Readline: Terminal Interaction

Version 4.1.4

January 20, 2009

The `"readline"` collection (not to be confused with MzScheme's `read-line` function) provides glue for using GNU's Readline library with the MzScheme `read-eval-print-loop`.

1 Normal Use of Readline

```
(require readline)
(require readline/rep-start)
```

The `readline` library installs a Readline-based input port, and hooks the prompt-and-read part of MzScheme's `read-eval-print-loop` to interact with it

You can start MzScheme with

```
mzscheme -il readline
```

or evaluate

```
(require readline)
```

in the MzScheme `read-eval-print-loop` to load Readline manually. You can also put (require readline) in your `"~/.mzschemerc"`, so that MzScheme automatically loads Readline support in interactive mode.

If you want to enable Readline support only sometimes—such as only when you use an `xterm`, and not when you use an Emacs shell—then you can use `dynamic-require`, as in the following example:

```
(when (regexp-match? #rx"xterm"
                     (getenv "TERM"))
  (dynamic-require 'readline #f))
```

The `readline` library automatically checks whether the current input port is a terminal, as determined by `terminal-port?`, and it installs Readline only to replace terminal ports. The `readline/rep-start` module installs Readline without a terminal check.

By default, Readline's completion is set to use the visible bindings in the current namespace. This is far from ideal, but it's better than Readline's default filename completion which is rarely useful. In addition, the Readline history is stored across invocations in MzScheme's preferences file, assuming that MzScheme exits normally.

`(install-readline!) → void?`

Adds `(require readline)` to the result of `(find-system-path 'init-file)`, which is `"~/.mzschemerc"` under Unix. Consequently, Readline will be loaded whenever MzScheme is started in interactive mode. The declaration is added only if it is not already present, as determined by `read`ing and checking all top-level expressions in the file. For more fine-grained control, such as conditionally loading Readline based on an environment variable, edit `"~/.mzschemerc"` manually.

2 Interacting with the Readline-Enabled Input Port

```
(require readline/pread)
```

The `readline/pread` library provides customization, and support for prompt-reading after `readline` installs the new input port.

The reading facility that the new input port provides can be customized with the following parameters.

```
(current-prompt) → bytes?
(current-prompt bstr) → void?
  bstr : bytes?
```

A parameter that determines the prompt that is used, as a byte string. Defaults to `#"> "`.

```
(show-all-prompts) → boolean?
(show-all-prompts on?) → void?
  on? : any/c
```

A parameter. If `#f`, no prompt is shown until you write input that is completely readable. For example, when you type

```
(foo bar) (+ 1
             2)
```

you will see a single prompt in the beginning.

The problem is that the first expression can be `(read-line)`, which normally consumes the rest of the text on the *same* line. The default value of this parameter is therefore `#t`, making it mimic plain I/O interactions.

```
(max-history) → exact-nonnegative-integer?
(max-history n) → void?
  n : exact-nonnegative-integer?
```

A parameter that determines the number of history entries to save, defaults to `100`.

```
(keep-duplicates) → (one-of/c #f 'unconsecutive #t)
(keep-duplicates keep?) → void?
  keep? : (one-of/c #f 'unconsecutive #t)
```

A parameter. If `#f` (the default), then when a line is equal to a previous one, the previous one is removed. If it set to `'unconsecutive` then this happens only for an line that duplicates the previous one, and if it is `#f` then all duplicates are kept.

```
(keep-blanks) → boolean?
(keep-blanks keep?) → void?
  keep? : any/c
```

A parameter. If `#f` (the default), blank input lines are not kept in history.

```
(readline-prompt) → (or/c false/c bytes? (one-of/c 'space))
(readline-prompt status) → void?
  status : (or/c false/c bytes? (one-of/c 'space))
```

The new input port that you get when you require `readline` is a custom port that uses Readline for all inputs. The problem is when you want to display a prompt and then read some input, Readline will get confused if it is not used when the cursor is at the beginning of the line (which is why it has a *`prompt`* argument.) To use this prompt:

```
(parameterize ([readline-prompt some-byte-string])
  ...code-that-reads...)
```

This expression makes the first call to Readline use the prompt, and subsequent calls will use an all-spaces prompt of the same length (for example, when you're reading an S-expression). The normal value of `readline-prompt` is `#f` for an empty prompt (and spaces after the prompt is used, which is why you should use `parameterize` to restore it to `#f`).

A proper solution would be to install a custom output port, too, which keeps track of text that is displayed without a trailing newline. As a cheaper solution, if line-counting is enabled for the terminal's output-port, then a newline is printed before reading if the column is not 0. (The `readline` library enables line-counting for the output port.)

Warning: The Readline library uses the output port directly. You should not use it when `current-input-port` has been modified, or when it was not a terminal port when MzScheme was started (eg, when reading input from a pipe). Expect some problems if you ignore this warning (not too bad, mostly problems with detecting an EOF).

3 Direct Bindings for Readline Hackers

```
(require readline/readline)
```

```
(readline prompt) → string?
  prompt : string?
```

Prints the given prompt string and reads a line.

```
(readline-bytes prompt) → bytes?
  prompt : bytes?
```

Like `readline`, but using raw byte-strings for the prompt and returning a byte string.

```
(add-history str) → void?
  str : string?
```

Adds the given string to the Readline history, which is accessible to the user via the up-arrow key.

```
(add-history-bytes str) → void?
  str : bytes?
```

Adds the given byte string to the Readline history, which is accessible to the user via the up-arrow key.

```
(history-length) → exact-nonnegative-integer?
```

Returns the length of the history list.

```
(history-get idx) → string?
  idx : integer?
```

Returns the history string at the *`idx`* position. *`idx`* can be negative, which will make it count from the last (i.e, `-1` returns the last item, `-2` returns the second-to-last, etc.)

```
(history-delete idx) → string?
  idx : integer?
```

Deletes the history string at the *`idx`* position. The position is specified in the same way as the argument for `history-get`.

```
(set-completion-function! proc [type]) → void?
  proc : ((or/c string? bytes?)
          . -> . (listof (or/c string? bytes?)))
  type : (one-of/c _string _bytes) = _string
```

Sets Readline's `rl_completion_entry_function` to *`proc`*. The *`type`* argument, whose possible values are from `scheme/foreign`, determines the type of value supplied to the *`proc`*.

4 License Issues

GNU's Readline library is covered by the GPL, and that applies to code that links with it. PLT Scheme is LGPL, so this code is not used by default; you should explicitly enable it if you want to. Also, be aware that if you write code that uses this library, it will make your code link to the Readline library when invoked, with the usual GPL implications.