursor into a paint-brush.

Currently, we do not provide any means to modify the face the brush paints with. Also, it might be desirable to modify the behaviour of the brush such that painting the same location a second time with the same face removes the paint. Finally, it is not clear whether painting of the input is desirable.

painting is now (more or less) confined to the output region, and copying output into another buffer preserves it. In particular, using enriched-mode, one can save the painted stuff to a file. (Don't forget to switch off font-lock-mode!)

⟨*axiom-paint*⟩≡

```
(defface axiom-output '((t (:background "lightgreen")))
  "Face used for output."
  :group 'axiom)

(defface axiom-paint-lightblue '((t (:background "lightblue")))
  "Lightblue face to use for painting."
  :group 'axiom)

(defface axiom-paint-red '((t (:background "red")))
  "Red face to use for painting."
  :group 'axiom)

(defface axiom-paint-custom '((t nil))
  "Custom face to use for painting."
  :group 'axiom)

(defvar axiom-paint-face-alist
  '(("lightblue" axiom-paint-lightblue)
    ("red"  axiom-paint-red)
    ("custom"  axiom-paint-custom)
    ("output"  axiom-output)))

(defun axiom-paint-face ()
  (interactive)
  (let ((newpaint (completing-read "New paint face: "
                                   axiom-paint-face-alist
                                   nil t)))
    (setq axiom-paint-face (cadr (assoc newpaint axiom-paint-face-alist)))))

(defun axiom-make-space-if-necessary-and-paint ()
  ;; The following is to make sure that a line does not end with a painted
  ;; character.  This would have the unwanted effect, that spaces appended by
```

```
  ;; either axiom-paint-previous-line or axiom-paint-next-line inherit the face
  ;; of the last character.
  (when (eolp)
    (insert-char 32 2 t)
    (backward-char 2))
  (forward-char 1)
  (when (eolp)
    (insert-char 32 1 t)
    (backward-char 1))
  (backward-char 1)

  (if (equal (get-text-property (point) 'face)
             axiom-paint-face)
      (if (axiom-output? (point))
          (put-text-property (point) (1+ (point)) 'face 'axiom-output)
        (remove-text-properties (point) (1+ (point)) '(face nil)))
    (put-text-property (point) (1+ (point)) 'face axiom-paint-face)))

(defun axiom-paint-previous-line ()
  (interactive)
  (when (axiom-output? (point))
    (let ((inhibit-read-only t)
          (old-column (current-column))
          (old-pos    (point)))
      (axiom-make-space-if-necessary-and-paint)
      (previous-line 1)
      (if (axiom-output? (point))
          (let ((difference (- old-column (current-column))))
            (when (> difference 0)
              (insert-char 32 difference t)))
        (goto-char old-pos)))))

(defun axiom-paint-next-line ()
  (interactive)
  (when (axiom-output? (point))
    (let ((inhibit-read-only t)
          (old-column (current-column))
          (old-pos    (point)))
      (axiom-make-space-if-necessary-and-paint)
      (next-line 1)
      (if (axiom-output? (point))
          (let ((difference (- old-column (current-column))))
            (when (> difference 0)
              (insert-char 32 difference t)))
        (goto-char old-pos)))))
```

```
(defun axiom-paint-previous-char ()
  (interactive)
  (when (axiom-output? (point))
    (let ((inhibit-read-only t))
      (axiom-make-space-if-necessary-and-paint)
      (when (axiom-output? (1- (point)))
        (backward-char 1)))))

(defun axiom-paint-next-char ()
  (interactive)
  (when (axiom-output? (point))
    (let ((inhibit-read-only t))
      (axiom-make-space-if-necessary-and-paint)
      (forward-char 1))))
```

9.9 Overwriting Old Evaluation Blocks

We would like to have the ability to evaluate new inputs and have the result overwrite an old one, if we execute the command on a previous line. Broken down, such an ability requires the following steps:

- Identifying and storing the "current" input entry number, since this will be preserved in the new entry - a re-evaluation on an old input line will not reuse the old input number. This will be assumed to be the input number assoicated with the last input prompt on the page, which will always be the default position for the next input
- Identifying the text positions denoting the beginning of the old input entry to be overwritten, and the location of the NEXT valid input prompt.
- Changing the handling of the "Enter" sequence to first redefine the input line number and remove the previous output, THEN produce the new output. Also, the new input line for the next entry should replace the old one at the end of the page.
- Deciding where to position the cursor for the next input. Probably the most logical place is at the beginning of the next input, but back at the end is another possibility.

Emacs comint mode uses a marker called `process-mark` to control the layout of input and output in such a way that it looks correct. This variable is set with a function called `comint-set-process-mark`.

This function is used at the end of the process to reset the prompts at the end of the page to their proper state.

⟨*axiom-repair-prompts*⟩≡

```
(defun axiom-repair-prompts ()
  "Repairs prompts at the end of the document once an overwrite eval is
complete.  Point is expected to be at the new prompt, after the last output.
Afterwards, point is where it was before."
  (save-excursion
    (let* ((new-prompt-start (re-search-backward axiom-prompt nil t))
           (new-prompt-end (re-search-forward axiom-prompt nil t))
           (new-prompt (buffer-substring new-prompt-start new-prompt-end))
           (inhibit-read-only t))
      ;; Remove the new prompt and put it in the kill ring
      (delete-region new-prompt-start new-prompt-end)
      ;; Weed out the extra newline, if present.  For example, in case of an
      ;; error it is not present.  I guess it would be better to detect
      ;; errors by parsing the prompt, but I leave this for somebody else.
      (when (char-equal (char-after) 10)
        (delete-char 1))
      ;;  Finally we get rid of the old prompt at the end of the buffer and
```

```
;;  insert the new one, which is currently somewhere in the middle of the
;;  buffer.
(goto-char (point-max))
(let ((old-prompt-start (re-search-backward axiom-prompt nil t)))
  (delete-region old-prompt-start (line-end-position))
  (insert new-prompt)
  (axiom-make-prompt old-prompt-start (point))))))
```

This is the core of the new evaluation routine, which handles most of the prompt arrangements. It makes use of the previous two functions.

⟨*axiom-overwrite-IO-eval*⟩≡

```
(defun axiom-overwrite-output-eval ()
  "Function which handles the actual mechanics of inserting a new IO pair.  It
expects point to be between the current and the next prompt."
  (let* (;; the old prompt is just before the input we want to evaluate
         (old-prompt-end (axiom-previous-prompt))
         (old-prompt-start (re-search-backward axiom-prompt nil t))
         ;; the new prompt is at the very end of the buffer
         (new-prompt (buffer-substring
                      (progn (goto-char (point-max))
                             (re-search-backward axiom-prompt nil t))
                      (re-search-forward axiom-prompt nil t)))
         (inhibit-read-only t))
    ;; Delete the old prompt
    (delete-region old-prompt-start old-prompt-end)
    ;; Clear out any pre-existing text overlays - needed for the case
    ;; where the same input is being re-evaluated - if this isn't
    ;; removed, the new prompt is bold.  If no overlay do nothing.
    (goto-char old-prompt-start)
    (axiom-clear-overlays (point))
    ;; Put the new prompt in
    (insert new-prompt)
    ;; Now we are done changing this prompt, and can prepare the input
    (comint-set-process-mark)
    (axiom-make-input (1- (point))
                      (setq axiom-end-of-input
                            (copy-marker (axiom-end-of-input))))
    ;; Delete the old output
    (delete-region (1+ axiom-end-of-input)
                   (axiom-next-prompt))

    ;; send new input
    (goto-char axiom-end-of-input)
    (comint-send-input)
    (axiom-wait-for-output)))
```

Normally, Return is simply bound to comint-send-input, so we will need this option for the “normal” case of evaluation at the last/most current input prompt.

⟨*axiom-normal-eval*⟩≡

```
(defun axiom-normal-eval ()
   "This function is used for evaluation at the 'front' Axiom prompt. It
expects point anywhere after the last prompt."
   (axiom-make-input (1- (axiom-previous-prompt))
                     (setq axiom-end-of-input
                           (copy-marker (axiom-end-of-input))))
   (comint-send-input)
   (axiom-wait-for-output))

(defun axiom-normal-eval-wait ()
   "This function is used for evaluation at the 'front' Axiom prompt. It
expects point anywhere after the last prompt.  It waits until the prompt appears."
   (axiom-make-input (1- (axiom-previous-prompt))
                     (setq axiom-end-of-input
                           (copy-marker (axiom-end-of-input))))
   (let ((pos (point)))
     (comint-send-input)
     (while (not (re-search-backward axiom-prompt pos t))
       (accept-process-output (get-buffer-process (current-buffer))))
     (comint-goto-process-mark)))
```

This is the function actually bound to the Return key - it decides which evaluation routine is needed based on the location of the input line.

⟨*axiom-eval-control*⟩≡

```
(defun axiom-eval ()
  "Evaluate the current input and insert output."
  (interactive)
  (let ((axiom-cd nil))
    (if axiom-system-command
        ;; we are responding to a system command.
        (progn
          (setq axiom-system-command nil
                axiom-end-of-input (copy-marker (axiom-end-of-input)))
          (comint-send-input)
          (axiom-wait-for-output)
          ;; If there is a prompt further down, we are overwriting old stuff.
          (when (re-search-forward axiom-prompt nil t)
            (re-search-backward axiom-prompt)
            (axiom-repair-prompts)
            (re-search-forward axiom-prompt nil t)
            (comint-set-process-mark)))

      ;; otherwise, we first check whether process-mark is at a prompt
      (when (axiom-prompt? (1- (process-mark (get-buffer-process (current-buffer)))))

        ;; do we have multiline input?
        (beginning-of-line)
        (if (looking-at ".*_ *$")
            (progn (end-of-line)
                   (newline))

          ;; move to the the end of the preceding prompt
          (axiom-previous-prompt)
          (comint-set-process-mark)
          ;; is it a system command?
          (setq axiom-system-command (looking-at " *)"))
          (when axiom-system-command
            (setq axiom-cd (looking-at " *)cd *")))

          ;; If there is a prompt further down, we are overwriting old stuff.
          (if (axiom-next-prompt)
              (progn (axiom-overwrite-output-eval)
                     ;; if we are now looking at a prompt, we are certainly not
                     ;; answering a question posed by axiom.
                     (when (looking-backward-at axiom-prompt)
                       (when axiom-cd (axiom-resync-directory))
```

```
                        (setq axiom-system-command nil)
                        (axiom-repair-prompts)
                        (end-of-line) ;; this moves point to the end of the prompt!
                        ;; it seems to work even if we type text
                        ;; during a computation.
                        (comint-set-process-mark)))
              (axiom-normal-eval)
              (when (looking-backward-at axiom-prompt)
                (when axiom-cd (axiom-resync-directory))
                (setq axiom-system-command nil))))))))

(defun axiom-eval-append ()
  "Evaluate the current input and append output."
  (interactive)
  (when (axiom-prompt? (1- (process-mark (get-buffer-process (current-buffer)))))
    (let* ((input (buffer-substring (axiom-previous-prompt)
                                    (axiom-end-of-input)))
           (end (progn (goto-char (point-max))
                       (point))))
      (delete-region (axiom-previous-prompt) end)
      (comint-set-process-mark)
      (insert input)
      (axiom-normal-eval))))
```

9.10 Command completion

It is not clear to what extend comman completion would be useful in Axiom. For the moment, I just enable filename completion. I'd like to restrict to `.input`, `.spad` and `.as` filenames, but I do not know how.

⟨*axiom-command-completion*⟩≡

```
(defun axiom-file-name-all-completions (pathnondir directory)
  "Returns all filenames relevant to axiom"
  (save-match-data
    (remove-if-not
     (function (lambda (f)
                 (or (and (string-match "\\.[^.]*\\'" f)
                          (member (match-string 0 f)
                                  (list ".input" ".spad" ".as")))
                     (string= (file-name-directory f) f))))
     (file-name-all-completions pathnondir directory))))
```

We are not using `file-name-extension`, since that would strip off trailing tildes.

The following probably could be simplified a lot. In principle, I need to find the position of the first mismatch.

⟨*axiom-command-completion*⟩+≡

```
(defun axiom-file-name-completion (file directory)
  "Returns the longest string common to all file names relevant to axiom in
DIRECTORY that start with FILE.  If there is only one and FILE matches it
exactly, returns t.  Returns nil if DIR contains no name starting with FILE."
  (let* ((completions (axiom-file-name-all-completions file directory))
         (frst (first completions))
         (len  (length frst))
         (start      0)
         (not-done   t))
    (cond ((consp (rest completions))
           (while (and not-done
                       (> len start))
             (let ((char (substring frst start (1+ start)))
                   (rst  (rest completions)))
               (while (and not-done
                           (consp rst))
                 (if (and (> (length (first rst)) start)
                          (string= (substring (first rst)
                                              start (1+ start))
                                   char))
                     (setq rst (rest rst))
                   (setq not-done nil))))
             (when not-done
               (setq start (1+ start))))
           (substring frst 0 start))
          ((string= frst file)
           t)
          (t
           frst))))
```

This is stolen from `comint.el`, I only changed `file-name-all-completions` to `axiom-file-name-all-completions`.

⟨*axiom-command-completion*⟩+≡

```
(defun axiom-dynamic-list-filename-completions ()
  "List in help buffer possible completions of the filename at point."
  (interactive)
  (let* ((completion-ignore-case (memq system-type '(ms-dos windows-nt)))
         ;; If we bind this, it breaks remote directory tracking in rlogin.el.
         ;; I think it was originally bound to solve file completion problems,
         ;; but subsequent changes may have made this unnecessary.  sm.
         ;;(file-name-handler-alist nil)
         (filename (or (comint-match-partial-filename) ""))
         (pathdir (file-name-directory filename))
         (pathnondir (file-name-nondirectory filename))
         (directory (if pathdir (comint-directory pathdir) default-directory))
         (completions (axiom-file-name-all-completions pathnondir directory)))
    (if (not completions)
        (message "No completions of %s" filename)
      (comint-dynamic-list-completions
       (mapcar 'comint-quote-filename completions)))))
```

Again, this is stolen from `comint.el`. I should specialise here, I think. Furthermore, directory tracking does not work yet.

⟨*axiom-command-completion*⟩+≡

```
(defun axiom-dynamic-complete ()
  "Dynamically perform completion at point."
  (interactive)
  (when (save-excursion
          (axiom-previous-prompt)
          (looking-at " *)"))
    (axiom-dynamic-complete-filename)))
```

Maybe the bit to detect whether we have a system command should go into a new function, since it's used in eval, too.

⟨*axiom-command-completion*⟩+≡

```
(defun axiom-dynamic-complete-filename ()
  "Dynamically complete at point as a filename.
See ‘comint-dynamic-complete-filename’.  Returns t if successful."
  (interactive)
  (let* ((completion-ignore-case (memq system-type '(ms-dos windows-nt)))
         (completion-ignored-extensions comint-completion-fignore)
         ;; If we bind this, it breaks remote directory tracking in rlogin.el.
         ;; I think it was originally bound to solve file completion problems,
         ;; but subsequent changes may have made this unnecessary.  sm.
         ;;(file-name-handler-alist nil)
         (minibuffer-p (window-minibuffer-p (selected-window)))
         (success t)
         (dirsuffix (cond ((not comint-completion-addsuffix)
                           "")
                          ((not (consp comint-completion-addsuffix))
                           (char-to-string directory-sep-char))
                          (t
                           (car comint-completion-addsuffix))))
         (filesuffix (cond ((not comint-completion-addsuffix)
                            "")
                           ((not (consp comint-completion-addsuffix))
                            " ")
                           (t
                            (cdr comint-completion-addsuffix))))
         (filename (or (comint-match-partial-filename) ""))
         (pathdir (file-name-directory filename))
         (pathnondir (file-name-nondirectory filename))
         (directory (if pathdir (comint-directory pathdir) default-directory))
         (completion (axiom-file-name-completion pathnondir directory)))
    (cond ((null completion)
           (message "No completions of %s" filename)
           (setq success nil))
          ((eq completion t)            ; Means already completed "file".
           (insert filesuffix)
           (unless minibuffer-p
             (message "Sole completion")))
          ((string-equal completion "") ; Means completion on "directory/".
           (axiom-dynamic-list-filename-completions))
          (t                            ; Completion string returned.
           (let ((file (concat (file-name-as-directory directory) completion)))
             (insert (comint-quote-filename
                      (substring (directory-file-name completion)
```

```
                              (length pathnondir))))
              (cond ((symbolp (axiom-file-name-completion completion directory))
                     ;; We inserted a unique completion.
                     (insert (if (file-directory-p file) dirsuffix filesuffix))
                     (unless minibuffer-p
                       (message "Completed")))
                    ((and comint-completion-recexact comint-completion-addsuffix
                          (string-equal pathnondir completion)
                          (file-exists-p file))
                     ;; It's not unique, but user wants shortest match.
                     (insert (if (file-directory-p file) dirsuffix filesuffix))
                     (unless minibuffer-p
                       (message "Completed shortest")))
                    ((or comint-completion-autolist
                         (string-equal pathnondir completion))
                     ;; It's not unique, list possible completions.
                     (axiom-dynamic-list-filename-completions))
                    (t
                     (unless minibuffer-p
                       (message "Partially completed")))))))
     success))
```

9.11 Inserting definitions from the kill ring

We want to copy function definitions from an `.input} file into the axiom`session. Because of differences in the parsers concerning multi-line input, the only practical way seems to be to write the definition into a temporary file and then `)read} it.`

⟨*axiom-yank*⟩≡

```
  (defun axiom-yank (&optional quiet)
    "puts the front item of the kill ring into a temporary file and makes axiom )read it
    (interactive "P")
    (let* ((tmp-file (make-temp-file "axiom" nil ".input"))
           (end (progn (goto-char (point-max))
                       (point))))
      (write-region (car kill-ring-yank-pointer) nil tmp-file)
      (delete-region (axiom-previous-prompt) end)
      (comint-set-process-mark)
      (insert (concat ")read " tmp-file
                      (if quiet " )quiet" "")))
      (axiom-normal-eval)
      (delete-file tmp-file)))
```

9.12 Restarting and Re-evaluating - Kill and Restart Axiom without Erasing the Document

There may be cases, such as the final preparation a printable form of a session, where it is desirable to start at the beginning - restart Axiom and re-evaluate all the inputs present. In this mode, this is handled by some special functions for restarting Axiom. There are a couple of options the user might wish to use - one is to reset Axiom and then automatically re-evaluate all inputs automatically, the other is to reset Axiom but leave the inputs unevaluated until the user manually evaluates them. Both are quite possible, although the automatic evaluation is somewhat more involved.

9.13 Axiom Specific Menus

It is common practice to define for a given mode a menu that contains utility functions for a given program. We will do so with Axiom.

9.14 Assembling the Pieces

Here we will put the functionality defined above into its proper order:

⟨*TerminalMode*⟩≡

```
;;################### Terminal Mode ########################
```

⟨*axiom-command-completion*⟩
⟨*axiom-yank*⟩
⟨*axiom-previous-input*⟩
⟨*axiom-next-input*⟩
⟨*axiom-scroll-input*⟩
⟨*axiom-backward-char*⟩
⟨*axiom-forward-char*⟩
⟨*axiom-mode-map*⟩
⟨*axiom-reset*⟩
⟨*axiom-text-selection*⟩
⟨*axiom-paint*⟩
⟨*axiom-changed-prompt-face*⟩
⟨*axiom-clear-text-properties*⟩
⟨*axiom-changed-function*⟩
⟨*axiom-flag-prompt*⟩
⟨*axiom-repair-prompts*⟩
⟨*axiom-overwrite-IO-eval*⟩
⟨*axiom-normal-eval*⟩
⟨*axiom-eval-control*⟩
⟨*axiom-maybe-save*⟩
⟨*defineaxiom-mode*⟩