# `mzpp` and `mztext`: Preprocessors

Version 9.4.0.5

Eli Barzilay

November 14, 2026

The `"preprocessor"` collection defines two Racket-based preprocessors for texts that can have embedded Racket code. The two processors share a few features, like several command-line flags and the fact that embedded Racket code is case-sensitive by default.

Note that these processors are **not** intended as preprocessors for Racket code, since you have macros to do that.

# Contents

**1 Overview** **3**

**2 `mzpp`** **4**

2.1 Invoking mzpp . . . . . 4

2.2 mzpp files . . . . . 4

2.3 Raw preprocessing directives . . . . . 6

2.4 The mzpp read-eval-print loop . . . . . 7

2.5 Provided bindings . . . . . 7

**3 `mztext`** **10**

3.1 Invoking mztext . . . . . 10

3.2 mztext processing: the standard command dispatcher . . . . . 10

3.3 Provided bindings . . . . . 13

# 1 Overview

`(require preprocessor/pp-run)` package: `preprocessor`

The preprocessors can be invoked from Racket programs, but the main usage should be through the launchers. Both launchers use code from `preprocessor/pp-run` that allows a special invocation mode through the `--run` flag.

The `--run` is a convenient way of making the preprocessors cooperate with some other command, making it possible to use preprocessed text without an additional glue script or a makefile. The following examples use `mzpp`, but they work with `mztext` too. `--run` uses a single argument which is a string specifying a command to run:

- 1. In its simplest form, the command string specifies some shell command which will be executed with its standard input piped in from the preprocessor's output. For example, `mzpp --run pr foo` is the same as `mzpp foo | pr`. An error is raised if an output file is specified with such an argument.
- 2. If the command string contains a `*` and an output file is specified, then the command will be executed on this output file after it is generated. For example, `mzpp --run 'pr *' -o foo x y z` is the same as `mzpp -o foo x y z; pr foo`.
- 3. If the command string contains a `*`, and no output file is specified, and there is exactly one input file, then a temporary file will be used to save the original while the command is running. For example, `mzpp --run 'pr *' foo` is the same as `mv foo foo-mzpp-temporary; mzpp -o foo foo-mzpp-temporary; pr foo; rm foo; mv foo-mzpp-temporary foo`. If there is an error while `mzpp` is running, the working file will be erased and the original will be renamed back.
- 4. Any other cases where the command string contains a `*` are invalid.

If an executed command fails with a return status different than 0, the preprocessor execution will signal a failure by returning 1.

# 2 mzpp

mzpp is a simple preprocessor that allows mixing Racket code with text files in a similar way to PHP or BRL. Processing of input files works by translating the input file to Racket code that prints the contents, except for marked portions that contain Racket code. The Racket parts of a file are marked with << and >> tokens by default. The Racket code is then passed through a read-eval-print loop that is similar to a normal REPL with a few differences in how values are printed.

## 2.1 Invoking mzpp

Use the --h flag to get the available flags. See above for an explanation of the --run flag.

## 2.2 mzpp files

Here is a sample file that mzpp can process, using the default beginning and ending markers:

```
<< (define bar "BAR") >>
foo1
foo2 << bar newline* bar >> baz
foo3
```

First, this file is converted to the following Racket code:

```
(thunk (cd "tmp/") (current-file "foo"))
(thunk (push-indentation ""))
 (define bar "BAR") (thunk (pop-indentation))
newline*
"foo1"
newline*
"foo2 "
(thunk (push-indentation "     "))
 bar newline* bar (thunk (pop-indentation))
" baz"
newline*
"foo3"
newline*
(thunk (cd "/home/eli") (current-file #f))
```

which is then fed to the REPL, resulting in the following output:

```
foo1
foo2 BAR
```

```
     BAR baz
foo3
```

To see the processed input that the REPL receives, use the `--debug` flag. Note that the processed code contains expressions that have no side-effects, only values—see below for an explanation of the REPL printing behavior. Some expressions produce values that change the REPL environment, for example, the indentation commands are used to keep track of the column where the Racket marker was found, and `cd` is used to switch to the directory where the file is (here it was in `"/home/foo/tmp"`) so including a relative file works. Also, note that the first `newline*` did not generate a newline, and that the one in the embedded Racket code added the appropriate spaces for indentation.

It is possible to temporarily switch from Racket to text-mode and back in a way that does not respect a complete Racket expression, but you should be aware that text is converted to a *sequence* of side-effect free expressions (not to a single string, and not expression that uses side effects). For example:

```
<< (if (zero? (random 2))
     (list >>foo1<<)
     (list >>foo2<<))
>>
<< (if (zero? (random 2)) (list >>
foo1
<<) (list >>
foo2
<<)) >>
```

will print two lines, each containing `foo1` or `foo` (the first approach plays better with the smart space handling). The `show` function can be used instead of `list` with the same results, since it will print out the values in the same way the REPL does. The conversion process does not transform every continuous piece of text into a single Racket string because doing this:

- the Racket process will need to allocating big strings which makes this unfeasible for big files,
- it will not play well with "interactive" input feeding, for example, piping in the output of some process will show results only on Racket marker boundaries,
- special treatment for newlines in these strings will become expensive.

(Note that this is different from the BRL approach.)

## 2.3 Raw preprocessing directives

Some preprocessing directives happen at the "raw level"—the stage where text is transformed into Racket expressions. These directives cannot be changed from within transformed text because they change the way this transformation happens. Some of these transformation

- Skipping input:

  First, the processing can be modified by specifying a `skip-to` string that disables any output until a certain line is seen. This is useful for script files that use themselves for input. For example, the following script:

  ```
  #!/bin/sh
  echo shell output
  exec mzpp -s "---TEXT-START---" "$0"
  exit 1
  ---TEXT-START---
  Some preprocessed text
  123*456*789 = << (* 123 456 789) >>
  ```

  will produce this output:

  ```
  shell output
  Some preprocessed text
  123*456*789 = 44253432
  ```

- Quoting the markers:

  In case you need to use the actual text of the markers, you can quote them. A backslash before a beginning or an ending marker will make the marker treated as text, it can also quote a sequence of backslashes and a marker. For example, using the default markers, `\<<\>>` will output `<<>>`, `\\<<\\\>>` will output `\<<\\>>` and `\a\b\<<` will output `\a\b<<`.

- Modifying the markers:

  Finally, if the markers collide with a certain file contents, it is possible to change them. This is done by a line with a special structure—if the current Racket markers are `<beg1>` and `<end1>` then a line that contains exactly:

  ```
  <beg1><beg2><beg1><end1><end2><end1>
  ```

  will change the markers to `<beg2>` and `<end2>`. It is possible to change the markers from the Racket side (see below), but this will not change already-transformed text, which is the reason for this special format.

## 2.4 The mzpp read-eval-print loop

The REPL is initialized by requiring `preprocessor/mzpp`, so the same module provides both the preprocessor functionality as well as bindings for embedded Racket code in processed files. The REPL is then fed the transformed Racket code that is generated from the source text (the same code that `--debug` shows). Each expression is evaluated and its result is printed using the `show` function (multiple values are all printed), where `show` works in the following way:

- `#<void>` and `#f` values are ignored.
- Structures of pairs are recursively scanned and their parts printed (no spaces are used, so to produce Racket code as output you must use format strings—again, this is not intended for preprocessing Racket code).
- Procedures are applied to zero arguments (so a procedure that doesn't accept zero arguments will cause an error) and the result is sent back to `show`. This is useful for using thunks to wrap side-effects as values (e.g, the `thunk` wraps shown by the debug output above).
- Promises are forced and the result is sent again to `show`.
- All other values are printed with `display`. No newlines are used after printing values.

## 2.5 Provided bindings

`(require preprocessor/mzpp)` package: `preprocessor`

First, bindings that are mainly useful for invoking the preprocessor:

```
(preprocess in ...) → void?
  in : (or/c path-string? input-port?)
```

This is the main entry point to the preprocessor—invoking it on the given list of files and input ports. This is quite similar to `include`, but it adds some setup of the preprocessed code environment (like requiring the `mzpp` module).

```
(skip-to) → string?
(skip-to str) → void?
  str : string?
```

A string parameter—when the preprocessor is started, it ignores everything until a line that contains exactly this string is encountered. This is primarily useful through a command-line flag for scripts that extract some text from their own body.

```
(debug?) → boolean?
(debug? on?) → void?
  on? : any/c
```

A boolean parameter. If true, then the REPL is not invoked, instead, the converted Racket code is printed as is.

```
(no-spaces?) → boolean?
(no-spaces? on?) → void?
  on? : any/c
```

A boolean parameter. If true, then the "smart" preprocessing of spaces is turned off.

```
(beg-mark) → string?
(beg-mark str) → void?
  str : string?
(end-mark) → string?
(end-mark str) → void?
  str : string?
```

These two parameters are used to specify the Racket beginning and end markers.

All of the above are accessible in preprocessed texts, but the only one that might make any sense to use is `preprocess` and `include` is a better choice. When `include` is used, it can be wrapped with parameter settings, which is why they are available. Note in particular that these parameters change the way that the text transformation works and have no effect over the current preprocessed document (for example, the Racket marks are used in a different thread, and `skip-to` cannot be re-set when processing has already began). The only one that could be used is `no-spaces?` but even that makes little sense on selected parts.

The following are bindings that are used in preprocessed texts:

```
(push-indentation str) → void?
  str : string?
(pop-indentation) → void?
```

These two calls are used to save the indentation column where the Racket beginning mark was found, and will be used by `newline*` (unless smart space handling mode is disabled).

```
(show v) → void?
  v : any/c
```

The arguments are displayed as specified above.

```
(newline*) → void?
```

This is similar to `newline` except that it tries to handle spaces in a "smart" way—it will print a newline and then spaces to reach the left margin of the opening `<<`. (Actually, it tries a bit more, for example, it won't print the spaces if nothing is printed before another newline.) Setting `no-spaces?` to true disable this leaving it equivalent to `newline`.

```
(include file ...) → void?
  file : path-string?
```

This is the preferred way of including another file in the processing. File names are searched relatively to the current preprocessed file, and during processing the current directory is temporarily changed to make this work. In addition to file names, the arguments can be input ports (the current directory is not changed in this case). The files that will be incorporated can use any current Racket bindings etc, and will use the current markers—but the included files cannot change any of the parameter settings for the current processing (specifically, the marks and the working directory will be restored when the included files are processed).

Note that when a sequence of files are processed (through command-line arguments or through a single `include` expression), then they are all taken as one textual unit—so changes to the markers, working directory etc in one file can modify the way sequential files are processed. This means that including two files in a single `include` expression can be different than using two expressions.

```
stdin : parameter?
stdout : parameter?
stderr : parameter?
cd : parameter?
```

These are shorter names for the corresponding port parameters and `current-directory`.

```
(current-file) → path-string?
(current-file path) → void?
  path : path-string?
```

This is a parameter that holds the name of the currently processed file, or #f if none.

```
(thunk expr ...)
```

Expands to `(lambda () expr ...)`.

# 3 mztext

`mztext` is another Racket-based preprocessing language. It can be used as a preprocessor in a similar way to `mzpp` since it also uses `preprocessor/pp-run` functionality. However, `mztext` uses a completely different processing principle, it is similar to TeX rather than the simple interleaving of text and Racket code done by `mzpp`.

Text is being input from file(s), and by default copied to the standard output. However, there are some magic sequences that trigger handlers that can take over this process—these handlers gain complete control over what is being read and what is printed, and at some point they hand control back to the main loop. On a high-level point of view, this is similar to "programming" in TeX, where macros accept as input the current input stream. The basic mechanism that makes this programming is a *composite input port* which is a prependable input port—so handlers are not limited to processing input and printing output, they can append their output back on the current input which will be reprocessed.

The bottom line of all this is that `mztext` is can perform more powerful preprocessing than the `mzpp`, since you can define your own language as the file is processed.

## 3.1 Invoking mztext

Use the `-h` flag to get the available flags. SEE above for an explanation of the `--run` flag.

## 3.2 mztext processing: the standard command dispatcher

`mztext` can use arbitrary magic sequences, but for convenience, there is a default built-in dispatcher that connects Racket code with the preprocessed text—by default, it is triggered by `@`. When file processing encounters this marker, control is transferred to the command dispatcher. In its turn, the command dispatcher reads a Racket expression (using `read`), evaluates it, and decides what to do next. In case of a simple Racket value, it is converted to a string and pushed back on the preprocessed input. For example, the following text:

```
foo
@"bar"
@(+ 1 2)
@"@(* 3 4)"
@(/ (read) 3)12
```

generates this output:

```
foo
bar
3
```

```
12
4
```

An explanation of a few lines:

- `@"bar"`, `@(+ 1 2)`—the Racket objects that is read is evaluated and displayed back on the input port which is then printed.
- `@"@(* 3 4)"` — demonstrates that the results are "printed" back on the input: the string that in this case contains another use of `@` which will then get read back in, evaluated, and displayed.
- `@(/ (read) 3)12` — demonstrates that the Racket code can do anything with the current input.

The complete behavior of the command dispatcher follows:

- If the marker sequence is followed by itself, then it is simply displayed, using the default, `@@` outputs a `@`.
- Otherwise a Racket expression is read and evaluated, and the result is processed as follows:
  - If the result consists of multiple values, each one is processed,
  - If it is `#<void>` or `#f`, nothing is done,
  - If it is a structure of pairs, this structure is processed recursively,
  - If it is a promise, it is forced and its value is used instead,
  - Strings, bytes, and paths are pushed back on the input stream,
  - Symbols, numbers, and characters are converted to strings and pushed back on the input,
  - An input port will be perpended to the input, both processed as a single input,
  - Procedures of one or zero arity are treated in a special way—see below, other procedures cause an error
  - All other values are ignored.
- When this processing is done, and printable results have been re-added to the input port, control is returned to the main processing loop.

A built-in convenient behavior is that if the evaluation of the Racket expression returned a `#<void>` or `#f` value (or multiple values that are all `#<void>` or `#f`), then the next newline is swallowed using `swallow-newline` (see below) if there is just white spaces before it.

During evaluation, printed output is displayed as is, without re-processing. It is not hard to do that, but it is a little expensive, so the choice is to ignore it. (A nice thing to do is to redesign

this so each evaluation is taken as a real filter, which is done in its own thread, so when a Racket expression is about to evaluated, it is done in a new thread, and the current input is wired to that thread's output. However, this is much too heavy for a "simple" preprocesser...)

So far, we get a language that is roughly the same as we get from `mzpp` (with the added benefit of reprocessing generated text, which could be done in a better way using macros). The special treatment of procedure values is what allows more powerful constructs. There are handled by their arity (preferring a the nullary treatment over the unary one):

- A procedure of arity 0 is simply invoked, and its resulting value is used. The procedure can freely use the input stream to retrieve arguments. For example, here is how to define a standard C function header for use in a Racket extension file:

```
@(define (cfunc)
   (format
    "Scheme_Object *~a(int argc, Scheme_Object *argv[])\n"
    (read-line)))
@cfunc foo
@cfunc bar

==>

Scheme_Object * foo(int argc, Scheme_Object *argv[])
Scheme_Object * bar(int argc, Scheme_Object *argv[])
```

  Note how `read-line` is used to retrieve an argument, and how this results in an extra space in the actual argument value. Replacing this with `read` will work slightly better, except that input will have to be a Racket token (in addition, this will not consume the final newline so the extra one in the format string should be removed). The `get-arg` function can be used to retrieve arguments more easily—by default, it will return any text enclosed by parenthesis, brackets, braces, or angle brackets (see below). For example:

```
@(define (tt)
   (format "<tt>~a</tt>" (get-arg)))
@(define (ref)
   (format "<a href=~s>~a</a>" (get-arg) (get-arg)))
@(define (ttref)
   (format "<a href=~s>@tt{~a}</a>" (get-arg) (get-arg)))
@(define (reftt)
   (format "<a href=~s>~a</a>" (get-arg) (tt)))
@ttref{racket-lang.org}{Racket}
@reftt{racket-lang.org}{Racket}

==>

<a href="racket-lang.org"><tt>Racket</tt></a>
<a href="racket-lang.org"><tt>Racket</tt></a>
```

Note that in `reftt` we use `tt` without arguments since it will retrieve its own arguments. This makes `ttref`'s approach more natural, except that "calling" `tt` through a Racket string doesn't seem natural. For this there is a `defcommand` command (see below) that can be used to define such functions without using Racket code:

```
@defcommand{tt}{X}{<tt>X</tt>}
@defcommand{ref}{url text}{<a href="url">text</a>}
@defcommand{ttref}{url text}{<a href="url">@tt{text}</a>}
@ttref{racket-lang.org}{Racket}

==>

<a href="racket-lang.org"><tt>Racket</tt></a>
```

- A procedure of arity 1 is invoked differently—it is applied on a thunk that holds the "processing continuation". This application is not expected to return, instead, the procedure can decide to hand over control back to the main loop by using this thunk. This is a powerful facility that is rarely needed, similarly to the fact that `call/cc` is rarely needed in Racket.

Remember that when procedures are used, generated output is not reprocessed, just like evaluating other expressions.

## 3.3 Provided bindings

`(require preprocessor/mztext)` package: `preprocessor`

Similarly to `mzpp`, `preprocessor/mztext` contains both the implementation as well as user-visible bindings.

Dispatching-related bindings:

```
(command-marker) → string?
(command-marker str) → void?
  str : string?
```

A string parameter-like procedure that can be used to set a different command marker string. Defaults to `@`. It can also be set to `#f` which will disable the command dispatcher altogether. Note that this is a procedure—it cannot be used with `parameterize`.

```
(dispatchers) → (listof list?)
(dispatchers disps) → void?
  disps : (listof list?)
```

A parameter-like procedure (same as `command-marker`) holding a list of lists—each one a dispatcher regexp and a handler function. The regexp should not have any parenthesized

subgroups, use `"(?:...)"` for grouping. The handler function is invoked whenever the regexp is seen on the input stream: it is invoked on two arguments—the matched string and a continuation thunk. It is then responsible for the rest of the processing, usually invoking the continuation thunk to resume the default preprocessing. For example:

```
@(define (foo-handler str cont)
   (add-to-input (list->string
                  (reverse (string->list (get-arg)))))
   (cont))
@(dispatchers (cons (list "foo" foo-handler) (dispatchers)))
foo{>Foo<oof}

==>

Foo
```

Note that the standard command dispatcher uses the same facility, and it is added by default to the dispatcher list unless `command-marker` is set to `#f`.

```
(make-composite-input v ...) → input-port?
  v : any/c
```

Creates a composite input port, initialized by the given values (input ports, strings, etc). The resulting port will read data from each of the values in sequence, appending them together to form a single input port. This is very similar to `input-port-append`, but it is extended to allow prepending additional values to the beginning of the port using `add-to-input`. The `mztext` executable relies on this functionality to be able to push text back on the input when it is supposed to be reprocessed, so use only such ports for the current input port.

```
(add-to-input v ...) → void?
  v : any/c
```

This should be used to "output" a string (or an input port) back on the current composite input port. As a special case, thunks can be added to the input too—they will be executed when the "read header" goes past them, and their output will be added back instead. This is used to plant handlers that happen when reading beyond a specific point (for example, this is how the directory is changed to the processed file to allow relative includes). Other simple values are converted to strings using `format`, but this might change.

```
(paren-pairs) → (listof (list/c string? string?))
(paren-pairs pairs) → void?
  pairs : (listof (list/c string? string?))
```

This is a parameter holding a list of lists, each one holding two strings which are matching open/close tokens for `get-arg`.

```
(get-arg-reads-word?) → boolean?
(get-arg-reads-word? on?) → void?
  on? : any/c
```

A parameter that holds a boolean value defaulting to `#f`. If true, then `get-arg` will read a whole word (non-whitespace string delimited by whitespaces) for arguments that are not parenthesized with a pair in `paren-pairs`.

```
(get-arg) → (or/c string? eof-object?)
```

This function will retrieve a text argument surrounded by a paren pair specified by `paren-pairs`. First, an open-pattern is searched, and then text is assembled making sure that open-close patterns are respected, until a matching close-pattern is found. When this scan is performed, other parens are ignored, so if the input stream has `{[(}`, the return value will be `"[("`. It is possible for both tokens to be the same, which will have no nesting possible. If no open-pattern is found, the first non-whitespace character is used, and if that is also not found before the end of the input, an `eof` value is returned. For example (using `defcommand` which uses `get-arg`):

```
  @(paren-pairs (cons (list "|" "|") (paren-pairs)))
  @defcommand{verb}{X}{<tt>X</tt>}
  @verb abc
  @(get-arg-reads-word? #t)
  @verb abc
  @verb |FOO|
  @verb

==>

  <tt>a</tt>bc
  <tt>abc</tt>
  <tt>FOO</tt>
  verb: expecting an argument for `X'
```

```
(get-arg*) → (or/c string? eof-object?)
```

Similar to `get-arg`, except that the resulting text is first processed. Since arguments are usually text strings, "programming" can be considered as lazy evaluation, which sometimes can be too inefficient (TeX suffers from the same problem). The `get-arg*` function can be used to reduce some inputs immediately after they have been read.

```
(swallow-newline) → void?
```

This is a simple command that simply does this:

```
(regexp-try-match #rx"^[ \t]*\r?\n" (stdin))
```

The result is that a newline will be swallowed if there is only whitespace from the current location to the end of the line. Note that as a general principle `regexp-try-match` should be preferred over `regexp-match` for `mztext`'s preprocessing.

```
(defcommand name args text) → void?
  name : any/c
  args : list?
  text : string?
```

This is a command that can be used to define simple template commands. It should be used as a command, not from Racket code directly, and it should receive three arguments:

- The name for the new command (the contents of this argument is converted to a string),
- The list of arguments (the contents of this is turned to a list of identifiers),
- Arbitrary text, with **textual** instances of the variables that denote places they are used.

For example, the sample code above:

```
@defcommand{ttref}{url text}{<a href="url">@tt{text}</a>}
```

is translated to the following definition expression:

```
(define (ttref)
  (let ([url (get-arg)] [text (get-arg)])
    (list "<a href=\"" url "\">@tt{" text "}</a>")))
```

which is then evaluated. Note that the arguments play a role as both Racket identifiers and textual markers.

```
(include file ...) → void?
  file : path-string?
```

This will add all of the given inputs to the composite port and run the preprocessor loop. In addition to the given inputs, some thunks are added to the input port (see `add-to-input` above) to change directory so relative includes work.

If it is called with no arguments, it will use `get-arg` to get an input filename, therefore making it possible to use this as a dispatcher command as well.

```
(preprocess in) → void?
  in : (or/c path-string? input-port?)
```

This is the main entry point to the preprocessor—creating a new composite port, setting internal parameters, then calling `include` to start the preprocessing.

```
stdin : parameter?
stdout : parameter?
stderr : parameter?
cd : parameter?
```

These are shorter names for the corresponding port parameters and `current-directory`.

```
(current-file) → path-string?
(current-file path) → void?
  path : path-string?
```

This is a parameter that holds the name of the currently processed file, or #f if none.