mzc: PLT Compilation and Packaging

Version 4.2.2

October 4, 2009

The mzc tool supports various PLT Scheme compilation and packaging tasks.

Contents

1 Running `mzc`

The main action of `mzc` is determined through one of the following command-line flags:

- `-make` (the default), `-k` or `-make-collection` : Compiles Scheme modules and all transitive imports to bytecode. See §2 “Compiling Modified Modules to Bytecode”.
- `-exe`, `-gui-exe`, or `-exe-dir` : Creates an executable to run a Scheme module, or assembles all support libraries to move an executable to a new filesystem. See §3.1 “Stand-Alone Executables from Scheme Code”.
- `-collection-plt` or `-plt` : packages Scheme code for installation into a different PLT Scheme installation. See §4 “Packaging Library Collections”. PLaneT is usually a better alternative.
- `-cc`, `-ld`, `-xform` or `-x` : Compiles, links or transforms (for GC cooperation) C code to extend the PLT Scheme runtime system. See §5 “Compiling and Linking C Extensions”. Using the `scheme/foreign` FFI is often better; see ***FFI**: PLT Scheme Foreign Interface*.
- `-c-mods` : Creates C source to embed Scheme modules into an executable that also embeds PLT Scheme. See §6 “Embedding Scheme Modules via C”.
- `-expand` : Pretty-prints the macro-expanded form of a Scheme program.
- `-decompile` : Parses a bytecode file and prints its content as quasi-Scheme. See §7 “Decompiling Bytecode”.
- `-zo`, `-z`, or `-collection-zo` : Compiles Scheme code to bytecode, without following transitive imports. See §8 “Compiling to Raw Bytecode”. *This mode is rarely useful.*
- `-extension`, `-e`, `-c-source`, or `-c` : Compiles Scheme code to a native-code extension via C. See §9 “Compiling to Native Code via C”. *This mode is rarely useful.*

2 Compiling Modified Modules to Bytecode

The default mode for `mzc` is to accept filenames for Scheme modules to be compiled to bytecode format. Modules are re-compiled only if the source Scheme file is newer than the bytecode file, or if any imported module is recompiled.

2.1 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.scm":

```
#lang scheme
(require "b.scm" "c.scm")
(+ b c)
```

- "b.scm":

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

- "c.scm":

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

then

```
mzc a.scm
```

triggers the creation of "compiled/a_ss.zo", "compiled/b_ss.zo", and "compiled/c_ss.zo". A subsequent

```
mzscheme a.scm
```

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

In contrast,

```
mzc b.scm c.scm
```

would create only "compiled/b_scm.zo" and "compiled/c_scm.zo", since neither "b.scm" nor "c.scm" imports "a.scm".

2.2 Dependency Files

In addition to a bytecode file, mzc creates a file "compiled/⟨*name*⟩_⟨*ext*⟩.dep" that records dependencies of the compiled module on other module files. Using this dependency information, a re-compilation request via mzc can consult both the source file's timestamp and the timestamps for the sources and 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 mzc a.scm example from the previous section, the mzc creates "compiled/a_scm.dep", "compiled/b_scm.dep", and "compiled/c_scm.dep" at the same time as the ".zo" files. The "compiled/a_scm.dep" file records the dependency of "a.scm" on "b.scm", "c.scm" and the scheme library. If the "b.scm" file is modified (so that its timestamp changes), then running

```
mzc a.scm
```

again rebuilds "compiled/a_ss.zo" and "compiled/b_ss.zo".

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

2.3 Scheme Compilation Manager API

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

The compiler/cm module implements the compilation and dependency management used by mzc and setup-plt.

```
(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)
  - 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 one recorded in the ".dep" file.

After the handler procedure compiles a ".zo" file, it creates a corresponding ".dep" file that lists the current version, plus the ".zo" timestamp for every file that is required by the module in the compiled file. Additional dependencies can be installed during compilation via compiler/cm-accomplice.

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

A parameter whose value is called for each file that is loaded and needs recompilation. If the procedure returns a number, then the file is skipped (i.e., not compiled), and the number is used as the timestamp for the file's bytecode. If the procedure returns `#f`, then the file is compiled as usual. The default is `(lambda (x) #f)`.

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

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

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

This is a function intended to be used with `manager-skip-file-handler`. It returns the date of the `.ss` or `.zo` file (whichever is newer) for any path that is inside the *paths* argument, and `#f` for any other path.

2.4 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.

3 Creating and Distributing Stand-Alone Executables

Whether bytecode or native code, the compiled code produced by `mzc` relies on PLT Scheme executables to provide run-time support to the compiled code. However, `mzc` can also package code together with its run-time support to form a complete executable, and then the executable can be packaged into a distribution that works on other machines.

3.1 Stand-Alone Executables from Scheme Code

The command-line flag `-exe` directs `mzc` to embed a module, from source or byte code, into a copy of the `mzscheme` 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-exe` flag is similar, but it copies the `mred` 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

```
mzc -gui-exe hello hello.ss
```

produces either `"hello.exe"` (Windows), `"hello.app"` (Mac OS X), or `"hello"` (Unix), which runs the same as invoking the `"hello.ss"` module in `mred`.

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 `mzc`'s `-lib` flag. 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 §3.2 "Distributing 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 `-exe` and `-gui-exe` flags work 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 `mzscheme`, `mred`, or DrScheme. However, the executable depends on PLT Scheme 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, as described in §3.2 "Distributing Stand-Alone Executables".

3.1.1 Scheme API for Creating Executables

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

The compiler/embed library provides a function to embed Scheme code into a copy of MzScheme or MrEd, thus creating a stand-alone Scheme executable. To package the executable into a distribution that is indpendent of your PLT 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 (MzScheme or MrEd). 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
  #:literal-files literal-files
  #:literal-expression literal-sexp
  #:literal-expressions literal-sexps
  #:cmdline cmdline
  #: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
 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
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 MzScheme (if *mred?* is `#f`) or MrEd (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.

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 "drscheme.filetypes" in the "drscheme" 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 "drscheme.utiexports" in the "drscheme" 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 MzScheme and MrEd frameworks (including a separating slash); note that when the prefix starts "@executable_path/" works for a MzScheme-based application, the corresponding prefix start for a MrEd-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 PLT DLLs needed by the executable, such as "pltmzsch⟨*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 MzScheme-based application and 'windows for a MrEd-based application; see also 'single-instance?, below.

- 'single-instance? (Windows) : A boolean for MrEd-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 MzScheme or MrEd executable, instead of a wrapper binary that execs the original; the default is #f.

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 MzScheme or MrEd 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 MzScheme/MrEd 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 MzScheme/MrEd 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 MzScheme/MrEd 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.

Executable Creation Signature

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

`compiler:embed^` : signature

Includes the identifiers provided by `compiler/embed`.

Executable Creation Unit

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

---

compiler:embed@ : unit?

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

3.2 Distributing Stand-Alone Executables

The command-line flag -exe-dir directs mzc to combine a stand-alone executable (created via -exe or -gui-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 -exe-dir flag, 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,

```
mzc -exe-dir geetings 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-exe executables go into the distribution directory, -exe 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 PLT Scheme installation or other executables created by mzc.

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 mzc's ++copy-collects flag 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 disrtibuted together, then separately creating the executables with -exe and -gui-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 -exe and -gui-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 -exe-dir, use the ++copy-collects flag to include the copied libraries in the distribution.

3.2.1 Scheme API for Distributing Executables

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

The compiler/distribute library provides a function to perform the same work as mzc -exe or mzc -gui-exe.

```
(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.

3.2.2 Scheme 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.

3.3 Installation-Specific Launchers for Scheme Code

```
(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 PLT Scheme installation. In the case of Unix, in particular, a launcher is simply a shell script. The mzc tool provides no direct support for creating launchers.

3.3.1 Creating Launchers

```
(make-mred-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 MrEd 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 MrEd. 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 MrEd; the following additional associations apply to launchers:

- 'independent? (Windows) — a boolean; #t creates an old-style launcher that is independent of the MzScheme or MrEd 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 MrEd executable as usual.
- 'relative? (all platforms) — a boolean, where #t means that the generated launcher should find the base MrEd 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-mzscheme-launcher args dest [aux]) → void?
  args : (listof string?)
  dest : path-string?
  aux : (listof (cons/c symbol? any/c)) = null
```

Like make-mred-launcher, but for starting MzScheme. Under Mac OS X, the 'exe-name *aux* association is ignored.

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

Calls make-mred-launcher with arguments that start the MrEd program implemented by *file* in *collection*: (list "-l-" (string-append *collection* "/" *file*)).

The *aux* argument to make-mred-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-mzscheme-program-launcher file
                                collection
                                dest)      → void?
  file : string?
  collection : string?
  dest : path-string?
```

Like make-mred-program-launcher, but for make-mzscheme-launcher.

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

Same as

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

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

Same as

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

3.3.2 Launcher Path and Platform Conventions

```
(mred-program-launcher-path name) → path?
```

name : string?

Returns a pathname for an executable in the PLT Scheme 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 PLT Scheme installation. For Mac OS X, the ".app" suffix is appended to *name*.

(mzscheme-program-launcher-path *name*) → path?
name : string?

Returns the same path as (mred-program-launcher-path *name*) for Unix and Windows. For Mac OS X, the result is the same as for Unix.

(mred-launcher-is-directory?) → boolean?

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

(mzscheme-launcher-is-directory?) → boolean?

Like mred-launcher-is-directory?, but for MzScheme launchers.

(mred-launcher-is-actually-directory?) → boolean?

Returns #t if MrEd 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.

(mzscheme-launcher-is-actually-directory?) → boolean?

Like mred-launcher-is-actuall-directory?, but for MzScheme launchers. The result is #f for all platforms.

(mred-launcher-add-suffix *path-string?*) → path?
path-string? : path

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

(mzscheme-launcher-add-suffix *path-string?*) → path?
path-string? : path

Like mred-launcher-add-suffix, but for MzScheme launchers.

```
(mred-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 MrEd 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.

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

Like `mred-launcher-get-file-extension+style+filters`, but for MzScheme launchers.

3.3.3 Launcher Configuration

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

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

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

Analogous to `mred-launcher-up-to-date?`, but for a MzScheme launcher.

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

Creates an association list suitable for use with `make-mred-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 MzScheme or MrEd 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 MrEd launchers.

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

Returns a list of symbols corresponding to available variants of MrEd in the current PLT Scheme 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-mzscheme-variants) → (listof symbol?)
```

Returns a list of symbols corresponding to available variants of MzScheme in the current PLT Scheme 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.

3.3.4 Launcher Creation Signature

```
(require launcher/launcher-sig)
```

`launcher^` : signature

Includes the identifiers provided by `launcher/launcher`.

3.3.5 Launcher Creation Unit

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

`launcher@` : `unit?`

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

4 Packaging Library Collections

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

The command-line flags `-plt` and `-collection-plt` direct `mzc` to create an archive for distributing library files to PLT Scheme users. A distribution archive usually has the suffix `".plt"`, which DrScheme recognizes as an archive to provide automatic unpacking facilities. The `setup-plt` program 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 PLT Scheme add-ons directory (which is user-specific), the PLT Scheme installation directory, or a user-selected directory.

  The files and directories for an archive are provided on the command line to `mzc`, either directly with `-plt` or in the form of collection names with `-collection-plt`.

  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 PLT Scheme 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 PLT Scheme 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 `mzc`'s `-replace` flag enables replacing for all files in the archive.

- A list of collections to be set-up (via Setup PLT) after the archive is unpacked; `mzc`'s `++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; `mzc`'s `-plt-name` flag sets the archive's name, but a default name is determined automatically for `-collection-plt`.

- A list of required collections (with associated version numbers) and a list of conflicting collections; `mzc` always names the `"mzscheme"` collection in the required list (using the collection's pack-time version), `mzc` 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 `mzc` extracts other requirements and conflicts from the `"info.ss"` files of collections for `-collection-plt`.

Use the `-plt` flag to specify individual directories and files for the archive. Each file and directory must be specified with a relative path. By default, if the archive is unpacked with DrScheme, the user will be prompted for a target directory, and if `setup-plt` 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 to `mzc`, the files and directories will be unpacked relative to the user's PLT Scheme add-ons directory, instead. Finally, if the `-all-users` flag is provided

to `mzc`, the files and directories will be unpacked relative to the PLT Scheme installation directory, instead.

Use the `-collection-plt` 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, `mzc` automatically uses paths relative to the PLT Scheme installation or add-ons directory for the archived files, and the collections will be set-up after unpacking. In addition, `mzc` consults each collection's `"info.ss"` file, as described below, to determine the set of required and conflicting collections. Finally, `mzc` 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:

```
mzc -collection-plt sirmail.plt sirmail
```

When packing collections, `mzc` checks the following fields of each collection's `"info.ss"` file (see §5 "`"info.ss"` 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.ss"` file in the `"sirmail"` collection might contain the following `info` declaration:

```
#lang setup/infotab
(define name "SirMail")
(define mred-launcher-libraries (list "sirmail.ss"))
(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 (not just MzScheme), and that MrEd has the same version as when `"sirmail.plt"` was created.

4.1 Scheme API for Packaging

Although `mzc`'s command-line interface is sufficient for most purposes, see the `setup/pack` library for a more general interface for constructing archives.

5 Compiling and Linking C Extensions

A *dynamic extension* is a shared library (a.k.a. DLL) that extends PLT Scheme 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**: PLT Scheme C API*.

Three `mzc` modes help for building extensions:

- `-cc` : Runs the host system's C compiler, automatically supplying flags to locate the PLT Scheme 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 PLT Scheme libraries and to generate a shared library.
- `-xform` : Transforms C code that is written without explicit GC-cooperation hooks to cooperate with PLT Scheme's 3m garbage collector; see §1 "Overview" in ***Inside**: PLT Scheme C API*.

Compilation and linking build on the `dynext/compile` and `dynext/link` libraries. The following `mzc` 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 PLT Scheme. The `-cgc` flag specifies that the extension is to be used with the CGC. The default depends on `mzc`: `-3m` if `mzc` itself is running in 3m, `-cgc` if `mzc` itself is running in CGC.

5.1 Scheme 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 PLT Scheme's 3m garbage collector; see §1 "Overview" in ***Inside**: PLT Scheme 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`.

6 Embedding Scheme Modules via C

The `-c-mods` mode for `mzc` takes a set of Scheme modules and generates a C source file that can be used as part of program that embeds the PLT Scheme run-time system. See §1.3 "Embedding MzScheme into a Program" in ***Inside**: PLT Scheme 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 `mzc -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.

7 Decompiling Bytecode

The `-decompile` mode for `mzc` takes a bytecode file (which usually has the file extension `".zo"`) and converts it back to an approximation of Scheme 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.

  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.

- 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 `scheme/unsafe/ops` are always inlined, so `#%in` is not shown for them.

- 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`.

- 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 Scheme 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 Scheme 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 PLT Scheme 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)
        #: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)
         #: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)
         #:transparent)
  name : symbol?
```

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

```
(struct module-variable (modidx sym pos phase)
         #: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)
         #:transparent)
  encoded : wrapped?
```

Wraps a syntax object in a `prefix`.

7.2.2 Forms

```
(struct 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)
         #: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)
         #: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)
         #: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)
         #: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)
         #: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)
         #: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
                    self-modidx
                    prefix
                    provides
                    requires
                    body
                    syntax-body
                    unexported
                    max-let-depth
                    dummy
                    lang-info
                    internal-context)
         #:transparent)
```

```
name : 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 syntax?)
```

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)
        #: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) ()
        #: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
                    max-let-depth
                    body)
        #: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))
  rest? : boolean?
```

```
  closure-map : (vectorof exact-nonnegative-integer?)
  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 or `'ref` for a boxed argument (representing a mutable local variable). The `closure-map` field is a vector of stack positions that are captured when evaluating the `lambda` form to create a closure.

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)
         #: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)
         #:mutable
         #:prefab)
  v : closure?
```

An indirection used in expression positions to form cycles.

```
(struct (case-lam expr) (name clauses)
         #: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)
         #:transparent)
  rhs : (or/c expr? seq? indirect? any/c)
  body : (or/c expr? seq? indirect? any/c)
```

Pushes an uninitialized slot onto the stack, evaluates `rhs` and puts its value into the slot, and then runs `body`.

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)
         #: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)
         #: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)
         #: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)
        #: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?)
        #:transparent)
  unbox? : boolean?
  pos : exact-nonnegative-integer?
  clear? : boolean?
  other-clears? : 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.

```
(struct (toplevel expr) (depth pos const? ready?)
        #: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)
        #: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)
        #: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)
        #: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)
        #: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)
        #: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)
        #:transparent)
  toplevel : toplevel?
```

Represents a `#%variable-reference` form.

```
(struct (assign expr) (id rhs undef-ok?)
        #: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)
        #: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)
        #: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)
        #:transparent)
  datum : any/c
```

```
  wraps : (listof wrap?)
  certs : list?
```

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 ()
        #:transparent)
```

A supertype for lexical-information elements.

```
(struct (lexical-rename wrap) (alist)
        #:transparent)
  alist : (listof (cons/c identifier? identifier?))
```

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

```
(struct (phase-shift wrap) (amt src dest)
        #: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?)
        #: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)
        #:transparent)
  path : module-path-index?
  phase : (or/c exact-integer? #f)
  src-phase : (or/c exact-integer? #f)
  exceptions : (listof symbol?)
  prefix : symbol?
```

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

```
(struct module-binding (path
                        mod-phase
                        import-phase
                        id
                        nominal-path
                        nominal-phase
                        nominal-id)
        #:transparent)
  path : module-path-index?
  mod-phase : (or/c exact-integer? #f)
  import-phase : (or/c exact-integer? #f)
  id : symbol?
  nominal-path : module-path-index?
  nominal-phase : (or/c exact-integer? #f)
  nominal-id : (or/c exact-integer? #f)
```

Represents a single identifier import (i.e., the general case) within a `module-rename`.

7.3 Scheme API for Marshaling Bytecode

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

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

Consumes a representation of bytecode and generates a byte string for the marshaled bytecode. Currently, syntax objects are not supported, including in `req` for a top-level `#%require`.

8 Compiling to Raw Bytecode

The `-zo`/`-z` mode for `mzc` is an improverished form of the default `-make`/`-k` mode (which is described in §2 “Compiling Modified Modules to Bytecode”), 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 `mzc -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 scheme/class)
(define f (class% object% (super-new)))
```

the `class` form from the `scheme/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, `mzc` 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 `mzc` 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 `mzc` in its default mode.

9 Compiling to Native Code via C

The `-extension`/`-e` mode for `mzc` is similar to the `-zo` mode, 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 PLT Scheme provides better performance with lower overhead on the platforms where it is supported (see §17 "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 `mzc` 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 §5 "Compiling and Linking C Extensions". In addition, a few flag provide some control over the Scheme-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.

10 Scheme API for 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.

10.1 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 ".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.ss" file (see §5 "\"info.ss\" 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 ".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.ss" fields, instead of using an "info.ss" file (if any) in the directory.

10.2 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.

10.3 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.

10.4 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 opera-

tions. 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`.

10.5 The Compiler as a Unit

10.5.1 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.

10.5.2 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^.

10.5.3 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^.

10.5.4 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^.

Index