

PLT **mzc**: MzScheme Compiler Manual

PLT (scheme@plt-scheme.org)

371

Released August 2007

Copyright notice

Copyright ©1996-2007 PLT

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Library General Public License, Version 2 published by the Free Software Foundation. A copy of the license is included in the appendix entitled "License."

Contents

1	About mzc	1
1.1	mzc Is...	1
1.1.1	Byte-Code Compilation	1
1.1.2	Native-Code Compilation	1
1.2	mzc Is Not...	2
1.3	Running mzc	2
1.4	Native Code Optimization	2
2	Foreign-Function Interface to C	5
3	Compiling Individual Files with mzc	9
3.1	Compiling with Modules	9
3.2	Compilation without Modules	9
3.3	Autodetecting Compiled Files for Loading	10
3.4	Compiling Multiple Files to a Single Native-Code Library	10
4	Compiling Collections with mzc	12
5	Building and Distributing Stand-alone Executables	13
5.1	Stand-Alone Executables from Scheme Code	13
5.2	Distributing Stand-Alone Executables	14
5.3	Stand-Alone Executables from Native Code	14
6	Creating Library Distribution Archives	16
7	info.ss File Format	18
	License	19

Index

23

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, which are just-in-time compiled to native code when loaded on x86, x86_64, and PowerPC platforms) or platform-specific native-code libraries (**.so** or **.dll** files) to be loaded into MzScheme (or MrEd). In the latter mode, **mzc** provides limited support 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 § 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.

Whether loading from source or byte code, MzScheme compiles as needed to native code on x86, x86_64, and PowerPC platforms. Setting the environment variable **PLTNOMZJIT** disables just-in-time compilation on all platforms. (In addition, the stand-alone MzScheme executable also accepts a `-j` or `--no-jit` flag to disable just-in-time compilation.) See §1.4 for information on obtaining the best possible performance.

1.1.2 Native-Code Compilation

A native-code file is a platform-specific shared library. Under Windows, native-code files use the extension **.dll**. Under Mac OS X, native-code files use the extension **.dylib**. Under Unix, native-code files use the extension **.so**.

Native-code files are loaded into MzScheme with the `load-extension` procedure (see § 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 ahead-of-time compiler uses C as an intermediate language, instead of byte code, and it works on all platforms (when a C compiler is available). The ahead-of-time native compiler can sometimes produce better performance than the just-in-time compiler (where available), but the difference is small compared to the difference between direct byte-code interpretation and just-in-time compilation. See §1.4 for information on obtaining the best possible performance from *mzc*-compiled programs.

The `ffi.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.

Since native-code compilation produces C source code in an intermediate stage, your system must provide an external C compiler for ahead-of-time native code.

- Under Unix and Mac OS X, `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.
- In either case, if the `MZSCHEME_DYNEXT_COMPILER` or `CC` environment variable is defined, it overrides the above search paths (and `MZSCHEME_DYNEXT_COMPILER` takes precedence over `CC`).

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*

Run *mzc* from a shell, passing flags and arguments on the command line.

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

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

To run this example, type the command line into a shell (replacing *mzc* with the path to *mzc* on your system, if necessary).

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

Native code compilation (either just-in-time or ahead-of-time) can provide significant speedups compared to interpreting byte code or running directly from source code (when just-in-time compilation is unavailable or disabled).

Significant speedup from native-code compilation is typically due to two optimizations:

- **Direct function calls** — When the compiler detects a function call to an immediately visible function, it generates more efficient code than for a generic call, especially for tail calls. For example, given the program

```
(letrec ([odd (lambda (x)
             (if (zero? x)
                 #f
                 (even (sub1 x))))])
  [even (lambda (x)
          (if (zero? x)
              #t
              (odd (sub1 x))))])
  (odd 400000))
```

the compiler 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,

```
(define (odd x) ...)
(define (even x) ...)
```

the compiler cannot assume an *odd-even* loop, because the global variables *odd* and *even* can be redefined at any time. Within a module, defined variables are lexically scoped like *letrec* variables, and module definitions therefore permit call 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 about 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 guaranteed to access the primitive procedure (assuming that the name is not lexically bound). The “modulized” version of the preceding program follows:

```
(module oe mzscheme
  (letrec ([odd (lambda (x)
                  (if (zero? x)
                      #f
                      (even (sub1 x))))])
    [even (lambda (x)
            (if (zero? x)
                #t
                (odd (sub1 x))))])
      (odd 400000)))
```

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

- **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 for *mzc* effectively prefixes the program with `(require mzscheme)`.

¹The compiler cannot always prove that module definitions have been evaluated before the corresponding variable is used in an expression. With ahead-of-time compilation via *mzc*, 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.

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

MzLib's **foreign.ss** provides an interface to dynamic C libraries that requires no C compiler and works completely at run time. See *PLT Foreign Interface Manual* for more information. The manual *Inside PLT MzScheme*, meanwhile, describes a C-level API for extending MzScheme. This section describes the **ffi.ss** library of the **compiler** collection, which provides a third alternative (in conjunction with **mzc**).

The **ffi.ss** library relies on a C compiler to statically construct an interface to C code through directives embedded in a Scheme program. The library implements a subset of Gambit-C's foreign-function interface (see Marc Feeley's *Gambit-C, version 3.0*).

The **ffi.ss** module defines three forms: `c-lambda`, `c-declare`, and `c-include`. When interpreted directly or compiled to byte code, `c-lambda` produces a function that always raises `exn:fail`, and `c-declare` and `c-include` raise `exn:fail`. When compiled by **mzc**, the forms provide access to C. The **mzc** compiler implicitly imports **ffi.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_byte_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 `c-include` form expands to a `c-declare` form using the content of a specified file. Use `(c-include file)` instead of `(c-declare "#include file")` when it's easier to have MzScheme resolve the file path than to have the C compiler resolve it.

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

When compiling for MzScheme3m (see *Inside PLT MzScheme*), C code inserted by `c-lambda`, `c-declare`, and `c-include` will be transformed in the same way as `mzc`'s `--xform` mode (which may or may not be enough to make the code work correctly in MzScheme3m; see *Inside PLT MzScheme* for more information).

The `c-lambda`, `c-declare`, and `c-include` 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-types*, 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 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.

In addition to `__arg1`, etc., the variable `argc` is bound in *funcname-or-body-string* to the number of arguments supplied to the function, and `argv` is bound to a `Scheme.Object*` array of length `argc` containing the function arguments as Scheme values. The `argv` and `argc` variables are mainly useful for error reporting (e.g., with `scheme_wrong_type`).

Each *argument-type* must be one of the following:

- `bool`
Scheme range: any value
C type: `int`
Scheme to C conversion: `#f` \Rightarrow 0, anything else \Rightarrow 1
C to Scheme conversion: 0 \Rightarrow `#f`, anything else \Rightarrow `#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: byte string or `#f`
 - C type: `char*`
 - Scheme to C conversion: string \Rightarrow contained byte-array pointer, `#f` \Rightarrow `NULL`
 - C to Scheme conversion: `NULL` \Rightarrow `#f`, anything else \Rightarrow new byte string created by copying the string
- `nonnull-char-string`
 - Scheme range: byte string

- C type: `char*`
- Scheme to C conversion: byte string's contained byte-array pointer
- C to Scheme conversion: new byte 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 bytes)`
- Scheme range: an opaque c-pointer value, identified as type `bytes`, or `#f`
- C type: `bytes*`
- Scheme to C conversion: `#f` \Rightarrow `NULL`, c-pointer \Rightarrow contained pointer cast to `bytes*`
- C to Scheme conversion: `NULL` \Rightarrow `#f`, anything else \Rightarrow new c-pointer containing the pointer and identified as type `bytes`

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 produce a sequence of declarations.

- `(c-include path-spec)` expands to a use of `c-declare` with the content of `path-spec`. The `path-spec` has the same form as for `include` in MzLib's `include.ss`.

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 `-k`, `--make`, `-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 difference between `-k/--make` and `-z/--zo` is that the former works only on modules, it recursively compiles imported modules, it reads and writes `.dep` files to manage dependencies, and it automatically places files in the right directory for autodetection (see §3.3).

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`, `.so`, or `.dylib`).

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 non-module 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 requires 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). In `--make` mode, `mzc` compiles imported modules before loading them for syntax exports.

3.2 Compilation without Modules

Outside of a module, top-level `define-syntax[es]`, `module`, `require`, `require-for-syntax`, `begin-for-syntax`, `define[-values]-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

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

```
(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 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.

Use `--auto-dir` instead of `--destination` to have **mzc** compute the autodetect location from the input file's path:

```
mzc --extension --auto-dir file.ss
```

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 § in *PLT MzScheme: Language Manual* for details. Every source file in a collection should contain a single `module` declaration.

The `--collection-zos` and `--collection-extension` flags direct `mzc` to compile a whole collection. The `--collection-zos` 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. Alternately, for compiling to `.zo`, use `Setup PLT` instead of `mzc`, because `Setup PLT` tracks dependencies reliably.

5. Building and Distributing Stand-alone Executables

For all compilation modes, the output of **mzc** relies on MzScheme (and MrEd) to provide run-time support to the compiled code. However, **mzc** can also package code together with its run-time support to form an executable that works on the source machine or a package that can be distributed to other machines.

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

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 the forms of the `runtime-path.ss` MzLib library 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 the following section).

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 `embed.ss` library in the **compiler** collection provides a more general interface to the embedding mechanism.

Example:

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

Under Windows, this command produces **hello.exe**, which runs the same as invoking the **hello.ss** module. Under Mac OS X, the resulting executable is an application **hello.app**.

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 MzScheme and/or MrEd shared libraries, and possibly other run-time files declared via `runtime-path.ss`. The executable can be packaged with support libraries to create a distribution, as described in the following section.

5.2 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 the **runtime-path.ss** MzLib library. 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:

Example:

```
mzc --exe-dir greetings hello.exe goodbye.exe
```

Under Windows, this example 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 the `++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 distributed 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.

5.3 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-glu` 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-glu` 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 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-glu --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 Library 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 flags indicating whether they are to be unpacked relative to the PLT add-ons directory (which is user-specific), the PLT 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 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 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 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 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 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 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 §7):

- **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 its **info.ss** file, and the default version is the empty list. The version sequence generalizes 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 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:

```
(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.
(string-constant *identifier*) ; a string constant defined in
; the string-constants collection

info-primitive is one of

```
cons car cdr list
list* reverse append
string-append
path->string 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.

License

GNU Library General Public License

Version 2, June 1991

Copyright (C) 1991 Free Software Foundation, Inc.

675 Mass Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the library GPL. It is numbered 2 because it goes with version 2 of the ordinary GPL.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor's protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect transforming the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License, which was designed for utility programs. This license, the GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary one; be sure to read it in full, and don't assume that anything in it is the same as in the ordinary license.

The reason we have a separate public license for some libraries is that they blur the distinction we usually make between modifying or adding to a program and simply using it. Linking a program with a library, without changing the library, is in some sense simply using the library, and is analogous to running a utility program or application program. However, in a textual and legal sense, the linked executable is a combined work, a derivative of the original library, and the ordinary General Public License treats it as such.

Because of this blurred distinction, using the ordinary General Public License for libraries did not effectively promote software sharing, because most developers did not use the libraries. We concluded that weaker conditions might promote sharing better.

However, unrestricted linking of non-free programs would deprive the users of those programs of all benefit from the free status of the libraries

themselves. This Library General Public License is intended to permit developers of non-free programs to use free libraries, while preserving your freedom as a user of such programs to change the free libraries that are incorporated in them. (We have not seen how to achieve this as regards changes in header files, but we have achieved it as regards changes in the actual functions of the Library.) The hope is that this will lead to faster development of free libraries.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a “work based on the library” and a “work that uses the library”. The former contains code derived from the library, while the latter only works together with the library.

Note that it is possible for a library to be covered by the ordinary General Public License rather than by this special one.

GNU LIBRARY GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Library General Public License (also called “this License”). Each licensee is addressed as “you”.

A “library” means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The “Library”, below, refers to any such software library or work which has been distributed under these terms. A “work based on the Library” means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term “modification”).

“Source code” for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library’s complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) The modified work must itself be a software library.
- b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
- c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
- d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a “work that uses the Library”. Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a “work that uses the Library” with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a “work that uses the library”. The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a “work that uses the Library” uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a “work that uses the Library” with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer’s own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

- a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable “work that uses the Library”, as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
- b) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
- c) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
- d) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the “work that uses the Library” must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

- a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
- b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients’ exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
13. The Free Software Foundation may publish revised and/or new versions of the Library General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.
- Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.
14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Index

`++copy-collects`, 14
`++setup`, 16
`--all-users`, 16
`--at-plt`, 16
`--auto-dir`, 10
`--collection-extension`, 12
`--collection-plt`, 16
`--collection-zos`, 12
`--collects-dest`, 14
`--embedded`, 15
`--exe`, 13
`--exe-dir`, 14
`--extension`, 9
`--gui-exe`, 13
`--help`, 2
`--lib`, 13
`--link-extension`, 10
`--link-glue`, 15
`--make`, 9
`--object`, 10, 15
`--plt`, 16
`--plt-name`, 16
`--prefix`, 10
`--prim`, 3
`--replace`, 16
`--xform`, 6
`--zo`, 9
`-e`, 9
`-g`, 15
`-h`, 2
`-k`, 9
`-l`, 10
`-o`, 10, 15
`-p`, 10
`-z`, 9
`.dll`, 1
`.dylib`, 1
`.plt`, 16
`.plt` distribution archives, 16
`.scm`, 9
`.so`, 1
`.ss`, 9
`.zo`, 1
`.loader.dll`, 11
`.loader.so`, 11
`bool`, 6
`byte code`, 1
`C compiler`, 2
`c-declare`, 5
`c-include`, 6
`c-lambda`, 5
CC, 2
ffi.ss, 2, 5
`char`, 6
`char-string`, 7
`command line flags`, 2
`compiling`
 collections, 12
 files, 9
 multiple files, 10
`double`, 7
`eval-when`, 10
`Feeley, Marc`, 5
`float`, 7
`foreign-function interface (FFI)`, 5
`Gambit-C`, 5
`help`, 2
info.ss, 12
info.ss format, 18
infotab.ss library, 18
`int`, 7
`loading compiled files`, 1, 10
`long`, 7
`module`, 9
mzc, 1
MZSCHEME_DYNEXT_COMPILER, 2
`native code`, 1
`nonnull-char-string`, 7
`pointer`, 8
`require`, 9
`running mzc`, 2
`scheme-object`, 8
`scheme_initialize`, 15
`short`, 7
`signed-char`, 7
`stand-alone executables`, 2, 13

STANDALONE_WITH_EMBEDDED_EXTENSION, 15
syntax, 9

unsigned-char, 7
unsigned-int, 7
unsigned-long, 7
unsigned-short, 7