`raco`: Racket Command-Line Tools

Version 5.0

June 6, 2010

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.

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,

```
racket 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)
```

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

```
(make-compilation-manager-load/use-compiled-handler)
 → (path? (or/c symbol? false/c) . -> . any)
```

Returns a procedure suitable as a value for the current-load/use-compiled parameter. The returned procedure passes it 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.

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.

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

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.

```
(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)
 → (path-string? . -> . void?)
  read-src-syntax : (any/c input-port? . -> . syntax?)
```

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.

```
(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.

1.4 Compilation Manager Hook for Syntax Transformers

```
(require compiler/cm-accomplice)
```

(register-external-file *file*) → void?
  file : (and path? complete-path?)

Logs a message (see log-message) at level 'info. The message data is a file-dependency prefab structure type with one field whose value is *file*.

A compilation manager implemented by compiler/cm looks for such messages to register an external dependency. 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.

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

1.5 Compiling to Raw Bytecode

The -zo/-z 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.

By default, the generated bytecode is placed in the same directory as the source file—which is not where it will be found automatically when loading the source. Use the -auto-dir flag to redirect the output to a "compiled" subdirectory, where it will be found automatically when loading the source file.

Outside of a module, top-level define-syntaxes, module, #%require, define-values-for-syntax, and and begin expressions are handled specially by raco make -zo: the compile-time portion of the expression is evaluated, because it might affect later expressions. (The -m or -module flag turns off this special handling.)

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 `-zo` mode, `raco make -zo` 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.

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

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 packaged the executable into a distribution that works on other machines.

The `raco make` command embeds a module, from source or byte code, into a copy of the `racket` executable. (Under 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").

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".

2.1 API for Creating Executables

```
(require compiler/embed)
```

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 indpendent 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 loaded directly by a running program, as long as the read-accept-compiled parameter is true.

```
(create-embedding-executable
 dest
 [#:modules mod-list
  #: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
  #:launcher? launcher?
  #:verbose? verbose?
  #:compiler compile-proc
  #:expand-namespace expand-namespace
  #:src-filter src-filter
  #:on-extension ext-proc
  #:get-extra-imports extras-proc])
→ void?
 dest : path-string?
 mod-list : (listof (list/c (or/c symbol? (one-of/c #t #f))
                            module-path?))
          = 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 : (one-of/c 'cgc '3m) = (system-type 'gc)
aux : (listof (cons/c symbol? any/c)) = null
collects-path : (or/c false/c                = #f
                      path-string?
                      (listof path-string?))
launcher? : any/c = #f
verbose? : any/c = #f
compile-proc : (any/c . -> . compiled-expression?)
             = (lambda (e)
                 (parameterize ([current-namespace
                                 expand-namespace])
                   (compile e)))
expand-namespace : namespace? = (current-namespace)
src-filter : (path? . -> . any) = (lambda (p) #t)
ext-proc : (or/c false/c (path-string? boolean? . -> . any))
         = #f
extras-proc : (path? compiled-module?
                . -> . (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. Under 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 `#:literal-files` and `#:literal-expressions` arguments specify literal code to be copied into the executable: the content of each file in *literal-files* is copied

in order (with no intervening space), followed by each element of *literal-sexps*. The *literal-files* files or *literal-sexps* list 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. If *literal-sexp* is `#f`, no literal expression is included in the executable. 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. See also §17.1.5 "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-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). The prefix can be a symbol, `#f` to indicate no prefix, or `#t` to indicate an auto-generated prefix. For example,

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

embeds the module `m` from the file `"m.ss"`, without prefixing the name of the module; the `literal-sexpr` argument to go with the above might be `'(require m)`.

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).

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; the executable will have 16x16, 32x32, and 48x48 icons at 4-bit, 8-bit, and 32-bit (RBBA) depths; the icons are copied and generated from any 16x16, 32x32, and 48x48 icons in the `".ico"` file.
- `'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.

- `'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 seperately); a `#f` value (the default) means that absolute paths should be used (so the generated executable can be moved).

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 non-empty 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 `#:launcher?` argument is `#t`, then no `modules` 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` 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.

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` 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.

```
(make-embedding-executable dest
                           mred?
                           verbose?
                           mod-list
                           literal-files
                           literal-sexp
                           cmdline
                           [aux
                           launcher?
                           variant])      → void?
  dest : path-string?
  mred? : any/c
  verbose? : any/c
  mod-list : (listof (list/c (or/c symbol? (one-of/c #t #f))
                             module-path?))
  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)
```

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

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

---

compiler:embed^ : signature

Includes the identifiers provided by `compiler/embed`.

2.1.2 Executable Creation Unit

```
(require compiler/embed-unit)
```

`compiler:embed@ : unit?`

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

2.2 Installation-Specific Launchers

```
(require launcher/launcher)
```

The `launcher/launcher` library provides functions for creating *launchers*, which are similar to stand-alone executables, but sometimes smaller because they depend permanently on the local Racket installation. In the case of Unix, in particular, a launcher is simply a shell script. The `raco exe` command provides no direct support 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 under Windows and Mac OS X GRacket; the following additional associations apply to launchers:

- `'independent?` (Windows) — a boolean; `#t` creates an old-style launcher that is independent of the MzRacket 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.
- `'relative?` (all platforms) — a boolean, where `#t` means that the generated launcher should find the base GRacket executable through a relative path.

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 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. Under 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) → path?
  name : string?
```

Returns a pathname for an executable in the Racket installation called something like *name*. 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) → path?
  name : string?
```

Returns the same path as (gracket-program-launcher-path *name*) 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) → path?
  name : string?
(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) → path?
  name : string?
(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.

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", and checking whether such a file exists.

The recognized suffixes are as follows:

- ".icns" → 'icns file for use under Mac OS X
- ".ico" → 'ico file for use under Windows
- ".lch" → 'independent? as #t (the file content is ignored) for use under Windows
- ".creator" → 'creator as the initial four characters in the file for use under 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 under Mac OS X
- ".utiexports" → 'uti-exports as read content (a single S-expression); for use under Mac OS X

(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). Under Unix and Windows, the possibilities are 'cgc and '3m. Under 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 under 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)
```

`launcher^` : signature

Includes the identifiers provided by `launcher/launcher`.

2.2.5 Launcher Creation Unit

```
(require launcher/launcher-unit)
```

`launcher@` : `unit?`

A unit that imports nothing and exports `launcher^`.

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, under 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:

- Under Windows, executables are put directly into the distribution directory, and DLLs and other run-time files go into a `"lib"` sub-directory.
- Under 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.
- Under 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 `++copy-collects` flag of `raco distribute` to supply a directory whose content is copied into the distribution's `"collects"` directory. The `++copy-collects` 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 `++copy-collects` flag to include the copied libraries in the distribution.

3.1 API for Distributing Executables

```
(require compiler/distribute)
```

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

```
(assemble-distribution dest-dir
                       exec-files
                      [#:collects-path path
                       #:copy-collects dirs]) → void?
  dest-dir : path-string?
  exec-files : (listof path-string?)
  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, and/or shared libraries that the executables need to run a different machine.

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.

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.

3.2 API for Bundling Distributions

```
(require compiler/bundle-dist)
```

The `compiler/bundle-dist` library provides a function to pack a directory (usually assembled by `assemble-distribution`) into a distribution file. Under Windows, the result is a `".zip"` archive; under Mac OS X, it's a `".dmg"` disk image; under 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 under 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 pack`: Packaging Library Collections

Before creating a `".plt"` archive to distribute, consider instead posting your package on PLaneT.

The `raco pack` command creates an archive for distributing library files to Racket users. A distribution archive usually has the suffix `".plt"`, which DrRacket recognizes as an archive to provide automatic unpacking facilities. The `raco setup` command also supports `".plt"` unpacking.

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.ss"` 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 §6.2 "`"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 it's `"info.ss"` 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 setup/infotab
(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.

4.1 API for Packaging

```
(require setup/pack)
```

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
          [#:file-filter filter-proc
           #:encode? encode?
           #:file-mode file-mode-sym
           #:unpack-unit unit200-expr
           #: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?)
  filter-proc : (path-string? . -> . boolean?) = std-filter
  encode? : boolean? = #t
  file-mode-sym : symbol? = 'file
  unit200-expr : 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 #: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 a mzlib/unit200-style unit that performs the work of unpacking; see §6.4.3 "Format of ".plt" Archives" more information about the unit. If the #:unpack-unit argument is #f, an appropriate unpacking unit 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.ss"` 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 : boolean? = #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 output filter file-mode) → void?
  path : path-string?
  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 *file-mode* argument specifies the default mode for packing a file, either `'file` or `'file-replace`.

5 `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.

6 `raco setup`: Installation Management

The `raco setup` command finds, compiles, configures, and installs documentation for all collections in a Racket installation. It can also install single `".plt"` files.

6.1 Running `raco setup`

The `raco setup` command performs two main services:

- **Compiling and setting up all (or some of the) collections:** When `raco setup` is run without any arguments, it finds all of the current collections (see §17.2 "Libraries and Collections") and compiles libraries in each 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 §6.1.1 "Controlling `raco setup` with `"info.rkt"` Files" for more information.

  The `-clean` (or `-c`) flag to `raco setup` causes it to 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.

  The `-l` flag takes one or more collection names and restricts `raco setup`'s action to those collections.

  The `-mode` ⟨*mode*⟩ flag causes `raco setup` to use a `".zo"` compiler other than the default compiler, and to 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.

- **Unpacking `".plt"` files:** A `".plt"` file is a platform-independent distribution archive for software based on Racket. When one or more file names are provided as the command line arguments to `raco setup`, 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.

Run `raco help setup` to see a list of all options accepted by the `raco setup` command.

6.1.1 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 §6.2 “"info.rkt" File Format” for information on the format of an "info.rkt" file.

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.

  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-string)

     flags = (list mode-symbol ...)

  category = (list category-symbol)
           | (list category-symbol sort-number)
```

  A document's list optionally continues with information on how to build the document. If a document's list contains a second item, it must be a list of mode symbols (described below). If a document's list contains a third item, it must be a list that categorizes the document (described further below). If a document's list contains a fourth item, it is a name to use for the generated documentation, instead of defaulting to the source file's name (sans extension).

  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-depends-on mode.
  - '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-depends-on mode.

- `'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. This mode is typically used with `'always-run` to avoid unnecessary dependencies that prevent reaching a stable point in building documentation.
- `'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.

The *category* list specifies how to show the document in the root table of contents. The list must start with a symbol, usually one of the following categories, which are ordered as below in the root documentation page:

- `'getting-started` : High-level, introductory documentation.
- `'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.
- `'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.

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.

- `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 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.
- `install-collection` : `path-string?` — A library module relative to the collection that provides `installer`. The `installer` procedure accepts either one or two 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 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.

- `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`. 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.

6.2 `"info.rkt"` File Format

```
#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.

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 intotab-mod-path
                (define id info-expr)
                ...)
```

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

       info-expr = 'datum
                 | `datum
                 | (info-primitive info-expr ...)
                 | id
                 | string
                 | number
                 | boolean
                 | (string-constant identifier)

  info-primitive = cons
                 | car
                 | cdr
                 | list
                 | list*
                 | reverse
                 | append
                 | string-append
                 | 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`, `racket-launcher-libraries`, and `racket-launcher-names`.

```
#lang setup/infotab
(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 `setup/infotab` language.

See also `get-info` from `setup/getinfo`.

6.3 API for Installation

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 ...])
  ...)
```

6.3.1 raco setup Unit

```
(require setup/setup-unit)
```

---

setup@ : unit?

Imports

- setup-option^
- compiler^
- compiler:option^
- launcher^

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

6.3.2 Options Unit

```
(require setup/option-unit)
```

---

setup:option@ : unit?

Imports nothing and exports setup-option^.

6.3.3 Options Signature

```
(require setup/option-sig)
```

---

setup-option^ : signature

Provides parameters used to control raco setup in unit form.

(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? false/c)
(compile-mode path) → void?
  path : (or/c path? false/c)

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

If on, compile `".so"`/`".dll"` files. The default is `#f`.

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

If on, make collection `"info.rkt"`-specified launchers. 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`.

```
(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`.

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

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

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

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

```
(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`.

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

A list of collections to set up; the empty list means set-up all collections if the archives list is also empty The default is `null`.

```
(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 pro-

cedure may be called multiple times. The default is `(lambda (preferred main-parent-dir choices) preferred)`.

6.4 API for Installing ".plt" Archives

6.4.1 Installing a Single ".plt" File

The `setup/plt-single-installer` module provides a function for installing a single ".plt" file, and `setup/plt-installer` wraps it with a GUI interface.

Non-GUI Installer

```
(require setup/plt-single-installer)
```

```
(run-single-installer file get-dir-proc) → void?
  file : path-string?
  get-dir-proc : (-> (or/c path-string? false/c))
```

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 §13.7 "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`. v

---

```
(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.

GUI Installer

```
(require setup/plt-installer)
```

The `setup/plt-installer` library in the setup collection defines procedures for installing a `".plt"` archive with a GUI (using the facilities of `racket/gui/base`).

```
(run-installer filename) → void?
  filename : path-string?
```

Run the installer on the `".plt"` file in *filename* and show the output in a window. This is a composition of `with-installer-window` and `run-single-installer` with a `get-dir-proc` that prompts the user for a directory (turning off the busy cursor while the dialog is active).

```
(on-installer-run) → (-> any)
(on-installer-run thunk) → void?
  thunk : (-> any)
```

A thunk that is run after a `".plt"` file is installed.

```
(with-installer-window do-install
                       cleanup-thunk) → void?
  do-install : ((or/c (is-a?/c dialog%) (is-a?/c frame%))
                . -> . void?)
```

```
  cleanup-thunk : (-> any)
```

Creates a frame, sets up the current error and output ports, and turns on the busy cursor before calling *do-install* in a separate thread.

Returns before the installation process is complete; *cleanup-thunk* is called on a queued callback to the eventspace active when with-installer-window is invoked.

```
(run-single-installer file get-dir-proc) → void?
  file : path-string?
  get-dir-proc : (-> (or/c path-string? false/c))
```

The same as the export from setup/plt-single-installer, but with a GUI.

GUI Unpacking Signature

```
(require setup/plt-installer-sig)
```

---

setup:plt-installer^ : signature

Provides two names: run-installer and on-installer-run.

GUI Unpacking Unit

```
(require setup/plt-installer-unit)
```

Imports mred^ and exports setup:plt-installer^.

6.4.2 Unpacking ".plt" Archives

```
(require setup/unpack)
```

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 : (path-string? any/c . -> . any/c)
  on-file : (path-string? input-port? 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 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, an input port containing the contents of the file, 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.

6.4.3 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.
- A procedure that takes a symbol and a failure thunk and returns information about

archive for recognized symbols and calls the failure thunk for unrecognized symbols. The information symbols are:

- `'name` — a human-readable string describing the archive's contents. This name is used only for printing messages to the user during unpacking.
- `'unpacker` — a symbol indicating the expected unpacking environment. Currently, the only allowed value is `'mzscheme`.
- `'requires` — collections required to be installed before unpacking the archive, which associated versions; see the documentation of `pack` for details.
- `'conflicts` — 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 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 procedure is extracted from the archive using the `read` and `eval` procedures in a fresh namespace.

- An old-style, unsigned unit using `(lib mzlib/unit200)` that drives the unpacking process. The unit accepts two imports: a path string for the parent of the main `"collects"` directory and an `unmztar` procedure. The remainder of the unpacking process consists of invoking this unit. It is expected that the unit will call `unmztar` procedure to unpack directories and files that are defined in the input archive after this unit. The result of invoking the unit must be 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 `unmztar` procedure takes one argument: a filter procedure. The filter procedure is called for each directory and file to be unpacked. It is called with three arguments:

  - `'dir`, `'file`, `'file-replace` — indicates whether the item to be unpacked is a directory, a file, or a file to be replaced,
  - a relative path string — the pathname of the directory or file to be unpacked, relative to the unpack directory, and
  - a path string for the unpack directory (which can vary for a Racket-relative install when elements of the archive start with `"collects"`, `"lib"`, etc.).

  If the filter procedure returns `#f` for a directory or file, the directory or file is not unpacked. If the filter procedure returns `#t` and the directory or file for `'dir` or `'file` already exists, it is not created. (The file for `file-replace` need not exist already.)

When a directory is unpacked, intermediate directories are created as necessary to create the specified directory. When a file is unpacked, the directory must already exist.

The unit is extracted from the archive using `read` and `eval`.

Assuming that the unpacking unit calls the `unmztar` procedure, the archive should continue 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. 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* `eval`uated).

- 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* `eval`uated). 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.

6.5 API for Finding Installation Directories

```
(require setup/dirs)
```

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

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

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.

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

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? false/c)`

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-dll-dir)` → `(or/c path? false/c)`

Returns a path to the directory that contains DLLs for use with the current executable (e.g., `"libmzsch.dll"` under 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-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 libraries. Unless it is configured otherwise, the result includes any non-`#f` result of `(find-lib-dir)`, `(find-dll-dir)`, and `(find-user-lib-dir)`—but the last is included only if the value of the `use-user-specific-search-paths` parameter is `#t`.

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

Returns a path to the installation's `"include"` directory, which contains `".h"` files for building MzRacket 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? false/c)`

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

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

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.

`absolute-installation?` : `boolean?`

A binary boolean flag that is true if this installation is using absolute path names.

6.6 API for Reading `"info.rkt"` Files

```
(require setup/getinfo)
```

The setup/getinfo library provides functions for accessing fields in "info.rkt" files.

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

Accepts a path to a directory. If it finds either a well-formed an "info.rkt" file or an "info.rkt" 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 setup/infotab or (lib "infotab.rkt" "setup")), 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.

```
(find-relevant-directories syms [mode]) → (listof path?)
  syms : (listof symbol?)
  mode : (symbols 'preferred 'all-available) = 'preferred
```

Returns a list of paths identifying installed directories (i.e., collections and installed PLane T 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` and stored in the `"info-domain"` sub-directory of each collection directory (as determined by the `PLT_COLLECTION_PATHS` environment variable, etc.) and the file `"cache.rkt"` in the user add-on directory.

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) or `'all-available`. If mode is `'all-available`, `find-relevant-collections` returns all installed directories whose info files contain the specified symbols—for instance, 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, and in particular only the directory containing the most recent version of any PLaneT package will be returned.

No matter what *mode* is specified, if more than one collection has the same name, `find-relevant-directories` will only search the one that occurs first in the `PLT_COLLECTION_PATHS` environment variable.

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

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

6.7 API for Paths Relative to `"collects"`

```
(require setup/main-collects)
```

```
(path->main-collects-relative path)
 → (or/c path? (cons/c 'collects (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`.

`(main-collects-relative->path rel)` → `path?`

```
  rel : (or/c bytes? path-string?
              (cons/c 'collects
                      (or/c (listof bytes?) 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)`.

For historical reasons, a single byte string is allowed in place of a list of byte strings after `'collects`, in which case it is assumed to be a relative path after conversion with `bytes->path`.

Also for historical reasons, if *rel* is any kind of value other than specified in the contract above, it is returned as-is.

6.8 API for Cross-References for Installed Manuals

```
(require setup/xref)
```

```
(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 `setup-plt`.

7 `raco decompile`: Decompiling Bytecode

The `raco decompile` command takes 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 it back to an approximation of Racket code. Decompiled bytecode is mostly useful for checking the compiler's transformation and optimization of the source program.

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 that indicates the source module.

  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*.

- Some applications of core primitives are annotated with `#%in`, which indicates that the JIT compiler will inline the operation. (Inlining information is not part of the bytecode, but is instead based on an enumeration of primitives that the JIT is known to handle specially.) Operations from `racket/flonum` and `racket/unsafe/ops` are always inlined, so `#%in` is not shown for them.

- Some applications of flonum operations from `racket/flonum` and `racket/unsafe/ops` are annotated with `#%flonum`, indicating a place where the JIT compiler might avoid allocation for intermediate flonum results. A single `#%flonum` by itself is not useful, but a `#%flonum` operation that consumes a `#%flonum` or `#%from-flonum` argument indicates a potential performance improvement. A `#%from-flonum` wraps an identifier that is bound by `let` with a `#%as-flonum` around its value, which indicates a local binding that can avoid boxing (when used as an argument to an operation that can work with unboxed values).

- A `#%decode-syntax` form corresponds to a syntax object. Future improvements to the decompiler will convert such syntax objects to a readable form.

7.1 API for Decompiling

```
(require compiler/decompile)
```

```
(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)
```

```
(zo-parse in) → compilation-top?
```

in : 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.

7.2.1 Prefix

```
(struct compilation-top (max-let-depth prefix code)
        #:extra-constructor-name make-compilation-top
        #:transparent)
  max-let-depth : exact-nonnegative-integer?
  prefix : prefix?
  code : (or/c form? indirect? any/c)
```

Wraps compiled code. The `max-let-depth` field indicates the maximum stack depth that `code` creates (not counting the `prefix` array). 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 (num-lifts toplevels stxs)
        #:extra-constructor-name make-prefix
        #:transparent)
  num-lifts : exact-nonnegative-integer?
  toplevels : (listof (or/c #f symbol? global-bucket? module-variable?))
  stxs : (listof stx?)
```

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 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.

```
(struct global-bucket (name)
        #:extra-constructor-name make-global-bucket
        #:transparent)
```

```
  name : symbol?
```

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

```
(struct module-variable (modidx sym pos phase)
        #:extra-constructor-name make-module-variable
        #:transparent)
  modidx : module-path-index?
  sym : symbol?
  pos : exact-integer?
  phase : (or/c 0 1)
```

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.

```
(struct stx (encoded)
        #:extra-constructor-name make-stx
        #:transparent)
  encoded : wrapped?
```

Wraps a syntax object in a `prefix`.

7.2.2 Forms

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

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

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

Represents a `define-syntaxes` or `define-values-for-syntax` form. The `rhs` expression has its own `prefix`, which is pushed before evaluating `rhs`; 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`).

```
(struct req form (reqs dummy)
        #:extra-constructor-name make-req
        #:transparent)
  reqs : syntax?
  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 to the top-level namespace.

```
(struct seq form (forms)
        #:extra-constructor-name make-seq
        #:transparent)
  forms : (listof (or/c form? indirect? 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
        #:transparent)
  forms : (listof (or/c form? indirect? 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 mod form (name
                  srcname
                  self-modidx
                  prefix
                  provides
                  requires
                  body
                  syntax-body
                  unexported
                  max-let-depth
                  dummy
                  lang-info
                  internal-context)
         #:extra-constructor-name make-mod
         #:transparent)
  name : 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? indirect? any/c))
  syntax-body : (listof (or/c def-syntaxes? def-for-syntax?))
  unexported : (list/c (listof symbol?) (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?)
```

Represents a `module` declaration. The `body` forms use `prefix`, rather than any prefix in place for the module declaration itself (and each `syntax-body` has its own prefix).

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 code, and `syntax-body` contains the module's compile-time code. After each form in `body` or `syntax-body` 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. The first list is phase-0 variables, the second is phase-0 syntax, and the last is phase-1 variables.

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-module-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.

```
(struct provided (name
                  src
                  src-name
                  nom-mod
                  src-phase
                  protected?
                  insp)
        #:extra-constructor-name make-provided
        #:transparent)
  name : symbol?
  src : (or/c module-path-index? #f)
  src-name : symbol?
  nom-mod : (or/c module-path-index? #f)
  src-phase : (or/c 0 1)
  protected? : boolean?
  insp : (or #t #f void?)
```

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

7.2.3 Expressions

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

A supertype for all expression forms that can appear in compiled code, except for literals that are represented as themselves, `indirect` structures to create cycles, 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
                  max-let-depth
                  body)
         #:extra-constructor-name make-lam
         #:transparent)
  name : (or/c symbol? vector?)
  flags : (listof (or/c 'preserves-marks 'is-method 'single-result))
  num-params : exact-nonnegative-integer?
  param-types : (listof (or/c 'val 'ref 'flonum))
  rest? : boolean?
  closure-map : (vectorof exact-nonnegative-integer?)
  closure-types : (listof (or/c 'val/ref 'flonum))
  max-let-depth : exact-nonnegative-integer?
  body : (or/c expr? seq? indirect? 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), or `'flonum` for a flonum 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 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.

```
(struct closure expr (code gen-id)
        #:extra-constructor-name make-closure
        #:transparent)
  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 refer back to the same `closure` through an `indirect` for a recursive constant procedure; the `gen-id` is different for each such constant.

```
(struct indirect (v)
        #:extra-constructor-name make-indirect
        #:mutable
        #:prefab)
  v : closure?
```

An indirection used in expression positions to form cycles.

```
(struct case-lam expr (name clauses)
        #:extra-constructor-name make-case-lam
        #:transparent)
  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 flonum? unused?)
        #:extra-constructor-name make-let-one
        #:transparent)
  rhs : (or/c expr? seq? indirect? any/c)
  body : (or/c expr? seq? indirect? any/c)
  flonum? : boolean?
  unused? : boolean?
```

Pushes an uninitialized slot onto the stack, evaluates `rhs` and puts its value into the slot, and then runs `body`. If `flonum?` is `#t`, then `rhs` must produce a flonum, and the slot must be accessed by `localref`s that expect a flonum. 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
        #:transparent)
  count : exact-nonnegative-integer?
  boxes? : boolean?
  body : (or/c expr? seq? indirect? 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
        #:transparent)
  count : exact-nonnegative-integer?
  pos : exact-nonnegative-integer?
  boxes? : boolean?
  rhs : (or/c expr? seq? indirect? any/c)
  body : (or/c expr? seq? indirect? 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
        #:transparent)
  procs : (listof lam?)
  body : (or/c expr? seq? indirect? 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
        #:transparent)
  pos : exact-nonnegative-integer?
  body : (or/c expr? seq? indirect? 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? flonum?)
        #:extra-constructor-name make-localref
        #:transparent)
  unbox? : boolean?
  pos : exact-nonnegative-integer?
  clear? : boolean?
  other-clears? : boolean?
  flonum? : boolean?
```

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 `flonum?` is `#t`, the slot holds to a flonum value.

```
(struct toplevel expr (depth pos const? ready?)
        #:extra-constructor-name make-toplevel
        #:transparent)
  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.

If `const?` is `#t`, then the variable definitely will be defined, and its value stays constant. If `ready?` is `#t`, then the variable definitely will be defined (but its value might change in the

future). If const? and ready? are both #f, then a check is needed to determine whether the variable is defined.

```
(struct topsyntax expr (depth pos midpt)
        #:extra-constructor-name make-topsyntax
        #:transparent)
  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
        #:transparent)
  rator : (or/c expr? seq? indirect? any/c)
  rands : (listof (or/c expr? seq? indirect? 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
        #:transparent)
  test : (or/c expr? seq? indirect? any/c)
  then : (or/c expr? seq? indirect? any/c)
  else : (or/c expr? seq? indirect? 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
        #:transparent)
  key : (or/c expr? seq? indirect? any/c)
  val : (or/c expr? seq? indirect? any/c)
  body : (or/c expr? seq? indirect? 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
        #:transparent)
  seq : (listof (or/c expr? seq? indirect? 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.

```
(struct varref expr (toplevel)
        #:extra-constructor-name make-varref
        #:transparent)
  toplevel : toplevel?
```

Represents a `#%variable-reference` form.

```
(struct assign expr (id rhs undef-ok?)
        #:extra-constructor-name make-assign
        #:transparent)
  id : toplevel?
  rhs : (or/c expr? seq? indirect? 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.)

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
        #:transparent)
  proc : (or/c expr? seq? indirect? any/c)
  args-expr : (or/c expr? seq? indirect? any/c)
```

Represents `(call-with-values (lambda () args-expr) proc)`, which is handled specially by the run-time system.

```
(struct primval expr (id)
        #:extra-constructor-name make-primval
        #:transparent)
  id : exact-nonnegative-integer?
```

Represents a direct reference to a variable imported from the run-time kernel.

7.2.4 Syntax Objects

```
(struct wrapped (datum wraps certs)
        #:extra-constructor-name make-wrapped
        #:transparent)
  datum : any/c
  wraps : (listof wrap?)
  certs : (or/c list? #f)
```

Represents a syntax object, where `wraps` contain the lexical information and `certs` is certificate information. When the `datum` part is itself compound, its pieces are wrapped, too.

```
(struct wrap ()
        #:extra-constructor-name make-wrap
        #:transparent)
```

A supertype for lexical-information elements.

```
(struct lexical-rename wrap (alist)
        #:extra-constructor-name make-lexical-rename
        #:transparent)
  alist : (listof (cons/c symbol? symbol?))
```

A local-binding mapping from symbols to binding-set names.

```
(struct phase-shift wrap (amt src dest)
        #:extra-constructor-name make-phase-shift
        #:transparent)
  amt : exact-integer?
  src : module-path-index?
  dest : module-path-index?
```

Shifts module bindings later in the wrap set.

```
(struct module-rename wrap (phase
                            kind
                            set-id
                            unmarshals
                            renames
                            mark-renames
                            plus-kern?)
        #:extra-constructor-name make-module-rename
        #:transparent)
  phase : exact-integer?
  kind : (or/c 'marked 'normal)
  set-id : any/c
  unmarshals : (listof make-all-from-module?)
  renames : (listof module-binding?)
  mark-renames : any/c
  plus-kern? : boolean?
```

Represents a set of module and import bindings.

```
(struct all-from-module (path phase src-phase exceptions prefix)
        #:extra-constructor-name make-all-from-module
        #:transparent)
  path : module-path-index?
  phase : (or/c exact-integer? #f)
  src-phase : (or/c exact-integer? #f)
  exceptions : (listof symbol?)
  prefix : (or/c symbol? #f)
```

Represents a set of simple imports from one module within a `module-rename`.

```
(struct module-binding ()
        #:extra-constructor-name make-module-binding
        #:transparent)
```

A supertype for module bindings.

```
(struct simple-module-binding module-binding (path)
        #:extra-constructor-name make-simple-module-binding
        #:transparent)
  path : module-path-index?
```

Represents a single identifier import within a `module-rename`.

```
(struct phased-module-binding module-binding (path
                                              phase
                                              export-name
                                              nominal-path
                                              nominal-export-name)
        #:extra-constructor-name make-phased-module-binding
        #:transparent)
  path : module-path-index?
  phase : exact-integer?
  export-name : any/c
  nominal-path : nominal-path?
  nominal-export-name : any/c
```

Represents a single identifier import within a module-rename.

```
(struct exported-nominal-module-binding module-binding (path
                                                        export-name
                                                        nominal-path
                                                        nominal-export-name)
        #:extra-constructor-name make-exported-nominal-module-binding
        #:transparent)
  path : module-path-index?
  export-name : any/c
  nominal-path : nominal-path?
  nominal-export-name : any/c
```

Represents a single identifier import within a module-rename.

```
(struct nominal-module-binding module-binding (path nominal-path)
        #:extra-constructor-name make-nominal-module-binding
        #:transparent)
  path : module-path-index?
  nominal-path : nominal-path?
```

Represents a single identifier import within a module-rename.

```
(struct exported-module-binding module-binding (path export-name)
        #:extra-constructor-name make-exported-module-binding
        #:transparent)
  path : module-path-index?
  export-name : any/c
```

Represents a single identifier import within a module-rename.

```
(struct nominal-path ()
        #:extra-constructor-name make-nominal-path
        #:transparent)
```

A supertype for nominal paths.

```
(struct simple-nominal-path nominal-path (value)
        #:extra-constructor-name make-simple-nominal-path
        #:transparent)
  value : module-path-index?
```

Represents a simple nominal path.

```
(struct imported-nominal-path nominal-path (value import-phase)
        #:extra-constructor-name make-imported-nominal-path
        #:transparent)
  value : module-path-index?
  import-phase : exact-integer?
```

Represents an imported nominal path.

```
(struct phased-nominal-path nominal-path (value import-phase phase)
        #:extra-constructor-name make-phased-nominal-path
        #:transparent)
  value : module-path-index?
  import-phase : (or/c false/c exact-integer?)
  phase : exact-integer?
```

Represents a phased nominal path.

7.3 API for Marshaling Bytecode

```
(require compiler/zo-marshal)
```

```
(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.

8 `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.

8.1 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.

8.1.1 API for 3m Transformation

```
(require compiler/xform)
```

```
(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`.

8.2 Embedding Modules via C

The `-c-mods` mode for `raco ctool` takes a set of Scheme modules and generates a C source file that can be used as part of program that embeds the Racket run-time system. See §1.4 "Embedding Racket 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 PLT Scheme with a set of modules so that it does not need a `"collects"` directory to load modules at run time.

8.3 Compiling to Native Code via C

The `-extension`/`-e` mode for `raco ctool` is similar to the `raco make -zo` (see §1.5 "Compiling to Raw Bytecode"), except that the compiled form of the module is a native-code shared library instead of bytecode. Native code is generated with the help of the host system's C compiler. This mode is rarely useful, because the just-in-time (JIT) compiler that

is built into Racket provides better performance with lower overhead on the platforms where it is supported (see §18 “Performance”).

As with `-zo` mode, the generated shared library by default is placed in the same directory as the source file—which is not where it will be found automatically when loading the source. Use the `-auto-dir` flag to redirect the output to a `(build-path "compiled" "native" (system-library-subpath))` subdirectory, where it will be found automatically when loading the source file.

The `-c-source/-c` mode for `raco ctool` is like the `-extension/-e` mode, except that compilation stops with the generation of C code.

All of the C compiler and linker flags that apply to `-cc` and `-ld` mode also apply to `-extension` mode; see §8.1 “Compiling and Linking C Extensions”. In addition, a few flag provide some control over the Racket-to-C compiler: `-no-prop`, `-inline`, `-no-prim`, `-stupid`, `-unsafe-disable-interrupts`, `-unsafe-skip-tests`, and `-unsafe-fixnum-arithmetic`. Use `mzc -help` for an explanation of each flag.

8.4 API for Raw Compilation

```
(require compiler/compiler)
```

The `compiler/compiler` library provides the functionality of `mzc` for compilation to bytecode and via C, but through a Scheme API.

8.4.1 Bytecode Compilation

```
((compile-zos  expr
               [#:module? module?
                #:verbose? verbose?])
               scheme-files
               dest-dir)              → void?
  expr : any/c
  module? : any/c = #f
  verbose? : any/c = #f
  scheme-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 Scheme 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-doc-sources? skip-docs?]) → void?
  collection : string?
  skip-path : (or/c path-string? #f) = #f
  skip-docs? : any/c = #f
```

Compiles the specified collection's files to ".zo" files. 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, execept that any file or directory whose path starts with scheme-path 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 §6.2 ""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 ".rkt", ".ss", and ".scm" files, unless the provided *skip-docs?* argument is a true value.

The compilation process for an individual file is driven by managed-compile-zo from compiler/cm.

```
(compile-directory-zos  path
                        info
                       [#:verbose verbose?
                        #:skip-path skip-path
                        #:skip-doc-sources? skip-docs?]) → void?
  path : path-string?
  info : ()
  verbose? : any/c = #f
  skip-path : (or/c path-string? #f) = #f
  skip-docs? : any/c = #f
```

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.

8.4.2 Compilation via C

```
((compile-extensions expr)
                     scheme-files
                     dest-dir)    → void?
  expr : any/c
  scheme-files : (listof path-string?)
  dest-dir : (or/c path-string? false/c (one-of/c 'auto))
```

Like compile-zos, but the *scheme-files* are compiled to native-code extensions via C. If *dest-dir* is 'auto, each extension file (".dll", ".so", or ".dylib") is placed in a subdirectory relative to the source produced by (build-path "compiled" "native" (system-library-subpath)); the directory is created if necessary.

```
((compile-extensions-to-c expr)
                          scheme-files
                          dest-dir)    → void?
  expr : any/c
  scheme-files : (listof path-string?)
  dest-dir : (or/c path-string? false/c (one-of/c 'auto))
```

Like compile-extensions, but only ".c" files are produced, not extensions.

```
(compile-c-extensions c-files dest-dir) → void?
  c-files : (listof path-string?)
  dest-dir : (or/c path-string? false/c (one-of/c 'auto))
```

Compiles each ".c" file (usually produced with compile-extensions-to-c) in *c-files* to an extension. The *dest-dir* argument is handled as in compile-extensions.

8.4.3 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.

8.4.4 Options for the Compiler

```
(require compiler/option)
```

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.

```
(setup-prefix) → string?
(setup-prefix str) → void?
  str : string?
```

A parameter that specifies a string to embed in public function names when compiling via C. This is used mainly for compiling extensions with the collection name so that cross-extension conflicts are less likely in architectures that expose the public names of loaded extensions. The default is "".

```
(clean-intermediate-files) → boolean?
(clean-intermediate-files clean?) → void?
  clean? : any/c
```

A #f value for the parameter keeps intermediate ".c" and ".o" files generated during compilation via C. The default is #t.

```
(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.

```
(compile-for-embedded) → boolean?
(compile-for-embedded embed?) → void?
  embed? : any/c
```

A `#t` values for this parameter creates `".c"` files and object files to be linked directly with an embedded PLT Scheme run-time system, instead of `".c"` files and object files to be dynamically loaded into PLT Scheme as an extension. The default is `#f`.

```
(propagate-constants) → boolean?
(propagate-constants prop?) → void?
  prop? : any/c
```

A parameter to control the compiler's constant propagating when compiling via C. The default is `#t`.

```
(assume-primitives) → boolean?
(assume-primitives assume?) → void?
  assume? : any/c
```

A `#t` parameter value effectively adds `(require mzscheme)` to the beginning of the program. This parameter is useful only when compiling non-`module` code. The default is `#f`.

```
(stupid) → boolean?
(stupid allow?) → void?
  allow? : any/c
```

A parameter that allow obvious non-syntactic errors, such as `((lambda () 0) 1 2 3)`, when compiling via C. The default is `#f`.

```
(vehicles) → symbol?
(vehicles mode) → void?
  mode : symbol?
```

A parameter that controls how closures are compiled via C. The possible values are:

- `'vehicles:automatic` : automatic grouping
- `'vehicles:functions` : groups within a procedure
- `'vehicles:monolithic` : groups randomly

```
(vehicles:monoliths) → exact-nonnegative-integer?
```

```
(vehicles:monoliths count) → void?
  count : exact-nonnegative-integer?
```

A parameter that determines the number of random groups for `'vehicles:monolithic` mode.

```
(seed) → exact-nonnegative-integer?
(seed val) → void?
  val : exact-nonnegative-integer?
```

Sets the randomizer seed for `'vehicles:monolithic` mode.

```
(max-exprs-per-top-level-set) → exact-nonnegative-integer?
(max-exprs-per-top-level-set n) → void?
  n : exact-nonnegative-integer?
```

A parameter that determines the number of top-level Scheme expressions crammed into one C function when compiling via C. The default is `25`.

```
(unpack-environments) → boolean?
(unpack-environments unpack?) → void?
  unpack? : any/c
```

Setting this parameter to `#f` might help compilation via C for register-poor architectures. The default is `#t`.

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

A `#t` creates a `"debug.txt"` debugging file when compiling via C. The default is `#f`.

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

A `#t` value for this parameter causes compilation via C to ignore top-level expressions with syntax errors. The default is `#f`.

8.4.5 The Compiler as a Unit

Signatures

```
(require compiler/sig)
```

`compiler^` : signature

Includes all of the names exported by `compiler/compiler`.

`compiler:option^` : signature

Includes all of the names exported by `compiler/option`.

`compiler:inner^` : signature

The high-level `compiler/compiler` interface relies on a low-level implementation of the extension compiler, which is available from `compiler/comp-unit` as implementing the `compiler:inner^` signature.

`(eval-compile-prefix` *expr*`)` → `void?`
  expr : `any/c`

Evaluates *expr*. Future calls to `compile-extension` or `compile-extension-to-c` see the effects of the evaluation.

`(compile-extension` *scheme-source dest-dir*`)` → `void?`
  scheme-source : `path-string?`
  dest-dir : `path-string?`

Compiles a single Scheme file to an extension.

`(compile-extension-to-c` *scheme-source*
  dest-dir`)` → `void?`
  scheme-source : `path-string?`
  dest-dir : `path-string?`

Compiles a single Scheme file to a `".c"` file.

`(compile-c-extension` *c-source dest-dir*`)` → `void?`
  c-source : `path-string?`
  dest-dir : `path-string?`

Compiles a single `".c"` file to an extension.

Main Compiler Unit

```
(require compiler/compiler-unit)
```

---

compiler@ : unit?

Provides the exports of compiler/compiler in unit form, where C-compiler operations are imports to the unit.

The unit imports compiler:option^, dynext:compile^, dynext:link^, and dynext:file^. It exports compiler^.

Options Unit

```
(require compiler/option-unit)
```

---

compiler:option@ : unit?

Provides the exports of compiler/option in unit form. It imports no signatures, and exports compiler:option^.

Compiler Inner Unit

```
(require compiler/comp-unit)
```

---

comp@ : unit?

The unit imports compiler:option^, dynext:compile^, dynext:link^, and dynext:file^. It exports compiler:inner^.

9 Adding a `raco` Command

The set of commands supported by `raco` can be extended by installed collections and PLane T packages. A command is added by defining `raco-commands` in the `"info.rkt"` library of a collection or package (see §6.2 “`"info.rkt"` File Format”).

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
      implementationmodule-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 *`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.

9.1 Command Argument Parsing

```
(require raco/command-name)
```

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).

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.