# `raco`: Racket Command-Line Tools

Version 6.4

February 8, 2016

The `raco` program supports various Racket tasks from a command line. The first argument to `raco` is always a specific command name. For example, `raco make` starts a command to compile a Racket source module to bytecode format.

The set of commands available through `raco` is extensible. Use `raco help` to get a complete list of available commands for your installation. This manual covers the commands that are available in a typical Racket installation.

# Contents

# 1 `raco make`: Compiling Source to Bytecode

The `raco make` command accept filenames for Racket modules to be compiled to bytecode format. Modules are re-compiled only if the source Racket file is newer than the bytecode file and has a different SHA-1 hash, or if any imported module is recompiled or has a different SHA-1 hash for its compiled form plus dependencies.

The `raco make` command accepts a few flags:

- `-j` ⟨*n*⟩ — Compiles argument modules in parallel, using up to ⟨*n*⟩ parallel tasks.
- `--disable-inline` — Disables function inlining while compiling (but does not re-compile files that are already up-to-date). This flag is often useful to simplify generated code before decompiling, and it corresponds to setting `compile-context-preservation-enabled` to `#t`.
- `--disable-constant` — Disables inference of definitions within a module as constant (but does not re-compile files that are already up-to-date). The value associated with a non-constant definition is never inlined or constant-propagated, either within its own module or an importing module. This flag corresponds to setting `compile-enforce-module-constants` to `#f`.
- `--no-deps` — Compiles a non-module file (i.e., one that is run via `load` instead of `require`). See §1.8 “Compiling to Raw Bytecode” for more information.
- `-p` ⟨*file*⟩ or `--prefix` ⟨*file*⟩ — For use with `--no-deps`; see §1.8 “Compiling to Raw Bytecode”.
- `-no-prim` — For use with `--no-deps`; see §1.8 “Compiling to Raw Bytecode”.
- `-v` — Verbose mode, which shows which files are compiled.
- `--vv` — Very verbose mode, which implies `-v` and also shows every dependency that is checked.

## 1.1 Bytecode Files

A file "⟨*name*⟩.⟨*ext*⟩" is compiled to bytecode that is saved as "`compiled/`⟨*name*⟩`_`⟨*ext*⟩`.zo`" relative to the file. As a result, the bytecode file is normally used automatically when "⟨*name*⟩.⟨*ext*⟩" is required as a module, since the underlying `load/use-compiled` operation detects such a bytecode file.

For example, in a directory that contains the following files:

- `"a.rkt"`:

```
#lang racket
(require "b.rkt" "c.rkt")
(+ b c)
```

- `"b.rkt"`:

```
#lang racket
(provide b)
(define b 1)
```

- `"c.rkt"`:

```
#lang racket
(provide c)
(define c 1)
```

then

```
raco make a.rkt
```

triggers the creation of `"compiled/a_rkt.zo"`, `"compiled/b_rkt.zo"`, and `"compiled/c_rkt.zo"`. A subsequent

```
racket a.rkt
```

loads bytecode from the generated `".zo"` files, paying attention to the `".rkt"` sources only to confirm that each `".zo"` file has a later timestamp.

In contrast,

```
raco make b.rkt c.rkt
```

would create only `"compiled/b_rkt.zo"` and `"compiled/c_rkt.zo"`, since neither `"b.rkt"` nor `"c.rkt"` imports `"a.rkt"`.

## 1.2 Dependency Files

In addition to a bytecode file, `raco make` creates a file `"compiled/`⟨*name*⟩`_`⟨*ext*⟩`.dep"` that records dependencies of the compiled module on other module files and the source file's SHA-1 hash. Using this dependency information, a re-compilation request via `raco make` can consult both the source file's timestamp/hash and the timestamps/hashes for the bytecode of imported modules. Furthermore, imported modules are themselves compiled as necessary, including updating the bytecode and dependency files for the imported modules, transitively.

Continuing the `raco make a.rkt` example from the previous section, the `raco make` command creates `"compiled/a_rkt.dep"`, `"compiled/b_rkt.dep"`, and `"compiled/c_rkt.dep"` at the same time as the `".zo"` files. The `"compiled/a_rkt.dep"` file records the dependency of `"a.rkt"` on `"b.rkt"`, `"c.rkt"` and the `racket` library. If the `"b.rkt"` file is modified (so that its timestamp and SHA-1 hash changes), then running

```
raco make a.rkt
```

again rebuilds `"compiled/a_rkt.zo"` and `"compiled/b_rkt.zo"`.

For module files that are within library collections, `raco setup` uses the same `".zo"` and `".dep"` conventions and files as `raco make`, so the two tools can be used together.

## 1.3 API for Making Bytecode

`(require compiler/cm)` package: `base`

The `compiler/cm` module implements the compilation and dependency management used by `raco make` and `raco setup`.

```
(make-compilation-manager-load/use-compiled-handler
 [delete-zos-when-rkt-file-does-not-exist?
  #:security-guard security-guard])
→ (path? (or/c symbol? false/c) . -> . any)
 delete-zos-when-rkt-file-does-not-exist? : any/c = #f
 security-guard : (or/c security-guard? #f) = #f
```

Returns a procedure suitable as a value for the `current-load/use-compiled` parameter. The returned procedure passes its arguments on to the `current-load/use-compiled` procedure that is installed when `make-compilation-manager-load/use-compiled-handler` is called, but first it automatically compiles a source file to a `".zo"` file if

- the file is expected to contain a module (i.e., the second argument to the handler is a symbol);
- the value of each of `(current-eval)`, `(current-load)`, and `(namespace-module-registry (current-namespace))` is the same as when `make-compilation-manager-load/use-compiled-handler` was called;
- the value of `use-compiled-file-paths` contains the first path that was present when `make-compilation-manager-load/use-compiled-handler` was called;
- the value of `current-load/use-compiled` is the result of this procedure; and
- one of the following holds:

- the source file is newer than the ".zo" file in the first sub-directory listed in use-compiled-file-paths (at the time that make-compilation-manager-load/use-compiled-handler was called), and either no ".dep" file exists or it records a source-file SHA-1 hash that differs from the current version and source-file SHA-1 hash;
- no ".dep" file exists next to the ".zo" file;
- the version recorded in the ".dep" file does not match the result of (version);
- one of the files listed in the ".dep" file has a ".zo" timestamp newer than the target ".zo", and the combined hashes of the dependencies recorded in the ".dep" file does not match the combined hash recorded in the ".dep" file.

If SHA-1 hashes override a timestamp-based decision to recompile the file, then the target ".zo" file's timestamp is updated to the current time.

After the handler procedure compiles a ".zo" file, it creates a corresponding ".dep" file that lists the current version and the identification of every file that is directly required by the module in the compiled file. Additional dependencies can be installed during compilation via compiler/cm-accomplice. The ".dep" file also records the SHA-1 hash of the module's source, and it records a combined SHA-1 hash of all of the dependencies that includes their recursive dependencies.

The handler caches timestamps when it checks ".dep" files, and the cache is maintained across calls to the same handler. The cache is not consulted to compare the immediate source file to its ".zo" file, which means that the caching behavior is consistent with the caching of the default module name resolver (see current-module-name-resolver).

If use-compiled-file-paths contains an empty list when make-compilation-manager-load/use-compiled-handler is called, then exn:fail:contract exception is raised.

If the *delete-zos-when-rkt-file-does-not-exist?* argument is a true value, then the returned handler will delete ".zo" files when there is no corresponding original source file.

If the *security-guard* argument is supplied, it is used when creating ".zo" files, ".dep" files, and "compiled/" directories, and when it adjusts the timestamps for existing files. If it is #f, then the security guard in the current-security-guard when the files are created is used (not the security guard at the point make-compilation-manager-load/use-compiled-handler is called).

The continuation the compilation of a module is marked with a managed-compiled-context-key and the module's source path.

*Do not* install the result of make-compilation-manager-load/use-compiled-handler when the current namespace contains already-loaded versions of modules that may need to be recompiled—unless the already-loaded modules are never referenced by not-yet-loaded

modules. References to already-loaded modules may produce compiled files with inconsistent timestamps and/or "`.dep`" files with incorrect information.

The handler logs messages to the topic `'compiler/cm` at the level `'info`. These messages are instances of a `compile-event` prefab structure:

```
(struct compile-event (timestamp path type) #:prefab)
```

The `timestamp` field is the time at which the event occured in milliseconds since the epoch. The `path` field is the path of a file being compiled for which the event is about. The `type` field is a symbol which describes the action the event corresponds to. The currently logged values are `'locking`, `'start-compile`, `'finish-compile`, and `'already-done`.

Changed in version 6.1.1.8 of package `base`: Added identification of the compilation context via `managed-compiled-context-key`.

```
(managed-compile-zo  file
                     [read-src-syntax
                      #:security-guard security-guard]) → void?
  file : path-string?
  read-src-syntax : (any/c input-port? . -> . syntax?)
                  = read-syntax
  security-guard : (or/c security-guard? #f) = #f
```

Compiles the given module source file to a "`.zo`", installing a compilation-manager handler while the file is compiled (so that required modules are also compiled), and creating a "`.dep`" file to record the timestamps of immediate files used to compile the source (i.e., files `required` in the source).

If *`file`* is compiled from source, then *`read-src-syntax`* is used in the same way as `read-syntax` to read the source module. The normal `read-syntax` is used for any required files, however.

If *`security-guard`* is not `#f`, then the provided security guard is used when creating the "`compiled/`" directories, "`.dep`" and "`.zo`" files, and when it adjusts the timestamps of existing files. If it is `#f`, then the security guard in the `current-security-guard` when the files are created is used (not the security guard at the point `managed-compile-zo` is called).

While compiling *`file`*, the `error-display-handler` parameter is set to `(make-compilation-context-error-display-handler (error-display-handler))`, so that errors from uncaught exceptions will report the compilation context.

Changed in version 6.1.1.8 of package `base`: Added `error-display-handler` configuration.

```
managed-compiled-context-key : any/c
```

A key used as a continuation mark key by make-compilation-manager-load/use-compiled-handler for the continuation of a module compilation. The associated value is a path to the module's source.

Added in version 6.1.1.8 of package base.

```
(make-compilation-context-error-display-handler orig-handlers)
 → (string? any/c . -> . void?)
  orig-handlers : (string? any/c . -> . void?)
```

Produces a handler suitable for use as a error-display-handler value, given an existing such value. The generated handler shows information about the compilation context when the handler's second argument is an exception whose continuation marks include managed-compiled-context-key keys.

Added in version 6.1.1.8 of package base.

```
(trust-existing-zos) → boolean?
(trust-existing-zos trust?) → void?
  trust? : any/c
```

A parameter that is intended for use by setup-plt when installing with pre-built ".zo" files. It causes a compilation-manager load/use-compiled handler to "touch" out-of-date ".zo" files instead of re-compiling from source.

```
(make-caching-managed-compile-zo
 [read-src-syntax
  #:security-guard security-guard])
→ (path-string? . -> . void?)
 read-src-syntax : (any/c input-port? . -> . syntax?)
                 = read-syntax
 security-guard : (or/c security-guard? #f) = #f
```

Returns a procedure that behaves like managed-compile-zo (providing the same *read-src-syntax* each time), but a cache of timestamp information is preserved across calls to the procedure.

```
(manager-compile-notify-handler) → (path? . -> . any)
(manager-compile-notify-handler notify) → void?
  notify : (path? . -> . any)
```

A parameter for a procedure of one argument that is called whenever a compilation starts. The argument to the procedure is the file's path.

```
(manager-trace-handler) → (string? . -> . any)
(manager-trace-handler notify) → void?
  notify : (string? . -> . any)
```

A parameter for a procedure of one argument that is called to report compilation-manager actions, such as checking a file. The argument to the procedure is a string.

The default value of the parameter logs the argument, along with `current-inexact-milliseconds` to, a logger named `'compiler/cm` at the `'debug` level.

```
(manager-skip-file-handler)
 → (-> path? (or/c (cons/c number? promise?) #f))
(manager-skip-file-handler proc) → void?
  proc : (-> path? (or/c (cons/c number? promise?) #f))
```

A parameter whose value is called for each file that is loaded and needs recompilation. If the procedure returns a pair, then the file is skipped (i.e., not compiled); the number in the pair is used as the timestamp for the file's bytecode, and the promise may be `force`d to obtain a string that is used as hash of the compiled file plus its dependencies. If the procedure returns `#f`, then the file is compiled as usual. The default is `(lambda (x) #f)`.

```
(file-stamp-in-collection p)
 → (or/c (cons/c number? promise?) #f)
  p : path?
```

Calls `file-stamp-in-paths` with *p* and `(current-library-collection-paths)`.

```
(file-stamp-in-paths p paths)
 → (or/c (cons/c number? promise?) #f)
  p : path?
  paths : (listof path?)
```

Returns the file-modification date and `delay`ed hash of *p* or its bytecode form (i.e., `".zo"` file), whichever exists and is newer, if *p* is an extension of any path in *paths* (i.e., exists in the directory, a subdirectory, etc.). Otherwise, the result is `#f`.

This function is intended for use with `manager-skip-file-handler`.

```
(get-file-sha1 p) → (or/c string? #f)
  p : path?
```

Computes a SHA-1 hash for the file *p*; the result is `#f` if *p* cannot be opened.

```
(get-compiled-file-sha1 p) → (or/c string? #f)
  p : path?
```

Computes a SHA-1 hash for the bytecode file *p*, appending any dependency-describing hash available from a `".dep"` file when available (i.e., the suffix on *p* is replaced by `".dep"` to locate dependency information). The result is `#f` if *p* cannot be opened.

```
(with-compile-output p proc) → any
  p : path-string?
  proc : ([port input-port?] [tmp-path path?]  . -> . any)
```

A wrapper on `call-with-atomic-output-file` that passes along any security guard put in place by `make-compilation-manager-load/use-compiled-handler`, etc.

```
(parallel-lock-client)
 → (or/c #f
         (->i ([command (or/c 'lock 'unlock)]
               [file bytes?])
              [res (command) (if (eq? command 'lock)
                                 boolean?
                                 void?)]))
(parallel-lock-client proc) → void?
  proc : (or/c #f
               (->i ([command (or/c 'lock 'unlock)]
                     [file bytes?])
                    [res (command) (if (eq? command 'lock)
                                       boolean?
                                       void?)]))
```

Holds the parallel compilation lock client, which is used by the result of `make-compilation-manager-load/use-compiled-handler` to prevent compilation races between parallel builders.

When *proc* is `#f` (the default), no checking for parallel compilation is done (and thus multiple threads or places running compilations via `make-compilation-manager-load/use-compiled-handler` will potentially corrupt each other's `".zo"` files).

When *proc* is a function, its first argument is a command, indicating if it wants to lock or unlock the path specified in the second argument.

When the *proc* `'lock` command returns `#t`, the current builder has obtained the lock for `zo-path`. Once compilation of `zo-path` is complete, the builder process must release the lock by calling *proc* `'unlock` with the exact same `zo-path`.

When the *proc* `'lock` command returns `#f`, another parallel builder obtained the lock first and has already compiled the zo. The parallel builder should continue without compiling `zo-path`. (In this case, `make-compilation-manager-load/use-compiled-handler`'s result will not call *proc* with `'unlock`.)

Example:

```
> (let* ([lc (parallel-lock-client)]
         [zo-name  #"collects/racket/compiled/draw_rkt.zo"]
```

```
          [locked? (and lc (lc 'lock zo-name))]
          [ok-to-compile? (or (not lc) locked?)])
     (dynamic-wind
       (lambda () (void))
       (lambda ()
         (when ok-to-compile?
           (printf "Do compile here ...\n")))
       (lambda ()
         (when locked?
           (lc 'unlock zo-name)))))
Do compile here ...
```

```
(compile-lock->parallel-lock-client  pc
                                     [cust])
 → (-> (or/c 'lock 'unlock) bytes? boolean?)
  pc : place-channel?
  cust : (or/c #f custodian?) = #f
```

Returns a function that follows the `parallel-lock-client` by communicating over *pc*. The argument must have be the result of `make-compile-lock`.

This communication protocol implementation is not kill safe. To make it kill safe, it needs a sufficiently powerful custodian, i.e., one that is not subject to termination (unless all of the participants in the compilation are also terminated). It uses this custodian to create a thread that monitors the threads that are doing the compilation. If one of them is terminated, the presence of the custodian lets another one continue. (The custodian is also used to create a thread that manages a thread safe table.)

```
(make-compile-lock) → place-channel?
```

Creates a `place-channel?` that can be used with `compile-lock->parallel-lock-client` to avoid concurrent compilations of the same racket source files in multiple places.

```
(install-module-hashes! bstr [start end]) → void?
  bstr : btyes?
  start : exact-nonnegatve-integer? = 0
  end : exact-nonnegatve-integer? = (bytes-length bstr)
```

Adjusts the bytecode representation in *bstr* (from bytes *start* to *end*) to install a hash code, including any submodules within the region. The existing representation should have zero bytes in place of each hash string, which is what `write` produces for a compiled form.

Added in version 6.3 of package `base`.

## 1.4 API for Parallel Builds

```
(require setup/parallel-build)
```
package: base

The setup/parallel-build library provides the parallel-compilation functionality of raco setup and raco make.

Both parallel-compile-files and parallel-compile log messages to the topic 'setup/parallel-build at the level 'info. These messages are instances of a parallel-compile-event prefab structure:

```
(struct parallel-compile-event (worker event) #:prefab)
```

The worker field is the index of the worker that the created the event. The event field is a compile-event as document in make-compilation-manager-load/use-compiled-handler.

```
(parallel-compile-files list-of-files
                        [#:worker-count worker-count
                         #:handler handler])
 → (or/c void? #f)
  list-of-files : (listof path-string?)
  worker-count : exact-positive-integer? = (processor-count)
  handler : (->i ([worker-id exact-integer?] = void
                  [handler-type symbol?]
                  [path path-string?]
                  [msg string?]
                  [out string?]
                  [err string?])
                 void?)
```

The parallel-compile-files utility function is used by raco make to compile a list of paths in parallel. The optional #:worker-count argument specifies the number of compile workers to spawn during parallel compilation. The callback, *handler*, is called with the symbol 'done as the *handler-type* argument for each successfully compiled file, 'output when a successful compilation produces stdout/stderr output, 'error when a compilation error has occured, or 'fatal-error when a unrecoverable error occurs. The other arguments give more information for each status update. The return value is (void) if it was successful, or #f if there was an error.

```
(parallel-compile-files
  source-files
  #:worker-count 4
  #:handler (lambda (type work msg out err)
    (match type
      ['done (when (verbose) (printf " Made ~a\n" work))]
```

```
      ['output (printf " Output from: ~a\n~a~a" work out err)]
      [else (printf " Error compiling ~a\n~a\n~a~a"
                    work
                    msg
                    out
                    err)])))
```

```
(parallel-compile worker-count
                  setup-fprintf
                  append-error
                  collects-tree) → (void)
  worker-count : non-negative-integer?
  setup-fprintf : (->i ([stage string?] [format string?])
                       ()
                       #:rest (listof any/c) void)
  append-error : (->i ([cc cc?]
                       [prefix string?]
                       [exn (or/c exn? (cons/c string? string?) #f)]
                       [out string?]
                       [err srtring?]
                       [message string?])
                      void?)
  collects-tree : (listof any/c)
```

The `parallel-compile` function is used by `raco setup` to compile collections in parallel. The *worker-count* argument specifies the number of compilation workers to spawn during parallel compilation. The *setup-fprintf* and *append-error* functions communicate intermediate compilation results and errors. The *collects-tree* argument is a compound datastructure containing an in-memory tree representation of the collects directory.

When the *exn* argument to *append-error* is a part of strings, the first string is a long form of the error message, and the second string is a short form (omitting evaluation context information, for example).

Changed in version 6.1.1.8 of package `base`: Changed *append-error* to allow a pair of error strings.

## 1.5 Compilation Manager Hook for Syntax Transformers

```
(require compiler/cm-accomplice)      package: base
```

```
(register-external-file file
                        [#:indirect? indirect?]) → void?
  file : (and path? complete-path?)
  indirect? : any/c = #f
```

Logs a message (see log-message) at level 'info to a logger named 'cm-accomplice. The message data is a file-dependency prefab structure type with two fields; the first field's value is *file* and the second field's value is #f (to indicate a non-module dependency). If the *indirect?* argument is true, the data is more specifically an instance of a file-dependency/options prefab structure type that is a subtype of file-dependency with one extra field: a hash table mapping 'indirect to #t.

A compilation manager implemented by compiler/cm looks for such messages to register an external dependency. In response, the compilation manager records (in a ".dep" file) the path as contributing to the implementation of the module currently being compiled. Afterward, if the registered file is modified, the compilation manager will know to recompile the module. An indirect dependency has no effect on recompilation, but it can signal to other tools, such as a package-dependency checker, that the dependency is indirect (and should not imply a direct package dependency).

The include macro, for example, calls this procedure with the path of an included file as it expands an include form.

```
(register-external-module file
                          [#:indirect? indirect?]) → void?
  file : (and path? complete-path?)
  indirect? : any/c = #f
```

Like register-external-file, but logs a message with a file-dependency prefab structure type whose second field is #t.

A compilation manager implemented by compiler/cm recognizes the message to register a dependency on a module (which implies a dependency on all of that module's dependencies, etc.).

## 1.6 API for Simple Bytecode Creation

```
(require compiler/compile-file)
```
package: base

```
(compile-file src [dest filter]) → path?
  src : path-string?
  dest : path-string?
       = (let-values ([(base name dir?) (split-path src)])
           (build-path base "compiled"
                       (path-add-suffix name #".zo")))
  filter : (any/c . -> . any/c) = values
```

Compiles the Racket file *src* and saves the compiled code to *dest*. If *dest* is not provided and the "compiled" subdirectory does not already exist, the subdirectory is created. The result of compile-file is the destination file's path.

If the *filter* procedure is provided, it is applied to each source expression, and the result is compiled.

The compile-file procedure is designed for compiling modules files, in that each expression in *src* is compiled independently. If *src* does not contain a single module expression, then earlier expressions can affect the compilation of later expressions when *src* is loaded directly. An appropriate *filter* can make compilation behave like evaluation, but the problem is also solved (as much as possible) by the compile-zos procedure.

See also managed-compile-zo.

## 1.7 API for Bytecode Paths

(require compiler/compilation-path) package: base

Added in version 6.0.1.10 of package base.

```
(get-compilation-dir+name path
                          [#:modes modes
                           #:roots roots]) → path? path?
  path : path-string?
  modes : (non-empty-listof (and/c path-string? relative-path?))
        = (use-compiled-file-paths)
  roots : (non-empty-listof (or/c path-string? 'same))
        = (current-compiled-file-roots)
```

Determines the directory that holds the bytecode form of *path* plus base name of *path*.

The directory is determined by checking *roots* in order, and for each element of *roots* checking *modes* in order. The first such directory that contains a file whose name matches *path* with ".zo" added (in the sense of path-add-suffix) is reported as the return directory path. If no such file is found, the result corresponds to the first elements of *modes* and *roots*.

```
(get-compilation-dir path
                     [#:modes modes
                      #:roots roots]) → path?
  path : path-string?
  modes : (non-empty-listof (and/c path-string? relative-path?))
        = (use-compiled-file-paths)
  roots : (non-empty-listof (or/c path-string? 'same))
        = (current-compiled-file-roots)
```

The same as get-compilation-dir+home, but returning only the first result.

```
(get-compilation-bytecode-file path
                               [#:modes modes
                                #:roots roots]) → path?
  path : path-string?
  modes : (non-empty-listof (and/c path-string? relative-path?))
        = (use-compiled-file-paths)
  roots : (non-empty-listof (or/c path-string? 'same))
        = (current-compiled-file-roots)
```

The same as `get-compilation-dir+home`, but combines the results and adds a `".zo"` suffix to arrive at a bytecode file path.

## 1.8 Compiling to Raw Bytecode

The `--no-deps` mode for `raco make` is an improverished form of the compilation, because it does not track import dependencies. It does, however, support compilation of non-module source in an namespace that initially imports `scheme`.

Outside of a module, top-level `define-syntaxes`, `module`, `#%require`, `define-values-for-syntax`, and `begin` expressions are handled specially by `raco make --no-deps`: the compile-time portion of the expression is evaluated, because it might affect later expressions.

For example, when compiling the file containing

```
(require racket/class)
(define f (class object% (super-new)))
```

the `class` form from the `racket/class` library must be bound in the compilation namespace at compile time. Thus, the `require` expression is both compiled (to appear in the output code) and evaluated (for further computation).

Many definition forms expand to `define-syntaxes`. For example, `define-signature` expands to `define-syntaxes`. In `--no-deps` mode, `raco make --no-deps` detects `define-syntaxes` and other expressions after expansion, so top-level `define-signature` expressions affect the compilation of later expressions, as a programmer would expect.

In contrast, a `load` or `eval` expression in a source file is compiled—but *not evaluated!*—as the source file is compiled. Even if the `load` expression loads syntax or signature definitions, these will not be loaded as the file is compiled. The same is true of application expressions that affect the reader, such as `(read-case-sensitive #t)`. The `-p` or `--prefix` flag for `raco make` takes a file and loads it before compiling the source files specified on the command line.

By default, the namespace for compilation is initialized by a `require` of `scheme`. If

the `--no-prim` flag is specified, the namespace is instead initialized with `namespace-require/copy`, which allows mutation and redefinition of all initial bindings (other than syntactic forms, in the case of mutation).

In general, a better solution is to put all code to compile into a module and use `raco make` in its default mode.

# 2 raco exe: Creating Stand-Alone Executables

Use a smaller base language to achieve a faster startup time such as #lang racket/base instead of #lang racket rather than relying on raco exe.

Compiled code produced by raco make relies on Racket executables to provide run-time support to the compiled code. However, raco exe can package code together with its run-time support to form an executable, and raco distribute can package the executable into a distribution that works on other machines. Running an executable produced by raco exe will not improve performance over raco make.

The raco exe command embeds a module, from source or byte code, into a copy of the racket executable. (On Unix, the embedding executable is actually a copy of a wrapper executable.) The created executable invokes the embedded module on startup. The --gui flag causes the program to be embedded in a copy of the gracket executable. If the embedded module refers to other modules via require, then the other modules are also included in the embedding executable.

For example, the command

```
raco exe --gui hello.rkt
```

produces either "hello.exe" (Windows), "hello.app" (Mac OS X), or "hello" (Unix), which runs the same as running the "hello.rkt" module in gracket.

Library modules or other files that are referenced dynamically—through eval, load, or dynamic-require—are not automatically embedded into the created executable. Such modules can be explicitly included using the ++lib flag to raco exe. Alternately, use define-runtime-path to embed references to the run-time files in the executable; the files are then copied and packaged together with the executable when creating a distribution (as described in §3 "raco distribute: Sharing Stand-Alone Executables"). Finally, a submodule is included if its enclosing module is included and the submodule contains a sub-submodule named declare-preserve-for-embedding (where the implementation of the sub-submodule is ignored).

Modules that are implemented directly by extensions—i.e., extensions that are automatically loaded from (build-path "compiled" "native" (system-library-subpath)) to satisfy a require—are treated like other run-time files: a generated executable uses them from their original location, and they are copied and packaged together when creating a distribution.

The raco exe command works only with module-based programs. The compiler/embed library provides a more general interface to the embedding mechanism.

A stand-alone executable is "stand-alone" in the sense that you can run it without starting racket, gracket, or DrRacket. However, the executable depends on Racket shared libraries, and possibly other run-time files declared via define-runtime-path. The executable can be packaged with support libraries to create a distribution using raco distribute, as described in §3 "raco distribute: Sharing Stand-Alone Executables".

The `raco exe` command accepts the following command-line flags:

- `-o` ⟨*file*⟩ — create the executable as ⟨*file*⟩, adding a suffix to ⟨*file*⟩ as appropriate for the platform and executable type. On Mac OS X in `--gui` mode, ⟨*file*⟩ is actually a bundle directory, but it appears as a file within Finder.

- `--gui` — create a graphical executable based on `gracket` instead of `racket`.

- `-l` or `--launcher` — create a launcher (see §2.2 "Installation-Specific Launchers"), instead of a stand-alone executable. Flags such as `--config-path`, `--collects-path`, and `--lib` have no effect on launchers.

- `--config-path` ⟨*path*⟩ — set ⟨*path*⟩ within the executable as the path to the configuration directory; if the path is relative, it will be treated as relative to the executable. The default path is `"etc"`, with the expectation that no such directory will exist at run time.

- `--collects-path` ⟨*path*⟩ — set ⟨*path*⟩ within the executable as the path to the main collection directory; if the path is relative, it will be treated as relative to the executable. The default is to have no path, which means that the `current-library-collection-paths` and `current-library-collection-links` parameters are initialized at empty when the executable starts. Beware that various other directories are located relative to the main collection directory by default (see §19 "Installation Configuration and Search Paths"), so that installing ⟨*path*⟩ may allow other directories to be found—intentional or not.

- `--collects-dest` ⟨*path*⟩ — write modules to be included with the executable into ⟨*path*⟩ (relative to the current directory), instead of embedded within the executable. The `--collects-dest` flag normally makes sense only in combination with `--collects-path`. This mode currently does not prune unreferenced submodules (and it pulls along any dependencies of submodules).

- `--ico` ⟨*.ico-path*⟩ — on Windows, set the icons for the generated executable to ones extracted from ⟨*.ico-path*⟩; see `create-embedding-executable`'s use of the `'ico` auxiliary association for more information about expected icon sizes and transformations.

- `--icns` ⟨*.icns-path*⟩ — on Mac OS X, set the icons for the generated executable to be the content of ⟨*.icns-path*⟩.

- `--orig-exe` — generate an executable that refers to the original `racket` or `gracket` executable, instead of making a copy. This flag is rarely useful, because the part of the executable that is copied is normally small, and `raco distribute` does not work with executables that are created with `--orig-exe`.

- `--3m` — generate an executable based on the 3m variant of Racket, which is the default unless running a `raco exe` that is based on the CGC variant.

- `--cgc` — generate an executable based on the CGC variant of Racket, which is the default only when running a `raco exe` that is based on the CGC variant.
- `++aux` ⟨*file*⟩ — attach information to the executable based on ⟨*file*⟩'s suffix; see `extract-aux-from-path` for a list of recognized suffixes and meanings, and see `create-embedding-executable`'s use of auxiliary association for more specific information about how each kind of file is used.
- `++lib` ⟨*module-path*⟩ — include ⟨*module-path*⟩ in the executable, even if it is not referenced by the main program, so that it is available via `dynamic-require`.
- `++exfl` ⟨*flag*⟩ — provide the ⟨*flag*⟩ command-line argument on startup to the embedded `racket` or `gracket`.
- `--exfl` ⟨*flag*⟩ — remove ⟨*flag*⟩ from the command-line arguments to be provided on startup to the embedded `racket` or `gracket`.
- `--exfl-clear` — remove all command-line arguments to be provided on startup to the embedded `racket` or `gracket`.
- `--exfl-show` — show (without changing) the command-line arguments to be provided on startup to the embedded `racket` or `gracket`.
- `-v` — report progress verbosely.
- `--vv` — report progress more verbosely than `-v`.

Changed in version 6.3.0.11: Added support for `declare-preserve-for-embedding`.

## 2.1 API for Creating Executables

```
(require compiler/embed)
```
package: `base`

The `compiler/embed` library provides a function to embed Racket code into a copy of Racket or GRacket, thus creating a stand-alone Racket executable. To package the executable into a distribution that is independent of your Racket installation, use `assemble-distribution` from `compiler/distribute`.

Embedding walks the module dependency graph to find all modules needed by some initial set of top-level modules, compiling them if needed, and combining them into a "module bundle." In addition to the module code, the bundle extends the module name resolver, so that modules can be `required` with their original names, and they will be retrieved from the bundle instead of the filesystem.

The `create-embedding-executable` function combines the bundle with an executable (Racket or GRacket). The `write-module-bundle` function prints the bundle to the current output port, instead; this stream can be `load`ed directly by a running program, as long as the `read-accept-compiled` parameter is true.

```
(create-embedding-executable
 dest
 #:modules mod-list
 [#:early-literal-expressions early-literal-sexps
 #:configure-via-first-module? config-via-first?
 #:literal-files literal-files
 #:literal-expression literal-sexp
 #:literal-expressions literal-sexps
 #:cmdline cmdline
 #:gracket? gracket?
 #:mred? mred?
 #:variant variant
 #:aux aux
 #:collects-path collects-path
 #:collects-dest collects-dest
 #:launcher? launcher?
 #:verbose? verbose?
 #:expand-namespace expand-namespace
 #:compiler compile-proc
 #:src-filter src-filter
 #:on-extension ext-proc
 #:get-extra-imports extras-proc])
→ void?
 dest : path-string?
 mod-list : (listof (or/c (list/c (or/c symbol? (one-of/c #t #f))
                                  (or/c module-path? path?))
                          (list/c (or/c symbol? (one-of/c #t #f))
                                  (or/c module-path? path?)
                                  (listof symbol?))))
 early-literal-sexps : list? = null
 config-via-first? : any/c = #f
 literal-files : (listof path-string?) = null
 literal-sexp : any/c = #f
 literal-sexps : list? = (if literal-sexp
                             (list literal-sexp)
                             null)
 cmdline : (listof string?) = null
 gracket? : any/c = #f
 mred? : any/c = #f
 variant : (or/c 'cgc '3m) = (system-type 'gc)
 aux : (listof (cons/c symbol? any/c)) = null
 collects-path : (or/c #f                      = #f
                       path-string?
                       (listof path-string?))
 collects-dest : (or/c #f path-string?) = #f
```

```
launcher? : any/c = #f
verbose? : any/c = #f
expand-namespace : namespace? = (current-namespace)
compile-proc : (any/c . -> . compiled-expression?)
             = (lambda (e)
                 (parameterize ([current-namespace
                                 expand-namespace])
                   (compile e)))
src-filter : (path? . -> . any) = (lambda (p) #t)
ext-proc : (or/c #f (path-string? boolean? . -> . any)) = #f
extras-proc : (path? compiled-module-expression?
               . -> . (listof module-path?))
            = (lambda (p m) null)
```

Copies the Racket (if *gracket?* and *mred?* are `#f`) or GRacket (otherwise) binary, embedding code into the copied executable to be loaded on startup. On Unix, the binary is actually a wrapper executable that execs the original; see also the `'original-exe?` tag for *aux*.

The embedding executable is written to *dest*, which is overwritten if it exists already (as a file or directory).

The embedded code consists of module declarations followed by additional (arbitrary) code. When a module is embedded, every module that it imports is also embedded. Library modules are embedded so that they are accessible via their `lib` paths in the initial namespace except as specified in *mod-list*, other modules (accessed via local paths and absolute paths) are embedded with a generated prefix, so that they are not directly accessible.

The `#:modules` argument *mod-list* designates modules to be embedded, as described below. The `#:early-literal-expressions`, `#:literal-files`, and `#:literal-expressions` arguments specify literal code to be copied into the executable: each element of *early-literal-sexps* is copied in order, then the content of each file in *literal-files* in order (with no intervening spaces), and then each element of *literal-sexps*. The *literal-files* files or *early-literal-sexps* or *literal-sexps* lists can contain compiled bytecode, and it's possible that the content of the *literal-files* files only parse when concatenated; the files and expression are not compiled or inspected in any way during the embedding process. Beware that the initial namespace contains no bindings; use compiled expressions to bootstrap the namespace. The `#:literal-expression` (singular) argument is for backward compatibility.

If the `#:configure-via-first-module?` argument is specified as true, then the language of the first module in *mod-list* is used to configure the run-time environment before the expressions added by `#:literal-files` and `#:literal-expressions` are evaluated, but after the expressions of `#:early-literal-expressions`. See also §18.1.5 "Language Run-Time Configuration".

The `#:cmdline` argument *cmdline* contains command-line strings that are prefixed onto

any actual command-line arguments that are provided to the embedding executable. A command-line argument that evaluates an expression or loads a file will be executed after the embedded code is loaded.

Each element of the `#:modules` argument *`mod-list`* is a two- or three-item list, where the first item is a prefix for the module name, and the second item is a module path datum (that's in the format understood by the default module name resolver), and the third is a list of submodule names to be included if they are available. The prefix can be a symbol, `#f` to indicate no prefix, or `#t` to indicate an auto-generated prefix. For example,

```
'((#f "m.rkt"))
```

embeds the module `m` from the file `"m.rkt"`, without prefixing the name of the module; the `literal-sexpr` argument to go with the above might be `'(require m)`. When submodules are available and included, the submodule is given a name by symbol-appending the `write` form of submodule path to the enclosing module's name.

Modules are normally compiled before they are embedded into the target executable; see also `#:compiler` and `#:src-filter` below. When a module declares run-time paths via `define-runtime-path`, the generated executable records the path (for use both by immediate execution and for creating a distribution that contains the executable).

If *`collects-dest`* is a path instead of `#f`, then instead of embedding collection-based modules into the executable, the modules (in compiled form, only) are copied into collections in the *`collects-dest`* directory.

The optional `#:aux` argument is an association list for platform-specific options (i.e., it is a list of pairs where the first element of the pair is a key symbol and the second element is the value for that key). See also `build-aux-from-path`. The currently supported keys are as follows:

- `'icns` (Mac OS X) : An icon file path (suffix `".icns"`) to use for the executable's desktop icon.
- `'ico` (Windows) : An icon file path (suffix `".ico"`) to use for the executable's desktop icon.

  Changed in version 6.3 of package `base`: All icons in the executable are replaced with icons from the file, instead of setting only certain sizes and depths.
- `'creator` (Mac OS X) : Provides a 4-character string to use as the application signature.
- `'file-types` (Mac OS X) : Provides a list of association lists, one for each type of file handled by the application; each association is a two-element list, where the first (key) element is a string recognized by Finder, and the second element is a plist value (see `xml/plist`). See `"drracket.filetypes"` in the `"drracket"` collection for an example.

- `'uti-exports` (Mac OS X) : Provides a list of association lists, one for each Uniform Type Identifier (UTI) exported by the executable; each association is a two-element list, where the first (key) element is a string recognized in a UTI declaration, and the second element is a plist value (see `xml/plist`). See `"drracket.utiexports"` in the `"drracket"` collection for an example.

- `'resource-files` (Mac OS X) : extra files to copy into the `"Resources"` directory of the generated executable.

- `'config-dir` : A string/path to a directory that contains configuration information, such as `"config.rtkd"` (see §19 "Installation Configuration and Search Paths"). If no value is supplied, the path is left as-is and converted to absolute form as needed. If `#f` is supplied, the path is left as-is (in potentially relative form). Note that if *`collects-path`* is provided as an empty list, then the configuration-directory path is not used by Racket's start up process (in contrast to a normal Racket start-up, where the configuration directory is consulted for information about collection link files).

- `'framework-root` (Mac OS X) : A string to prefix the executable's path to the Racket and GRacket frameworks (including a separating slash); note that when the prefix starts `"@executable_path/"` works for a Racket-based application, the corresponding prefix start for a GRacket-based application is `"@executable_path/../../../"`; if `#f` is supplied, the executable's framework path is left as-is, otherwise the original executable's path to a framework is converted to an absolute path if it was relative.

- `'dll-dir` (Windows) : A string/path to a directory that contains Racket DLLs needed by the executable, such as `"racket`⟨*version*⟩`.dll"`, or a boolean; a path can be relative to the executable; if `#f` is supplied, the path is left as-is; if `#t` is supplied, the path is dropped (so that the DLLs must be in the system directory or the user's `PATH`); if no value is supplied the original executable's path to DLLs is converted to an absolute path if it was relative.

- `'subsystem` (Windows) : A symbol, either `'console` for a console application or `'windows` for a consoleless application; the default is `'console` for a Racket-based application and `'windows` for a GRacket-based application; see also `'single-instance?`, below.

- `'single-instance?` (Windows) : A boolean for GRacket-based apps; the default is `#t`, which means that the app looks for instances of itself on startup and merely brings the other instance to the front; `#f` means that multiple instances are expected.

- `'forget-exe?` (Windows, Mac OS X) : A boolean; `#t` for a launcher (see *`launcher?`* below) does not preserve the original executable name for `(find-system-path 'exec-file)`; the main consequence is that library collections will be found relative to the launcher instead of the original executable.

- `'original-exe?` (Unix) : A boolean; `#t` means that the embedding uses the original Racket or GRacket executable, instead of a wrapper binary that execs the original; the default is `#f`.

- `'relative?` (Unix, Windows, Mac OS X) : A boolean; `#t` means that, to the degree that the generated executable must refer to another, it can use a relative path (so the executables can be moved together, but not separately), and it implies `#f` for `'config-dir`, `'framework-dir`, and `'dll-dir`, unless those are explicitly provided; a `#f` value (the default) means that absolute paths should be used (so the generated executable can be moved).
- `'wm-class` (Unix) : A string; used as the default `WM_CLASS` program class for the program's windows.

If the `#:collects-path` argument is `#f`, then the created executable maintains its built-in (relative) path to the main `"collects"` directory—which will be the result of `(find-system-path 'collects-dir)` when the executable is run—plus a potential list of other directories for finding library collections—which are used to initialize the `current-library-collection-paths` list in combination with `PLTCOLLECTS` environment variable. Otherwise, the argument specifies a replacement; it must be either a path, string, or list of paths and strings. In the last case, the first path or string specifies the main collection directory, and the rest are additional directories for the collection search path (placed, in order, after the user-specific `"collects"` directory, but before the main `"collects"` directory; then the search list is combined with `PLTCOLLECTS`, if it is defined). If the list is empty, then `(find-system-path 'collects-dir)` will return the directory of the executable, but `current-library-collection-paths` is initialized to an empty list, and `use-collection-link-paths` is set to false to disable the use of collection links files.

If the `#:launcher?` argument is `#t`, then `lid-list` should be null, *`literal-files`* should be null, *`literal-sexp`* should be `#f`, and the platform should be Windows or Mac OS X. The embedding executable is created in such a way that `(find-system-path 'exec-file)` produces the source Racket or GRacket path instead of the embedding executable (but the result of `(find-system-path 'run-file)` is still the embedding executable).

The `#:variant` argument indicates which variant of the original binary to use for embedding. The default is `(system-type 'gc)`; see also `current-launcher-variant`.

The `#:compiler` argument is used to compile the source of modules to be included in the executable (when a compiled form is not already available). It should accept a single argument that is a syntax object for a `module` form. The default procedure uses `compile` parameterized to set the current namespace to *`expand-namespace`*.

The `#:expand-namespace` argument selects a namespace for expanding extra modules (and for compiling using the default *`compile-proc`*). Extra-module expansion is needed to detect run-time path declarations in included modules, so that the path resolutions can be directed to the current locations (and, ultimately, redirected to copies in a distribution).

The `#:src-filter` *`src-filter`* argument takes a path and returns true if the corresponding file source should be included in the embedding executable in source form (instead of compiled form), `#f` otherwise. The default returns `#f` for all paths. Beware that the current output port may be redirected to the result executable when the filter procedure is called.

Each path given to *src-filter* corresponds to the actual file name (e.g., ".ss"/".rkt" conversions have been applied as needed to refer to the existing file).

If the #:on-extension argument is a procedure, the procedure is called when the traversal of module dependencies arrives at an extension (i.e., a DLL or shared object). The default, #f, causes a reference to a single-module extension (in its current location) to be embedded into the executable. The procedure is called with two arguments: a path for the extension, and a #f (for historical reasons).

The #:get-extra-imports *extras-proc* argument takes a source pathname and compiled module for each module to be included in the executable. It returns a list of quoted module paths (absolute, as opposed to relative to the module) for extra modules to be included in the executable in addition to the modules that the source module requires. For example, these modules might correspond to reader extensions needed to parse a module that will be included as source, as long as the reader is referenced through an absolute module path. Each path given to *extras-proc* corresponds to the actual file name (e.g., ".ss"/".rkt" conversions have been applied as needed to refer to the existing file).

```
(make-embedding-executable  dest
                            mred?
                            verbose?
                            mod-list
                            literal-files
                            literal-sexp
                            cmdline
                           [aux
                            launcher?
                            variant
                            collects-path]) → void?
  dest : path-string?
  mred? : any/c
  verbose? : any/c
  mod-list : (listof (or/c (list/c (or/c symbol? (one-of/c #t #f))
                                   (or/c module-path? path?))
                           (list/c (or/c symbol? (one-of/c #t #f))
                                   (or/c module-path? path?)
                                   (listof symbol?))))
  literal-files : (listof path-string?)
  literal-sexp : any/c
  cmdline : (listof string?)
  aux : (listof (cons/c symbol? any/c)) = null
  launcher? : any/c = #f
  variant : (one-of/c 'cgc '3m) = (system-type 'gc)
  collects-path : (or/c #f                      = #f
                        path-string?
                        (listof path-string?))
```

Old (keywordless) interface to create-embedding-executable.

```
(write-module-bundle verbose?
                     mod-list
                     literal-files
                     literal-sexp) → void?
  verbose? : any/c
  mod-list : (listof (or/c (list/c (or/c symbol? (one-of/c #t #f))
                                   (or/c module-path? path?))
                           (list/c (or/c symbol? (one-of/c #t #f))
                                   (or/c module-path? path?)
                                   (listof symbol?))))
  literal-files : (listof path-string?)
  literal-sexp : any/c
```

Like make-embedding-executable, but the module bundle is written to the current output port instead of being embedded into an executable. The output of this function can be read to load and instantiate *mod-list* and its dependencies, adjust the module name resolver to find the newly loaded modules, evaluate the forms included from *literal-files*, and finally evaluate literal-sexpr. The read-accept-compiled parameter must be true to read the stream.

```
(embedding-executable-is-directory? mred?) → boolean
  mred? : any/c
```

Indicates whether Racket/GRacket executables for the current platform correspond to directories from the user's perspective. The result is currently #f for all platforms.

```
(embedding-executable-is-actually-directory? mred?) → boolean?
  mred? : any/c
```

Indicates whether Racket/GRacket executables for the current platform actually correspond to directories. The result is #t on Mac OS X when *mred?* is #t, #f otherwise.

```
(embedding-executable-put-file-extension+style+filters mred?)
 → (or/c string? false/c)
    (listof (one-of/c 'packages 'enter-packages))
    (listof (list/c string? string?))
  mred? : any/c
```

Returns three values suitable for use as the extension, style, and filters arguments to put-file, respectively.

If Racket/GRacket launchers for the current platform were directories form the user's perspective, the style result is suitable for use with get-directory, and the extension result may be a string indicating a required extension for the directory name.

```
(embedding-executable-add-suffix path
                                 mred?) → path-string?
  path : path-string?
  mred? : any/c
```

Adds a suitable executable suffix, if it's not present already.

### 2.1.1 Executable Creation Signature

```
(require compiler/embed-sig)
```
package: compiler-lib

```
compiler:embed^ : signature
```

Includes the identifiers provided by compiler/embed.

### 2.1.2 Executable Creation Unit

```
(require compiler/embed-unit)
```
package: compiler-lib

```
compiler:embed@ : unit?
```

A unit that imports nothing and exports compiler:embed^.

### 2.1.3 Finding the Racket Executable

```
(require compiler/find-exe)
```
package: base

```
(find-exe [#:cross? cross? gracket? variant]) → path?
  cross? : any/c = #f
  gracket? : any/c = #f
  variant : (or/c 'cgc '3m) = (if cross?
                                  (cross-system-type 'gc)
                                  (system-type 'gc))
```

Finds the path to the racket or gracket (when *gracket?* is true) executable.

If *cross?* is true, the executable is found for the target platform in cross-installation mode.

Changed in version 6.3 of package base: Added the #:cross? argument.

## 2.2 Installation-Specific Launchers

A *launcher* is similar to a stand-alone executable, but a launcher is usually smaller and can be created more quickly, because it depends permanently on the local Racket installation and the program's sources. In the case of Unix, a launcher is simply a shell script that runs `racket` or `gracket`. Launchers *cannot* be packaged into a distribution using `raco distribute`. The `raco exe` command creates a launcher when the `-l` or `--launcher` flag is specified.

```
(require launcher/launcher)
```
package: `base`

The `launcher/launcher` library provides functions for creating launchers.

### 2.2.1 Creating Launchers

```
(make-gracket-launcher args dest [aux]) → void?
  args : (listof string?)
  dest : path-string?
  aux : (listof (cons/c symbol? any/c)) = null
```

Creates the launcher *dest*, which starts GRacket with the command-line arguments specified as strings in *args*. Extra arguments passed to the launcher at run-time are appended (modulo special Unix/X flag handling, as described below) to this list and passed on to GRacket. If *dest* exists already, as either a file or directory, it is replaced.

The optional *aux* argument is an association list for platform-specific options (i.e., it is a list of pairs where the first element of the pair is a key symbol and the second element is the value for that key). See also `build-aux-from-path`. See `create-embedding-executable` for a list that applies to both stand-alone executables and launchers on Windows and Mac OS X GRacket; the following additional associations apply to launchers:

- `'independent?` (Windows) — a boolean; `#t` creates an old-style launcher that work with any Racket or GRacket binary, like `setup-plt.exe`. No other *aux* associations are used for an old-style launcher.
- `'exe-name` (Mac OS X, `'script-3m` or `'script-cgc` variant) — provides the base name for a `'3m`-/`'cgc`-variant launcher, which the script will call ignoring *args*. If this name is not provided, the script will go through the GRacket executable as usual.
- `'exe-is-gracket` (when `'exe-name` is used) — indicates that `'exe-name` refers to the GRacket executable, which is potentially in a `"lib"` subdirectory instead of with other GUI applications.
- `'relative?` (all platforms) — a boolean, where `#t` means that the generated launcher should find the base GRacket executable through a relative path.

- `'install-mode` (Windows, Unix) — either `'user` or `'main`, indicates that the launcher is being installed to a user-specific place or to an installation-wide place, which in turn determines where to record `'start-menu`, `'extension-registry`, and/or `'desktop` information.

- `'start-menu` (Windows) — a boolean or real number; `#t` indicates that the launcher should be in the Start menu by an installer that includes the launcher. A number value is treated like `#t`, but also requests that the installer automatically start the application, where the number determines a precedence relative to other launchers that may request starting. A `'start-menu` value is used only when `'install-mode` is also specified.

- `'extension-register` (Windows) — a list of document types for file-extension registrations to be performed by an installer. Each document type is described by a list of six items:

  – a human-readable string describing the document type, such as `"Racket Document"`;

  – a string to use as a key for the document type, such as `"Racket.Document"`;

  – a list of strings, where each string is a file extension without the dot, such as `'("rkt" "rktl" "rktd")`;

  – a path to a file that supplies the icon, such as `"doc.ico"`;

  – a string to represent the command line to handle a document with a matching extension, such as `"\"%1\""`, where the string will be prefixed with a path to the launcher, and where `%1` will be replaced with the document path

  An `'extension-registry` value is used only when `'install-mode` is also specified.

- `'desktop` (Unix) — a string containing the content of a `".desktop"` file for the launcher, where `Exec` and `Icon` entries are added automatically. If an `Exec` entry exists in the string, and if its value starts with a non-empty sequence of alpha-numeric ASCII characters followed by a space, then the space and remainder of the value is appended to the automatically generated value. The `".desktop"` file is written to the directory produced by `(find-apps-dir)` or `(find-user-apps-dir)`. A `'desktop` value is used only when `'install-mode` is also specified.

- `'png` (Unix) : An icon file path (suffix `".png"`) to be referenced by a `".desktop"` file (if any); a `'png` value takes precedence over a `'ico` value, but neither is used unless a `'desktop` value is also present.

- `'ico` (Unix, in addition to more general Windows use) : An icon file path (suffix `".ico"`) that is used in the same way as `'png` if no `'png` value is available.

For Unix/X, the script created by `make-mred-launcher` detects and handles X Windows flags specially when they appear as the initial arguments to the script. Instead of appending these arguments to the end of *`args`*, they are spliced in after any X Windows flags already listed in *`args`*. The remaining arguments (i.e., all script flags and arguments after the last X Windows flag or argument) are then appended after the spliced *`args`*.

```
(make-racket-launcher args dest [aux]) → void?
  args : (listof string?)
  dest : path-string?
  aux : (listof (cons/c symbol? any/c)) = null
```

Like make-gracket-launcher, but for starting Racket. On Mac OS X, the 'exe-name *aux* association is ignored.

```
(make-gracket-program-launcher file
                               collection
                               dest)      → void?
  file : string?
  collection : string?
  dest : path-string?
```

Calls make-gracket-launcher with arguments that start the GRacket program implemented by *file* in *collection*: (list "-l-" (string-append *collection* "/" *file*)). The *aux* argument to make-gracket-launcher is generated by stripping the suffix (if any) from *file*, adding it to the path of *collection*, and passing the result to build-aux-from-path.

```
(make-racket-program-launcher file
                              collection
                              dest)      → void?
  file : string?
  collection : string?
  dest : path-string?
```

Like make-gracket-program-launcher, but for make-racket-launcher.

```
(install-gracket-program-launcher file
                                  collection
                                  name)      → void?
  file : string?
  collection : string?
  name : string?
```

Same as

```
(make-gracket-program-launcher
 file collection
 (gracket-program-launcher-path name))
```

```
(install-racket-program-launcher file
                                 collection
                                 name)      → void?
  file : string?
  collection : string?
  name : string?
```

Same as

```
(make-racket-program-launcher
 file collection
 (racket-program-launcher-path name))
```

```
(make-mred-launcher args dest [aux]) → void?
  args : (listof string?)
  dest : path-string?
  aux : (listof (cons/c symbol? any/c)) = null
(make-mred-program-launcher file
                            collection
                            dest)      → void?
  file : string?
  collection : string?
  dest : path-string?
(install-mred-program-launcher file
                               collection
                               name)      → void?
  file : string?
  collection : string?
  name : string?
```

Backward-compatible version of `make-gracket-launcher`, etc., that adds `"-I" "scheme/gui/init"` to the start of the command-line arguments.

```
(make-mzscheme-launcher args dest [aux]) → void?
  args : (listof string?)
  dest : path-string?
  aux : (listof (cons/c symbol? any/c)) = null
(make-mzscheme-program-launcher file
                                collection
                                dest)      → void?
  file : string?
  collection : string?
  dest : path-string?
```

```
(install-mzscheme-program-launcher file
                                   collection
                                   name)      → void?
  file : string?
  collection : string?
  name : string?
```

Backward-compatible version of `make-racket-launcher`, etc., that adds `"-I" "scheme/init"` to the start of the command-line arguments.

### 2.2.2 Launcher Path and Platform Conventions

```
(gracket-program-launcher-path  name
                                [#:user? user?]) → path?
  name : string?
  user? : any/c = #f
```

Returns a pathname for an executable called something like *name* in the Racket installation (if *user?* is `#f`) or the user's Racket executable directory (if *user?* is `#t`). For Windows, the `".exe"` suffix is automatically appended to *name*. For Unix, *name* is changed to lowercase, whitespace is changed to `-`, and the path includes the `"bin"` subdirectory of the Racket installation. For Mac OS X, the `".app"` suffix is appended to *name*.

```
(racket-program-launcher-path  name
                               [#:user? user?]) → path?
  name : string?
  user? : any/c = #f
```

Returns the same path as `(gracket-program-launcher-path name #:user? user?)` for Unix and Windows. For Mac OS X, the result is the same as for Unix.

```
(gracket-launcher-is-directory?) → boolean?
```

Returns `#t` if GRacket launchers for the current platform are directories from the user's perspective. For all currently supported platforms, the result is `#f`.

```
(racket-launcher-is-directory?) → boolean?
```

Like `gracket-launcher-is-directory?`, but for Racket launchers.

```
(gracket-launcher-is-actually-directory?) → boolean?
```

Returns `#t` if GRacket launchers for the current platform are implemented as directories from the filesystem's perspective. The result is `#t` for Mac OS X, `#f` for all other platforms.

```
(racket-launcher-is-actually-directory?) → boolean?
```

Like `gracket-launcher-is-actuall-directory?`, but for Racket launchers. The result is `#f` for all platforms.

```
(gracket-launcher-add-suffix path-string?) → path?
  path-string? : path
```

Returns a path with a suitable executable suffix added, if it's not present already.

```
(racket-launcher-add-suffix path-string?) → path?
  path-string? : path
```

Like `gracket-launcher-add-suffix`, but for Racket launchers.

```
(gracket-launcher-put-file-extension+style+filters)
 → (or/c string? false/c)
    (listof (one-of/c 'packages 'enter-packages))
    (listof (list/c string? string?))
```

Returns three values suitable for use as the `extension`, `style`, and `filters` arguments to `put-file`, respectively.

If GRacket launchers for the current platform were directories form the user's perspective, the `style` result is suitable for use with `get-directory`, and the `extension` result may be a string indicating a required extension for the directory name.

```
(racket-launcher-put-file-extension+style+filters)
 → (or/c string? false/c)
    (listof (one-of/c 'packages 'enter-packages))
    (listof (list/c string? string?))
```

Like `gracket-launcher-get-file-extension+style+filters`, but for Racket launchers.

```
(mred-program-launcher-path  name
                             [#:user? user?]) → path?
  name : string?
  user? : any/c = #f
(mred-launcher-is-directory?) → boolean?
(mred-launcher-is-actually-directory?) → boolean?
(mred-launcher-add-suffix path-string?) → path?
  path-string? : path
(mred-launcher-put-file-extension+style+filters)
 → (or/c string? false/c)
    (listof (one-of/c 'packages 'enter-packages))
    (listof (list/c string? string?))
```

Backward-compatible aliases for gracket-program-launcher-path, etc.

```
(mzscheme-program-launcher-path  name
                                 [#:user? user?]) → path?
  name : string?
  user? : any/c = #f
(mzscheme-launcher-is-directory?) → boolean?
(mzscheme-launcher-is-actually-directory?) → boolean?
(mzscheme-launcher-add-suffix path-string?) → path?
  path-string? : path
(mzscheme-launcher-put-file-extension+style+filters)
 → (or/c string? false/c)
    (listof (one-of/c 'packages 'enter-packages))
    (listof (list/c string? string?))
```

Backward-compatible aliases for racket-program-launcher-path, etc.

```
(installed-executable-path->desktop-path exec-path
                                         user?)
 → (and/c path? complete-path?)
  exec-path : path-string?
  user? : any/c
```

Returns a path for a ".desktop" file to describe the installed executable at *exec-path*. Only the filename part of *exec-path* is used. The *user?* argument should be true if *exec-path* is installed in a user-specific location (in which case the result path will also be user-specific).

```
(installed-desktop-path->icon-path desktop-path
                                   user?
                                   suffix)
 → (and/c path? complete-path?)
  desktop-path : path-string?
  user? : any/c
  suffix : bytes?
```

Returns a path for an icon file to be referenced by the "desktop" file at *desktop-path*. Only the filename part of *desktop-path* is used. The *user?* argument should be true if *desktop-path* is installed in a user-specific location (in which case the result path will also be user-specific). The *suffix* argument provides the icon-file suffix, normally either #"png" or #"ico".

### 2.2.3 Launcher Configuration

```
(gracket-launcher-up-to-date? dest aux) → boolean?
  dest : path-string?
  aux : (listof (cons/c symbol? any/c))
```

Returns `#t` if the GRacket launcher *dest* does not need to be updated, assuming that *dest* is a launcher and its arguments have not changed.

```
(racket-launcher-up-to-date? dest aux) → boolean?
  dest : path-string?
  aux : (listof (cons/c symbol? any/c))
```

Analogous to `gracket-launcher-up-to-date?`, but for a Racket launcher.

```
(build-aux-from-path path) → (listof (cons/c symbol? any/c))
  path : path-string?
```

Creates an association list suitable for use with `make-gracket-launcher` or `create-embedding-executable`. It builds associations by adding to *path* suffixes, such as `".icns"`, checking whether such a file exists, and calling `extract-aux-from-path` if so. The results from all recognized suffixes are appended together.

```
(extract-aux-from-path path) → (listof (cons/c symbol? any/c))
  path : path-string?
```

Creates an association list suitable for use with `make-gracket-launcher` or `create-embedding-executable`. It builds associations by recognizing the suffix of *path*, where the recognized suffixes are as follows:

- `".icns"` → `'icns` file for use on Mac OS X
- `".ico"` → `'ico` file for use on Windows or Unix
- `".png"` → `'png` file for use on Unix
- `".lch"` → `'independent?` as `#t` (the file content is ignored) for use on Windows
- `".creator"` → `'creator` as the initial four characters in the file for use on Mac OS X
- `".filetypes"` → `'file-types` as `read` content (a single S-expression), and `'resource-files` as a list constructed by finding `"CFBundleTypeIconFile"` entries in `'file-types` (and filtering duplicates); for use on Mac OS X
- `".utiexports"` → `'uti-exports` as `read` content (a single S-expression); for use on Mac OS X

- `".wmclass"` → `'wm-class` as the literal content, removing a trailing newline if any; for use on Unix
- `".desktop"` → `'desktop` as the literal content; for use on Unix
- `".startmenu"` → `'start-menu` as the file content if it `read`s as a real number, `#t` otherwise, for use on Windows
- `".extreg"` → `'extension-register` as `read` content (a single S-expression), but with relative (to the `".extreg"` file) paths converted absolute; for use on Windows

```
(current-launcher-variant) → symbol?
(current-launcher-variant variant) → void?
  variant : symbol?
```

A parameter that indicates a variant of Racket or GRacket to use for launcher creation and for generating launcher names. The default is the result of `(system-type 'gc)`. On Unix and Windows, the possibilities are `'cgc` and `'3m`. On Mac OS X, the `'script-3m` and `'script-cgc` variants are also available for GRacket launchers.

```
(available-gracket-variants) → (listof symbol?)
```

Returns a list of symbols corresponding to available variants of GRacket in the current Racket installation. The list normally includes at least one of `'3m` or `'cgc`— whichever is the result of `(system-type 'gc)`—and may include the other, as well as `'script-3m` and/or `'script-cgc` on Mac OS X.

```
(available-racket-variants) → (listof symbol?)
```

Returns a list of symbols corresponding to available variants of Racket in the current Racket installation. The list normally includes at least one of `'3m` or `'cgc`—whichever is the result of `(system-type 'gc)`—and may include the other.

```
(mred-launcher-up-to-date? dest aux) → boolean?
  dest : path-string?
  aux : (listof (cons/c symbol? any/c))
(mzscheme-launcher-up-to-date? dest aux) → boolean?
  dest : path-string?
  aux : (listof (cons/c symbol? any/c))
(available-mred-variants) → (listof symbol?)
(available-mzscheme-variants) → (listof symbol?)
```

Backward-compatible aliases for `gracket-launcher-up-to-date?`, etc.

### 2.2.4 Launcher Creation Signature

```
(require launcher/launcher-sig)
```
package: compiler-lib

```
launcher^ : signature
```

Includes the identifiers provided by launcher/launcher.

### 2.2.5 Launcher Creation Unit

```
(require launcher/launcher-unit)
```
package: compiler-lib

```
launcher@ : unit?
```

A unit that imports nothing and exports launcher^.

## 2.3 Mac OS X Dynamic Library Paths

```
(require compiler/exe-dylib-path)
```
package: base

The compiler/exe-dylib-path library provides functions for reading and adjusting dynamic-library references in a Mac OS X executable.

Added in version 6.3 of package base.

```
(find-matching-library-path exe-path
                            library-str) → (or/c #f string?)
  exe-path : path-string?
  library-str : string?
```

Searches dynamic-linking information in *exe-path* for a library reference whose name includes *library-str* and returns the executable's path to the library for the first match. If no match is found, the result is #f.

```
(update-matching-library-path exe-path
                              library-str
                              library-path-str) → void?
  exe-path : path-string?
  library-str : string?
  library-path-str : string?
```

Searches dynamic-linking information in *exe-path* for each library reference whose name includes *library-str* and replaces the executable's path to that library with *library-path-str*.

A single match is expected, and the update assumes enough space for the new path, perhaps because the executable is linked with `-headerpad_max_install_names`.

# 3 `raco distribute`: Sharing Stand-Alone Executables

The `raco distribute` command combines a stand-alone executable created by `raco exe` with all of the shared libraries that are needed to run it, along with any run-time files declared via `define-runtime-path`. The resulting package can be moved to other machines that run the same operating system.

After the `raco distribute` command, supply a directory to contain the combined files for a distribution. Each command-line argument is an executable to include in the distribution, so multiple executables can be packaged together. For example, on Windows,

```
raco distribute greetings hello.exe goodbye.exe
```

creates a directory `"greetings"` (if the directory doesn't exist already), and it copies the executables `"hello.exe"` and `"goodbye.exe"` into `"greetings"`. It also creates a `"lib"` sub-directory in `"greetings"` to contain DLLs, and it adjusts the copied `"hello.exe"` and `"goodbye.exe"` to use the DLLs in `"lib"`.

The layout of files within a distribution directory is platform-specific:

- On Windows, executables are put directly into the distribution directory, and DLLs and other run-time files go into a `"lib"` sub-directory.
- On Mac OS X, GUI executables go into the distribution directory, other executables go into a `"bin"` subdirectory, and frameworks (i.e., shared libraries) go into a `"lib"` sub-directory along with other run-time files. As a special case, if the distribution has a single `--gui-exe` executable, then the `"lib"` directory is hidden inside the application bundle.
- On Unix, executables go into a `"bin"` subdirectory, shared libraries (if any) go into a `"lib"` subdirectory along with other run-time files, and wrapped executables are placed into a `"lib/plt"` subdirectory with version-specific names. This layout is consistent with Unix installation conventions; the version-specific names for shared libraries and wrapped executables means that distributions can be safely unpacked into a standard place on target machines without colliding with an existing Racket installation or other executables created by `raco exe`.

A distribution also has a `"collects"` directory that is used as the main library collection directory for the packaged executables. By default, the directory is empty. Use the `++collects-copy` flag of `raco distribute` to supply a directory whose content is copied into the distribution's `"collects"` directory. The `++collects-copy` flag can be used multiple times to supply multiple directories.

When multiple executables are distributed together, then separately creating the executables with `raco exe` can generate multiple copies of collection-based libraries that are used by multiple executables. To share the library code, instead, specify a target directory for library

copies using the `--collects-dest` flag with `raco exe`, and specify the same directory for each executable (so that the set of libraries used by all executables are pooled together). Finally, when packaging the distribution with `raco distribute`, use the `++collects-copy` flag to include the copied libraries in the distribution.

## 3.1 API for Distributing Executables

```
(require compiler/distribute)
```
package: base

The `compiler/distribute` library provides a function to perform the same work as `raco distribute`.

```
(assemble-distribution dest-dir
                       exec-files
                       [#:executables? executables?
                        #:relative-base relative-base
                        #:collects-path path
                        #:copy-collects dirs])        → void?
  dest-dir : path-string?
  exec-files : (listof path-string?)
  executables? : any/c = #t
  relative-base : (or/c path-string? #f) = #f
  path : (or/c false/c (and/c path-string? relative-path?)) = #f
  dirs : (listof path-string?) = null
```

Copies the executables in *exec-files* to the directory *dest-dir*, along with DLLs, frameworks, shared libraries, and/or runtime files that the executables need to run a different machine. If *executables?* is `#f`, then the *exec-files* are treated as plain data files, instead of executables, and they are modified in-place.

The arrangement of the executables and support files in *dest-dir* depends on the platform. In general, `assemble-distribution` tries to do the Right Thing, but a non-`#f` value for *relative-base* specifies a path for reaching the assembled content relative to the executable at run time. When *executables?* is `#f`, then the default access path is *dest-dir*, with its relativeness preserved.

If a `#:collects-path` argument is given, it overrides the default location of the main `"collects"` directory for the packaged executables. It should be relative to the *dest-dir* directory (typically inside it).

The content of each directory in the `#:copy-collects` argument is copied into the main `"collects"` directory for the packaged executables.

Changed in version 6.3 of package `base`: Added the `#:executables?` and `#:relative-base` arguments.

## 3.2 API for Bundling Distributions

```
(require compiler/bundle-dist)
```
package: compiler-lib

The compiler/bundle-dist library provides a function to pack a directory (usually assembled by assemble-distribution) into a distribution file. On Windows, the result is a ".zip" archive; on Mac OS X, it's a ".dmg" disk image; on Unix, it's a ".tgz" archive.

```
(bundle-directory dist-file dir [for-exe?]) → void?
  dist-file : file-path?
  dir : file-path?
  for-exe? : any/c = #f
```

Packages *dir* into *dist-file*. If *dist-file* has no extension, a file extension is added automatcially (using the first result of bundle-put-file-extension+style+filters).

The created archive contains a directory with the same name as *dir*—except on Mac OS X when *for-exe?* is true and *dir* contains a single a single file or directory, in which case the created disk image contains just the file or directory. The default for *for-exe?* is #f.

Archive creation fails if *dist-file* exists.

```
(bundle-put-file-extension+style+filters)
 → (or/c string? false/c)
    (listof (one-of/c 'packages 'enter-packages))
    (listof (list/c string? string?))
```

Returns three values suitable for use as the extension, style, and filters arguments to put-file, respectively to select a distribution-file name.

# 4 `raco planet`: Automatic Package Distribution

See *PLaneT: Automatic Package Distribution* for information on the `raco planet` command, which is used for managing packages that can be automatically downloaded and installed from the PLaneT server.

# 5 `raco pkg`: Package Management

See *Package Management in Racket* for information on the `raco pkg` command, which is used for managing external code packages.

# 6 `raco setup`: Installation Management

The `raco setup` command builds bytecode, documentation, executables, and metadata indexes for all installed collections.

The collections that are built by `raco setup` can be part of the original Racket distribution, installed via the package manager (see *Package Management in Racket*), installed via PLaneT (see *PLaneT: Automatic Package Distribution*), linked via `raco link`, in a directory that is listed in the `PLTCOLLECTS` environment variable, or placed into one of the default collection directories.

The `raco setup` tool itself does not directly support the installation of collections, except through the now-discouraged `-A` flag (see §6.2 "Installing `".plt"` Archives"). The `raco setup` command is used by installation tools such as the package manager or PLaneT. Programmers who modify installed collections may find it useful to run `raco setup` as an alternative to un-installing and re-installing a set of collections.

## 6.1 Running `raco setup`

With no command-line arguments, `raco setup` finds all of the current collections—see §18.2 "Libraries and Collections"—and compiles libraries in each collection. (Directories that are named `".git"` or `".svn"` are not treated as collections.)

To restrict `raco setup` to a set of collections, provide the collection names as arguments. For example, `raco setup scribblings/raco` would only compile and render the documentation for `raco`, which is implemented in a `"scribblings/raco"` collection.

An optional `"info.rkt"` within the collection can indicate specifically how the collection's files are to be compiled and other actions to take in setting up a collection, such as creating executables or building documentation. See §6.3 "Controlling `raco setup` with `"info.rkt"` Files" for more information.

The `raco setup` command accepts the following command-line flags:

- Constraining to specified collections or PLaneT packages:
  - `--only` — restrict setup to specified collections and PLaneT packages, even if none are specified. This mode is the default if any collection is specified as a command-line argument or through the `-l`, `--pkgs`, or `-P` flag.
  - `-l` ⟨*collection*⟩ `...` — constrain setup actions to the specified ⟨*collection*⟩s (i.e., the same as providing ⟨*collections*⟩s without a flag, but with no possibility that a ⟨*collection*⟩ is interpreted as a flag).
  - `--pkgs` ⟨*pkg*⟩ `...` — constrain setup actions to collections that are within (or partially within) the named ⟨*pkg*⟩s.

  - `-P` ⟨*owner*⟩ ⟨*package-name*⟩ ⟨*maj*⟩ ⟨*min*⟩ — constrain setup actions to the specified PLaneT package, in addition to any other specified PLaneT packages or collections.
  - `--doc-index` — build collections that implement documentation indexes (when documentation building is enabled), in addition to specified collections.
  - `--tidy` — remove metadata cache information and documentation for non-existent collections or documentation (to clean up after removal), even when setup actions are otherwise confined to specified collections.

- Constraining to specific tasks:
  - `--clean` or `-c` — delete existing `".zo"` files, thus ensuring a clean build from the source files. The exact set of deleted files can be controlled by `"info.rkt"`; see `clean` for more information.
  - `--fast-clean` or `-c` — like `--clean`, but without forcing a bootstrap of `raco setup` from source (which means that `--fast-clean` cannot clean corruption that affects `raco setup` itself).
  - `--no-zo` or `-n` — refrain from compiling source files to `".zo"` files.
  - `--trust-zos` — fix timestamps on `".zo"` files on the assumption that they are already up-to-date.
  - `--no-launcher` or `-x` — refrain from creating executables or installing `man` pages (as specified in `"info.rkt"`; see §6.3 “Controlling `raco setup` with `"info.rkt"` Files”).
  - `--no-foreign-libs` or `-F` — refrain from installing foreign libraries (as specified in `"info.rkt"`; see §6.3 “Controlling `raco setup` with `"info.rkt"` Files”).
  - `--only-foreign-libs` — disable actions other than installing foreign libraries; equivalent to `-nxiIdD`, except that `--only-foreign-libs` doesn't reject (redundant) specification of those individual flags.
  - `--no-install` or `-i` — refrain from running pre-install actions (as specified in `"info.rkt"` files; see §6.3 “Controlling `raco setup` with `"info.rkt"` Files”).
  - `--no-post-install` or `-I` — refrain from running post-install actions (as specified in `"info.rkt"` files; see §6.3 “Controlling `raco setup` with `"info.rkt"` Files”).
  - `--no-info-domain` or `-d` — refrain from building a cache of metadata information from `"info.rkt"` files. This cache is needed by other tools. For example, `raco` itself uses the cache to locate plug-in tools.
  - `--no-docs` or `-D` — refrain from building documentation.
  - `--doc-pdf` ⟨*dir*⟩ — in addition to building HTML documentation, render documentation to PDF and place files in ⟨*dir*⟩.

  - `--no-pkg-deps` or `-K` — refrain from checking whether dependencies among libraries are properly reflected by package-level dependency declarations, whether modules are declared by multiple packages, and whether package version dependencies are satisfied. See §6.5 “Package Dependency Checking” for more information.
  - `--check-pkg-deps` — checks package dependencies (unless explicitly disabled) even when specific collections are provided to `raco setup`, and even for packages that have no dependency declarations. See §6.5 “Package Dependency Checking” for more information.
  - `--fix-pkg-deps` — attempt to correct dependency mismatches by adjusting package `"info.rkt"` files (which makes sense only for packages that are installed as links). See §6.5 “Package Dependency Checking” for more information.
  - `--unused-pkg-deps` — attempt to report dependencies that are declared but are unused. Beware that some package dependencies may be intentionally unused (e.g., declared to force installation of other packages as a convenience), and beware that package dependencies may be reported as unused only because compilation of relevant modules has been suppressed. See §6.5 “Package Dependency Checking” for more information.

- Constraining user versus installation setup:
  - `--no-user` or `-U` — refrain from any user-specific (as opposed to installation-specific) setup actions.
  - `--no-planet` — refrain from any setup actions for PLaneT actions; this flags is implied by `--no-user`.
  - `--avoid-main` — refrain from any setup actions that affect the installation, as opposed to user-specific actions.
  - `--force-user-docs` — when building documentation, create a user-specific documentation entry point even if it has the same content as the main installation.

- Selecting parallelism and other build modes:
  - `--jobs` ⟨*n*⟩, `--workers` ⟨*n*⟩, or `-j` ⟨*n*⟩ — use up to ⟨*n*⟩ parallel processes. By default, `raco setup` uses `(processor-count)` jobs, which typically uses all of the machine’s processing cores.
  - `--verbose` or `-v` — more verbose output about `raco setup` actions.
  - `--make-verbose` or `-m` — more verbose output about dependency checks.
  - `--compiler-verbose` or `-r` — even more verbose output about dependency checks and compilation.
  - `--mode` ⟨*mode*⟩ — use a `".zo"` compiler other than the default compiler, and put the resulting `".zo"` files in a subdirectory (of the usual place) named by ⟨*mode*⟩. The compiler is obtained by using ⟨*mode*⟩ as a collection name, finding a `"zo-compile.rkt"` module in that collection, and extracting its `zo-compile`

export. The `zo-compile` export should be a function like `compile`; see the `"errortrace"` collection for an example.

  - `--fail-fast` — attempt to break as soon as any error is discovered.
  - `--pause` or `-p` — pause for user input if any errors are reported (so that a user has time to inspect output that might otherwise disappear when the `raco setup` process ends).

- Unpacking `".plt"` archives:
  - `-A` ⟨*archive*⟩ `...` — Install each ⟨*archive*⟩; see §6.2 “Installing `".plt"` Archives”.
  - `--force` — for use with `-A`, treat version mismatches for archives as mere warnings.
  - `--all-users` or `-a` — for use with `-A`, install archive into the installation instead of a user-specific location.

When building `racket`, flags can be provided to `raco setup` as run by `make install` by setting the `PLT_SETUP_OPTIONS` environment variable. For example, the following command line uses a single process to build collections during an install:

```
env PLT_SETUP_OPTIONS="-j 1" make install
```

Changed in version 6.1: Added the `--pkgs`, `--check-pkg-deps`, and `--fail-fast` flags. Changed in version 6.1.1: Added the `--force-user-docs` flag. Changed in version 6.1.1.6: Added the `--only-foreign-libs` flag.

## 6.2 Installing `".plt"` Archives

A `".plt"` file is a platform-independent distribution archive for software based on Racket. A typical `".plt"` file can be installed as a package using `raco pkg` (see *Package Management in Racket*), in which case `raco pkg` supplies facilities for uninstalling the package and managing dependencies.

An older approach is to supply a `".plt"` file to `raco setup` with the `-A` flag, the files contained in the `".plt"` archive are unpacked (according to specifications embedded in the `".plt"` file) and only collections specified by the `".plt"` file are compiled and setup. Archives processed in this way can include arbitrary code that is executed at install time, in addition to any actions triggered by the normal collection-setup part of `raco setup`.

Finally, the `raco unpack` (see §11 “`raco unpack`: Unpacking Library Collections”) command can list the content of a `".plt"` archive or unpack the archive without installing it as a package or collection.

## 6.3 Controlling `raco setup` with `"info.rkt"` Files

To compile a collection's files to bytecode, `raco setup` uses the `compile-collection-zos` procedure. That procedure, in turn, consults the collection's `"info.rkt"` file, if it exists, for specific instructions on compiling the collection. See `compile-collection-zos` for more information on the fields of `"info.rkt"` that it uses, and see §6.4 "`"info.rkt"` File Format" for information on the format of an `"info.rkt"` file.

Additional fields are used by the Racket package manager and are documented in §4 "Package Metadata". The `raco test` command also recognizes additional fields, which are documented in §13.2 "Test Configuration by `"info.rkt"`".

Optional `"info.rkt"` fields trigger additional actions by `raco setup`:

- `scribblings` : `(listof (cons/c string? list?))` — A list of documents to build. Each document in the list is itself represented as a list, where each document's list starts with a string that is a collection-relative path to the document's source file. A document name (which is derived from the source module's name by default) is intended to be globally unique in the same way as a package or module name.

  More precisely a `scribblings` entry must be a value that can be generated from an expression matching the following `entry` grammar:

```
      entry = (list doc ...)

        doc = (list src-string)
            | (list src-string flags)
            | (list src-string flags category)
            | (list src-string flags category name)
            | (list src-string flags category name out-k)
            | (list src-string flags category name out-k order-n)

      flags = (list mode-symbol ...)

   category = (list category-string-or-symbol)
            | (list category-string-or-symbol sort-number)

       name = string
            | #f
```

  A document's list optionally continues with information on how to build the document. If a document's list contains a second item, *flags*, it must be a list of mode symbols (described below). If a document's list contains a third item, *category*, it must be a list that categorizes the document (described further below). If a document's list contains a fourth item, *name*, it is a name to use for the generated documentation, instead of defaulting to the source file's name (sans extension), where `#f` means to use the default; a non-`#f` value for *name* must fit the grammar of a collection-name

element as checked by `collection-name-element?`. If a document's list contains a fifth item, *`out-k`*, it is used a hint for the number of files to use for the document's cross-reference information; see below. If a document's list contains a fourth item, *`order-n`*, it is used a hint for the order of rendering; see below.

Each mode symbol in *`flags`* can be one of the following, where only `'multi-page` is commonly used:

- `'multi-page` : Generates multi-page HTML output, instead of the default single-page format.
- `'main-doc` : Indicates that the generated documentation should be written into the main installation directory, instead of to a user-specific directory. This mode is the default for a collection that is itself located in the main installation.
- `'user-doc` : Indicates that the generated documentation should be written a user-specific directory. This mode is the default for a collection that is not itself located in the main installation.
- `'depends-all` : Indicates that the document should be re-built if any other document is rebuilt—except for documents that have the `'no-depend-on` flag.
- `'depends-all-main` : Indicates that the document should be re-built if any other document is rebuilt that is installed into the main installation—except for documents that have the `'no-depend-on` flag.
- `'depends-all-user` : Indicates that the document should be re-built if any other document is rebuilt that is installed into the user's space—except for documents that have the `'no-depend-on` flag.
- `'always-run` : Build the document every time that `raco setup` is run, even if none of its dependencies change.
- `'no-depend-on` : Removes the document for consideration for other dependencies. Furthermore, references from the document to other documents are always direct, instead of potentially indirect (i.e., resolved at document-viewing time and potentially redirected to a remote site).
- `'main-doc-root` : Designates the root document for the main installation. The document that currently has this mode should be the only one with the mode.
- `'user-doc-root` : Designates the root document for the user-specific documentation directory. The document that currently has this mode should be the only one with the mode.
- `'keep-style` : Leave the document's style as-is, instead of imposing the document style for manuals.
- `'no-search` : Build the document without a search box.

The *`category`* list specifies how to show the document in the root table of contents. The list must start with a category, which determines where the manual appears in the root documentation page. A category is either a string or a symbol. If it is a string, then the string is the category label on the root page. If it is a symbol, then a default category label is used. The available symbols and the order of categories on the root documentation page is as below:

- `'getting-started` : High-level, introductory documentation, typeset at the same level as other category titles.
- `'language` : Documentation for a prominent programming language.
- `'tool` : Documentation for an executable.
- `'gui-library` : Documentation for GUI and graphics libraries.
- `'net-library` : Documentation for networking libraries.
- `'parsing-library` : Documentation for parsing libraries.
- `'tool-library` : Documentation for programming-tool libraries (i.e., not important enough for the more prominent `'tool` category).
- `'interop` : Documentation for interoperability tools and libraries.
- All string categories as ordered by `string<=?`.
- `'library` : Documentation for libraries; this category is the default and used for unrecognized category symbols.
- `'legacy` : Documentation for deprecated libraries, languages, and tools.
- `'experimental` : Documentation for an experimental language or library.
- `'other` : Other documentation.
- `'omit` : Documentation that should not be listed on the root page or indexed for searching.
- `'omit-start` : Documentation that should not be listed on the root page but should be indexed for searching.

If the category list has a second element, it must be a real number that designates the manual's sorting position with the category; manuals with the same sorting position are ordered alphabetically. For a pair of manuals with sorting numbers *n* and *m*, the groups for the manuals are separated by space if `(truncate (/ n 10))`and `(truncate (/ m 10))` are different.

The *out-k* specification is a hint on whether to break the document's cross-reference information into multiple parts, which can reduce the time and memory use for resolving a cross-reference into the document. It must be a positive, exact integer, and the default is `1`.

The *order-n* specification is a hint for ordering document builds, since documentation references can be mutually recursive. The order hint can be any real number. A value of `-10` or less disables running the document in parallel to other documents. The main Racket reference is given a value of `-11`, the search page is given a value of `10`, and the default is `0`.

A directory for pre-rendered documentation is computed from the source file name by starting with the directory of the `"info.rkt"` file, adding `"doc"`, and then using the document name (which is usually the source file's name without a suffix); if such a directory exists and does not have a `"synced.rktd"` file, then it is treated as pre-rendered documentation and moved into place, in which case the documentation source file need not be present. Moving documentation into place may require no

movement at all, depending on the way that the enclosing collection is installed, but movement includes adding a `"synced.rktd"` file to represent the installation.

Changed in version 6.4: Allow a category to be a string instead of a symbol.

- `release-note-files` : `(listof (cons/c string? (cons/c string? list?)))` — A list of release-notes text files to link from the main documentation pages. Each note is itself represented as a list, and the list can specify auxiliary notes that are grouped with the main note.

  A `release-note-files` entry must be a value that can be generated from an expression matching the following `entry` grammar:

```
      entry = (list note ...)

        doc = (list label-string note-path)
            | (list label-string note-path order-integer)
            | (list label-string note-path order-integer
                    (list sub-note ...))

   sub-note = (list label-string note-path)
```

  The *order-integer* is used to order notes and defaults to `0`.

- `racket-launcher-names` : `(listof string?)` — A list of executable names to be generated in the installation's executable directory to run Racket-based programs implemented by the collection. A parallel list of library names must be provided by `racket-launcher-libraries` or `racket-launcher-flags`.

  For each name, a launching executable is set up using `make-racket-launcher`. The arguments are `-l-` and ⟨*colls*⟩`/.../`⟨*file*⟩, where ⟨*file*⟩ is the file named by `racket-launcher-libraries` and ⟨*colls*⟩`/...` are the collections (and subcollections) of the `"info.rkt"` file.

  In addition,

```
(build-aux-from-path
 (build-path (collection-path ⟨colls⟩ ...) ⟨suffixless-file⟩))
```

  is provided for the optional *aux* argument (for icons, etc.) to `make-racket-launcher`, where ⟨*suffixless-file*⟩ is ⟨*file*⟩ without its suffix.

  If `racket-launcher-flags` is provided, it is used as a list of command-line arguments passed to `racket` instead of the above default, allowing arbitrary command-line arguments. If `racket-launcher-flags` is specified together with `racket-launcher-libraries`, then the flags will override the libraries, but the libraries can still be used to specify a name for `build-aux-from-path` (to find related information like icon files etc).

- `racket-launcher-libraries` : `(listof path-string?)` — A list of library names in parallel to `racket-launcher-names`.

- `racket-launcher-flags` : `(listof string?)` — A list of command-line flag lists, in parallel to `racket-launcher-names`.

- `mzscheme-launcher-names`, `mzscheme-launcher-libraries`, and `mzscheme-launcher-flags` — Backward-compatible variant of `racket-launcher-names`, etc.

- `gracket-launcher-names` : `(listof string?)` — Like `racket-launcher-names`, but for GRacket-based executables. The launcher-name list is treated in parallel to `gracket-launcher-libraries` and `gracket-launcher-flags`.

- `gracket-launcher-libraries` : `(listof path-string?)` — A list of library names in parallel to `gracket-launcher-names`.

- `gracket-launcher-flags` : `(listof string?)` — A list of command-line flag lists, in parallel to `gracket-launcher-names`.

- `mred-launcher-names`, `mred-launcher-libraries`, and `mred-launcher-flags` — Backward-compatible variant of `gracket-launcher-names`, etc.

- `copy-foreign-libs` : `(listof (and/c path-string? relative-path?))` — Files to copy into a directory where foreign libraries are found by `ffi-lib`. If `install-platform` is defined, then the files are copied only if the current platform matches the definition.

  On Mac OS X, when a Mach-O file is copied, if the copied file includes a library reference that starts `@loader_path/`, and if the referenced library exists in a different location among the paths listed by `(get-lib-search-dirs)`, then the library reference is updated to an absolute path.

  On Unix, when an ELF file is copied, if the copied file includes an RPATH setting of `$ORIGIN` and the file is being installed to a user-specific location, then the file's RPATH is adjusted to `$ORIGIN:` followed by the path to the main installation's library directory as reported by `(find-lib-dir)`.

  On Windows, deleting a previously installed foreign library may be complicated by a lock on the file, if it is in use. To compensate, `raco setup` deletes a foreign-library file by first renaming the file to have the prefix `"raco-setup-delete-"`; it then attempts to delete the renamed file and merely issues a warning on a failure to delete the renamed file. Meanwhile, in modes where `raco setup` removes uninstalled libraries, it attempts to delete any file in the foreign-library directory whose name starts with `"raco-setup-delete-"` (in an attempt to clean up after previous failures).

- `move-foreign-libs` : `(listof (and/c path-string? relative-path?))` — Like `copy-foreign-libs`, but the original file is removed after it is copied (which makes sense for precompiled packages).

- `copy-shared-files` : `(listof (and/c path-string? relative-path?))` — Files to copy into a directory where shared files are found. If `install-platform` is defined, then the files are copied only if the current platform matches the definition.

On Windows, uninstalled files are deleted in the same way as for `copy-foreign-libs`, and the name prefix `"raco-setup-delete-"` is similarly special.

- `move-shared-files` : `(listof (and/c path-string? relative-path?))` — Like `copy-shared-files`, but the original file is removed after it is copied (which makes sense for precompiled packages).

- `copy-man-pages` : `(listof (and/c path-string? relative-path? filename-extension))` — Files to copy into a `man` directory. The file suffix determines its category; for example, `.1` should be used for a `man` page describing an executable.

  On Windows, uninstalled files are deleted in the same way as for `copy-foreign-libs`, and the name prefix `"raco-setup-delete-"` is similarly special.

- `move-man-pages` : `(listof (and/c path-string? relative-path? filename-extension))` — Like `copy-man-pages`, but the original file is removed after it is copied (which makes sense for precompiled packages).

- `install-platform` : `platform-spec?` — Determines whether files are copied or moved for `copy-foreign-libs`, `move-foreign-libs`, `copy-shared-files`, or `move-shared-files`. See `matching-platform?` for information on the way that the specification is compared to `(system-type)` and `(system-library-subpath #f)`.

- `install-collection` : `path-string?` — A library module relative to the collection that provides `installer`. The `installer` procedure accepts one to three arguments. The first argument is a directory path to the parent of the Racket installation's `"collects"` directory; the second argument, if accepted, is a path to the collection's own directory; the third argument, if accepted, is a boolean indicating whether the collection is installed as user-specific (`#t`) or installation-wide (`#f`). The procedure should perform collection-specific installation work, and it should avoid unnecessary work in the case that it is called multiple times for the same installation.

- `pre-install-collection` : `path-string?` — Like `install-collection`, except that the corresponding installer is called *before* the normal `".zo"` build, instead of after. The provided procedure should be named `pre-installer` in this case, so it can be provided by the same file that provides an `installer`.

- `post-install-collection` : `path-string?` — Like `install-collection`. It is called right after the `install-collection` procedure is executed. The only difference between these is that the `--no-install` flag can be used to disable the previous two installers, but not this one. It is therefore expected to perform operations that are always needed, even after an installation that contains pre-compiled files. The provided procedure should be named `post-installer` in this case, so it can be provided by the same file that provides the previous two.

- `assume-virtual-sources` : `any/c` — A true value indicates that bytecode files without a corresponding source file should not be removed from `"compiled"` directories, and no files should not be removed when the `--clean` or `-c` flag is passed to `raco setup`.

- `clean` : `(listof path-string?)` — A list of pathnames to be deleted when the `--clean` or `-c` flag is passed to `raco setup`. The pathnames must be relative to the collection. If any path names a directory, each of the files in the directory are deleted, but none of the subdirectories of the directory are checked. If the path names a file, the file is deleted. The default, if this flag is not specified, is to delete all files in the `"compiled"` subdirectory, and all of the files in the platform-specific subdirectory of the compiled directory for the current platform.

  Just as compiling `".zo"` files will compile each module used by a compiled module, deleting a module's compiled image will delete the `".zo"` of each module that is used by the module. More specifically, used modules are determined when deleting a `".dep"` file, which would have been created to accompany a `".zo"` file when the `".zo"` was built by `raco setup` or `raco make` (see §1.2 "Dependency Files"). If the `".dep"` file indicates another module, that module's `".zo"` is deleted only if it also has an accompanying `".dep"` file. In that case, the `".dep"` file is deleted, and additional used modules are deleted based on the used module's `".dep"` file, etc. Supplying a specific list of collections to `raco setup` disables this dependency-based deletion of compiled files.

- `compile-omit-paths`, `compile-omit-files`, and `compile-include-files` — Used indirectly via `compile-collection-zos`.

- `module-suffixes` and `doc-module-suffixes` — Used indirectly via `get-module-suffixes`.

## 6.4 `"info.rkt"` File Format

```
#lang info      package: base
 #lang setup/infotab
```

In each collection, a special module file `"info.rkt"` provides general information about a collection for use by various tools. For example, an `"info.rkt"` file specifies how to build the documentation for a collection, and it lists plug-in tools for DrRacket or commands for `raco` that the collection provides.

The fields specified in an `"info.rkt"` file are documented in §4 "Package Metadata" for packages and in §6.3 "Controlling `raco setup` with `"info.rkt"` Files" for collections.

Although an `"info.rkt"` file contains a module declaration, the declaration has a highly constrained form. It must match the following grammar of *info-module*:

```
info-module = (module info info-mod-path
                decl
                ...)
```

```
 info-mod-path = info
               | setup/infotab
               | (lib "info/main.rkt")
               | (lib "setup/infotab.ss")
               | (lib "setup/infotab.rkt")
               | (lib "main.rkt" "info")
               | (lib "infotab.rkt" "setup")
               | (lib "infotab.ss" "setup")

          decl = (define id info-expr)

     info-expr = (quote datum)
               | (quasiquote datum)
               | (info-primitive info-expr ...)
               | id
               | string
               | number
               | boolean

info-primitive = cons
               | car
               | cdr
               | list
               | list*
               | reverse
               | append
               | string-append
               | make-immutable-hash
               | hash
               | hash-set
               | hash-set*
               | hash-remove
               | hash-clear
               | hash-update
               | path->string
               | build-path
               | collection-path
               | system-library-subpath
```

For example, the following declaration could be the `"info.rkt"` library of the `"games"` collection. It contains definitions for three info tags, `name`, `gracket-launcher-libraries`, and `gracket-launcher-names`.

```
#lang info
(define name "Games")
(define gracket-launcher-libraries '("main.rkt"))
```

```
(define gracket-launcher-names     '("PLT Games"))
```

As illustrated in this example, an "info.rkt" file can use #lang notation, but only with the info (or setup/infotab) language.

See also get-info from setup/getinfo.

## 6.5 Package Dependency Checking

When raco setup is run with no arguments, after building all collections and documentation, raco setup check package dependencies. Specifically, it inspects compiled files and documentation to check that references across package boundaries are reflected by dependency declarations in each package-level "info.rkt" file (see §4 “Package Metadata”).

Unless --check-pkg-deps is specified, dependency checking is disabled if any collection is specified for raco setup.

Dependency checking in raco setup is intended as an aid to package developers to help them declare dependencies correctly. The raco setup process itself does not depend on package dependency declarations. Similarly, a package with a missing dependency declaration may install successfully for other users, as long as they happen to have the dependencies installed already. A missing dependency creates trouble for others who install a package without having the dependency installed already.

Practically every package depends on the "base" package, which includes the collections that are in a minimal variant of Racket. Declaring a dependency on "base" may seem unnecessary, since its collections are always installed. In a future version of Racket, however, the minimal collections may change, and the new set of minimal collections will then have a package name, such as "base2". Declaring a dependency on "base" ensures forward compatibility, and raco setup complains if the declaration is missing.

To accommodate the early stages of package development, missing dependencies are not treated as an error for a package that has no dependency declarations.

### 6.5.1 Declaring Build-Time Dependencies

A build-time dependency is one that is not present in a package if it is converted to a binary package (see §5 “Source, Binary, and Built Packages”). For example, "tests" and "scribblings" directories are stripped away in a binary package by default, so cross-package references from directories with those names are treated as build dependencies. Similarly, test and doc submodules are stripped away, so references within those submodules create build dependencies.

Build-time-only dependencies can be listed as build-deps instead of deps in a package’s "info.rkt" file. Dependencies listed in deps, meanwhile, are treated as both run-time and

build-time dependencies. The advantage of using `build-deps`, instead of listing all dependencies in `deps`, is that a binary version of the package can install with fewer dependencies.

### 6.5.2 How Dependency Checking Works

Dependency checking uses `".zo"` files, associated `".dep"` files (see §1.2 “Dependency Files”), and the documentation index. Dynamic references, such as through `dynamic-require`, are not visible to the dependency checker; only dependencies via `require`, `define-runtime-module-path-index`, and other forms that cooperate with `raco make` are visible for dependency checking.

Dependency checking is sensitive to whether a dependency is needed only as a build-time dependency. If `raco setup` detects that a missing dependency could be added as a built-time dependency, it will suggest the addition, but `raco setup` will not suggest converting a normal dependency to a build-time dependency (since every normal dependency counts as a build-time dependency, too).

## 6.6 API for Setup

`(require setup/setup)` package: `base`

```
(setup [#:file file
        #:collections collections
        #:planet-specs planet-specs
        #:make-user? make-user?
        #:avoid-main? avoid-main?
        #:make-docs? make-docs?
        #:make-doc-index? make-doc-index?
        #:force-user-docs? force-user-docs?
        #:clean? clean?
        #:tidy? tidy?
        #:jobs jobs
        #:fail-fast? fail-fast?
        #:get-target-dir get-target-dir])   → boolean?
  file : (or/c #f path-string?) = #f
  collections : (or/c #f (listof (listof path-string?))) = #f
  planet-specs : (or/c #f
                       (listof (list/c string?
                                       string?
                                       exact-nonnegative-integer?
                                       exact-nonnegative-integer?)))
               = #f
  make-user? : any/c = #t
```

```
avoid-main? : any/c = #f
make-docs? : any/c = #t
make-doc-index? : any/c = #f
force-user-docs? : any/c = #f
clean? : any/c = #f
tidy? : any/c = #f
jobs : exact-nonnegative-integer? = #f
fail-fast? : any/c = #f
get-target-dir : (or/c #f (-> path-string?)) = #f
```

Runs `raco setup` with various options:

- *file* — if not `#f`, installs *file* as a `".plt"` archive.
- *collections* — if not `#f`, constrains setup to the named collections, along with *planet-specs*, if any
- planet-spec — if not `#f`, constrains setup to the named PLaneT packages, along with *collections*, if any
- *make-user?* — if `#f`, disables any user-specific setup actions
- *avoid-main?* — if true, avoids setup actions that affect the main installation, as opposed to user directories
- *make-docs?* — if `#f`, disables any documentation-specific setup actions
- *make-doc-index?* — if true, builds documentation index collections in addition to *collections*, assuming that documentation is built
- *force-user-docs?* — if true, then when building documentation, create a user-specific documentation entry point even if it has the same content as the installation
- *clean?* — if true, enables cleaning mode instead of setup mode
- *tidy?* — if true, enables global tidying of documentation and metadata indexes even when *collections* or *planet-specs* is non-`#f`
- *jobs* — if not `#f`, determines the maximum number of parallel tasks used for setup
- *fail-fast?* — if true, breaks the current thread as soon as an error is discovered
- *get-target-dir* — if not `#f`, treated as a value for current-target-directory-getter

The result is `#t` if `raco setup` completes without error, `#f` otherwise.

Changed in version 6.1 of package `base`: Added the *fail-fast?* argument. Changed in version 6.1.1: Added the *force-user-docs?* argument.

### 6.6.1 `raco setup` Unit

```
(require setup/setup-unit)
```
package: `compiler-lib`

The `setup/setup-unit` library provides `raco setup` in unit form. The associated `setup/option-sig` and `setup/option-unit` libraries provides the interface for setting options for the run of `raco setup`.

For example, to unpack a single `".plt"` archive `"x.plt"`, set the `archives` parameter to `(list "x.plt")` and leave `specific-collections` as `null`.

Link the options and setup units so that your option-setting code is initialized between them, e.g.:

```
(compound-unit
  ...
  (link ...
    [((OPTIONS : setup-option^)) setup:option@]
    [() my-init-options@ OPTIONS]
    [() setup@ OPTIONS ...])
  ...)
```

```
setup@ : unit?
```

Imports

- `setup-option^`
- `compiler^`
- `compiler:option^`
- `launcher^`
- `dynext:file^`

and exports nothing. Invoking `setup@` starts the setup process.

### 6.6.2 Options Unit

```
(require setup/option-unit)
```
package: `compiler-lib`

```
setup:option@ : unit?
```

Imports nothing and exports `setup-option^`.

### 6.6.3 Options Signature

```
(require setup/option-sig)
```
package: compiler-lib

```
setup-option^ : signature
```

Provides parameters used to control `raco setup` in unit form.

```
(setup-program-name) → string?
(setup-program-name name) → void?
  name : string?
```

The prefix used when printing status messages. The default is `"raco setup"`.

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

If on, prints message from `make` to `stderr`. The default is `#f`.

```
(make-verbose) → boolean?
(make-verbose on?) → void?
  on? : any/c
```

If on, verbose `make`. The default is `#f`.

```
(compiler-verbose) → boolean?
(compiler-verbose on?) → void?
  on? : any/c
```

If on, verbose `compiler`. The default is `#f`.

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

If on, delete `".zo"` and `".so"/".dll"/".dylib"` files in the specified collections. The default is `#f`.

```
(compile-mode) → (or/c path? #f)
(compile-mode path) → void?
  path : (or/c path? #f)
```

If a *path* is given, use a `".zo"` compiler other than plain `compile`, and build to `(build-path "compiled" (compile-mode))`. The default is `#f`.

```
(make-zo) → boolean?
(make-zo on?) → void?
  on? : any/c
```

If on, compile `".zo"`. The default is `#t`.

```
(make-info-domain) → boolean?
(make-info-domain on?) → void?
  on? : any/c
```

If on, update `"info-domain/compiled/cache.rkt"` for each collection path. The default is `#t`.

```
(make-launchers) → boolean?
(make-launchers on?) → void?
  on? : any/c
```

If on, make collection `"info.rkt"`-specified launchers and `man` pages. The default is `#t`.

```
(make-foreign-lib) → boolean?
(make-foreign-lib on?) → void?
  on? : any/c
```

If on, install collection `"info.rkt"`-specified libraries. The default is `#t`.

```
(make-docs) → boolean?
(make-docs on?) → void?
  on? : any/c
```

If on, build documentation. The default is `#t`.

```
(make-user) → boolean?
(make-user on?) → void?
  on? : any/c
```

If on, build the user-specific collection tree. The default is `#t`.

```
(make-planet) → boolean?
(make-planet on?) → void?
  on? : any/c
```

If on, build the planet cache. The default is `#t`.

```
(avoid-main-installation) → boolean?
(avoid-main-installation on?) → void?
  on? : any/c
```

If on, avoid building bytecode in the main installation tree when building other bytecode (e.g., in a user-specific collection). The default is `#f`.

```
(make-tidy) → boolean?
(make-tidy on?) → void?
  on? : any/c
```

If on, remove metadata cache information and documentation for non-existent collections (to clean up after removal) even when `specific-collections` or `specific-planet-dirs` is non-`'()` or `make-only` is true. The default is `#f`.

```
(call-install) → boolean?
(call-install on?) → void?
  on? : any/c
```

If on, call collection `"info.rkt"`-specified setup code. The default is `#t`.

```
(call-post-install) → boolean?
(call-post-install on?) → void?
  on? : any/c
```

If on, call collection `"info.rkt"`-specified post-install code. The default is `#t`.

```
(pause-on-errors) → boolean?
(pause-on-errors on?) → void?
  on? : any/c
```

If on, in the event of an error, prints a summary error and waits for `stdin` input before terminating. The default is `#f`.

```
(parallel-workers) → exact-nonnegative-integer?
(parallel-workers num) → void?
  num : exact-nonnegative-integer?
```

Determines the number of places to use for compiling bytecode and for building the documentation. The default is `(min (processor-count) 8)`.

```
(fail-fast) → boolean?
(fail-fast on?) → void?
  on? : any/c
```

If on, breaks the original thread as soon as an error is discovered. The default is `#f`.

Added in version 1.2 of package `compiler-lib`.

```
(force-unpacks) → boolean?
(force-unpacks on?) → void?
  on? : any/c
```

If on, ignore version and already-installed errors when unpacking a ".plt" archive. The default is #f.

```
(specific-collections) → (listof (listof path-string?))
(specific-collections colls) → void?
  colls : (listof (listof path-string?))
```

A list of collections to set up; the empty list means set-up all collections if the archives list and specific-planet-dirs is also '(). The default is '().

```
(specific-planet-dirs)
 → (listof (list/c string?
                   string?
                   exact-nonnegative-integer?
                   exact-nonnegative-integer?))
(specific-planet-dirs dir) → void?
  dir : (listof (list/c string?
                        string?
                        exact-nonnegative-integer?
                        exact-nonnegative-integer?))
```

A list of planet package version specs to set up; the empty list means to set-up all planet collections if the archives list and specific-collections is also '(). The default is '().

```
(make-only) → boolean?
(make-only on?) → void?
  on? : any/c
```

If true, set up no collections if specific-collections and specific-planet-dirs are both '().

```
(archives) → (listof path-string?)
(archives arch) → void?
  arch : (listof path-string?)
```

A list of ".plt" archives to unpack; any collections specified by the archives are set-up in addition to the collections listed in specific-collections. The default is null.

```
(archive-implies-reindex) → boolean?
(archive-implies-reindex on?) → void?
  on? : any/c
```

If on, when archives has a non-empty list of packages, if any documentation is built, then suitable documentation start pages, search pages, and master index pages are re-built. The default is #t.

```
(current-target-directory-getter) → (-> path-string?)
(current-target-directory-getter thunk) → void?
  thunk : (-> path-string?)
```

A thunk that returns the target directory for unpacking a relative `".plt"` archive; when unpacking an archive, either this or the procedure in `current-target-plt-directory-getter` will be called. The default is `current-directory`.

```
(current-target-plt-directory-getter)
 → (path-string?
    path-string?
    (listof path-string?) . -> . path-string?)
(current-target-plt-directory-getter proc) → void?
  proc : (path-string?
          path-string?
          (listof path-string?) . -> . path-string?)
```

A procedure that takes a preferred path, a path to the parent of the main `"collects"` directory, and a list of path choices; it returns a path for a "plt-relative" install; when unpacking an archive, either this or the procedure in `current-target-directory-getter` will be called, and in the former case, this procedure may be called multiple times. The default is `(lambda (preferred main-parent-dir choices) preferred)`.

## 6.7 API for Installing `".plt"` Archives

The `setup/plt-single-installer` module provides a function for installing a single `".plt"` file.

### 6.7.1 Non-GUI Installer

`(require setup/plt-single-installer)` package: `base`

```
(run-single-installer
 file
 get-dir-proc
 [#:show-beginning-of-file? show-beginning-of-file?])
→ void?
 file : path-string?
 get-dir-proc : (-> (or/c path-string? #f))
 show-beginning-of-file? : any/c = #f
```

Creates a separate thread and namespace, runs the installer in that thread with the new namespace, and returns when the thread completes or dies. It also creates a custodian (see §14.7 “Custodians”) to manage the created thread, sets the exit handler for the thread to shut down the custodian, and explicitly shuts down the custodian when the created thread terminates or dies.

The *get-dir-proc* procedure is called if the installer needs a target directory for installation, and a `#f` result means that the user canceled the installation. Typically, *get-dir-proc* is `current-directory`.

If *show-beginning-of-file?* is a true value and the installation fails, then `run-single-installer` prints the first 1,000 characters of the file (in an attempt to help debug the cause of failures).

```
(install-planet-package file directory spec) → void?
  file : path-string?
  directory : path-string?
  spec : (list/c string? string?
                 (listof string?)
                 exact-nonnegative-integer?
                 exact-nonnegative-integer?)
```

Similar to `run-single-installer`, but runs the setup process to install the archive *file* into *directory* as the PLaneT package described by *spec*. The user-specific documentation index is not rebuilt, so `reindex-user-documentation` should be run after a set of PLaneT packages are installed.

```
(reindex-user-documentation) → void?
```

Similar to `run-single-installer`, but runs only the part of the setup process that rebuilds the user-specific documentation start page, search page, and master index.

```
(clean-planet-package directory spec) → void?
  directory : path-string?
  spec : (list/c string? string?
                 (listof string?)
                 exact-nonnegative-integer?
                 exact-nonnegative-integer?)
```

Undoes the work of `install-planet-package`. The user-specific documentation index is not rebuilt, so `reindex-user-documentation` should be run after a set of PLaneT packages are removed.

## 6.8 API for Finding Installation Directories

`(require setup/dirs)` package: `base`

The setup/dirs library provides several procedures for locating installation directories:

```
(find-collects-dir) → (or/c path? #f)
```

Returns a path to the installation's main "collects" directory, or #f if none can be found. A #f result is likely only in a stand-alone executable that is distributed without libraries.

```
(find-user-collects-dir) → path?
```

Returns a path to the user-specific "collects" directory; the directory indicated by the returned path may or may not exist.

```
(get-collects-search-dirs) → (listof path?)
```

Returns the same result as (current-library-collection-paths), which means that this result is not sensitive to the value of the use-user-specific-search-paths parameter.

```
(get-main-collects-search-dirs) → (listof path?)
```

Returns a list of paths to installation "collects" directories, including the result of find-collects-dir. These directories are normally included in the result of (current-library-collection-paths), but a PLTCOLLECTS setting or change to the parameter may cause them to be omitted. Any other path in (current-library-collection-paths) is treated as user-specific. The directories indicated by the returned paths may or may not exist.

```
(find-config-dir) → (or/c path? #f)
```

Returns a path to the installation's "etc" directory, which contains configuration and package information—including configuration of some of the other directories (see §19 "Installation Configuration and Search Paths"). A #f result indicates that no configuration directory is available.

```
(find-links-file) → path?
```

Returns a path to the installation's collection links file. The file indicated by the returned path may or may not exist.

```
(find-user-links-file [vers]) → path?
  vers : string? = (get-installation-name)
```

Returns a path to the user's collection links file. The file indicated by the returned path may or may not exist.

```
(get-links-search-files) → path?
```

Returns a list of paths to installation collection links files that are search in order. (Normally, the result includes the result of `(find-links-file)`, which is where new installation-wide links are installed by `raco link` or `links`.) The files indicated by the returned paths may or may not exist.

```
(find-pkgs-dir) → path?
```

Returns a path to the directory containing packages with installation scope; the directory indicated by the returned path may or may not exist.

```
(find-user-pkgs-dir [vers]) → path?
  vers : string? = (get-installation-name)
```

Returns a path to the directory containing packages with user-specific scope for installation name *vers*; the directory indicated by the returned path may or may not exist.

```
(get-pkgs-search-dirs) → (listof path?)
```

Returns a list of paths to the directories containing packages in installation scope. (Normally, the result includes the result of `(find-pkgs-dir)`, which is where new packages are installed by `raco pkg install`.) The directories indicated by the returned paths may or may not exist.

```
(find-doc-dir) → (or/c path? #f)
```

Returns a path to the installation's `"doc"` directory. The result is `#f` if no such directory is available.

```
(find-user-doc-dir) → path?
```

Returns a path to a user-specific `"doc"` directory. The directory indicated by the returned path may or may not exist.

```
(get-doc-search-dirs) → (listof path?)
```

Returns a list of paths to search for documentation, not including documentation stored in individual collections. Unless it is configured otherwise, the result includes any non-`#f` result of `(find-doc-dir)` and `(find-user-doc-dir)`—but the latter is included only if the value of the `use-user-specific-search-paths` parameter is `#t`.

```
(find-lib-dir) → (or/c path? #f)
```

Returns a path to the installation's "lib" directory, which contains libraries and other build information. The result is #f if no such directory is available.

```
(find-user-lib-dir) → path?
```

Returns a path to a user-specific "lib" directory; the directory indicated by the returned path may or may not exist.

```
(get-lib-search-dirs) → (listof path?)
```

Returns a list of paths to search for foreign libraries. Unless it is configured otherwise, the result includes any non-#f result of (find-lib-dir) and (find-user-lib-dir)—but the latter is included only if the value of the use-user-specific-search-paths parameter is #t.

Changed in version 6.1.1.4 of package base: Dropped (find-dll-dir) from the set of paths to explicitly include in the default.

```
(find-dll-dir) → (or/c path? #f)
```

Returns a path to the directory that contains DLLs for use with the current executable (e.g., "libracket.dll" on Windows). The result is #f if no such directory is available, or if no specific directory is available (i.e., other than the platform's normal search path).

```
(find-share-dir) → (or/c path? #f)
```

Returns a path to the installation's "share" directory, which contains installed packages and other platform-independent files. The result is #f if no such directory is available.

```
(find-user-share-dir) → path?
```

Returns a path to a user-specific "share" directory; the directory indicated by the returned path may or may not exist.

```
(find-include-dir) → (or/c path? #f)
```

Returns a path to the installation's "include" directory, which contains ".h" files for building Racket extensions and embedding programs. The result is #f if no such directory is available.

```
(find-user-include-dir) → path?
```

Returns a path to a user-specific "include" directory; the directory indicated by the returned path may or may not exist.

```
(get-include-search-dirs) → (listof path?)
```

Returns a list of paths to search for `".h"` files. Unless it is configured otherwise, the result includes any non-`#f` result of `(find-include-dir)` and `(find-user-include-dir)`—but the latter is included only if the value of the `use-user-specific-search-paths` parameter is `#t`.

```
(find-console-bin-dir) → (or/c path? #f)
```

Returns a path to the installation's executable directory, where the stand-alone Racket executable resides. The result is `#f` if no such directory is available.

```
(find-gui-bin-dir) → (or/c path? #f)
```

Returns a path to the installation's executable directory, where the stand-alone GRacket executable resides. The result is `#f` if no such directory is available.

```
(find-user-console-bin-dir) → path?
```

Returns a path to the user's executable directory; the directory indicated by the returned path may or may not exist.

```
(find-user-gui-bin-dir) → path?
```

Returns a path to the user's executable directory for graphical programs; the directory indicated by the returned path may or may not exist.

```
(find-apps-dir) → (or/c path? #f)
```

Returns a path to the installation's directory `".desktop"` files (for Unix). The result is `#f` if no such directory exists.

```
(find-user-apps-dir) → path?
```

Returns a path to the user's directory for `".desktop"` files (for Unix); the directory indicated by the returned path may or may not exist.

```
(find-man-dir) → (or/c path? #f)
```

Returns a path to the installation's man-page directory. The result is `#f` if no such directory exists.

```
(find-user-man-dir) → path?
```

Returns a path to the user's man-page directory; the directory indicated by the returned path may or may not exist.

```
(get-doc-search-url) → string?
```

Returns a string that is used by the documentation system, augmented with a version and search-key query, for remote documentation links.

```
(get-doc-open-url) → (or/c string? #f)
```

Returns `#f` or a string for a root URL to be used as an alternative to opening a local file for documentation. A non-`#f` configuration means that DrRacket, for example, performs keyword searches for documentation via the specified URL instead of from locally installed documentation.

Added in version 6.0.1.6 of package `base`.

```
(get-installation-name) → string?
```

Returns the current installation's name, which is often `(version)` but can be configured via `'installation-name` in `"config.rktd"` (see §19 "Installation Configuration and Search Paths").

```
(get-build-stamp) → (or/c #f string?)
```

Returns a string that identifies an installation build, which can be used to augment the Racket version number to more specifically identify the build. An empty string is normally produced for a release build. The result is `#f` if no build stamp is available.

```
(get-absolute-installation?) → boolean?
```

Returns `#t` if this installation uses absolute path names for executable and library references, `#f` otherwise.

## 6.9 API for Reading `"info.rkt"` Files

```
(require setup/getinfo)
```
package: `base`

The `setup/getinfo` library provides functions for accessing fields in `"info.rkt"` files. The file format for `"info.rkt"` files is documented in §6.4 "`"info.rkt"` File Format".

```
(get-info collection-names
          [#:namespace namespace
           #:bootstrap? bootstrap?])
```

```
→ (or/c
    (symbol? [(-> any)] . -> . any)
    #f)
  collection-names : (listof string?)
  namespace : (or/c namespace? #f) = #f
  bootstrap? : any/c = #f
```

Accepts a list of strings naming a collection or sub-collection, and calls `get-info/full` with the full path corresponding to the named collection and the *namespace* argument.

```
(get-info/full  path
                [#:namespace namespace
                 #:bootstrap? bootstrap?])
 → (or/c (->* (symbol?) ((-> any)) any)
         #f)
  path : path-string?
  namespace : (or/c namespace? #f) = #f
  bootstrap? : any/c = #f
```

Accepts a path to a directory. If it finds either a well-formed an `"info.rkt"` file or an `"info.ss"` file (with preference for the `"info.rkt"` file), it returns an info procedure that accepts either one or two arguments. The first argument to the info procedure is always a symbolic name, and the result is the value of the name in the `"info.rkt"` file, if the name is defined. The optional second argument, *thunk*, is a procedure that takes no arguments to be called when the name is not defined; the result of the info procedure is the result of the *thunk* in that case. If the name is not defined and no *thunk* is provided, then an exception is raised.

The `get-info/full` function returns `#f` if there is no `"info.rkt"` (or `"info.ss"`) file in the directory. If there is a `"info.rkt"` (or `"info.ss"`) file that has the wrong shape (i.e., not a module using `info` or `setup/infotab`), or if the `"info.rkt"` file fails to load, then an exception is raised. If the `"info.rkt"` file loaded, `get-info/full` returns the `get-info` file. If the `"info.rkt"` file does not exist, then `get-info/full` does the same checks for the `"info.rkt"` file, either raising an exception or returning the `get-info` function from the `"info.rkt"` file.

The `"info.rkt"` (or `"info.ss"`) module is loaded into *namespace* if it is not `#f`, or a private, weakly-held namespace otherwise.

If *bootstrap?* is true, then `use-compiled-file-paths` is set to `'()` while reading `"info.rkt"` (or `"info.ss"`), in case an existing compiled file is broken. Furthermore, the `info` and `setup/infotab` modules are attached to *namespace* from the namespace of `get-info/full` before attempting to load `"info.rkt"` (or `"info.ss"`).

```
(find-relevant-directories syms [mode]) → (listof path?)
  syms : (listof symbol?)
```

```
  mode : (or/c 'preferred 'all-available 'no-planet 'no-user)
       = 'preferred
```

Returns a list of paths identifying collections and installed PLaneT packages whose `"info.rkt"` file defines one or more of the given symbols. The result is based on a cache that is computed by `raco setup`.

Note that the cache may be out of date by the time you call `get-info/full`, so do not assume that every returned directory's `"info.rkt"` file will supply one of the requested symbols.

The result is in a canonical order (sorted lexicographically by directory name), and the paths it returns are suitable for providing to `get-info/full`.

If *mode* is specified, it must be either `'preferred` (the default), `'all-available`, `'no-planet`, or `'no-user`. If *mode* is `'all-available`, `find-relevant-collections` returns all installed directories whose info files contain the specified symbols—for instance, all versions of all installed PLaneT packages will be searched if `'all-available` is specified. If *mode* is `'preferred`, then only a subset of "preferred" packages will be searched: only the directory containing the most recent version of any PLaneT package will be returned. If *mode* is `'no-planet`, then PLaneT packages are not included in the search. If *mode* is `'no-user`, then only installation-wide directories are search, which means omitting PLane T package directories.

Collection links from the installation-wide collection links file or packages with installation scope are cached with the installation's main `"lib"` directory, and links from the user-specific collection links file and packages are cached with the user-specific directory `(build-path (find-system-path 'addon-dir) "collects")` for all-version cases, and in `(build-path (find-system-path 'addon-dir) (version) "collects")` for version-specific cases.

```
(find-relevant-directory-records syms key)
 → (listof directory-record?)
  syms : (listof symbol?)
  key : (or/c 'preferred 'all-available)
```

Like `find-relevant-directories`, but returns `directory-record` structs instead of `path?`s.

```
(struct directory-record (maj min spec path syms)
    #:extra-constructor-name make-directory-record)
  maj : integer?
  min : integer?
  spec : any/c
  path : path?
  syms : (listof symbol?)
```

A struct that records information about a collection or a PLaneT package that has been installed. Collections will have the major version being `1` and the minor version being `0`. The `spec` field is a quoted module spec; the `path` field is where the `info.rkt` file for this collection or PLaneT package exists on the filesystem the `syms` field holds the identifiers defined in that file.

```
(reset-relevant-directories-state!) → void?
```

Resets the cache used by `find-relevant-directories`.

## 6.10 API for Relative Paths

The Racket installation tree can usually be moved around the filesystem. To support this, care must be taken to avoid absolute paths. The following two APIs cover two aspects of this: a way to convert a path to a value that is relative to the `"collects"` tree, and a way to display such paths (e.g., in error messages).

### 6.10.1 Representing Collection-Based Paths

```
(require setup/collects)      package: base
```

```
(path->collects-relative path
                         #:cache cache)
 → (or/c path-string?
         (cons/c 'collects
                 (cons/c bytes? (non-empty-listof bytes?))))
  path : path-string?
  cache : (or/c #f (and/c hash? (not/c immutable?)))
```

Checks whether *path* (normalized by `path->complete-path` and `simplify-path` with `#f` as its second argument) matches the result of `collection-file-path`. If so, the result is a list starting with `'collects` and containing the relevant path elements as byte strings. If not, the path is returned as-is.

The *cache* argument is used with `path->pkg`, if needed.

```
(collects-relative->path rel) → path-string?
  rel : (or/c path-string?
              (cons/c 'collects
                      (cons/c bytes? (non-empty-listof bytes?))))
```

The inverse of `path->collects-relative`: if *rel* is a pair that starts with `'collects`, then it is converted back to a path using `collection-file-path`.

```
(path->module-path path #:cache cache)
 → (or/c path-string? module-path?)
  path : path-string?
  cache : (or/c #f (and/c hash? (not/c immutable?)))
```

Like path->collects-relative, but the result is either *path* or a normalized (in the sense of collapse-module-path) module path.

### 6.10.2 Representing Paths Relative to "collects"

```
(require setup/main-collects)
```
package: base

```
(path->main-collects-relative path)
 → (or/c path? (cons/c 'collects (non-empty-listof bytes?)))
  path : (or/c bytes? path-string?)
```

Checks whether *path* has a prefix that matches the prefix to the main "collects" directory as determined by (find-collects-dir). If so, the result is a list starting with 'collects and containing the remaining path elements as byte strings. If not, the path is returned as-is.

The *path* argument should be a complete path. Applying simplify-path before path->main-collects-relative is usually a good idea.

For historical reasons, *path* can be a byte string, which is converted to a path using bytes->path.

See also collects-relative->path.

```
(main-collects-relative->path rel) → path>
  rel : (or/c bytes?
              path-string?
              (cons/c 'collects (non-empty-listof bytes?)))
```

The inverse of path->main-collects-relative: if *rel* is a pair that starts with 'collects, then it is converted back to a path relative to (find-collects-dir).

### 6.10.3 Representing Paths Relative to the Documentation

```
(require setup/main-doc)
```
package: base

```
(path->main-doc-relative path)
 → (or/c path? (cons/c 'doc (non-empty-listof bytes?)))
  path : (or/c bytes? path-string?)
```

Like path->main-collects-relative, except that it checks for a prefix relative to (find-doc-dir) and returns a list starting with 'doc if so.

```
(main-doc-relative->path rel) → path>
  rel : (or/c bytes?
              path-string?
              (cons/c 'doc (non-empty-listof bytes?)))
```

Like path->main-collects-relative, except it is the inverse of path->main-doc-relative.

### 6.10.4 Displaying Paths Relative to a Common Root

```
(require setup/path-to-relative)
```
package: base

```
(path->relative-string/library path
                               [default
                                #:cache cache]) → any/c
  path : path-string?
  default : (or/c (-> path-string? any/c) any/c)
          = (lambda (x) (if (path? x) (path->string x) x))
  cache : (or/c #f (and/c hash? (not/c immutable?))) = #f
```

Produces a string suitable for display in error messages. If the path is an absolute one that is inside a package, the result is a string that begins with "<pkgs>/". If the path is an absolute one that is inside the "collects" tree, the result is a string that begins with "<collects>/". Similarly, a path in the user-specific collects results in a prefix of "<user-collects>/", a PLaneT path results in "<planet>/", and a path into documentation results in "<doc>/" or "<user-doc>/".

If *cache* is not #f, it is used as a cache argument for pkg->path to speed up detection and conversion of package paths.

If the path is not absolute, or if it is not in any of these, it is returned as-is (converted to a string if needed). If *default* is given, it specifies the return value instead: it can be a procedure that is applied onto the path to get the result, or the result itself.

Note that this function can return a non-string only if *default* is given and it does not return a string.

```
(path->relative-string/setup path
                             [default
                              #:cache cache]) → any/c
  path : path-string?
```

```
  default : (or/c (-> path-string? any/c) any/c)
          = (lambda (x) (if (path? x) (path->string x) x))
  cache : (or/c #f (and/c hash? (not/c immutable?))) = #f
```

The same as path->relative-string/library, for backward compatibility.

```
(make-path->relative-string dirs [default])
 → (path-string? any/c . -> . any)
  dirs : (listof (cons (-> path?) string?))
  default : (or/c (-> path-string? any/c) any/c)
          = (lambda (x) (if (path? x) (path->string x) x))
```

This function produces functions like path->relative-string/library and path->relative-string/setup.

The *dirs* argument determines the prefix substitutions. It must be an association list mapping a path-producing thunk to a prefix string for paths in the specified path.

*default* determines the default for the resulting function (which can always be overridden by an additional argument to this function).

## 6.11 API for Collection Names

```
(require setup/collection-name)
```
package: base

```
(collection-name? v) → boolean?
  v : any/c
```

Returns #t if *v* is a string that is syntactically valid as a collection name, which means that it is one or more /-separated strings for which collection-name-element? returns true.

```
(collection-name-element? v) → boolean?
  v : any/c
```

Returns #t if *v* is a string that is syntactically valid as a top-level collection name or as a part of a collection name, which means that it is non-empty and contains only ASCII letters, ASCII digits, -, +, _, and %, where a % is allowed only when followed by two lowercase hexadecimal digits, and the digits must form a number that is not the ASCII value of a letter, digit, -, +, or _.

## 6.12 API for Collection Searches

```
(require setup/collection-search)
```
package: base

Added in version 6.3 of package base.

```
(collection-search mod-path
                   [#:init result
                    #:combine combine
                    #:break? break?
                    #:all-possible-roots? all-possible-roots?])
 → any/c
  mod-path : normalized-lib-module-path?
  result : any/c = #f
  combine : (any/c (and/c path? complete-path?) . -> . any/c)
          = (lambda (r v) v)
  break? : (any/c . -> . any/c) = (lambda (r) #f)
  all-possible-roots? : any/c = #f
```

Generalizes collection-file-path to support folding over all possible locations of a collection-based file in the current configuration. Unlike collection-file-path, collection-search takes the file to location in module-path form, but always as a 'lib path.

Each possible path for the file (not counting a ".ss" to/from ".rkt" conversion) is provided as a second argument to the *combine* function, where the first argument is the current result, and the value produced by *combine* becomes the new result. The #:init argument provides the initial result.

The *break?* function short-circuits a search based on the current value. For example, it could be used to short-circuit a search after a suitable path is found.

If *all-possible-roots?* is #f, then *combine* is called only on paths within "collects"-like directories (for the current configuration) where at least a matching collection directory exists.

```
(normalized-lib-module-path? v) → boolean?
  v : any/c
```

Returns #t if *v* is a module path (in the sense of module-path?) of the form '(lib str) where *str* contains at least one slash. The collapse-module-path function produces such module paths for collection-based module references.

## 6.13 API for Platform Specifications

```
(require setup/matching-platform)
```
package: base

Added in version 6.0.1.13 of package base.

```
(platform-spec? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a symbol, string, or regexp value (in the sense of `regexp?`), `#f` otherwise.

```
(matching-platform? spec
                    [#:cross? cross?
                     #:system-type sys-type
                     #:system-library-subpath sys-lib-subpath])
 → boolean?
  spec : platform-spec?
  cross? : any/c = #f
  sys-type : (or/c #f symbol?) = (if cross?
                                     (cross-system-type)
                                     (system-type))
  sys-lib-subpath : (or/c #f path-for-some-system?)
                  = (if cross?
                        (cross-system-library-subpath #f)
                        (system-library-subpath #f))
```

Reports whether *spec* matches *sys-type* or *sys-lib-subpath*, where `#f` values for the latter are replaced with the default values.

If *spec* is a symbol, then the result is `#t` if *sys-type* is the same symbol, `#f` otherwise.

If *spec* is a string, then the result is `#t` if `(path->string sys-lib-subpath)` is the same string, `#f` otherwise.

If *spec* is a regexp value, then the result is `#t` if the regexp matches `(path->string sys-lib-subpath)`, `#f` otherwise.

Changed in version 6.3 of package `base`: Added `#:cross?` argument and changed the contract on *sys-lib-subpath* to accept `path-for-some-system?` instead of just `path?`.

## 6.14 API for Cross-Platform Configuration

`(require setup/cross-system)` package: `base`

The `setup/cross-system` library provides functions for querying the system properties of a destination platform, which can be different than the current platform in cross-installation modes.

A Racket installation includes a `"system.rktd"` file in the directory reported by `(find-lib-dir)`. When the information in that file does not match the running Racket's infor-

mation, then the setup/cross-system module infers that Racket is being run in cross-installation mode.

For example, if an in-place Racket installation for a different platform resides at ⟨*cross-dir*⟩, then

```
racket -G ⟨cross-dir⟩/etc -X ⟨cross-dir⟩/collects -l- raco pkg
```

runs raco pkg using the current platform's racket executable, but using the collections and other configuration information of ⟨*cross-dir*⟩, as well as modifying the packages of ⟨*cross-dir*⟩. That can work as long as no platform-specific libraries need to run to perform the requested raco pkg action (e.g., when installing built packages).

Added in version 6.3 of package base.

```
(cross-system-type [mode])
 → (or/c symbol? string? bytes? exact-positive-integer? vector?)
  mode : (or/c 'os 'word 'gc 'link 'machine    = 'os
               'so-suffix 'so-mode 'fs-change)
```

Like system-type, but for the target platform instead of the current platform in cross-installation mode. When not in cross-installation mode, the results are the same as for system-type.

```
(cross-system-library-subpath [mode]) → path-for-some-system?
  mode : (or/c 'cgc '3m #f) = (system-type 'gc)
```

Like system-library-subpath, but for the target platform instead of the current platform in cross-installation mode. When not in cross-installation mode, the results are the same as for system-library-subpath.

In cross-installation mode, the target platform may have a different path convention than the current platform, so the result is path-for-some-system? instead of path?.

```
(cross-installation?) → boolean?
```

Returns #t if cross-installation mode has been detected, #f otherwise.

## 6.15 API for Cross-References for Installed Manuals

```
(require setup/xref)      package: racket-index
```

```
(load-collections-xref [on-load]) → xref?
  on-load : (-> any/c) = (lambda () (void))
```

Like load-xref, but automatically find all cross-reference files for manuals that have been installed with raco setup.

A cached copy of cross-reference information can be used, in which case *on-load* is *not* called.

```
(make-collections-xref
 [#:no-user? no-user?
  #:no-main? no-main?
  #:doc-db db-path
  #:quiet-fail? quiet-fail?
  #:register-shutdown! register-shutdown!])
→ xref?
 no-user? : any/c = #f
 no-main? : any/c = #f
 db-path : (or/c #f path?) = #f
 quiet-fail? : any/c = #f
 register-shutdown! : ((-> any) . -> . any) = void
```

Like load-collections-xref, but takes advantage of a cross-reference database *db-path*, when support is available, to delay the loading of cross-reference details until needed.

Cross-reference information is skipped when it is installed in the main installation or in a user-specific location, respectively, if *no-main?* or *no-user?* is #t.

If *quiet-fail?* is true, then errors are suppressed while loading cross-reference information.

The *register-shutdown!* callback may be called to register a function that closes database connections when the result of make-collections-xref is no longer needed. If *register-shutdown!* is not supplied or if a function sent to *register-shutdown!* is never called, database connections will be closed only though a custodian.

```
(get-rendered-doc-directories no-user?
                              no-main?) → (listof path?)
  no-user? : any/c
  no-main? : any/c
```

Returns a list of directories for all documentation for all installed collections, omitting documentation that is installed in the main installation or in a user-specific location, respectively, if *no-main?* or *no-user?* is #t.

```
(get-current-doc-state) → doc-state?
```

Records the time stamps of files that are touched whenever the documentation is changed.

Added in version 1.2 of package racket-index.

```
(doc-state-changed? doc-state) → boolean?
  doc-state : doc-state?
```

Returns #t when the time stamps of the files in *doc-state* changed (or new files appeared) and #f otherwise.

If the result is #t, then the documentation in this installation of Racket has changed and otherwise it hasn't.

Added in version 1.2 of package racket-index.

```
(doc-state? v) → boolean?
  v : any/c
```

A predicate to recognize the result of get-current-doc-state.

Added in version 1.2 of package racket-index.

## 6.16 API for Materializing User-Specific Documentation

```
(require setup/materialize-user-docs)
                        package: racket-index
```

Added in version 1.1 of package racket-index.

```
(materialize-user-docs [on-setup]) → void?
  on-setup : ((-> boolean?) -> any) = (lambda (setup) (setup))
```

Checks whether a user-specific documentation entry point already exists in (find-user-doc-dir), and if not, runs raco setup in a mode that will create the entry point (to have the same content as the installation's documentation entry point.)

The run of raco setup is packaged in a thunk that is provided to *on-setup*, which can adjust the current output and error ports as appropriate and check the thunk's result for success.

The *on-setup* argument is not called if the documentation entry point already exists in (find-user-doc-dir).

# 7 `raco decompile`: Decompiling Bytecode

The `raco decompile` command takes the path of a bytecode file (which usually has the file extension `".zo"`) or a source file with an associated bytecode file (usually created with `raco make`) and converts the bytecode file's content back to an approximation of Racket code. Decompiled bytecode is mostly useful for checking the compiler's transformation and optimization of the source program.

The `raco decompile` command accepts the following command-line flags:

- `--force` — skip modification-date comparison on the given file's path and an associated `".zo"` file (if any)
- `-n` ⟨*n*⟩ or `--columns` ⟨*n*⟩ — format output for a display with ⟨*n*⟩ columns

Many forms in the decompiled code, such as `module`, `define`, and `lambda`, have the same meanings as always. Other forms and transformations are specific to the rendering of bytecode, and they reflect a specific execution model:

- Top-level variables, variables defined within the module, and variables imported from other modules are prefixed with `_`, which helps expose the difference between uses of local variables versus other variables. Variables imported from other modules, moreover, have a suffix starting with `@` that indicates the source module. Finally, imported variables with constantness have a midfix: `:c` to indicate constant shape across all instantiations, `:f` to indicate a fixed value after initialization, `:p` to indicate a procedure, `:P` to indicate a procedure that preserves continuation marks on return, `:t` to indicate a structure type, `:mk` to indicate a structure constructor, `:?` to indicate a structure predicate, `:ref` to indicate a structure accessor, or `:set!` to indicate a structure mutator.

  Non-local variables are always accessed indirectly though an implicit `#%globals` or `#%modvars` variable that resides on the value stack (which otherwise contains local variables). Variable accesses are further wrapped with `#%checked` when the compiler cannot prove that the variable will be defined before the access.

  Uses of core primitives are shown without a leading `_`, and they are never wrapped with `#%checked`.

- Local-variable access may be wrapped with `#%sfs-clear`, which indicates that the variable-stack location holding the variable will be cleared to prevent the variable's value from being retained by the garbage collector. Variables whose name starts with `unused` are never actually stored on the stack, and so they never have `#%sfs-clear` annotations. (The bytecode compiler normally eliminates such bindings, but sometimes it cannot, either because it cannot prove that the right-hand side produces the right number of values, or the discovery that the variable is unused happens too late with the compiler.)

Mutable variables are converted to explicitly boxed values using `#%box`, `#%unbox`, and `#%set-boxes!` (which works on multiple boxes at once). A `set!-rec-values` operation constructs mutually-recursive closures and simultaneously updates the corresponding variable-stack locations that bind the closures. A `set!`, `set!-values`, or `set!-rec-values` form is always used on a local variable before it is captured by a closure; that ordering reflects how closures capture values in variable-stack locations, as opposed to stack locations.

- In a `lambda` form, if the procedure produced by the `lambda` has a name (accessible via `object-name`) and/or source-location information, then it is shown as a quoted constant at the start of the procedure's body. Afterward, if the `lambda` form captures any bindings from its context, those bindings are also shown in a quoted constant. Neither constant corresponds to a computation when the closure is called, though the list of captured bindings corresponds to a closure allocation when the `lambda` form itself is evaluated.

  A `lambda` form that closes over no bindings is wrapped with `#%closed` plus an identifier that is bound to the closure. The binding's scope covers the entire decompiled output, and it may be referenced directly in other parts of the program; the binding corresponds to a constant closure value that is shared, and it may even contain cyclic references to itself or other constant closures.

- A form `(#%apply-values proc expr)` is equivalent to `(call-with-values (lambda () expr) proc)`, but the run-time system avoids allocating a closure for *expr*. Similarly, a `#%call-with-immediate-continuation-mark` call is equivalent to a `call-with-immediate-continuation-mark` call, but avoiding a closure allocation.

- A `define-values` form may have `(begin '%%inline-variant%% expr1 expr2)` for its expression, in which case *expr2* is the normal result, but *expr1* may be inlined for calls to the definition from other modules. Definitions of functions without an `'%%inline-variant%%` are never inlined across modules.

- Function arguments and local bindings that are known to have a particular type have names that embed the known type. For example, an argument might have a name that starts `argflonum` or a local binding might have a name that starts `flonum` to indicate a flonum value.

- A `#%decode-syntax` form corresponds to a syntax object.

## 7.1 API for Decompiling

```
(require compiler/decompile)
```
package: `compiler-lib`

```
(decompile top) → any/c
  top : compilation-top?
```

Consumes the result of parsing bytecode and returns an S-expression (as described above) that represents the compiled code.

## 7.2 API for Parsing Bytecode

```
(require compiler/zo-parse)
```
package: zo-lib

The compiler/zo-parse module re-exports compiler/zo-structs in addition to zo-parse.

```
(zo-parse [in]) → compilation-top?
  in : input-port? = (current-input-port)
```

Parses a port (typically the result of opening a ".zo" file) containing bytecode. Beware that the structure types used to represent the bytecode are subject to frequent changes across Racket versons.

The parsed bytecode is returned in a compilation-top structure. For a compiled module, the compilation-top structure will contain a mod structure. For a top-level sequence, it will normally contain a seq or splice structure with a list of top-level declarations and expressions.

The bytecode representation of an expression is closer to an S-expression than a traditional, flat control string. For example, an if form is represented by a branch structure that has three fields: a test expression, a "then" expression, and an "else" expression. Similarly, a function call is represented by an application structure that has a list of argument expressions.

Storage for local variables or intermediate values (such as the arguments for a function call) is explicitly specified in terms of a stack. For example, execution of an application structure reserves space on the stack for each argument result. Similarly, when a let-one structure (for a simple let) is executed, the value obtained by evaluating the right-hand side expression is pushed onto the stack, and then the body is evaluated. Local variables are always accessed as offsets from the current stack position. When a function is called, its arguments are passed on the stack. A closure is created by transferring values from the stack to a flat closure record, and when a closure is applied, the saved values are restored on the stack (though possibly in a different order and likely in a more compact layout than when they were captured).

When a sub-expression produces a value, then the stack pointer is restored to its location from before evaluating the sub-expression. For example, evaluating the right-hand size for a let-one structure may temporarily push values onto the stack, but the stack is restored to its pre-let-one position before pushing the resulting value and continuing with the body. In addition, a tail call resets the stack pointer to the position that follows the enclosing function's arguments, and then the tail call continues by pushing onto the stack the arguments

for the tail-called function.

Values for global and module-level variables are not put directly on the stack, but instead stored in "buckets," and an array of accessible buckets is kept on the stack. When a closure body needs to access a global variable, the closure captures and later restores the bucket array in the same way that it captured and restores a local variable. Mutable local variables are boxed similarly to global variables, but individual boxes are referenced from the stack and closures.

Quoted syntax (in the sense of `quote-syntax`) is treated like a global variable, because it must be instantiated for an appropriate phase. A `prefix` structure within a `compilation-top` or `mod` structure indicates the list of global variables and quoted syntax that need to be instantiated (and put into an array on the stack) before evaluating expressions that might use them.

```
(decode-module-binding binding name) → decoded-module-binding?
  binding : module-binding?
  name : symbol?
```

Given a compact-form representation of a module binding and the name from which the binding is mapped, returns a normalized form of the binding.

## 7.3 API for Marshaling Bytecode

```
(require compiler/zo-marshal)
```
package: `zo-lib`

```
(zo-marshal-to top out) → void?
  top : compilation-top?
  out : output-port?
```

Consumes a representation of bytecode and writes it to *out*.

```
(zo-marshal top) → bytes?
  top : compilation-top?
```

Consumes a representation of bytecode and generates a byte string for the marshaled bytecode.

## 7.4 Bytecode Representation

```
(require compiler/zo-structs)
```
package: `zo-lib`

The `compiler/zo-structs` library defines the bytecode structures that are produced by `zo-parse` and consumed by `decompile` and `zo-marshal`.

**Warning:** The `compiler/zo-structs` library exposes internals of the Racket bytecode abstraction. Unlike other Racket libraries, `compiler/zo-structs` is subject to incompatible changes across Racket versions.

```
(struct zo ()
    #:extra-constructor-name make-zo
    #:prefab)
```

A supertype for all forms that can appear in compiled code.

### 7.4.1 Prefix

```
(struct compilation-top zo (max-let-depth
                            binding-namess
                            prefix
                            code)
    #:extra-constructor-name make-compilation-top
    #:prefab)
  max-let-depth : exact-nonnegative-integer?
  binding-namess : (hash/c exact-nonnegative-integer?
                           (hash/c symbol? stx?))
  prefix : prefix?
  code : (or/c form? any/c)
```

Wraps compiled code.

The `max-let-depth` field indicates the maximum stack depth that `code` creates (not counting the `prefix` array).

The `binding-namess` field provides a per-phase mapping from symbols that appear in `prefix` for top-level `def-values` forms and in top-level `def-syntaxes` forms. Each symbol is mapped to an identifier that will be bound (after introduction into the namespace) by the definition.

The `prefix` field describes top-level variables, module-level variables, and quoted syntax-objects accessed by `code`.

The `code` field contains executable code; it is normally a `form`, but a literal value is represented as itself.

```
(struct prefix zo (num-lifts toplevels stxs src-inspector-desc)
    #:extra-constructor-name make-prefix
    #:prefab)
  num-lifts : exact-nonnegative-integer?
```

```
  toplevels : (listof (or/c #f symbol? global-bucket?
                            module-variable?))
  stxs : (listof (or stx? #f))
  src-inspector-desc : symbol?
```

Represents a “prefix” that is pushed onto the stack to initiate evaluation. The prefix is an array, where buckets holding the values for `toplevels` are first, then the buckets for the `stxs`, then a bucket for another array if `stxs` is non-empty, then `num-lifts` extra buckets for lifted local procedures.

In `toplevels`, each element is one of the following:

- a `#f`, which indicates a dummy variable that is used to access the enclosing module/namespace at run time;
- a symbol, which is a reference to a variable defined in the enclosing module;
- a `global-bucket`, which is a top-level variable (appears only outside of modules); or
- a `module-variable`, which indicates a variable imported from another module.

The variable buckets and syntax objects that are recorded in a prefix are accessed by `toplevel` and `topsyntax` expression forms.

When an element of `stxs` is `#f`, it coresponds to a syntax object that was optimized away at the last minute. The slot must not be referenced vt a `topsyntax` form.

The `src-inspector-desc` field provides an inspector name that is used within syntax-object bindings. At run time, the prefix gets an inspector, and bindings that reference the same inspector name are granted access capabilities through that inspector.

```
(struct global-bucket zo (name)
    #:extra-constructor-name make-global-bucket
    #:prefab)
  name : symbol?
```

Represents a top-level variable, and used only in a `prefix`.

```
(struct module-variable zo (modidx sym pos phase constantness)
    #:extra-constructor-name make-module-variable
    #:prefab)
  modidx : module-path-index?
  sym : symbol?
  pos : exact-integer?
  phase : exact-nonnegative-integer?
  constantness : (or/c #f 'constant 'fixed
                       function-shape? struct-shape?)
```

Represents a top-level variable, and used only in a `prefix`. The `pos` may record the variable's offset within its module, or it can be `-1` if the variable is always located by name. The `phase` indicates the phase level of the definition within its module. The `constantness` field is either `'constant`, a `function-shape` value, or a `struct-shape` value to indicate that variable's value is always the same for every instantiation of its module; `'fixed` to indicate that it doesn't change within a particular instantiation of the module; or `#f` to indicate that the variable's value can change even for one particular instantiation of its module.

```
(struct function-shape (arity preserves-marks?)
    #:extra-constructor-name make-function-shape
    #:prefab)
  arity : procedure-arity?
  preserves-marks? : boolean?
```

Represents the shape of an expected import, which should be a function having the arity specified by `arity`. The `preserves-marks?` field is true if calling the function is expected to leave continuation marks unchanged by the time it returns.

```
(struct struct-shape ()
    #:extra-constructor-name make-struct-shape
    #:prefab)
(struct struct-type-shape struct-shape (field-count)
    #:extra-constructor-name make-struct-type-shape
    #:prefab)
  field-count : exact-nonnegative-integer?
(struct constructor-shape struct-shape (arity)
    #:extra-constructor-name make-constructor-shape
    #:prefab)
  arity : exact-nonnegative-integer?
(struct predicate-shape struct-shape ()
    #:extra-constructor-name make-predicate-shape
    #:prefab)
(struct accessor-shape struct-shape (field-count)
    #:extra-constructor-name make-accessor-shape
    #:prefab)
  field-count : exact-nonnegative-integer?
(struct mutator-shape struct-shape (field-count)
    #:extra-constructor-name make-mutator-shape
    #:prefab)
  field-count : exact-nonnegative-integer?
(struct struct-other-shape struct-shape ()
    #:extra-constructor-name make-struct-other-shape
    #:prefab)
```

Represents the shape of an expected import as a structure-type binding, constructor, etc.

```
(struct stx zo (content)
    #:extra-constructor-name make-stx
    #:prefab)
  content : stx-obj?
```

Wraps a syntax object as it appears in a `prefix`.

### 7.4.2 Forms

```
(struct form zo ()
    #:extra-constructor-name make-form
    #:prefab)
```

A supertype for all forms that can appear in compiled code (including `expr`s), except for literals that are represented as themselves.

```
(struct def-values form (ids rhs)
    #:extra-constructor-name make-def-values
    #:prefab)
  ids : (listof toplevel?)
  rhs : (or/c expr? seq? inline-variant? any/c)
```

Represents a `define-values` form. Each element of `ids` will reference via the prefix either a top-level variable or a local module variable.

After `rhs` is evaluated, the stack is restored to its depth from before evaluating `rhs`.

```
(struct def-syntaxes form (ids rhs prefix max-let-depth dummy)
    #:extra-constructor-name make-def-syntaxes
    #:prefab)
  ids : (listof symbol?)
  rhs : (or/c expr? seq? any/c)
  prefix : prefix?
  max-let-depth : exact-nonnegative-integer?
  dummy : (or/c toplevel? #f)
(struct seq-for-syntax form (forms prefix max-let-depth dummy)
    #:extra-constructor-name make-seq-for-syntax
    #:prefab)
  forms : (listof (or/c form? any/c))
  prefix : prefix?
  max-let-depth : exact-nonnegative-integer?
  dummy : (or/c toplevel? #f)
```

Represents a `define-syntaxes` or `begin-for-syntax` form. The `rhs` expression or set of `forms` forms has its own `prefix`, which is pushed before evaluating `rhs` or the `forms`; the stack is restored after obtaining the result values. The `max-let-depth` field indicates the maximum size of the stack that will be created by `rhs` (not counting `prefix`). The `dummy` variable is used to access the enclosing namespace.

```
(struct req form (reqs dummy)
    #:extra-constructor-name make-req
    #:prefab)
  reqs : stx?
  dummy : toplevel?
```

Represents a top-level `#%require` form (but not one in a `module` form) with a sequence of specifications `reqs`. The `dummy` variable is used to access the top-level namespace.

```
(struct seq form (forms)
    #:extra-constructor-name make-seq
    #:prefab)
  forms : (listof (or/c form? any/c))
```

Represents a `begin` form, either as an expression or at the top level (though the latter is more commonly a `splice` form). When a `seq` appears in an expression position, its `forms` are expressions.

After each form in `forms` is evaluated, the stack is restored to its depth from before evaluating the form.

```
(struct splice form (forms)
    #:extra-constructor-name make-splice
    #:prefab)
  forms : (listof (or/c form? any/c))
```

Represents a top-level `begin` form where each evaluation is wrapped with a continuation prompt.

After each form in `forms` is evaluated, the stack is restored to its depth from before evaluating the form.

```
(struct inline-variant form (direct inline)
    #:extra-constructor-name make-inline-variant
    #:prefab)
  direct : expr?
  inline : expr?
```

Represents a function that is bound by `define-values`, where the function has two variants. The first variant is used for normal calls to the function. The second may be used for cross-module inlining of the function.

```
(struct mod form (name
                  srcname
                  self-modidx
                  prefix
                  provides
                  requires
                  body
                  syntax-bodies
                  unexported
                  max-let-depth
                  dummy
                  lang-info
                  internal-context
                  binding-names
                  flags
                  pre-submodules
                  post-submodules)
  #:extra-constructor-name make-mod
  #:prefab)
 name : (or/c symbol? (listof symbol?))
 srcname : symbol?
 self-modidx : module-path-index?
 prefix : prefix?
 provides : (listof (list/c (or/c exact-integer? #f)
                            (listof provided?)
                            (listof provided?)))
 requires : (listof (cons/c (or/c exact-integer? #f)
                            (listof module-path-index?)))
 body : (listof (or/c form? any/c))
 syntax-bodies : (listof (cons/c exact-positive-integer?
                                 (listof (or/c def-syntaxes?
                                               seq-for-syntax?))))
 unexported : (listof (list/c exact-nonnegative-integer?
                              (listof symbol?)
                              (listof symbol?)))
 max-let-depth : exact-nonnegative-integer?
 dummy : toplevel?
 lang-info : (or/c #f (vector/c module-path? symbol? any/c))
 internal-context : (or/c #f #t stx? (vectorof stx?))
 binding-names : (hash/c exact-integer?
                         (hash/c symbol? (or/c #t stx?)))
 flags : (listof (or/c 'cross-phase))
 pre-submodules : (listof mod?)
 post-submodules : (listof mod?)
```

Represents a `module` declaration.

The `provides` and `requires` lists are each an association list from phases to exports or imports. In the case of `provides`, each phase maps to two lists: one for exported variables, and another for exported syntax. In the case of `requires`, each phase maps to a list of imported module paths.

The `body` field contains the module's run-time (i.e., phase 0) code. The `syntax-bodies` list has a list of forms for each higher phase in the module body; the phases are in order starting with phase 1. The `body` forms use `prefix`, rather than any prefix in place for the module declaration itself, while members of lists in `syntax-bodies` have their own prefixes. After each form in `body` or `syntax-bodies` is evaluated, the stack is restored to its depth from before evaluating the form.

The `unexported` list contains lists of symbols for unexported definitions that can be accessed through macro expansion and that are implemented through the forms in `body` and `syntax-bodies`. Each list in `unexported` starts with a phase level.

The `max-let-depth` field indicates the maximum stack depth created by `body` forms (not counting the `prefix` array). The `dummy` variable is used to access to the top-level namespace.

The `lang-info` value specifies an optional module path that provides information about the module's implementation language.

The `internal-context` value describes the lexical context of the body of the module. This value is used by `module->namespace`. A `#f` value means that the context is unavailable or empty. A `#t` value means that the context is computed by re-importing all required modules. A syntax-object value embeds an arbitrary lexical context.

The `binding-names` value provides additional information to `module->namespace` to correlate symbol names for variables and syntax definitions to identifiers that map to those variables. A separate table of names exists for each phase, and a `#t` mapping for a name indicates that it is mapped but inaccessible (because the relevant scopes are inaccessible).

The `flags` field records certain properties of the module. The `'cross-phase` flag indicates that the module body is evaluated once and the results shared across instances for all phases; such a module contains only definitions of functions, structure types, and structure type properties.

The `pre-submodules` field records `module`-declared submodules, while the `post-submodules` field records `module*`-declared submodules.

```
(struct provided zo (name
                     src
                     src-name
                     nom-src
                     src-phase
                     protected?)
```

```
    #:extra-constructor-name make-provided
    #:prefab)
  name : symbol?
  src : (or/c module-path-index? #f)
  src-name : symbol?
  nom-src : (or/c module-path-index? #f)
  src-phase : exact-nonnegative-integer?
  protected? : boolean?
```

Describes an individual provided identifier within a `mod` instance.

### 7.4.3 Expressions

```
(struct expr form ()
    #:extra-constructor-name make-expr
    #:prefab)
```

A supertype for all expression forms that can appear in compiled code, except for literals that are represented as themselves and some `seq` structures (which can appear as an expression as long as it contains only other things that can be expressions).

```
(struct lam expr (name
                  flags
                  num-params
                  param-types
                  rest?
                  closure-map
                  closure-types
                  toplevel-map
                  max-let-depth
                  body)
    #:extra-constructor-name make-lam
    #:prefab)
  name : (or/c symbol? vector?)
  flags : (listof (or/c 'preserves-marks 'is-method 'single-result
                        'only-rest-arg-not-used 'sfs-clear-rest-args))
  num-params : exact-nonnegative-integer?
  param-types : (listof (or/c 'val 'ref 'flonum 'fixnum 'extflonum))
  rest? : boolean?
  closure-map : (vectorof exact-nonnegative-integer?)
  closure-types : (listof (or/c 'val/ref 'flonum 'fixnum 'extflonum))
  toplevel-map : (or/c #f (set/c exact-nonnegative-integer?))
  max-let-depth : exact-nonnegative-integer?
  body : (or/c expr? seq? any/c)
```

Represents a `lambda` form. The `name` field is a name for debugging purposes. The `num-params` field indicates the number of arguments accepted by the procedure, not counting a rest argument; the `rest?` field indicates whether extra arguments are accepted and collected into a "rest" variable. The `param-types` list contains `num-params` symbols indicating the type of each argumet, either `'val` for a normal argument, `'ref` for a boxed argument (representing a mutable local variable), `'flonum` for a flonum argument, or `'extflonum` for an extflonum argument.

The `closure-map` field is a vector of stack positions that are captured when evaluating the `lambda` form to create a closure. The `closure-types` field provides a corresponding list of types, but no distinction is made between normal values and boxed values; also, this information is redundant, since it can be inferred by the bindings referenced though `closure-map`.

When a closure captures top-level or module-level variables or refers to a syntax-object constant, the variables and constants are represented in the closure by capturing a prefix (in the sense of `prefix`). The `toplevel-map` field indicates which top-level and lifted variables are actually used by the closure (so that variables in a prefix can be pruned by the run-time system if they become unused) and whether any syntax objects are used (so that the syntax objects as a group can be similarly pruned). A `#f` value indicates either that no prefix is captured or all variables and syntax objects in the prefix should be considered used. Otherwise, numbers in the set indicate which variables and lifted variables are used. Variables are numbered consecutively by position in the prefix starting from `0`, but the number equal to the number of non-lifted variables corresponds to syntax objects (i.e., the number is include if any syntax-object constant is used). Lifted variables are numbered immediately afterward—which means that, if the prefix contains any syntax objects, lifted-variable numbers are shifted down relative to a `toplevel` by the number of syntax object in the prefix (which makes the `toplevel-map` set more compact).

When the function is called, the rest-argument list (if any) is pushed onto the stack, then the normal arguments in reverse order, then the closure-captured values in reverse order. Thus, when `body` is run, the first value on the stack is the first value captured by the `closure-map` array, and so on.

The `max-let-depth` field indicates the maximum stack depth created by `body` plus the arguments and closure-captured values pushed onto the stack. The `body` field is the expression for the closure's body.

Changed in version 6.1.1.8 of package `zo-lib`: Added a number to `toplevel-map` to indicate whether any syntax object is used, shifting numbers for lifted variables up by one if any syntax object is in the prefix.

```
(struct closure expr (code gen-id)
    #:extra-constructor-name make-closure
    #:prefab)
  code : lam?
  gen-id : symbol?
```

A `lambda` form with an empty closure, which is a procedure constant. The procedure constant can appear multiple times in the graph of expressions for bytecode, and the `code` field can be a cycle for a recursive constant procedure; the `gen-id` is different for each such constant.

```
(struct case-lam expr (name clauses)
    #:extra-constructor-name make-case-lam
    #:prefab)
  name : (or/c symbol? vector?)
  clauses : (listof lam?)
```

Represents a `case-lambda` form as a combination of `lambda` forms that are tried (in order) based on the number of arguments given.

```
(struct let-one expr (rhs body type unused?)
    #:extra-constructor-name make-let-one
    #:prefab)
  rhs : (or/c expr? seq? any/c)
  body : (or/c expr? seq? any/c)
  type : (or/c #f 'flonum 'fixnum 'extflonum)
  unused? : boolean?
```

Pushes an uninitialized slot onto the stack, evaluates `rhs` and puts its value into the slot, and then runs `body`. If `type` is not `#f`, then `rhs` must produce a value of the corresponding type, and the slot must be accessed by `localref`s that expect the type. If `unused?` is `#t`, then the slot must not be used, and the value of `rhs` is not actually pushed onto the stack (but `rhs` is constrained to produce a single value).

After `rhs` is evaluated, the stack is restored to its depth from before evaluating `rhs`. Note that the new slot is created before evaluating `rhs`.

```
(struct let-void expr (count boxes? body)
    #:extra-constructor-name make-let-void
    #:prefab)
  count : exact-nonnegative-integer?
  boxes? : boolean?
  body : (or/c expr? seq? any/c)
```

Pushes `count` uninitialized slots onto the stack and then runs `body`. If `boxes?` is `#t`, then the slots are filled with boxes that contain `#<undefined>`.

```
(struct install-value expr (count pos boxes? rhs body)
    #:extra-constructor-name make-install-value
    #:prefab)
  count : exact-nonnegative-integer?
  pos : exact-nonnegative-integer?
```

```
  boxes? : boolean?
  rhs : (or/c expr? seq? any/c)
  body : (or/c expr? seq? any/c)
```

Runs `rhs` to obtain `count` results, and installs them into existing slots on the stack in order, skipping the first `pos` stack positions. If `boxes?` is `#t`, then the values are put into existing boxes in the stack slots.

After `rhs` is evaluated, the stack is restored to its depth from before evaluating `rhs`.

```
(struct let-rec expr (procs body)
    #:extra-constructor-name make-let-rec
    #:prefab)
  procs : (listof lam?)
  body : (or/c expr? seq? any/c)
```

Represents a `letrec` form with `lambda` bindings. It allocates a closure shell for each `lambda` form in `procs`, installs each onto the stack in previously allocated slots in reverse order (so that the closure shell for the last element of `procs` is installed at stack position `0`), fills out each shell's closure (where each closure normally references some other just-created closures, which is possible because the shells have been installed on the stack), and then evaluates `body`.

```
(struct boxenv expr (pos body)
    #:extra-constructor-name make-boxenv
    #:prefab)
  pos : exact-nonnegative-integer?
  body : (or/c expr? seq? any/c)
```

Skips `pos` elements of the stack, setting the slot afterward to a new box containing the slot's old value, and then runs `body`. This form appears when a `lambda` argument is mutated using `set!` within its body; calling the function initially pushes the value directly on the stack, and this form boxes the value so that it can be mutated later.

```
(struct localref expr (unbox? pos clear? other-clears? type)
    #:extra-constructor-name make-localref
    #:prefab)
  unbox? : boolean?
  pos : exact-nonnegative-integer?
  clear? : boolean?
  other-clears? : boolean?
  type : (or/c #f 'flonum 'fixnum 'extflonum)
```

Represents a local-variable reference; it accesses the value in the stack slot after the first `pos` slots. If `unbox?` is `#t`, the stack slot contains a box, and a value is extracted from the box.

If clear? is #t, then after the value is obtained, the stack slot is cleared (to avoid retaining a reference that can prevent reclamation of the value as garbage). If other-clears? is #t, then some later reference to the same stack slot may clear after reading. If type is not #f, the slot is known to hold a specific type of value.

```
(struct toplevel expr (depth pos const? ready?)
    #:extra-constructor-name make-toplevel
    #:prefab)
  depth : exact-nonnegative-integer?
  pos : exact-nonnegative-integer?
  const? : boolean?
  ready? : boolean?
```

Represents a reference to a top-level or imported variable via the prefix array. The depth field indicates the number of stack slots to skip to reach the prefix array, and pos is the offset into the array.

When the toplevel is an expression, if both const? and ready? are #t, then the variable definitely will be defined, its value stays constant, and the constant is effectively the same for every module instantiation. If only const? is #t, then the value is constant, but it may vary across instantiations. If only ready? is #t, then the variable definitely will be defined, but its value may change. If const? and ready? are both #f, then a check is needed to determine whether the variable is defined.

When the toplevel is the right-hand side for def-values, then const? is #f. If ready? is #t, the variable is marked as immutable after it is defined.

```
(struct topsyntax expr (depth pos midpt)
    #:extra-constructor-name make-topsyntax
    #:prefab)
  depth : exact-nonnegative-integer?
  pos : exact-nonnegative-integer?
  midpt : exact-nonnegative-integer?
```

Represents a reference to a quoted syntax object via the prefix array. The depth field indicates the number of stack slots to skip to reach the prefix array, and pos is the offset into the array. The midpt value is used internally for lazy calculation of syntax information.

```
(struct application expr (rator rands)
    #:extra-constructor-name make-application
    #:prefab)
  rator : (or/c expr? seq? any/c)
  rands : (listof (or/c expr? seq? any/c))
```

Represents a function call. The rator field is the expression for the function, and rands are the argument expressions. Before any of the expressions are evaluated, (length rands) uninitialized stack slots are created (to be used as temporary space).

```
(struct branch expr (test then else)
    #:extra-constructor-name make-branch
    #:prefab)
  test : (or/c expr? seq? any/c)
  then : (or/c expr? seq? any/c)
  else : (or/c expr? seq? any/c)
```

Represents an `if` form.

After `test` is evaluated, the stack is restored to its depth from before evaluating `test`.

```
(struct with-cont-mark expr (key val body)
    #:extra-constructor-name make-with-cont-mark
    #:prefab)
  key : (or/c expr? seq? any/c)
  val : (or/c expr? seq? any/c)
  body : (or/c expr? seq? any/c)
```

Represents a `with-continuation-mark` expression.

After each of `key` and `val` is evaluated, the stack is restored to its depth from before evaluating `key` or `val`.

```
(struct beg0 expr (seq)
    #:extra-constructor-name make-beg0
    #:prefab)
  seq : (listof (or/c expr? seq? any/c))
```

Represents a `begin0` expression.

After each expression in `seq` is evaluated, the stack is restored to its depth from before evaluating the expression.

Unlike the `begin0` source form, the first expression in `seq` is never in tail position, even if it is the only expression in the list.

```
(struct varref expr (toplevel dummy)
    #:extra-constructor-name make-varref
    #:prefab)
  toplevel : (or/c toplevel? #t)
  dummy : (or/c toplevel? #f)
```

Represents a `#%variable-reference` form. The `toplevel` field is `#t` if the original reference was to a constant local binding. The `dummy` field accesses a variable bucket that strongly references its namespace (as opposed to a normal variable bucket, which only weakly references its namespace); it can be `#f`.

```
(struct assign expr (id rhs undef-ok?)
    #:extra-constructor-name make-assign
    #:prefab)
  id : toplevel?
  rhs : (or/c expr? seq? any/c)
  undef-ok? : boolean?
```

Represents a `set!` expression that assigns to a top-level or module-level variable. (Assignments to local variables are represented by `install-value` expressions.) If `undef-ok?` is true, the assignment to `id` succeeds even if `id` was not previously defined (see also `compile-allow-set!-undefined`).

After `rhs` is evaluated, the stack is restored to its depth from before evaluating `rhs`.

```
(struct apply-values expr (proc args-expr)
    #:extra-constructor-name make-apply-values
    #:prefab)
  proc : (or/c expr? seq? any/c)
  args-expr : (or/c expr? seq? any/c)
```

Represents `(call-with-values (lambda () args-expr) proc)`, which is handled specially by the run-time system.

```
(struct with-immed-mark expr (key val body)
    #:extra-constructor-name make-with-immed-mark
    #:prefab)
  key : (or/c expr? seq? any/c)
  val : (or/c expr? seq? any/c)
  body : (or/c expr? seq? any/c)
```

Represents a `(call-with-immediate-continuation-mark key (lambda (arg) body) val)` expression that is handled specially by the run-time system to avoid a closure allocation. One initialized slot is pushed onto the stack after `expr` and `val` are evaluated and before `body` is evaluated.

After each of `key` and `val` is evaluated, the stack is restored to its depth from before evaluating `key` or `val`.

```
(struct primval expr (id)
    #:extra-constructor-name make-primval
    #:prefab)
  id : exact-nonnegative-integer?
```

Represents a direct reference to a variable imported from the run-time kernel.

### 7.4.4 Syntax Objects

```
(struct stx-obj zo (datum wrap srcloc props tamper-status)
    #:extra-constructor-name make-stx-obj
    #:prefab)
  datum : any/c
  wrap : wrap?
  srcloc : (or/c #f srcloc?)
  props : (hash/c symbol? any/c)
  tamper-status : (or/c 'clean 'armed 'tainted)
```

Represents a syntax object, where `wrap` contains lexical information, `srcloc` is the source location, `props` contains preserved properties, and `tamper-status` is taint information. When the `datum` part is itself compound, its pieces are wrapped as `stx-obj`s, too.

The content of `wrap` is typically cyclic, since it includes scopes that contain bindings that refer to scopes.

```
(struct wrap zo (shifts simple-scopes multi-scopes)
    #:extra-constructor-name make-wrap
    #:prefab)
  shifts : (listof module-shift?)
  simple-scopes : (listof scope?)
  multi-scopes : (listof (list/c multi-scope? (or/c #f exact-integer?)))
```

Lexical information for a syntax object. The `shifts` field allows binding information to be relative to the enclosing module's run-time path. The `simple-scopes` field records scopes that are attached to the syntax object at all phases, and `multi-scopes` records phase-specific scopes (which are always attached as a group) along with a phase shift for every scope within the group).

```
(struct module-shift zo (from
                         to
                         from-inspector-desc
                         to-inspector-desc)
    #:extra-constructor-name make-module-shift
    #:prefab)
  from : (or/c #f module-path-index?)
  to : (or/c #f module-path-index?)
  from-inspector-desc : (or/c #f symbol?)
  to-inspector-desc : (or/c #f symbol?)
```

Records a history of module path index replacements. These replacements are applied in reverse order, and a module instantiation typically adds one more shift to replace the current

"self" module path index with a run-time module path. The `from` and `to` fields should be both `#f` or both non-`#f`.

The `from-inspector-desc` and `to-inspector-desc` fields similarly should be both `#f` or both non-`#f`. They record a history of code-inspector replacements.

```
(struct scope zo (name kind bindings bulk-bindings multi-owner)
    #:extra-constructor-name make-scope
    #:prefab)
  name : (or/c 'root exact-nonnegative-integer?)
  kind : symbol?
  bindings : (listof (list/c symbol? (listof scope?) binding?))
  bulk-bindings : (listof (list/c (listof scope?) all-from-module?))
  multi-owner : (or/c #f multi-scope?)
```

Represents a scope. When `name` is `'root` then the scope represents the unique all-phases scope that is shared among non-module namespaces. Otherwise, `name` is intended to be distinct for each `scope` instance within a module or top-level compilation, but the `eq?`-identity of the `scope` instance ultimately determines its identity. The `kind` symbol similarly acts as a debugging hint in the same way as for `syntax-debug-info`.

The `bindings` list indicates some bindings that are associated with the scope. Each element of the list includes a symbolic name, a list of scopes (including the enclosing one), and the binding for the combination of name and scope set. A given symbol can appear in multiple elements of `bindings`, but the combination of the symbol and scope set are unique within `bindings` and across all scopes. The mapping of a symbol and scope set to a binding is recorded with an arbitrary member of the scope set.

The `bulk-bindings` field lists bindings of all exports from a given module, which is an optimization over including each export in `bindings`. Elements of `bindings` take precedence over elements of `bulk-bindings`, and earlier elements of `bulk-bindings` take precedence over later elements.

If the `scope` represents a scope at a particular phase for a group of phase-specific scopes, `mark-owner` refers to the group.

```
(struct multi-scope zo (name src-name scopes)
    #:extra-constructor-name make-multi-scope
    #:prefab)
  name : exact-nonnegative-integer?
  src-name : any/c
  scopes : (listof (list/c (or/c #f exact-integer?) scope?))
```

Represents a set of phase-specific scopes that are added or removed from lexical information as a group. As for `scope`, the `name` field is intended to be distinct for different groups, but the `eq?` identity of the `multi-scope` record ultimately determines its identity. The `src-name` field similarly acts as a debugging hint in the same way as for `syntax-debug-info`.

Scopes within the group are instantiated at different phases on demand. The `scopes` field lists all of the scopes instantiated for the group, and the phase at which it is instantiated. Each element of `scopes` must have a `multi-owner` field value that refers back to the `multi-scope`.

```
(struct binding zo ()
    #:extra-constructor-name make-binding
    #:prefab)
```

A supertype for all binding representations.

```
(struct module-binding binding (encoded)
    #:extra-constructor-name make-module-binding
    #:prefab)
  encoded : any/c
```

Represents a binding to a module or top-level definition. The `encoded` field can be unpacked using `decode-module-binding`, providing the symbol name for which the binding is the target (since `encoded` can be relative to that name).

```
(struct decoded-module-binding binding (path
                                        name
                                        phase
                                        nominal-path
                                        nominal-export-name
                                        nominal-phase
                                        import-phase
                                        inspector-desc)
    #:extra-constructor-name make-decoded-module-binding
    #:prefab)
  path : (or/c #f module-path-index?)
  name : symbol?
  phase : exact-integer?
  nominal-path : (or/c #f module-path-index?)
  nominal-export-name : symbol?
  nominal-phase : (or/c #f exact-integer?)
  import-phase : (or/c #f exact-integer?)
  inspector-desc : (or/c #f symbol?)
```

Represents a binding to a module or top-level definition—like `module-binding`, but in normalized form:

- `path`: the referenced module.
- `name`: the referenced definition within its module.

- `phase`: the phase of the referenced definition within its module.
- `nominal-path`: the module that was explicitly imported into the binding context; this path can be different from `path` when a definition is re-exported.
- `nominal-export-name`: the name of the binding as exported from `nominal-path`, which can be different from `name` due to renaming on export.
- `nominal-phase`: the phase of the export from `nominal-path`, which can be different from `phase` due to re-export from a module that imports at a phase level other than `0`.
- `import-phase`: the phase of the import of `nominal-path`, which shifted (if non-`0`) the binding phase relative to the export phase from `nominal-path`.
- `inspector-desc`: a name for an inspector (mapped to a specific inspector at run time) that determines access to the definition.

```
(struct local-binding binding (name)
    #:extra-constructor-name make-local-binding
    #:prefab)
  name : symbol?
```

Represents a local binding (i.e., not at the top level or module level). Such bindings rarely appear in bytecode, since `quote-syntax` prunes them.

```
(struct free-id=?-binding binding (base id phase)
    #:extra-constructor-name make-free-id=?-binding
    #:prefab)
  base : (and/c binding?
                (not/c free-id=?-binding?))
  id : stx-obj?
  phase : (or/c #f exact-integer?)
```

Represents a binding that includes a `free-identifier=?` alias (to an identifier with a particular phase shift) as well as a base binding.

```
(struct all-from-module zo (path
                            phase
                            src-phase
                            inspector-desc
                            exceptions
                            prefix)
    #:extra-constructor-name make-all-from-module
    #:prefab)
  path : module-path-index?
  phase : (or/c exact-integer? #f)
  src-phase : (or/c exact-integer? #f)
```

```
inspector-desc : symbol?
exceptions : (listof symbol?)
prefix : (or/c symbol? #f)
```

Describes a bulk import as an optimization over individual imports of a module's exports:

- `path`: the imported module.
- `phase`: the phase of the import module's exports.
- `src-phase`: the phase at which `path` was imported; `src-phase` combined with `phase` determines the phase of the bindings.
- `inspector-desc`: a name for an inspector (mapped to a specific inspector at run time) that determines access to the definition.
- `exceptions`: exports of `path` that are omitted from the bulk import.
- `prefix`: a prefix, if any, applied (after `exceptions`) to each of the imported names.

# 8 `raco demod`: Demodularizing Programs

The `raco demod` command takes a racket module and flattens all of its dependencies into a single compiled module. A file "⟨*name*⟩`.rkt`" is demodularized into "⟨*name*⟩`_rkt_merged.zo`".

The demodularized zo file can be run by passing it as an argument to the racket command-line program.

# 9 `raco link`: Library Collection Links

The `raco link` command inspects and modifies a collection links file to display, add, or remove mappings from collection names to filesystem directories.

Managing links directly is somewhat discouraged. Instead, use the package manager (see *Package Management in Racket*), which installs and manages links (i.e., it builds on `raco link`) in a way that more gracefully leads to sharing collections with others. Nevertheless, `raco link` is available for direct use.

For example, the command

```
raco link maze
```

installs a user-specific and version-specific link for the `"maze"` collection, mapping it to the `"maze"` subdirectory of the current directory. Supply multiple directory paths to create multiple links at once, especially with a command-shell wildcard:

```
raco link *
```

By default, the linked collection name is the same as each directory's name, but the collection name can be set separately for a single directory with the `--name` flag.

To remove the link created by the first example above, use

```
raco link --remove maze
```

or

```
raco link -r maze
```

Like link-adding mode, removing mode accepts multiple directory paths to remove multiple links, and all links that match any directory are removed. If `--name` is used with `--remove`, then only links matching both the collection name and directory are removed.

Full command-line options:

- `-l` or `--list` — Shows the current link table. If any other command-line arguments are provided that modify the link table, the table is shown after modifications. If no directory arguments are provided, and if none of `-u`, `--user`, `-i`, `--installation`, `-f`, or `--file` are specified, then the link table is shown for all user-specific and installation-wide collection links files.

- `-n` ⟨*name*⟩ or `--name` ⟨*name*⟩ — Sets the collection name for adding a single link or removing matching links. By default, the collection name for an added link is derived from the directory name. When the `-r` or `--remove` flag is also used, only links with a collection name matching ⟨*name*⟩ are removed, and if no directory arguments

are provided, all links with a match to ⟨*name*⟩ are removed. This flag is mutually exclusive with `-d` and `--root`.

- `-d` or `--root` — Treats each directory as a collection root that contains collection directories, instead of a directory for a specific collection. When the `-r` or `--remove` flag is also used, only collection-root links that match a directory are removed. This flag is mutually exclusive with `-n` and `--name`.

- `-D` or `--static-root` — Like `-d` or `--root`, but each directory is assumed to have a constant set of subdirectories (to improve the use of collection-search caches) as long as the links file itself does not change.

- `-x` ⟨*regexp*⟩ or `--version-regexp` ⟨*regexp*⟩ — Sets a version regexp that limits the link to use only by Racket versions (as reported by `version`) matching ⟨*regexp*⟩. This flag is normally used with `-u` or `--user` with installations that have different versions but the same installation name. When the `-r` or `--remove` flag is also used, only links with a version regexp matching ⟨*regexp*⟩ are removed.

- `-r` or `--remove` — Selects remove mode instead of add mode.

- `-u` or `--user` — Limits listing and removal of links to the user-specific collection links file and not the installation-wide collection links file. This flag is mutually exclusive with `-i`, `--installation`, `-f`, and `--file`.

- `-i` or `--installation` — Reads and writes links in installation-wide collection links file and not the user-specific collection links file. This flag is mutually exclusive with `-u`, `--user`, `-f`, and `--file`.

- `-f` ⟨*file*⟩ or `--file` ⟨*file*⟩ — Reads and writes links in ⟨*file*⟩ instead of the user-specific collection links file. This flag is mutually exclusive with `-u`, `--user`, `-s`, `--shared`, `-i`, and `--installation`.

- `-v` ⟨*vers*⟩ or `--version` ⟨*vers*⟩ — Selects ⟨*vers*⟩ as relevant installation name for operations on the user-specific collection links file.

- `--repair` — Enables repairs to the existing file content when the content is erroneous. The file is repaired by deleting individual links when possible.

## 9.1 API for Collection Links

`(require setup/link)` package: `base`

```
(links  dir
        ...
       [#:user? user?
        #:user-version user-version
        #:file file
        #:name name
        #:root? root?
        #:static-root? static-root?
        #:version-regexp version-regexp
        #:error error-proc
        #:remove? remove?
        #:show? show?
        #:repair? repair?
        #:with-path? with-path?])        → list?
  dir : path?
  user? : any/c = #t
  user-version : string? = (get-installation-name)
  file : (or/c path-string? #f) = #f
  name : (or/c string? #f) = #f
  root? : any/c = #f
  static-root? : any/c = #f
  version-regexp : (or/c regexp? #f) = #f
  error-proc : (symbol? string? any/c ... . -> . any) = error
  remove? : any/c = #f
  show? : any/c = #f
  repair? : any/c = #f
  with-path? : any/c = #f
```

A function version of the `raco link` command that always works on a single file—either *file* if it is a path string, the user–specific collection links file if *user?* is true, or the installation-wide collection links file otherwise. If *user?* is true, then *user-version* determines the relevant installation name (defaulting to the current installation's name).

The *static-root?* flag value is ignored unless *root?* is true and *remove?* is false, in which case each given *dir* is added as a static root if *static-root?* is true.

The *error-proc* argument is called to raise exceptions that would be fatal to the `raco link` command.

If *remove?* is true, the result is a list of entries that were removed from the file. If *remove?* is `#f` but *root?* is true, the result is a list of paths for collection roots. If *remove?* and *root?* are both `#f`, the result is a list for top-level collections that are mapped by *file* and that apply to the running version of Racket; the list is a list of strings for collection names if *with-path?* is `#f`, or it is a list of pairs of collection-name strings and complete paths if *with-path?* is true.

# 10 `raco pack`: Packing Library Collections

The `raco pack` command creates an archive of files and directories. Formerly, such archives were used directly to distribute library files to Racket users, but the package manager (see *Package Management in Racket*) is now the preferred mechanism for distribution.

A packed archive usually has the suffix `".plt"`. The `raco pkg` command recognizes a `".plt"` archive for installation as a package. The `raco setup` command (see §6 "`raco setup`: Installation Management") also supports `".plt"` unpacking and installation when using the `-A` flag, but such installations do not benefit from the more general management facilities of `raco pkg`, while the `raco unpack` command (see §11 "`raco unpack`: Unpacking Library Collections") unpacks an archive locally without attempting to install it. DrRacket recognizes the `".plt"` and currently treats such an archive in the same way as `raco setup -A`.

An archive contains the following elements:

- A set of files and directories to be unpacked, and flags indicating whether they are to be unpacked relative to the Racket add-ons directory (which is user-specific), the Racket installation directory, or a user-selected directory.

  The files and directories for an archive are provided on the command line to `raco pack`, either directly or in the form of collection names when the `--collect` flag is used.

  The `--at-plt` flag indicates that the files and directories should be unpacked relative to the user's add-ons directory, unless the user specifies the Racket installation directory when unpacking. The `--collection-plt` flag implies `--at-plt`. The `--all-users` flag overrides `--at-plt`, and it indicates that the files and directories should be unpacked relative to the Racket installation directory, always.

- A flag for each file indicating whether it overwrites an existing file when the archive is unpacked; the default is to leave the old file in place, but the `--replace` flag enables replacing for all files in the archive.

- A list of collections to be set-up (via `raco setup`) after the archive is unpacked; the `++setup` flag adds a collection name to the archive's list, but each collection for `--collection-plt` is added automatically.

- A name for the archive, which is reported to the user by the unpacking interface; the `--plt-name` flag sets the archive's name, but a default name is determined automatically when using `--collect`.

- A list of required collections (with associated version numbers) and a list of conflicting collections; the `raco pack` command always names the `"racket"` collection in the required list (using the collection's pack-time version), `raco pack` names each packed collection in the conflict list (so that a collection is not unpacked on top of a

different version of the same collection), and `raco pack` extracts other requirements and conflicts from the `"info.rkt"` files of collections when using `--collect`.

Specify individual directories and files for the archive when not using `--collect`. Each file and directory must be specified with a relative path. By default, if the archive is unpacked with DrRacket, the user will be prompted for a target directory, and if `raco setup` is used to unpack the archive, the files and directories will be unpacked relative to the current directory. If the `--at-plt` flag is provided, the files and directories will be unpacked relative to the user's Racket add-ons directory, instead. Finally, if the `--all-users` flag is provided, the files and directories will be unpacked relative to the Racket installation directory, instead.

Use the `--collect` flag to pack one or more collections; sub-collections can be designated by using a `/` as a path separator on all platforms. In this mode, `raco pack` automatically uses paths relative to the Racket installation or add-ons directory for the archived files, and the collections will be set-up after unpacking. In addition, `raco pack` consults each collection's `"info.rkt"` file, as described below, to determine the set of required and conflicting collections. Finally, `raco pack` consults the first collection's `"info.rkt"` file to obtain a default name for the archive. For example, the following command creates a `"sirmail.plt"` archive for distributing a `"sirmail"` collection:

```
raco pack --collect sirmail.plt sirmail
```

When packing collections, `raco pack` checks the following fields of each collection's `"info.rkt"` file (see §6.4 "`"info.rkt"` File Format"):

- `requires` — A list of the form `(list (list coll vers) ...)` where each *coll* is a non-empty list of relative-path strings, and each *vers* is a (possibly empty) list of exact integers. The indicated collections must be installed at unpacking time, with version sequences that match as much of the version sequence specified in the corresponding `vers`.

  A collection's version is indicated by a `version` field in its `"info.rkt"` file, and the default version is the empty list. The version sequence generalized major and minor version numbers. For example, version `'(2 5 4 7)` of a collection can be used when any of `'()`, `'(2)`, `'(2 5)`, `'(2 5 4)`, or `'(2 5 4 7)` is required.

- `conflicts` — A list of the form `(list coll ...)` where each *coll* is a non-empty list of relative-path strings. The indicated collections must *not* be installed at unpacking time.

For example, the `"info.rkt"` file in the `"sirmail"` collection might contain the following `info` declaration:

```
#lang info
(define name "SirMail")
(define mred-launcher-libraries (list "sirmail.rkt"))
```

```
(define mred-launcher-names (list "SirMail"))
(define requires (list (list "mred")))
```

Then, the "sirmail.plt" file (created by the command-line example above) will contain the name "SirMail." When the archive is unpacked, the unpacker will check that the "mred" collection is installed, and that "mred" has the same version as when "sirmail.plt" was created.

## 10.1 Format of ".plt" Archives

The extension ".plt" is not required for a distribution archive, but the ".plt"-extension convention helps users identify the purpose of a distribution file.

The raw format of a distribution file is described below. This format is uncompressed and sensitive to communication modes (text vs. binary), so the distribution format is derived from the raw format by first compressing the file using gzip, then encoding the gzipped file with the MIME base64 standard (which relies only the characters A-Z, a-z, 0-9, +, /, and =; all other characters are ignored when a base64-encoded file is decoded).

The raw format is

- PLT are the first three characters.
- An S-expression matching

```
(lambda (request failure)
  (case request
    [(name) name]
    [(unpacker) (quote mzscheme)]
    [(requires) (quote requires)]
    [(conflicts) (quote conflicts)]
    [(plt-relative?) plt-relative?]
    [(plt-home-relative?) plt-home-relative?]
    [(test-plt-dirs) test-dirs]
    [else (failure)]))
```

  where the *name*, *requires*, etc., meta-variables stand for S-expressions as follows:

  - *name* — a human-readable string describing the archive's contents. This name is used only for printing messages to the user during unpacking.
  - *requires* — a list of collections required to be installed before unpacking the archive, which associated versions; see the documentation of pack for details.
  - *conflicts* — a list of collections required *not* to be installed before unpacking the archive.

- *plt-relative?* — a boolean; if true, then the archive's content should be unpacked relative to the plt add-ons directory.
- *plt-home-relative?* — a boolean; if true and if 'plt-relative? is true, then the archive's content should be unpacked relative to the Racket installation.
- *test-plt-dirs* — #f or a 'paths where *paths* is a list of path strings; in the latter case, a true value of *plt-home-relative?* is cancelled if any of the directories in the list (relative to the Racket installation) is unwritable by the user.

The S-expression is extracted from the archive using read (and the result is *not* evaluated).

- An S-expression matching

```
(unit (import main-collects-parent-dir mzuntar)
      (export)
      (mzuntar void)
      (quote collections))
```

where *collections* is a list of collection paths (where each collection path is a list of strings); once the archive is unpacked, `raco setup` will compile and setup the specified collections.

The S-expression is extracted from the archive using read (and the result is *not* evaluated).

The archive continues with unpackables. Unpackables are extracted until the end-of-file is found (as indicated by an = in the base64-encoded input archive).

An *unpackable* is one of the following:

- The symbol 'dir followed by a list S-expression. The build-path procedure will be applied to the list to obtain a relative path for the directory (and the relative path is combined with the target directory path to get a complete path).

  The 'dir symbol and list are extracted from the archive using read (and the result is *not* evaluated).

- The symbol 'file, a list, a number, an asterisk, and the file data. The list specifies the file's relative path, just as for directories. The number indicates the size of the file to be unpacked in bytes. The asterisk indicates the start of the file data; the next n bytes are written to the file, where n is the specified size of the file.

  The symbol, list, and number are all extracted from the archive using read (and the result is *not* evaluated). After the number is read, input characters are discarded until an asterisk is found. The file data must follow this asterisk immediately.

- The symbol 'file-replace is treated like 'file, but if the file exists on disk already, the file in the archive replaces the file on disk.

## 10.2 API for Packing

```
(require setup/pack)
```
package: base

Although the `raco pack` command can be used to create most `".plt"` files, the `setup/pack` library provides a more general API for making `".plt"` archives.

```
(pack-collections-plt
 dest
 name
 collections
 [#:replace? replace?
  #:at-plt-home? at-home?
  #:test-plt-collects? test?
  #:extra-setup-collections collection-list
  #:file-filter filter-proc])
→ void?
 dest : path-string?
 name : string?
 collections : (listof (listof path-string?))
 replace? : boolean? = #f
 at-home? : boolean? = #f
 test? : boolean? = #t
 collection-list : (listof path-string?) = null
 filter-proc : (path-string? . -> . boolean?) = std-filter
```

Creates the `".plt"` file specified by the pathname *dest*, using the *name* as the name reported to `raco setup` as the archive's description.

The archive contains the collections listed in *collections*, which should be a list of collection paths; each collection path is, in turn, a list of relative-path strings.

If the `#:replace?` argument is `#f`, then attempting to unpack the archive will report an error when any of the collections exist already, otherwise unpacking the archive will overwrite an existing collection.

If the `#:at-plt-home?` argument is `#t`, then the archived collections will be installed into the Racket installation directory instead of the user's directory if the main `"collects"` directory is writable by the user. If the `#:test-plt-collects?` argument is `#f` (the default is `#t`) and the `#:at-plt-home?` argument is `#t`, then installation fails if the main `"collects"` directory is not writable.

The optional `#:extra-setup-collections` argument is a list of collection paths that are not included in the archive, but are set-up when the archive is unpacked.

The optional `#:file-filter` argument is the same as for `pack-plt`.

```
(pack-collections  dest
                   name
                   collections
                   replace?
                   extra-setup-collections
                  [filter
                   at-plt-home?])          → void?
  dest : path-string?
  name : string?
  collections : (listof (listof path-string?))
  replace? : boolean?
  extra-setup-collections : (listof path-string?)
  filter : (path-string? . -> . boolean?) = std-filter
  at-plt-home? : boolean? = #f
```

Old, keywordless variant of `pack-collections-plt` for backward compatibility.

```
(pack-plt  dest
           name
           paths
          [#:as-paths as-paths
           #:file-filter filter-proc
           #:encode? encode?
           #:file-mode file-mode-sym
           #:unpack-unit unpack-spec
           #:collections collection-list
           #:plt-relative? plt-relative?
           #:at-plt-home? at-plt-home?
           #:test-plt-dirs dirs
           #:requires mod-and-version-list
           #:conflicts mod-list])          → void?
  dest : path-string?
  name : string?
  paths : (listof path-string?)
  as-paths : (listof path-string?) = paths
  filter-proc : (path-string? . -> . boolean?) = std-filter
  encode? : boolean? = #t
  file-mode-sym : symbol? = 'file
  unpack-spec : any/c = #f
  collection-list : (listof path-string?) = null
  plt-relative? : any/c = #f
  at-plt-home? : any/c = #f
  dirs : (or/c (listof path-string?) false/c) = #f
  mod-and-version-list : (listof (listof path-string?)    = null
                                 (listof exact-integer?))
```

*mod-list* : (listof (listof path-string?)) = null

Creates the ".plt" file specified by the pathname *dest*, using the string *name* as the name reported to raco setup as the archive's description. The *paths* argument must be a list of relative paths for directories and files; the contents of these files and directories will be packed into the archive. The optional *as-paths* list provides the path to be recorded in the archive for each element of *paths* (so that the unpacked paths can be different from the packed paths).

The #:file-filter procedure is called with the relative path of each candidate for packing. If it returns #f for some path, then that file or directory is omitted from the archive. If it returns 'file or 'file-replace for a file, the file is packed with that mode, rather than the default mode. The default is std-filter.

If the #:encode? argument is #f, then the output archive is in raw form, and still must be gzipped and mime-encoded (in that order). The default value is #t.

The #:file-mode argument must be 'file or 'file-replace, indicating the default mode for a file in the archive. The default is 'file.

The #:unpack-unit argument is usually #f. Otherwise, it must be an S-expression for the S-expression that describes unpacking; see §10.1 "Format of ".plt" Archives" more information about the unit. If the #:unpack-unit argument is #f, an appropriate S-expression is generated.

The #:collections argument is a list of collection paths to be compiled after the archive is unpacked. The default is the null.

If the #:plt-relative? argument is true (the default is #f), the archive's files and directories are to be unpacked relative to the user's add-ons directory or the Racket installation directories, depending on whether the #:at-plt-home? argument is true and whether directories specified by #:test-plt-dirs are writable by the user.

If the #:at-plt-home? argument is true (the default is #f), then #:plt-relative? must be true, and the archive is unpacked relative to the Racket installation directory. In that case, a relative path that starts with "collects" is mapped to the installation's main "collects" directory, and so on, for the following the initial directory names:

- "collects"
- "doc"
- "lib"
- "include"

If #:test-plt-dirs is a list, then #:at-plt-home? must be #t. In that case, when the archive is unpacked, if any of the relative directories in the #:test-plt-dirs list is

unwritable by the current user, then the archive is unpacked in the user's add-ons directory after all.

The `#:requires` argument should have the shape `(list (list coll-path version) ...)` where each *coll-path* is a non-empty list of relative-path strings, and each *version* is a (possibly empty) list of exact integers. The indicated collections must be installed at unpacking time, with version sequences that match as much of the version sequence specified in the corresponding *version*. A collection's version is indicated by the `version` field of its `"info.rkt"` file.

The `#:conflicts` argument should have the shape `(list coll-path ...)` where each *coll-path* is a non-empty list of relative-path strings. The indicated collections must *not* be installed at unpacking time.

```
(pack  dest
       name
       paths
       collections
      [filter
       encode?
       file-mode
       unpack-unit
       plt-relative?
       requires
       conflicts
       at-plt-home?]) → void?
  dest : path-string?
  name : string?
  paths : (listof path-string?)
  collections : (listof path-string?)
  filter : (path-string? . -> . boolean?) = std-filter
  encode? : boolean? = #t
  file-mode : symbol? = 'file
  unpack-unit : any/c = #f
  plt-relative? : boolean? = #t
  requires : (listof (listof path-string?)    = null
                     (listof exact-integer?))
  conflicts : (listof (listof path-string?)) = null
  at-plt-home? : boolean? = #f
```

Old, keywordless variant of `pack-plt` for backward compatibility.

```
(std-filter p) → boolean?
  p : path-string?
```

Returns `#t` unless *p*, after stripping its directory path and converting to a byte string, matches

one of the following regular expressions: `^[.]git`, `^[.]svn$`, `^CVS$`, `^[.]cvsignore`, `^compiled$`, `^doc`, `~$`, `^#.*#$`, `^[.]#`, or `[.]plt$`.

```
(mztar path
       [#:as-path as-path]
        output
        filter
        file-mode)          → void?
  path : path-string?
  as-path : path-string? = path
  output : output-port?
  filter : (path-string? . -> . boolean?)
  file-mode : (symbols 'file 'file-replace)
```

Called by `pack` to write one directory/file *path* to the output port *output* using the filter procedure *filter* (see `pack` for a description of *filter*). The *path* is recorded in the output as *as-path*, in case the unpacked path should be different from the original path. The *file-mode* argument specifies the default mode for packing a file, either `'file` or `'file-replace`.

# 11 `raco unpack`: Unpacking Library Collections

The `raco unpack` command unpacks a `".plt"` archive (see §10 "`raco pack`: Packing Library Collections") to the current directory without attempting to install any collections. Use `raco pkg` (see *Package Management in Racket*) to install a `".plt"` archive as a package, or use `raco setup -A` (see §6 "`raco setup`: Installation Management") to unpack and install collections from a `".plt"` archive.

Command-line flags:

- `-l` or `--list` — lists the content of the archive without unpacking it.
- `-c` or `--config` — shows the archive configuration before unpacking or listing the archive content.
- `-f` or `--force` — replace files that exist already; fails that the archive says should be replaced will be replaced without this flag.

## 11.1 Unpacking API

```
(require setup/unpack)
```
package: `base`

The `setup/unpack` library provides raw support for unpacking a `".plt"` file.

```
(unpack archive
        [main-collects-parent-dir
         print-status
         get-target-directory
         force?
         get-target-plt-directory]) → void?
  archive : path-string?
  main-collects-parent-dir : path-string? = (current-directory)
  print-status : (string? . -> . any)
               = (lambda (x) (printf "~a\n" x))
  get-target-directory : (-> path-string?)
                       = (lambda () (current-directory))
  force? : any/c = #f
  get-target-plt-directory : (path-string?
                              path-string?
                              (listof path-string?)
                              . -> . path-string?)
                           = (lambda (preferred-dir main-dir options)
                               preferred-dir)
```

Unpacks *archive*.

The *main-collects-parent-dir* argument is passed along to *get-target-plt-directory*.

The *print-status* argument is used to report unpacking progress.

The *get-target-directory* argument is used to get the destination directory for unpacking an archive whose content is relative to an arbitrary directory.

If *force?* is true, then version and required-collection mismatches (comparing information in the archive to the current installation) are ignored.

The *get-target-plt-directory* function is called to select a target for installation for an archive whose is relative to the installation. The function should normally return one if its first two arguments; the third argument merely contains the first two, but has only one element if the first two are the same. If the archive does not request installation for all uses, then the first two arguments will be different, and the former will be a user-specific location, while the second will refer to the main installation.

```
(fold-plt-archive archive
                  on-config-fn
                  on-setup-unit
                  on-directory
                  on-file
                  initial-value) → any/c
  archive : path-string?
  on-config-fn : (any/c any/c . -> . any/c)
  on-setup-unit : (any/c input-port? any/c . -> . any/c)
  on-directory : ((or/c path-string?
                        (list/c (or/c 'collects 'doc 'lib 'include)
                                path-string?))
                  any/c
                  . -> . any/c)
  on-file : (or/c ((or/c path-string?
                         (list/c (or/c 'collects 'doc 'lib 'include)
                                 path-string?))
                   input-port?
                   any/c
                   . -> . any/c)
                  ((or/c path-string?
                         (list/c (or/c 'collects 'doc 'lib 'include)
                                 path-string?))
                   input-port?
                   (one-of/c 'file 'file-replace)
                   any/c
                   . -> . any/c))
  initial-value : any/c
```

Traverses the content of *archive*, which must be a ".plt" archive that is created with the default unpacking unit and configuration expression. The configuration expression is not evaluated, the unpacking unit is not invoked, and not files are unpacked to the filesystem. Instead, the information in the archive is reported back through on-config, *on-setup-unit*, *on-directory*, and *on-file*, each of which can build on an accumulated value that starts with *initial-value* and whose final value is returned.

The *on-config-fn* function is called once with an S-expression that represents a function to implement configuration information. The second argument to on-config is *initial-value*, and the function's result is passes on as the last argument to *on-setup-unit*.

The *on-setup-unit* function is called with the S-expression representation of the installation unit, an input port that points to the rest of the file, and the accumulated value. This input port is the same port that will be used in the rest of processing, so if *on-setup-unit* consumes any data from the port, then that data will not be consumed by the remaining functions. (This means that on-setup-unit can leave processing in an inconsistent state, which is not checked by anything, and therefore could cause an error.) The result of *on-setup-unit* becomes the new accumulated value.

For each directory that would be created by the archive when unpacking normally, *on-directory* is called with the directory path (described more below) and the accumulated value up to that point, and its result is the new accumulated value.

For each file that would be created by the archive when unpacking normally, *on-file* is called with the file path (described more below), an input port containing the contents of the file, an optional mode symbol indicating whether the file should be replaced, and the accumulated value up to that point; its result is the new accumulated value. The input port can be used or ignored, and parsing of the rest of the file continues the same either way. After *on-file* returns control, however, the input port is drained of its content.

A directory or file path can be a plain path, or it can be a list containing 'collects, 'doc, 'lib, or 'include and a relative path. The latter case corresponds to a directory or file relative to a target installation's collection directory (in the sense of find-collects-dir), documentation directory (in the sense of find-doc-dir), library directory (in the sense of find-lib-dir), or "include" directory (in the sense of find-include-dir).

# 12 `raco ctool`: Working with C Code

The `raco ctool` command works in various modes (as determined by command-line flags) to support various tasks involving C code.

## 12.1 Compiling and Linking C Extensions

A *dynamic extension* is a shared library (a.k.a. DLL) that extends Racket using the C API. An extension can be loaded explicitly via `load-extension`, or it can be loaded implicitly through `require` or `load/use-compiled` in place of a source *file* when the extension is located at

```
(build-path "compiled" "native" (system-library-subpath)
            (path-add-suffix file (system-type 'so-suffix)))
```

relative to *file*.

For information on writing extensions, see *Inside: Racket C API*.

Three `raco ctool` modes help for building extensions:

- `--cc` : Runs the host system's C compiler, automatically supplying flags to locate the Racket header files and to compile for inclusion in a shared library.
- `--ld` : Runs the host system's C linker, automatically supplying flags to locate and link to the Racket libraries and to generate a shared library.
- `--xform` : Transforms C code that is written without explicit GC-cooperation hooks to cooperate with Racket's 3m garbage collector; see §1 "Overview" in *Inside: Racket C API*.

Compilation and linking build on the `dynext/compile` and `dynext/link` libraries. The following `raco ctool` flags correspond to setting or accessing parameters for those libraries: `--tool`, `--compiler`, `--ccf`, `--ccf`, `--ccf-clear`, `--ccf-show`, `--linker`, `++ldf`, `--ldf`, `--ldf-clear`, `--ldf-show`, `++ldl`, `--ldl-show`, `++cppf`, `++cppf ++cppf-clear`, and `--cppf-show`.

The `--3m` flag specifies that the extension is to be loaded into the 3m variant of Racket. The `--cgc` flag specifies that the extension is to be used with the CGC. The default depends on `raco`: `--3m` if `raco` itself is running in 3m, `--cgc` if `raco` itself is running in CGC.

### 12.1.1 API for 3m Transformation

`(require compiler/xform)` package: `cext-lib`

```
(xform  quiet?
        input-file
        output-file
        include-dirs
       [#:keep-lines? keep-lines?]) → any/c
  quiet? : any/c
  input-file : path-string?
  output-file : path-string?
  include-dirs : (listof path-string?)
  keep-lines? : boolean? = #f
```

Transforms C code that is written without explicit GC-cooperation hooks to cooperate with Racket's 3m garbage collector; see §1 "Overview" in *Inside: Racket C API*.

The arguments are as for `compile-extension`; in addition *keep-lines?* can be `#t` to generate GCC-style annotations to connect the generated C code with the original source locations.

The file generated by `xform` can be compiled via `compile-extension`.

## 12.2 Embedding Modules via C

The `--c-mods` mode for `raco ctool` takes a set of Racket modules and generates a C source file that can be used as part of program that embeds the Racket run-time system. See §2 "Embedding into a Program" in *Inside: Racket C API* for an explanation of embedding programs.

The generated source file embeds the specified modules, and it defines a `declare_modules` function that puts the module declarations into a namespace. Thus, using the output of `raco ctool --c-mods`, a program can embed Racket with a set of modules so that it does not need a `"collects"` directory to load modules at run time.

If the embedded modules refer to runtime files, the files can be gathers by supplying the `--runtime` argument to `raco ctool --cmods`, specifying a directory ⟨*dir*⟩ to hold the files. Normally, ⟨*dir*⟩ is a relative path, and files are found at run time in ⟨*dir*⟩ relative to the executable, but a separate path (usually relative) for run time can be specified with `--runtime-access`.

## 12.3 API for Raw Compilation

```
(require compiler/compiler)
```
package: `base`

The compiler/compiler library provides the functionality of raco make for compilation to bytecode, but through a Racket API.

### 12.3.1 Bytecode Compilation

```
((compile-zos  expr
               [#:module? module?
                #:verbose? verbose?])
               racket-files
               dest-dir)              → void?
  expr : any/c
  module? : any/c = #f
  verbose? : any/c = #f
  racket-files : (listof path-string?)
  dest-dir : (or/c path-string? false/c (one-of/c 'auto))
```

Supplying just *expr* returns a compiler that is initialized with the expression *expr*, as described below.

The compiler takes a list of Racket files and compiles each of them to bytecode, placing the resulting bytecode in a ".zo" file within the directory specified by *dest-dir*. If *dest-dir* is #f, each bytecode result is placed in the same directory as its source file. If *dest-dir* is 'auto, each bytecode file is placed in a "compiled" subdirectory relative to the source; the directory is created if necessary.

If *expr* is anything other than #f, then a namespace is created for compiling the files that are supplied later, and *expr* is evaluated to initialize the created namespace. For example, *expr* might load a set of macros. In addition, the expansion-time part of each expression later compiled is evaluated in the namespace before being compiled, so that the effects are visible when compiling later expressions.

If *expr* is #f, then no compilation namespace is created (the current namespace is used), and expressions in the files are assumed to compile independently (so there's no need to evaluate the expansion-time part of an expression to compile).

Typically, *expr* is #f for compiling module files, and it is (void) for compiling files with top-level definitions and expressions.

If *module?* is #t, then the given files are read and compiled as modules (so there is no dependency on the current namespace's top-level environment).

If *verbose?* is #t, the output file for each given file is reported through the current output port.

```
(compile-collection-zos
 collection ...+
 [#:skip-path skip-path
  #:skip-paths skip-paths
  #:skip-doc-sources? skip-docs?
  #:managed-compile-zo managed-compile-zo])
→ void?
 collection : string?
 skip-path : (or/c path-string? #f) = #f
 skip-paths : (listof path-string?) = null
 skip-docs? : any/c = #f
 managed-compile-zo : (path-string? . -> . void?)
                    = (make-caching-managed-compile-zo)
```

Compiles the specified collection's files to `".zo"` files by using *managed-compile-zo* on each source file. The `".zo"` files are placed into the collection's `"compiled"` directory.

By default, all files with the extension `".rkt"`, `".ss"`, or `".scm"` in a collection are compiled, as are all such files within subdirectories; the set of such suffixes is extensible globally as described in `get-module-suffixes`, and `compile-collection-zos` recognizes suffixes from the `'libs` group. However, any file or directory whose path starts with *skip-path* or an element of *skip-paths* is skipped. ("Starts with" means that the simplified path *p*'s byte-string form after `(simplify-path p #f)`starts with the byte-string form of `(simplify-path skip-path #f)`.)

The collection compiler reads the collection's `"info.rkt"` file (see §6.4 "`"info.rkt"` File Format") to obtain further instructions for compiling the collection. The following fields are used:

- `name` : The name of the collection as a string, used only for status and error reporting.
- `compile-omit-paths` : A list of immediate file and directory paths that should not be compiled. Alternatively, this field's value `'all`, which is equivalent to specifying all files and directories in the collection (to effectively ignore the collection for compilation). Automatically omitted files and directories are `"compiled"`, `"doc"`, and those whose names start with `.`.

  Files that are required by other files, however, are always compiled in the process of compiling the requiring file—even when the required file is listed with this field or when the field's value is `'all`.
- `compile-omit-files` : A list of filenames (without directory paths) that are not compiled, in addition to the contents of `compile-omit-paths`. Do not use this field; it is for backward compatibility.
- `scribblings` : A list of pairs, each of which starts with a path for documentation source. The sources (and the files that they require) are compiled in the same way as other module files, unless *skip-docs?* is a true value.

- `compile-include-files` : A list of filenames (without directory paths) to be compiled, in addition to files that are compiled based on the file's extension, being in `scribblings`, or being `required` by other compiled files.

- `module-suffixes` and `doc-module-suffixes` — Used indirectly via `get-module-suffixes`.

Changed in version 6.3 of package `base`: Added support for `compile-include-files`.

```
(compile-directory-zos
 path
 info
 [#:verbose verbose?
  #:skip-path skip-path
  #:skip-paths skip-paths
  #:skip-doc-sources? skip-docs?
  #:managed-compile-zo managed-compile-zo])
→ void?
 path : path-string?
 info : ()
 verbose? : any/c = #f
 skip-path : (or/c path-string? #f) = #f
 skip-paths : (listof path-string?) = null
 skip-docs? : any/c = #f
 managed-compile-zo : (path-string? . -> . void?)
                    = (make-caching-managed-compile-zo)
```

Like `compile-collection-zos`, but compiles the given directory rather than a collection. The *info* function behaves like the result of `get-info` to supply `"info.rkt"` fields, instead of using an `"info.rkt"` file (if any) in the directory.

### 12.3.2 Recognizing Module Suffixes

`(require compiler/module-suffix)` package: `base`

The `compiler/module-suffix` library provides functions for recognizing file suffixes that correspond to Racket modules for the purposes of compiling files in a directory, running tests for files in a directory, and so on. The set of suffixes always includes `".rkt"`, `".ss"`, and `".scm"`, but it can be extended globally by `"info.rkt"` configuration in collections.

Added in version 6.3 of package `base`.

```
(get-module-suffixes [#:group group
                      #:mode mode
                      #:namespace namespace]) → (listof bytes?)
```

```
group : (or/c 'all 'libs 'docs) = 'all
mode : (or/c 'preferred 'all-available 'no-planet 'no-user)
     = 'preferred
namespace : (or/c #f namespace?) = #f
```

Inspects `"info.rkt"` files (see §6.4 “`"info.rkt"` File Format”) of installed collections to produce a list of file suffixes that should be recognized as Racket modules. Each suffix is reported as a byte string that does not include the `.` that precedes a suffix.

The *mode* and *namespace* arguments are propagated to `find-relevant-directories` to determine which collection directories might configure the set of suffixes. Consequently, suffix registrations are found reliably only if `raco setup` (or package installations or updates that trigger `raco setup`) is run.

The *group* argument determines whether the result includes all registered suffixes, only those that are registered as general library suffixes, or only those that are registered as documentation suffixes. The set of general-library suffixes always includes `".rkt"`, `".ss"`, and `".scm"`. The set of documentation suffixes always includes `".scrbl"`.

The following fields in an `"info.rkt"` file extend the set of suffixes:

- `module-suffixes` : A list of byte strings that correspond to general-library module suffixes (without the `.` that must appear before the suffix). Non-lists or non-byte-string elements of the list are ignored.
- `doc-module-suffixes` : A list of byte strings as for `module-suffixes`, but for documentation modules.

```
(get-module-suffix-regexp [#:group group
                           #:mode mode
                           #:namespace namespace]) → byte-regexp?
  group : (or/c 'all 'libs 'docs) = 'all
  mode : (or/c 'preferred 'all-available 'no-planet 'no-user)
       = 'preferred
  namespace : (or/c #f namespace?) = #f
```

Returns a regexp value that matches paths ending with a suffix as reported by `get-module-suffixes`. The pattern includes a subpatterns for the suffix without its leading `.`

### 12.3.3 Loading Compiler Support

The compiler unit loads certain tools on demand via `dynamic-require` and `get-info`. If the namespace used during compilation is different from the namespace used to load the

compiler, or if other load-related parameters are set, then the following parameter can be used to restore settings for `dynamic-require`.

```
(current-compiler-dynamic-require-wrapper)
 → ((-> any) . -> . any)
(current-compiler-dynamic-require-wrapper proc) → void?
  proc : ((-> any) . -> . any)
```

A parameter whose value is a procedure that takes a thunk to apply. The default wrapper sets the current namespace (via `parameterize`) before calling the thunk, using the namespace in which the `compiler/compiler` library was originally instantiated.

### 12.3.4 Options for the Compiler

`(require compiler/option)` package: `base`

The `compiler/option` module provides options (in the form of parameters) that control the compiler's behaviors.

More options are defined by the `dynext/compile` and `dynext/link` libraries, which control the actual C compiler and linker that are used for compilation via C.

```
(somewhat-verbose) → boolean?
(somewhat-verbose on?) → void?
  on? : any/c
```

A `#t` value for the parameter causes the compiler to print the files that it compiles and produces. The default is `#f`.

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

A `#t` value for the parameter causes the compiler to print verbose messages about its operations. The default is `#f`.

```
(compile-subcollections) → (one-of/c #t #f)
(compile-subcollections cols) → void?
  cols : (one-of/c #t #f)
```

A parameter that specifies whether sub-collections are compiled by `compile-collection-zos`. The default is `#t`.

### 12.3.5 The Compiler as a Unit

#### Signatures

```
(require compiler/sig)
```
package: compiler-lib

```
compiler^ : signature
```

Includes all of the names exported by `compiler/compiler`.

```
compiler:option^ : signature
```

Includes all of the names exported by `compiler/option`.

#### Main Compiler Unit

```
(require compiler/compiler-unit)
```
package: compiler-lib

```
compiler@ : unit?
```

Provides the exports of `compiler/compiler` in unit form, where C-compiler operations are imports to the unit, although they are not used.

The unit imports `compiler:option^`, `dynext:compile^`, `dynext:link^`, and `dynext:file^`. It exports `compiler^`.

#### Options Unit

```
(require compiler/option-unit)
```
package: compiler-lib

```
compiler:option@ : unit?
```

Provides the exports of `compiler/option` in unit form. It imports no signatures, and exports `compiler:option^`.

# 13 `raco test`: Run tests

The `raco test` command requires and runs the (by default) `test` submodule associated with each path given on the command line. Command-line flags can control which submodule is run, whether to run the main module if no submodule is found, and whether to run tests directly, in separate processes (the default), or in separate places. The current directory is set to a test file's directory before running the file.

When an argument path refers to a directory, `raco test` recursively discovers and runs all files within the directory that end in a module suffix (see `get-module-suffixes`, but the suffixes always include `".rkt"`, `".scrbl"`, `".ss"`, and `".scm"`) or have a (possibly empty) list of command-line arguments provided by `test-command-line-arguments` in an `"info.rkt"` file, or as directed by `test-include-paths` in an `"info.rkt"` file. At the same time, `raco test` omits files and directories within a directory as directed by `test-omit-paths` in an `"info.rkt"` file.

A test is counted as failing if it logs a failing test code via `test-log!`, causes Racket to exit with a non-zero exit code, or (when `-e` or `--check-stderr` is specified) if it produces output on the error port.

The `raco test` command accepts several flags:

- `-c` or `--collection` — Interprets the arguments as collections whose content should be tested (in the same way as directory content).
- `-p` or `--package` — Interprets the arguments as packages whose contents should be tested (in the same way as directory content). All package scopes are searched for the first, most specific package scope.
- `-l` or `--lib` — Interprets the arguments as libraries that should be tested.
- `-m` or `--modules` — Not only interprets the arguments as paths (which is the default mode), but treats them the same as paths found in a directory, which means ignoring a file argument that does not have a module extension or is not enabled explicitly via `test-command-line-arguments` or `test-include-paths` in an `"info.rkt"` file; meanwhile, paths that are otherwise enabled can be disabled via `test-omit-paths` in an `"info.rkt"` file.
- `--drdr` — Configures defaults to imitate the DrDr continuous testing system: ignore non-modules, run tests in separate processes, use as many jobs as available processors, set the default timeout to 90 seconds, create a fresh `PLTUSERHOME` and `TMPDIR` for each test, count stderr output as a test failure, quiet program output, provide empty program input, and print a table of results.
- `-s` ⟨*name*⟩ or `--submodule` ⟨*name*⟩ — Requires the submodule ⟨*name*⟩ rather than `test`. Supply `-s` or `--submodule` to run multiple submodules, or combine multiple submodules with `--first-avail` to run the first available of the listed modules.

- `-r` or `--run-if-absent` — Requires the top-level module of a file if a relevant submodule is not present. This is the default mode.

- `-x` or `--no-run-if-absent` — Ignores a file if the relevant submodule is not present.

- `--first-avail` — When multiple submodule names are provided with `-s` or `--submodule`, runs only the first available submodule.

- `--configure-runtime` — Run a `configure-runtime` submodule (if any) of each specified module before the module or a submodule is run. This mode is the default when only a single module is provided or when `--process` or `--place` mode is specified, unless a submodule name is provided via `-s` or `--submodule`.

- `--direct` — Runs each test in a thread. This mode is the default if a single file is specified. Multiple tests can interfere with each other and the overall test run by exiting, unsafe operations that block (and thus prevent timeout), and so on.

- `--process` — Runs each test in a separate operating-system process. This mode is the default if multiple files are specified or if a directory, collection, or package is specified.

- `--place` — Runs each test in a place, instead of in an operating-system process.

- `-j` ⟨*n*⟩ or `--jobs` ⟨*n*⟩ — Runs up to ⟨*n*⟩ tests in parallel.

- `--timeout` ⟨*seconds*⟩ — Sets the default timeout (after which a test counts as failed) to ⟨*seconds*⟩. Use `+inf.0` to allow tests to run without limit but allow `timeout` sub-submodule configuration. If any test fails due to a timeout, the exit status of `raco test` is 2 (as opposed to 1 for only non-timeout failures or 0 for success).

- `--fresh-user` — When running tests in a separate process, creates a fresh directory and sets `PLTUSERHOME` and `TMPDIR`. The `PLTADDONDIR` environment variable is also set so that the add-on directory (which is where packages are installed, for example) does *not* change for each test process.

- `--empty-stdin` — Provide an empty stdin to each test program.

- `-Q` or `--quiet-program` — Suppresses output from each test program.

- `-e` or `--check-stderr` — Count any stderr output as a test failure.

- `++ignore-stderr` ⟨*pattern*⟩ — Don't count stderr output as a test failure if it matches ⟨*pattern*⟩. This flag can be used multiple times, and stderr output is treated as success as long as it matches any one ⟨*pattern*⟩.

- `-q` or `--quiet` — Suppresses output of progress information, responsible parties, and varying output (see §13.3 "Responsible-Party and Varying-Output Logging").

- `--heartbeat` — Periodically report that a test is still running after the test has been running at least 5 seconds.

- `--table` or `-t` — Print a summary table after all tests. If a test uses `rackunit`, or if a test at least uses `test-log!` from `rackunit/log` to log successes and failures, the table reports test and failure counts based on the log.

Changed in version 1.1: Added `--heartbeat`. Changed in version 1.4: Changed recognition of module suffixes to use `get-module-suffixes`, which implies recognizing `".ss"` and `".rkt"`.

## 13.1 Test Configuration by Submodule

When `raco test` runs a test in a submodule, a `config` sub-submodule can provide additional configuration for running the test. The `config` sub-submodule should use the `info` module language to define the following identifiers:

- `timeout` — a real number to override the default timeout for the test, which applies only when timeouts are enabled.
- `responsible` — a string, symbol, or list of symbols and strings identifying a responsible party that should be notified when the test fails. See §13.3 "Responsible-Party and Varying-Output Logging".
- `lock-name` — a string that names a lock file that is used to serialize tests (i.e., tests that have the same lock name do not run concurrently). The lock file's location is determined by the `PLTLOCKDIR` enviornment varible or defaults to `(find-system-path 'temp-dir)`. The maximum time to wait on the lock file is determined by the `PLTLOCKTIME` environment variable or defaults to 4 hours.
- `random?` — if true, indicates that the test's output is expected to vary. See §13.3 "Responsible-Party and Varying-Output Logging".

In order to prevent evaluation of a file for testing purposes, it suffices to create a submodule that does not perform any tests and does not trigger the evaluation of the enclosing module. So, for instance, a file might look like this:

```
#lang racket

(/ 1 0)

; don't run this file for testing:
(module test racket/base)
```

## 13.2 Test Configuration by `"info.rkt"`

Submodule-based test configuration is preferred (see §13.1 "Test Configuration by Submodule"). In particular, to prevent `raco test` from running a particular file, normally the file

should contain a submodule that takes no action.

In some cases, however, adding a submodule is inconvenient or impossible (e.g., because the file will not always compile). Thus, `raco test` also consults any `"info.rkt"` file in the candidate test file's directory. In the case of a file within a collection, `"info.rkt"` files from any enclosing collection directories are also consulted for `test-omit-paths` and `test-include-paths`. Finally, for a file within a package, the package's `"info.rkt"` is consulted for `pkg-authors` to set the default responsible parties (see §13.3 "Responsible-Party and Varying-Output Logging") for all files in the package.

The following `"info.rkt"` fields are recognized:

- `test-omit-paths` — a list of path strings (relative to the enclosing directory) and regexp values (to omit all files within the enclosing directory matching the expression), or `'all` to omit all files within the enclosing directory. When a path string refers to a directory, all files within the directory are omitted.
- `test-include-paths` — a list of path strings (relative to the enclosing directory) and regexp values (to include all files within the enclosing directory matching the expression), or `'all` to include all files within the enclosing directory. When a path string refers to a directory, all files within the directory are included.
- `test-command-line-arguments` — a list of `(list` *`module-path-string`* `(list` *`argument-path-string`* `...))`, where `current-command-line-arguments` is set to a vector that contains the *`argument-path-string`* when running *`module-path-string`*.
- `test-timeouts` — a list of `(list` *`module-path-string real-number`*`)` to override the default timeout for *`module-path-string`*.
- `test-responsibles` — a list of `(list` *`module-path-string party`*`)` or `(list 'all` *`party`*`)` to override the default responsible party for *`module-path-string`* or all files within the directory (except as overridden), respectively. Each *`party`* is a string, symbol, or list of symbols and strings. See §13.3 "Responsible-Party and Varying-Output Logging".
- `test-lock-names` — a list of `(list` *`module-path-string lock-string`*`)` to declare a lock file name for *`module-path-string`*. See `lock-name` in §13.1 "Test Configuration by Submodule".
- `test-randoms` — a list of path strings (relative to the enclosing directory) for modules whose output varies. See §13.3 "Responsible-Party and Varying-Output Logging".
- `module-suffixes` and `doc-module-suffixes` — Used indirectly via `get-module-suffixes`.

## 13.3 Responsible-Party and Varying-Output Logging

When a test has a declared responsible party, then the test's output is prefixed with a

```
raco test:⟨which⟩ @(test-responsible '⟨responsible⟩)
```

line, where ⟨*which*⟩ is a space followed by an exact non-negative number indicating a parallel task when parallelism is enabled (or empty otherwise), and ⟨*responsible*⟩ is a string, symbol, or list datum.

When a test's output (as written to stdout) is expected to vary across runs—aside from varying output that has the same form as produced by `time`—then it should be declared as varying. In that case, the test's output is prefixed with a

```
raco test:⟨which⟩ @(test-random #t)
```

line.

# 14 `raco docs`: Documentation Search

The `raco docs` command searches the documentation for the given identifiers or search terms.

Command-line flags:

- `-h` or `--help` — show help information for this command
- `--` — do not treat remaining arguments as switches

# 15 `raco expand`: Macro Expansion

The `raco expand` command macro-expands and pretty-prints the contents of the given source files. See also `expand`.

Command-line flags:

- `-n` ⟨*n*⟩ or `--columns` ⟨*n*⟩ — format output for a display with ⟨*n*⟩ columns
- `-h` or `--help` — show help information for this command
- `--` — do not treat remaining arguments as switches

# 16 `raco read`: Reading and Pretty-Printing

The `raco read` command `read`s and pretty-prints the contents of the given files. This command is useful for showing how a `#reader` or `#lang`-based reader extension converts input to an S-expression. It is also useful for pretty-printing a term that is already in S-expression form.

Command-line flags:

- `-n` ⟨*n*⟩ or `--columns` ⟨*n*⟩ — format output for a display with ⟨*n*⟩ columns
- `-h` or `--help` — show help information for this command
- `--` — do not treat remaining arguments as switches

Added in version 1.3 of package `compiler-lib`.

# 17 `raco scribble`: Building Documentation

See *Scribble: The Racket Documentation Tool* for information on the `raco scribble` command, which is used to run and render a Scribble document.

# 18 Adding a `raco` Command

The set of commands supported by `raco` can be extended by installed packages, PLaneT packages, and other collections. A command is added by defining `raco-commands` in the `"info.rkt"` library of a collection (see §6.4 “`"info.rkt"` File Format”), and then `raco setup` (as called directly or as part of a package or PLaneT installation) must index the `"info.rkt"` file.

The value bound to `raco-commands` must be a list of command specifications, where each specification is a list of four values:

```
(list command-string
      implementation-module-path
      description-string
      prominence)
```

The *command-string* is the command name. Any unambiguous prefix of a command name can be supplied to `raco` to invoke the command.

The *implementation-module-path* names the implementation though a module path (in the sense of `module-path?`). The module is loaded and invoked through `dynamic-require` to run the command. The module can access command-line arguments through the `current-command-line-arguments` parameter, which is adjusted before loading the command module to include only the arguments to the command. The `current-command-name` parameter is also set to the command name used to load the command. When `raco help` is used on a command, the command is launched with an initial `--help` argument in `current-command-line-arguments`.

The *description-string* is a short string used to describe the command in response to `raco help`. The description should not be capitalized or end with a period.

The *prominence* value should be a read number or `#f`. A `#f` value means that the command should not be included in the short list of “frequently used commands.” A number indicates the relative prominence of the command; the `help` command has a value of `110`, and probably no command should be more prominent. The `pack` tool, which is currently ranked as the least-prominent of the frequently used commands, has a value of `10`.

As an example, the `"info.rkt"` of the `"compiler"` collection might contain the

```
(define raco-commands
  '(("make" compiler/commands/make "compile source to byte-
code" 100)
    ("decompile" compiler/commands/decompile "decompile byte-
code" #f)))
```

so that `make` is treated as a frequently used command, while `decompile` is available as an

infrequently used command.

## 18.1 Command Argument Parsing

```
(require raco/command-name)
```
package: base

The raco/command-name library provides functions to help a raco command identify itself to users.

```
(current-command-name) → (or/c string? #f)
(current-command-name name) → void?
  name : (or/c string? #f)
```

The name of the command currently being loaded via dynamic-require, or #f if raco is not loading any command.

A command implementation can use this parameter to determine whether it was invoked via raco or through some other means.

```
(short-program+command-name) → string?
```

Returns a string that identifies the current command. When current-command-name is a string, then the result is the short name of the raco executable followed by a space and the command name. Otherwise, it is the short name of the current executable, as determined by stripping the path from the result of (find-system-path 'run-file). In either case, on Windows, an ".exe" extension is removed from the executable name.

The result of this function is suitable for use with command-line. For example, the decompile tool parses command-line arguments with

```
(define source-files
  (command-line
   #:program (short-program+command-name)
   #:args source-or-bytecode-file
   source-or-bytecode-file))
```

so that raco decompile --help prints

```
raco decompile [ <option> ... ] [<source-or-bytecode-file>] ...
where <option> is one of
 --help, -h : Show this help
 -- : Do not treat any remaining argument as a switch (at this
level)
Multiple single-letter switches can be combined after one `-'; for
 example: `-h-' is the same as `-h --'
```

(program+command-name) → string?

Like short-program+command-name, but the path (if any) is not stripped from the current executable's name.

# 19 Installation Configuration and Search Paths

A *configuration directory* path is built into the Racket executable as selected at install time, and its location can be changed via the `PLTCONFIGDIR` directory or `--config`/`-G` command-line flag. Use `find-config-dir` to locate the configuration directory.

Modify the `"config.rktd"` file in the configuration directory to configure other directories as described below. Use the `setup/dirs` library (which combines information from the configuration files and other sources) to locate configured directories, instead of reading `"config.rktd"` directly.

The path of the *main collection directory* is built into the Racket executable, and it can be changed via the `--collects`/`-X` flag, so it has no entry in `"config.rktd"`. Most paths that are specified in `"config.rktd"` have default values that are relative to the main collection directory. The paths of the configuration directory and main collection directory thus work together to determine a Racket configuration.

A `"config.rktd"` file in the configuration directory should contain a `read`able hash table with any of the following symbolic keys, where a relative path is relative to the main collection directory:

- `'installation-name` — a string for the installation name, which is used to determine user- and version-specific paths, such as the initial path produced by `find-library-collection-paths` and the location of packages that are installed in `user` package scope. The default is `(version)`.
- `'lib-dir` — a path, string, or byte string for the *main library directory*. It defaults to a `"lib"` sibling directory of the main collection directory.
- `'lib-search-dirs` — a list of paths, strings, byte strings, or `#f` representing the search path for directories containing foreign libraries. Each `#f` in the list, if any, is replaced with the default search path, which is the user- and version-specific `"lib"` directory followed by the main library directory.
- `'dll-dir` — a path, string, or byte string for a directory containing shared libraries for the main executable. It defaults to the main library directory.
- `'share-dir` — a path, string, or byte string for the *main shared-file directory*, which normally includes installed packages. It defaults to a `"share"` sibling directory of the main collection directory.
- `'links-file` — a path, string, or byte string for the collection links file. It defaults to a `"links.rktd"` file in the main shared-file directory.
- `'links-search-files` — like `'lib-search-dirs`, but for collection links file.
- `'pkgs-dir` — a path, string, or byte string for packages that have `installation` package scope. It defaults to `"pkgs"` in the main shared-file directory.

- 'pkgs-search-dirs — like 'lib-search-dirs, but for packages in installation package scope.

- 'bin-dir — a path, string, or byte string for the installation's directory containing executables. It defaults to a "bin" sibling directory of the main collection directory.

- 'apps-dir — a path, string, or byte string for the installation's directory for ".desktop" files. It defaults to a "applications" subdirectory of the main shared-file directory.

- 'man-dir — a path, string, or byte string for the installation's man-page directory. It defaults to a "man" sibling directory of the main collection directory.

- 'doc-dir — a path, string, or byte string for the main documentation directory. The value defaults to a "doc" sibling directory of the main collection directory.

- 'doc-search-dirs — like 'lib-search-dirs, but for directories containing documentation.

- 'doc-search-url — a URL string that is augmented with version and search-tag queries to form a remote documentation reference.

- 'doc-open-url — a URL string or #f; a string supplies a URL that is used instead of a local path to search and maybe open documentation pages (which normally makes sense only in an environment where opening a local HTML file does not work).

- 'include-dir — a path, string, or byte string for the main directory containing C header files. It defaults to an "include" sibling directory of the main collection directory.

- 'include-search-dirs — like doc-search-dirs, but for directories containing C header files.

- 'catalogs — a list of URL strings used as the search path for resolving package names. An #f in the list is replaced with the default search path. A string that does not start with alphabetic characters followed by :// is treated as a path, where a relative path is relative to the configuration directory.

- 'default-scope — either "user" or "installation", determining the default package scope for package-management operations.

- 'download-cache-dir — a path string used as the location for storing downloaded package archives. When not specified, packages are cached in a "download-cache" directory in the user's add-on directory as reported by (find-system-path 'addon-dir).

- 'download-cache-max-files and 'download-cache-max-bytes — real numbers that determine limits on the download cache. When not specified, the cache is allowed to hold up to 1024 files that total up to 64 MB.

- `'build-stamp` — a string that identifies a build, which can be used to augment the Racket version number to more specifically identify the build. An empty string is normally appropriate for a release build.

- `'absolute-installation?` — a boolean that is `#t` if the installation uses absolute path names, `#f` otherwise.

- `'cgc-suffix` — a string used as the suffix (before the actual suffix, such as `".exe"`) for a `"CGC"` executable. Use Windows-style casing, and the string will be downcased as appropriate (e.g., for a Unix binary name). A `#f` value means that if the `racket` binary identifies itself as CGC, then the suffix is `""`, otherwise it is `"CGC"`.

- `'3m-suffix` — analogous to `'cgc-suffix`, but for 3m. A `#f` value means that if the `"racket"` binary identifies itself as CGC, then the suffix is `"3m"`, otherwise it is `""`.