PLT **mzc**: MzScheme Compiler Manual

PLT (scheme@plt-scheme.org)

Copyright notice

Contents

1.   About **mzc**

1.1   **mzc** Is...

The **mzc** compiler takes MzScheme (or MrEd) source code and produces either platform-independent byte-code compiled files (**.zo** files) or platform-specific native-code libraries (**.so** or **.dll** files) to be loaded into MzScheme (or MrEd). In the latter mode, **mzc** provides limited suport for interfacing directly to C libraries.

mzc works on either individual files or on collections. (A *collection* is a group of files that conform to MzScheme's library collection system; see §16 in *PLT MzScheme: Language Manual*). In general, **mzc** works best with code using the **module** form.

As a convenience for programmers writing low-level MzScheme extensions, **mzc** can compile and link plain C files that use MzScheme's **escheme.h** header. This facility is described in *Inside PLT MzScheme*.

Finally, **mzc** can perform miscellaneous tasks, such as embedding Scheme code in a copy of the MzScheme (or MrEd) binary to produce a stand-alone executable, or creating **.plt** distribution archives.

1.1.1   Byte-Code Compilation

A byte-code file typically uses the file extension **.zo**. The file starts with #∼ followed by the byte-code data.

Byte-code files are loaded into MzScheme in the same way as regular Scheme source files (e.g., with `load`). The #∼ marker causes MzScheme's reader to load byte codes instead of normal Scheme expressions. When a **.zo** file exists in a **compiled** subdirectory, it is sometimes loaded in place of a source file; see §3.3 for details.

Byte-code programs produced by **mzc** run exactly the same as source code compiled by MzScheme directly (assuming the same set of bindings are in place at compile time and load time). In other words, byte-code compilation does not optimize the code any more than MzScheme's normal evaluator. However, a byte-code file can be loaded into MzScheme much faster than a source-code file.

1.1.2   Native-Code Compilation

A native-code file is a platform-specific shared library. Under Windows, native-code files typically use the extension **.dll**. Under Unix and MacOS, native-code files typically use the extension **.so**.

Native-code files are loaded into MzScheme with the `load-extension` procedure (see §14.7 in *PLT MzScheme: Language Manual*). When a native-code file exists in a **compiled** subdirectory, it is sometimes loaded in place of a source file; see §3.3 for details.

The native-code compiler attempts to optimize a source program so that it runs faster than the source-code or byte-code version of the program. See §1.4 for information on obtaining the best possible performance from **mzc**-compiled programs.

The **cffi.ss** library of the **compiler** collection defines Scheme forms, such as **c-lambda**, for accessing C

functions from Scheme. The forms produce run-time errors when interpreted directly or compiled to byte code. See §2 for further information.

Native-code compilation produces C source code in an intermediate stage; your system must provide an external C compiler to produce native code. The **mzc** compiler cannot produce native code directly from Scheme code.

- Under Unix, **gcc** is used as the C compiler if it can be found in any of the directories listed in the PATH environment variable. If **gcc** is not found, **cc** is used if it can be found.

- Under Windows, **cl.exe**, Microsoft Visual C, is used as the C compiler if it can be found in any of the directories listed in the PATH environment variable. If **cl.exe** is not found, then **gcc.exe** is used if it can be found. If neither **cl.exe** nor **gcc.exe** is found, then **bcc32.exe** (Borland) is used if it can be found.

- Under MacOS, Metrowerks CodeWarrior is used as the C compiler if it can be found.

Except for MacOS, the C compiler and compiler flags used by **mzc** can be adjusted via command line flags.

1.2   **mzc** Is Not...

mzc does not generally produce stand-alone executables from Scheme source code. The compiler's output is intended to be loaded into MzScheme (or MrEd or DrScheme). However, see also §5 for information about embedding code into a copy of the MzScheme (or MrEd) executable.

mzc does not translate Scheme code into similar C code. Native-code compilation produces C code that relies on MzScheme to provide run-time support, which includes memory management, closure creation, procedure application, and primitive operations.

1.3   Running **mzc**

Under Unix and Windows, run **mzc** from a shell, passing in flags and arguments on the command line.

Under MacOS, double-click on the **mzc** launcher application with the Command key pressed, then provide arguments in the command line dialog that appears. (Close the **MzScheme** application first if it is already running, since **mzc** is itself a MzScheme-based application.) If the Command key is not pressed while **mzc** is started, the command-line dialog will not appear. If a file is dragged onto the **mzc** icon, then the command-line will contain the file's path; this is useful for compiling a Scheme file directly to an extension. If a file is dragged onto the **mzc** icon, additional command-line argument can be provided by holding down the Command key, but the arguments will go after the file name, which is almost never useful (since the order of command-line arguments is important).

In this manual, each example command line is shown as follows:

 `mzc --extension --prefix` **macros.ss file.ss**

To run this example under Unix or Windows, type the command line into a shell (replacing **mzc** with the path to **mzc** on your system, if necessary). Under MacOS, launch **mzc** with the Command key pressed, and enter everything *after* **mzc** into the dialog that appears.

Simple on-line help is available for **mzc**'s command-line arguments by running **mzc** with the `-h` or `--help` flag.

1.4 Native Code Optimization from **mzc**

Compiling a program to native code with **mzc** can provide significant speedups compared to interpreting byte code (or running the program directly from source code), but only for certain kinds of programs. The speedup from native-code compilation is typically due to two optimizations:

- **Loop Optimization** — When **mzc** statically detects a tail-recursive loop, it compiles the Scheme loop to a C loop that has no interpreter overhead. For example, given the program

$$
\begin{aligned}
(\textbf{letrec } ([odd \ (\textbf{lambda } (x) \\
(\textbf{if } (\texttt{zero? } x) \\
\texttt{\#f} \\
(even \ (\texttt{sub1 } x)))))] \\
[even \ (\textbf{lambda } (x) \\
(\textbf{if } (\texttt{zero? } x) \\
\texttt{\#t} \\
(odd \ (\texttt{sub1 } x)))))]) \\
(odd \ \texttt{40000}))
\end{aligned}
$$

  mzc can detect the *odd–even* loop and produce native code that runs twice as fast as byte-code interpretation. In contrast, given a similar program using top-level definitions,

$$
\begin{aligned}
(\textbf{define } (odd \ x) \ldots) \\
(\textbf{define } (even \ x) \ldots)
\end{aligned}
$$

  the compiler cannot assume an *odd–even* loop, because the global variables *odd* and *even* can be redefined at any time. Note that **define**d variables in a **module** expression are lexically scoped like **letrec** variables, and **module** definitions therefore permit loop optimizations.[1]

- **Primitive Inlining** — When **mzc** encounters the application of certain primitives, it inlines the primitive procedure. However, the compiler must be certain that a variable reference will resolve to a primitive procedure when the code is loaded into MzScheme. In the preceding example, the compiler cannot inline the application of `sub1` because the global variable `sub1` might be redefined. To encourage the inlining of primitives—which produces native code that runs *30 times faster* than byte-code interpretation for the preceding example—the programmer has three options:
  - **Use module** — If the original example is encapsulated in a module that imports `mzscheme`, then each primitive name, such as `sub1`, is guranteed to access the primitive procedure (assuming that the name is not lexically bound). The "modulized" version of the preceding program follows:

$$
\begin{aligned}
(\textbf{module } oe \ \texttt{mzscheme} \\
(\textbf{letrec } ([odd \ (\textbf{lambda } (x) \\
(\textbf{if } (\texttt{zero? } x) \\
\texttt{\#f} \\
(even \ (\texttt{sub1 } x)))))] \\
[even \ (\textbf{lambda } (x) \\
(\textbf{if } (\texttt{zero? } x) \\
\texttt{\#t} \\
(odd \ (\texttt{sub1 } x)))))]) \\
(odd \ \texttt{40000})))
\end{aligned}
$$

  To run this program, the *oe* module must be **require**d at the top level.

---

[1]The compiler cannot always prove that **module** definitions have been evaluated before the corresponding variable is used in an expression. Use the `-v` or `--verbose` flag to check whether **mzc** reports a "last known module binding" warning when compiling a **module** expression, which indicates that definitions after a particular line in the source file might be referenced before they are defined.

- **Use a (require mzscheme) prefix** — If the preceding example is prefixed with **(require mzscheme)**, then sub1 refers not to the global variable, but to the sub1 export of the mzscheme module. See §3.2 for more information about prefixing compilation.

- **Use the --prim flag** — The --prim flag alters the semantics of the langugage for compilation such that every reference to a global variable that is built into MzScheme is converted to its keyword form. Actually, specifying the --prim flag causes **mzc** to automatically prefix the program with **(require mzscheme)**.

Programs that permit these optimizations also to encourage a host of other optimizations, such as procedure inlining (for programmer-defined procedures) and static closure detection. In general, **module**-based programs provide the most opportunities for optimization.

Native-code compilation rarely produces significant speedup for programs that are not loop-intensive, programs that are heavily object-oriented, programs that are allocation-intensive, or programs that exploit built-in procedures (e.g., list operations, regular expression matching, or file manipulations) to perform most of the program's work.

2. Foreign-Function Interface to C

The **cffi.ss** library of the **compiler** collection implements a subset of Gambit-C's foreign-function interface (see Marc Feeley's *Gambit-C, version 3.0*). The **cffi.ss** module defines two forms: **c-lambda** and **c-declare**. When interpreted directly or compiled to byte code, **c-lambda** produces a function that always raises `exn:user`, and **c-declare** raises `exn:user`. When compiled by **mzc**, the forms provide access to C. The **mzc** compiler implicitly imports **cffi.ss** into the top-level environment.

The **c-lambda** form creates a Scheme procedure whose body is implemented in C. Instead of declaring argument names, a **c-lambda** form declares argument types, as well as a return type. The implementation can be simply the name of a C function, as in the following definition of `fmod`:

(**define** *fmod* (**c-lambda** (**double double**) **double** "fmod"))

Alternatively, the implementation can be C code to serve as the body of a function, where the arguments are bound to `___arg1` (three underscores), etc., and the result is installed into `___result` (three underscores):

(**define** *machine-string->float*
  (**c-lambda** (**char-string**) **float**
      "___result = *(float *)___arg1;"))

The **c-lambda** form provides only limited conversions between C and Scheme data. For example, the following function does not reliably produce a string of four characters:

(**define** *broken-machine-float->string*
  (**c-lambda** (**float**) **char-string**
      "char b[5]; *(float *)b = ___arg1; b[4] = 0; ___result = b;"))

because the representation of a `float` can contain null bytes, which terminate the string. However, the full MzScheme API, which is described in *Inside PLT MzScheme*, can be used in a function body:

(**define** *machine-float->string*
  (**c-lambda** (**float**) **scheme-object**
      "char b[4]; *(float *)b = ___arg1; ___result = scheme_make_sized_string(b, 4, 1);"))

The **c-declare** form declares arbitrary C code to appear after **escheme.h** or **scheme.h** is included, but before any other code in the compilation environment of the declaration. It is often used to declare C header file inclusions. For example, a proper definition of `fmod` needs the **math.h** header file:

(**c-declare** "#include <math.h>")
(**define** *fmod* (**c-lambda** (**double double**) **double** "fmod"))

The **c-declare** form can also be used to define helper C functions to be called through **c-lambda**.

The **plt/collects/mzscheme/examples** directory in the PLT distribution contains additional examples.

The **c-lambda** and **c-declare** forms are defined as follows:

- (**c-lambda** (*argument-type* ⋯) *result-type funcname-or-body-string*) creates a Scheme procedure whose body is implemented in C. The procedure takes as many arguments as the supplied *argument-type*s, and it returns one value. If *return-type* is **void**, the procedure's result is always void. The *funcname-or-body-string* is either the name of a C function (or macro) or the body of a C function.

  If *funcname-or-body-string* is a string containing only alphanumeric characters and _, then the created Scheme procedure passes all of its arguments to the named C function (or macro) and returns the function's result. Each argument to the Scheme procedure is converted according to the corresponding *argument-type* (as described below) to produce an argument to the C function. Unless *return-type* is **void**, the C function's result is converted according to *return-type* for the Scheme procedure's result.

  If *funcname-or-body-string* contains more than alphanumeric characters and _, then it must contain C code to implement the function body. The converted arguments for the function will be in variables `___arg1`, `___arg2`, ... (with three underscores in each name) in the context where the *funcname-or-body-string* is placed for compilation. Unless *return-type* is **void**, the *funcname-or-body-string* code should should assign a result to the variable `___result` (three underscores), which will be declared but not initialized. The *funcname-or-body-string* code should not return explicitly; control should always reach the end of the body. If the *funcname-or-body-string* code defines the pre-processor macro `___AT_END` (with three leading underscores), then the macro's value should be C code to execute after the value `___result` is converted to a Scheme result, but before the result is returned, all in the same block; defining `___AT_END` is primarily useful for deallocating a string in `___result` that has been copied by conversion. The *funcname-or-body-string* code will start on a new line at the beginning of a block in its compilation context, and `___AT_END` will be undefined after the code.

  Each *argument-type* must be one of the following:
  - **bool**
    Scheme range: any value
    C type: `int`
    Scheme to C conversion: #f ⇒ 0, anything else ⇒ 1
    C to Scheme conversion: 0 ⇒ #f, anything else ⇒ #t
  - **char**
    Scheme range: character
    C type: `char`
    Scheme to C conversion: character's ASCII value cast to signed byte
    C to Scheme conversion: ASCII value from unsigned cast mapped to character
  - **unsigned-char**
    Scheme range: character
    C type: `unsigned char`
    Scheme to C conversion: character's ASCII value
    C to Scheme conversion: ASCII value mapped to character
  - **signed-char**
    Scheme range: character
    C type: `signed char`
    Scheme to C conversion: character's ASCII value cast to signed byte
    C to Scheme conversion: ASCII value from unsigned cast mapped to character
  - **int**
    Scheme range: exact integer that fits into an `int`
    C type: `int`
    conversions: (obvious and precise)
  - **unsigned-int**
    Scheme range: exact integer that fits into an `unsigned int`
    C type: `unsigned int`
    conversions: (obvious and precise)
  - **long**
    Scheme range: exact integer that fits into a `long`

C type: `long`
conversions: (obvious and precise)

– **unsigned-long**
Scheme range: exact integer that fits into an `unsigned long`
C type: `unsigned long`
conversions: (obvious and precise)

– **short**
Scheme range: exact integer that fits into a `short`
C type: `short`
conversions: (obvious and precise)

– **unsigned-short**
Scheme range: exact integer that fits into an `unsigned short`
C type: `unsigned short`
conversions: (obvious and precise)

– **float**
Scheme range: real number
C type: `float`
Scheme to C conversion: number converted to inexact and cast to `float`
C to Scheme conversion: cast to `double` and encapsulated as an inexact number

– **double**
Scheme range: real number
C type: `double`
Scheme to C conversion: number converted to inexact
C to Scheme conversion: encapsulated as an inexact number

– **char-string**
Scheme range: string or #f
C type: `char*`
Scheme to C conversion: string ⇒ contained character array pointer, #f ⇒ `NULL`
C to Scheme conversion: `NULL` ⇒ #f, anything else ⇒ new string created by copying the string

– **nonnull-char-string**
Scheme range: string
C type: `char*`
Scheme to C conversion: string's contained character array pointer
C to Scheme conversion: new string created by copying the string

– **scheme-object**
Scheme range: any value
C type: `Scheme_Object*`
Scheme to C conversion: no conversion
C to Scheme conversion: no conversion

– **(pointer** *string*)
Scheme range: an opaque c-pointer value identified as type *string* or #f
C type: *string*`*`
Scheme to C conversion: #f ⇒ `NULL`, c-pointer ⇒ contained pointer cast to *string*`*`
C to Scheme conversion: `NULL` ⇒ #f, anything else ⇒ new c-pointer containing the pointer and identified as type *string*

The *return-type* must be **void** or one of the *arg-type* keywords.

- (**c-declare** *code-string*) declares arbitrary C code to appear after **escheme.h** or **scheme.h** is included, but before any other code in the compilation environment of the declaration. A **c-declare** form can appear only at the top-level or within a module's top-level sequence.

  The *code-string* code will appear on a new line in the file for C compilation. Multiple **c-include** declarations are concatenated (with newlines) in order to produces a sequence of declarations.

3.   Compiling Individual Files with **mzc**

To compile an individual file with **mzc**, provide the file name as the command line argument to **mzc**. To compile to byte code, use the `-z` or `--zo` flag; to compile to native code, use the `-e` or `--extension` flag. If no compilation mode flag is specified, `--extension` is assumed.

The input file must have a file extension that designates it as a Scheme file, either **.ss** or **.scm**. The output file will have the same base name and same directory (by default) as the input file, but with an extension appropriate to the type of the output file (either **.zo**, **.dll**, or **.so**).

Example:

```
 mzc --extension file.ss
```

Under Windows, the above command reads **file.ss** from the current directory and produces **file.dll** in the current directory.

Multiple Scheme files can be specified for compilation at once. A separate compiled file is produced for each Scheme file. By default, each compiled file is placed in the directory containing the corresponding input file. When multiple files are compiled at once, macros defined in a file are visible in the files that are compiled afterwards.

3.1   Compiling with Modules

In terms of both optimization and proper loading of syntax definitions, **mzc** works best with programs that are encapsulated within per-file **module** expressions. Using a single **module** expression in a file eliminates the code's dependence on the top-level environment. Consequently, all dependencies of the code on loadable syntax extensions are evident to the compiler.

When compiling a module that **require**s another module (that is not built into MzScheme), **mzc** loads the required module, but does not invoke it. Instead, **mzc** uses the loaded module only for its syntax exports, if any (which means that **mzc** executes the transformer code in the module, but not any of its normal code).

3.2   Compilation without Modules

Top-level **define-syntax(es)**, **module**, **require**, **require-for-syntax**, and **begin** expressions are handled specially by **mzc**: the compile-time portion of the expression is evaluated, because it might affect later expressions.[1] For example, when compiling the file containing

    (**require** (**lib** "etc.ss"))
    (**define** f (**opt-lambda** (a [b 7]) (+ a b)))

the **opt-lambda** syntax from the "etc.ss" library must be bound in the compilation namespace at compile

---

[1]The `-m` or `--module` flag turns off this special handling.

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-syntax**. For example, **define-signature** expands to a **define-syntax** definition. **mzc** detects **define-syntax** 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`).

mzc's `-p` or `--prefix` flag 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 compiled code into **module** expressions, as explained in §3.1.

Note that MzScheme provides no **eval-when** form for controlling the evaluation of compiled code, because **module** provides a simpler and more consistent interface for separating compile-time and run-time code.

3.3   Autodetecting Compiled Files for Loading

When MzScheme's `load/use-compiled`, `load-relative`, or **require** is used to load a file, MzScheme automatically detects an alternate byte-code and/or native-code file that resides near the requested file. Byte-code files are found in a **compiled** subdirectory in the directory of the requested file. Native-code files are found in (`build-path` *dir* `"compiled" "native"` (`system-library-subpath`)) where *dir* is the directory of the requested file. A byte-code or native-code file is used in place of the requested file only if its modification date is later than the requested file, or if the requested file does not exist. If both byte-code and native-code files can be loaded, the native-code file is loaded.

Example:

    mzc `--extension` `--destination` **compiled/native/i386-linux file.ss**

Under Linux, the above command compiles **file.ss** in the current directory and produces **compiled/native/i386-linux/file.so**. Evaluating (`load/use-compiled "file.ss"`) in MzScheme will then load **compiled/native/i386-linux/file.so** instead of **file.ss**. If **file.ss** is changed without recreating **file.so**, then `load/use-compiled` loads **file.ss**, because **file.so** is out-of-date.

3.4   Compiling Multiple Files to a Single Native-Code Library

When the `-o` or `--object` flag is provided to **mzc**, **.kp** and **.o**/**.obj** files are produced instead of a loadable library. The **.o**/**.obj** files contain the native code for a single source file. The **.kp** files contain information used for global optimizations.

Multiple **.kp** and **.o**/**.obj** files are linked into a single library using **mzc** with the `-l` or `--link-extension` flag. All of the **.kp** and **.o**/**.obj** files to be linked together are provided on the command line to **mzc**. The output library is always named **_loader.so** or **_loader.dll**.

Example:

```
mzc --object file1.ss
mzc --object file2.ss
mzc --link-extension file1.kp file1.o file2.kp file2.o
```

Under Unix, the above commands produce a **＿loader.so** library that encapsulates both **file1.ss** and **file2.ss**.

Loading **＿loader** into MzScheme is not quite the same as loading all of the Source files that are encapsulated by **＿loader**. The return value from (`load-extension "＿loader.so"`) is a procedure that takes a symbol or #t. If a symbol is provided and it is the same as the base name of a source file (i.e., the name without a path or file extension) encapsulated by **＿loader**, then a thunk is returned, along with a symbol (or #f) indicating a module name declared by the file. Applying the thunk has the same effect as loading the corresponding source file. If a symbol is not recognized by the **＿loader** procedure, #f is returned instead of a thunk. If #t is provided, a thunk is returned that "loads" all of the files (using the order of the **.o/.obj** files provided to **mzc**) and returns the result from loading the last one.

The **＿loader** procedure can be called any number of times to obtain thunks, and each thunk can be applied any number of times (where each application has the same effect as loading the source file again). Evaluating (`load-extension "＿loader.so"`) multiple times returns an equivalent loader procedure each time.

Given the **＿loader.so** constructed by the example commands above, the following Scheme expressions have the same effect as loading **file1.ss** and **file2.ss**:

    (**let-values** ([(*go modname*) ((`load-extension "＿loader.so"`) 'file1)]) (*go*))
    (**let-values** ([(*go modname*) ((`load-extension "＿loader.so"`) 'file2)]) (*go*))

or, equivalently:

    (**let-values** ([(*go modname*) ((`load-extension "＿loader.so"`) #t)]) (*go*))

The special **＿loader** convention is recognized by MzScheme's `load/use-compiled`, `load-relative`, and **require**. MzScheme automatically detects **＿loader.so** or **＿loader.dll** in the same directory as individual native-code files (see §3.3). If both an individual native-code file and a **＿loader** are available, the **＿loader** file is used.

4. Compiling Collections with **mzc**

A *collection* is a group of files that conform to MzScheme's library collection system; see §16 in *PLT MzScheme: Language Manual* for details. Every source file in a collection should contain a single **module** declaration.

The `--collection-zo` and `--collection-extension` flags direct **mzc** to compile a whole collection. The `--collection-zo` flag produces individual **.zo** files for each library in the collection. The `--collection-extension` flag produces a single **_loader** library for the collection.

The (sub-)collection to compile is specified on the command line for **mzc**. The specified collection must contain an **info.ss** library that provides information about how to compile the collection. (See §7 for information on the format of **info.ss**.)

To compile a collection, **mzc** extracts **info.ss** information for the following fields:

- `name` — the name of the collection as a string.

- `compile-omit-files` — a list of library filenames (without paths); all Scheme files in the collection are compiled except for the files in this list. This information is optional.

- `compile-zo-omit-files` — a list of library filenames that should not be compiled to byte code (but possibly to native code). This information is optional.

- `compile-extension-omit-files` — a list of library filenames that should not be compiled to native code (but possibly to byte code). This information is optional.

- `compile-subcollections` — a list of sub-collection sub-paths, where each sub-path is a list of strings; each full sub-collection path is formed by appending the sub-path to the path of the collection being compiled. Each sub-collection is compiled in the same way as the current collection, using the **info.ss** library of the sub-collection. This information is optional.

When compiling a collection to byte-code files, **mzc** automatically creates a **compiled** directory in the collection directory and puts **.zo** files there.

When compiling a collection to native code, **mzc** automatically created a **compiled** directory in the collection directory, a **native** directory in that **compiled** directory, and a platform-specific directory in **native** using the directory name returned by `system-library-subpath`. Intermediate **.c** and **.kp** files are kept in **native**. The platform-specific directory gets intermediate **.o**/**.obj** files and the final **_loader.so** or **_loader.dll**.

To compile a collection, **mzc** compiles only the library files that have changed since the last compilation. This form of dependency checking is usually too weak. For example, when a signature file changes, **mzc** does not automatically recompile all files that rely on the signatures. In this case, delete the **compiled** directory when a macro or signature file changes to ensure that the collection is compiled correctly.

5. Building a Stand-alone Executable

Since the output of **mzc** relies on MzScheme to provide all run-time support, there is no way to use **mzc** to obtain *small* stand-alone executables. However, it is possible to produce a *large* stand-alone executable that contains an embedded copy of the MzScheme (or MrEd) run-time engine.

5.1 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. The created executable invokes the embedded module on startup. The `--gui-exe` flag is similar, but copies the MrEd executable.

If the embedded module refers statically (i.e., through **require**) to modules in MzLib or other collections, then those modules are also included in the embedding executable.

Library modules that are referenced dynamically—through `eval`, `load`, or `dynamic-require`—are not automatically embedded into the created executable, but they can be explicitly included using **mzc**'s `--lib` flag.

The `--exe` and `--gui-exe` flags work only with **module**-based programs. The **embed.ss** library in the **compiler** collection provides a more general interface to the embedding mechanism.

5.2 Stand-Alone Executables from Native Code

Creating a stand-alone executable that embeds native code from **mzc** requires downloading the MzScheme source code and using a C compiler and linker directly.

To build an executable with an embedded MzScheme engine:

- Download the source code from `http://www.drscheme.org/` and compile MzScheme.

- Recompile MzScheme's **main.c** with the preprocessor symbol `STANDALONE_WITH_EMBEDDED_EXTENSION` defined. Under Unix, the **Makefile** distributed with MzScheme provides a target **ee-main** that performs this step.

  The preprocessor symbol causes MzScheme's startup code to skip command line parsing, the user's initialization file, and the `read-eval-print` loop. Instead, the C function `scheme_initialize` is called, which is the entry point into **mzc**-compiled Scheme code. After compiling **main.c** with `STANDALONE_WITH_EMBEDDED_EXTENSION` defined, MzScheme will not link by itself; it must be linked with objects produced by **mzc**.

- Compile each Scheme source file in the program with **mzc**'s `-o` or `--object` flag and the `--embedded` flag, producing a set of **.kp** files and object (**.o** or **.obj**) files.

- After each Scheme file is compiled, run **mzc** with the `-g` or `--link-glue` and the `--embedded` flag, providing all of the **.kp** files and object files on the command line. (Put the object files in the order

that they should be "loaded.") The `-g` or `--link-glue` step produces a new object file, **_loader.o** or **_loader.obj**.

Each of the Scheme source files in the program must have a different base name (i.e., the file name without its directory path or extension), otherwise **_loader** cannot distinguish them. The files need not reside in the same directory.

- Link all of the **mzc**-created object files with the MzScheme implementation (having compiled **main.c** with `STANDALONE_WITH_EMBEDDED_EXTENSION` defined) to produce a stand-alone executable.

  Under Unix, the **Makefile** distributed with MzScheme provides a target **ee-app** that performs the final linking step. To use the target, call **mzmake** with a definition for the makefile macro **EEAPP** to the output file name, and a definition for the makefile macro **EEOBJECTS** to to the list of **mzc**-created object files. (The example below demonstrates how to define makefile variables on the command line.)

For example, under Unix, to create a standalone executable **MyApp** that is equivalent to

```
mzscheme -mv -f file1.ss -f file2.ss
```

unpack the MzScheme source code and perform the following steps:

```
cd plt/src/mzscheme
./mzmake
./mzmake ee-main
mzc --object --embedded file1.ss
mzc --object --embedded file2.ss
mzc --link-glue --embedded file1.kp file1.o file2.kp file2.o
./mzmake EEAPP=MyApp EEOBJECTS="file1.o file2.o _loader.o" ee-app
```

To produce an executable that embeds the MrEd engine, the procedure is essentially the same; MrEd's main file is **mrmain.cxx** instead of **main.c**. See the compilation notes in the MrEd source code distribution for more information.

6.   Creating Distribution Archives

The command-line flags `--plt` and `--collection-plt` direct **mzc** to create an archive for distributing files to PLT users. A distribution archive usually has the suffix **.plt**, which Help Desk and DrScheme recognize as archives 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 a flag indicating whether they are always to be unpacked relative to the PLT installation 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`. In the latter case, the files are unpacked relative to the PLT installation directory.

- 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 Help Desk or 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 PLT installation directory, instead.

Use the `--collection-plt` flag to pack one or more collections; sub-collections can be designated by using a forward slash ("/") as a path separator on all platforms. In this mode, **mzc** automatically uses paths relative to the PLT installation 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 §7):

14

- `requires` — a list of the form (`list` (`list` *coll-path vers*) ⋯) where each *coll-path* 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 defalt 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-path* ⋯) where each *coll-path* is a non-empty list of relative-path strings. The inidicated collections must *not* be installed at unpacking time.

For example, the **info.ss** file in the **sirmail** collection might contain the following `info` declaration:

```
(module info (lib "infotab.ss" "setup")
  (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.

Although **mzc**'s command-line interface is sufficient for most purposes, the **pack.ss** library of the **setup** collection provides a general interface for constructing archives.

7.   **info.ss** File Format

An **info.ss** file provides general information about a collection. The file must have the following format:

```
(module info (lib "infotab.ss" "setup")
    (define identifier info-expr)
    ···)
```

info-expr is one of
```
    (quote datum)
    (quasiquote datum) ; with unquote and unquote-splicing
    (info-primitive info-expr ···)
    identifier ; an identifier defined in the info module
    literal ; a string, number, boolean, etc.
```

info-primitive is one of
```
    cons car cdr list
    list* reverse append
    build-path collection-path
    system-library-subpath
```

For example, the following declaration is in the **info.ss** library of the **help** collection. It contains definitions for three info tags:

```
(module info (lib "infotab.ss" "setup")
    (define name "Help")
    (define mred-launcher-libraries (list "help.ss"))
    (define mred-launcher-names (list "Help Desk")))
```

The **setup** collection's **getinfo.ss** library defines a `get-info` function for extracting field values from a collection's **info.ss** file. See the **setup** collection's documentation for details.

Index