Readline: Terminal Interaction

Version 9.4.0.5

November 15, 2026

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

Due to licensing issues, the Readline collection is by default backed by the Editline library. To switch to GNU's Readline library, either install the "readline-gpl" package, or set the PLT_READLINE_LIB environment variable to the library, which configures the readline collection to use Readline.

1 Normal Use of Readline

```
(require readline)      package: readline-lib
(require readline/rep-start)
```

The `readline` library installs a Readline-based input port (whose name is `'readline-input`) and hooks the prompt-and-read part of Racket's `read-eval-print-loop` to interact with it.

You can start Racket with

```
racket -il readline
```

or evaluate

```
(require readline)
```

in the Racket `read-eval-print-loop` to load Readline manually. You can also put (require readline) in your `"~/.racketrc"`, so that Racket 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 also checks that its name, as determined by `object-name`, is `'stdin`, and it installs Readline only to replace the stdin terminal port. 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 Racket's preferences file, assuming that Racket exits normally.

The `readline` library adjusts `read-eval-print-loop` by setting the prompt read handler as determined by `current-prompt-read`. The call to the read interaction handler (as determined by `current-read-interaction`) is parameterized to set `readline-prompt`, so that a prompt will be printed when reading starts. To compensate for the prompt printed via `readline-prompt`, when the interaction input port's name (as produced by function in the `current-get-interaction-input-port` parameter) is `'readline-input`, the prompt read handler skips printing a prompt; otherwise, it displays a prompt as determined by `current-prompt`.

```
(install-readline!) → void?
```

Adds `(require readline/rep)` to the result of `(find-system-path 'init-file)`, which is `"~/.racketrc"` on Unix. Consequently, Readline will be loaded whenever Racket is started in interactive mode. The declaration is added only if it is not already present, as determined by reading 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 `"~/.racketrc"` manually.

```
pre-readline-input-port : (or/c input-port? false/c)
```

If required through `readline/rep-start`, `pre-readline-input-port` will always be the input port replaced by the readline input port.

If required through `readline`, `pre-readline-input-port` will be an input port only when the `current-input-port` is actually replaced. Otherwise, it is `#f`.

Using `pre-readline-input-port` is useful for sending the original stdin to subprocesses. Subprocesses generally require an input port backed by a file descriptor, and many interactive programs behave differently when they have a terminal file descriptor. Otherwise, `pre-readline-input-port` should not be used, as reading from it will interfere with the readline port.

Added in version 1.1 of package `readline-lib`.

2 Interacting with the Readline-Enabled Input Port

```
(require readline/pread)
```
package: readline-lib

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 #"> ".

```
(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 #t 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 Racket 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)
```
package: readline-lib

```
(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 `ffi/unsafe`, determines the type of value supplied to the *proc*.

In the CS variant of Racket, *proc* is called in atomic mode.

```
(set-completion-append-character! c) → void?
  c : char?
```

Sets Readline's `rl_completion_append_character` to *c*. The value is reset by the readline library, so it must be set inside a completion function each time it is called. The default is `#\space`. Set it to `#\nul` to have no character appended to the completion result.

The `rl_completion_append_character` is used by the readline library whenever a completion function returns a single option, and it is therefore chosen. The choice will be filled in on the command line, and then the `rl_completion_append_character` is appended.

As an example, you could set the completion append character to a slash when completing the name of a directory, a space when you know the user will want to write another argument, or `#\nul` to avoid appending when a user might want to write something directly after the completion, such as punctuation or an extension of a word.

If you want to make a completion function to more easily write the names of your favorite characters (and share your excitement about them), a use of `set-completion-append-character!` may look like this:

```
(define (christmas-character-complete name-str)
  (set-completion-append-character! #\!)
  (filter (λ (x) (string-prefix? x name-str))
          '("Rudolf" "Hermie" "Bumble" "Yukon" "Clarise" "Santa")))
```

Added in version 1.1 of package `readline-lib`.

```
(readline-newline) → void?
```

Sets the cursor to the start of a new line.

```
(readline-redisplay) → void?
```

Forces a redisplay of the Readline prompt and current user input.

The `readline-redisplay` function can be used together with `readline-newline` to prevent a background thread from cluttering up the user input by interleaving its output. For example, an unsafe wrapper function for the thread's output might look like the following:

```
(define (with-thread-safe-output output-thunk)
  (dynamic-wind
```

```
(lambda ()
  (start-atomic)
  (readline-newline))
output-thunk
(lambda ()
  (readline-redisplay)
  (end-atomic))))
```