PLT MzScheme: Language Manual

Matthew Flatt (mflatt@plt-scheme.org)

370
Released May 2007

Copyright notice

Send us your Web links

If you use any parts or all of the PLT Scheme package (software, lecture notes) for one of your courses, for your research, or for your work, we would like to know about it. Furthermore, if you use it and publicize the fact on some Web page, we would like to link to that page. Please drop us a line at *scheme@plt-scheme.org*. Evidence of interest helps the DrScheme Project to maintain the necessary intellectual and financial support. We appreciate your help.

Thanks

Thanks to Brent Benson for `libscheme`, and to Hans Boehm for the conservative garbage collector and their help.

This manual was typeset using LaTeX, SLaTeX, and `tex2page`. Some typesetting macros were originally taken from Julian Smart's *Reference Manual for wxWindows 1.60: a portable C++ GUI toolkit*.

This manual was typeset on May 20, 2007.

Contents

1 Introduction **1**

1.1 MrEd, DrScheme, and **mzc** 1

1.2 Unicode, Locales, Strings, and Ports 2

1.2.1 Unicode 2

1.2.2 Locale 2

1.2.3 Encodings and Ports 3

1.3 Notation 4

2 Basic Syntax Extensions **5**

2.1 Evaluation Order 5

2.2 Multiple Return Values 5

2.3 Cond and Case 5

2.4 When and Unless 6

2.5 And and Or 6

2.6 Sequences 6

2.7 Quote and Quasiquote 6

2.8 Binding Forms 7

2.8.1 Definitions 7

2.8.2 Local Bindings 8

2.8.3 Assignments 9

2.8.4 Fluid-Let 9

2.8.5 Syntax Expansion and Internal Definitions 9

2.9 Case-Lambda 10

2.10 Procedure Application 11

2.11 Variable Reference 11

2.12 Forcing Expression Parsing . . . 11

3 Basic Data Extensions **12**

3.1 Void and Undefined . . . 12

3.2 Booleans . . . 12

3.3 Numbers . . . 13

3.4 Characters . . . 15

3.5 Strings . . . 17

3.6 Byte Strings . . . 19

3.7 Symbols . . . 23

3.8 Keywords . . . 23

3.9 Vectors . . . 24

3.10 Lists . . . 24

3.11 Boxes . . . 24

3.12 Procedures . . . 25

3.12.1 Arity . . . 25

3.12.2 Primitives . . . 26

3.12.3 Procedure Names . . . 26

3.12.4 Closure Equality . . . 26

3.13 Promises . . . 26

3.14 Hash Tables . . . 26

4 Structures **29**

4.1 Defining Structure Types . . . 29

4.2 Creating Subtypes . . . 30

4.3 Structure Types with Automatic Fields, Immutable Fields, and Properties . . . 30

4.4 Structure Type Properties . . . 32

4.5 Structure Inspectors . . . 33

4.6 Structures as Procedures . . . 35

4.7 Structures as Synchronizable Events . . . 36

4.8 Structures as Ports . . . 37

4.9 Structure Utilities . . . 37

5 Modules **38**

5.1 Module Expansion and Execution . . . 38

5.2 Module Bodies . . . 39

5.3 Modules and Macros . . . 41

5.4 Module Paths . . . 42

5.4.1 Module Name Resolver . . . 42

5.4.2 Module Names and Compilation . . . 43

5.5 Dynamic Module Access . . . 44

5.6 Re-declaring Modules . . . 45

5.7 Built-in Modules . . . 45

5.8 Modules and Load Handlers . . . 45

6 Exceptions and Control Flow **47**

6.1 Exceptions . . . 47

6.1.1 Primitive Exceptions . . . 48

6.2 Errors . . . 50

6.2.1 Application Errors . . . 50

6.2.2 Syntax Errors . . . 51

6.2.3 Inferred Value Names . . . 51

6.3 Continuations . . . 52

6.4 Dynamic Wind . . . 53

6.5 Prompts and Composable Continuations . . . 54

6.6 Continuation Marks . . . 55

6.7 Breaks . . . 58

6.8 Error Escape Handler . . . 59

7 Threads **61**

7.1 Suspending, Resuming, and Killing Threads . . . 61

7.2 Synchronizing Thread State . . . 62
7.3 Additional Thread Utilities . . . 63
7.4 Semaphores . . . 63
7.5 Channels . . . 64
7.6 Alarms . . . 65
7.7 Synchronizing Events . . . 65
7.8 Thread-Local Storage Cells . . . 68
7.9 Parameters . . . 69
7.9.1 Built-in Parameters . . . 71
7.9.2 Parameter Utilities . . . 77

8 Namespaces **78**
8.1 Identifier Resolution in Namespaces . . . 79
8.2 Initial Namespace . . . 79
8.3 Namespace Utilities . . . 79

9 Security **82**
9.1 Security Guards . . . 82
9.2 Custodians . . . 83
9.3 Thread Groups . . . 85
9.4 Inspectors and Modules . . . 85

10 Regular Expressions **87**

11 Input and Output **95**
11.1 Ports . . . 95
11.1.1 End-of-File Constant . . . 95
11.1.2 Current Ports . . . 95
11.1.3 Opening File Ports . . . 96
11.1.4 Pipes . . . 97
11.1.5 String Ports . . . 97

11.1.6 File-Stream Ports 98
11.1.7 Custom Ports 99
11.2 Reading and Writing 111
11.2.1 Reading Bytes, Characters, and Strings 112
11.2.2 Writing Bytes, Characters, and Strings 115
11.2.3 Writing Structured Data 117
11.2.4 Default Reader 118
11.2.5 Default Printer 122
11.2.6 Replacing the Reader 124
11.2.7 Replacing the Printer 124
11.2.8 Customizing the Reader through Readtables 125
11.2.9 Reader-Extension Procedures 129
11.2.10 Customizing the Printer through Custom-Write Procedures 131
11.3 Filesystem Utilities 132
11.3.1 Paths 132
11.3.2 Locating Paths 136
11.3.3 Files 138
11.3.4 Directories 139
11.4 Networking 139
11.4.1 TCP 140
11.4.2 UDP 142

12 Syntax and Macros **145**

12.1 `syntax-rules` Extensions 145
12.2 Syntax Objects 147
12.2.1 Syntax Patterns 148
12.2.2 Syntax Object Content 151
12.3 Syntax and Lexical Scope 153
12.3.1 Syntax Object Comparisons 154
12.3.2 Syntax Object Bindings 154

12.3.3 Transformer Environments . . . 155
12.3.4 Module Environments . . . 156
12.3.5 Macro-Generated Top-Level and Module Definitions . . . 158
12.4 Binding Multiple Syntax Identifiers . . . 160
12.5 Special Syntax Identifiers . . . 161
12.6 Macro Expansion . . . 163
12.6.1 Expanding Expressions to Primitive Syntax . . . 167
12.6.2 Syntax Object Properties . . . 169
12.6.3 Certificates for Protected References . . . 170
12.6.4 Information on Structure Types . . . 174
12.6.5 Information on Expanded and Compiled Modules . . . 175

13 Memory Management **177**
13.1 Weak Boxes . . . 177
13.2 Ephemerons . . . 177
13.3 Will Executors . . . 178
13.4 Garbage Collection . . . 179

14 Evaluation and Compilation **180**
14.1 Eval and Load . . . 180
14.2 Exiting . . . 181
14.3 Compilation . . . 181
14.4 Dynamic Extensions . . . 182

15 System Utilities **183**
15.1 Time . . . 183
15.1.1 Real Time and Date . . . 183
15.1.2 Machine Time . . . 183
15.1.3 Timing Execution . . . 184
15.2 Operating System Processes . . . 184
15.3 Windows Actions . . . 185

15.4 Operating System Environment Variables 186
15.5 Runtime Information 186

16 Library Collections and MzLib **189**

17 Running MzScheme **191**
17.1 Command-Line Conventions 192
17.2 Executable Name 192
17.3 Initialization 193

18 Writing and Running Scripts **194**

19 Honu **196**
19.1 Honu Input Parsing 196
19.1.1 Numbers 197
19.1.2 Identifiers 197
19.1.3 Strings 197
19.1.4 Characters 197
19.1.5 Parentheses, Brackets, and Braces 198
19.1.6 Comments 198
19.2 Honu Output Printing 198

20 Platform-Specific Path Conventions **200**
20.1 Unix and Mac OS X Path Conventions 200
20.2 Windows Path Conventions 200

License **204**

Index **208**

1. Introduction

The core of the Scheme programming language is described in *Revised^5 Report on the Algorithmic Language Scheme*. This manual assumes familiarity with Scheme and only contains information specific to MzScheme. (Many sections near the front of this manual simply clarify MzScheme's position with respect to the standard report.)

MzScheme (pronounced "miz scheme", as in "Ms. Scheme") is mostly R^5RS-compliant. Certain parameters in MzScheme can change features affecting R^5RS-compliance; for example, case-sensitivity is initially enabled (see §7.9.1.3).

MzScheme provides several notable extensions to R^5RS Scheme:

- A module system for namespace and compilation management (see Chapter 5).
- An exception system that is used for all primitive errors (see Chapter 6).
- Pre-emptive threads (see Chapter 7).
- A class and object system (see Chapter 6 of *PLT MzLib: Libraries Manual*).
- A unit system for defining and linking program components (see Chapter 55 of *PLT MzLib: Libraries Manual*).
- Extensive Unicode and character-encoding support (see §1.2).

MzScheme can be run as a stand-alone application, or it can be embedded within other applications. Most of this manual describes the language that is common to all uses of MzScheme. For information about running the stand-alone version of MzScheme, see Chapter 17.

1.1 MrEd, DrScheme, and **mzc**

MrEd is an extension of MzScheme for graphical programming. MrEd is described separately in *PLT MrEd: Graphical Toolbox Manual*.

DrScheme is a development environment for writing MzScheme- and MrEd-based programs. DrScheme provides debugging and project-management facilities, which are *not* provided by the stand-alone MzScheme application, and a user-friendly interface with special support for using Scheme as a pedagogical tool. DrScheme is described in *PLT DrScheme: Development Environment Manual*.

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**, **.dll**, or **.dylib** files) to be loaded into MzScheme (or MrEd). The **mzc** compiler is described in *PLT mzc: MzScheme Compiler Manual*.

MzScheme and its extensions are available with two different variants that use slightly different approaches to memory management. The default variant is MzScheme3m; its "precise" memory management usually provides better overall performance than the alternative. The alternative is MzSchemeCGC; its "conservative" memory management allows it to cooperate more easily with extension and embedding code written in C. See *Inside PLT MzScheme* for more information.

1.2 Unicode, Locales, Strings, and Ports

As explained in the following subsections, MzScheme distinguishes characters from bytes and character strings from byte strings. MzScheme's notion of "character" corresponds to a Unicode scalar value (i.e., a Unicode code point that is not a surrogate), and many operations assume the UTF-8 encoding when converting between characters and bytes. For a handful of conversions, the user's chosen locale determines an encoding, instead. The chosen locale also affects string case folding and comparison for operations whose name includes `locale`.

1.2.1 Unicode

Unicode defines a standard mapping between sequences of integers and human-readable "characters." More precisely, Unicode distinguishes between *glyphs*, which are printed for humans to read, and *characters*, which are abstract entities that map to glyphs, sometimes in a way that's sensitive to surrounding characters. Furthermore, different sequences of integers—or *code points* in Unicode terminology—sometimes correspond to the same character. The relationships among code points, characters, and glyphs are subtle and complex.

Despite this complexity, most things that a literate human would call a "character" can be represented by a single code point in Unicode (though it may also be represented by other sequences). For example, Roman letters, Cyrillic letters, Chinese characters, and Hebrew consonants all fall into this category. The "code point" approximation of "character" thus works well for many purposes, and MzScheme defines the `char` datatype to correspond to a Unicode code point. (More precisely, a `char` corresponds to a Unicode scalar value, which excludes *surrogate* code points that are used to encode other code points in certain contexts.) For the remainder of this manual, we use "character" interchangeably with "code point" or "MzScheme's `char` datatype."

Besides printing and reading characters, humans also compare characters or character strings, and humans perform operations such as changing characters to uppercase. To make programs geographically portable, humans must agree to compare or upcase characters consistently, at least in certain contexts. The Unicode standard provides such a standard mapping on code points, and this mapping is used to case-normalize symbols in MzScheme. In other contexts, global agreement is unnecessary, and the user's culture should determine the operation, such as when displaying a list of file names. Cultural dependencies are captured by the user's *locale*, which is discussed in the next section.

Most computing devices are built around the concept of *byte* (an integer from 0 to 255) instead of character. To communicate character sequences among devices, then, requires an encoding of characters into bytes. *UTF-8* is one such encoding; due to its nice properties, the UTF-8 encoding is in many ways hard-wired into MzScheme's primitives, such as `read-char`. Encodings are discussed further in the following sections. For byte-based communication, MzScheme supports byte strings as a separate datatype from character strings (see §3.6).

For official information on the Unicode standard, see http://www.unicode.org/. For a thorough but more accessible introduction, see *Unicode Demystified* by Richard Gillam.

1.2.2 Locale

A *locale* captures information about a user's culture-specific interpretation of character sequences. In particular, a locale determines how strings are "alphabetized," how a lowercase character is converted to an uppercase character, and how strings are compared without regard to case. String operations such as `string-ci?` are *not* sensitive to the current locale, but operations such as `string-locale-ci?` (see §3.5) produce results consistent with the current locale.

Under Unix, a locale also designates a particular encoding of code-point sequences into byte sequences. MzScheme generally ignores this aspect of the locale, with a few notable exceptions: command-line arguments passed to MzScheme as byte strings are converted to character strings using the locale's encoding; command-line strings passed as byte strings to other processes (through `subprocess`) are converted to byte strings using the locale's encoding; environment variables are converted to and from strings using the locale's encoding; filesystem paths are converted to and from strings (for display purposes) using the locale's encoding; finally, MzScheme provides functions such as `string->bytes/locale` to specifically invoke a locale-specific encoding.

A Unix user selects a locale by setting environment variables, such as **LC_ALL**. Under Windows and Mac OS X, the operating system provides other mechanisms for setting the locale. Within MzScheme, the current locale can be changed by setting the `current-locale` parameter (see §7.9 and §7.9.1.11). The locale name within MzScheme is a string, and the available locale names depend on the platform and its configuration, but the `""` locale means the current user's default locale; under Windows and Mac OS X, the encoding for `""` is always UTF-8, and locale-sensitive operations use the operating system's native interface.[1] Setting the current locale to `#f` makes locale-sensitive operations locale-insensitive, which means using the Unicode mapping for case operations and using UTF-8 for encoding.

1.2.3 Encodings and Ports

The *UTF-8* encoding of characters to bytes has a number of important properties:

- Each code point from 0 to 127 (i.e., each ASCII character) is encoded by the corresponding byte from 0 to 127.
- Other code points are represented by a sequence of two to six bytes, where each byte is in the range 128 to 253. Furthermore, the first byte in the sequence is between 192 and 253, and each subsequent byte is between 128 and 191.
- Not every sequence starting with 192-to-253 followed by 128-to-191 encodes a code point. The bytes 254 and 255 are never used to encode any code point.
- Every code-point sequence has a unique encoding in bytes, and every valid encoding in bytes has a unique decoding into code points.

For a more complete description of UTF-8, see `http://www.cl.cam.ac.uk/~mgk25/unicode.html`.

Another useful encoding is *Latin-1*, where every code point from 0 to 255 is encoded by the corresponding byte, and no other code points can be encoded.[2] Every byte sequence is therefore a valid encoding with a unique decoding, but not every character string can be encoded.

MzScheme supports these two encodings through functions such as `bytes->string/utf-8` and `string->bytes/latin-1` (see §3.6). These functions accept an extra argument so that an un-encodable character or un-decodeable sequence is replaced by a specific character or byte, instead of raising an exception. MzScheme also provides `bytes->string/locale` and `string->bytes/locale`; typically, a locale-specific encoding cannot encode all characters, and not all byte sequences are valid encodings in the encoding.

All ports in MzScheme produce and consume bytes. When a port is provided to character-based operations, such as `read`, the port's bytes are interpreted as a UTF-8 encoding of characters. Moreover, when tracking position, line, and column information for an input port, position and column are computed in terms of decoded characters, rather than bytes.

Bytes streams that correspond to other encodings must be transformed to or from a UTF-8 byte stream, possibly using a converter produced by `bytes-convert` (see §3.6). When an input port produces a sequence of bytes that is not a valid UTF-8 encoding in a character-reading context, certain bytes in the sequence are converted to the character "?" (see §11.1).

[1] In particular, setting the **LC_ALL** and **LC_CTYPE** environment variables do not affect the locale `""` under Mac OS X. Use `getenv` and `current-locale` to explicitly install the environment-specified locale, if desired.

[2] Technically, Latin-1 (as defined by ISO standard 8859) doesn't include control characters in 0 to 31 and 127 to 159. Like much other software, MzScheme uses an extended definition of Latin-1 that includes those control characters. Beware of encodings that claim to be Latin-1/ISO-8859-1 but that are actually Windows-1252, because Windows-1252 is an extension of Latin-1 that is not a subset of Unicode.

1.3 Notation

Throughout this manual, the syntax for new forms is described using a pattern notation with ellipses. Plain, centered ellipses ($\cdots$) indicate *zero* or more repetitions of the preceding pattern. Ellipses with a "1" superscript ($\cdots^1$) indicate *one* or more repetitions of the preceding pattern.

For example:

```
(let-values (((variable ···) expr) ···)
 body-expr
 ···¹)
```

The first set of ellipses indicate that any number of `variable`s, possibly none, can be provided with a single `expr`. The second set of ellipses indicate that any number of `((variable ···) expr)` combinations, possibly none, can appear in the parentheses following the `let-values` syntax name. The last set of ellipses indicate that a `let-values` expression can contain any number of `body-expr` expressions, as long as at least one expression is provided. In describing parts of the `let-values` syntax, the name `variable` is used to refer to a single binding variable in a `let-values` expression.

Some examples contain simple ellipses (`...`), which is an identifier, albeit one that has special meaning in syntax patterns and templates.

Square brackets ("[" and "]") are normally treated as parentheses by MzScheme, and this manual uses square brackets as parentheses in example code. However, in describing a MzScheme procedure, this manual uses square brackets to designate optional arguments. For example,

```
(regexp-match pattern string [start-k end-k])
```

describes the calling convention for a procedure `regexp-match` where the `pattern` and `string` arguments are required, and the `start-k` and `end-k` arguments are optional (but `start-k` must be provided if `end-k` is provided).

In grammar specifications for syntactic forms, `variable` and `identifier` are equivalent, but `variable` is often used when the identifier corresponds to a location that holds a value at run time.

2. Basic Syntax Extensions

2.1 Evaluation Order

In an application expression, the procedure expression and the argument expressions are always evaluated left-to-right. Similarly, expressions for `let` and `letrec` bindings are evaluated in sequence from left to right.

2.2 Multiple Return Values

MzScheme supports the *R^5RS* `values` and `call-with-values` procedure, and also provides binding forms for multiple-value expressions, discussed in §2.8.

Multiple return values are legal in MzScheme whenever the return value of an expression is ignored. For example, all but the last expression in a `begin` form can legally return multiple values in any context. If a built-in procedure takes a procedure argument, and the built-in procedure does not inspect the result of the supplied procedure, then the supplied procedure can return multiple values. For example, the procedure supplied to `for-each` can return any number of values, but the procedure supplied to `map` must return a single value.

When the number of values returned by an expression does not match the number of values expected by the expression's context, the `exn:fail:contract:arity` exception is raised (at run time).

Examples:

```
(- (values 1)) ; ⇒ -1
(- (values 1 2)) ; ⇒ error: returned 2 values to single-value context
(- (values)) ; ⇒ error: returned 0 values to single-value context
(call-with-values
  (lambda () (values 1 2))
  (lambda (x y) y)) ; ⇒ 2
(call-with-values
  (lambda () (values 1 2))
  (lambda z z)) ; ⇒ (1 2)
(call-with-values
  (lambda () (let/cc k (k 3 4)))
  (lambda (x y) y)) ; ⇒ 4
(call-with-values
  (lambda () (values 'hello 1 2 3 4))
  (lambda (s . l)
    (format "~s = ~s" s l))) ; ⇒ "hello = (1 2 3 4)"
```

2.3 Cond and Case

The `else` and `=>` identifiers in a `cond` or `case` statement are handled specially only when they are not lexically bound or module-bound:

```
(cond [1 => add1]) ; ⇒ 2
(let ([=> 5]) (cond [1 => add1])) ; ⇒ #<primitive:add1>
```

2.4 When and Unless

The `when` and `unless` forms conditionally evaluate a single body of expressions:

- `(when` *`test-expr expr`* $\cdots^1$`)` evaluates the *`expr`* body expressions only when *`test-expr`* returns a true value.
- `(unless` *`test-expr expr`* $\cdots^1$`)` evaluates the *`expr`* body expressions only when *`test-expr`* returns `#f`.

The result of a `when` or `unless` expression is the result of the last body expression if the body is evaluated, or void (see §3.1) if the body is not evaluated.

2.5 And and Or

In an `and` or `or` expression, the last test expression can return multiple values (see §2.2). If the last expression is evaluated and it returns multiple values, then the result of the entire `and` or `or` expression is the multiple values. Other sub-expressions in an `and` or `or` expression must return a single value.

2.6 Sequences

As a top-level form, `begin` wraps each sub-expression but the last with a prompt (see §6.5) using the default prompt tag and an abort handler that re-aborts. This wrapping helps maintains the equivalence between wrapping a sequence top-level forms with a `begin` and splicing the sequence into the enclosing context.

The `begin0` form is like `begin`, but the value of the first expression in the form is returned instead of the value of the last expression:

```
(let ([x 4])
  (begin0 x (set! x 9) (display x))) ; ⇒ displays 9 then returns 4
```

The first sub-expression in a `begin0` expression is in tail position if and only if it is the only sub-expression.

2.7 Quote and Quasiquote

The `quote` form never allocates, so that the result of multiple evaluations of a single `quote` expression are always `eq?`. Nevertheless, a quoted cons cell, vector, or list is mutable; mutations to the result of a `quote` application are visible to future evaluations of the `quote` expression.

The `quasiquote` form allocates only as many fresh cons cells, vectors, and boxes as are needed without analyzing `unquote` and `unquote-splicing` expressions. For example, in

```
`(,1 2 3)
```

a single reader-allocated tail `'(2 3)` is used for every evaluation of the `quasiquote` expression.

The standard Scheme `quasiquote` has been extended so that `unquote` and `unquote-splicing` work within immediate boxes:

```
`#&(,(- 2 1) ,@(list 2 3)) ; ⇒ #&(1 2 3)
```

See §11.2.4 for more information about immediate boxes.

MzScheme defines the `unquote` and `unquote-splicing` identifiers as top-level syntactic forms that always report a syntax error. The `quasiquote` form recognizes normal `unquote` and `unquote-splicing` uses via `module-identifier=?`. (See §12.3.1 for more information on identifier comparisons.)

2.8 Binding Forms

2.8.1 Definitions

A procedure definition

```
(define variable (lambda formals expr ···¹))
```

can be abbreviated

```
(define (variable . formals) expr ···¹)
```

In addition to this standard Scheme abbreviation, MzScheme supports an MIT-style generalization, so that a definition

```
(define header (lambda formals expr ···¹))
```

can be abbreviated

```
(define (header . formals) expr ···¹))
```

even if *header* is itself a parenthesized procedure abbreviation. The general syntax of `define` is as follows:

```
(define variable expr)
(define (header . formals) expr ···¹)

header is one of
  variable
  (header . formals)

formals is one of
  variable
  (variable ···)
  (variable variable ··· . variable)
```

Multiple values can be bound to multiple variables at once using `define-values`:

```
(define-values (variable ···) expr)
```

The number of values returned by *expr* must match the number of *variable*s provided, and the *variable*s must be distinct. No procedure-definition abbreviation is available for `define-values`.

Examples:

```
(define x 1)
x ; ⇒ 1
(define (f x) (+ x 1))
```

```
(f 2) ; ⇒ 3
(define (((g x) y z) . w) (list x y z w))
(let ([h ((g 1) 2 3)])
  (list (h 4 5) (h 6))) ; ⇒ '((1 2 3 (4 5)) (1 2 3 (6)))

(define-values (x) 2)
x ; ⇒ 2
(define-values (x y) (values 3 4))
x ; ⇒ 3
y ; ⇒ 4
(define-values (x y) (values 5 (add1 x)))
y ; ⇒ 4
(define-values () (values)) ; same as (void)
(define x (values 7 8)) ; ⇒ error: 2 values for 1-value context
(define-values (x y) 7) ; ⇒ error: 1 value for 2-value context
(define-values () 7) ; ⇒ error: 1 value for 0-value context
```

2.8.2 Local Bindings

Local variables are bound with standard Scheme's `let`, `let*`, and `letrec`. MzScheme's `letrec` form guarantees sequential left-to-right evaluation of the binding expressions. (The `letrec` bound in the result of `(scheme-report-environment 5)`, however, is defined exactly as in R^5RS.)

Multiple values are bound to multiple local variables at once with `let-values`, `let*-values`, and `letrec-values`. The syntax for `let-values` is:

```
(let-values (((variable ···) expr) ···) body-expr ···¹)
```

As in `define-values`, the number of values returned by each *`expr`* must match the number of *`variable`*s declared in the corresponding clause. Each *`expr`* remains outside of the scope of all variables bound by the `let-values` expression.

The syntax for `let*-values` and `letrec-values` is the same as for `let-values`, and the binding semantics for each form corresponds to the single-value binding form:

- In a `let*-values` expression, the scope of the variables of each clause includes all of the remaining binding clauses. The clause expressions are evaluated and bound to variables sequentially.
- In a `letrec-values` expression, the scope of the variables of each clause includes all of the binding clauses. The clause expressions are evaluated and bound to variables sequentially.

When a `letrec` or `letrec-values` expression is evaluated, each variable binding is initially assigned the special undefined value (see §3.1); the undefined value is replaced after the corresponding expression is evaluated.

Examples:

```
(define x 0)
(let ([x 5] [y x]) y) ; ⇒ 0
(let* ([x 5] [y x]) y) ; ⇒ 5
(letrec ([x 5] [y x]) y) ; ⇒ 5
(letrec ([x y] [y 5]) x) ; ⇒ undefined
(let-values ([(x) 5] [(y) x]) y) ; ⇒ 0
(let-values ([(x y) (values 5 x)]) y) ; ⇒ 0
(let*-values ([(x) 5] [(y) x]) y) ; ⇒ 5
```

```
(let*-values ([(x y) (values 5 x)]) y) ; ⇒ 0
(letrec-values ([(x) 5] [(y) x]) y) ; ⇒ 5
(letrec-values ([(x y) (values 5 x)]) y) ; ⇒ undefined
(letrec-values ([(odd even) (values
                              (lambda (n) (if (zero? n) #f (even (sub1 n))))
                              (lambda (n) (if (zero? n) #t (odd (sub1 n)))))])
   (odd 17)) ; ⇒ #t
```

2.8.3 Assignments

The standard `set!` form assigns a value to a single global, local, or module variable. Multiple variables can be assigned at once using `set!-values`:

```
(set!-values (variable ···) expr)
```

The number of values returned by *expr* must match the number of *variable*s provided.

The *variable*s, which must be distinct, can be any mixture of global, local, and module variables. Assignments are performed sequentially from the first *variable* to the last. If an error occurs in one of the assignments (perhaps because a global variable is not yet bound), then the assignments for the preceding *variable*s will have already completed, but assignments for the remaining *variable*s will never complete.

2.8.4 Fluid-Let

The syntax for a `fluid-let` expression is the same as for `let`:

```
(fluid-let ((variable expr) ···) body-expr ···1)
```

Each *variable* must be either a local variable or a global or module variable that is bound before the `fluid-let` expression is evaluated. Before the *body-expr*s are evaluated, the bindings for the *variable*s are `set!` to the values of the corresponding *expr*s. Once the *body-expr*s have been evaluated, the values of the variables are restored. The value of the entire `fluid-let` expression is the value of the last *body-expr*.

2.8.5 Syntax Expansion and Internal Definitions

All binding forms are syntax-expanded into `define-values`, `let-values`, `letrec-values`, `define-syntaxes`, and `letrec-syntaxes+values` expressions. The `set!-values` form is expanded to `let-values` with `set!`. See §12.6.1 for more information.

All `define-values` expressions that are inside only `begin` expressions are treated as top-level definitions. Body `define-values` expressions in a `module` expression are handled specially as described in §5.1. Any other `define-values` expression is either an *internal definition* or syntactically illegal. The same is true of `define-syntaxes` expressions.

Internal definitions can appear at the start of a sequence of expressions, such as the start of a `lambda`, `case-lambda`, or `let` body. At least one non-definition expression must follow a sequence of internal definitions. The first expression in a `begin0` expression cannot be an internal definition; for the purposes of internal definitions, the second expression is the start of the sequence.

When a `begin` expression appears within a sequence, its content is inlined into the sequence (recursively, if the `begin` expression contains other `begin` expressions). Like top-level `begin` expressions (and unlike other `begin` expressions), a `begin` expression within an internal definition sequence can be empty.

An internal `define-values` or `define-syntaxes` expression is transformed, along with the expressions following it, into a `letrec-syntaxes+values` expression: the identifiers bound by the internal definitions become the binding identifiers of the new `letrec-syntaxes+values` expression, and the expressions that follow the definitions become the body of the new `letrec-syntaxes+values` expression.

Multiple adjacent definitions are collected into a single `letrec-syntaxes+values` transformation, so that the definitions can be mutually recursive, but the definitions expressions must be adjacent. A non-definition marks the start of a sequence of expressions to be moved into the body of the newly created `letrec-syntaxes+values` form.

Internal definitions are detected after a partial syntax expansion that stops at core forms, and thus exposes `begin`, `define-values`, and `define-syntaxes`. Forms are expanded left to right, and whenever a definition is discovered, a binding is introduced immediately for further expansion, so a definition can shadow variables when later forms are expanded. Furthermore, when a `define-syntaxes` form is discovered, the right-hand side is immediately evaluated, and the result is bound as syntax to the corresponding identifier(s); thus, a locally defined macro can be used to generate later definitions in the same internal-definition context.

2.9 Case-Lambda

The `case-lambda` form creates a procedure that dispatches to a particular body of expressions based on the number of arguments that the procedure receives. The `case-lambda` form provides a mechanism for creating variable-arity procedures with more control and efficiency than using a `lambda` "rest argument," such as the *x* in `(lambda (`*a* `.` *x*`)` *expr* $\cdots^1$`)`.

A `case-lambda` expression has the form:

```
(case-lambda
  (formals expr ···¹)
  ···)

formals is one of
  variable
  (variable ···)
  (variable ··· . variable)
```

Each `(`*formals expr* $\cdots^1$`)` clause of a `case-lambda` expression is analogous to a `lambda` expression of the form `(lambda` *formals expr* $\cdots^1$`)`. The scope of the *variable*s in each clause's *formals* includes only the same clause's *expr*s. The *formals* variables are bound to actual arguments in an application in the same way that `lambda` variables are bound in an application.

When a `case-lambda` procedure is invoked, one clause is selected and its *expr*s are evaluated for the application; the result of the last *expr* in the clause is the result of the application. The clause that is selected for an application is the first one with a *formals* specification that can accommodate the number of arguments in the application.[1]

Examples:

```
(define f
  (case-lambda
    [(x) x]
    [(x y) (+ x y)]
    [(a . any) a]))
(f 1) ; ⇒ 1
(f 1 2) ; ⇒ 3
```

[1]It is possible that a clause in a `case-lambda` expression can never be evaluated because a preceding clause always matches the arguments.

```
(f 4 5 6 7) ; ⇒ 4
(f) ; ⇒ raises exn:fail:contract:arity
```

The result of a `case-lambda` expression is a procedure, just like the result of a `lambda` expression. Thus, the `procedure?` predicate returns `#t` when applied to the result of a `case-lambda` expression.

2.10 Procedure Application

The "empty application" form `()` expands to the quoted empty list `'()`.

2.11 Variable Reference

The `#%variable-reference` form returns a value representing the address of a top-level or module variable:

```
(#%variable-reference variable)
(#%variable-reference (#%top . variable))
```

For either form, a syntax error is reported if *`variable`* is not bound to a top-level or module variable.

The result of a `#%variable-reference` expression is opaque, with no useful operation in MzScheme. See *Inside PLT MzScheme* for information on its use in low-level extensions to MzScheme.

2.12 Forcing Expression Parsing

In certain contexts, the expansion and parsing of a syntactic form can differ depending on whether it is parsed as an expression or a top-level form. For example, at the top level, `begin` acts as a splicing form that wraps each sub-expression evaluation with a prompt (see §6.5), but no such prompts are inserted for `begin` parsed as a sequencing expression.

The `#%expression` form wraps a *`datum`* so that it must be parsed as an expression:

```
(#%expression expr)
```

A fully-expanded expression eliminates `#%expression` except as a top-level form.

3. Basic Data Extensions

3.1 Void and Undefined

MzScheme returns the unique *void* value — printed as `#<void>` — for expressions that have unspecified results in *R*^{5}*RS*. The procedure `void` takes any number of arguments and returns void:

- `(void v ···)` returns void.
- `(void? v)` returns `#t` if *v* is void, `#f` otherwise.

Variables bound by `letrec-values` that are accessible but not yet initialized are bound to the unique *undefined* value, printed as `#<undefined>`.

3.2 Booleans

Unless otherwise specified, two instances of a particular MzScheme data type are `equal?` only when they are `eq?`. Two values are `eqv?` only when they are either `eq?`, both `+nan.0`, or both = and have the same exactness and sign. (The inexact numbers `0.0` and `-0.0` are not `eqv?`, although they are =.)

The `andmap` and `ormap` procedures apply a test procedure to the elements of a list, returning immediately when the result for testing the entire list is determined. The arguments to `andmap` and `ormap` are the same as for `map`, but a single boolean value is returned as the result, rather than a list:

- `(andmap proc list ···`^{1}`)` applies *proc* to elements of the *list*s from the first elements to the last, returning `#f` as soon as any application returns `#f`. If no application of *proc* returns `#f`, then the result of the last application of *proc* is returned; more specifically, the application of *proc* to the last elements in the *list*s is in tail position with respect to the `andmap` call. If the *list*s are empty, then `#t` is returned.
- `(ormap proc list ···`^{1}`)` applies *proc* to elements of the *list*s from the first elements to the last. If any application returns a value other than `#f`, that value is immediately returned as the result of the `ormap` application. If all applications of *proc* return `#f`, then the result is `#f`; more specifically, if *proc* is applied to the last elements of the *list*s, the application is in tail position with respect to the `ormap` call. If the *list*s are empty, then `#f` is returned.

Examples:

```
(andmap positive? '(1 2 3)) ; ⇒ #t
(ormap eq? '(a b c) '(a b c)) ; ⇒ #t
(andmap positive? '(1 2 a)) ; ⇒ raises exn:fail:contract
(ormap positive? '(1 2 a)) ; ⇒ #t
(andmap positive? '(1 -2 a)) ; ⇒ #f
(andmap + '(1 2 3) '(4 5 6)) ; ⇒ 9
(ormap + '(1 2 3) '(4 5 6)) ; ⇒ 5
```

3.3 Numbers

A number in MzScheme is one of the following:

- a *fixnum* exact integer (30 bits[1] plus a sign bit)
- a *bignum* exact integer (cannot be represented in a fixnum)
- a *fraction* exact rational (represented by two exact integers)
- a *flonum* inexact rational (double-precision floating-point number)
- a *complex* number; either the real and imaginary parts are both exact or inexact, or the number has an exact zero real part and an inexact imaginary part; a complex number with an inexact zero imaginary part is a real number

MzScheme extends the number syntax of R^5RS in three ways:

- All input radixes (`#b`, `#o`, `#d`, and `#x`) allow "decimal" numbers that contain a period or exponent marker. For example, `#b1.1` is equivalent to `1.5`. In hexadecimal numbers, `e` and `d` always stand for a hexadecimal digit, not an exponent marker.
- The mantissa of a number with an exponent marker can be expressed as a fraction. For example, `1/2e3` is equivalent to `500.0`, and `1/2e2+1/2e4i` is equivalent to `50.0+5000.0i`.
- The following are inexact numerical constants: `+inf.0` (infinity), `-inf.0` (negative infinity), `+nan.0` (not a number), and `-nan.0` (same as `+nan.0`). These names can also be used within complex constants, as in `-inf.0+inf.0i`. These names are case-insensitive.

The special inexact numbers `+inf.0`, `-inf.0`, and `+nan.0` have no exact form. Dividing by an inexact zero returns `+inf.0` or `-inf.0`, depending on the sign of the dividend. The infinities are integers, and they answer `#t` for both `even?` and `odd?`. The `+nan.0` value is not an integer and is not `=` to itself, but `+nan.0` is `eqv?` to itself.[2] Similarly, `(= 0.0 -0.0)` is `#t`, but `(eqv? 0.0 -0.0)` is `#f`.

All multi-argument arithmetic procedures operate pairwise on arguments from left to right.

The `string->number` procedure works on all number representations and exact integer radix values in the range `2` to `16` (inclusive). The `number->string` procedure accepts all number types and the radix values `2`, `8`, `10`, and `16`; however, if an inexact number is provided with a radix other than `10`, the `exn:fail:contract` exception is raised.

The `add1` and `sub1` procedures work on any number:

- `(add1 z)` returns $z+1$.
- `(sub1 z)` returns $z-1$.

The following procedures work on integers:

- `(quotient/remainder n1 n2)` returns two values: `(quotient n1 n2)` and `(remainder n1 n2)`.

[1]30 bits for a 32-bit architecture, 62 bits for a 64-bit architecture.
[2]This definition of `eqv?` technically contradicts R^5RS, but R^5RS does not address strange "numbers" like `+nan.0`.

- `(integer-sqrt` *`n`*`)` returns the integer square-root of *n*. For positive *n*, the result is the largest positive integer bounded by the `(sqrt` *`n`*`)`. For negative *n*, the result is `(* (integer-sqrt (-` *`n`*`)) 0+i)`.
- `(integer-sqrt/remainder` *`n`*`)` returns two values: `(integer-sqrt` *`n`*`)` and `(-` *`n`* `(expt (integer-sqrt` *`n`*`) 2))`.

The following procedures work on exact integers in their (semi-infinite) two's complement representation:

- `(bitwise-ior` *`n`* `···)` returns the bitwise "inclusive or" of the *n*s. If no arguments are provided, the result is `0`.
- `(bitwise-and` *`n`* `···)` returns the bitwise "and" of the *n*s. If no arguments are provided, the result is `-1`.
- `(bitwise-xor` *`n`* `···)` returns the bitwise "exclusive or" of the *n*s. If no arguments are provided, the result is `0`.
- `(bitwise-not` *`n`*`)` returns the bitwise "not" of *n*.
- `(arithmetic-shift` *`n m`*`)` returns the bitwise "shift" of *n*. The integer *n* is shifted left by *m* bits; i.e., *m* new zeros are introduced as rightmost digits. If *m* is negative, *n* is shifted right by $-m$ bits; i.e., the rightmost *m* digits are dropped.
- `(integer-length` *`n`*`)` returns the number of bits in the representation of *n* after removing all leading zeros (for non-negative *n*) or ones (for negative *n*).

The `random` procedure generates pseudo-random numbers:

- `(random` *`k`*`)` returns a random exact integer in the range `0` to $k-1$ where *k* is an exact integer between 1 and $2^{31}-1$, inclusive. The number is provided by the current pseudo-random number generator, which maintains an internal state for generating numbers.[3]
- `(random)` returns a random inexact number between `0` and `1`, exclusive, using the current pseudo-random number generator.
- `(random-seed` *`k`*`)` seeds the current pseudo-random number generator with *k*, an exact integer between 0 and $2^{31}-1$, inclusive. Seeding a generator sets its internal state deterministically; seeding a generator with a particular number forces it to produce a sequence of pseudo-random numbers that is the same across runs and across platforms.
- `(pseudo-random-generator->vector` *`generator`*`)` produces a vector that represents the complete internal state of *generator*. The vector is suitable as an argument to `vector->pseudo-random-generator` to recreate the generator in its current state (across runs and across platforms).
- `(vector->pseudo-random-generator` *`vec`*`)` produces a pseudo-random number generator whose internal state corresponds to *vec*. The vector *vec* must contain six exact integers; the first three integers must be in the range `0` to `4294967086`, inclusive; the last three integers must be in the range `0` to `4294944442`, inclusive; at least one of the first three integers must be non-zero; and at least one of the last three integers must be non-zero.
- `(current-pseudo-random-generator)` returns the current pseudo-random number generator, and `(current-pseudo-random-generator` *`generator`*`)` sets the current generator to *generator*. See also §7.9.1.10.
- `(make-pseudo-random-generator)` returns a new pseudo-random number generator. The new generator is seeded with a number derived from `(current-milliseconds)`.

[3]The random number generator uses a 54-bit version of L'Ecuyer's MRG32k3a algorithm.

- `(pseudo-random-generator? v)` returns `#t` if *v* is a pseudo-random number generator, `#f` otherwise.

The following procedures convert between Scheme numbers and common machine byte representations:

- `(integer-bytes->integer bytes signed? [big-endian?])` converts the machine-format number encoded in *bytes* to an exact integer. The *bytes* must contain either 2, 4, or 8 bytes. If *signed?* is true, then the bytes are decoded as a two's-complement number, otherwise it is decoded as an unsigned integer. If *big-endian?* is true, then the first character's ASCII value provides the most significant eight bits of the number, otherwise the first character provides the least-significant eight bits, and so on. The default value of *big-endian?* is the result of `system-big-endian?`.

- `(integer->integer-bytes n size-n signed? [big-endian? to-bytes])` converts the exact integer *n* to a machine-format number encoded in a byte string of length *size-n*, which must be 2, 4, or 8. If *signed?* is true, then the number is encoded with two's complement, otherwise it is encoded as an unsigned bit stream. If *big-endian?* is true, then the most significant eight bits of the number are encoded in the first character of the resulting byte string, otherwise the least-significant bits are encoded in the first byte, and so on. The default value of *big-endian?* is the result of `system-big-endian?`.

  If *to-bytes* is provided, it must be a mutable byte string of length *size-n*; in that case, the encoding of *n* is written into *to-bytes*, and *to-bytes* is returned as the result. If *to-bytes* is not provided, the result is a newly allocated byte string.

  If *n* cannot be encoded in a string of the requested size and format, the `exn:fail:contract` exception is raised. If *to-bytes* is provided and it is not of length *size-n*, the `exn:fail:contract` exception is raised.

- `(floating-point-bytes->real bytes [big-endian?])` converts the IEEE floating-point number encoded in *bytes* to an inexact real number. The *bytes* must contain either 4 or 8 bytes. If *big-endian?* is true, then the first byte's ASCII value provides the most significant eight bits of the IEEE representation, otherwise the first byte provides the least-significant eight bits, and so on. The default value of *big-endian?* is the result of `system-big-endian?`.

- `(real->floating-point-bytes x size-n [big-endian? to-bytes])` converts the real number *x* to its IEEE representation in a byte string of length *size-n*, which must be 4 or 8. If *big-endian?* is true, then the most significant eight bits of the number are encoded in the first byte of the resulting byte string, otherwise the least-significant bits are encoded in the first character, and so on. The default value of *big-endian?* is the result of `system-big-endian?`.

  If *to-bytes* is provided, it must be a mutable byte string of length *size-n*; in that case, the encoding of *n* is written into *to-bytes*, and *to-bytes* is returned as the result. If *to-bytes* is not provided, the result is a newly allocated byte string.

  If *to-bytes* is provided and it is not of length *size-n*, the `exn:fail:contract` exception is raised.

- `(system-big-endian?)` returns `#t` if the native encoding of numbers is big-endian for the machine running MzScheme, `#f` if the native encoding is little-endian.

3.4 Characters

MzScheme characters range over Unicode scalar values (see §1.2.1), which includes characters whose values range from `#x0` to `#x10FFFF`, but not including `#xD800` to `#xDFFF`. The procedure `char->integer` returns a character's code-point number, and `integer->char` converts a code-point number to a character. If `integer->char` is given an integer that is either outside `#x0` to `#x10FFFF` or in the excluded range `#xD800` to `#xDFFF`, the `exn:fail:contract` exception is raised.

Character constants include special named characters, such as `#\newline`, plus octal representations (e.g., `#\251`), and Unicode-style hexadecimal representations (e.g., `#\u03BB`). See §11.2.4 for more information on character constants.

The character comparison procedures `char=?`, `char<?`, `char-ci=?`, etc. take two or more character arguments and check the arguments pairwise (like the numerical comparison procedures). Two characters are `eq?` whenever they are `char=?`. The expression `(char<? `*`char1 char2`*`)` produces the same result as `(< (char->integer `*`char1`*`) (char->integer `*`char2`*`))`, etc. The case-independent `-ci` procedures compare characters after case-folding with `char-foldcase` (described below).

The character predicates produce results consistent with the Unicode database[4] and (usually) SRFI-14. These procedures are fully portable; their results do not depend on the current platform or locale.

- `(char-alphabetic? `*`char`*`)` — returns `#t` if *`char`*'s Unicode general category is `Lu`, `Ll`, `Lt`, `Lm`, or `Lo`, `#f` otherwise.
- `(char-lower-case? `*`char`*`)` — returns `#t` if *`char`* has the Unicode "Lowercase" property.
- `(char-upper-case? `*`char`*`)` — returns `#t` if *`char`* has the Unicode "Uppercase" property.
- `(char-title-case? `*`char`*`)` — returns `#t` if *`char`*'s Unicode general category is `Lt`, `#f` otherwise.
- `(char-numeric? `*`char`*`)` — returns `#t` if *`char`*'s Unicode general category is `Nd`, `#f` otherwise.
- `(char-symbolic? `*`char`*`)` — returns `#t` if *`char`*'s Unicode general category is `Sm`, `Sc`, `Sk`, or `So`, `#f` otherwise.
- `(char-punctuation? `*`char`*`)` — returns `#t` if *`char`*'s Unicode general category is `Pc`, `Pd`, `Ps`, `Pe`, `Pi`, `Pf`, or `Po`, `#f` otherwise.
- `(char-graphic? `*`char`*`)` — returns `#t` if *`char`*'s Unicode general category is `Mn`, `Mc`, `Me`, or if one of the following produces `#t` when applied to *`char`*: `char-alphabetic?`, `char-numeric?`, `char-symbolic?`, or `char-punctuation?`.
- `(char-whitespace? `*`char`*`)` — returns `#t` if *`char`*'s Unicode general category is `Zs`, `Zl`, or `Zp`, or if *`char`* is one of the following: `#\tab`, `#\newline`, `#\vtab`, `#\page`, `#\return`, or `#\u0085`.
- `(char-blank? `*`char`*`)` — returns `#t` if *`char`*'s Unicode general category is `Zs` or if *`char`* is `#\tab`. (These correspond to horizontal whitespace.)
- `(char-iso-control? `*`char`*`)` — return `#t` if *`char`* is between `#\u0000` and `#\u001F` inclusive or `#\u007F` and `#\u009F` inclusive.
- `(char-general-category `*`char`*`)` — returns a symbol representing the character's Unicode general category, which is `'lu`, `'ll`, `'lt`, `'lm`, `'lo`, `'mn`, `'mc`, `'me`, `'nd`, `'nl`, `'no`, `'ps`, `'pe`, `'pi`, `'pf`, `'pd`, `'pc`, `'po`, `'sc`, `'sm`, `'sk`, `'so`, `'zs`, `'zp`, `'zl`, `'cc`, `'cf`, `'cs`, `'co`, or `'cn`.

Character conversions are also consistent with the 1-to-1 code point mapping defined by Unicode. String procedures (see §3.5) handle the case where Unicode defines a locale-independent mapping from the code point to a code-point sequence (in addition to the 1-1 mapping on scalar values).

- `(char-upcase `*`char`*`)` produces a character according to the upcase mapping provided by the Unicode database for *`char`*; if *`char`* has no upcase mapping, `char-upcase` produces *`char`*.
- `(char-downcase `*`char`*`)` produces a character according to the downcase mapping provided by the Unicode database for *`char`*; if *`char`* has no downcase mapping, `char-downcase` produces *`char`*.

[4]The current version of MzScheme uses Unicode version 4.1.

- `(char-titlecase` *`char`*`)` produces a character according to the titlecase mapping provided by the Unicode database for *`char`*; if *`char`* has no titlecase mapping, `char-titlecase` produces *`char`*.
- `(char-foldcase` *`char`*`)` produces a character according to the case-folding mapping provided by the Unicode database for *`char`*.

`(make-known-char-range-list)` produces a list of three-element lists, where each three-element list represents a set of consecutive code points for which the Unicode standard specifies character properties. Each three-element list contains two integers and a boolean; the first integer is a starting code-point value (inclusive), the second integer is an ending code-point value (inclusive), and the boolean is `#t` when all characters in the code-point range have identical results for all of the character predicates above. The three-element lists are ordered in the overall result list such that later lists represent larger code-point values, and all three-element lists are separated from every other by at least one code-point value that is not specified by Unicode.

`(char-utf-8-length` *`char`*`)` produces the same result as `(bytes-length (`*`string->bytes/utf-8`*` (`*`string char`*`)))`.

3.5 Strings

Since a string consists of a sequence of characters, a string in MzScheme is a Unicode code-point sequence. MzScheme also provides byte strings, as well as functions to convert between byte strings and strings with respect to various encodings, including UTF-8 and the current locale's encoding. See §1.2 for an overview of Unicode, locales, and encodings, and see §3.6 for more specific information on byte-string conversions.

A string can be mutable or immutable. When an immutable string is provided to a procedure like `string-set!`, the `exn:fail:contract` exception is raised. String constants generated by `read` are immutable. `(string->immutable-string` *`string`*`)` returns an immutable string with the same content as *`string`*, and it returns *`string`* itself if *`string`* is immutable. (See also `immutable?` in §3.10.)

`(substring` *`string start-k`* [*`end-k`*]`)` returns a mutable string, even if the *`string`* argument is immutable. The *`end-k`* argument defaults to `(string-length` *`string`*`)`; otherwise, `substring` is as specified by *R^5RS*.

`(string-copy!` *`dest-string dest-start-k src-string`* [*`src-start-k src-end-k`*]`)` changes the characters of *`dest-string`* from positions *`dest-start-k`* (inclusive) to *`dest-end-k`* (exclusive) to match the characters in *`src-string`* from *`src-start-k`* (inclusive). If *`src-start-k`* is not provided, it defaults to `0`. If *`src-end-k`* is not provided, it defaults to `(string-length` *`src-string`*`)`. The strings *`dest-string`* and *`src-string`* can be the same string, and in that case the destination region can overlap with the source region; the destination characters after the copy match the source characters from before the copy. If any of *`dest-start-k`*, *`src-start-k`*, or *`src-end-k`* are out of range (taking into account the sizes of the strings and the source and destination regions), the `exn:fail:contract` exception is raised.

When a string is created with `make-string` without a fill value, it is initialized with the null character (`#\nul`) in all positions.

The string comparison procedures `string=?`, `string<?`, `string-ci=?`, etc. take two or more string arguments and check the arguments pairwise (like the numerical comparison procedures). String comparisons are performed through pairwise comparison of characters; for the `-ci` operations, the two strings are first case-folded using `string-foldcase` (described below). Comparisons using all of these functions are fully portable; the results do not depend on the current platform or locale.

The following string-conversion procedures take into account Unicode's locale-independent conversion rules that map code-point sequences to code-point sequences (instead of simply mapping a 1-to-1 function on code points over the string). In each case, the string produced by the conversion can be longer than the input string.

- `(string-upcase` *`string`*`)` returns a string whose characters are the upcase conversion of the characters in *`string`*.
- `(string-downcase` *`string`*`)` returns a string whose characters are the downcase conversion of the characters in *`string`*.
- `(string-titlecase` *`string`*`)` returns a string where the first character in each sequence of cased characters in *`string`* (ignoring case-ignorable characters) is converted to titlecase, and all other cased characters are downcased.
- `(string-foldcase` *`string`*`)` returns a string whose characters are the case-fold conversion of the characters in *`string`*.

Examples:

```
(string-upcase "abc!") ; ⇒ "ABC!"
(string-upcase "Stra\xDFe") ; ⇒ "STRASSE"

(string-downcase "aBC!") ; ⇒ "abc!"
(string-downcase "Stra\xDFe") ; ⇒ "stra\xDFe"
(string-downcase "\u039A\u0391\u039F\u03A3") ; ⇒ "\u03BA\u03b1\u03BF\u03C2"
(string-downcase "\u03A3") ; ⇒ "\u03C3"

(string-titlecase "aBC  twO") ; ⇒ "Abc  Two"
(string-titlecase "y2k") ; ⇒ "Y2K"
(string-titlecase "main stra\xDFe") ; ⇒ "Main Stra\xDFe"
(string-titlecase "stra \xDFe") ; ⇒ "Stra Sse"

(string-foldcase "aBC!") ; ⇒ "abc!"
(string-foldcase "Stra\xDFe") ; ⇒ "strasse"
(string-foldcase "\u039A\u0391\u039F\u03A3") ; ⇒ "\u03BA\u03b1\u03BF\u03C3"
```

In addition to the character-based string procedures, MzScheme provides the following locale-sensitive procedures (see also §1.2.2 and §7.9.1.11):

- `(string-locale=?` *`string1 string2`* $\cdots^1$`)`
- `(string-locale<?` *`string1 string2`* $\cdots^1$`)`
- `(string-locale>?` *`string1 string2`* $\cdots^1$`)`
- `(string-locale-ci=?` *`string1 string2`* $\cdots^1$`)`
- `(string-locale-ci<?` *`string1 string2`* $\cdots^1$`)`
- `(string-locale-ci>?` *`string1 string2`* $\cdots^1$`)`
- `(string-locale-upcase` *`string`*`)` — may produce a string that is longer or shorter than *`string`* if the current locale has complex case-folding rules.
- `(string-locale-downcase` *`string`*`)` — like `string-locale-upcase`, may produce a string that is longer or shorter than *`string`*

These procedures depend only on the current locale's case-conversion and collation rules, and not on its encoding rules.

MzScheme provides four Unicode-normalization procedures:

- `(string-normalize-nfd` *`string`*`)` — returns a string that is the Unicode normalized form D of *`string`*.
- `(string-normalize-nfkd` *`string`*`)` — returns a string that is the Unicode normalized form KD of *`string`*.
- `(string-normalize-nfc` *`string`*`)` — returns a string that is the Unicode normalized form C of *`string`*.
- `(string-normalize-nfkc` *`string`*`)` — returns a string that is the Unicode normalized form KC of *`string`*.

For each of the normalization procedures, if the given string is already in the corresponding Unicode normal form, the string may be returned directly as the result (instead of a newly allocated string).

3.6 Byte Strings

A *byte string* is like a string, but it a sequence of bytes instead of characters. A *byte* is an exact integer between `0` and `255` inclusive; `(byte?` *`v`*`)` produces `#t` if *`v`* is such an exact integer, `#f` otherwise. Two bytes strings are `equal?` if they are bytewise equal, and two byte strings are `eqv?` only if they are `eq?`.

MzScheme provides byte-string operations in parallel to the character-string operations:

- `(bytes?` *`v`*`)`
- `(bytes` *`byte`* $\cdots^1$`)`
- `(make-bytes` *`k`* [*`byte`*]`)`
- `(bytes-length` *`bytes`*`)`
- `(bytes-ref` *`bytes k`*`)`
- `(bytes-set!` *`bytes k byte`*`)`
- `(bytes-fill!` *`bytes byte`*`)`
- `(subbytes` *`bytes start-k`* [*`end-k`*]`)`
- `(bytes-append` *`bytes`* $\cdots^1$`)`
- `(bytes-copy` *`bytes`*`)`
- `(bytes-copy!` *`dest-bytes dest-start-k src-bytes`* [*`src-start-k src-end-k`*]`)`
- `(bytes->list` *`bytes`*`)`
- `(list->bytes` *`byte-list`*`)`
- `(bytes->immutable-bytes` *`bytes`*`)`
- `(bytes=?` *`bytes1 bytes2`* $\cdots^1$`)`
- `(bytes<?` *`bytes1 bytes2`* $\cdots^1$`)`
- `(bytes>?` *`bytes1 bytes2`* $\cdots^1$`)`

A byte-string constant is written like a string, but prefixed with `#` (with no space between `#` and the opening double-quote). A byte-string constant can contain escape sequences, as in `#"\n"`, just like strings; an `exn:fail:read` exception is raised if a "\u" sequence appears within a byte string and the given hexadecimal value is larger than 255.

Like character strings, byte strings generated by `read` are immutable, and when an immutable string is provided to a procedure like `bytes-set!`, the `exn:fail:contract` exception is raised.

The following procedures convert between byte strings and character strings:

- `(bytes->string/utf-8 `*`bytes`*` [`*`err-char start-k end-k`*`])` — produces a string by decoding the *`start-k`* to *`end-k`* substring of *`bytes`* as a UTF-8 encoding of Unicode code points. If *`err-char`* is provided and not `#f`, then it is used for bytes that fall in the range `#o200` to `#o377` but are not part of a valid encoding sequence. (This is consistent with reading characters from a port; see §11.1 for more details.) If *`err-char`* is `#f` or not provided, and if the *`start-k`* to *`end-k`* substring of *`bytes`* is not a valid UTF-8 encoding overall, then the `exn:fail:contract` exception is raised. If *`start-k`* or *`end-k`* are not provided, they default to `0` and `(bytes-length `*`bytes`*`)`, respectively.

- `(bytes->string/locale `*`bytes`*` [`*`err-char start-k end-k`*`])` — produces a string by decoding the *`start-k`* to *`end-k`* substring of *`bytes`* using the current locale's encoding (see also §1.2.2). If *`err-char`* is provided and not `#f`, it is used for each byte in *`bytes`* that is not part of a valid encoding; if *`err-char`* is `#f` or not provided, and if the *`start-k`* to *`end-k`* substring of *`bytes`* is not a valid encoding overall, then the `exn:fail:contract` exception is raised. If *`start-k`* or *`end-k`* are not provided, they default to `0` and `(bytes-length `*`bytes`*`)`, respectively.

- `(bytes->string/latin-1 `*`bytes`*` [`*`err-char start-k end-k`*`])` — produces a string by decoding the *`start-k`* to *`end-k`* substring of *`bytes`* as a Latin-1 encoding of Unicode code points; i.e., each byte is translated directly to a character using `integer->char`, so the decoding always succeeds.[5] The *`err-char`* argument is ignored, but for consistency with the other operations, it must be a character or `#f` if provided. If *`start-k`* or *`end-k`* are not provided, they default to `0` and `(bytes-length `*`bytes`*`)`, respectively.

- `(string->bytes/utf-8 `*`string`*` [`*`err-byte start-k end-k`*`])` — produces a byte string by ending the *`start-k`* to *`end-k`* substring of *`string`* via UTF-8 (always succeeding). The *`err-char`* argument is ignored, but for consistency with the other operations, it must be a byte or `#f` if provided. If *`start-k`* or *`end-k`* are not provided, they default to `0` and `(string-length `*`string`*`)`, respectively.

- `(string->bytes/locale `*`string`*` [`*`err-byte start-k end-k`*`])` — produces a string by encoding the *`start-k`* to *`end-k`* substring of *`string`* using the current locale's encoding (see also §1.2.2). If *`err-byte`* is provided and not `#f`, it is used for each character in *`string`* that cannot be encoded for the current locale; if *`err-byte`* is `#f` or not provided, and if the *`start-k`* to *`end-k`* substring of *`string`* cannot be encoded, then the `exn:fail:contract` exception is raised. If *`start-k`* or *`end-k`* are not provided, they default to `0` and `(string-length `*`string`*`)`, respectively.

- `(string->bytes/latin-1 `*`string`*` [`*`err-byte start-k end-k`*`])` — produces a string by encoding the *`start-k`* to *`end-k`* substring of *`string`* using Latin-1; i.e., each character is translated directly to a byte using `char->integer`. If *`err-byte`* is provided and not `#f`, it is used for each character in *`string`* whose value is greater than `255`;[6] if *`err-byte`* is `#f` or not provided, and if the *`start-k`* to *`end-k`* substring of *`string`* has a character with a value greater than `255`, then the `exn:fail:contract` exception is raised. If *`start-k`* or *`end-k`* are not provided, they default to `0` and `(string-length `*`string`*`)`, respectively.

- `(string-utf-8-length `*`string`*` [`*`start-k end-k`*`])` returns the length in bytes of the UTF-8 encoding of *`string`*'s substring from *`start-k`* to *`end-k`*, but without actually generating the encoded bytes. If *`start-k`* is not provided, it defaults to `0`, and *`end-k`* defaults to `(string-length `*`string`*`)`.

[5] See also the Latin-1 footnote of §1.2.3.
[6] See also the Latin-1 footnote of §1.2.3.

- `(bytes-utf-8-length` *`bytes`* [*`err-char start-k end-k`*]`)` returns the length in characters of the UTF-8 decoding of *`bytes`*'s substring from *`start-k`* to *`end-k`*, but without actually generating the decoded characters. If *`start-k`* is not provided, it defaults to `0`, and *`end-k`* defaults to `(bytes-length` *`bytes`*`)`. If *`err-char`* is `#f` and the substring is not a UTF-8 encoding overall, the result is `#f`. Otherwise, *`err-char`* is used to resolve decoding errors as in `bytes->string/utf-8`.
- `(bytes-utf-8-ref` *`bytes`* [*`skip-k err-char start-k end-k`*]`)` returns the *`skip-k`*th character in the UTF-8 decoding of *`bytes`*'s substring from *`start-k`* to *`end-k`*, but without actually generating the other decoded characters. If *`start-k`* is not provided, it defaults to `0`, and *`end-k`* defaults to `(bytes-length` *`bytes`*`)`. If the substring is not a UTF-8 encoding up to the *`skip-k`*th character (when *`err-char`* is `#f`), or if the substring decoding produces fewer than *`skip-k`* characters, the result is `#f`. If *`err-char`* is not `#f`, it is used to resolve decoding errors as in `bytes->string/utf-8`.
- `(bytes-utf-8-index` *`bytes`* [*`skip-k err-char start-k end-k`*]`)` returns the offset in bytes into *`bytes`* at which the *`skip-k`*th character's encoding starts in the UTF-8 decoding of *`bytes`*'s substring from *`start-k`* to *`end-k`* (but without actually generating the other decoded characters). If *`start-k`* is not provided, it defaults to `0`, and *`end-k`* defaults to `(bytes-length` *`bytes`*`)`. The result is relative to the start of *`bytes`*, not to *`start-k`*. If the substring is not a UTF-8 encoding up to the *`skip-k`*th character (when *`err-char`* is `#f`), or if the substring decoding produces fewer than *`skip-k`* characters, the result is `#f`. If *`err-char`* is not `#f`, it is used to resolve decoding errors as in `bytes->string/utf-8`.

A *string converter* can be used to convert directly from one byte-string encoding of characters to another byte-string encoding.

- `(bytes-open-converter` *`from-name-string to-name-string`*`)` — produces a string converter to go from the encoding named by *`from-name-string`* to the encoding named by *`to-name-string`*. If the requested conversion pair is not available, `#f` is returned instead of a converter.

  Certain encoding combinations are always available:

  - `(bytes-open-converter "UTF-8" "UTF-8")` — the identity conversion, except that encoding errors in the input lead to a decoding failure.
  - `(bytes-open-converter "UTF-8-permissive" "UTF-8")` — the identity conversion, except that any input byte that is not part of a valid encoding sequence is effectively replaced by `(char->integer #\?)`. (This handling of invalid sequences is consistent with the interpretation of port bytes streams into characters; see §11.1.)
  - `(bytes-open-converter "" "UTF-8")` — converts from the current locale's default encoding (see §1.2.2) to UTF-8.
  - `(bytes-open-converter "UTF-8" "")` — converts from UTF-8 to the current locale's default encoding (see §1.2.2).
  - `(bytes-open-converter "platform-UTF-8" "platform-UTF-16")` — converts UTF-8 to UTF-16 under Unix and Mac OS X, where each UTF-16 code unit is a sequence of two bytes ordered by the current platform's endianess. Under Windows, the input can include encodings that are not valid UTF-8, but which naturally extend the UTF-8 encoding to support unpaired surrogate code units, and the output is a sequence of UTF-16 code units (as little-endian byte pairs), potentially including unpaired surrogates.
  - `(bytes-open-converter "platform-UTF-8-permissive" "platform-UTF-16")` — like `(bytes-open-converter "platform-UTF-8" "platform-UTF-16")`, but an input byte that is not part of a valid UTF-8 encoding sequence (or valid for the unpaired-surrogate extension under Windows) is effectively replaced with `(char->integer #\?)`.
  - `(bytes-open-converter "platform-UTF-16" "platform-UTF-8")` — converts UTF-16 (bytes orderd by the current platform's endianness) to UTF-8 under Unix and Mac OS X. Under Windows, the input can include UTF-16 code units that are unpaired surrogates, and the corresponding output includes an encoding of each surrogate in a natural extension of UTF-8. Under Unix and Mac OS X, surrogates are assumed to be paired: a pair of bytes with the bits `#xD800` starts a surrogate pair, and

the `#x03FF` bits are used from the pair and following pair (independent of the value of the `#xDC00` bits). On all platforms, performance may be poor when decoding from an odd offset within an input byte string.

A newly opened byte converter is registered with the current custodian (see §9.2), so that the converter is closed when the custodian is shut down. A converter is not registered with a custodian (and does not need to be closed) if it is one of the guaranteed combinations not involving `""` under Unix, or if it is any of the guaranteed combinations (including `""`) under Windows and Mac OS X.

The set of available encodings and combinations varies by platform, depending on the **iconv** library that is installed. Under Windows, **iconv.dll** or **libiconv.dll** must be in the same directory as **libmzsch*VERS*.dll** (where ***VERS*** is a version number),[7] in the user's path, in the system directory, or in the current executable's directory at run time, and the DLL must either supply `_errno` or link to **msvcrt.dll** for `_errno`; otherwise, only the guaranteed combinations are available.

- `(bytes-close-converter` *`bytes-converter`*`)` — closes the given converter, so that it can no longer be used with *`bytes-convert`* or *`bytes-convert-end`*.

- `(bytes-convert` *`bytes-converter src-bytes`* [*`src-start-k src-end-k dest-bytes dest-start-k dest-end-k`*]`)` converts the bytes from *`src-start-k`* to *`src-end-k`* in *`src-bytes`*. If *`dest-bytes`* is supplied and not `#f`, the converted byte are written into *`dest-bytes`* from *`dest-start-k`* to *`dest-end-k`*. If *`dest-bytes`* is not supplied or is `#f`, then a newly allocated byte string holds the conversion results, and the size of the result byte string is no more than `(-` *`dest-end-k start-start-k`*`)`.

  If *`src-start-k`* or *`dest-start-k`* is not provided, it defaults to `0`. If *`src-end-k`* is not provided, it defaults to `(bytes-length` *`src-bytes`*. If *`src-end-k`* is not provided or is `#f`, then it defaults to `(bytes-length` *`dest-bytes`*`)` when *`dest-bytes`* is a byte string or to an arbitrarily large integer otherwise.

  The result of `bytes-convert` is three values:

  - *`result-bytes`* or *`dest-wrote-k`* — a byte string if *`dest-bytes`* is `#f` or not provided, or the number of bytes written into *`dest-bytes`* otherwise.
  - *`src-read-k`* — the number of bytes successfully converted from *`src-bytes`*.
  - `'complete`, `'continues`, `'aborts`, or `'error` — indicates how conversion terminated.
    * `'complete`: The entire input was processed, and *`src-read-k`* will be equal to `(-` *`src-end-k src-start-k`*`)`.
    * `'continues`: Conversion stopped due to the limit on the result size or the space in *`dest-bytes`*; in this case, fewer than `(-` *`dest-end-k dest-start-k`*`)` bytes may be returned if more space is needed to process the next complete encoding sequence in *`src-bytes`*.
    * `'aborts`: The input stopped part-way through an encoding sequence, and more input bytes are necessary to continue. For example, if the last byte of input is `#o303` for a `"UTF-8-permissive"` decoding, the result is `'aborts`, because another byte is needed to determine how to use the `#o303` byte.
    * `'error`: The bytes starting at `(+` *`src-start-k src-read-k`*`)` bytes in *`src-bytes`* do not form a legal encoding sequence. This result is never produced for some encodings, where all byte sequences are valid encodings. For example, since `"UTF-8-permissive"` handles an invalid UTF-8 sequence by dropping characters or generating “?”, every byte sequence is effectively valid.

  Applying a converter accumulates state in the converter (even when the third result of `bytes-convert` is `'complete`). This state can affect both further processing of input and further generation of output, but only for conversions that involve “shift sequences” to change modes within a stream. To terminate an input sequence and reset the converter, use `bytes-convert-end`.

- `(bytes-convert-end` *`bytes-converter`* [*`dest-bytes dest-start-k dest-end-k`*]`)` — like `bytes-convert`, but instead of converting bytes, this procedure generates an ending sequence for the conversion (sometimes called a “shift sequence”), if any. Few encodings use shift sequences, so this function

[7] In PLT's software distributions for Windows, a suitable **iconv.dll** is included with **libmzsch*VERS*.dll**.

will succeed with no output for most encodings. In any case, successful output of a (possibly empty) shift sequence resets the converter to its initial state.

The result of `bytes-convert-end` is two values:

- *`result-bytes`* or *`dest-wrote-k`* — a byte string if *`dest-bytes`* is `#f` or not provided, or the number of bytes written into *`dest-bytes`* otherwise.
- `'complete` or `'continues` — indicates whether conversion completed. If `'complete`, then an entire ending sequence was produced. If `'continues`, then the conversion could not complete due to the limit on the result size or the space in *`dest-bytes`*, and the first result is either an empty byte string or `0`.

- `(bytes-converter?` *`v`*`)` returns `#t` if *`v`* is a byte converter produced by `bytes-open-converter`, `#f` otherwise.

- `(locale-string-encoding)` returns a string for the current locale's encoding (i.e., the encoding normally identified by `""`). See also `system-language+country` in §15.5.

3.7 Symbols

For information about symbol parsing and printing, see §11.2.4 and §11.2.5, respectively.

MzScheme provides two ways of generating an *uninterned symbol*, i.e., a symbol that is not `eq?`, `eqv?`, or `equal?` to any other symbol, although it may print the same as another symbol:

- `(string->uninterned-symbol` *`string`*`)` is like `(string->symbol` *`string`*`)`, but the resulting symbol is a new uninterned symbol. Calling `string->uninterned-symbol` twice with the same *`string`* returns two distinct symbols.

- `(gensym [`*`symbol/string`*`])` creates an uninterned symbol with an automatically-generated name. The optional *`symbol/string`* argument is a prefix symbol or string.

Regular (interned) symbols are only weakly held by the internal symbol table. This weakness can never affect the result of an `eq?`, `eqv?`, or `equal?` test, but a symbol may disappear when placed into a weak box (see §13.1) used as the key in a weak hash table (see §3.14), or used as an ephemeron key (see §13.2).

3.8 Keywords

A symbol-like datum that starts with a hash and colon ("#:") is parsed as a *keyword* constant. Keywords behave like symbols — two keywords are `eq?` if and only if they print the same — but they are a distinct set of values.

- `(keyword?` *`v`*`)` returns `#t` if *`v`* is a keyword, `#f` otherwise.

- `(keyword->string` *`keyword`*`)` returns a string for the `display`ed form of *`keyword`*, not including the leading `#:`.

- `(string->keyword` *`string`*`)` returns a keyword whose `display`ed form is the same as that of *`string`*, but with a leading `#:`.

Like symbols, keywords are only weakly held by the internal keyword table; see §3.7 for more information.

3.9 Vectors

When a vector is created with `make-vector` without a fill value, it is initialized with `0` in all positions. A vector can be immutable, such as a vector returned by `syntax-e`, but vectors generated by `read` are mutable. (See also `immutable?` in §3.10.)

`(vector->immutable-vector` *`vec`*`)` returns an immutable vector with the same content as *`vec`*, and it returns *`vec`* itself if *`vec`* is immutable. (See also `immutable?` in §3.10.)

`(vector-immutable` *`v`* $\cdots^1$`)` is like `(`*`vector v`* $\cdots^1$`)` except that the resulting vector is immutable. (See also `immutable?` in §3.10.)

3.10 Lists

A cons cell can be mutable or immutable. When an immutable cons cell is provided to a procedure like `set-cdr!`, the `exn:fail:contract` exception is raised. Cons cells generated by `read` are always mutable.

The global variable `null` is bound to the empty list.

`(reverse!` *`list`*`)` is the same as `(reverse` *`list`*`)`, but *`list`* is destructively reversed using `set-cdr!` (i.e., each cons cell in *`list`* is mutated).

`(append!` *`list`* $\cdots^1$`)` is like `(append` *`list`*`)`, but it destructively appends the *`list`*s (i.e., except for the last *`list`*, the last cons cell of each *`list`* is mutated to append the lists; empty lists are essentially dropped).

`(list*` *`v`* $\cdots^1$`)` is similar to `(list` *`v`* $\cdots^1$`)` but the last argument is used directly as the `cdr` of the last pair constructed for the list:

```
(list* 1 2 3 4) ; ⇒ '(1 2 3 . 4)
```

`(cons-immutable` *`v1 v2`*`)` returns an immutable pair whose `car` is *`v1`* and `cdr` is *`v2`*.

`(list-immutable` *`v`* $\cdots^1$`)` is like `(list` *`v`* $\cdots^1$`)`, but using immutable pairs.

`(list*-immutable` *`v`* $\cdots^1$`)` is like `(list*` *`v`* $\cdots^1$`)`, but using immutable pairs.

`(immutable?` *`v`*`)` returns `#t` if *`v`* is an immutable cons cell, string, vector, box, or hash table, `#f` otherwise.

The `list-ref` and `list-tail` procedures accept an improper list as a first argument. If either procedure is applied to an improper list and an index that would require taking the `car` or `cdr` of a non-cons-cell, the `exn:fail:contract` exception is raised.

The `member`, `memv`, and `memq` procedures accept an improper list as a second argument. If the membership search reaches the improper tail, the `exn:fail:contract` exception is raised.

The `assoc`, `assv`, and `assq` procedures accept an improperly formed association list as a second argument. If the association search reaches an improper list tail or a list element that is not a pair, the `exn:fail:contract` exception is raised.

3.11 Boxes

MzScheme provides *boxes*, which are records that have a single field:

- `(box` *`v`*`)` returns a new mutable box that contains *`v`*.

- `(box-immutable` *`v`*`)` returns a new immutable box that contains *`v`*.
- `(unbox` *`box`*`)` returns the content of *`box`*. For any *`v`*, `(unbox (box` *`v`*`))` returns *`v`*.
- `(set-box!` *`mutable-box v`*`)` sets the content of *`mutable-box`* to *`v`*.
- `(box?` *`v`*`)` returns `#t` if *`v`* is a box, `#f` otherwise.

Two boxes are `equal?` if the contents of the boxes are `equal?`.

A box returned by `syntax-e` (see §12.2.2) is immutable; if `set-box!` is applied to such a box, the `exn:fail:contract` exception is raised. A box produced by `read` (via `#&`) is mutable. (See also `immutable?` in §3.10.)

3.12 Procedures

See §4.6 for information on defining new procedure types.

3.12.1 Arity

MzScheme's `procedure-arity` procedure returns the input arity of a procedure:

- `(procedure-arity` *`proc`*`)` returns information about the number of arguments accepted by the procedure *`proc`*. The result *`a`* is either:
  - an exact non-negative integer ⇒ the procedure always takes exactly *`a`* arguments;
  - an `arity-at-least`[8] instance ⇒ the procedure takes `(arity-at-least-value` *`a`*`)` or more arguments; or
  - a list containing integers and `arity-at-least` instances ⇒ the procedure takes any number of arguments that can match one of the arities in the list.
- `(procedure-arity-includes?` *`proc k`*`)` returns `#t` if the procedure can accept *`n`* arguments (where *`k`* is an exact, non-negative integer), `#f` otherwise.

Examples:

```
(procedure-arity cons) ; ⇒ 2
(procedure-arity list) ; ⇒ #<struct:arity-at-least>
(arity-at-least? (procedure-arity list)) ; ⇒ #t
(arity-at-least-value (procedure-arity list)) ; ⇒ 0
(arity-at-least-value (procedure-arity (lambda (x . y) x))) ; ⇒ 1
(procedure-arity (case-lambda [(x) 0] [(x y) 1])) ; ⇒ '(1 2)
(procedure-arity-includes? cons 2) ; ⇒ #t
(procedure-arity-includes? display 3) ; ⇒ #f
```

When compiling a `lambda` or `case-lambda` expression, MzScheme looks for a `'method-arity-error` property attached to the expression (see §12.6.2). If it is present with a true value, and if no case of the procedure accepts zero arguments, then the procedure is marked so that an `exn:fail:contract:arity` exception involving the procedure will hide the first argument, if one was provided. (Hiding the first argument is useful when the procedure implements a method, where the first argument is implicit in the original source). The property affects only the format of `exn:fail:contract:arity` exceptions, not the result of `procedure-arity`.

[8]The `arity-at-least` structure type is transparent to all inspectors (see §4.5).

3.12.2 Primitives

A *primitive procedure* is a built-in procedure that is implemented in low-level language. Not all built-in procedures are primitives, but almost all R^5RS procedures are primitives, as are most of the procedures described in this manual.

- `(primitive? v)` returns `#t` if *v* is a primitive procedure or `#f` otherwise.
- `(primitive-result-arity prim-proc)` returns the arity of the result of the primitive procedure *prim-proc* (as opposed to the procedure's input arity as returned by `arity`; see §3.12.1). For most primitives, this procedure returns `1`, since most primitives return a single value when applied. For information about arity values, see §3.12.1.
- `(primitive-closure? v)` returns `#t` if *v* is internally implemented as a primitive closure rather than a simple primitive procedure, `#f` otherwise. This information is intended for use by the **mzc** compiler.

3.12.3 Procedure Names

See §6.2.3 for information about the names of primitives, and the names inferred for `lambda` and `case-lambda` procedures.

3.12.4 Closure Equality

`(procedure-closure-contents-eq? proc1, proc2)` return `#t` if the procedures *proc1* and *proc2* refer to the same code closed over the same values, where each value is compared with `eq?`.

Inlining and other compiler optimizations limit the usefulness of this procedure, because code can be duplicated or merged. Since the amount of duplication from inlining is limited, however, `procedure-closure-contents-eq?` is useful for some caching purposes.

Example:

```
(let ([f #f])
  ;; Using set! likely prevents inlining:
  (set! f (lambda (x) (lambda () x)))
  (procedure-closure-contents-eq? (f 'a) (f 'a)) ; ⇒ #t, probably
  (procedure-closure-contents-eq? (f 'a) (f 'b))) ; ⇒ #f, definitely

(let ([f (lambda (x) (lambda () x))])
  (procedure-closure-contents-eq? (f 'a) (f 'a)))
;; ⇒ #f, probably, because inling likely duplicates f's body
```

3.13 Promises

The `force` procedure can only be applied to values returned by `delay`, and promises are never implicitly `force`d.

`(promise? v)` returns `#t` if *v* is a promise created by `delay`, `#f` otherwise.

3.14 Hash Tables

`(make-hash-table [flag-symbol flag-symbol])` creates and returns a new hash table. If provided, each *flag-symbol* must one of the following:

- `'weak` — creates a hash table with weakly-held keys (see §13.1).
- `'equal` — creates a hash table that compares keys using `equal?` instead of `eq?` (needed, for example, when using strings as keys).

By default, key comparisons use `eq?`. If the second *`flag-symbol`* is redundant, the `exn:fail:contract` exception is raised.

Two hash tables are `equal?` if they are created with the same flags, and if they map the same keys to `equal?` values (where "same key" means either `eq?` or `equal?`, depending on the way the hash table compares keys).

`(make-immutable-hash-table` *`assoc-list`* [*`flag-symbol`*]`)` creates an immutable hash table. (See also `immutable?` in §3.10.) The *`assoc-list`* must be a list of pairs, where the `car` of each pair is a key, and the `cdr` is the corresponding value. The mappings are added to the table in the order that they appear in *`assoc-list`*, so later mappings can hide earlier mappings. If the optional *`flag-symbol`* argument is provided, it must be `'equal`, and the created hash table compares keys with `equal?`; otherwise, the created table compares keys with `eq?`.

`(hash-table?` *`v`* [*`flag-symbol flag-symbol`*]`)` returns `#t` if *`v`* was created by `make-hash-table` or *`make-immutable-hash-table`* with the given *`flag-symbol`*s (or more), `#f` otherwise. Each provided *`flag-symbol`* must be a distinct flag supported by `make-hash-table`; if the second *`flag-symbol`* is redundant, the `exn:fail:contract` exception is raised.

`(hash-table-put!` *`hash-table key-v v`*`)` maps *`key-v`* to *`v`* in *`hash-table`*, overwriting any existing mapping for *`key-v`*. If *`hash-table`* is immutable, the `exn:fail:contract` exception is raised.

`(hash-table-get` *`hash-table key-v`* [*`failure-thunk-or-value`*]`)` returns the value for *`key-v`* in *`hash-table`*. If no value is found for *`key-v`*, then *`failure-thunk-or-value`* determines the result: if *`failure-thunk-or-value`* is not provided, the `exn:fail:contract` exception is raised; if *`failure-thunk-or-value`* is a procedure, it is called (through a tail call) with no arguments to produce the result; finally, if *`failure-thunk-or-value`* is provided and not a procedure, it is used as the result.

`(hash-table-remove!` *`hash-table key-v`*`)` removes the value mapping for *`key-v`* if it exists in *`hash-table`*. If *`hash-table`* is immutable, the `exn:fail:contract` exception is raised.

`(hash-table-map` *`hash-table proc`*`)` applies the procedure *`proc`* to each element in *`hash-table`*, accumulating the results into a list. The procedure *`proc`* must take two arguments: a key and its value. See the caveat below about concurrent modification.

`(hash-table-for-each` *`hash-table proc`*`)` applies the procedure *`proc`* to each element in *`hash-table`* (for the side-effects of *`proc`*) and returns void. The procedure *`proc`* must take two arguments: a key and its value. See the caveat below about concurrent modification.

`(hash-table-count` *`hash-table`*`)` returns the number of keys mapped by *`hash-table`*. If *`hash-table`* is not created with `'weak`, then the result is computed in constant time and atomically. If *`hash-table`* is created with `'weak`, see the caveat below about concurrent modification.

`(hash-table-copy` *`hash-table`*`)` returns a mutable hash table with the same mappings, same key-comparison mode, and same key-holding strength as *`hash-table`*.

`(eq-hash-code` *`v`*`)` returns an exact integer; for any two `eq?` values, the returned integer is the same. Furthermore, for the result integer *`k`* and any other exact integer *`j`*, `(=` *`k j`*`)` implies `(eq?` *`k j`*`)`.

`(equal-hash-code` *`v`*`)` returns an exact integer; for any two `equal?` values, the returned integer is the same. Furthermore, for the result integer *`k`* and any other exact integer *`j`*, `(=` *`k j`*`)` implies `(eq?` *`k j`*`)`. If *`v`* contains a cycle through pairs, vectors, boxes, and inspectable structure fields, then `equal-hash-code` applied to *`v`* will loop indefinitely.

Caveat concerning concurrent modification: A hash table can be manipulated with `hash-table-get`, `hash-table-put!`, and `hash-table-remove!` concurrently by multiple threads, and the operations are protected by a table-specific semaphore as needed. A few caveats apply, however:

- If a thread is terminated while applying `hash-table-get`, `hash-table-put!`, or `hash-table-remove!` to a hash table that uses `equal?` comparisons, all current and future operations on the hash table block indefinitely.
- The `hash-table-map`, `hash-table-for-each`, and `hash-table-count` procedures do not use the table's semaphore. Consequently, if a hash table is extended with new keys by another thread while a map, for-each, or count is in process, arbitrary key–value pairs can be dropped or duplicated in the map or for-each. Similarly, if a map or for-each procedure itself extends the table, arbitrary key–value pairs can be dropped or duplicated. However, key mappings can be deleted or remapped by any thread with no adverse affects (i.e., the change does not affect a traversal if the key has been seen already, otherwise the traversal skips a deleted key or uses the remapped key's new value).

Caveat concerning mutable keys: If a key into an `equal?`-based hash table is mutated (e.g., a key string is modified with `string-set!`), then the hash table's behavior for put and get operations becomes unpredictable.

4. Structures

A *structure type* is a record datatype composing a number of *fields*. A *structure*, an instance of a structure type, is a first-class value that contains a value for each field of the structure type. A structure instance is created with a type-specific constructor procedure, and its field values are accessed and changed with type-specific selector and setter procedures. In addition, each structure type has a predicate procedure that answers `#t` for instances of the structure type and `#f` for any other value.

4.1 Defining Structure Types

A new structure type can be created with one of four `define-struct` forms:

```
(define-struct s (field ···) [inspector-expr])
(define-struct (s t) (field ···) [inspector-expr])
```

where *s*, *t*, and each *field* are identifiers. The latter form is described in §4.2. The optional *inspector-expr*, which should produce an inspector or `#f`, is explained in §4.5.

A `define-struct` expression with *n* *field*s defines $4+2n$ names:

- `struct:`*s*, a *structure type descriptor* value that represents the new datatype. This value is rarely used directly.
- `make-`*s*, a constructor procedure that takes *n* arguments and returns a new structure value.
- *s*`?`, a predicate procedure that returns `#t` for a value constructed by `make-`*s* (or the constructor for a subtype; see §4.2) and `#f` for any other value.
- *s*`-`*field*, for each *field*, an accessor procedure that takes a structure value and extracts the value for *field*.
- `set-`*s*`-`*field*`!`, for each *field*, a mutator procedure that takes a structure and a new field value. The field value in the structure is destructively updated with the new value, and void is returned.
- *s*, a syntax binding that encapsulates information about the structure type declaration. This binding is used to define subtypes (see §4.2). It also works with the `shared` and `match` forms (see Chapter 44 and Chapter 27 of *PLT MzLib: Libraries Manual*). For detailed information about the expansion-time information stored in *s*, see §12.6.4.

Each time a `define-struct` expression is evaluated, a new structure type is created with distinct constructor, predicate, accessor, and mutator procedures. If the same `define-struct` expression is evaluated twice, instances created by the constructor returned by the first evaluation will answer `#f` to the predicate returned by the second evaluation.

Examples:

```
(define-struct cons-cell (car cdr))
(define x (make-cons-cell 1 2))
```

```
(cons-cell? x) ; ⇒ #t
(cons-cell-car x) ; ⇒ 1
(set-cons-cell-car! x 5)
(cons-cell-car x) ; ⇒ 5

(define orig-cons-cell? cons-cell?)
(define-struct cons-cell (car cdr))
(define y (make-cons-cell 1 2))
(cons-cell? y) ; ⇒ #t
(cons-cell? x) ; ⇒ #f, cons-cell? now checks for a different type
(orig-cons-cell? x) ; ⇒ #t
(orig-cons-cell? y) ; ⇒ #f
```

The `let-struct` form binds structure identifiers in a lexical scope; it does not support an *`inspector-expr`*.

```
(let-struct s (field ···)
 body-expr ···¹)
(let-struct (s t) (field ···)
 body-expr ···¹)
```

4.2 Creating Subtypes

The latter `define-struct` form shown in §4.1 creates a new structure type that is a *structure subtype* of an existing base structure type. An instance of a structure subtype can always be used as an instance of the base structure type, but the subtype gets its own predicate procedure and may have its own fields in addition to the fields of the base type.

The *`t`* identifier in a subtyping `define-struct` form must be bound to syntax describing a structure type declaration. Normally, it is the name of a structure type previously declared with `define-struct`. The information associated with *`t`* is used to access the base structure type for the new subtype.

A structure subtype "inherits" the fields of its base type. If the base type has *`m`* fields, and if *`n`* fields are specified in the subtyping `define-struct` expression, then the resulting structure type has $m+n$ fields. Consequently, $m+n$ field values must be provided to the subtype's constructor procedure. Values for the first *`m`* fields of a subtype instance are accessed with selector procedures for the original base type, and the last *`n`* are accessed with subtype-specific selectors. Subtype-specific accessors and mutators for the first *`m`* fields are not created.

Examples:

```
(define-struct cons-cell (car cdr))
(define x (make-cons-cell 1 2))
(define-struct (tagged-cons-cell cons-cell) (tag))
(define z (make-tagged-cons-cell 3 4 't))
(cons-cell? z) ; ⇒ #t
(tagged-cons-cell? z) ; ⇒ #t
(tagged-cons-cell? x) ; ⇒ #f
(cons-cell-car z) ; ⇒ 3
(tagged-cons-cell-tag z) ; ⇒ 't
```

4.3 Structure Types with Automatic Fields, Immutable Fields, and Properties

The `make-struct-type` procedure creates a new structure type in the same way as the `define-struct` form of §4.1, but provides a more general interface. In particular, the `make-struct-type` procedure supports structure type properties.

- `(make-struct-type` *`name-symbol super-struct-type init-field-k auto-field-k`* [*`auto-v prop-value-list inspector-or-false proc-spec immutable-k-list guard-proc`*]`)` creates a new structure type. The *`name-symbol`* argument is used as the type name. If *`super-struct-type`* is not `#f`, the new type is a subtype of the corresponding structure type, as described in §4.2.

  The new structure type has *`init-field-k`* + *`auto-field-k`* fields (in addition to any fields from *`super-struct-type`*), but only *`init-field-k`* constructor arguments (in addition to any constructor arguments from *`super-struct-type`*). The remaining fields are initialized with *`auto-v`*, which defaults to `#f`.

  The *`prop-value-list`* argument is a list of pairs, where the `car` of each pair is a structure type property descriptor, and the `cdr` is an arbitrary value. The default is `null`. See §4.4 for more information about properties.

  The *`inspector-or-false`* argument controls access to debugging information about the structure type and its instances; see §4.5 for more information.

  The *`proc-spec`* argument can be `#f`, an exact non-negative integer, or a procedure. The default is `#f`. If an integer or procedure is provided, instances of the structure type act as procedures. See §4.6 for further information. Providing a non-`#f` value for *`proc-spec`* is the same as pairing the value with `prop:procedure` in *`prop-value-list`*, plus including *`proc-spec`* in *`immutable-k-list`* when *`proc-spec`* is an integer.

  The *`immutable-k-list`* argument provides a list of exact, non-negative integers that identify immutable field positions. Each element in the list should be unique, otherwise `exn:fail:contract` exception is raised. Each element should also fall in the range `0` (inclusive) and *`init-field-k`* (exclusive), otherwise `exn:fail:contract` exception is raised.

  The *`guard-proc`* argument is either a procedure of $n+1$ arguments or `#f`, where *`n`* is the number of arguments for the new structure type's constructor (i.e., *`init-field-k`* plus constructor arguments implied by *`super-struct-type`*, if any). If *`guard-proc`* is a procedure, then the procedure is called whenever an instance of the type is constructed, or whenever an instance of a subtype is created. The arguments to *`guard-proc`* are the values provided for the structure's first *`n`* fields, followed by the name of the instantiated structure type (which is *`name-symbol`*, unless a subtype is instantiated). The *`guard-proc`* result should be *`n`* values, which become the actual value for the structure's fields. The *`guard-proc`* can raise an exception to prevent creation of a structure with the given field values. If a structure subtype has its own guard, the subtype guard is applied first, and the first *`n`* values produced by the subtype's guard procedure become the first *`n`* arguments to *`guard-proc`*.

  The result of *`make-struct-type`* is five values, which are similar to the values produced by `define-struct` (see §4.1):

  - a structure type descriptor,
  - a constructor procedure,
  - a predicate procedure,
  - an accessor procedure, which consumes a structure and a field index between 0 (inclusive) and *`init-field-k`* + *`auto-field-k`* (exclusive), and
  - a mutator procedure, which consumes a structure, a field index, and a field value.

Unlike `define-struct`, `make-struct-type` returns a single accessor procedure and a single mutator procedure for all fields. The `make-struct-field-accessor` and `make-struct-field-mutator` procedures convert a type-specific accessor or mutator returned by `make-struct-type` into a field-specific accessor or mutator:

- `(make-struct-field-accessor` *`accessor-proc field-pos-k field-name-symbol`*`)` returns a field accessor that is equivalent to

```
(lambda (s) (accessor-proc s field-pos-k))
```

The *accessor-proc* must be an accessor returned by `make-struct-type`. The name of the resulting procedure for debugging purposes is derived from *field-name-symbol* and the name of *accessor-proc*'s structure type.

- `(make-struct-field-mutator mutator-proc field-pos-k field-name-symbol)` returns a field mutator that is equivalent to

  ```
  (lambda (s v) (mutator-proc s field-pos-k v))
  ```

  The *mutator-proc* must be a mutator returned by `make-struct-type`. The name of the resulting procedure for debugging purposes is derived from *field-name-symbol* and the name of *mutator-proc*'s structure type.

Examples:

```
(define-values (struct:a make-a a? a-ref a-set!)
  (make-struct-type 'a #f 2 1 'uninitialized))
(define an-a (make-a 'x 'y))
(a-ref an-a 1) ; ⇒ 'y
(a-ref an-a 2) ; ⇒ 'uninitialized
(define a-first (make-struct-field-accessor a-ref 0))
(a-first an-a) ; ⇒ 'x

(define-values (struct:b make-b b? b-ref b-set!)
  (make-struct-type 'b struct:a 1 2 'b-uninitialized))
(define a-b (make-b 'x 'y 'z))
(a-ref a-b 1) ; ⇒ 'y
(a-ref a-b 2) ; ⇒ 'uninitialized
(b-ref a-b 0) ; ⇒ z
(b-ref a-b 1) ; ⇒ 'b-uninitialized
(b-ref a-b 2) ; ⇒ 'b-uninitialized

(define-values (struct:c make-c c? c-ref c-set!)
  (make-struct-type 'c struct:b 0 0 #f null (make-inspector) #f null
                    ;; Guard checks for a number, and makes it inexact
                    (lambda (a1 a2 b1 name)
                      (unless (number? a2)
                        (error (string->symbol (format "make-~a" name))
                               "second field must be a number"))
                      (values a1 (exact->inexact a2) b1))))
(make-c 'x 'y 'z) ; ⇒ error: "make-c: second field must be a number"
(define a-c (make-c 'x 2 'z))
(a-ref a-c 1) ; ⇒ 2.0
```

4.4 Structure Type Properties

A *structure type property* allows per-type information to be associated with a structure type (as opposed to per-instance information associated with a structure value). A property value is associated with a structure type through the `make-struct-type` procedure (see §4.3). Subtypes inherit the property values of their parent types, and subtypes can override an inherited property value with a new value. (See §11.2.10 for a built-in property that controls how struct values are printed.)

`(make-struct-type-property name-symbol [guard-proc])` creates a new structure type property and returns three values:

- a structure property type descriptor, for use with `make-struct-type`;
- a predicate procedure, which takes an arbitrary value and returns `#t` if the value is a descriptor or instance of a structure type that has a value for the property, `#f` otherwise;
- an accessor procedure, which returns the value associated with structure type given its descriptor or one of its instances; if the structure type does not have a value for the property, or if any other kind of value is provided, the `exn:fail:contract` exception is raised.

If the optional *`guard-proc`* is supplied, it is called by `make-struct-type` before attaching the property to a new structure type. The *`guard-proc`* must accept two arguments: a value for the property supplied to `make-struct-type`, and a list containing information about the new structure type. The list contains the values that `struct-type-info` would return for the new structure type if it skipped the immediate current-inspector control check (but not the check for exposing an ancestor structure type, if any; see §4.5).

The result of calling *`guard-proc`* is associated with the property in the target structure type, instead of the value supplied to `make-struct-type`. To reject a property association (e.g., because the value supplied to `make-struct-type` is inappropriate for the property), the guard can raise an exception. Such an exception prevents `make-struct-type` from returning a structure type descriptor.

`(struct-type-property? v)` returns `#t` if *v* is a structure type property descriptor value, `#f` otherwise.

Examples:

```
(define-values (prop:p p? p-ref) (make-struct-type-property 'p))

(define-values (struct:a make-a a? a-ref a-set!)
  (make-struct-type 'a #f 2 1 'uninitialized (list (cons prop:p 8))))
(p? struct:a) ; ⇒ #t
(p? 13) ; ⇒ #f
(define an-a (make-a 'x 'y))
(p? an-a) ; ⇒ #t
(p-ref an-a) ; ⇒ 8

(define-values (struct:b make-b b? b-ref b-set!)
  (make-struct-type 'b #f 0 0 #f))
(p? struct:b) ; ⇒ #f
```

4.5 Structure Inspectors

An *inspector* provides access to structure fields and structure type information without the normal field accessors and mutators. (Inspectors are also used to control access to module bindings; see §9.4.) Inspectors are primarily intended for use by debuggers.

When a structure type is created, an inspector can be supplied. The given inspector is not the one that will control the new structure type; instead, the given inspector's parent will control the type. By using the parent of the given inspector, the structure type remains opaque to "peer" code that cannot access the parent inspector. Thus, an expression of the form

```
(define-struct s (field ···))
```

creates a structure type whose instances are opaque to peer code. In contrast, the following idiom creates a structure type that is transparent to peer code, because the supplied inspector is a newly created child of the current inspector:

```
(define-struct s (field ···) (make-inspector))
```

Instead of supplying an inspector, `#f` can be provided, which makes the structure transparent to all code. Thus,

```
(define-struct s (field ···) #f)
```

creates a structure type that is transparent to all code.

The `current-inspector` parameter determines a default inspector argument for new structure types. An alternate inspector can be provided though the optional *`inspector-expr`* expression of the `define-struct` form (see §4.1), as shown above, or through an optional *`inspector`* argument to `make-struct-type` (see §4.3).

`(make-inspector [`*`inspector`*`])` returns a new inspector that is a subinspector of *`inspector`*. If *`inspector`* is not provided, the new inspector is a subinspector of the current inspector. Any structure type controlled by the new inspector is also controlled by its ancestor inspectors, but no other inspectors.

`(inspector? `*`v`*`)` returns `#t` if *`v`* is an inspector, `#f` otherwise.

The `struct-info` and `struct-type-info` procedures provide inspector-based access to structure and structure type information:

- `(struct-info `*`v`*`)` returns two values:
  - *`struct-type`*: a structure type descriptor or `#f`; the result is a structure type descriptor of the most specific type for which *`v`* is an instance, and for which the current inspector has control, or the result is `#f` if the current inspector does not control any structure type for which the *`struct`* is an instance.
  - *`skipped?`*: `#f` if the first result corresponds to the most specific structure type of *`v`*, `#t` otherwise.

- `(struct-type-info `*`struct-type`*`)` returns eight values that provide information about the structure type descriptor *`struct-type`*, assuming that the type is controlled by the current inspector:
  - *`name-symbol`*: the structure type's name as a symbol;
  - *`init-field-k`*: the number of fields defined by the structure type provided to the constructor procedure (not counting fields created by its ancestor types);
  - *`auto-field-k`*: the number of fields defined by the structure type without a counterpart in the constructor procedure (not counting fields created by its ancestor types);
  - *`accessor-proc`*: an accessor procedure for the structure type, like the one returned by `make-struct-type`;
  - *`mutator-proc`*: a mutator procedure for the structure type, like the one returned by `make-struct-type`;
  - *`immutable-k-list`*: an immutable list of exact non-negative integers that correspond to immutable fields for the structure type;
  - *`super-struct-type`*: a structure type descriptor for the most specific ancestor of the type that is controlled by the current inspector, or `#f` if no ancestor is controlled by the current inspector;
  - *`skipped?`*: `#f` if the seventh result is the most specific ancestor type or if the type has no supertype, `#t` otherwise.

  If the type for *`struct-type`* is not controlled by the current inspector, the `exn:fail:contract` exception is raised.

- `(struct-type-make-constructor `*`struct-type`*`)` returns a constructor procedure to create instances of the type for *`struct-type`*. If the type for *`struct-type`* is not controlled by the current inspector, the `exn:fail:contract` exception is raised.

- `(struct-type-make-predicate `*`struct-type`*`)` returns a predicate procedure to recognize instances of the type for *`struct-type`*. If the type for *`struct-type`* is not controlled by the current inspector, the `exn:fail:contract` exception is raised.

4.6 Structures as Procedures

If the `prop:procedure` property is specified for a structure type (see §4.3), instances of the new structure type are procedures. In particular, when `procedure?` is applied to the instance, the result will be `#t`. When an instance is used in the function position of an application expression, a procedure is extracted from the instance and used to complete the procedure call.

If the `prop:procedure` property value is an integer, it designates a field within the structure that should contain a procedure. The integer must be between 0 (inclusive) and the third argument to `make-struct-type`, *`init-field-k`* (exclusive). The designated field must also be specified as immutable, so that after an instance of the structure is created, its procedure cannot be changed. (Otherwise, the arity and name of the instance could change, and such mutations are generally not allowed for procedures.) When the instance is used as the procedure in an application expression, the value of the designated field in the instance is used to complete the procedure call.[1] That procedure receives all of the arguments from the application expression. The procedure's name (see §6.2.3) and arity (see §3.12.1) are also used for the name and arity of the structure. If the value in the designated field is not a procedure, then the instance behaves like `(case-lambda)` (i.e., a procedure which does not accept any number of arguments).

Providing an integer *`proc-spec`* argument to `make-struct-type` is the same as both supplying the value with the `prop:procedure` property and designating the field as immutable (so that a property binding or immutable designation is redundant and disallowed).

Example:

```
(define-values (struct:ap make-annotated-proc annotated-proc? ap-ref ap-set!)
  (make-struct-type 'annotated-proc #f 2 0 #f null #f 0))
(define (proc-annotation p) (ap-ref p 1))
(define plus1 (make-annotated-proc
                (lambda (x) (+ x 1))
                "adds 1 to its argument"))
(procedure? plus1) ; ⇒ #t
(annotated-proc? plus1) ; ⇒ #t
(plus1 10) ; ⇒ 11
(proc-annotation plus1) ; ⇒ "adds 1 to its argument"
```

When the `prop:procedure` value is a procedure, it should accept at least one argument. When an instance of the structure is used in an application expression, the property-value procedure is called with the instance as the first argument. The remaining arguments to the property-value procedure are the arguments from the application expression. Thus, if the application expression contained five arguments, the property-value procedure is called with six arguments. The name of the instance (see §6.2.3) is unaffected by the property-value procedure, but the instance's arity is determined by subtracting one from every possible argument count of the property-value procedure. If the property-value procedure cannot accept at least one argument, then the instance behaves like `(case-lambda)`.

Providing a procedure *`proc-spec`* argument to `make-struct-type` is the same as supplying the value with the `prop:procedure` property (so that a specific property binding is disallowed).

```
(define-values (struct:fish make-fish fish? fish-ref fish-set!)
  (make-struct-type 'fish #f 2 0 #f null #f
                    (lambda (f n) (fish-set! f 0 (+ n (fish-ref f 0))))))
(define (fish-weight f) (fish-ref f 0))
(define (fish-color f) (fish-ref f 1))
(define wanda (make-fish 12 'red))
(fish? wanda) ; ⇒ #t
```

[1]This procedure can be another structure that acts as a procedure. The immutability of procedure fields disallows cycles in the procedure graph, so that the procedure call will eventually continue with a non-structure procedure.

```
(procedure? wanda) ; ⇒ #t
(fish-weight wanda) ; ⇒ 12
(for-each wanda '(1 2 3))
(fish-weight wanda) ; ⇒ 18
```

If a structure type generates procedure instances, then subtypes of the type also generate procedure instances. The instances behave the same as instances of the original type. When a `prop:procedure` property or non-`#f` *`proc-spec`* is supplied to `make-struct-type` with a supertype that already behaves as a procedure, the `exn:fail:contract` exception is raised.

`(procedure-struct-type? struct-type)` returns `#t` if instances of the structure type represented by *`struct-type`* are procedures (according to `procedure?`), `#f` otherwise.

4.7 Structures as Synchronizable Events

The built-in `prop:evt` structure type property identifies structure types whose instances can serve as synchronizable events; see §7.7 for information on synchronization and events.

The property value can be any of the following:

- An event *`evt`*: In this case, using the structure as an event is equivalent to using *`evt`*.
- A procedure *`proc`* of one argument: In this case, the structure is similar to an event generated by `guard-evt`, except that the would-be guard procedure *`proc`* receives the structure as an argument, instead of no arguments.
- An exact, non-negative integer between `0` (inclusive) and *`init-field-k`* (exclusive): The integer identifies a field in the structure, and the field must be designated as immutable. If the field contains an object or an event-generating procedure of one argument, the event or procedure is used as above. Otherwise, the structure acts as an event that is never ready.

Instances of a structure type with the `prop:input-port` or `prop:output-port` property (see §4.8) is also synchronizable by virtue of being a port. If the structure type has more than one of `prop:evt`, `prop:input-port`, and `prop:output-port`, then the `prop:evt` value (if any) takes precedence for determing the instance's behavior as an event, and the `prop:input-port` property takes precedence over `prop:output-port` for synchronization.

Examples:

```
(define-values (struct:wt make-wt wt? wt-ref wt-set!)
  (make-struct-type 'wt #f 2 0 #f (list (cons prop:evt 0)) #f #f '(0)))

(define sema (make-semaphore))
(sync/timeout 0 (make-wt sema #f)) ; ⇒ #f
(semaphore-post sema)
(sync/timeout 0 (make-wt sema #f)) ; ⇒ sema
(semaphore-post sema)
(sync/timeout 0 (make-wt (lambda (self) (wt-ref self 1)) sema)) ; ⇒ sema
(semaphore-post sema)
(define my-wt (make-wt (lambda (self) (wrap-evt
                                         (wt-ref self 1)
                                         (lambda (x) self)))
                       sema))
(sync/timeout 0 my-wt) ; ⇒ my-wt
(sync/timeout 0 my-wt) ; ⇒ #f
```

4.8 Structures as Ports

The built-in `prop:input-port` and `prop:output-port` structure type properties identify structure types whose instances can serve as input and output ports, respectively.

The property value can be any of the following:

- An input port (for `prop:input-port`) or output port (for `prop:input-port`): In this case, using the structure as port is equivalent to using the given one.
- An exact, non-negative integer between `0` (inclusive) and *`init-field-k`* (exclusive): The integer identifies a field in the structure, and the field must be designated as immutable. If the field contains an input port (for `prop:input-port`) or output port (for `prop:input-port`), the port is used. Otherwise, an empty string input port is used for `prop:input-port`, and a port that discards all data is used for `prop:output-port`.

Some procedures, such as `file-position`, work on both input and output ports. When given an instance of a structure type with both the `prop:input-port` and `prop:output-port` properties, the instance is used as an input port.

4.9 Structure Utilities

The following utility procedures work on all structure instances:

- `(struct->vector` *`v`* [*`opaque-v`*]`)` creates a vector representing *`v`*. The first slot of the result vector contains a symbol of the form `struct:`*`s`*. Each remaining slot contains either the value of a field in *`v`* if it is accessible via the current inspector, or *`opaque-v`* for a field that is not accessible. A single *`opaque-v`* value is used in the vector for contiguous inaccessible fields. (Consequently, the size of the vector does not match the size of the *`struct`* if more than one field is inaccessible.) The symbol `'...` is the default value for *`opaque-v`*.
- `(struct?` *`v`*`)` returns `#t` if `struct->vector` exposes any fields of *`v`* with the current inspector, `#f` otherwise.

Two structure values are `eqv?` if and only if they are `eq?`. Two structure values are `equal?` if and only if they are instances of the same structure type, no fields are opaque, and the results of applying `struct->vector` to the structs are `equal?`. (Consequently, `equal?` testing for structures depends on the current inspector.)

Each kind of value returned by `define-struct` and `make-struct-type` has a recognizing predicate:

- `(struct-type?` *`v`*`)` returns `#t` if *`v`* is a structure type descriptor value, `#f` otherwise.
- `(struct-constructor-procedure?` *`v`*`)` returns `#t` if *`v`* is a constructor procedure generated by `define-struct` or `make-struct-type`, `#f` otherwise.
- `(struct-predicate-procedure?` *`v`*`)` returns `#t` if *`v`* is a predicate procedure generated by `define-struct` or `make-struct-type`, `#f` otherwise.
- `(struct-accessor-procedure?` *`v`*`)` returns `#t` if *`v`* is an accessor procedure generated by `define-struct`, `make-struct-type`, or `make-struct-field-accessor`, `#f` otherwise.
- `(struct-mutator-procedure?` *`v`*`)` returns `#t` if *`v`* is a mutator procedure generated by `define-struct`, `make-struct-type`, or `make-struct-field-mutator`, `#f` otherwise.

5. Modules

MzScheme provides a module system for managing the scope of variable and syntax definitions, and for directing compilation. Module declarations can appear only at the top level. The space of module names is separate from the space of top-level variable and syntax names.

A `module` declaration consists of the name for the module, the name of a module to supply an initial set of syntax and variable bindings, and a module body:

```
(module module-identifier initial-required-module-name body-datum ···)
```

A module encapsulates syntax definitions to be used in expanding the body of the module, as well as expressions and definitions to be evaluated when the module is executed. When a syntax identifier is exported with `provide` (as described in §5.2), its transformer can be used during the expansion of an importing module; when a variable identifier is exported, its value can be used (but not assigned with `set!`) during the execution of an importing module.

A module named `mzscheme` is built in, and it exports the procedures and syntactic forms described in R^5RS and this manual. The `mzscheme` module supplies the initial syntax and variable bindings for a typical module.

Example:

```
(module hello-world    ; the module name
        mzscheme       ; initial syntax and variable bindings
                       ; for the module body
  ; the module body
  (display "Hello world!")
  (newline))
```

In general, the initial import serves as a kind of ”language” declaration. By initially importing a module other than `mzscheme`, a module can be defined in terms of a commonly-used variant of Scheme that contains more than the MzScheme built-in syntax and procedures, or a variant of Scheme that contains fewer constructs. The initial import might even omit syntax for declaring additional imports. For example, §12.5 shows an example module that defines a *`lambda-calculus`* language.

5.1 Module Expansion and Execution

When a module declaration is evaluated, the module's body is syntax-expanded and compiled, but not executed. The body is executed only when the module is explicitly invoked, via a `require` or `require-for-syntax` expression at the top level, or a call to `dynamic-require`.

When a module is invoked, its body definitions and expressions are evaluated. First, however, the definitions and expressions are evaluated for each module imported (via `require`) by the invoked module. The import-initialization rule applies up the chain of modules, so that every module used (directly or indirectly) by the invoked module is executed before any module that uses its exports. A module can only import from previously declared modules, so the module-import relationship is acyclic.

Every module is executed at most once in response to an invocation, regardless of the number of times it is imported into other modules. Every top-level invocation executes only the modules needed by the invocation that have not been executed by previous invocations.

Example:

```
(module never-used                ; unused module
        mzscheme
  (display "This is never printed")
  (newline))

(module hello-world-printer       ; module used by hello-world2
        mzscheme
  (define (print-hello-world)
    (display "Hello world!")
    (newline))
  (display "printer ready")
  (newline)
  (provide print-hello-world))    ; export

(module hello-world2
        mzscheme                  ; initial import
  (require hello-world-printer)   ; additional import
  (print-hello-world))

(require hello-world2)    ; ⇒ prints "printer ready", then "Hello world!"
```

Separating module declarations from module executions benefits compilation in the presence of expressive syntax transformers, as explained in §12.3.4.

5.2 Module Bodies

In general, the format of a module body depends on the initial import. Since the `mzscheme` module defines the procedures and syntactic forms described in R^5RS and this manual, the *`body-datum`*s of a module using `mzscheme` as its initial import must conform to the usual MzScheme top-level grammar.

The `require` form is used both to invoke a module at the top level, and to import syntax and variables into a module.

```
(require require-spec ···)

require-spec is one of
  module-name
  (only module-name identifier ···)
  (prefix prefix-identifier module-name)
  (all-except module-name identifier ···)
  (prefix-all-except prefix-identifier module-name identifier ···)
  (rename module-name local-identifier exported-identifier)
```

The *`module-name`* form imports all exported identifiers from the named module. The `(only` *`module-name identifier`* `···)` form imports only the listed identifiers from the named module. The `(prefix` *`prefix-identifier module-name`*`)` form imports all identifiers from the named module, but locally prefixes each identifier with *`prefix-identifier`*. The `(all-except` *`module-name identifier`* `···)` form imports all identifiers from the named module, except for the listed identifiers. The `(prefix-all-except` `prefix-identifier module-name identifier ···)` form combines the `prefix` and `all-except` forms. Finally, the `(rename module-name local-identifier exported-identifier)` imports `exported-identifier` from `module-name`, binding it locally to `identifier`.

The `provide` form (legal only within a module declaration) exports syntax and variable bindings from the current module for use by other modules. The exported identifiers must be either defined or imported in the module, but the export of an identifier may precede its definition or import.

```
(provide provide-spec ···)

provide-spec is one of
  identifier
  (rename local-identifier export-identifier)
  (struct struct-identifier (field-identifier ···))
  (all-from module-name)
  (all-from-except module-name identifier ···)
  (all-defined)
  (all-defined-except identifier ···)
  (prefix-all-defined prefix-identifier)
  (prefix-all-defined-except prefix-identifier identifier ···)
  (protect provide-spec ···)
```

The `identifier` form exports the (imported or defined) identifier from the module. The `(rename local-identifier export-identifier)` form exports `local-identifier` from the module with the external name `export-identifier`; other modules importing from this one will see `export-identifier` instead of `local-identifier`. The `(struct struct-identifier (field-identifier ···))` form exports the names that `(define-struct struct-identifier (field-identifier ···))` generates. The `(all-from module-name)` form exports all of the identifiers imported from the named module, using their local names. The `(all-from-except module-name identifier ···)` form is similar, except that the listed imported identifiers are not exported. The `(all-defined)` form exports all of the identifiers defined (not imported) in the module. The `(all-defined-except identifier ···)` form is similar, except that the listed defined identifiers are not exported. The `(prefix-all-defined prefix-identifier)` and `(prefix-all-defined-except prefix-identifier identifier ···)` forms are like `all-defined` and `all-defined-except`, but `prefix-identifier` is prefixed onto each defined identifier for its external name.

The `(protect provide-spec ···)` form is like the sequence of individual `provide-spec`s, but the provided identifiers are protected (see §9.4); the `provide-spec`s must not contain another `protect` form, an `all-from` form, or an `all-from-except` form, and they must not name any identifier that is imported into the providing module, instead of defined within the module.

The scope of all imported identifiers covers the entire module body, as does the scope of any identifier defined within the module body. See §12.3.5 for additional information concerning macro-generated definitions, `require` declarations, and `provide` declarations. An `identifier` can be defined by a definition or import at most once, except than an `identifier` can be imported multiple times if each import is from the same module. All exports must be unique. A module body cannot contain free variables. A module is not permitted to mutate an imported variable with `set!`. However, mutations to an exported variable performed by its defining module are visible to modules that import the variable.

At syntax-expansion time, expressions and definitions within a module are partially expanded, just enough to determine whether the expression is a definition, syntax definition, import, export, or a non-definition. If a partially expanded expression is a syntax definition, the syntax transformer is immediately evaluated and the syntax name is available for expanding successive expressions. Import expressions are treated similarly, so that imported syntax is available for expansion following its import. (The ordering of syntax definitions does not affect the scope of the syntax names; a transformer for `A` can produce expressions containing `B`, while the transformer for `B` produces expressions containing *A*, regardless of the order of declarations for *A* and *B*. However, a syntactic form that produces syntax definitions must be defined before it is used.) The `begin` form at the top level for a module body works like `begin` at the top level, so that the sub-expressions are flattened out into the module's body. Expressions determined to be non-definitions are expanded in an expression context; if further expansion (due to a macro definition processes later in the module body) produces a definition, a syntax exception is raised.

At run time, expressions and definitions are evaluated in order as they appear within the module. Accessing a (non-syntax) identifier before it is initialized signals a run-time error, just like accessing an undefined global variable.

Example:

```
(module a mzscheme
  (provide x)
  (define x 1))

(module b mzscheme
  (provide f (rename x y))
  (define x 2)
  (define (f) (set! x 7)))

(module c mzscheme
  (require (prefix a. a) (prefix b. b))
  (b.f)
  (display (+ a.x b.y))
  (newline))

(require c)  ; ⇒ executes c, prints 8
```

5.3 Modules and Macros

Macros defined with `syntax-rules` follow the rules specified in R^5RS regarding the binding and free references in the macro template. In particular, the template of an exported macro may refer to an identifier defined in the module or imported into the module; uses of the macro in other modules expand to references of the identifier defined or imported at the macro-definition site, as opposed to the use site. Uses of a macro in a module must not expand to a `set!` assignment of an identifier from any other module (including the module that defines the macro).

Example:

```
(module a mzscheme
  (provide xm)
  (define y 2)
  (define-syntax xm     ; a macro that expands to y
    (syntax-rules ()
      [(xm) y])))

(module b mzscheme
  (require a)
  (printf "~a~n" (xm)))

(require b)   ; ⇒ prints 2
```

For further information about syntax definitions, see §12.3.4. See §12.6.5 for information on extracting details about an expanded or compiled module declaration. See §9.4 for information on how unexported and protected identifiers in a macro expansion are constrained to their macro-introduced contexts.

5.4 Module Paths

In practice, the modules composing a program are rarely declared together in a single file. Multiple module-declaring files can be loaded in sequence with `load`, but modules that are intended as libraries have complex interdependencies; constructing an appropriate sequence of `load` expressions — one that loads each module declaration exactly once and before all of its uses — can be difficult and tedious. Worse, even though module declarations prevent collisions among syntax and variable names, module names themselves can collide.

To solve these problems, a *`module-name`* can describe a path to a module source file, which is resolved by the current *module name resolver*. The default module name resolver loads the source for a given module path the first time that the source is referenced. To avoid module name collisions, the module in the referenced file is assigned a name that identifies its source file.

A module path resolved by the standard resolver can take any of four forms:

```
unix-relative-path-string
(file path-string)
(lib filename-string collection-string ···)
(planet . datum)
path
```

- When a module name is a string, *`unix-relative-path-string`*, it is interpreted as a path relative to the source of the containing module (as determined by `current-load-relative-directory` or `current-directory`). Regardless of the platform running MzScheme, the path is always parsed as a Unix-format path: **/** is the path delimiter (multiple adjacent **/** are treated as a single delimiter), **..** accesses the parent directory, and **.** accesses the current directory. To avoid portability problems, the path elements are further constrained to contain only ASCII alpha-numeric characters plus **-**, **_**, **.**, and space, and the path may not be empty or contain a leading or trailing slash.

- When a module name has the form `(file` *`path-string`*`)`, then *`path-string`* is interpreted as a file path using the current platform's path conventions. If *`path-string`* is a relative path, it is resolved relative to the source of the containing module (as determined by `current-load-relative-directory` or `current-directory`).

- When a module name has the form `(lib` *`filename-string collection-string`* `···)`, it specifies a collection-based library; see Chapter 16 for more information about libraries and collections.

- When a module name has the form `(planet .` *`datum`*`)`, it is passed to the PLaneT resolver as described in §5.4.1.

- Since path values (see §11.3.1) cannot be written as literal syntax, a *`path`* never appears in `require` forms. However, an absolute path value may be passed to `dynamic-require`, and it is treated in the same way as a `file` form.

A source file that is referenced by a module path must contain a single module declaration. The name of the declared module must match the source's filename, minus its suffix.

Different module paths can access the same module, but for the purposes of `provide` declarations using `all-from` and `all-from-except`, source module paths are compared syntactically (instead of comparing resolved module names).

5.4.1 Module Name Resolver

In general, the module name resolver is invoked by MzScheme when a *`module-name`* is not an identifier. The grammar of non-symbolic module names is determined by the module name resolver. The module name resolver, in turn, is determined by the current-module-name-resolver parameter (see also §7.9.1.12). The resolver is a function that takes one, three, and four arguments:

- When given one argument, it is a symbol for a module that is already loaded. Such a call to the module name resolver is a notification that the corresponding module does not need to be loaded (for the current namespace, or any other namespace that shared the module registry). The procedure result is ignored.
- When given three arguments, the first is an arbitrary value for the module path, a symbol for the source module's name, and a syntax object or #f. The procedure result must be a symbol for the resolved name.
- The four-argument case is the same as the three-argument case, but with a boolean argument that can be #f to request resolving a name without loading the module (if it is not already loaded).

Except for (planet . *datum*) paths (which are handled as described below), the standard module name resolver creates a module identifier as the expanded, simplified, case-normalized, and de-suffixed path of the file designated by the module path—transformed to a symbol by composing path->bytes, bytes->string/latin-1, and string->symbol, and then prefixed with a comma. (See §11.3 for details on platform-specific path handling.) To better support dynamic-require, the standard module name resolver accepts a path object (see §11.3.1) and treats it like a file module path.

The standard module name resolver keeps a per-registry table of loaded module identifiers (where the registry is obtained from a namespace; see Chapter 8). If the resolved identifier is not in the table, and #f is not provided as the module name resolver's fourth argument, then the identifier is put into the table and the corresponding file is loaded with a variant of load/use-compiled that passes the expected module name to the load handler.

While loading a file, the standard resolver sets the current-module-name-prefix parameter, so that the name of any module declared in the loaded file is given a prefix. This mechanism enables the resolver to avoid module name collisions. The resolver sets the prefix to the resolved module name, minus the de-suffixed file name. It also loads the file by calling the load handler or load extension handler with the name of the expected module (see §5.8).

Module loading is suppressed (i.e., #f is supplied as a fourth argument to the module name resolver) when resolving module paths in syntax objects (see §12.2). When a syntax object is manipulated, the current namespace might not match the original namespace for the syntax object, and the module should not necessarily be loaded in the current namespace.

The current module name resolver is called with a single argument by namespace-attach-module to notify the resolver that a module was attached to the current namespace (and should not be loaded in the future for the namespace's registry). No other MzScheme operation invokes the module name resolver with a single argument, but other tools (such as DrScheme) might call this resolver in this mode to avoid redundant module loads.

When the default module name resolver is given a module path of the form (planet . *datum*) as its first argument, it provides all of the resolver arguments to the PLaneT resolver. If the PLaneT resolver has not yet been loaded, it is loaded in the initial namespace by requiring planet-module-name-resolver from (lib "resolver.ss" "planet"). Thereafter, the PLaneT resolver is called for every one-argument call to the default module name resolver, in addition to calls for handle (planet . *datum*) paths.

5.4.2 Module Names and Compilation

When syntax-expanding or compiling a module declaration, MzScheme resolves module names for imports (since some imported identifier may have syntax bindings), but it also preserves the module path name. Consequently, a compiled module can be moved to another filesystem, where the module name resolver can resolve inter-module references among compiled code.

When a module reference is extracted from compiled code (see §12.6.5) or from syntax objects in macro expansion (see

§12.2.2), the module reference is typically reported in the form of a *module path index*. An index is a semi-interned,[1] opaque value that encodes a relative module path (see §5.4) and another index to which it is relative.

An index that returns `#f` for its path and base index represents "self" — i.e., the module declaration that was the source of the index — and such an index is always used as the root for a chain of indices. For example, when extracting information about an identifier's binding within a module, if the identifier is bound by a definition within the same module, the identifier's source module will be reported using the "self" index. If the identifier is instead defined in a module that is imported via a module path (as opposed to a literal module name), then the identifier's source module will be reported using an index that contains the `require`d module path and the "self" index.

An index has state; when it is *resolved* to a symbolic module name, then the symbol is stored with the index. In particular, when a module is loaded, its root index is resolved to match the module's declaration-time name. This resolved path is forgotten, however, in identifiers that the module contributes to the compiled and marshaled form of other modules. This transient nature of resolved names allows the module code to be loaded with a different resolved name than the name when it was compiled.

Where an index is expected, a symbol can usually take its place, representing a literal module name. A symbol is used instead of an index when a module is imported using its name directly with `require` instead of a module path.

- `(module-path-index? v)` returns `#t` if *v* is a module path index, `#f` otherwise.
- `(module-path-index-resolve module-path-index)` returns a symbol for the resolved module name, computing the resolved symbol (and storing it in *module-path-index*) if it has not been computed before. Resolving an index uses the current module name resolver (see §5.4.1). Depending on the kind of module paths encapsulated by *module-path-index*, the computed resolved symbol can depend on the current load directory or current directory.
- `(module-path-index-split module-path-index)` returns two values: a non-symbol S-expression representing a module path, and a base index (to which the module path is relative), symbol, or `#f`. A `#f` second result means "relative to a top-level environment". A `#f` for the first result implies a `#f` for the second result, and means that *module-path-index* represents "self" (see above).
- `(module-path-index-join module-path module-path-index)` combines *module-path* and *module-path-index* to create a new module path index. The *module-path* argument can be anything except a symbol, and the *module-path-index* argument can be a index, symbol, or `#f`.

5.5 Dynamic Module Access

`(dynamic-require module-path-v provided-symbol)` dynamically invokes the module specified by *module-path-v* in the current namespace's registry if it is not yet invoked. If *module-path-v* is not a symbol, the current module name resolver may load a module declaration to resolve it. For example, the default module-name resolver accepts a path value as *module-path-v*. The path is not resolved with respect to any other module, even if the current namespace corresponds to a module body.

If *provided-symbol* is `#f`, then the result is void. Otherwise, when *provided-symbol* is a symbol, the value of the module's export with the given name is returned. If the module exports *provide-symbol* as syntax, then a use of the binding is expanded and evaluated (in a fresh namespace to which the module is attached). If the module has no such exported variable or syntax, or if the variable is protected (see §9.4), the `exn:fail:contract` exception is raised. The expansion-time portion of the module is not executed.

If *provided-symbol* is void, then the module is partially invoked, where its expansion-time expressions are evaluated, but not its normal expressions (though the module may have been invoked previously in the current namespace's registry). The result is void.

[1] Multiple references to the same relative module tend to use the same index value, but not always.

`(dynamic-require-for-syntax` *`module-path-v provided-symbol-or-#f`*`)` is similar to `dynamic-require`, except that it accesses a value from an expansion-time module instance (the one that could be used by transformers in expanding top-level expressions in the current namespace). As with `dynamic-require`, the module name resolver may load a module declaration to resolve *`module-path-v`* if it is not a symbol.

5.6 Re-declaring Modules

When a module is re-declared in a namespace whose registry already contains a declaration of the module (see Chapter 8), the new declaration's syntax and variable definitions replace and extend the old declarations. If a variable in the old declaration has no counterpart in the new declaration, it continues to exist, but becomes inaccessible to newly compiled code. In other words, a module name in a particular registry maps to a namespace containing the module body's definitions; see also `module->namespace` in §8.3.

If a module is invoked before it is re-declared, each re-declaration of the module is immediately invoked. The immediate invocation is necessary to keep the module-specific namespace consistent with the module declaration.

When a module re-declaration implies invocation, the invocation can fail at the definition of a binding that was constant in the original module (where any definition without a `set!` within the module counts as a constant definition); preventing re-definition protects potential optimizations (for the original declaration) that rely on constant bindings. Set the `compile-enforce-module-constants` parameter (see §7.9) to `#f` to disable optimizations that rely on constant bindings and to allow unrestricted re-definition of module bindings. To enable re-definition, the `compile-enforce-module-constants` parameter must be set before the original declaration of the module.

In addition to the constraint on constant definitions, a module can be redeclared only when the current code inspector — as determined by the `current-code-inspector` parameter (see §7.9.1.8) — controls the invocation of the module in the current namespace's registry. If the current code inspector does not control the invocation at the time of a re-declaration attempt, the `exn:fail:contract` exception is raised.

5.7 Built-in Modules

The built-in `mzscheme` module is implemented by several primitive modules whose names start with `#%`. In general, module names starting with `#%` are reserved for use by MzScheme and embedding applications. The built-in modules are declared in the initial namespace's registry via `namespace-attach-module`, so they cannot be re-declared and their private namespaces are not available via `module->namespace`.

5.8 Modules and Load Handlers

The second argument to a load handler or load extension handler indicates whether the load is expected (and required) to produce a module declaration. If the second argument is `#f`, the file is loaded normally, otherwise the argument will be a symbol and the file must be checked specially before it is loaded.

When the second argument to the local handler is a symbol, the handler is responsible for ensuring that the file-to-load actually contains a `module` declaration (possibly compiled); if not, it must raise an exception without evaluating the declaration. The handler must also raise an `exn:fail` exception if the name in the module declaration is not the same as the symbol argument to the handler (before applying any prefix in `current-module-name-prefix`).

Furthermore, while reading the file and expanding the module declaration, the load handler must set reader parameter values (see §7.9.1.3) to the following states:

```
(read-case-sensitive #t)
(read-square-bracket-as-paren #t)
(read-curly-brace-as-paren #t)
```

```
(read-accept-box #t)
(read-accept-compiled #t)
(read-accept-bar-quote #t)
(read-accept-graph #t)
(read-decimal-as-inexact #t)
(read-accept-dot #t)
(read-accept-quasiquote #t)
(read-accept-reader #t)
```

These states are the same as the normal defaults, except that compiled-code reading is enabled. Note that a module body can be made case insensitive by prefixing the module with `#ci` (see §11.2.4).

Finally, before compiling or evaluating a module declaration from source, the handler must replace a leading `module` identifier with an identifier that is bound to the `module` export of MzScheme. Evaluating the expression will then produce a module declaration, regardless of the binding of `module` in the current namespace.

Separate compilation of `module` declarations introduces the possibility of import cycles when the module declarations are executed. The `exn:fail` exception is raised when such a cycle is detected.

6. Exceptions and Control Flow

6.1 Exceptions

MzScheme supports a variant of the exception system proposed by Friedman, Haynes, and Dybvig.[1] MzScheme's implementation extends that proposal by defining the specific exception values that are raised by each primitive error.

- `(raise` *`v`*`)` raises an exception, where *`v`* represents the exception being raised. The *`v`* argument can be anything; it is passed to the current *exception handler*. Breaks are disabled from the time the exception is raised until the exception handler obtains control, and the handler itself is `parameterize-break`ed to disable breaks initially; see §6.7 for more information on breaks.

- `(call-with-exception-handler` *`f thunk`*`)` installs *`f`*, which must be a procedure of one argument, as the exception handler for the current continuation (i.e., for the dynamic extent of a call to *`thunk`*). The *`thunk`* is called in tail position with respect to the call to `call-with-exception-handler`. If an exception is raised during the evaluation of *`thunk`* (in an extension of the current continuation that does not have its own exception handler), then *`f`* is applied to the `raise`d value in the continuation of the `raise` call (but extended with a continuation barrier; see §6.3).

  Any procedure that takes one argument can be an exception handler. If the exception handler returns a value when invoked by `raise`, then `raise` propagates the value to the "previous" exception handler (still in the dynamic extent of the call to `raise`). The previous exception handler is the exception handler associated with the rest of the continuation after the point where the called exception handler was associated with the continuation; if no previous handler is available, the uncaught-exception handler is used (see below). In all cases, a call to an exception handler is `parameterize-break`ed to disable breaks, and it is wrapped with `call-with-exception-handler` to install the an exception handler that reports both the original and newly raised exceptions.

- `(uncaught-exception-handler)` returns an exception handler that is used by `raise` when the relevant continuation has no exception handler installed with `call-with-exception-handler` or `with-handlers`. The `uncaught-exception-handler` procedure is a parameter (see §7.9). Unlike exception handlers installed with `call-with-exception-handler`, the handler for uncaught exceptions must not return a value when called by `raise`; if it returns, an exception is raised (to be handled by an exception handler that reports both the original and newly raised exception).

  The default uncaught-exception handler prints an error message using the current error display handler (see `error-display-handler` in §7.9.1.7) and then escapes by calling the current error escape handler (see `error-escape-handler` in §7.9.1.7). The call to each handler is `parameterize`d to set `error-display-handler` to the default error display handler,[2] and it is `parameterize-break`ed to disable breaks. The call to the error escape handler is further parameterized to set `error-escape-handler` to the default error escape handler.

- `(with-handlers ((`*`pred handler`*`)` $\cdots$`)` *`expr`* $\cdots^1$`)` is a syntactic form that evaluates the *`expr`* body, installing a new exception handler during the dynamic extent of the *`expr`*s. The *`pred`* and *`handler`*

[1] See http://www.cs.indiana.edu/scheme-repository/doc.proposals.exceptions.html

[2] If the current error display handler is the default handler, then the error-display call is parameterized to install an emergency error display handler that attempts to print directly to a console and never fails.

expressions are evaluated in the order that they are specified, before the first `expr` and before the exception handler is changed.

The new exception handler processes an exception only if one of the `pred` procedures returns a true value when applied to the exception, otherwise the exception handler is invoked from the continuation of the `with-handlers` expression (by raising the exception again). If an exception is handled by one of the `handler` procedures, the result of the entire `with-handlers` expression is the return value of the handler.

When an exception is raised during the evaluation of `expr`s, each predicate procedure `pred` is applied to the exception value; if a predicate returns a true value, the corresponding `handler` procedure is invoked with the exception as an argument. The predicates are tried in the order that they are specified.

Before any predicate or handler procedure is invoked, the continuation of the entire `with-handlers` expression is restored, but also `parameterize-break`ed to disable breaks. Thus, breaks are disabled by default during the predicate and handler procedures (see §6.7), and the exception handler is the one from the continuation of the `with-handlers` expression.

The `exn:fail?` procedure is useful as a handler predicate to catch all error exceptions. Avoid using `(lambda (x) #t)` as a predicate, because the `exn:break` exception typically should not be caught (unless it will be re-raised to cooperatively break). Beware, also, of catching and discarding exceptions, because discarding an error message can make debugging unnecessarily difficult.

- `(with-handlers* ((pred handler) ···) expr ···`^{1}`)` is the same as `with-handlers`, but if a `handler` procedure is called, breaks are not explicitly disabled, and the call is in tail position with respect to the `with-handlers*` form.

The following example defines a divide procedure that returns `+inf.0` when dividing by zero instead of signaling an exception (other exceptions raised by `/` are signaled):

```
(define div-w-inf
  (lambda (n d)
    (with-handlers ([exn:fail:contract:divide-by-zero?
                     (lambda (exn) +inf.0)])
      (/ n d))))
```

The following example catches and ignores file exceptions, but lets the exception handler of the enclosing continuation handle other exceptions, including breaks:

```
(define (file-date-if-there filename)
  (with-handlers ([exn:fail:filesystem? (lambda (exn) #f)])
    (file-or-directory-modify-seconds filename)))
```

6.1.1 Primitive Exceptions

Whenever a primitive error occurs in MzScheme, an exception is raised. The value that is passed to the current exception handler is always an instance of the `exn` structure type. Every `exn` structure value has a `message` field that is a string, the primitive error message. The default exception handler recognizes exception values with the `exn?` predicate and passes the error message to the current error display handler (see `error-display-handler` in §7.9.1.7).

Primitive errors do not create immediate instances of the `exn` structure type. Instead, an instance from a hierarchy of subtypes of `exn` is instantiated. The subtype more precisely identifies the error that occurred and may contain additional information about the error. The table below defines the type hierarchy that is used by primitive errors and matches each subtype with the primitive errors that instantiate it. In the table, each bulleted line is a separate structure type. A type is nested under another when it is a subtype. The full name of the structure type (as used by predicates and selectors in the global environment) is built by combining the full name of the immediate supertype with ":" and the subtype name.

For example, reading an ill-formed expression raises an exception as an instance of `exn:fail:read`. An exception handler can test for this kind of exception using the global `exn:fail:read?` predicate. Given such an exception, an error string can be extracted using `exn-message`, while `exn:fail:read-source` accesses a list of source locations for the error.

Fields of the built-in `exn` structure types are immutable, so field mutators are not provided. Field-type contracts are enforced through guards; for example, `(make-exn "Hello" #f)` raises `exn:fail:contract` because the second argument is not a continuation mark set. A mutable string, however, is allowed as the first argument to an exception constructor, in which case it is converted to an immutable string by the guard. All built-in `exn` structure types are transparent to all inspectors (see §4.5).

- `exn` : *not instantiated directly by any primitive*
  fields: `message` — error message (type: *immutable string*)
  `continuation-marks` — value returned by `current-continuation-marks` immediately before the exception is raised (type: *mark set*)
  - `fail` : exceptions that represent errors
    - `contract` : inappropriate run-time use of a function or syntactic form
      - `arity` : application with the wrong number of arguments
      - `divide-by-zero` : divide by zero
      - `continuation` : attempt to cross a continuation barrier
      - `variable` : unbound/not-yet-defined global or module variable
        fields: `id` — the variable's identifier (type: *symbol*)
    - `syntax` : syntax error, but not a `read` error
      fields: `exprs` — illegal expression(s) (type: *immutable list of syntax objects*)
    - `read` : `read` parsing error
      fields: `srclocs` — source location(s) of error (type: *immutable list of* `srclocs` *(see §11.2.1.1)*)
      - `eof` : unexpected end-of-file
      - `non-char` : unexpected non-character
    - `filesystem` : error manipulating a filesystem object
      - `exists` : attempt to create a file that exists already
      - `version` : version mismatch loading an extension
    - `network` : TCP and UDP errors
    - `out-of-memory` : out of memory
    - `unsupported` : unsupported feature
    - `user` : for end users
  - `break` : asynchronous break signal
    fields: `continuation` — resumes from the break (type: *escape continuation*)

In addition to the built-in structure types for exceptions, MzScheme provides one built-in structure-type property (see §4.4):

- The `prop:exn:srclocs` property identifies exceptions that have a list of source locations, which includes `exn:fail:read` and `exn:fail:syntax`. The `exn:srclocs?` predicate recognizes structures and structure types that have the `prop:exn:srclocs` property. The `exn:srclocs-accessor` procedure takes a structure or structure type with the `exn:srclocs` property and returns a procedure; when the procedure is applied to a structure of the same type, it returns a list of `srclocs` (see §11.2.1.1).

Primitive procedures that accept a procedure argument with a particular required arity (e.g., `call-with-input-file`, `call/cc`) check the argument's arity immediately, raising `exn:fail:contract` if the arity is incorrect.

6.2 Errors

The procedure `error` raises the exception `exn:fail` (which contains an error string). The `error` procedure has three forms:

- `(error` *`symbol`*`)` creates a message string by concatenating `"error: "` with the string form of *`symbol`*.
- `(error` *`msg-string v`* `···)` creates a message string by concatenating *`msg-string`* with string versions of the *`v`*s (as produced by the current error value conversion handler; see §7.9.1.7). A space is inserted before each *`v`*.
- `(error` *`src-symbol format-string v`* `···)` creates a message string equivalent to the string created by:

```
(format (string-append "~s: " format-string)
 src-symbol v ···)
```

In all cases, the constructed message string is passed to `make-exn:fail` and the resulting exception is raised.

The `raise-user-error` procedure is the same as `error`, except that it constructs an exception with `make-exn:fail:user` instead of `make-exn:fail`. The default error display handler does not show a "stack trace" for `exn:fail:user` exceptions (see §6.6), so `raise-user-error` should be used for errors that are intended for end users. Like `error`, `raise-user-error` has three forms:

- `(raise-user-error` *`symbol`*`)`
- `(raise-user-error` *`msg-string v`* `···)`
- `(raise-user-error` *`src-symbol format-string v`* `···)`

6.2.1 Application Errors

`(raise-type-error` *`name-symbol expected-string v`*`)` creates an `exn:fail:contract` value and `raise`s it as an exception. The *`name-symbol`* argument is used as the source procedure's name in the error message. The *`expected-string`* argument is used as a description of the expected type, and *`v`* is the value received by the procedure that does not have the expected type.

`(raise-type-error` *`name-symbol expected-string bad-k v`*`)` is similar, except that the bad argument is indicated by an index (from 0), and all of the original arguments *`v`* are provided (in order). The resulting error message names the bad argument and also lists the other arguments. If *`bad-k`* is not less than the number of *`v`*s, the `exn:fail:contract` exception is raised.

`(raise-mismatch-error` *`name-symbol message-string v`*`)` creates an `exn:fail:contract` value and `raise`s it as an exception. The *`name-symbol`* is used as the source procedure's name in the error message. The *`message-string`* is the error message. The *`v`* argument is the improper argument received by the procedure. The printed form of *`v`* is appended to *`message-string`* (using the error value conversion handler; see §7.9.1.7).

`(raise-arity-error` *`name-symbol-or-procedure arity-v`* [*`arg-v`* `···`]`)` creates an `exn:fail:contract:arit` value and `raise`s it as an exception. The *`name-symbol-or-procedure`* is used for the source procedure's name in the error message. The *`arity-v`* value must be a possible result from `procedure-arity` (see §3.12.1), and it is used for the procedure's arity in the error message; if *`name-symbol-or-procedure`* is a procedure, its actual arity is ignored. The *`arg-v`* arguments are the actual supplied arguments, which are shown in the error message (using the error value conversion handler; see §7.9.1.7); also, the number of supplied *`arg-v`*s is explicitly mentioned in the message.

6.2.2 Syntax Errors

`(raise-syntax-error` *`name message-string`* [*`expr sub-expr`*]`)` creates an `exn:fail:syntax` value and `raise`s it as an exception. Macros use this procedure to report syntax errors. The *`name`* argument is usually `#f` when *`expr`* is provided; it is described in more detail below. The *`message-string`* is used as the main body of the error message. The optional *`expr`* argument is the erroneous source syntax object or S-expression. The optional *`sub-expr`* argument is a syntax object or S-expression within *`expr`* that more precisely locates the error. If *`sub-expr`* is provided, it is used (in syntax form) for the `exprs` field of the generated exception record, else the *`expr`* is used if provided, otherwise the `exprs` field is the empty list. Source location information in the error-message text is similarly extracted from *`sub-expr`* or *`expr`*, when at least one is a syntax object.

The form name used in the generated error message is determined through a combination of the *`name`*, *`expr`*, and *`sub-expr`* arguments. The *`name`* argument can `#f` or a symbol:

- `#f`: When *`name`* is `#f`, and when *`expr`* is either an identifier or a syntax pair containing an identifier as its first element, then the form name from the error message is the identifier's symbol.

  If *`expr`* is not provided, or if it is not an identifier or a syntax pair containing and identifier as its first element, then the form name in the error message is `"?"`.
- *`symbol`*: When *`name`* is a symbol, then the symbol is used as the form name in the generated error message.

See also §7.9.1.7.

6.2.3 Inferred Value Names

To improve error reporting, names are inferred at compile-time for certain kinds of values, such as procedures. For example, evaluating the following expression:

```
(let ([f (lambda () 0)]) (f 1 2 3))
```

produces an error message because too many arguments are provided to the procedure. The error message is able to report "f" as the name of the procedure. In this case, MzScheme decides, at compile-time, to name as *`f`* all procedures created by the `let`-bound `lambda`.

Names are inferred whenever possible for procedures. Names closer to an expression take precedence. For example, in

```
(define my-f
  (let ([f (lambda () 0)]) f))
```

the procedure bound to *`my-f`* will have the inferred name "f".

When an `'inferred-name` property is attached to a syntax object for an expression (see §12.6.2), the property value is used for naming the expression, and it overrides any name that was inferred from the expression's context.

When an inferred name is not available, but a source location is available, a name is constructed using the source location information. Inferred and property-assigned names are also available to syntax transformers, via *`syntax-local-name`*; see §12.6 for more information.

`(object-name` *`v`*`)` returns a value for the name of *`v`* if *`v`* has a name, `#f` otherwise. The argument *`v`* can be any value, but only (some) procedures, structs, struct types, struct type properties, regexp values, and ports have names. The name of a procedure, struct, struct type, or struct type property is always a symbol. The name of a regexp value is a string, and a byte-regexp value's name is a byte string. The name of a port is typically a path or a string, but it can be arbitrary. All primitive procedures have names (see §3.12.2).

6.3 Continuations

MzScheme supports delimited continuations, and even continuations captured by `call-with-current-continuation` (or `call/cc`) are delimited by a prompt. Prompt instances are tagged, and continuations are captured with respect to a particular prompt. Thus, MzScheme's `call-with-current-continuation` accepts an optional prompt-tag argument: `(call-with-current-continuation` *`proc`* [*`prompt-tag`*]`)`, where *`prompt-tag`* must be a result from either `default-continuation-prompt-tag` (the default) or `make-continuation-prompt-tag`. Prompts, prompt tags, and composable continuations are described further in §6.5.

The macro `let/cc` binds a variable to the continuation in an immediate body of expressions:

```
 (let/cc k expr ···¹)
=expands=>
 (call/cc (lambda (k) expr ···¹))
```

Capturing a continuation also captures the current continuation marks (see §6.6) up to the relevant prompt. The current parameterization (see §7.9) is captured if it was extended via `paramaterize` or installed via `call-with-parameterization` since the prompt.

A continuation can be invoked from the thread (see Chapter 7) other than the one where it was captured. Multiple return values can be passed to a continuation (see §2.2).

MzScheme installs a *continuation barrier* around evaluation in the following contexts, preventing full-continuation jumps across the barrier:

- applying an exception handler, an error escape handler, or an error display handler (see §6.1);
- applying a macro transformer (see §12.6), evaluating a compile-time expression, or applying a module name resolver (see §5.4.1);
- applying a custom-port procedure (see §11.1.7), an event guard procedure (see §7.7), or a parameter guard procedure (see §7.9);
- applying a security-guard procedure (see §9.1);
- applying a will procedure (see §13.3); or
- evaluating or loading code from the stand-alone MzScheme command line (see §17).

In addition, extensions of MzScheme may install barriers in additional contexts. In particular, MrEd installs a continuation barrier around most every callback. Finally, `(call-with-continuation-barrier` *`thunk`*`)` applies *`thunk`* with a barrier between the application and the current continuation.

In addition to regular `call/cc`, MzScheme provides `call-with-escape-continuation` (or `call/ec`) and `let/ec`. A continuation obtained from `call/ec` is actually a kind of prompt: applying an escape continuation can only *escape* back to the continuation (possibly past a continuation barrier); that is, an escape continuation is only valid when the current continuation is an extension of the escape continuation. Further, the application of `call/ec`'s argument is not a tail call. Escape continuations are provided mainly for backward compatibility, since they pre-date general prompts in MzScheme.

The `exn:fail:contract:continuation` exception is raised when a continuation application would cross a continuation barrier, or when an escape continuation is applied outside of its dynamic scope.

6.4 Dynamic Wind

(dynamic-wind *pre-thunk value-thunk post-thunk*) applies its three thunk arguments in order. The value of a dynamic-wind expression is the value returned by *value-thunk*. The *pre-thunk* procedure is invoked before calling *value-thunk* and *post-thunk* is invoked after *value-thunk* returns. The special properties of dynamic-wind are manifest when control jumps into or out of the *value-thunk* application (either due to a prompt abort or a continuation invocation): every time control jumps into the *value-thunk* application, *pre-thunk* is invoked, and every time control jumps out of *value-thunk*, *post-thunk* is invoked. (No special handling is performed for jumps into or out of the *pre-thunk* and *post-thunk* applications.)

When dynamic-wind calls *pre-thunk* for normal evaluation of *value-thunk*, the continuation of the *pre-thunk* application calls *value-thunk* (with dynamic-wind's special jump handling) and then *post-thunk*. Similarly, the continuation of the *post-thunk* application returns the value of the preceding *value-thunk* application to the continuation of the entire dynamic-wind application.

When *pre-thunk* is called due to a continuation jump, the continuation of *pre-thunk*

1. jumps to a more deeply nested *pre-thunk*, if any, or jumps to the destination continuation; then

2. continues with the context of the *pre-thunk*'s dynamic-wind call.

Normally, the second part of this continuation is never reached, due to a jump in the first part. However, the second part is relevant because it enables jumps to escape continuations that are contained in the context of the dynamic-wind call. Furthermore, it means that the continuation marks (see §6.6) and parameterization (see §7.9) for *pre-thunk* correspond to those of the dynamic-wind call that installed *pre-thunk*. The *pre-thunk* call, however, is parameterize-breaked to disable breaks (see also §6.7).

Similarly, when *post-thunk* is called due to a continuation jump, the continuation of *post-thunk* jumps to a less deeply nested *post-thunk*, if any, or jumps to a *pre-thunk* protecting the destination, if any, or jumps to the destination continuation, then continues from the *post-thunk*'s dynamic-wind application. As for *pre-thunk*, the parameterization of the original dynamic-wind call is restored for the call, and the call is parameterize-breaked to disable breaks.

In both cases, the target for a jump is recomputed after each *pre-thunk* or *post-thunk* completes. When a prompt-delimited continuation (see §6.5) is captured in a *post-thunk*, it might be delimited and instantiated in such a way that the target of a jump turns out to be different when the continuation is applied than when the continuation was captured. There may even be no appropriate target, if a relevant prompt or escape continuation is not in the continuation after the restore; in that case, the first step in a *pre-thunk* or *post-thunk*'s continuation can raise an exception.

Example:

```
(let ([v (let/ec out
           (dynamic-wind
            (lambda () (display "in "))
            (lambda ()
              (display "pre ")
              (display (call/cc out))
              #f)
            (lambda () (display "out "))))])
  (when v (v "post ")))
 ; ⇒ displays in pre out in post out

(let/ec k0
    (let/ec k1
      (dynamic-wind
       void
       (lambda () (k0 'cancel))
       (lambda () (k1 'cancel-canceled)))))
  ; ⇒ 'cancel-canceled

(let* ([x (make-parameter 0)]
       [l null]
       [add (lambda (a b)
              (set! l (append l (list (cons a b)))))])
  (let ([k (parameterize ([x 5])
             (dynamic-wind
                 (lambda () (add 1 (x)))
                 (lambda () (parameterize ([x 6])
                              (let ([k+e (let/cc k (cons k void))])
                                (add 2 (x))
                                ((cdr k+e))
                                (car k+e))))
                 (lambda () (add 3 (x)))))])
    (parameterize ([x 7])
      (let/cc esc
        (k (cons void esc)))))
  l) ; ⇒ '((1 . 5) (2 . 6) (3 . 5) (1 . 5) (2 . 6) (3 . 5))
```

6.5 Prompts and Composable Continuations

For an introduction to composable continuations, see Sitaram and Felleisen, "Control Delimiters and Their Hierarchies," *Lisp and Symbolic Computation*, 1990.

MzScheme's support for prompts and composable continuations most closely resembles Dorai Sitaram's `%` and `fcontrol` operators (see "Handling Control," *Proc. Conference on Programming Language Design and Implementation*, 1993). Since composable continuations capture and invoke `dynamic-wind` thunks, however, the `fcontrol` operator is split into separate capture and abort operations, giving programmers more flexibility with respect to escapes. Composable continuations also capture continuation marks (see §6.6) and exception handlers (see §6.1).

See also Chapter 15 of *PLT MzLib: Libraries Manual* for wrappers of MzScheme's primitives. The wrapper are generally simpler to use and have more standard names.

`(call-with-continuation-prompt` *`thunk`* [*`prompt-tag handler-proc-or-false`*]`)` calls *`thunk`* with the current continuation extended by a prompt. The prompt is tagged by *`prompt-tag`*, which must be a result from either `default-continuation-prompt-tag` (the default) or `make-continuation-prompt-tag`. The *`handler-proc-or-false`* argument specifies a handler procedure; the handler is called in tail position with repsect to the `call-with-continuation-prompt` call when the installed prompt is the target of a `abort-current-continuation` call with *`prompt-tag`*, and the remaining arguments of `abort-current-continuation` are supplied to the handler procedure. If *`handler-proc-or-false`* is `#f` or not supplied, the default handler accepts a single *`abort-thunk`* argument and calls `(call-with-continuation-prompt` *`abort-thunk prompt-tag`* `#f)`; that is, the default handler re-installs the prompt and continues with a given thunk.

`(abort-current-continuation` *`prompt-tag obj`* $\cdots^1$`)` resets the current continuation to that of the nearest prompt tagged by *`prompt-tag`* in the current continuation; if no such prompt exists, the `exn:fail:contract:continuation` exception is raised. The *`obj`*s are delivered as arguments to the target prompt's handler procedure.

The protocol for *obj*s supplied to an abort is specific to the *prompt-tag*. When *abort-current-continuation* is used with `(default-continuation-prompt-tag)`, generally a single thunk should be supplied that is suitable for use with the default prompt handler. Similarly, when `call-with-continuation-prompt` is used with `(default-continuation-prompt-tag)`, the associated handler should generally accept a single thunk argument.

`(make-continuation-prompt-tag` [*symbol*]`)` creates a prompt tag that is not `equal?` to the result of any other value (including prior or future results from `make-continuation-prompt-tag`). The optional *symbol* argument, if supplied, is used when printing the prompt tag.

`(default-continuation-prompt-tag)` returns a fixed, constant prompt tag for a which a prompt is installed at the start of every thread's continuation; the handler for each thread's initial prompt accepts any number of values and returns. The result of `default-continuation-prompt-tag` is the default tag for more any procedure that accepts a prompt tag.

A continuation captured by `(call-with-current-continuation ...` *promt-tag*`)` is truncated at the nearest prompt tagged by *prompt-tag* in the current continuation; if no such prompt exists, the `exn:fail:contract:continuation` exception is raised. The truncated continuation includes only `dynamic-wind` thunks (see §6.4) installed since the prompt.

When a continuation procedure is applied, it removes the portion of the current continuation up to the nearest prompt tagged by *prompt-tag* (not including the prompt; if no such prompt exists, the `exn:fail:contract:continuation` exception is raised), or up to the nearest continuation frame (if any) shared by the current and captured continuations — whichever is first. While removing continuation frames, `dynamic-wind` *post-thunk*s are executed. Finally, the (unshared portion of the) captured continuation is appended to the remaining continuation, applying `dynamic-wind` *pre-thunk*s.

`(call-with-composable-continuation` *proc* [*prompt-tag*]`)` is similar to `call-with-current-continuation` but applying the resulting continuation procedure does not remove any portion of the current continuation. Instead, application always extends the current continuation with the captured continuation (without installing any prompts other than those be captured in the continuation). When `call-with-composable-continuation` is called, if a continuation barrier appears in the continuation before the closest prompt tagged by *prompt-tag*, the `exn:fail:contract:continuation` exception is raised.

`(continuation-prompt-available?` *prompt-tag* [*cont*]`)` returns `#t` if *cont* includes a prompt tagged by *prompt-tag*, `#f` otherwise. The *cont* argument defaults to the current continuation.

6.6 Continuation Marks

To evaluate a sub-expression, MzScheme creates a continuation for the sub-expression that extends the current continuation. For example, to evaluate $expr_1$ in the expression

```
(begin
  expr1
  expr2)
```

MzScheme extends the continuation of the `begin` expression with one *continuation frame* to create the continuation for $expr_1$. In contrast, $expr_2$ is in *tail position* for the `begin` expression, so its continuation is the same as the continuation of the `begin` expression.

A *continuation mark* is a keyed mark in a continuation frame. A program can install a mark in the first frame of its current continuation, and it can extract the marks from all of the frames in any continuation (up to the nearest prompt for a specified prompt tag).

Continuation marks support debuggers and other program-tracing facilities; in particular, continuation frames roughly correspond to stack frames in traditional languages. For example, when a procedure is called, MzScheme automatically installs a continuation mark with the procedure's name and source location; when an exception occurs, the marks can be extracted from the current continuation to produce a "stack trace" for the exception.[3] A more sophisticated debugger can annotate a source program to store continuation marks that relate individual expressions to source locations.

The list of continuation marks for a key *k* and a continuation *C* that extends C_0 is defined as follows:

- If *C* is an empty continuation, then the mark list is `null`.
- If *C*'s first frame contains a mark *m* for *k*, then the mark list for *C* is `(cons` *m* l_0`)`, where l_0 is the mark list for *k* in C_0.
- If *C*'s first frame does not contain a mark keyed by *k*, then the mark list for *C* is the mark list for C_0.

The `with-continuation-mark` form installs a mark on the first frame of the current continuation:

```
(with-continuation-mark key-expr mark-expr
  body-expr)
```

The *key-expr*, *mark-expr*, and *body-expr* expressions are evaluated in order. After *key-expr* is evaluated to obtain a key and *mark-expr* is evaluated to obtain a mark, the key is mapped to the mark in the current continuation's initial frame. If the frame already has a mark for the key, it is replaced. Finally, the *body-expr* is evaluated; the continuation for evaluating *body-expr* is the continuation of the `with-continuation-mark` expression (so the result of the *body-expr* is the result of the `with-continuation-mark` expression, and *body-expr* is in tail position for the `with-continuation-mark` expression).

The `continuation-marks` procedure extracts the complete set of continuation marks from a continuation (up to a prompt), and the `continuation-mark-set->list` procedure extracts mark values for a particular key from a continuation mark set. The complete set of continuation-mark procedures follows:

- `(continuation-marks` *cont* [*prompt-tag*]`)` returns an opaque value containing the set of continuation marks for all keys in the continuation *cont* up to the prompt tagged by *prompt-tag*. If *cont* is an escape continuation (see §6.3), then the current continuation must extend *cont*, or the `exn:fail:contract` exception is raised. If *cont* was not captured with respect to *prompt-tag* and does not include a prompt for *prompt-tag*, the `exn:fail:contract` exception is raised. The *prompt-tag* argument defaults to `(default-continuation-prompt-tag)`
- `(current-continuation-marks` [*prompt-tag*]`)` returns an opaque value containing the set of continuation marks for all keys in the current continuation up to *prompt-tag*. In other words, it produces the same value as `(call-with-current-continuation (lambda (`*k*`) (continuation-marks` *k* *prompt-tag*`))` *prompt-tag*`)`. As usual, *prompt-tag* defaults to `(default-continuation-prompt-tag)`.
- `(continuation-mark-set->list` *mark-set* *key-v* [*prompt-tag*]`)` returns a newly-created list containing the marks for *key-v* in *mark-set*, which is a set of marks returned by `current-continuation-marks`. The result list is truncated at the first point, if any, where continuation frames were originally separated by a prompt tagged with *prompt-tag*. As usual, *prompt-tag* defaults to `(default-continuation-prompt-tag)`.

[3]Since stack-trace marks are applied dynamically, they do not necessarily correspond to uses of `with-continuation-mark` on the source, and stack-trace marks can be affected by optimization or just-in-time compilation of the code. A stack traces is therefore useful as a debugging hint only.

- `(continuation-mark-set->list*` *`mark-set key-list`* `[`*`none-v prompt-tag`*`])` returns a newly-created list containing vectors of marks in *`mark-set`* for the keys in *`key-list`*, up to *`prompt-tag`*. The length of each vector in the result list is the same as the length of *`key-list`*, and a value in a particular vector position is the value for the corresponding key in *`key-list`*. Values for multiple keys appear in a single vector only when the marks are for the same continuation frame in *`mark-set`*. If *`none-v`* is supplied, it is used for vector elements to indicate the lack of a value; the default is `#f`.
- `(continuation-mark-set-first` *`optional-mark-set key-v`* `[`*`prompt-tag`*`])` returns the first element of the list that would be returned by `(continuation-mark-set->list (or` *`optional-mark-set`* `(current-continuation-marks` *`prompt-tag`*`))` *`key-v prompt-tag`*`)`, or `#f` if the result would be the empty list. Typically, this result can be computed more quickly using `continuation-mark-set-first`.
- `(continuation-mark-set?` *`v`*`)` returns `#t` if *`v`* is a mark set created by `continuation-marks` or `current-continuation-marks`, `#f` otherwise.
- `(continuation-mark-set->context` *`mark-set`*`)` returns a list representing a "stack trace" for *`mark-set`*'s continuation. The list contains pairs, where the `car` of each pair contains either `#f` or a symbol for a procedure name, and the `cdr` of each pair contains either `#f` or a `srcloc` value for the procedure's source location (see §11.2.1.1); the `car` and `cdr` are never both `#f`.

  The stack-trace list is the result of `continuation-mark-set->list` with *`mark-set`* and MzScheme's private key for procedure-call marks. A stack trace is extracted from an exception and displayed by the default error display handler (see §6) for exceptions other than `exn:fail:user` (see `raise-user-error` in §6.2).

Examples:

```
(define (extract-current-continuation-marks key)
   (continuation-mark-set->list
    (current-continuation-marks)
    key))

(with-continuation-mark 'key 'mark
  (extract-current-continuation-marks 'key)) ; ⇒ '(mark)

(with-continuation-mark 'key1 'mark1
  (with-continuation-mark 'key2 'mark2
    (list
     (extract-current-continuation-marks 'key1)
     (extract-current-continuation-marks 'key2)))) ; ⇒ '((mark1) (mark2))

(with-continuation-mark 'key 'mark1
  (with-continuation-mark 'key 'mark2 ; replaces the previous mark
    (extract-current-continuation-marks 'key)))) ; ⇒ '(mark2)

(with-continuation-mark 'key 'mark1
  (list ; continuation extended to evaluate the argument
   (with-continuation-mark 'key 'mark2
      (extract-current-continuation-marks 'key)))) ; ⇒ '((mark1 mark2))

(let loop ([n 1000])
  (if (zero? n)
      (extract-current-continuation-marks 'key)
      (with-continuation-mark 'key n
        (loop (sub1 n))))) ; ⇒ '(1)
```

In the final example, the continuation mark is set 1000 times, but *`extract-current-continuation-marks`* returns only one mark value. Because *`loop`* is called tail-recursively, the continuation of each call to *`loop`* is always the continuation of the entire expression. Therefore, the `with-continuation-mark` expression replaces the existing mark each time rather than adding a new one.

Whenever MzScheme creates an exception record, it fills the `continuation-marks` field with the value of `(current-continuation-marks)`, thus providing a snapshot of the continuation marks at the time of the exception.

When a continuation procedure returned by `call-with-current-continuation` or `call-with-composable-continuat` is invoked, it restores the captured continuation, and also restores the marks in the continuation's frames to the marks that were present when `call-with-current-continuation` or `call-with-composable-continuation` was invoked.

6.7 Breaks

A *break* is an asynchronous exception, usually triggered through an external source controlled by the user, or through the `break-thread` procedure (see §7.3). A break exception can only occur in a thread while breaks are enabled. When a break is detected and enabled, the `exn:break` exception is raised in the thread sometime afterward; if breaking is disabled when `break-thread` is called, the break is suspended until breaking is again enabled for the thread. While a thread has a suspended break, additional breaks are ignored.

Breaks are enabled through the `break-enabled` parameter-like procedure, and through the `parameterize-break` form, which is analogous to `parameterize` (see §7.9). The `break-enabled` procedure does not represent a parameter to be used with `parameterize`, because changing the break-enabled state of a thread requires an explicit check for breaks, and this check is incompatible with the tail evaluation of a `parameterize` expression's body.

- `(break-enabled` [*`on?`*]`)` — gets or sets the break enabled state of the current thread. If *`on?`* is not supplied, the result is `#t` if break are currently enabled, `#f` otherwise. If *`on?`* is supplied as `#f`, breaks are disabled, and if *`on?`* is a true value, breaks are enabled.

- `(parameterize-break` *`boolean-expr expr`* $\cdots^1$`)` evaluates *`boolean-expr`* to determine whether breaks are initially enabled in while evaluating *`expr`*s in sequence. The result of the *`parameter-break`* expression is the result of the last *`expr`*.

  Like `parameterize` (see §7.9), a fresh thread cell (see §7.8) is allocated to hold the break-enabled state of the continuation, and calls to `break-enabled` within the continuation access or modify the new cell.

- `(current-break-parameterization)` is analogous to `(current-parameterization)` (see §7.9); it returns a break-parameterization (effectively a thread cell) that holds the current continuation's break-enable state.

- `(call-with-break-parameterization` *`break-param thunk`*`)` is analogous to `(call-with-parameterizat` `parameterization thunk)` (see §7.9); it calls *`thunk`* in a continuation whose break-enabled state is in *`break-param`*. The *`thunk`* is *not* called in tail position with respect to the `call-with-break-parameterization` call.

Certain procedures, such as `semaphore-wait/enable-break`, enable breaks temporarily while performing a blocking action. If breaks are enabled for a thread, and if a break is triggered for the thread but not yet delivered as an `exn:break` exception, then the break is guaranteed to be delivered before breaks can be disabled in the thread. The timing of `exn:break` exceptions is not guaranteed in any other way.

Before calling a `with-handlers` predicate or handler, an exception handler, an error display handler, an error escape handler, an error value conversion handler, or a *`pre-thunk`* or *`post-thunk`* for a `dynamic-wind` (see

§6.4), the call is parameterize-breaked to disable breaks. Furthermore, breaks are disabled during the transitions among handlers related to exceptions, during the transitions between *pre-thunk*s and *post-thunk*s for dynamic-wind, and during other transitions for a continuation jump. For example, if breaks are disabled when a continuation is invoked, and if breaks are also disabled in the target continuation, then breaks will remain disabled until from the time of the invocation until the target continuation executes unless a relevant dynamic-wind *pre-thunk* or *post-thunk* explicitly enables breaks.

If a break is triggered for a thread that is blocked on a nested thread (see call-in-nested-thread), and if breaks are enabled in the blocked thread, the break is implicitly handled by transferring it to the nested thread.

When breaks are enabled, they can occur at any point within execution, which makes certain implementation tasks subtle. For example, assuming breaks are enabled when the following code is executed,

```
(with-handlers ([exn:break? (lambda (x) (void))])
  (semaphore-wait s))
```

then it is *not* the case that a void result means the semaphore was decremented or a break was received, *exclusively*. It is possible that *both* occur: the break may occur after the semaphore is successfully decremented but before a void result is returned by semaphore-wait. A break exception will never damage a semaphore, or any other built-in construct, but many built-in procedures (including semaphore-wait) contain internal sub-expressions that can be interrupted by a break.

In general, it is impossible using only semaphore-wait to implement the guarantee that either the semaphore is decremented or an exception is raised, but not both. MzScheme therefore supplies semaphore-wait/enable-break (see §7.4), which does permit the implementation of such an exclusive guarantee:

```
(parameterize ([break-enabled #f])
  (with-handlers ([exn:break? (lambda (x) (void))])
    (semaphore-wait/enable-break s)))
```

In the above expression, a break can occur at any point until breaks are disabled, in which case a break exception is propagated to the enclosing exception handler. Otherwise, the break can only occur within semaphore-wait/enable-break, which guarantees that if a break exception is raised, the semaphore will not have been decremented.

To allow similar implementation patterns over blocking port operations, MzScheme provides read-bytes-avail!/enable-brea (see §11.2.1), write-bytes-avail/enable-break (see §11.2.2), and other procedures.

6.8 Error Escape Handler

Special control flow for exceptions is performed by an *error escape handler* that is called by the default exception handler. An error escape handler takes no arguments and must escape from the expression that raised the exception. The error escape handler is obtained or set using the error-escape-handler parameter (see §7.9.1.7).

An error escape handler cannot invoke a full continuation that was created prior to the exception, but it can abort to a prompt (see §6.5) or invoke an escape continuation (see §6.3).

The error escape handler is normally called directly by an exception handler, in a parameterization that sets the error display and escape handlers to the default handlers, and parameterize-breaked to disable breaks. To escape from a run-time error, use raise (see §6.1) or error (see §6.2).

The default error escape handler escapes using (*abort-current-continuation* (default-continuation-prompt-tag void). Thus, error escapes are best handled by installing a prompt, rather than by changing the error escape handler.

In the following example, a prompt is installed that so errors do not escape from a custom `read-eval-print` loop:

```
(let retry-loop ()
  (call-with-continuation-prompt
   (lambda ()
     (let loop ()
       (let ([e (my-read)])
         (unless (eof-object? e)
           (let ([v (my-eval e)])
             (my-print v))
           (loop)))))
   (default-continuation-prompt-tag)
   (lambda args (retry-loop))))
```

See also `read-eval-print-loop` in §14.1 for a simpler and more complete implementation of a custom `read-eval-print` loop.

7. Threads

MzScheme supports multiple threads of control within a program. Threads are implemented for all operating systems, even when the operating system does not provide primitive thread support.

`(thread` *`thunk`*`)` invokes the procedure *`thunk`* with no arguments in a new thread of control. The `thread` procedure returns immediately with a *thread descriptor* value. When the invocation of *`thunk`* returns, the thread created to invoke *`thunk`* terminates.

Example:

```
(thread (lambda () (sleep 2) (display 7) (newline))) ; ⇒ a thread descriptor
```

`displays 7` after two seconds pass

Each thread has its own parameter settings (see §7.9), such as the current directory or current exception handler. A newly-created thread inherits the parameter settings of the creating thread.

When a thread is created, it is placed into the management of the current custodian (see §9.2) and added to the current thread group (see §9.3). A thread can have any number of custodian managers added through `thread-resume`.

A thread that has not terminated can be "garbage collected" if it is unreachable and suspended, or if it is unreachable and blocked on a set of unreachable events through `semaphore-wait` or `semaphore-wait/enable-break` (see §7.4), `channel-put` or `channel-get` (see §7.5), `sync` or `sync/enable-break` (see §7.7), or `thread-wait`.[1]

All constant-time procedures and operations provided by MzScheme are thread-safe because they are *atomic*. For example, `set!` assigns to a variable as an atomic action with respect to all threads, so that no thread can see a "half-assigned" variable. Similarly, `vector-set!` assigns to a vector atomically. The `hash-table-put!` procedure is not atomic, but the table is protected by a lock; see §3.14 for more information. Port operations are generally not atomic, but they are thread-safe in the sense that a byte consumed by one thread from an input port will not be returned also to another thread, and procedures like `port-commit-peeked` (see §11.2.1) and `write-bytes-avail` (see §11.2.2) offer specific concurrency guarantees.

7.1 Suspending, Resuming, and Killing Threads

`(thread-suspend` *`thread`*`)` immediately suspends the execution of *`thread`* if it is running. If the thread has terminated or is already suspended, `thread-suspend` has no effect. The thread remains suspended (i.e., it does not execute) until it is resumed with `thread-resume`. If the current custodian (see §9.2) does not manage *`thread`* (and none of its subordinates manages *`thread`*), the `exn:fail:contract` exception is raised, and the thread is not suspended.

`(thread-resume` *`thread`* [*`thread-or-custodian`*]`)` resumes the execution of *`thread`* if it is suspended and has at least one custodian (possibly added through *`thread-or-custodian`*, as described below). If the thread

[1] In MrEd, a handler thread for an eventspace is blocked on an internal semaphore when its event queue is empty. Thus, the handler thread is collectible when the eventspace is unreachable and contains no visible windows or running timers.

has terminated, or if the thread is already running and *thread-or-custodian* is not supplied, or if the thread has no custodian and *thread-or-custodian* is not supplied, then `thread-resume` has no effect. Otherwise, if *thread-or-custodian* is supplied, it triggers up to three additional actions:

- If *thread-or-custodian* is a thread, whenever it is resumed from a suspended state in the future, then *thread* is also resumed. (Resuming *thread* may trigger the resumption of other threads that were previously attached to *thread* through `thread-resume`.)
- New custodians may be added to *thread*'s set of managers. If *thread-or-custodian* is a thread, then all of the thread's custodians are added to *thread*. Otherwise, *thread-or-custodian* is a custodian, and it is added to *thread* (unless the custodian is already shut down). If *thread* becomes managed by both a custodian and one or more of its subordinates, the redundant subordinates are removed from *thread*. If *thread* is suspended and a custodian is added, then *thread* is resumed only after the addition.
- If *thread-or-custodian* is a thread, whenever it receives a new managing custodian in the future, then *thread* also receives the custodian. (Adding custodians to *thread* may trigger adding the custodians to other threads that were previously attached to *thread* through `thread-resume`.)

`(kill-thread` *thread*`)` terminates the specified thread immediately, or suspends the thread if *thread* was created with `thread/suspend-to-kill`. Terminating the main thread exits the application. If *thread* has already terminated, `kill-thread` does nothing. If the current custodian (see §9.2) does not manage *thread* (and none of its subordinates manages *thread*), the `exn:fail:contract` exception is raised, and the thread is not killed or suspended.

Unless otherwise noted, procedures provided by MzScheme (and MrEd) are kill-safe and suspend-safe; that is, killing or suspending a thread never interferes with the application of procedures in other threads. For example, if a thread is killed while extracting a character from an input port, the character is either completely consumed or not consumed, and other threads can safely use the port.

`(thread/suspend-to-kill` *thunk*`)` is like `(thread` *thunk*`)`, except that "killing" the current thread through `kill-thread` or `custodian-shutdown-all` (see §9.2) merely suspends the thread instead of terminating it.

7.2 Synchronizing Thread State

`(thread-wait` *thread*`)` blocks execution of the current thread until *thread* has terminated. Note that `(thread-wait (current-thread))` deadlocks the current thread, but a break can end the deadlock (if breaking is enabled; see §6.7).

`(thread-dead-evt` *thread*`)` returns a synchronizable event (see §7.7) that is ready if and only if *thread* has terminated. Unlike using *thread* directly, however, a reference to the event does not prevent *thread* from being "garbage collected."

`(thread-resume-evt` *thread*`)` returns a synchronizable event (see §7.7) that becomes ready when *thread* is running. (If *thread* has terminated, the event never becomes ready.) If *thread* runs and is then suspended after a call to `thread-resume-evt`, the result event remains ready; after each suspend of *thread* a fresh event is generated to be returned by `thread-resume-evt`. The result of the event is *thread*, but if *thread* is never resumed, then reference to the event does not prevent *thread* from being "garbage collected."

`(thread-suspend-evt` *thread*`)` returns a synchronizable event (see §7.7) that becomes ready when *thread* is suspended. (If *thread* has terminated, the event will never unblock.) If *thread* is suspended and then resumes after a call to `thread-suspend-evt`, the result event remains ready; after each resume of *thread* created a fresh event to be returned by `thread-suspend-evt`.

7.3 Additional Thread Utilities

`(current-thread)` returns the thread descriptor for the currently executing thread.

`(thread? v)` returns `#t` if *v* is a thread descriptor, `#f` otherwise.

`(sleep [x])` causes the current thread to sleep until at least *x* seconds have passed after it starts sleeping, where *x* is a non-negative real number. The *x* argument defaults to 0, which simply acts as a hint to allow other threads to execute. The value of *x* can be non-integral to request a sleep duration to any precision; the precision of the actual sleep time is unspecified.

`(thread-running? thread)` returns `#t` if *thread* has not terminated and is not suspended, `#f` otherwise.

`(thread-dead? thread)` returns `#t` if *thread* has terminated, `#f` otherwise.

`(break-thread thread)` registers a break with the specified thread. If breaking is disabled in *thread*, the break will be ignored until breaks are re-enabled (see §6.7).

`(call-in-nested-thread thunk [custodian])` creates a nested thread managed by *custodian* to execute *thunk*.[2] The current thread blocks until *thunk* returns, and the result of the `call-in-nested-thread` call is the result returned by *thunk*. The default value of *custodian* is the current custodian (see §9.2).

The nested thread's exception handler is initialized to a procedure that jumps to the beginning of the thread and transfers the exception to the original thread. The handler thus terminates the nested thread and re-raises the exception in the original thread.

If the thread created by `call-in-nested-thread` dies before *thunk* returns, the `exn:fail` exception is raised in the original thread. If the original thread is killed before *thunk* returns, a break is queued for the nested thread.

If a break is queued for the original thread (with `break-thread`) while the nested thread is running, the break is redirected to the nested thread. If a break is already queued on the original thread when the nested thread is created, the break is moved to the nested thread. If a break remains queued on the nested thread when it completes, the break is moved to the original thread.

7.4 Semaphores

A *semaphore* is a value that is used to synchronize MzScheme threads. Each semaphore has an internal counter; when this counter is zero, the semaphore can block a thread's execution (through `semaphore-wait`) until another thread increments the counter (using `semaphore-post`). The maximum value for a semaphore's internal counter is platform-specific, but always at least 10000.

A semaphore's counter is updated in a single-threaded manner, so that semaphores can be used for reliable synchronization. Semaphore waiting is *fair*: if a thread is blocked on a semaphore and the semaphore's internal value is non-zero infinitely often, then the thread is eventually unblocked.

- `(make-semaphore [init-k])` creates and returns a new semaphore with the counter initially set to *init-k*, which defaults to `0`. If *init-k* is larger than a semaphore's maximum internal counter value, the `exn:fail:contract` exception is raised.
- `(semaphore? v)` returns `#t` if *v* is a semaphore created by `make-semaphore`, `#f` otherwise.
- `(semaphore-post sema)` increments the semaphore's internal counter and returns void. If the semaphore's internal counter has already reached its maximum value, the `exn:fail` exception is raised.

[2] The nested thread's current custodian is inherited from the creating thread, independent of the *custodian* argument.

- `(semaphore-wait` *`sema`*`)` blocks until the internal counter for semaphore *`sema`* is non-zero. When the counter is non-zero, it is decremented and `semaphore-wait` returns void.

- `(semaphore-try-wait?` *`sema`*`)` is like `semaphore-wait`, but `semaphore-try-wait?` never blocks execution. If *`sema`*'s internal counter is zero, `semaphore-try-wait?` returns `#f` immediately without decrementing the counter. If *`sema`*'s counter is positive, it is decremented and `#t` is returned.

- `(semaphore-wait/enable-break` *`sema`*`)` is like `semaphore-wait`, but breaking is enabled (see §6.7) while waiting on *`sema`*. If breaking is disabled when `semaphore-wait/enable-break` is called, then either the semaphore's counter is decremented or the `exn:break` exception is raised, but not both.

- `(semaphore-peek-evt` *`sema`*`)` creates and returns a new synchronizable event (for use with `sync`, for example) that is ready when *`sema`* is ready, but synchronizing the event does not decrement *`sema`*'s internal count.

- `(call-with-semaphore` *`sema proc`* [*`try-fail-thunk arg`* ···]`)` waits on *`sema`* using `semaphore-wait`, calls *`proc`* with all *`arg`*s, and then posts to *`sema`*. A continuation barrier blocks full continuation jumps into or out of *`proc`* (see §6.3), but escape jumps are allowed, and *`sema`* is posted on escape. If *`try-fail-thunk`* is provided and is not `#f`, then `semaphore-try-wait?` is called on *`sema`* instead of `semaphore-wait`, and *`try-fail-thunk`* is called if the wait fails.

- `(call-with-semaphore/enable-break` *`sema proc`* [*`try-fail-thunk arg`* ···]`)` is like `call-with-semaphore`, except that `semaphore-wait/enable-break` is used with `sema` in non-try mode. When *`try-fail-thunk`* is provided and not `#f`, then breaks are enabled around the use of `semaphore-try-wait?` on *`sema`*.

See also `sync` in §7.7.

7.5 Channels

A *synchronous channel* is a value that is used to synchronize MzScheme threads: one thread sends a value to another thread, and both the sender and the receiver block until the (atomic) transaction is complete. Multiple senders and receivers can access a channel at once, but a single sender and receiver is selected for each transaction.

Channel synchronization is *fair*: if a thread is blocked on a channel and transaction opportunities for the channel occur infinitely often, then the thread eventually participates in a transaction.

For buffered asynchronous channels, see Chapter 4 of *PLT MzLib: Libraries Manual*.

- `(make-channel)` creates and returns a new channel. The channel can be used with `channel-get`, with `channel-try-get`, or as a synchronizable event (see §7.7) to receive a value through the channel. The channel can be used with `channel-put` or through the result of `channel-put-evt` to send a value through the channel.

- `(channel?` *`v`*`)` returns `#t` if *`v`* is a channel created by `make-channel`, `#f` otherwise.

- `(channel-get` *`channel`*`)` blocks until a sender is ready to provide a value through *`channel`*. The result is the sent value.

- `(channel-try-get` *`channel`*`)` receives and returns a value from *`channel`* if a sender is immediately ready, otherwise returns `#f`.

- `(channel-put` *`channel v`*`)` blocks until a receiver is ready to accept the value *`v`* through *`channel`*. The result is void.

- `(channel-put-evt` *`channel v`*`)` returns a fresh synchronizable event for use with `sync` (see §7.7). The event is ready when `(channel-put` *`channel v`*`)` would not block, and the event's synchronization result is the event itself.

7.6 Alarms

An *alarm* is a synchronizable event (see §7.7) that is ready only after particular date and time. The time is specified as a real number that is consistent with `current-inexact-milliseconds` (see §15.1.2).

`(alarm-evt` *`msecs-n`*`)` returns a synchronizable event for use with `sync`. The event is not ready when `(current-inexact-milliseconds)` would return a value that is less than *`msecs-n`*, and it is ready when `(current-inexact-milliseconds)` would return a value that is more than *`msecs-n`*.

The `sync` function accepts a timeout argument in addition to alarm events. Unlike the timeout, however, the result of `alarm-evt` can be combined with `wrap-evt` and other event operations.

7.7 Synchronizing Events

`(sync` *`evt`* $\cdots^{1}$`)` blocks as long as none of the synchronizable events *`evt`*s are ready, as defined below. Certain kinds of objects double as events, including ports and threads, and other kinds of objects exist only for their use as events.

`(sync/timeout` *`timeout evt`* $\cdots^{1}$`)` is like `sync`, but with a timeout. If no *`evt`* is ready before *`timeout`* seconds have passed, the result is `#f`. The *`timeout`* argument can be a real number or `#f`; if *`timeout`* is `#f`, then `sync/timeout` behaves like `sync`. If *`timeout`* is `0`, each *`evt`* is checked at least once, so a *`timeout`* value of `0` can be used for polling. (See `alarm-evt` in §7.6 for an alternative timeout mechanism.)

For either `sync` or `sync/timeout`, when at least one *`evt`* is ready, its result (often *`evt`* itself) is returned. If multiple *`evt`*s are ready, one of the *`evt`*s is chosen pseudo-randomly for the result. (The `current-evt-pseudo-random-generator` parameter sets the random-number generator that controls this choice; see §7.9.1.10.)

Choosing a ready *`evt`* may affect the state of *`evt`*. For example, if the chosen ready *`evt`* is a semaphore, then the semaphore's internal count is decremented, just as with `semaphore-wait`. For most kinds of events, however (such as a port), *`evt`*'s state is not modified.

Only certain kinds of built-in values, listed below, act as events in stand-alone MzScheme. If any other kind of value is provided to `sync`, the `exn:fail:contract` exception is raised. An extension or embedding application can extend the set of primitive events — in particular, an eventspace in MrEd is an event — and new structure types can generate events (see §4.7).

- *`semaphore`* — a semaphore is ready only when `semaphore-wait` (see §7.4) would not block. The synchronization result of *`semaphore`* is *`semaphore`* itself.
- *`semaphore-peek`* — a semaphore returned by `semaphore-peek-evt` applied to *`semaphore`* (see §7.4) is ready exactly when *`semaphore`* is ready. The synchronization result of *`semaphore-peek`* is *`semaphore-peek`* itself.
- *`channel`* — a channel returned by `make-channel` is ready when `channel-get` would not block (see §7.5). The channel's result as an event is the same as the `channel-get` result.
- *`channel-put`* — an event returned by `channel-put-evt` applied to *`channel`* is ready when `channel-put` would not block on *`channel`* (see §7.5). The synchronization result of *`channel-put`* is *`channel-put`* itself.

- *input-port* — an input port is ready as an event when *read-byte* would not block. The synchronization result of *input-port* is *input-port* itself.
- *output-port* — an output port is ready when *write-bytes-avail* would not block (see §11.2.2) or when the port contains buffered characters and *write-bytes-avail** can flush part of the buffer (although *write-bytes-avail* might block). The synchronization result of *output-port* is *output-port* itself.
- *progress* — an event produced by `port-progress-evt` applied to *input-port* is ready after any subsequent read from *input-port*. The synchronization result of *progress* is *progress* itself.
- *tcp-listener* — a TCP listener is ready when `tcp-accept` (see §11.4.1) would not block. The synchronization result of *listener* is *listener* itself.
- *thread* — a thread is ready when `thread-wait` (see §7.2) would not block. The synchronization result of *thread* is *thread* itself.
- *thread-dead* — an event returned by `thread-dead-evt` (see §7.2) applied to *thread* is ready when *thread* has terminated. The synchronization result of *thread-dead* is *thread-dead* itself.
- *thread-resume* — an event returned by `thread-resume-evt` (see §7.2) applied to *thread* is ready when *thread* subsequently resumes execution (if it was not already running). The event's result is *thread*.
- *thread-suspend* — an event returned by `thread-suspend-evt` (see §7.2) applied to *thread* is ready when *thread* subsequently suspends execution (if it was not already suspended). The event's result is *thread*.
- *alarm* — an event returned by `alarm-evt` (see §7.6) is ready after a particular date and time. The synchronization result of *alarm* is *alarm* itself.
- *subprocess* — a subprocess is ready when `subprocess-wait` (see §15.2) would not block. The synchronization result of *subprocess* is *subprocess* itself.
- *will-executor* — a will executor is ready when `will-execute` (see §13.3) would not block. The synchronization result of *will-executor* is *will-executor* itself.
- *udp* — an event returned by `udp-send-evt` or `udp-receive!-evt` (see §11.4.2) is ready when a send or receive on the original socket would block, respectively. The synchronization result of *udp* is *udp* itself.
- *choice* — an event returned by `choice-evt` (see below) is ready when one or more of the *evt*s supplied to `chocie-evt` are ready. If the choice event is chosen, one of its ready *evt*s is chosen pseudo-randomly, and the result is the chosen *evt*'s result.
- *wrap* — an event returned by `wrap-evt` applied to *evt* and *proc* is ready when *evt* is ready. The event's result is obtained by a call to *proc* (with breaks disabled) on the result of *evt*.
- *handle* — an event returned by `handle-evt` applied to *evt* and *proc* is ready when *evt* is ready. The event's result is obtained by a tail call to *proc* on the result of *evt*.
- *guard* — an event returned by `guard-evt` applied to *thunk* generates a new event every time that *guard* is used with `sync` (or whenever it is part of a choice event used with `sync`, etc.); the generated event is the result of calling *thunk* when the synchronization begins; if *thunk* returns a non-event, then *thunk*'s result is replaced with an event that is ready and whose result is *guard*.
- *nack-guard* — an event returned by `nack-guard-evt` applied to *proc* generates a new event every time that *nack-guard* is used with `sync` (or whenever it is part of a choice event used with `sync`, etc.); the generated event is the result of calling *proc* with a NACK ("negative acknowledgment") event when the synchronization begins; if *proc* returns a non-event, then *proc*'s result is replaced with an event that is ready and whose result is *nack-guard-evt*.

If the event from *proc* is not ultimately chosen as the unblocked event, then the NACK event supplied to *proc* becomes ready with a void value. This NACK event becomes ready when the event is abandoned because some other event is chosen, because the synchronizing thread is dead, or because control escaped from the call to `sync` (even if *nack-guard*'s *proc* has not yet returned a value). If the event returned by *proc* is chosen, then the NACK event never becomes ready.

- *poll-guard* — an event returned by `poll-guard-evt` applied to *proc* generates a new event every time that *poll-guard* is used with `sync` (or whenever it is part of a choice event used with `sync`, etc.); the generated event is the result of calling *proc* with a boolean: `#t` if the event will be used for a poll, `#f` for a blocking synchronization.

  If `#t` is supplied to *proc*, if breaks are disabled, if the polling thread is not terminated, and if polling the resulting event produces a result, the event will certainly be chosen for its result.

- *struct* — a structure whose type has the `prop:evt` property identifies/generates an event through the property; see §4.7 for further information.

- `always-evt` — a constant event that is always ready. The synchronization result of `always-evt` is `always-evt` itself.

- `never-evt` — a constant event that is never ready.

- *idle-evt* — an event produced by `system-idle-evt` is ready when, if this event were replaced by `never-evt`, no thread in the system would be available to run. In other words, all threads must be suspended or blocked on events with timeouts that have not yet expired. The event's result is void.

`(sync/enable-break` *evt* `···`^{1}`)` is like `sync`, but breaking is enabled (see §6.7) while waiting on the *evt*s. If breaking is disabled when `sync/enable-break` is called, then either all *evt*s remain unchosen or the `exn:break` exception is raised, but not both.

`(sync/timeout/enable-break` *timeout evt* `···`^{1}`)` is like `sync/enable-break`, but with a timeout in seconds (or `#f`, as for `sync/timeout`).

`(choice-evt` *evt* `···)` creates and returns a single event that combines the *evt*s. Supplying the result to `sync` is the same as supplying each *evt* to the same call.

`(wrap-evt` *evt wrap-proc*`)` creates an event that is in a ready when *evt* is ready, but whose result is determined by applying *wrap-proc* to the result of *evt*. The call to *wrap-proc* is `parameterize-break`ed to disable breaks initially. The *evt* cannot be an event created by `handle-evt` or any combination of `choice-evt` involving an event from `handle-evt`.

`(handle-evt` *evt handle-proc*`)` is like *wrap-evt*, except that *handle-proc* is called in tail position with respect to the synchronization request, and without breaks explicitly disabled.

`(guard-evt` *generator-thunk*`)` creates a value that behaves as an event, but that is actually an event generator. For details, see `sync`, above.

`(nack-guard-evt` *generator-proc*`)` creates a value that behaves as an event, but that is actually an event generator; the generator procedure receives an event that becomes ready with a void value if the generated event was not ultimately chosen. For details, see `sync`, above.

`(poll-guard-evt` *generator-proc*`)` creates a value that behaves as an event, but that is actually an event generator; the generator procedure receives a boolean indicating whether the event is used for polling. For details, see `sync`, above.

`always-evt` is a global constant event that is always ready, with itself as its result.

`never-evt` is a global constant event that is never ready.

`(system-idle-evt)` returns an event that is ready when the system is otherwise idle; see `sync` above for more information. The result of the `system-idle-evt` procedure is always the same event.

`(evt? v)` returns `#t` if *v* is a synchronizable event, `#f` otherwise. See `sync`, above, for the list of built-in types that act as synchronizable events.

`(handle-evt? evt)` returns `#t` if *evt* was created by `handle-evt` or by `choice-evt` applied to another event for which `handle-evt?` produces `#t`. Such events are illegal as an argument to `handle-evt` or `wrap-evt`, because they cannot be wrapped further. For any other event, `handle-evt?` produces `#f`, and the event is a legal argument to `handle-evt` or `wrap-evt` for further wrapping.

7.8 Thread-Local Storage Cells

A *thread cell* contains a thread-specific value; that is, it contains a specific value for each thread, but it may contain different values for different threads. A thread cell is created with a default value that is used for all existing threads. When the cell's content is changed with `thread-cell-set!`, the cell's value changes only for the current thread. Similarly, `thread-cell-ref` obtains the value of the cell that is specific to the current thread.

A thread cell's value can be *preserved*, which means that when a new thread is created, the cell's initial value for the new thread is the same as the creating thread's current value. If a thread cell is non-preserved, then the cell's initial value for a newly created thread is the default value (which was supplied when the cell was created).

Within the current thread, the current values of all preserved threads cells can be captured through `current-preserved-thread-cell-values`. The captured set of values can be imperatively installed into the current thread through another call to `current-preserved-thread-cell-values`. The capturing and restoring threads can be different.

- `(make-thread-cell v [preserved?])` creates and returns a new thread cell. Initially, *v* is the cell's value for all threads. If *preserved?* is true, then the cell's initial value for a newly created threads is the creating thread's value for the cell, otherwise the cell's value is initially *v* in all future threads. The default value of *preserved?* is `#f`.
- `(thread-cell? v)` returns `#t` if *v* is a thread cell created by `make-thread-cell`, `#f` otherwise.
- `(thread-cell-ref cell)` returns the current value of *cell* for the current thread.
- `(thread-cell-set! cell v)` sets the value in *cell* to *v* for the current thread.
- `(current-preserved-thread-cell-values [thread-cell-vals])` when called with no arguments produces a *thread-cell-vals* that represents the current values (in the current thread) for all preserved thread cells. When called with a *thread-cell-vals* generated by a previous call to `current-preserved-thread-cell-values`, the values of all preserved thread cells (in the current thread) are set to the values captured in *thread-cell-vals*; if a preserved thread cell was created after *thread-cell-vals* was generated, then the thread cell's value for the current thread reverts to its initial value.

Examples:

```
(define cnp (make-thread-cell '(nerve) #f))
(define cp (make-thread-cell '(cancer) #t))

(thread-cell-ref cnp) ; ⇒ '(nerve)
(thread-cell-ref cp) ; ⇒ '(cancer)
```

```
(thread-cell-set! cnp '(nerve nerve))
(thread-cell-set! cp '(cancer cancer))

(thread-cell-ref cnp) ; ⇒ '(nerve nerve)
(thread-cell-ref cp) ; ⇒ '(cancer cancer)

(define ch (make-channel))
(thread (lambda ()
          (channel-put ch (thread-cell-ref cnp))
          (channel-put ch (thread-cell-ref cp))
          (channel-get ch) ; to wait
          (channel-put ch (thread-cell-ref cp))))

(channel-get ch) ; ⇒ '(nerve)
(channel-get ch) ; ⇒ '(cancer cancer)

(thread-cell-set! cp '(cancer cancer cancer))

(thread-cell-ref cp) ; ⇒ '(cancer cancer cancer)
(channel-put ch 'ok)
(channel-get ch) ; ⇒ '(cancer cancer)
```

7.9 Parameters

A *parameter* is a setting that is both thread-specific and continuation-specific, such as the current output port or the current directory for resolving relative file paths. A *parameter procedure* retrieves and sets the value of a specific parameter. For example, the `current-output-port` parameter procedure sets and retrieves a port value that is used by `display` when a specific output port is not provided. Applying a parameter procedure without an argument obtains the current value of a parameter in the current thread and continuation, and applying a parameter procedure to a single argument sets the parameter's value in the current thread and continuation (returning void). For example, `(current-output-port)` returns the current default output port, while `(current-output-port p)` sets the default output port to *p*.

In the empty continuation, each parameter corresponds to a preserved thread cell (see §7.8); the parameter procedure accesses and sets the thread cell's value (for the current thread). To parameterize code in a continuation-friendly manner, use `parameterize`. The `parameterize` form introduces a fresh thread cell for the dynamic extent of its body expressions. The syntax of `parameterize` is:

```
(parameterize ((parameter-expr value-expr) ···) body-expr ···1)
```

The result of a `parameterize` expression is the result of the last *body-expr*. The *parameter-expr*s determine the parameters to set, and the *value-expr*s determine the corresponding values to install while evaluating the *body-expr*s. All of the *parameter-expr*s are evaluated first (and checked with `parameter?`), then all *value-expr*s are evaluated, and then the parameters are bound in the continuation to preserved thread cells that contain the values of the *value-expr*s. The last *body-expr* is in tail position with respect to the entire `parameterize` form.

Outside the dynamic extent of a `parameterize` expression, parameters remain bound to other thread cells. Effectively, therefore, old parameters settings are restored as control exits the `parameterize` expression.

If a continuation is captured during the evaluation of `parameterize`, invoking the continuation effectively reintroduces the parameterization. More generally, a continuation's parameter-to-thread-cell mapping is called a *parameterization*, and a parameterization is associated to a continuation via a continuation mark (see §6.6) using a private key. The `current-parameterization` procedure returns the current continuation's parameterization. The `call-with-parameterization` procedure takes a parameterization and a thunk; it sets the current continuation's parameterization to the given one, and calls the thunk through a tail call.

When a new thread is created, the parameterization for the new thread's initial continuation is the parameterization of the creator thread. Since each parameter's thread cell is preserved, the new thread "inherits" the parameter values of its creating thread. When a continuation is moved from one thread to another, settings introduced with `parameterize` effectively move with the continuation. In contrast, direct assignment to a parameter (by calling the parameter procedure with a value) changes the value in a thread cell, and therefore changes the setting only for the current thread. (Consequently, as far as the memory manager is concerned, the value originally associated with a parameter through `parameterize` remains reachable as long the continuaton is reachable, even if the parameter is mutated.)

The `parameterize*` form has the same syntax as `parameterize`:

```
(parameterize* ((parameter-expr value-expr) ···) body-expr ···1)
```

Analogous to `let*` compared to `let`, `parameterize*` is the same as a nested series of single-parameter `parameterize` forms.

Examples:

```
(parameterize ([exit-handler (lambda (x) 'no-exit)])
  (exit)) ; ⇒ void

(define p1 (make-parameter 1))
(define p2 (make-parameter 2))
(parameterize ([p1 3]
               [p2 (p1)])
  (cons (p1) (p2))) ; ⇒ '(3 . 1)

(let ([k (let/cc out
           (parameterize ([p1 2])
             (p1 3)
             (cons (let/cc k
                     (out k))
                   (p1))))])
  (if (procedure? k)
      (k (p1))
      k)) ; ⇒ '(1 . 3)

(define ch (make-channel))
(parameterize ([p1 0])
  (thread (lambda ()
            (channel-put ch (cons (p1) (p2))))))
(channel-get ch)  ; ⇒ '(0 . 2)

(define k-ch (make-channel))
(define (send-k)
  (parameterize ([p1 0])
    (thread (lambda ()
              (let/ec esc
                (channel-put ch
                             ((let/cc k
                                (channel-put k-ch k)
                                (esc)))))))))
```

```
(send-k)
(thread (lambda () ((channel-get k-ch) (let ([v (p1)]) (lambda () v)))))
(channel-get ch) ; ⇒ 1
(send-k)
(thread (lambda () ((channel-get k-ch) p1)))
(channel-get ch) ; ⇒ 0
```

MzScheme parameters correspond to *preserved thread fluids* in Scsh. See also “Processes vs. User-Level Threads in Scsh” by Gasbichler and Sperber (proceedings of the 2002 Scheme Workshop).

7.9.1 Built-in Parameters

MzScheme’s built-in parameter procedures are listed in the following sections. The `make-parameter` procedure, described in §7.9.2, creates a new parameter and returns a corresponding parameter procedure.

7.9.1.1 Current Directory

- `(current-directory [`*`path`*`])` gets or sets a path that determines the current directory. When the parameter procedure is called to set the current directory, the path argument is expanded, simplified using `simplify-path` (see §11.3.1), and then converted to a directory path with `path->directory-path`; expansion and simplification raise an exception if the path is ill-formed. The path is not checked for existence when the parameter is set.

7.9.1.2 Ports

- `(current-input-port [`*`input-port`*`])` gets or sets an input port used by `read`, *`read-byte`*, `read-char`, etc. when a specific input port is not provided.
- `(current-output-port [`*`output-port`*`])` gets or sets an output port used by `display`, `write`, `print`, `write-char`, etc. when a specific output port is not provided.
- `(current-error-port [`*`output-port`*`])` gets or sets an output port used by the default error display handler.
- `(global-port-print-handler [`*`proc`*`])` gets or sets a procedure that takes an arbitrary value and an output port. This *global port print handler* is called by the default port print handler (see §11.2.7) to `print` values into a port.
- `(port-count-lines-enabled [`*`on?`*`])` gets or sets a boolean value that determines whether line counting is enabled automatically for newly created ports; see also §11.2.1.1. The default value is `#f`.

7.9.1.3 Parsing

- `(read-case-sensitive [`*`on?`*`])` gets or sets a boolean value that controls parsing input symbols. When this parameter’s value is `#f`, the reader case-folds symbols (e.g., `hi` when the input is any one of `hi`, `Hi`, `HI`, or `hI`). The parameter also affects the way that `write` prints symbols containing uppercase characters; if the parameter’s value is `#f`, then symbols are printed with uppercase characters quoted by a backslash (\) or vertical bar (|). The parameter’s value is overridden by backslash and vertical-bar quotes and the `#cs` and `#ci` prefixes; see §11.2.4 for more information. While a module is loaded, the parameter is set to `#t` (see §5.8).
- `(read-square-bracket-as-paren [`*`on?`*`])` gets or sets a boolean value that controls whether square brackets (“[” and “]”) are treated as parentheses. See §11.2.4 for more information.

- `(read-curly-brace-as-paren [`*on?*`])` gets or sets a boolean value that controls whether curly braces ("{" and "}") are treated as parentheses. See §11.2.4 for more information.
- `(read-accept-box [`*on?*`])` gets or sets a boolean value that controls parsing `#\&` input. See §11.2.4 for more information.
- `(read-accept-compiled [`*on?*`])` gets or sets a boolean value that controls parsing pre-compiled input. See §11.2.4 for more information.
- `(read-accept-bar-quote [`*on?*`])` gets or sets a boolean value that controls parsing and printing a vertical bar (|) in symbols. See §11.2.4 and §11.2.5 for more information.
- `(read-accept-graph [`*on?*`])` gets or sets a boolean value that controls parsing input with sharing. See §11.2.5.1 for more information.
- `(read-decimal-as-inexact [`*on?*`])` gets or sets a boolean value that controls parsing input numbers with a decimal point or exponent (but no explicit exactness tag). See §11.2.5.1 for more information.
- `(read-accept-dot [`*on?*`])` gets or sets a boolean value that controls parsing input with a dot, which is normally used for literal cons cells. See §11.2.4 for more information.
- `(read-accept-quasiquote [`*on?*`])` gets or sets a boolean value that controls parsing input with a backquote or comma, which is normally used for `quasiquote`, `unquote`, and `unquote-splicing` abbreviations. See §11.2.4 for more information.
- `(read-accept-reader [`*on?*`])` gets or sets a boolean value that controls whether `#reader` is allowed for selecting a parser. See §11.2.4 for more information.
- `(current-reader-guard [`*proc*`])` gets or sets a procedure of one argument that converts or rejects (by raising an exception) a module-path datum following `#reader`. See §11.2.4 for more information.
- `(current-readtable [`*readtable-or-false*`])` gets or sets a readtable that adjust the parsing of S-expression input, or `#f` for the default behavior. See §11.2.8 for more information.

7.9.1.4 Printing

- `(print-unreadable [`*on?*`])` gets or sets a boolean value that controls printing values that have no `read`able form (using the default reader), including structures that have a custom-write procedure (see §11.2.10); defaults to `#t`. See §11.2.5 for more information.
- `(print-graph [`*on?*`])` gets or sets a boolean value that controls printing data with sharing; defaults to `#f`. See §11.2.5.1 for more information.
- `(print-struct [`*on?*`])` gets or sets a boolean value that controls printing structure values in vector form; defaults to `#t`. See §11.2.5 for more information. This parameter has no effect on the printing of structures that have a custom-write procedure (see §11.2.10).
- `(print-box [`*on?*`])` gets or sets a boolean value that controls printing box values; defaults to `#t`. See §11.2.5 for more information.
- `(print-vector-length [`*on?*`])` gets or sets a boolean value that controls printing vectors; defaults to `#t`. See §11.2.5 for more information.
- `(print-hash-table [`*on?*`])` gets or sets a boolean value that controls printing hash tables; defaults to `#f`. See §11.2.5 for more information.
- `(print-honu [`*on?*`])` gets or sets a boolean value that controls printing values in an alternate syntax. See §19 for more information.

7.9.1.5 Read-Eval-Print

- `(current-prompt-read [`*`proc`*`])` gets or sets a procedure that takes no arguments, displays a prompt string, and returns an expression to evaluate. This *prompt read handler* is called by the read phase of `read-eval-print-loop` (see §14.1). The default prompt read handler prints "> " and returns the result of

```
(parameterize ((read-accept-reader #t))
  (read-syntax name-string))
```

  where *`name-string`* corresponds to the current input source.

- `(current-eval [`*`proc`*`])` gets or sets a procedure that takes an expression—in the form of syntax object, S-expression, compiled expression, or compiled expression wrapped in a syntax object—and returns the expression's value (or values; see §2.2). This *evaluation handler* is called by `eval`, `eval-syntax`, the default load handler, and `read-eval-print-loop` to evaluate an expression (see §14.1). The handler should evaluate its argument in tail position, like `eval`. The default evaluation handler compiles and executes the expression in the current namespace (determined by the `current-namespace` parameter); if the argument is a syntax object, it is treated like an argument to `eval-syntax` and not given additional context. The default evaluation handler also partly expands expressions to splice the body of top-level `begin` forms into the top level (the compiler is called only on the individual spliced forms, and not the top-level `begin` form), and each spliced top-level form is evaluated before the next one is compiled.

- `(current-compile [`*`proc`*`])` gets or sets a procedure that takes two arguments—a syntax object and a boolean—and returns the compiled form of its first argument. This *compilation handler* is called by `compile` (see §14.3), and indirectly by `eval`, `eval-syntax`, the default load handler, and `read-eval-print-loop` (see §14.1). The compilation handler's first argument has the lexical content needed for expansion and compilation. The compilation handler's second argument is `#t` if the compiled expression will be used only for immediate evaluation, or `#f` if the compiled form may be saved for later use; the default compilation handler is optimized for the special case of immediate evaluation. The result of a compilation handler must be a compiled expression (see §14.3).

- `(current-namespace [`*`namespace`*`])` gets or sets a namespace value (see §8) that determines the namespace used to resolve module and identifier references. The *current namespace* is used by the default evaluation handler, the `compile` procedure, and other built-in procedures that operate on "global" bindings.

- `(current-print [`*`proc`*`])` gets or sets a procedure that takes a value to print. This *print handler* is called by `read-eval-print-loop` (see §14.1) to print the result of an evaluation (and the result is ignored). The default print handler `print`s the value to the current output port (determined by the `current-output-port` parameter) and then outputs a newline, except that it does nothing when the value is void.

- `(compile-allow-set!-undefined [`*`on?`*`])` gets or sets a boolean value indicating how to compile a `set!` expression that mutates a global variable. If the value of this parameter is a true value, `set!` expressions for global variables are compiled so that the global variable is set even if it was not previously defined. Otherwise, `set!` expressions for global variables are compiled to raise the *`exn:fail:contract:variable`* exception if the global variable is not defined at the time the `set!` is performed. Note that this parameter is used when an expression is *compiled*, not when it is *evaluated*.

- `(compile-enforce-module-constants [`*`on?`*`])` gets or sets a boolean value indicating how a `module` form should be compiled. When constants are enforced, and when the macro-expanded body of a module contains no `set!` assignment to a particular variable defined within the module, then the variable is marked as constant when the definition is evaluated. Afterward, the variable's value cannot be assigned or undefined through `module->namespace`, and it cannot be defined by redeclaring the module. Enforcing constants allows the compiler to inline some variable values, and it allows the native-code just-in-time compiler to generate code that skips certain run-time checks.

- `(eval-jit-enabled [`*`on?`*`])` gets or sets a boolean value that determines whether the native-code just-in-time compiler (JIT) is enabled for code (compiled or not) that is passed to the default evaluation handler.

The default is `#t`, unless the JIT is disabled through the `--no-jit` or `-j` command-line flag to stand-alone MzScheme (or MrEd), or through the **PLTNOMZJIT** environment variable (set to any value).

7.9.1.6 Loading

- `(current-load [`*proc*`])` gets or sets a procedure that loads a file and returns the value (or values; see §2.2) of the last expression read from the file. This *load handler* is called by `load`, `load-relative`, `load/use-compiled`, and `load/cd`.

  A load handler procedure takes two arguments: a path (see §11.3.1) and an expected module name. The expected module name is either a symbol or `#f`; see §5.8 for further information.

  The default load handler reads expressions from the file (with compiled expressions enabled, `#reader` enabled, and line-counting enabled) and passes each expression to the current evaluation handler. The default load handler also treats a hash mark on the first line of the file as a comment (see §11.2.4). The current load directory for loading the file is set before the load handler is called (see §14.1).

- `(current-load-extension [`*proc*`])` gets or sets a procedure that loads a dynamic extension (see §14.4) and returns the extension's value(s). This *load extension handler* is called by `load-extension`, `load-relative`, and `load/use-compiled`.

  A load extension handler procedure takes two arguments: a path (see §11.3.1) and an expected module name. The expected module name is either a symbol or `#f`; see §5.8 for further information.

  The default load extension handler loads an extension using operating system primitives.

- `(current-load/use-compiled [`*proc*`])` gets or sets a procedure that loads a file or a compiled version of the file; see §14.1 for more information. A *load/use-compiled handler* procedure takes the same arguments as a load handler. The handler is expected to call the load handler or the load-extension handler. Unlike a load handler or load-extension handler, a load/use-compiled handler is expected to set the current `load-relative` directory.

- `(current-load-relative-directory [`*path*`])` gets or sets a complete directory path (see §11.3.1) or `#f`. This current `load-relative` directory is set by `load`, `load-relative`, `load/use-compiled`, `load/cd`, `load-extension`, and `load-relative-extension` to the directory of the file being loaded. This parameter is used by `load-relative`, `load/use-compiled` and `load-relative-extension` (see §14.1). When a new path or string is provided to the parameter procedure `current-load-relative-directory`, it is immediately expanded (see §11.3.1) and converted to a path. (The directory need not exist.)

- `(current-write-relative-directory [`*path*`])` gets or sets a complete directory path (see §11.3.1) or `#f`. This path is used when writing compiled code that contains source-location pathnames for procedures; paths within this directory (syntactically) are converted to relative paths. When compiled code is read, relative paths are converted back to complete paths using the current load-relative directory (if it is not `#f`).

- `(use-compiled-file-paths [`*path-list*`])` gets or sets a list of paths, which defaults to `(list (string->path "compiled"))`. It is used by `load/used-compiled` (and thus `require`) as a search path for compiled versions of files. See §14.1 for more information. When a new list of paths and strings is provided to the parameter procedure, it is converted to an immutable list of paths.

- `(current-library-collection-paths [`*path-list*`])` gets or sets a list of complete directory paths (see §11.3.1) for library collections used by `require`. See Chapter 16 for more information. When a new list of paths and strings is provided to the parameter procedure, it is converted to an immutable list of paths.

- `(use-user-specific-search-paths [`*on?*`])` gets or sets a boolean value that determines whether user-specific paths, which are in the directory produced by `(find-system-path 'addon-dir)`, are included in search paths for collections, C libraries, etc. For example, `find-library-collection-paths` (see §16) omits the user-specific collection directory when this parameter's value is `#f`.

- `(current-command-line-arguments` [*string-vector*]`)` gets or sets a vector of strings representing command-line arguments. When a new vector of strings is provided to the parameter procedure, it is converted to an immutable vector of immutable strings. The stand-alone version of MzScheme (and MrEd) initializes the parameter to contain command-line arguments that are not processed directly by MzScheme and MrEd. (The same vector is also installed as the value of the `argv` global.) If command-line arguments are provided to MzScheme/MrEd as a byte strings, they are converted to strings using the current locale's encoding (see §1.2.2).

7.9.1.7 Exceptions

- `(uncaught-exception-handler` [*proc*]`)` gets or sets a procedure that is invoked to handle an exception when no handler is associated with the current continuation. See §6.1 for more information about exceptions.

- `(error-escape-handler` [*proc*]`)` gets or sets a procedure that takes no arguments and escapes from the dynamic context of an exception. See §6.8 for further information about the error escape handler. The default error escape handler escapes using `(`*`abort-current-continuation`* `(default-continuation-prompt-tag) void)`; see §6.5 for more information about `abort-current-continuat`

- `(error-display-handler` [*proc*]`)` gets or sets a procedure that takes two arguments: a string to print as an error message, and a value representing a raised exception. This *error display handler* is called by the default exception handler with an error message and the exception value (see §6.1). The default error display handler `display`s its first argument to the current error port (determined by the `current-error-port` parameter) and extracts a stack trace (see §6.6) to display from the second argument if it is an `exn` value but not an `exn:fail:user` value.[3] To report a run-time error, use `raise` (see §6.1) or procedures like `error` (see §6.2) instead of calling the error display procedure directly.

- `(error-print-width` [*k*]`)` gets or sets an exact integer greater than `3`. This value is used as the maximum number of characters used to print a Scheme value that is embedded in a primitive error message.

- `(error-print-context-length` [*k*]`)` gets or sets a non-negative, exact integer. This value is used by the default error display handler as the maximum number of lines of context (or "stack trace") to print; a single "..." line is printed if more lines are available after the first *k* lines. A `0` value for *k* disables context printing entirely.

- `(error-value->string-handler` [*proc*]`)` gets or sets a procedure that takes an arbitrary Scheme value and an integer and returns a string. This *error value conversion handler* is used to print a Scheme value that is embedded in a primitive error message. The integer argument to the handler specifies the maximum number of characters that should be used to represent the value in the resulting string. The default error value conversion handler `print`s the value into a string;[4] if the printed form is too long, the printed form is truncated and the last three characters of the return string are set to "...".

  If the string returned by an error value conversion handler is longer than requested, the string is destructively "truncated" by setting the first extra position in the string to the null character. If a non-string is returned, then the string `"..."` is used. If a primitive error string needs to be generated before the handler has returned, the default error value conversion handler is used.

  Call to an error value conversion handler are *`parameterized`* to re-install the default error value conversion handler, and to enable printing of unreadable values (see §7.9.1.4).

- `(error-print-source-location` [*include?*]`)` gets or sets a boolean that controls whether read and syntax error messages include source information, such as the source line and column or the expression. This parameter also controls the error message when a module-defined variable is accessed before its definition is executed; the parameter determines whether the message includes a module name. Only the message field of an `exn:fail:read`, `exn:fail:syntax`, or `exn:fail:contract:variable` structure is affected by the parameter. The default is `#t`.

[3]The default error display handler in DrScheme also uses the second argument to highlight source locations.
[4]Using the current global port print handler; see §7.9.1.2.

7.9.1.8 Security

- `(current-security-guard [`*`security-guard`*`])` gets or sets a security guard (see §9.1) that controls access to the filesystem and network.
- `(current-custodian [`*`custodian`*`])` gets or sets a custodian (see §9.2) that assumes responsibility for newly created threads, ports, TCP listeners, UDP sockets, and byte converters.
- `(current-thread-group [`*`thread-group`*`])` gets or sets a thread group (see §9.3) that determines CPU allocation for newly created threads.
- `(current-inspector [`*`inspector`*`])` gets or sets an inspector (see §4.5) that controls debugging access to newly created structure types.
- `(current-code-inspector [`*`inspector`*`])` gets or sets an inspector (see §9.4) that controls debugging access to module bindings and redefinitions.

7.9.1.9 Exiting

- `(exit-handler [`*`proc`*`])` gets or sets a procedure that takes a single argument. This *exit handler* is called by `exit`. The default exit handler takes any argument and shuts down MzScheme; see §14.2 for information about exit codes.

7.9.1.10 Random Numbers

- `(current-pseudo-random-generator [`*`generator`*`])` gets or sets a pseudo-random number generator (see §3.3) used by `random` and `random-seed`.
- `(current-evt-pseudo-random-generator [`*`generator`*`])` gets or sets a pseudo-random number generator (see §3.3) used by `sync` and `sync/enable-break` (see §7.7).

7.9.1.11 Locale

- `(current-locale [`*`string-or-#f`*`])` gets or sets a string/boolean value that controls the interpretation of characters for functions such as `string-locale<?`, and *`string-locale-upcase`* (see §1.2.2 and §3.5). When locale sensitivity is disabled by setting the parameter to `#f`, strings are compared in a fully portable manner, which is the same as the standard procedures; otherwise, they are interpreted according to a locale setting (in the sense of the C library's `setlocale`). The `""` locale is always a synonym for the current machine's default locale; other locale names are platform-specific.[5] String or character printing with `write` is not affected by the parameter, and neither are symbol case or regular expressions (see §10). The parameter's default value is `""`.

7.9.1.12 Modules

- `(current-module-name-resolver [`*`proc`*`])` gets or sets a procedure used to resolve module paths. See §5.4 for more information.
- `(current-module-name-prefix [`*`symbol-or-false`*`])` gets or sets a symbol to be prefixed onto a module declaration when it is evaluated, where `#f` means no prefix. This parameter is intended for use by a module name resolver; see §5.4 for more information.

[5] The `"C"` locale is also always available; setting the locale to `"C"` is the same as disabling locale sensitivity with `#f` only when string operations are restricted to the first 128 characters.

7.9.1.13 Performance Tuning

- `(current-thread-initial-stack-size [`*`k`*`])` gets or sets a positive exact integer; the integer provides a hint about how much space to reserve for a thread's local variables. The actual space used by a computation is affected by just-in-time (JIT) compilation, but it is otherwise platform-independent.

7.9.2 Parameter Utilities

`(make-parameter `*`v`*` [`*`guard-proc`*`])` returns a new parameter procedure. The value of the parameter is initialized to *`v`* in all threads. If *`guard-proc`* is supplied, it is used as the parameter's guard procedure. A guard procedure takes one argument. Whenever the parameter procedure is applied to an argument, the argument is passed on to the guard procedure. The result returned by the guard procedure is used as the new parameter value. A guard procedure can raise an exception to reject a change to the parameter's value. The *`guard-proc`* is not applied to the initial *`v`*.

`(parameter? `*`v`*`)` returns `#t` if *`v`* is a parameter procedure, `#f` otherwise.

`(parameter-procedure=? `*`a b`*`)` returns `#t` if the parameter procedures *`a`* and *`b`* always modify the same parameter, `#f` otherwise.

`(current-parameterization)` returns the current continuation's parameterization.

`(call-with-parameterization `*`parameterization thunk`*`)` calls *`thunk`* (via a tail call) with *`parameterization`* as the current parameterization.

`(parameterization? `*`v`*`)` returns `#t` if *`v`* is a parameterization returned by `current-parameterization`, `#f` otherwise.

8. Namespaces

MzScheme supports multiple *namespaces* for top-level variable bindings, syntax bindings, module imports, and module declarations.

A new namespace is created with the `make-namespace` procedure, which returns a first-class namespace value. A namespace is used by setting the `current-namespace` parameter value (see §7.9.1.5), by providing the namespace to procedures such as `eval` and `eval-syntax`. The MzScheme versions of the R^5RS procedures `scheme-report-environment` and `null-environment` produce namespaces.[1]

The current namespace is used by many procedures, including `eval`, `load`, `compile`, and `expand`.[2] After an expression is `eval`ed, the global variable references in the expression are permanently attached to a particular namespace, so the current namespace at the time that the code is executed is *not* used as the namespace for referencing global variables in the expression.

Example:

```
(define x 'orig) ; define in the original namespace
;; The following let expression is compiled in the original
;; namespace, so direct references to x see 'orig.
(let ([n (make-namespace)]) ; make new namespace
  (parameterize ([current-namespace n])
    (eval '(define x 'new)) ; evals in the new namespace
    (display x) ; displays 'orig
    (display (eval 'x)))) ; displays 'new
```

A namespace actually encapsulates three top-level environments: one for normal expressions, one for macro transformer expressions, and one for macro templates; see §12 for more information about the transformer environment, and see §12.3.4 for more information about the template environment. Module declarations are shared by the environments, but module instances, variable bindings, syntax bindings, and module imports are distinct. More precisely, the transformer environment never contains any syntax bindings, and its variables, module instances, and module imports are distinct from the variables, module instances, and module imports of the normal top-level environment. The template environment contains module imports, only.

Each namespace has a *module registry* that maps module names to module declarations (see Chapter 5). The `module->namespace` procedure returns a namespace with the same module registry as the current namespace, but whose environment and bindings correspond to the body of an instantiated module. (This facility is primarily useful for debugging, and its use is limited by the current code inspector; see also §9.4.)

[1]The resulting namespace contains syntax imports for `#%app`, `#%datum`, and `#%top`, because syntax expansion requires them (see §12.5), but those names are not legal R^5RS identifiers.

[2]More precisely, the current namespace is used by the evaluation and load handlers, rather than directly by `eval` and `load`.

8.1 Identifier Resolution in Namespaces

Identifier resolution in a namespace's top-level environment, for compilation or expansion, proceeds in two steps. First, the environment determines whether the identifier is mapped to a top-level variable, to syntax, or to a module import (which can be either syntax or a variable). Second, if the identifier is mapped to a top-level variable, then the variable's location is found; if the identifier is mapped to syntax, then the expansion-time binding is found; and if the identifier is mapped to an import, then the source module is consulted.

Importing a variable from a module with `require` is *not* the same as defining the variable; the import does not create a new top-level variable in the environment, but instead maps an identifier to the module's variable, in the same way that a syntax definition maps an identifier to a transformer.

Redefining a previously-defined variable is the same as mutating the variable with `set!`. Rebinding a syntax-bound or import-bound identifier (to syntax or an import) replaces the old binding with the new one for future uses of the environment.

If an identifier is bound to syntax or to an import, then defining the identifier as a variable shadows the syntax or import in future uses of the environment. Similarly, if an identifier is bound to a top-level variable, then binding the identifier to syntax or an import shadows the variable; the variable's value remains unchanged, however, and may be accessible through previously evaluated expressions.

Example:

```
(define x 5)
(define (f) x)
x ; ⇒ 5
(f) ; ⇒ 5
(define-syntax x (syntax-rules ()))
x ; ⇒ bad syntax
(f) ; ⇒ 5
(define x 7)
x ; ⇒ 7
(f) ; ⇒ 7
(module m mzscheme (define x 8) (provide x))
(require m)
x ; ⇒ 8
(f) ; ⇒ 7
```

8.2 Initial Namespace

In the stand-alone MzScheme application, the initial namespace's module registry contains declarations for `mzscheme` and the primitive `#%`-named modules (see §5.7). The normal top-level environment of the initial namespace contains imports for all MzScheme syntax, and it contains variable bindings (as opposed to imports) for every built-in procedure and constant. The transformer top-level environment of the initial namespace imports all MzScheme syntax, procedures, and constants.

Applications embedding MzScheme may extend or modify the set of initial bindings, but they will usually only add primitive modules with `#%`-prefixed names. (MrEd adds `#%mred-kernel` for its graphical toolbox.)

8.3 Namespace Utilities

`(make-namespace [`*`flag-symbol`*`])` creates a new namespace with a new module registry; the *`flag-symbol`* is an option that determines the initial bindings in the namespace. The allowed values for *`flag-symbol`* are:

- `'initial` (the default) — the new namespace contains the module declarations of the initial namespace (see §8.2), and the new namespace's normal top-level environment contains bindings and imports as in the initial namespace. However, the namespace's transformer top-level environment is empty.
- `'empty` — creates a namespace with no initial bindings or module declarations.

`(namespace? v)` returns `#t` if *v* is a namespace value, `#f` otherwise.

`(namespace-symbol->identifier symbol)` is similar to `datum->syntax-object` (see §12.2.2) restricted to symbols. The lexical context of the resulting identifier corresponds to the top-level environment of the current namespace; the identifier has no source location or properties.

`(namespace-variable-value symbol [use-mapping? failure-thunk namespace])` returns a value for *symbol* in *namespace*, where *namespace* defaults to the current namespace. The returned value depends on *use-mapping?*:

- If *use-mapping?* is true (the default), and if *symbol* maps to a top-level variable or an imported variable (see §8.1), then the result is the same as evaluating *symbol* as an expression. If *symbol* maps to syntax or imported syntax, the `exn:fail:syntax` exception is raised (or *failure-thunk* is called; see below). If *symbol* is mapped to an undefined variable or an uninitialized module variable, the `exn:fail:contract:variable` exception is raised (or *failure-thunk* is called).
- If *use-mapping?* is false, the namespace's syntax and import mappings are ignored. Instead, the value of the top-level variable named *symbol* in namespace is returned. If the variable is undefined, the `exn:fail:contract:variable` exception is raised (or *failure-thunk* is called).

If *failure-thunk* is provided, `namespace-variable-value` calls *failure-thunk* to produce the return value in place of raising an `exn:fail:contract:variable` or `exn:fail:syntax` exception.

`(namespace-set-variable-value! symbol v [map? namespace])` sets the value of *symbol* in the top-level environment of *namespace* (where *namespace* defaults to the current namespace) defining *symbol* if it is not already defined. If *map?* is supplied as true, then the namespace's identifier mapping is also adjusted (see §8.1) so that *symbol* maps to the variable. The default value for *map?* is `#f`.

`(namespace-undefine-variable! symbol namespace)` removes the *symbol* variable, if any, in the top-level environment of the *namespace*, where *namespace* defaults to the current namespace. The namespace's identifier mapping is unaffected.

`(namespace-mapped-symbols [namespace])` returns a list of all symbols that are mapped to variables, syntax, and imports in *namespace*, where *namespace* defaults to the current namespace.

`(namespace-require quoted-require-spec)` performs the import corresponding to *quoted-require-spec* in the top-level environment of the current namespace (like a top-level `require` expression). See also Chapter 5. Module paths in *quoted-require-spec* are not resolved with respect to any other module, even if the current namespace corresponds to a module body.

`(namespace-transformer-require quoted-require-spec)` performs the import corresponding to *quoted-require-spec* in the top-level transformer environment (like a top-level `require-for-syntax` expression). See also Chapter 5. Module paths in *quoted-require-spec* are not resolved with respect to any other module, even if the current namespace corresponds to a module body.

`(namespace-require/copy quoted-require-spec)` is like `namespace-require` for syntax exported from the module, but exported variables are treated differently: the export's current value is copied to a top-level variable in the current namespace.

(namespace-require/expansion-time *quoted-require-spec*) is like namespace-require, but only the transformer part of the module is executed. If the required module has not been invoked before, the module's variables remain undefined.

(namespace-attach-module *src-namespace module-path-v* [*dest-namespace*]) attaches the instantiated module named by *module-path-v* in *src-namespace* to the registry of *dest-namespace* (which is the current namespace if *dest-namespace* is not supplied). If *module-path-v* is not a symbol, the current module name resolver is called to resolve the path, but no module is loaded; the resolved form of *module-path-v* is used as the module name in *dest-namespace*. In addition to *module-path-v*, every module that it imports (directly or indirectly) is also recorded in the current namespace's registry. If *module-path-v* does not refer to an instantiated module in *src-namespace*, or if the name of any module to be attached already has a different declaration or instance in *dest-namespace*, then the exn:fail:contract exception is raised. The inspector of the module invocation in *dest-namespace* is the same as inspector of the invocation in *src-namespace*.

(namespace-unprotect-module *inspector module-path-v*) namespace changes the inspector for the instance of the module referenced by *module-path-v* in *namespace*'s registry so that it is controlled by the current code inspector. If *namespace* is not supplied, it is the current namespace. The given *inspector* must currently control the invocation of the module in *namespace*'s registry, otherwise the exn:fail:contract exception is raised. See also §9.4.

(namespace-module-registry *namespace*) returns the registry of the given namespace. This value is useful only for identification via eq?.

(module->namespace *module-path-v*) returns a namespace that corresponds to the body of an instantiated module in the current namespace's registry. The returned namespace has the same module registry as the current namespace. Modifying a binding in the namespace changes the binding seen in modules that require the namespace's module. Module paths in a top-level require expression are resolved with respect to the namespace's module. New provide declarations are not allowed. If the current code inspector does not control the invocation of the module in the current namespace's registry, the exn:fail:contract exception is raised; see also §9.4. Bindings in the namespace cannot be modified if the compile-enforce-module-constants parameter was true when the module was declared, unless the module declaration itself included assignments to the binding via set!.

(namespace-syntax-introduce *stx*) returns a syntax object like *stx*, except that the current namespace's bindings are included in the syntax object's context (see §12.3). The additional context is overridden by any existing top-level context in the syntax object, or by any existing or future module context. See §12.2 for more information about syntax objects.

(module-provide-protected? *module-path-index symbol*) returns #f if the module declaration for *module-path-index* defines *symbol* and exports it unprotected, #t otherwise (which may mean that the symbol corresponds to an unexported definition, a protected export, or an identifier that is not defined at all within the module). The *module-path-index* argument can be a symbol; see §5.4.2 for more information on module path indices. Typically, the arguments to module-provide-protected? correspond to the first two elements of a list produced by identifier-binding (see §12.3).

9. Security

MzScheme offers several mechanisms for managing security:

- Custodians (§9.2) manage resource allocation.
- Security guards (§9.1) control access to the filesystem and network.
- Inspectors (§4.5) control access to the content of otherwise opaque structures and modules (see §9.4).
- Namespaces (§8) control access to Scheme bindings.
- Thread groups (§9.3) control CPU allocation.

All security mechanisms rely on thread-specific parameters (see §7.9).

9.1 Security Guards

A *security guard* provides a set of access-checking procedures to be called when a thread initiates access of a file, directory, or network connection through a primitive procedure. For example, when a thread calls `open-input-file`, the thread's current security guard is consulted to check whether the thread is allowed read access to the file. If access is granted, the thread receives a port that it may use indefinitely, regardless of changes to the security guard (although the port's custodian could shut down the port; see §9.2).

A thread's current security guard is determined by the `current-security-guard` parameter (see §7.9.1.8). Every security guard has a parent, and a parent's access procedures are called whenever a child's access procedures are called. Thus, a thread cannot increase its own access arbitrarily by installing a new guard. The initial security guard enforces no access restrictions other than those enforced by the host platform.

`(make-security-guard` *`parent-security-guard file-proc network-proc`* [*`link-proc`*]`)` creates a new security guard whose parent is *`parent-security-guard`*.

The *`file-proc`* procedure must accept three arguments:

- a symbol for the primitive procedure that triggered the access check, which is useful for raising an exception to deny access.
- a path (see §11.3.1) or `#f` for pathless queries, such as `(current-directory)`, `(filesystem-root-list)`, and `(find-system-path` *`symbol`*`)`. A path provided to *`file-proc`* is not expanded or otherwise normalized before checking access; it may be a relative path, for example.
- an immutable list containing one or more of the following symbols:
  - `'read` — read a file or directory
  - `'write` — modify or create a file or directory
  - `'execute` — execute a file

  - `'delete` — delete a file or directory
  - `'exists` — determine whether a file or directory exists, or that a path string is well-formed

  The `'exists` symbol is never combined with other symbols in the last argument to *`file-proc`*, but any other combination is possible. When the second argument to *`file-proc`* is `#f`, the last argument always contains only `'exists`.

The *`network-proc`* procedure must accept four arguments:

- a symbol for the primitive operation that triggered the access check, which is useful for raising an exception to deny access.
- an immutable string representing the target hostname for a client connection or the accepting hostname for a listening server; `#f` for a listening server or UDP socket that accepts connections at all of the host's address; or `#f` an unbound UDP socket.
- an exact integer between `1` and `65535` (inclusive) representing the port number, or `#f` for an unbound UDP socket. In the case of a client connection, the port number is the target port on the server. For a listening server, the port number is the local port number.
- a symbol, either `'client` or `'server`, indicating whether the check is for the creation of a client connection or a listening server. The opening of an unbound UDP socket is identified as a `'client` connection; explicitly binding the socket is identified as a `'server` action.

The *`link-proc`* argument can be `#f` (the default) or a procedure of three arguments:

- a symbol for the primitive procedure that triggered the access check, which is useful for raising an exception to deny access.
- a complete path (see §11.3.1) representing the file to create as link.
- a path representing the content of the link, which may be relative the second-argument path; this path is not expanded or otherwise normalized before checking access.

If *`link-proc`* is `#f` or unprovided, then a default procedure is used that always raises `exn:fail`.

The return value of *`file-proc`*, *`network-proc`*, or *`link-proc`* is ignored. To deny access, the procedure must raise an exception or otherwise escape from the context of the primitive call. If the procedure returns, the parent's corresponding procedure is called on the same inputs, and so on up the chain of security guards.

The *`file-proc`*, *`network-proc`*, and *`link-proc`* procedures are invoked in the thread that called the access-checked primitive. Breaks may or may not be enabled (see §6.7). Full continuation jumps are blocked going into or out of the *`file-proc`* or *`network-proc`* call (see §6.3).

`(security-guard? v)` returns `#t` if *`v`* is a security guard value, `#f` otherwise.

9.2 Custodians

A *custodian* manages a collection of threads, file-stream ports, TCP ports, TCP listeners, UDP sockets, and byte converters.[1] Whenever a thread, file-stream port, TCP port, TCP listener, or UDP socket is created, it is placed under the management of the current custodian (as determined by the `current-custodian` parameter; see §7.9.1.8).

[1]In MrEd, custodians also manage eventspaces.

The main operation on a custodian is to shut down its managed values via `custodian-shutdown-all`. In other words, `custodian-shutdown-all` generalizes `kill-thread` to forcibly and immediately close a set of ports, TCP connections, etc., as well as terminate (or suspend) a set of threads. For example, a web server might use a custodian to manage all of the resources of a particular session so that the session can be cleanly terminated if it exceeds its allowed lifetime.

A custodian that has been shut down cannot manage new objects. If the current custodian is shut down before a procedure is called to create a managed resource (e.g., `open-input-port`, `thread`), the `exn:fail:contract` exception is raised.

A thread can have multiple managing custodians, and a suspended thread created with `thread/suspend-to-kill` can have zero custodians. Extra custodians become associated with a thread through `thread-resume` (see §7.1). When a thread has multiple custodians, it is not necessarily killed by a `custodian-shutdown-all`, but shut-down custodians are removed from the thread's managing set, and the thread is killed when its managing set becomes empty.

The values managed by a custodian are only weakly held by the custodian. As a result, a will (see §13.3) can be executed for a value that is managed by a custodian. In addition, a custodian only weakly references its subordinate custodians; if a subordinate custodian is unreferenced but has its own subordinates, then the custodian may be collected, at which point its subordinates become immediately subordinate to the collected custodian's superordinate custodian.

In addition to the other entities managed by a custodian, a *custodian box* created with `make-custodian-box` strongly holds onto a value placed in the box until the box's custodian is shut down. The custodian only weakly retains the box itself, however (so the box and its content can be collected if there are no other references to them).

When MzScheme is compiled with support for per-custodian memory accounting (see `custodian-memory-accounting-availa` below), the `current-memory-use` procedure (see §13.4) can report a custodian-specific result. This result determines how much memory is occupied by objects that are reachable from the custodian's managed values, especially its threads, and including its sub-custodians' managed values. If an object is reachable from two custodians where neither is an ancestor of the other, an object is arbitrarily charged to one of the other, and the choice can change after each collection; objects reachable from both a custodian and its descendant, however, are reliably charged to the descendant. Reachability for per-custodian accounting does not include weak references, references to threads managed by non-descendant custodians, references to non-descendant custodians, or references to custodian boxes for non-descendant custodians.

`(make-custodian [`*`custodian`*`])` creates a new custodian that is subordinate to *`custodian`*. When *`custodian`* is directed (via `custodian-shutdown-all`) to shut down all of its managed values, the new subordinate custodian is automatically directed to shut down its managed values as well. The default value for *`custodian`* is the current custodian.

`(custodian-shutdown-all `*`custodian`*`)` closes all open ports and closes all active TCP listeners and UDP sockets that are managed by *`custodian`*. It also removes *`custodian`* (and its subordinates) as managers of all threads; when a thread has no managers, it is killed.[2] If the current thread is to be killed, all other shut-down actions take place before killing the thread.

`(custodian? `*`v`*`)` returns `#t` if *`v`* is a custodian value, `#f` otherwise.

`(custodian-managed-list `*`custodian super-custodian`*`)` returns a list of immediately managed objects and subordinate custodians for *`custodian`*, where *`custodian`* is itself subordinate to *`super-custodian`* (directly or indirectly). If *`custodian`* is not strictly subordinate to *`super-custodian`*, the `exn:fail:contract` exception is raised.

`(custodian-memory-accounting-available?)` returns `#t` if MzScheme is compiled with support for

[2]"Killing" a thread created with `thread/suspend-to-kill` merely suspends the thread.

per-custodian memory accounting,[3] `#f` otherwise.

`(custodian-require-memory` *`limit-custodian need-k stop-custodian`*`)` registers a require check if MzScheme is compiled with support for per-custodian memory accounting, otherwise the `exn:fail:unsupported` exception is raised. If a check is registered, and if MzScheme later reaches a state after garbage collection (see §13.4) where allocating *`need-k`* bytes charged to *`limit-custodian`* would fail or tigger some shutdown, then *`stop-custodian`* is shut down.

`(custodian-limit-memory` *`limit-custodian limit-k`* [*`stop-custodian`*]`)` registers a limit check if MzScheme is compiled with support for per-custodian memory accounting, otherwise the `exn:fail:unsupported` exception is raised. If a check is registered, and if MzScheme later reaches a state after garbage collection (see §13.4) where *`limit-custodian`* owns more than *`limit-k`* bytes, then *`stop-custodian`* is shut down. The *`stop-custodian`* argument defaults to *`limit-custodian`*.

For reliable shutdown, *`limit-k`* for `custodian-limit-memory` must be much lower than the total amount of memory available (minus the size of memory that is potentially used and not charged to *`limit-custodian`*). Moreover, if indvidual allocations that are initially charged to *`limit-custodian`* can be arbitraily large, then *`stop-custodian`* must be the same as *`limit-custodian`*, so that excessively large immediate allocations can be rejected with an `exn:fail:out-of-memory` exception.

`(make-custodian-box` *`custodian v`*`)` returns a custodian box that contains *`v`* as long as *`custodian`* has not been shut down.

`(custodian-box-value` *`custodian-box`*`)` returns the value in the given custodian box, or `#f` if the value has been removed.

`(custodian-box?` *`v`*`)` returns `#t` if *`v`* is a custodian box produced by `make-custodian-box`, `#f` otherwise.

9.3 Thread Groups

A *thread group* is a collection of threads and other thread groups that have equal claim to the CPU. By nesting thread groups and by creating certain threads within certain groups, a programmer can control the amount of CPU allocated to a set of threads. Every thread belongs to a thread group, which is determined by the `current-thread-group` parameter (see §7.9.1.8) when the thread is created. Thread groups and custodians (see §9.2) are independent.

The root thread group receives all of the CPU that the operating system gives MzScheme. Every thread or nested group in a particular thread group receives equal allocation of the CPU (a portion of the group's access), although a thread may relinquish part of its allocation by sleeping or synchronizing with other processes.

`(make-thread-group` [*`thread-group`*]`)` creates a new thread group that belongs to *`thread-group`*. The default value for *`thread-group`* is the current thread group, as determined by the `current-thread-group` parameter.

`(thread-group?` *`v`*`)` returns `#t` if *`v`* is a thread group value, `#f` otherwise.

9.4 Inspectors and Modules

In the same way that inspectors control access to structure fields (see §4.5), inspectors also control access to module bindings (see §5). The default inspector for module bindings is determined by the `current-code-inspector` parameter, instead of the `current-inspector` parameter.

[3]Memory accounting is normally available in MzScheme3m, which is the main variant of MzScheme, and not normally available in MzSchemeCGC.

When a `module` declaration is evaluated, the value of the `current-code-inspector` parameter is associated with the module declaration. When the module is invoked via `require` or `dynamic-require`, a sub-inspector of the module's declaration-time inspector is created, and this sub-inspector is associated with the module invocation. Any inspector that controls the sub-inspector (i.e., the declaration-time inspector and its superior) controls the module invocation.

Control over a module invocation enables

- the use of `module->namespace` on the module;
- access to the module's protected identifiers, i.e. those identifiers exported from the module with `protect`; and
- access to the module's protected and unexported variables within compiled code from `read` (see §14.3).

If the value of `current-code-inspector` never changes, then no control is lost for any module invocation, since the module's invocation is associated with a sub-inspector of `current-code-inspector`.

The inspector for a module invocation is specific to a particular module registry, in case a module is attached to a new registry via `namespace-attach-module`. The invocation inspector in a particular registry can be changed via `namespace-unprotect-module` (but changing the inspector requires control over the old one).

Control over a module declaration (as opposed to a mere invocation) enables the reconstruction of syntax objects that contain references to the module's unexported identifiers. Otherwise, the compiler and macro expander prevent any reference to an unexported identifier, unless the reference appears within an expression that was generated by the module's macros (or, more precisely, a macro from a module whose declaration inspector controls the invocation of the identifier's module). See §12.6.3 for further information.

10. Regular Expressions

MzScheme provides built-in support for regular expression pattern matching on strings, byte strings, and input ports.[1] Regular expressions are specified as strings or byte strings, using the same pattern language as the Unix utility `egrep` or Perl. A string-specified pattern produces a character regexp matcher, and a byte-string pattern produces a byte regexp matcher. If a character regexp is used with a byte string or input port, it matches UTF-8 encodings (see §1.2.3) of matching character streams; if a byte regexp is used with a character string, it matches bytes in the UTF-8 encoding of the string.

Regular expressions can be compiled into a *regexp value* for repeated matches. The `regexp` and *`byte-regexp`* procedures convert a string or byte string (respectively) into a regexp value using one syntax of regular expressions that is most compatible to `egrep`. The `pregexp` and *`byte-pregexp`* procedures produce a regexp value using a slightly different syntax of regular expressions that is more compatible with Perl. In addition, Scheme constants written with `#rx` or `#px` (see §11.2.4) produce compiled regexp values.[2]

For a gentle introduction to regular expression using the `pregexp` syntax, see Chapter 36 of *PLT MzLib: Libraries Manual*.

The two supported regular expression syntaxes share a common core that is shown in Figures 10.1 and 10.2. Figure 10.3 completes the grammar for `regexp`, which treats curly braces ("{" and "}") as literals, backslash ("\") as a literal within ranges, and backslash ("\") as a literal producer outside of ranges. Figures 10.4 and 10.5 complete the grammar for `pregexp`, which uses curly braces ("{" and "}") for bounded repetition and uses backslash ("\") for meta-characters both inside and outside of ranges.

In addition to matching a grammars, regular expressions must meet two syntactic restrictions:

- In a *`Repeat`* other than *`Atom`*?, then *`Atom`* must not match an empty sequence.
- In a (?<=*`Regexp`*) or (?<!*`Regexp`*), the *`Regexp`* must match a bounded sequence, only.

These contraints are checked syntactically by the type system in Figure 10.6 at the end of this chapter. A type $\langle n,m\rangle$ corresponds to an expression that matches between n and m characters. In the rule for (*`Regexp`*), *`N`* means the number such that the opening parenthesis is the *`N`*th opening parenthesis for collecting match reports. Non-emptiness is inferred for a backreference pattern, \*`N`*, so that a backreference can be used for repetition patterns; in the case of mutual dependencies among backreferences, the inference chooses the fixpoint that maximizes non-emptiness. Finiteness is not inferred for backreferences (i.e., a backreference is assumed to match an arbitrarily large sequence).

If a byte string is used to express a grammar, its bytes are interpreted as Latin-1 encodings of characters (see §1.2.3), and the resulting regexp "matches a character" by matching a byte whose Latin-1 decoding is the character. The exception is that \p{*`Property`*} and \P{*`Property`*} match UTF-8 encoded characters with the corresponding *`Property`*.

By default, a regular expression matches characters case-sensitively, and newlines are not treated specially. The *`Mode`*

[1]The implementation is based on Henry Spencer's package.

[2]The internal size of a regexp value is limited to 32 kilobytes; this limit roughly corresponds to a source string with 32,000 literal characters or 5,000 operators.

Regexp	::=	*Pieces*	Match *Pieces*
	|	*Regexp*|*Regexp*	Match either *Regexp*, try left first
Pieces	::=	*Piece*	Match *Piece*
	|	*PiecePieces*	Match first *Piece* followed by second *Pieces*
Piece	::=	*Repeat*	Match *Repeat*, longest possible
	|	*Repeat*?	Match *Repeat*, shortest possible
	|	*Atom*	Match *Atom* exactly once
Repeat	::=	*Atom**	Match *Atom* 0 or more times
	|	*Atom*+	Match *Atom* 1 or more times
	|	*Atom*?	Match *Atom* 0 or 1 times
Atom	::=	(*Regexp*)	Match sub-expression *Regexp* and report match
	|	[*Range*]	Match any character in *Range*
	|	[^*Range*]	Match any character not in *Range*
	|	.	Match any character (except newline in multi mode)
	|	^	Match start of input (or after newline in multi mode)
	|	$	Match end of input (or before newline in multi mode)
	|	*Literal*	Match a single literal character
	|	(?*Mode*:*Regexp*)	Match sub-expression *Regexp* using *Mode*
	|	(?>*Regexp*)	Match sub-expression *Regexp*, only first possible
	|	*Look*	Match empty if *Look* matches
	|	(?*PredPieces*|*Pieces*)	Match first *Pieces* if *Pred*, second *Pieces* if not *Pred*
	|	(?*PredPieces*)	Match *Pieces* if *Pred*, empty if not *Pred*
Range	::=	]	*Range* contains] only
	|	-	*Range* contains - only
	|	*Mrange*	*Range* contains everything in *Mrange*
	|	*Mrange*-	*Range* contains - and everything in *Mrange*
Mrange	::=	]*Lrange*	*Mrange* contains] and everything in *Lrange*
	|	-*Lrange*	*Mrange* contains - and everything in *Lrange*
	|	*Lrange*	*Mrange* contains everything in *Lrange*
Lrange	::=	*Rliteral*	*Lrange* contains a literal character
	|	*Rliteral*-*Rliteral*	*Lrange* contains Unicode range inclusive
	|	*LrangeLrange*	*Lrange* contains everything in both

Figure 10.1: Common grammar for regular expressions

portion of an (?*Mode*:*Regexp*) form changes the matching mode for *Regexp*:

- If the new mode is case-insensitive, then *Regexp* is generalized so that where it matches a particular character, then it also matches lowercase, uppercase, titlecase, and case-folded variants of the same character. For byte-string regular expressions, matching is case-insensitive on ASCII characters, only.
- If the new mode is multi, then a dot (".") in *Regexp* never matches a newline character, but a caret ("^") matches after a newline (in addition to the beginning of the input), and a dollar sign ("$") matches before a newline (in addition to the end of the input).

A few subtle points about the regexp language are worth noting:

- When an opening square bracket ("[") that starts a range is immediately followed by a closing square bracket ("]"), then the closing square bracket is part of the range, instead of ending an empty range. For example, `"[]a]"` matches any string that contains a lowercase "a" or a closing square bracket. A dash ("-") at the start or end of a range is treated specially in the same way.
- When a caret ("^") or dollar sign ("$") appears in the middle of a regular expression (not in a range) and outside of "multi" mode, the resulting regexp is legal even though it is usually not matchable. For example, `"a$b"`

Look	::=	(?=*Regexp*)	Match if *Regexp* matches
	|	(?!*Regexp*)	Match if *Regexp* doesn't match
	|	(?<=*Regexp*)	Match if *Regexp* matches immediately preceeding
	|	(?<!*Regexp*)	Match if *Regexp* doesn't match immediately preceeding
Pred	::=	(*N*)	True if *N*th (has a match
	|	*Look*	True if *Look* matches
Mode	::=		Like the enclosing mode
	|	*Mode*i	Like *Mode*, but case-insensitive
	|	*Mode*-i	Like *Mode*, but sensitive
	|	*Mode*s	Like *Mode*, but not in multi mode
	|	*Mode*-s	Like *Mode*, but in multi mode
	|	*Mode*m	Like *Mode*, but in multi mode
	|	*Mode*-m	Like *Mode*, but not in multi mode

Figure 10.2: Common predicate, lookahead/lookbehind, and mode grammar

Literal	::=	Any character except (,), *, +, ?, [, ., ^, \, or |	
	|	\Aliteral	Match *Aliteral*
Aliteral	::=	Any character	
Rliteral	::=	Any character except] or -	

Figure 10.3: Specific grammar for `regexp`, `byte-regexp`, and `#rx`

is unmatchable, because no string can contain the letter "b" after the end of the string. In contrast, `"a$b*"` matches any string that ends with a lowercase "a", since zero "b"s will match the part of the regexp after "$".

- A backslash ("\") in a regexp pattern specified with a Scheme string literal must be protected with an additional backslash. For example, the string `"\\."` describes a pattern that matches any string containing a period. In this case, the first backslash protects the second to generate a Scheme string containing two characters; the second backslash (which is the first slash in the actual string value) protects the period in the regexp pattern.

The regular expression procedures are as follows:

- `(regexp` *`string`*`)` takes a string representation of a regular expression (using the syntax of Figure 10.3) and compiles it into a regexp value. Other regular expression procedures accept either a string or a regexp value as the matching pattern. If a regular expression string is used multiple times, it is faster to compile the string once to a regexp value and use it for repeated matches instead of using the string each time.

  The `object-name` procedure (see §6.2.3) returns the source string for a regexp value.

- `(pregexp` *`string`*`)` is like `regexp`, except that it uses the syntax of Figure 10.4. The result can be used with `regexp-match`, etc., just like the result from `regexp`.

- `(regexp?` *`v`*`)` returns `#t` if *v* is a regexp value created by `regexp` or `pregexp`, `#f` otherwise.

- `(pregexp?` *`v`*`)` returns `#t` if *v* is a regexp value created by `pregexp` (not `regexp`), `#f` otherwise.

- `(byte-regexp` *`bytes`*`)` takes a byte-string representation of a regular expression (using the syntax of Figure 10.3) and compiles it into a byte-regexp value. The `object-name` procedure (see §6.2.3) returns the source byte string for a regexp value.

- `(byte-pregexp` *`string`*`)` is like `byte-regexp`, except that it uses the syntax of Figure 10.4. The result can be used with `regexp-match`, etc., just like the result from `byte-regexp`.

Repeat	::=	...	see Figure 10.1
	|	*Atom*{*N*}	Match *Atom* exactly *N* times
	|	*Atom*{*N*,}	Match *Atom* *N* or more times
	|	*Atom*{,*M*}	Match *Atom* between 0 and *M* times
	|	*Atom*{*N*,*M*}	Match *Atom* between *N* and *M* times
Atom	::=	...	see Figure 10.1
	|	\N	Match latest reported match for *N*th (
	|	*Class*	Match any character in *Class*
	|	\\b	Match between \\w and \\W, start, or end
	|	\\B	Match between \\w and \\w or \\W and \\W, start, or end
	|	\\p{*Property*}	Match a (UTF-8 encoded) character in *Property*
	|	\\P{*Property*}	Match a (UTF-8 encoded) character not in *Property*
Literal	::=	Any character except (,), *, +, ?, [,], {, }, ., ^, \\, or |	
	|	\Aliteral	Match *Aliteral*
Aliteral	::=	Any character except a-z, A-Z, 0-9	
Lrange	::=	...	see Figure 10.1
	|	*Class*	*Lrange* contains all characters in *Class*
	|	*Posix*	*Lrange* contains all characters in *Posix*
Rliteral	::=	Any character except], \\, or -	

Figure 10.4: Specific grammar for `pregexp`, `byte-pregexp`, and `#px`

- `(byte-regexp? v)` returns `#t` if *v* is a regexp value created by `byte-regexp` or `byte-pregexp`, `#f` otherwise.
- `(byte-pregexp? v)` returns `#t` if *v* is a regexp value created by *byte-pregexp* (not *byte-regexp*), `#f` otherwise.
- `(regexp-match pattern string [start-k end-k output-port])` attempts to match *pattern* (a string, byte string, regexp value, or byte-regexp value) once to a portion of *string*; see below for information on using a byte string or input port in place of *string*.

  The optional *start-k* and *end-k* arguments select a substring of *string* for matching, and the default is the entire string. The *end-k* argument can be `#f`, which is the same as not supplying *end-k*. The matcher finds a portion of *string* that matches *pattern* and is closest to the start of the selected substring.

  If the match fails, `#f` is returned. If the match succeeds, a list containing strings, and possibly `#f`, is returned. The list contains byte strings (substrings of the UTF-8 encoding of *string*) if *pattern* is a byte string or a byte regexp value.

  The first [byte] string in a result list is the portion of *string* that matched *pattern*. If two portions of *string* can match *pattern*, then the match that starts earliest is found.

  Additional [byte] strings are returned in the list if *pattern* contains parenthesized sub-expressions (but not when the open parenthesis is followed by "?:"). Matches for the sub-expressions are provided in the order of the opening parentheses in *pattern*. When sub-expressions occur in branches of an "or" ("|"), in a "zero or more" pattern ("*"), or in a "zero or one" pattern ("?"), a `#f` is returned for the expression if it did not contribute to the final match. When a single sub-expression occurs in a "zero or more" pattern ("*") or a "one or more" pattern ("+") and is used multiple times in a match, then the rightmost match associated with the sub-expression is returned in the list.

  If the optional *output-port* is provided, the part of *string* that precedes the match is written to the port. All of *string* up to *end-k* is written to the port if no match is found. This functionality is not especially useful, but it is provided for consistency with `regexp-match` on input ports. The *output-port* argument can be `#f`, which is the same as not supplying it.
- `(regexp-match pattern bytes [start-k end-k output-port])` is analogous to `regexp-match` with a string (see above). The result is always a list of byte strings and `#f`, even if *pattern* is a character string or a character regexp value.

Class	::=	\d	*Class* contains 0-9
	|	\D	*Class* contains ASCII other than those in \d
	|	\w	*Class* contains a-z, A-Z, 0-9, _
	|	\W	*Class* contains ASCII other than those in \w
	|	\s	*Class* contains space, tab, newline, formfeed, return
	|	\S	*Class* contains ASCII other than those in \s
Posix	::=	[:alpha:]	*Posix* contains a-z, A-Z
	|	[:alnum:]	*Posix* contains a-z, A-Z, 0-9
	|	[:ascii:]	*Posix* contains all ASCII characters
	|	[:blank:]	*Posix* contains space and tab
	|	[:cntrl:]	*Posix* contains all characters with scalar value ¡ 32
	|	[:digit:]	*Posix* contains 0-9
	|	[:graph:]	*Posix* contains all ASCII characters that use ink
	|	[:lower:]	*Posix* contains space, tab, and ASCII ink users
	|	[:print:]	*Posix* contains A-Z
	|	[:space:]	*Posix* contains space, tab, newline, formfeed, return
	|	[:upper:]	*Posix* contains A-Z
	|	[:word:]	*Posix* contains a-z, A-Z, 0-9, _
	|	[:xdigit:]	*Posix* contains 0-9, a-f, A-F
Property	::=	*Category*	*Property* includes all characters in *Category*
	|	ˆ*Category*	*Property* includes all characters not in *Category*
Category	::=	Ll | Lu | Lt | Lm	Unicode general category
	|	L&	Union of Ll, Lu, Lt, and Lm
	|	Lo	Unicode general category
	|	L	Union of L& and Lo
	|	Nd | Nl | No	Unicode general category
	|	*N*	Union of Nd, Nl, and No
	|	Ps | Pe | Pi | Pf	Unicode general category
	|	Pc | Pd | Po	Unicode general category
	|	P	Union of Ps, Pe, Pi, Pf, Pc, Pd, and Po
	|	Mn | Mc | Me	Unicode general category
	|	*M*	Union of Mn, Mc, and Me
	|	Sc | Sk | Sm | So	Unicode general category
	|	S	Union of Sc, Sk, Sm, and So
	|	Zl | Zp | Zs	Unicode general category
	|	Z	Union of Zl, Zp, and Zs
	|	.	Union of all general categories

Figure 10.5: Properties and classes for `pregexp` (Figure 10.4)

- `(regexp-match` *`pattern input-port`* `[`*`start-k end-k output-port`*`])` is similar to `regexp-match` with a byte string (see above), except that the match is found in the stream of bytes produced by *`input-port`*. The optional *`start-k`* argument indicates the number of bytes to skip before matching *`pattern`*, and *`end-k`* indicates the maximum number of bytes to consider (including skipped bytes). The *`end-k`* argument can be `#f`, which is the same as not supplying *`end-k`*. The default is to skip no bytes and read until the end-of-file if necessary. If the end-of-file is reached before *`start-k`* bytes are skipped, the match fails.

  In *`pattern`*, a start-of-string caret ("ˆ") refers to the first read position after skipping, and the end-of-string dollar sign ("$") refers to the *`end-k`*th read byte or the end of file, whichever comes first.

  The optional *`output-port`* receives all bytes that precede a match in the input port, or up to *`end-k`* bytes (by default the entire stream) if no match is found. The *`output-port`* argument can be `#f`, which is the same as not supplying it.

  When matching an input port stream, a match failure reads up to *`end-k`* bytes (or end-of-file), even if *`pattern`* begins with a start-of-string caret ("ˆ"); see also `regexp-match/fail-without-reading` in Chapter 45 of *PLT MzLib: Libraries Manual*. On success, all bytes up to and including the match are eventually read from the port, but matching proceeds by first peeking bytes from the port (using `peek-bytes-avail!`; see §11.2.1), and then (re-)reading matching bytes to discard them after the match result is determined. Non-matching bytes may be read and discarded before the match is determined. The matcher peeks in blocking mode only as far as necessary to determine a match, but it may peek extra bytes to fill an internal buffer if immediately available (i.e., without blocking). Greedy repeat operators in *pattern*, such as "*" or "+", tend to force reading the entire content of the port (up to *end-k*) to determine a match.

If the port is read simultaneously by another thread, or if the port is a custom port with inconsistent reading and peeking procedures (see §11.1.7), then the bytes that are peeked and used for matching may be different than the bytes read and discarded after the match completes; the matcher inspects only the peeked bytes. To avoid such interleaving, use `regexp-match-peek` (with a *progress-evt* argument) followed by `port-commit-peeked`.

- `(regexp-match-positions` *pattern string/bytes/input-port* [*start-k end-k output-port*]`)` is like `regexp-match`, but returns a list of number pairs (and `#f`) instead of a list of strings. Each pair of numbers refers to a range of characters or bytes in *string/bytes/input-port*. If the result for the same arguments with `regexp-match` would be a list of byte strings, the resulting ranges correspond to byte ranges; in that case, if *string/bytes/input-port* is a character string, the byte ranges correspond to bytes in the UTF-8 encoding of the string.

  Range results are returned in a `substring`- and `subbytpe`-compatible manner, independent of *start-k*. In the case of an input port, the returned positions indicate the number of bytes that were read, including *start-k*, before the first matching byte.

- `(regexp-match?` *pattern string/bytes/input-port* [*start-k end-k output-port*]`)` is like `regexp-match`, but returns merely `#t` when the match succeeds, `#f` otherwise.

- `(regexp-match-peek` *pattern input-port* [*start-k end-k progress-evt*]`)` is like `regexp-match` on input ports, but only peeks bytes from *input-port* instead of reading them. Furthermore, instead of an output port, the last optional argument is a progress event for *input-port* (see §11.2.1). If *progress-evt* becomes ready, then the match stops peeking from *input-port* and returns `#f`. The *progress-evt* argument can be `#f`, in which case the peek may continue with inconsistent information if another process meanwhile reads from *input-port*.

- `(regexp-match-peek-positions` *pattern input-port* [*start-k end-k progress-evt*]`)` is like `regexp-match-positions` on input ports, but only peeks bytes from *input-port* instead of reading them.

- `(regexp-match-peek-immediate` *pattern input-port* [*start-k end-k progress-evt*]`)` is like `regexp-match-peek`, but it attempts to match only bytes that are available from *input-port* without blocking. The match fails if not-yet-available characters might be used to match *pattern*.

- `(regexp-match-peek-positions-immediate` *pattern input-port* [*start-k end-k progress-evt*]`)` is like `regexp-match-peek-positions`, but it attempts to match only bytes that are available from *input-port* without blocking. The match fails if not-yet-available characters might be used to match *pattern*.

- `(regexp-replace` *char-pattern string insert-string*`)` performs a match using *char-pattern* on *string* and then returns a string in which the matching portion of *string* is replaced with *insert-string*. If *char-pattern* matches no part of *string*, then *string* is returned unmodified.

  The *char-pattern* must be a string or a character regexp value (not a byte string or a byte regexp value).

  If *insert-string* contains "&", then "&" is replaced with the matching portion of *string* before it is substituted into *string*. If *insert-string* contains "\n" (for some integer *n*), then it is replaced with the *n*th matching sub-expression from *string*.[3] "&" and "\0" are synonymous. If the *n*th sub-expression was not

[3]The backslash is a character in the string, so an extra backslash is required to specify the string as a Scheme constant. For example, the Scheme constant `"\\1"` is "\1".

used in the match or if *n* is greater than the number of sub-expressions in *pattern*, then "\n" is replaced with the empty string.

A literal "&" or "\\" is specified as "\\&" or "\\\\", respectively. If *insert-string* contains "\\$", then "\\$" is replaced with the empty string. (This can be used to terminate a number *n* following a backslash.) If a "\\" is followed by anything other than a digit, "&", "\\", or "$", then it is treated as "\\0".

- `(regexp-replace` *byte-pattern string-or-bytes insert-string-or-bytes*`)` is analogous to `regexp-replace` on strings, where *byte-pattern* is a byte string or a byte regexp value. The result is always a byte string.
- `(regexp-replace` *char-pattern string proc*`)` is like `regexp-replace`, but instead of an *insert-string* third argument, the third argument is a procedure that accepts match strings and produces a string to replace the match. The *proc* must accept the same number of arguments as `regexp-match` produces list elements for a successful match with *char-pattern*.
- `(regexp-replace` *byte-pattern string-or-bytes proc*`)` is analogous to `regexp-replace` on strings and a procedure argument, but the procedure accepts byte strings to produce a byte string, instead of character strings.
- `(regexp-replace*` *pattern string insert-string*`)` is the same as `regexp-replace`, except that every instance of *pattern* in *string* is replaced with *insert-string*. Only non-overlapping instances of *pattern* in the original *string* are replaced, so instances of *pattern* within inserted strings are *not* replaced recursively. If, in the process of repeating matches, *pattern* matches an empty string, the `exn:fail` exception is raised.
- `(regexp-replace*` *byte-pattern bytes insert-bytes*`)` is analogous to `regexp-replace*` on strings.
- `(regexp-replace*` *char-pattern string proc*`)` is like `regexp-replace` with a procedure argument, but with multiple instances replaced. The given *proc* is called once for each match.
- `(regexp-replace*` *byte-pattern bytes proc*`)` is like `regexp-replace*` with a string and procedure argument, but the procedure accepts and produces byte strings.

Examples:

```
(define r (regexp "(-[0-9]*)+"))
(regexp-match r "a-12--345b") ; ⇒ '("-12--345" "-345")
(regexp-match-positions r "a-12--345b") ; ⇒ '((1 . 10) (5 . 10))
(regexp-match "x+" "12345") ; ⇒ #f
(regexp-replace "mi" "mi casa" "su") ; ⇒ "su casa"
(regexp-replace "mi" "mi casa" string-upcase) ; ⇒ "MI casa"

(define r2 (regexp "([Mm])i ([a-zA-Z]*)"))
(define insert "\\1y \\2")
(regexp-replace r2 "Mi Casa" insert) ; ⇒ "My Casa"
(regexp-replace r2 "mi cerveza Mi Mi Mi" insert) ; ⇒ "my cerveza Mi Mi Mi"
(regexp-replace* r2 "mi cerveza Mi Mi Mi" insert) ; ⇒ "my cerveza My Mi Mi"
(regexp-replace* r2 "mi cerveza Mi Mi Mi"
                 (lambda (all one two)
                   (string-append (string-downcase one) "y"
                                  (string-upcase two)))) ; ⇒ "myCERVEZA myMI Mi"

(define p (open-input-string "a abcd"))
(regexp-match-peek ".*bc" p) ; ⇒ '("a abc")
(regexp-match-peek ".*bc" p 2) ; ⇒ '("abc")
```

```
(regexp-match ".*bc" p 2) ; ⇒ '("abc")
(peek-char p) ; ⇒ #\d
(regexp-match ".*bc" p) ; ⇒ #f
(peek-char p) ; ⇒ #<eof>

(define p (open-input-string "aaaaaaaaaaa abcd"))
(define o (open-output-string))
(regexp-match "abc" p 0 #f o) ; ⇒ '("abc")
(get-output-string o) ; ⇒ "aaaaaaaaaaa "

(define r (byte-regexp #"(-[0-9]*)+"))
(regexp-match r #"a-12--345b") ; ⇒ '(#"-12--345" "-345")
(regexp-match #".." #"\uC8x")  ; ⇒ '(#"\310x")
;; The UTF-8 encoding of #\uC8 is two bytes: 195 followed by 136
(regexp-match #".." "\uC8x")  ; ⇒ '(#"\303\210")
```

$$\frac{\mathit{Regexp}_1 : \langle n_1,m_1\rangle \quad \mathit{Regexp}_2 : \langle n_2,m_2\rangle}{\mathit{Regexp}_1|\mathit{Regexp}_2 : <(n_1,n_2),(m_1,m_2)>} \qquad \frac{\mathit{Piece} : \langle n_1,m_1\rangle \quad \mathit{Pieces} : \langle n_2,m_2\rangle}{\mathit{PiecePieces} : \langle n_1+n_2, m_1+m_2\rangle}$$

$$\frac{\mathit{Repeat} : \langle n,m\rangle}{\mathit{Repeat}? : \langle n,m\rangle} \qquad \frac{\mathit{Atom} : \langle n,m\rangle \quad n>0}{\mathit{Atom}* : \langle 0,\infty\rangle}$$

$$\frac{\mathit{Atom} : \langle n,m\rangle \quad n>0}{\mathit{Atom}+ : \langle 1,\infty\rangle} \qquad \frac{\mathit{Atom} : \langle n,m\rangle}{\mathit{Atom}? : \langle 0,m\rangle} \qquad \frac{\mathit{Atom} : \langle n,m\rangle \quad n>0}{\mathit{Atom}\{N\} : \langle n\cdot N, m\cdot N\rangle}$$

$$\frac{\mathit{Atom} : \langle n,m\rangle \quad n>0}{\mathit{Atom}\{N,\} : \langle n\cdot N,\infty\rangle} \qquad \frac{\mathit{Atom} : \langle n,m\rangle \quad n>0}{\mathit{Atom}\{,M\} : \langle 0, m\cdot M\rangle} \qquad \frac{\mathit{Atom} : \langle n,m\rangle \quad n>0}{\mathit{Atom}\{N,M\} : \langle n\cdot N, m\cdot M\rangle}$$

$$\frac{\mathit{Regexp} : \langle n,m\rangle}{(\mathit{Regexp}) : \langle n,m\rangle \quad \alpha_N = n} \qquad \frac{\mathit{Regexp} : \langle n,m\rangle}{(?\mathit{Mode}:\mathit{Regexp}) : \langle n,m\rangle} \qquad \frac{\mathit{Regexp} : \langle n,m\rangle}{(?=\mathit{Regexp}) : \langle 0,0\rangle}$$

$$\frac{\mathit{Regexp} : \langle n,m\rangle}{(?!\mathit{Regexp}) : \langle 0,0\rangle} \qquad \frac{\mathit{Regexp} : \langle n,m\rangle \quad m<\infty}{(?<=\mathit{Regexp}) : \langle 0,0\rangle} \qquad \frac{\mathit{Regexp} : \langle n,m\rangle \quad m<\infty}{(?<!\mathit{Regexp}) : \langle 0,0\rangle}$$

$$\frac{\mathit{Regexp} : \langle n,m\rangle}{(?>\mathit{Regexp}) : \langle n,m\rangle} \qquad \frac{\mathit{Pred} : \langle n_0,m_0\rangle \quad \mathit{Pieces}_1 : \langle n_1,m_1\rangle \quad \mathit{Pieces}_2 : \langle n_2,m_2\rangle}{(?\mathit{PredPieces}_1|\mathit{Pieces}_2) : <(n_1,n_2),(m_1,m_2)>}$$

$$\frac{\mathit{Pred} : \langle n_0,m_0\rangle \quad \mathit{Pieces} : \langle n_1,m_1\rangle}{(?\mathit{PredPieces}) : \langle 0,m_1\rangle} \qquad (N) : \langle \alpha_N,\infty\rangle$$

$$[\mathit{Range}] : \langle 1,1\rangle \qquad [\hat{}\mathit{Range}] : \langle 1,1\rangle \qquad . : \langle 1,1\rangle \qquad \hat{} : \langle 0,0\rangle$$

$$\$: \langle 0,0\rangle \qquad \mathit{Literal} : \langle 1,1\rangle \qquad \backslash N : \langle \alpha_N,\infty\rangle \qquad \mathit{Class} : \langle 1,1\rangle$$

$$\backslash\mathrm{b} : \langle 0,0\rangle \qquad \backslash\mathrm{B} : \langle 0,0\rangle \qquad \backslash\mathrm{p}\{\mathit{Property}\} : \langle 1,6\rangle \qquad \backslash\mathrm{P}\{\mathit{Property}\} : \langle 1,6\rangle$$

Figure 10.6: Type rules for regular expressions

11. Input and Output

11.1 Ports

By definition, ports in MzScheme produce and consume bytes. When a port is provided to a character-based operation, such as `read`, the port's bytes are read and interpreted as a UTF-8 encoding of characters (see also §1.2.3). Thus, reading a single character may require reading multiple bytes, and a procedure like `char-ready?` may need to peek several bytes into the stream to determine whether a character is available. In the case of a byte stream that does not correspond to a valid UTF-8 encoding, functions such as `read-char` may need to peek one byte ahead in the stream to discover that the stream is not a valid encoding.

When an input port produces a sequence of bytes that is not a valid UTF-8 encoding in a character-reading context, then bytes that constitute an invalid sequence are converted to the character "?". Specifically, bytes 255 and 254 are always converted to "?", bytes in the range 192 to 253 produce "?" when they are not followed by bytes that form a valid UTF-8 encoding, and bytes in the range 128 to 191 are converted to "?" when they are not part of a valid encoding that was started by a preceding byte in the range 192 to 253. To put it another way, when reading a sequence of bytes as characters, a minimal set of bytes are changed to 63[1] so that the entire sequence of bytes is a valid UTF-8 encoding.

See §3.6 for procedures that facilitate conversions using UTF-8 or other encodings. See also `reencode-input-port` and `reencode-output-port` in Chapter 35 of *PLT MzLib: Libraries Manual* for obtaining a UTF-8-based port from one that uses a different encoding of characters.

`(port? v)` returns `#t` if either `(input-port? v)` or `(output-port? v)` is `#t`, `#f` otherwise.

`(port-closed? port)` returns `#t` if the input or output port *port* is closed, `#f` otherwise.

`(file-stream-port? port)` returns `#t` if the given port is a file-stream port (see §11.1.6, `#f` otherwise.

`(terminal-port? port)` returns `#t` if the given port is attached to an interactive terminal, `#f` otherwise.

11.1.1 End-of-File Constant

The global variable `eof` is bound to the end-of-file value. The standard Scheme predicate `eof-object?` returns `#t` only when applied to this value.

Reading from a port produces an end-of-file result when the port has no more data, but some ports may also return end-of-file mid-stream. For example, a port connected to a Unix terminal returns an end-of-file when the user types control-d; if the user provides more input, the port returns additional bytes after the end-of-file.

11.1.2 Current Ports

The standard Scheme procedures `current-input-port` and `current-output-port` are implemented as parameters in MzScheme. See §7.9.1.2 for more information.

[1] 63 is the same as `(char->integer #\?)`.

11.1.3 Opening File Ports

The `open-input-file` and `open-output-file` procedures accept an optional flag argument after the filename that specifies a mode for the file:

- `'binary` — bytes are returned from the port exactly as they are read from the file. Binary mode is the default mode.
- `'text` — return and linefeed bytes (10 and 13) are written to and read from the file are filtered by the port in a platform specific manner:
  - **Unix and Mac OS X**: no filtering occurs.
  - **Windows reading**: a return-linefeed combination from a file is returned by the port as a single linefeed; no filtering occurs for return bytes that are not followed by a linefeed, or for a linefeed that is not preceded by a return.
  - **Windows writing**: a linefeed written to the port is translated into a return-linefeed combination in the file; no filtering occurs for returns.

  In Windows, `'text` mode works only with regular files; attempting to use `'text` with other kinds of files triggers an `exn:fail:filesystem` exception.

The `open-output-file` procedure can also take a flag argument that specifies how to proceed when a file with the specified name already exists:

- `'error` — raise `exn:fail:filesystem` (this is the default)
- `'replace` — remove the old file and write a new one
- `'truncate` — overwrite the old data
- `'truncate/replace` — try `'truncate`; if it fails, try `'replace`
- `'append` — append to the end of the file under Unix and Mac OS X; under Windows, `'append` is equivalent to `'update`, except that the file position is immediately set to the end of the file after opening it
- `'update` — open an existing file without truncating it; if the file does not exist, the `exn:fail:filesystem` exception is raised

The `open-input-output-file` procedure takes the same arguments as `open-output-file`, but it produces two values: an input port and an output port. The two ports are connected in that they share the underlying file device. This procedure is intended for use with special devices that can be opened by only one process, such as **COM1** in Windows. For regular files, sharing the device can be confusing. For example, using one port does not automatically flush the other port's buffer (see §11.1.6 for more information about buffers), and reading or writing in one port moves the file position (if any) for the other port. For regular files, use separate `open-input-file` and `open-output-file` calls to avoid confusion.

Extra flag arguments are passed to `open-output-file` in any order. Appropriate flag arguments can also be passed as the last argument(s) to `call-with-input-file`, `with-input-from-file`, `call-with-output-file`, and `with-output-to-file`. When conflicting flag arguments (e.g., both `'error` and `'replace`) are provided to `open-output-file`, `with-output-to-file`, or `call-with-output-file`, the `exn:fail:contract` exception is raised.

Both `with-input-from-file` and `with-output-to-file` close the port they create if control jumps out of the supplied thunk (either through a continuation or an exception), and the port remains closed if control jumps back into the thunk. The current input or output port is installed and restored with `parameterize` (see §7.9.2).

See §11.1.6 for more information on file ports. When an input or output file-stream port is created, it is placed into the management of the current custodian (see §9.2).

11.1.4 Pipes

`(make-pipe [`*`limit-k input-name-v output-name-v`*`])` returns two port values (see §2.2): the first port is an input port and the second is an output port. Data written to the output port is read from the input port. The ports do not need to be explicitly closed.

The optional *`limit-k`* argument can be `#f` or a positive exact integer. If *`limit-k`* is omitted or `#f`, the new pipe holds an unlimited number of unread bytes (i.e., limited only by the available memory). If *`limit-k`* is a positive number, then the pipe will hold at most *`limit-k`* unread/unpeeked bytes; writing to the pipe's output port thereafter will block until a read or peek from the input port makes more space available. (Peeks effectively extend the port's capacity until the peeked bytes are read.)

The optional *`input-name-v`* and *`output-name-v`* are used as the names for the returned input and out ports, respectively, if they are supplied. Otherwise, the name of each port is `'pipe`.

`(pipe-content-length `*`pipe-port`*`)` returns the number of bytes contained in a pipe, where *`pipe-port`* is either of the pipe's ports produced by `make-pipe`. The pipe's content length counts all bytes that have been written to the pipe and not yet read (though possibly peeked).

11.1.5 String Ports

Scheme input and output can be read from or collected into a string or byte string:

- `(open-input-bytes `*`bytes`*` [`*`name-v`*`])` creates an input port that reads characters from *`bytes`* (see §3.6). Modifying *`bytes`* afterward does not affect the byte stream produced by the port. The optional *`name-v`* argument is used as the name for the returned port; the default is `'string`.
- `(open-input-string `*`string`*` [`*`name-v`*`])` creates an input port that reads bytes from the UTF-8 encoding (see §1.2.3) of *`string`*. The optional *`name-v`* argument is used as the name for the returned port; the default is `'string`.
- `(open-output-bytes [`*`name-v`*`])` creates an output port that accumulates the output into a byte string. The optional *`name-v`* argument is used as the name for the returned port; the default is `'string`.
- `(open-output-string [`*`name-v`*`])` creates an output port that accumulates the output into a byte string. This procedure is the same as `open-output-bytes`.
- `(get-output-bytes `*`string-output-port`*` [`*`reset? start-k end-k`*`])` returns the bytes accumulated in *`string-output-port`* so far in a freshly-allocated byte string (including any bytes written after the port's current position, if any). If *`reset?`* is true, then all bytes are removed from the port, and the port's position is reset to `0`; if *`reset?`* is `#f` (the default), then all bytes remain in the port for further accumulation (so they are returned for later calls to `get-output-bytes` or `get-output-string`), and the port's position is unchanged. The *`start-k`* and *`end-k`* arguments specify the range of bytes in the port to return; supplying *`start-k`* and *`end-k`* is the same as using `subbytes` on the result of `get-output-bytes`, but supplying them to `get-output-bytes` can avoid an allocation. The *`end-k`* argument can be `#f`, which corresponds to not passing a second argument to `subbytes`.
- `(get-output-string `*`string-output-port`*`)` returns `(bytes->string/utf-8 (get-output-bytes `*`string-output-port`*`) #\?)`; see also §3.6.

String input and output ports do not need to be explicitly closed. The `file-position` procedure, described in §11.1.6, works for string ports in position-setting mode.

Example:

```
(define i (open-input-string "hello world"))
(define o (open-output-string))
(write (read i) o)
(get-output-string o) ; ⇒ "hello"
```

11.1.6 File-Stream Ports

A port created by `open-input-file`, `open-output-file`, `subprocess`, and related functions is a *file-stream port*. The initial input, output, and error ports in stand-alone MzScheme are also file-stream ports. The `file-stream-port?` predicate recognizes file-stream ports.

An input port is block buffered by default, which means that on any read, the buffer is filled with immediately-available bytes to speed up future reads. Thus, if a file is modified between a pair of reads to the file, the second read can produce stale data. Calling `file-position` to set an input port's file position flushes its buffer.

Most output ports are block buffered by default, but a terminal output port is line buffered, and the error output port is unbuffered. An output buffer is filled with a sequence of written bytes to be committed as a group, either when the buffer is full (in block mode) or when a newline is written (in line mode).

A port's buffering can be changed via `file-stream-buffer-mode` (described below). The two ports produced by `open-input-output-file` have independent buffers.

The following procedures work primarily on file-stream ports:

- `(flush-output [`*`output-port`*`])` forces all buffered data in the given output port to be physically written. If *`output-port`* is omitted, then the current output port is flushed. Only file-stream ports and custom ports (see §11.1.7) use buffers; when called on a port without a buffer, `flush-output` has no effect.

  By default, a file-stream port is block-buffered, but this behavior can be modified with `file-stream-buffer-mode`. In addition, the initial current output and error ports are automatically flushed when `read`[2], `read-line`, `read-bytes`, `read-string`, etc. are performed on the initial standard input port.

- `(file-stream-buffer-mode` *`port`* `[`*`mode-symbol`*`])` gets or sets the buffer mode for *`port`*, if possible. All file-stream ports support setting the buffer mode, TCP ports (see §11.4) support setting and getting the buffer mode, and custom ports (see §11.1.7) may support getting and setting buffer modes.

  If *`mode-symbol`* is provided, it must be one of `'none`, `'line` (output only), or `'block`, and the port's buffering is set accordingly. If the port does not support setting the mode, the `exn:fail` exception is raised.

  If *`mode-symbol`* is not provided, the current mode is returned, or `#f` is returned if the mode cannot be determined. If *`file-stream-port`* is an input port and *`mode-symbol`* is `'line`, the `exn:fail:contract` exception is raised.

  For an input port, peeking always places peeked bytes into the port's buffer, even when the port's buffer mode is `'none`; furthermore, on some platforms, testing the port for input (via `char-ready?` or `sync`) may be implemented with a peek. If an input port's buffer mode is `'none`, then at most one byte is read for `read-bytes-avail!*`, `read-bytes-avail!`, `peek-bytes-avail!*`, or `peek-bytes-avail!`; if any bytes are buffered in the port (e.g., to satisfy a previous peek), the procedures may access multiple buffered bytes, but no further bytes are read.

- `(file-position` *`port`*`)` returns the current read/write position of *`port`*. For file-stream and string ports, `(file-position` *`port k-or-eof`*`)` sets the read/write position to *`k-or-eof`* relative to the beginning of the file/string if *`k-or-eof`* is a number, or to the current end of the file/string if *`k-or-eof`* is `eof`. In position-setting mode, `file-position` raises the `exn:fail:contract` exception for port kinds other than file-stream and string ports. Calling `file-position` without a position on a non-file/non-string input

[2]Flushing is performed by the default port read handler (see §11.2.6) rather than by `read` itself.

port returns the number of bytes that have been read from that port if the position is known (see §11.2.1.1), otherwise the `exn:fail:filesystem` exception is raised.

When `(file-position` *`port k`*`)` sets the position *`k`* beyond the current size of an output file or string, the file/string is enlarged to size *`k`* and the new region is filled with `#\nul`. If *`k`* is beyond the end of an input file or string, then reading thereafter returns `eof` without changing the port's position.

Not all file-stream ports support setting the position. If `file-position` is called with a position argument on such a file-stream port, the `exn:fail:filesystem` exception is raised.

When changing the file position for an output port, the port is first flushed if its buffer is not empty. Similarly, setting the position for an input port clears the port's buffer (even if the new position is the same as the old position). However, although input and output ports produced by *`open-input-output-file`* share the file position, setting the position via one port does not flush the other port's buffer.

- `(port-file-identity` *`file-stream-port`*`)` returns an exact positive integer that represents the identity of the device and file read or written by *`file-stream-port`*. For two ports whose open times overlap, the result of `port-file-identity` is the same for both ports if and only if the ports access the same device and file. For ports whose open times do not overlap, no guarantee is provided for the port identities (even if the ports actually access the same file) — except as can be inferred through relationships with other ports. If *`file-stream-port`* is closed, the `exn:fail` exception is raised. Under Windows 95, 98, and Me, if *`file-stream-port`* is connected to a pipe instead of a file, the `exn:fail:filesystem` exception is raised.

11.1.7 Custom Ports

The `make-input-port` and `make-output-port` procedures create custom ports with arbitrary control procedures. Correctly implementing a custom port can be tricky, because it amounts to implementing a device driver. Custom ports are mainly useful to obtain fine control over the action of committing bytes as read or written.

Many simple port variations can be implemented using threads and pipes. For example, if *`get-next-char`* is a function that produces either a character or `eof`, it can be turned into an input port as follows

```
(let-values ([(r w) (make-pipe 4096)])
  ;; Create a thread to move chars from get-next-char to the pipe
  (thread (lambda () (let loop ()
                         (let ([v (get-next-char)])
                           (if (eof-object? v)
                               (close-output-port w)
                               (begin
                                 (write-char v w)
                                 (loop)))))))
   ;; Return the read end of the pipe
   r)
```

The **port.ss** in MzLib provides several other port constructors; see Chapter 35 of *PLT MzLib: Libraries Manual*.

11.1.7.1 Custom Input

`(make-input-port` *`name-v read-proc optional-peek-proc close-proc`* `[`*`optional-progress-evt-proc`*
`optional-commit-proc optional-location-proc count-lines!-proc init-position optional-buffe`
creates an input port. The port is immediately open for reading. If *`close-proc`* procedure has no side effects, then the port need not be explicitly closed.

- *`name-v`* — the name for the input port, which is reported by `object-name` (see §6.2.3).

- `read-proc` — a procedure that takes a single argument: a mutable byte string to receive read bytes. The procedure's result is one of the following:
  - the number of bytes read, as an exact, non-negative integer;
  - `eof`;
  - a procedure of arity four (representing a "special" result, as discussed further below) and optionally of arity zero, but a procedure result is allowed only when `optional-peek-proc` is not `#f`; or
  - a synchronizable event (see §7.7) that becomes ready when the read is complete (roughly): the event's value can one of the above three results or another event like itself; in the last case, a reading process loops with `sync` until it gets a non-event result.

  The `read-proc` procedure must not block indefinitely. If no bytes are immediately available for reading, the `read-proc` must return `0` or an event, and preferably an event (to avoid busy waits). The `read-proc` should not return `0` (or an event whose value is `0`) when data is available in the port, otherwise polling the port will behave incorrectly. An event result from an event can also break polling.

  If the result of a `read-proc` call is not one of the above values, the `exn:fail:contract` exception is raised. If a returned integer is larger than the supplied byte string's length, the `exn:fail:contract` exception is raised. If `optional-peek-proc` is `#f` and a procedure for a special result is returned, the `exn:fail:contract` exception is raised.

  The `read-proc` procedure can report an error by raising an exception, but only if no bytes are read. Similarly, no bytes should be read if `eof`, an event, or a procedure is returned. In other words, no bytes should be lost due to spurious exceptions or non-byte data.

  A port's reading procedure may be called in multiple threads simultaneously (if the port is accessible in multiple threads), and the port is responsible for its own internal synchronization. Note that improper implementation of such synchronization mechanisms might cause a non-blocking read procedure to block indefinitely.

  If `optional-peek-proc`, `optional-progress-evt-proc`, and `optional-commit-proc` are all provided and non-`#f`, then the following is an acceptable implementation of `read-proc`:

```
(lambda (bstr)
  (let* ([progress-evt (progress-evt-proc)]
         [v (peek-proc bstr 0 progress-evt)])
    (cond
     [(sync/timeout 0 progress-evt) 0] ; try again
     [(evt? v) (wrap-evt v (lambda (x) 0))] ; sync, then try again
     [(and (number? v) (zero? v)) 0] ; try again
     [else
      (if (optional-commit-proc (if (number? v) v 1)
                                progress-evt
                                always-evt)
          v      ; got a result
          0)]))) ; try again
```

  An implementor may choose not to implement the `optional-` procedures, however, and even an implementor who does supply `optional-` procedures may provide a different `read-proc` that uses a fast path for non-blocking reads.

- `optional-peek-proc` — either `#f` or a procedure that takes three arguments:
  - a mutable byte string to receive peeked bytes;
  - a non-negative number of bytes (or specials) to skip before peeking; and
  - either `#f` or a progress event produced by `optional-progress-evt-proc`.

  The results and conventions for `optional-peek-proc` are mostly the same as for `read-proc`. The main difference is in the handling of the progress event, if it is not `#f`. If the given progress event becomes ready, the `optional-peek-proc` must abort any skip attempts and not peek any values. In particular, `optional-peek-proc` must not peek any values if the progress event is initially ready.

Unlike *read-proc*, *optional-peek-proc* should produce `#f` (or an event whose value is `#f`) if no bytes were peeked because the progress event became ready. Like *read-proc*, a `0` result indicates that another attempt is likely to succeed, so `0` is inappropriate when the progress event is ready. Also like *read-proc*, *optional-peek-proc* must not block indefinitely.

The skip count provided to *optional-peek-proc* is a number of bytes (or specials) that must remain present in the port—in addition to the peek results—when the peek results are reported. If a progress event is supplied, then the peek is effectively canceled when another process reads data before the given number can be skipped. If a progress event is not supplied and data is read, then the peek must effectively restart with the original skip count.

The system does not check that multiple peeks return consistent results, or that peeking and reading produce consistent results.

If *optional-peek-proc* is `#f`, then peeking for the port is implemented automatically in terms of reads, but with several limitations. First, the automatic implementation is not thread-safe. Second, the automatic implementation cannot handle special results (non-byte and non-eof), so *read-proc* cannot return a procedure for a special when *optional-peek-proc* is `#f`. Finally, the automatic peek implementation is incompatible with progress events, so if *optional-peek-proc* is `#f`, then *progress-evt-proc* and *optional-commit-proc* must be `#f`. See also `make-input-port/peek-to-read` in Chapter 35 of *PLT MzLib: Libraries Manual*.

- *close-proc* — a procedure of zero arguments that is called to close the port. The port is not considered closed until the closing procedure returns. The port's procedures will never be used again via the port after it is closed. However, the closing procedure can be called simultaneously in multiple threads (if the port is accessible in multiple threads), and it may be called during a call to the other procedures in another thread; in the latter case, any outstanding reads and peeks should be terminated with an error.

- *optional-progress-evt-proc* — either `#f` (the default), or a procedure that takes no arguments and returns an event. The event must become ready only after data is next read from the port or the port is closed. After the event becomes ready, it must remain so. (See also `semaphore-peek-evt` in §7.4.)

  If *optional-progress-evt-proc* is `#f`, then `port-provides-progress-evts?` applied to the port will produce `#f`, and the port will not be a valid argument to `port-progress-evt`.

- *optional-commit-proc* — either `#f` (the default), or a procedure that takes three arguments:
  - an exact, positive integer k_r;
  - a progress event produced by *optional-progress-evt-proc*;
  - an event, *done-evt*, that is either a channel-put event, channel, semaphore, semaphore-peek event, always event, or never event.

  A *commit* corresponds to removing data from the stream that was previously peeked, but only if no other process removed data first. (The removed data does not need to be reported, because it has been peeked already.) More precisely, assuming that k_p bytes, specials, and mid-stream `eof`s have been previously peeked or skipped at the start of the port's stream, *optional-commit-proc* must satisfy the following constraints:
  - It must return only when the commit is complete or when the given progress event becomes ready.
  - It must commit only if k_p is positive.
  - If it commits, then it must do so with either k_r items or k_p items, whichever is smaller, and only if k_p is positive.
  - It must never choose *done-evt* in a synchronization after the given progress event is ready, or after *done-evt* has been synchronized once.
  - It must not treat any data as read from the port unless *done-evt* is chosen in a synchronization.
  - It must not block indefinitely if *done-evt* is ready; it must return soon after the read completes or soon after the given progress event is ready, whichever is first.
  - It can report an error by raising an exception, but only if no data is committed. In other words, no data should be lost due to an exception, including a break exception.
  - It must return a true value if data is committed, `#f` otherwise. When it returns a value, the given progress event must be ready (perhaps because data was just committed).

  - It must raise an exception if no data (including `eof`) has been peeked from the beginning of the port's stream, or if it would have to block indefinitely to wait for the given progress event to become ready.

  A call to *`optional-commit-proc`* is `parameterize-break`ed to disable breaks.

- *`optional-location-proc`* — either `#f` (the default), or a procedure that takes no arguments and returns three values: the line number for the next item in the port's stream (a positive number or `#f`), the column number for the next item in the port's stream (a non-negative number or `#f`), and the position for the next item in the port's stream (a positive number or `#f`). See also §11.2.1.1.

  This procedure is only called if line counting is enabled for the port via `port-count-lines!` (in which case *`count-lines!-proc`* is called). The `read`, `read-syntax`, `read-honu`, and `read-honu-syntax` procedures assume that reading a non-whitespace character increments the column and position by one.

- *`count-lines!-proc`* — a procedure of no arguments that is called if and when line counting is enabled for the port. The default procedure is `void`.

- *`init-position`* — an exact, positive integer that determines the position of the port's first item, used when line counting is *not* enabled for the port. The default is `1`.

- *`optional-buffer-mode-proc`* — either `#f` (the default) or a procedure that accepts zero or one arguments. If *`optional-buffer-mode-proc`* is `#f`, then the resulting port does not support a buffer-mode setting. Otherwise, the procedure is called with one symbol argument (`'block` or `'none`) to set the buffer mode, and it is called with zero arguments to get the current buffer mode. In the latter case, the result must be `'block`, `'none`, or `#f` (unknown). See §11.1.6 for more information on buffer modes.

When *`read-proc`* or *`optional-peek-proc`* (or an event produced by one of these) returns a procedure, and the procedure is used to obtain a non-byte result.[3] The procedure is called by `read`,[4] `read-syntax`, `read-honu`, `read-honu-syntax`, `read-byte-or-special`, `read-char-or-special`, `peek-byte-or-special`, or `peek-char-or-special`. The special-value procedure can return an arbitrary value, and it will be called zero or one times (not necessarily before further reads or peeks from the port). See §11.2.9 for more details on the procedure's arguments and result.

If *`read-proc`* or *`optional-peek-proc`* returns a special procedure when called by any reading procedure other than `read`, `read-syntax`, `read-honu`, `read-honu-syntax`, `read-char-or-special`, `peek-char-or-special`, `read-byte-or-special`, or `peek-byte-or-special`, then the `exn:fail:contract` exception is raised.

Examples:

```
;; A port with no input...
;; Easy: (open-input-bytes #"")
;; Hard:
(define /dev/null-in
  (make-input-port 'null
                   (lambda (s) eof)
                   (lambda (skip s progress-evt) eof)
                   void
                   (lambda () never-evt)
                   (lambda (k progress-evt done-evt)
                     (error "no successful peeks!"))))
(read-char /dev/null-in) ; ⇒ eof
(peek-char /dev/null-in) ; ⇒ eof
(read-byte-or-special /dev/null-in)      ; ⇒ eof
```

[3]This non-byte result is *not* intended to return a character or `eof`; in particular, `read-char` raises an exception if it encounters a non-byte from a port.

[4]More precisely, the procedure is used by the default port read handler; see also §11.2.6.

```
(peek-byte-or-special /dev/null-in 100) ; ⇒ eof

;; A port that produces a stream of 1s:
(define infinite-ones
  (make-input-port
   'ones
   (lambda (s)
     (bytes-set! s 0 (char->integer #\1)) 1)
   #f
   void))
(read-string 5 infinite-ones) ; ⇒ "11111"

;; But we can't peek ahead arbitrarily far, because the
;; automatic peek must record the skipped bytes:
(peek-string 5 (expt 2 5000) infinite-ones) ; ⇒ error: out of memory

;; An infinite stream of 1s with a specific peek procedure:
(define infinite-ones
  (let ([one! (lambda (s)
                (bytes-set! s 0 (char->integer #\1)) 1)])
    (make-input-port
     'ones
     one!
     (lambda (s skip progress-evt) (one! s))
     void)))
(read-string 5 infinite-ones) ; ⇒ "11111"

;; Now we can peek ahead arbitrarily far:
(peek-string 5 (expt 2 5000) infinite-ones) ; ⇒ "11111"

;; The port doesn't supply procedures to implement progress events:
(port-provides-progress-evts? infinite-ones) ; ⇒ #f
(port-progress-evt infinite-ones) ; error: no progress events

;; Non-byte port results:
(define infinite-voids
  (make-input-port
   'voids
   (lambda (s) (lambda args 'void))
   (lambda (skip s) (lambda args 'void))
   void))
(read-char infinite-voids) ; ⇒ error: non-char in an unsupported context
(read-char-or-special infinite-voids) ; ⇒ 'void

;; This port produces 0, 1, 2, 0, 1, 2, etc., but it is not
;; thread-safe, because multiple threads might read and change n.
(define mod3-cycle/one-thread
  (let* ([n 2]
         [mod! (lambda (s delta)
                 (bytes-set! s 0 (+ 48 (modulo (+ n delta) 3)))
                 1)])
    (make-input-port
     'mod3-cycle/not-thread-safe
     (lambda (s)
      (set! n (modulo (add1 n) 3))
      (mod! s 0))
    (lambda (s skip)
      (mod! s skip))
    void)))
(read-string 5 mod3-cycle/one-thread) ; ⇒ "01201"
(peek-string 5 (expt 2 5000) mod3-cycle/one-thread) ; ⇒ "20120"

;; Same thing, but thread-safe and kill-safe, and with progress
;; events. Only the server thread touches the stateful part
;; directly. (See the output port examples for a simpler thread-safe
;; example, but this one is more general.)
(define (make-mod3-cycle)
  (define read-req-ch (make-channel))
  (define peek-req-ch (make-channel))
  (define progress-req-ch (make-channel))
  (define commit-req-ch (make-channel))
  (define close-req-ch (make-channel))
  (define closed? #f)
  (define n 0)
  (define progress-sema #f)
  (define (mod! s delta)
    (bytes-set! s 0 (+ 48 (modulo (+ n delta) 3)))
    1)
  ;; ----------------------------------------
  ;; The server has a list of outstanding commit requests,
  ;; and it also must service each port operation (read,
  ;; progress-evt, etc.)
  (define (serve commit-reqs response-evts)
    (apply
     sync
     (handle-evt read-req-ch (handle-read commit-reqs response-evts))
     (handle-evt progress-req-ch (handle-progress commit-reqs response-evts))
     (handle-evt commit-req-ch (add-commit commit-reqs response-evts))
     (handle-evt close-req-ch (handle-close commit-reqs response-evts))
     (append
      (map (make-handle-response commit-reqs response-evts) response-evts)
      (map (make-handle-commit commit-reqs response-evts) commit-reqs))))
  ;; Read/peek request: fill in the string and commit
  (define ((handle-read commit-reqs response-evts) r)
    (let ([s (car r)]
          [skip (cadr r)]
          [ch (caddr r)]
          [nack (cadddr r)]
          [peek? (cddddr r)])
      (unless closed?
        (mod! s skip)
        (unless peek?
          (commit! 1)))
      ;; Add an event to respond:
      (serve commit-reqs
             (cons (choice-evt nack
                               (channel-put-evt ch (if closed? 0 1)))
                   response-evts))))
```

```
;; Progress request: send a peek evt for the current
;; progress-sema
(define ((handle-progress commit-reqs response-evts) r)
  (let ([ch (car r)]
        [nack (cdr r)])
    (unless progress-sema
      (set! progress-sema (make-semaphore (if closed? 1 0))))
    ;; Add an event to respond:
    (serve commit-reqs
           (cons (choice-evt nack
                             (channel-put-evt
                              ch
                              (semaphore-peek-evt progress-sema)))
                 response-evts))))
;; Commit request: add the request to the list
(define ((add-commit commit-reqs response-evts) r)
  (serve (cons r commit-reqs) response-evts))
;; Commit handling: watch out for progress, in which case
;; the response is a commit failure; otherwise, try
;; to sync for a commit. In either event, remove the
;; request from the list
(define ((make-handle-commit commit-reqs response-evts) r)
  (let ([k (car r)]
        [progress-evt (cadr r)]
        [done-evt (caddr r)]
        [ch (cadddr r)]
        [nack (cddddr r)])
    ;; Note: we don't check that k is ≤ the sum of
    ;; previous peeks, because the entire stream is actually
    ;; known, but we could send an exception in that case.
    (choice-evt
     (handle-evt progress-evt
                 (lambda (x)
                   (sync nack (channel-put-evt ch #f))
                   (serve (remq r commit-reqs) response-evts)))
     ;; Only create an event to satisfy done-evt if progress-evt
     ;; isn't already ready.
     ;; Afterward, if progress-evt becomes ready, then this
     ;; event-making function will be called again, because
     ;; the server controls all posts to progress-evt.
     (if (sync/timeout 0 progress-evt)
         never-evt
         (handle-evt done-evt
                     (lambda (v)
                       (commit! k)
                       (sync nack (channel-put-evt ch #t))
                       (serve (remq r commit-reqs) response-evts)))))))
;; Response handling: as soon as the respondee listens,
;; remove the response
(define ((make-handle-response commit-reqs response-evts) evt)
  (handle-evt evt
              (lambda (x)
                (serve commit-reqs
                  (remq evt response-evts)))))
;; Close handling: post the progress sema, if any, and set
;; the closed? flag
(define ((handle-close commit-reqs response-evts) r)
  (let ([ch (car r)]
        [nack (cdr r)])
    (set! closed? #t)
    (when progress-sema
      (semaphore-post progress-sema))
    (serve commit-reqs
           (cons (choice-evt nack
                             (channel-put-evt ch (void)))
                 response-evts))))
;; Helper for reads and post-peek commits:
(define (commit! k)
  (when progress-sema
    (semaphore-post progress-sema)
    (set! progress-sema #f))
  (set! n (+ n k)))
;; Start the server thread:
(define server-thread (thread (lambda () (serve null null))))
;; ----------------------------------------
;; Client-side helpers:
(define (req-evt f)
  (nack-guard-evt
   (lambda (nack)
    ;; Be sure that the server thread is running:
     (thread-resume server-thread (current-thread))
    ;; Create a channel to hold the reply:
     (let ([ch (make-channel)])
       (f ch nack)
       ch))))
(define (read-or-peek-evt s skip peek?)
  (req-evt (lambda (ch nack)
             (channel-put read-req-ch (list* s skip ch nack peek?)))))
;; Make the port:
(make-input-port 'mod3-cycle
                 ;; Each handler for the port just sends
                 ;; a request to the server
                 (lambda (s) (read-or-peek-evt s 0 #f))
                 (lambda (s skip) (read-or-peek-evt s skip #t))
                 (lambda () ; close
                   (sync (req-evt
                          (lambda (ch nack)
                            (channel-put progress-req-ch (list* ch nack))))))
                 (lambda () ; progress-evt
                   (sync (req-evt
                          (lambda (ch nack)
                            (channel-put progress-req-ch (list* ch nack))))))
                 (lambda (k progress-evt done-evt)  ; commit
                   (sync (req-evt
                          (lambda (ch nack)
                            (channel-put commit-req-ch
                                               (list* k progress-evt done-evt ch nack)))))))

(let ([mod3-cycle (make-mod3-cycle)])
(let ([result1 #f]
      [result2 #f])
  (let ([t1 (thread (lambda ()
                      (set! result1 (read-string 5 mod3-cycle))))]
        [t2 (thread (lambda ()
                      (set! result2 (read-string 5 mod3-cycle))))])
    (thread-wait t1)
    (thread-wait t2)
    (string-append result1 "," result2))) ; ⇒ "02120,10201", maybe
(let ([s (make-bytes 1)]
      [progress-evt (port-progress-evt mod3-cycle)])
  (peek-bytes-avail! s 0 progress-evt mod3-cycle) ; ⇒ 1
  s                                               ; ⇒ #"1"
  (port-commit-peeked 1 progress-evt (make-semaphore 1)
                      mod3-cycle)    ; ⇒ #t
  (sync/timeout 0 progress-evt)       ; ⇒ progress-evt
  (peek-bytes-avail! s 0 progress-evt mod3-cycle) ; ⇒ 0
  (port-commit-peeked 1 progress-evt (make-semaphore 1)
                      mod3-cycle))   ; ⇒ #f
(close-input-port mod3-cycle))
```

11.1.7.2 Custom Output

`(make-output-port` *`name-v evt write-proc close-proc`* `[`*`optional-write-special-proc optional-write-evt-proc optional-special-evt-proc optional-location-proc count-lines!-proc init-position optional-buffer-mode-proc`*`])` creates an output port. The port is immediately open for writing. If *`close-proc`* procedure has no side effects, then the port need not be explicitly closed. The port can buffer data within its *`write-proc`* and *`optional-write-special-proc`* procedures.

- *`name-v`* — the name for the output port, which is reported by `object-name` (see §6.2.3).
- *`evt`* — a synchronization event (see §7.7; e.g., a semaphore or another port). The event is used in place of the output port when the port is supplied to synchronization procedures like `sync`. Thus, the event should be unblocked when the port is ready for writing at least one byte without blocking, or ready to make progress in flushing an internal buffer without blocking. The event must not unblock unless the port is ready for writing; otherwise, the guarantees of `sync` will be broken for the output port. Use `always-evt` if writes to the port always succeed without blocking.
- *`write-proc`* — a procedure of five arguments:
  - an immutable byte string containing bytes to write;
  - a non-negative exact integer for a starting offset (inclusive) into the byte string;
  - a non-negative exact integer for an ending offset (exclusive) into the byte string;
  - a boolean; `#f` indicates that the port is allowed to keep the written bytes in a buffer, and that it is allowed to block indefinitely; `#t` indicates that the write should not block, and that the port should attempt to flush its buffer and completely write new bytes instead of buffering them;
  - a boolean; `#t` indicates that if the port blocks for a write, then it should enable breaks while blocking (e.g., using `sync/enable-break`; this argument is always `#f` if the fourth argument is `#t`.

  The procedure returns one of the following:

  - a non-negative exact integer representing the number of bytes written or buffered;

- `#f` if no bytes could be written, perhaps because the internal buffer could not be completely flushed;
- a synchronizable event (see §7.7) that acts like the result of `write-bytes-avail-evt` to complete the write.

Since *`write-proc`* can produce an event, an acceptable implementation of *`write-proc`* is to pass its first three arguments to the port's *`optional-write-evt-proc`*. Some port implementors, however, may choose not to provide *`optional-write-evt-proc`* (perhaps because writes cannot be made atomic), or may implement *`write-proc`* to enable a fast path for non-blocking writes or to enable buffering.

From a user's perspective, the difference between buffered and completely written data is (1) buffered data can be lost in the future due to a failed write, and (2) `flush-output` forces all buffered data to be completely written. Under no circumstances is buffering required.

If the start and end indices are the same, then the fourth argument to *`write-proc`* will be `#f`, and the write request is actually a flush request for the port's buffer (if any), and the result should be `0` for a successful flush (or if there is no buffer).

The result should never be `0` if the start and end indices are different, otherwise the `exn:fail:contract` exception is raised. If a returned integer is larger than the supplied byte-string range, the `exn:fail:contract` exception is raised.

The `#f` result should be avoided, unless the next write attempt is likely to work. Otherwise, if data cannot be written, return an event instead.

An event returned by *`write-proc`* can return `#f` or another event like itself, in contrast to events produced by `write-bytes-avail-evt` or *`optional-write-evt-proc`*. A writing process loops with `sync` until it obtains a non-event result.

The *`write-proc`* procedure is always called with breaks disabled, independent of whether breaks were enabled when the write was requested by a client of the port. If breaks were enabled for a blocking operation, then the fifth argument to *`write-proc`* will be `#t`, which indicates that *`write-proc`* should re-enable breaks while blocking.

If the writing procedure raises an exception, due either to write or commit operations, it must not have committed any bytes (though it may have committed previously buffered bytes).

A port's writing procedure may be called in multiple threads simultaneously (if the port is accessible in multiple threads). The port is responsible for its own internal synchronization. Note that improper implementation of such synchronization mechanisms might cause a non-blocking write procedure to block.

- *`close-proc`* — a procedure of zero arguments that is called to close the port. The port is not considered closed until the closing procedure returns. The port's procedures will never be used again via the port after it is closed. However, the closing procedure can be called simultaneously in multiple threads (if the port is accessible in multiple threads), and it may be called during a call to the other procedures in another thread; in the latter case, any outstanding writes or flushes should be terminated immediately with an error.

- *`optional-write-special-proc`* — either `#f` (the default), or a procedure to handle `write-special` calls for the port. If `#f`, then the port does not support special output, and `port-writes-special?` will return `#f` when applied to the port.

  If a procedure is supplied, it takes three arguments: the special value to write, a boolean that is `#f` if the procedure can buffer the special value and block indefinitely, and a boolean that is `#t` if the procedure should enable breaks while blocking. The result is one of the following:

  - a non-event true value, which indicates that the special is written;
  - `#f` if the special could not be written, perhaps because an internal buffer could not be completely flushed;
  - a synchronizable event (see §7.7) that acts like the result of `write-special-evt` to complete the write.

  Since *`optional-write-special-proc`* can return an event, passing the first argument to an implementation of *`option-write-special-evt-proc`* is acceptable as an *`optional-write-special-proc`*.

As for *`write-proc`*, the `#f` result is discouraged, since it can lead to busy waiting. Also as for *`write-proc`*, an event produced by *`optional-write-special-proc`* is allowed to produce `#f` or another event like itself. The *`optional-write-special-proc`* procedure is always called with breaks disabled, independent of whether breaks were enabled when the write was requested by a client of the port.

- *`optional-write-evt-proc`* — either `#f` (the default) or a procedure of three arguments:
  - an immutable byte string containing bytes to write;
  - a non-negative exact integer for a starting offset (inclusive) into the byte string, and
  - a non-negative exact integer for an ending offset (exclusive) into the byte string.

  The result is a synchronizable event (see §7.7) to act as the result of `write-bytes-avail-evt` for the port (i.e., to complete a write or flush), which becomes available only as data is committed to the port's underlying device, and whose result is the number of bytes written.

  If *`optional-write-evt-proc`* is `#f`, then `port-writes-atomic?` will produce `#f` with applied to the port, and the port will not be a valid argument to procedures such as `write-bytes-avail-evt`.

  Otherwise, an event returned by *`optional-write-evt-proc`* must not cause data to be written to the port unless the event is chosen in a synchronization, and it must write to the port if the event is chosen (i.e., the write must appear atomic with respect to the synchronization).

  If the event's result integer is larger than the supplied byte-string range, the `exn:fail:contract` exception is raised by a wrapper on the event. If the start and end indices are the same (i.e., no bytes are to be written), then the event should produce `0` when the buffer is completely flushed. (If the port has no buffer, then it is effectively always flushed.)

  If the event raises an exception, due either to write or commit operations, it must not have committed any new bytes (though it may have committed previously buffered bytes).

  Naturally, a port's events may be used in multiple threads simultaneously (if the port is accessible in multiple threads). The port is responsible for its own internal synchronization.

- *`optional-write-special-evt-proc`* — either `#f` (the default), or a procedure to handle `write-special-evt` calls for the port. This argument must be `#f` if either *`optional-write-special-proc`* or *`optional-write-evt-proc`* is `#f`, and it must be a procedure if both of those arguments are procedures.

  If it is a procedure, it takes one argument: the special value to write. The resulting event (with its constraints) is analogous to the result of *`optional-write-evt-proc`*.

  If the event raises an exception, due either to write or commit operations, it must not have committed the special value (though it may have committed previously buffered bytes and values).

- *`optional-location-proc`* — either `#f` (the default), or a procedure that takes no arguments and returns three values: the line number for the next item written to the port's stream (a positive number or `#f`), the column number for the next item written to port's stream (a non-negative number or `#f`), and the position for the next item written to port's stream (a positive number or `#f`). See also §11.2.1.1.

  This procedure is only called if line counting is enabled for the port via `port-count-lines!` (in which case *`count-lines!-proc`* is called).

- *`count-lines!-proc`* — a procedure of no arguments that is called if and when line counting is enabled for the port. The default procedure is `void`.

- *`init-position`* — an exact, positive integer that determines the position of the port's first output item, used when line counting is *not* enabled for the port. The default is `1`.

- *`optional-buffer-mode-proc`* — either `#f` (the default) or a procedure that accepts zero or one arguments. If *`optional-buffer-mode-proc`* is `#f`, then the resulting port does not support a buffer-mode setting. Otherwise, the procedure is called with one symbol argument (`'block`, `'line`, or `'none`) to set the buffer mode, and it is called with zero arguments to get the current buffer mode. In the latter case, the result must be `'block`, `'line`, `'none`, or `#f` (unknown). See §11.1.6 for more information on buffer modes.

Examples:

```
;; A port that writes anything to nowhere:
(define /dev/null-out
  (make-output-port
   'null
   always-evt
   (lambda (s start end non-block? breakable?) (- end start))
   void
   (lambda (special non-block? breakable?) #t)
   (lambda (s start end) (wrap-evt
                          always-evt
                          (lambda (x)
                            (- end start))))
   (lambda (special) always-evt)))
(display "hello" /dev/null-out)            ; ⇒void
(write-bytes-avail #"hello" /dev/null-out) ; ⇒5
(write-special 'hello /dev/null-out)       ; ⇒#t
(sync (write-bytes-avail-evt #"hello" /dev/null-out)) ; ⇒5

;; A part that accumulates bytes as characters in a list,
;; but not in a thread-safe way:
(define accum-list null)
(define accumulator/not-thread-safe
  (make-output-port
   'accum/not-thread-safe
   always-evt
   (lambda (s start end non-block? breakable?)
     (set! accum-list
           (append accum-list
                   (map integer->char
                        (bytes->list (subbytes s start end)))))
     (- end start))
   void))
(display "hello" accumulator/not-thread-safe)
accum-list ; ⇒ '(#\h #\e #\l #\l #\o)

;; Same as before, but with simple thread-safety:
(define accum-list null)
(define accumulator
  (let* ([lock (make-semaphore 1)]
         [lock-peek-evt (semaphore-peek-evt lock)])
    (make-output-port
     'accum
     lock-peek-evt
     (lambda (s start end non-block? breakable?)
       (if (semaphore-try-wait? lock)
           (begin
             (set! accum-list
                   (append accum-list
                           (map integer->char
                                (bytes->list (subbytes s start end)))))
             (semaphore-post lock)
             (- end start))
```

```
          ;; Cheap strategy: block until the list is unlocked,
          ;; then return 0, so we get called again
          (wrap-evt
           lock-peek
           (lambda (x) 0))))
    void)))
(display "hello" accumulator)
accum-list ; ⇒ '(#\h #\e #\l #\l #\o)
```

```
;; A port that transforms data before sending it on
;; to another port. Atomic writes exploit the
;; underlying port's ability for atomic writes.
(define (make-latin-1-capitalize port)
  (define (byte-upcase s start end)
    (list->bytes
     (map (lambda (b) (char->integer
                       (char-upcase
                        (integer->char b))))
          (bytes->list (subbytes s start end)))))
  (make-output-port
   'byte-upcase
   ;; This port is ready when the original is ready:
   port
   ;; Writing procedure:
   (lambda (s start end non-block? breakable?)
     (let ([s (byte-upcase s start end)])
       (if non-block?
           (write-bytes-avail* s port)
           (begin
             (display s port)
             (bytes-length s)))))
   ;; Close procedure --- close original port:
   (lambda () (close-output-port port))
   #f
   ;; Write event:
   (and (port-writes-atomic? port)
        (lambda (s start end)
          (write-bytes-avail-evt (byte-upcase s start end) port)))))
(define orig-port (open-output-string))
(define cap-port (make-latin-1-capitalize orig-port))
(display "Hello" cap-port)
(get-output-string orig-port) ; ⇒ "HELLO"
(sync (write-bytes-avail-evt #"Bye" cap-port)) ; ⇒ 3
(get-output-string orig-port) ; ⇒ "HELLOBYE"
```

11.2 Reading and Writing

MzScheme's support for reading and writing includes many extensions compared to R^5RS, both at the level of individual bytes and characters and at the level of S-expressions.

11.2.1 Reading Bytes, Characters, and Strings

In addition to the standard reading procedures, MzScheme provides byte-reading procedure, block-reading procedures such as `read-line`, and more.

- `(read-line [`*`input-port mode-symbol`*`])` returns a string containing the next line of bytes from *`input-port`*. If *`input-port`* is omitted, the current input port is used.

  Characters are read from *`input-port`* until a line separator or an end-of-file is read. The line separator is not included in the result string (but it is removed from the port's stream). If no characters are read before an end-of-file is encountered, `eof` is returned.

  The *`mode-symbol`* argument determines the line separator(s). It must be one of the following symbols:

  - `'linefeed` breaks lines on linefeed characters; this is the default.
  - `'return` breaks lines on return characters.
  - `'return-linefeed` breaks lines on return-linefeed combinations. If a return character is not followed by a linefeed character, it is included in the result string; similarly, a linefeed that is not preceded by a return is included in the result string.
  - `'any` breaks lines on any of a return character, linefeed character, or return-linefeed combination. If a return character is followed by a linefeed character, the two are treated as a combination.
  - `'any-one` breaks lines on either a return or linefeed character, without recognizing return-linefeed combinations.

  Return and linefeed characters are detected after the conversions that are automatically performed when reading a file in text mode. For example, reading a file in text mode under Windows automatically changes return-linefeed combinations to a linefeed. Thus, when a file is opened in text mode, `'linefeed` is usually the appropriate `read-line` mode.

- `(read-bytes-line [`*`input-port mode-symbol`*`])` is analogous to `read-line`, but it reads bytes and produces a byte string.

- `(read-string `*`k`*` [`*`input-port`*`])` returns a string containing the next *`k`* characters from *`input-port`*. The default value of *`input-port`* is the current input port.

  If *`k`* is `0`, then the empty string is returned. Otherwise, if fewer than *`k`* characters are available before an end-of-file is encountered, then the returned string will contain only those characters before the end-of-file (i.e., the returned string's length will be less than *`k`*). [5] If no characters are available before an end-of-file, then `eof` is returned.

  If an error occurs during reading, some characters may be lost (i.e., if `read-string` successfully reads some characters before encountering an error, the characters are dropped.)

- `(read-bytes `*`k`*` [`*`input-port`*`])` is analogous to `read-string`, but it reads bytes and produces a byte string.

- `(read-string! `*`string`*` [`*`input-port start-k end-k`*`])` reads characters from *`input-port`* like `read-string`, but puts them into *`string`* starting from index *`start-k`* (inclusive) up to *`end-k`* (exclusive). The default value of *`input-port`* is the current input port. The default value of *`start-k`* is `0`. The default value of *`end-k`* is the length of the *`string`*. Like `substring`, the `exn:fail:contract` exception is raised if *`start-k`* or *`end-k`* is out-of-range for *`string`*.

  If the difference between *`start-k`* and *`end-k`* is `0`, then `0` is returned and *`bytes`* is not modified. If no bytes are available before an end-of-file, then `eof` is returned. Otherwise, the return value is the number of bytes read. If m bytes are read and $m <$ *`end-k`* $-$ *`start-k`*, then *`bytes`* is not modified at indices *`start-k`* $+ m$ though *`end-k`*.

- `(read-bytes! `*`string`*` [`*`input-port start-k end-k`*`])` is analogous to `read-string!`, but it reads bytes and puts them into a byte string.

[5] A temporary string of size *`k`* is allocated while reading the input, even if the size of the result is less than *`k`* characters.

- `(read-bytes-avail!` *`bytes`* [*`input-port start-k end-k`*]`)` is like `read-bytes!`, but returns without blocking after reading immediately-available bytes, and it may return a procedure for a "special" result. The `read-bytes-avail!` procedure blocks only if no bytes (or specials) are yet available. Also unlike `read-bytes!`, `read-bytes-avail!` never drops bytes; if `read-bytes-avail!` successfully reads some bytes and then encounters an error, it suppresses the error (treating it roughly like an end-of-file) and returns the read bytes. (The error will be triggered by future reads.) If an error is encountered before any bytes have been read, an exception is raised.

  When *`input-port`* produces a special value, as described in §11.1.7, the result is a procedure of four arguments. The four arguments correspond to the location of the special value within the port, as described in §11.1.7. If the procedure is called more than once with valid arguments, the `exn:fail:contract` exception is raised. If *`read-bytes-avail`* returns a special-producing procedure, then it does not place characters in *`bytes`*. Similarly, *`read-bytes-avail`* places only as many bytes into *`bytes`* as are available before a special value in the port's stream.

- `(read-bytes-avail!*` *`bytes`* [*`input-port start-k end-k`*]`)` is like `read-bytes-avail!`, except that it returns `0` immediately if no bytes (or specials) are available for reading and the end-of-file is not reached.

- `(read-bytes-avail!/enable-break` *`bytes`* [*`input-port start-k end-k`*]`)` is like `read-bytes-avail!`, except that breaks are enabled during the read (see also §6.7). If breaking is disabled when `read-bytes-avail!/enable-break` is called, and if the `exn:break` exception is raised as a result of the call, then no bytes will have been read from *`input-port`*.

- `(peek-string` *`k skip-k`* [*`input-port`*]`)` is similar to `read-string`, except that the returned characters are preserved in the port for future reads. (More precisely, undecoded bytes are left for future reads.) The *`skip-k`* argument indicates a number of bytes (*not* characters) in the input stream to skip before collecting characters to return; thus, in total, the next *`skip-k`* bytes plus *`k`* characters are inspected.

  For most kinds of ports, inspecting *`skip-k`* bytes and *`k`* characters requires at least *`skip-k`*+*`k`* bytes of memory overhead associated with the port, at least until the bytes/characters are read. No such overhead is required when peeking into a string port (see §11.1.5), a pipe port (see §11.1.4), or a custom port with a specific peek procedure (depending on how the peek procedure is implemented; see §11.1.7).

  If a port produces `eof` mid-stream, peek skips beyond the `eof` always produce `eof` until the `eof` is read.

- `(peek-bytes` *`k skip-k`* [*`input-port`*]`)` is analogous to `peek-string`, but it peeks bytes and produces a byte string.

- `(peek-string!` *`string skip-k`* [*`input-port start-k end-k`*]`)` is like `read-string!`, but for peeking, and with a *`skip-k`* argument like `peek-string`.

- `(peek-bytes!` *`bytes skip-k`* [*`input-port start-k end-k`*]`)` is analogous to `peek-string!`, but it peeks bytes and puts them into a byte string.

- `(peek-bytes-avail!` *`bytes skip-k`* [*`progress-evt input-port start-k end-k`*]`)` is like `read-bytes-avail!`, but for peeking, and with two extra arguments. The *`skip-k`* argument is as in `peek-bytes`. The *`progress-evt`* argument must be either `#f` (the default) or an event produced by `port-progress-evt` for *`input-port`*.

  To peek, `peek-bytes-avail!` blocks until finding an end-of-file, at least one byte (or special) past the skipped bytes, or until a non-`#f` *`progress-evt`* becomes ready. Furthermore, if *`progress-evt`* is ready before bytes are peeked, no bytes are peeked or skipped, and *`progress-evt`* may cut short the skipping process if it becomes available during the peek attempt.

  The result of `peek-bytes-avail!` is `0` only in the case that *`progress-evt`* becomes ready before bytes are peeked.

- `(peek-bytes-avail!*` *`bytes skip-k`* [*`progress-evt input-port start-k end-k`*]`)` is like `read-bytes-avail!*`, but for peeking, and with *`skip-k`* and *`progress-evt`* arguments like `peek-bytes-avail!`. Since this procedure never blocks, it may return before even *`skip-k`* bytes are available from the port.

- `(peek-bytes-avail!/enable-break` *`bytes skip-k`* [*`progress-evt input-port start-k end-k`*]`)` is the peeking version of `read-bytes-avail!/enable-break`, with *`skip-k`* and *`progress-evt`* arguments like `peek-bytes-avail!`.

- `(read-byte` [*`input-port`*]`)` is analogous to `read-char`, but it reads and returns a byte (or `eof`) instead of a character.

- `(read-char-or-special` [*`input-port`*]`)` is the same as `read-char`, except that if the input port returns a non-byte value (through a value-generating procedure in a custom port; see §11.1.7 and §11.2.9.1 for details), the non-byte value is returned.

- `(read-byte-or-special` [*`input-port`*]`)` is analogous to `read-char-or-special`, but it reads and returns a byte instead of a character.

- `(peek-char` [*`input-port skip-k`*]`)` extends the standard `peek-char` with an optional argument (defaulting to `0`) that represents the number of bytes (not characters) to skip.

- `(peek-byte` [*`input-port skip-k`*]`)` is analogous to `peek-char`, but it reads and returns a byte instead of a character.

- `(peek-char-or-special` [*`input-port skip-k`*]`)` is the same as `peek-char`, except that if the input port returns a non-byte value after *`skip-k`* byte positions, it is returned.

- `(peek-byte-or-special` [*`input-port skip-k progress-evt`*]`)` is analogous to `peek-char-or-special`, but it reads and returns a byte instead of a character, and it supports a *`progress-evt`* argument (which is `#f` by default) like `peek-bytes-avail!`.

- `(port-progress-evt` [*`input-port`*]`)` returns an event that becomes ready after any subsequent read from *`input-port`*, or after *`input-port`* is closed. After the event becomes ready, it remains ready. If progress events are unavailable for *`input-port`* (as reported by `port-provides-progress-evts?`) the `exn:fail:contract` exception is raised.

- `(port-provides-progress-evts?` *`input-port`*`)` returns `#t` if `port-progress-evt` can return an event for *`input-port`*. All built-in kinds of ports support progress events, but ports created with `make-input-port` (see §11.1.7) may not.

- `(port-commit-peeked` *`k progress-evt evt`* [*`input-port`*]`)` attempts to commit as read the first *`k`* previously peeked bytes, non-byte specials, and `eof`s from *`input-port`*, or the first `eof` or special value peeked from *`input-port`*.[6] The read commits only if *`progress-evt`* does not become ready first (i.e., if no other process reads from *`input-port`* first), and only if *`evt`* is chosen by a `sync` within `port-commit-peeked` (in which case the event result is ignored); the *`evt`* must be either a channel-put event, channel, semaphore, semaphore-peek event, always event, or never event. Suspending the thread that calls `port-commit-peeked` may or may not prevent the commit from proceeding. The result from `port-commit-peeked` is `#t` if data is committed, and `#f` otherwise.

  If no data has been peeked from *`input-port`* and *`progress-evt`* is not ready, then `exn:fail:contract` exception is raised. If fewer than *`k`* items have been peeked at the current start of *`input-port`*'s stream, then only the peeked items are committed as read. If *`input-port`*'s stream currently starts at an `eof` or a non-byte special value, then only the `eof` or special value is committed as read.

  If *`progress-evt`* is not a result of `port-progress-evt` applied to *`input-port`*, then `exn:fail:contract` exception is raised.

[6] Only mid-stream `eof`s can be committed. A `eof` when the port is exhausted does not correspond to data in the stream.

11.2.1.1 Counting Positions, Lines, and Columns

By default, MzScheme keeps track of the *position* in a port as the number of bytes that have been read from or written to any port (independent of the read/write position, which is accessed or changed with `file-position`). Optionally, however, MzScheme can track the position in terms of characters (after UTF-8 decoding), instead of bytes, and it can track *line locations* and *column locations*; this optional tracking must be specifically enabled for a port via `port-count-lines!` or the `port-count-lines-enabled` parameter (see §7.9.1.2). Position, line, and column locations for a port are used by `read-syntax` (see §12.2 for more information) and `read-honu-syntax`. Position and line locations are numbered from 1; column locations are numbered from 0.

- `(port-count-lines!` *`port`*`)` turns on line and column counting for a port. Counting can be turned on at any time, though generally it is turned on before any data is read from or written to a port. When a port is created, if the value of the `port-count-lines-enabled` parameter is true (see §7.9.1.2), then line counting is automatically enabled for the port. Line counting cannot be disabled for a port after it is enabled.

When counting lines, MzScheme treats linefeed, return, and return-linefeed combinations as a line terminator and as a single position (on all platforms). Each tab advances the column count to one before the next multiple of 8. When a sequence of bytes in the range 128 to 253 forms a UTF-8 encoding of a character, the position/column is incremented is incremented once for each byte, and then decremented appropriately when a complete encoding sequence is discovered. See also §11.1 for more information on UTF-8 decoding for ports.

A position is known for any port as long as its value can be expressed as a fixnum (which is more than enough tracking for realistic applications in, say, syntax-error reporting). If the position for a port exceeds the value of the largest fixnum, then the position for the port becomes unknown, and line and column tacking is disabled. Return-linefeed combinations are treated as a single character position only when line and column counting is enabled.

- `(port-next-location` *`port`*`)` returns three values: a positive exact integer or `#f` for the line number of the next read/written item, a non-negative exact integer or `#f` for the next item's column, and a positive exact integer or `#f` for the next item's position. The next column and position normally increases as bytes are read from or written to the port, but if line/character counting is enabled for *`port`*, the column and position results can decrease after reading or writing a byte that ends a UTF-8 encoding sequence.

Certain kinds of exceptions (see §6.1) encapsulate source-location information using a `srcloc` structure, which has five fields:

- `source` — An arbitrary value identifying the source, often a path (see §11.3.1).
- `line` — The line number, a positive exact integer (counts from 1) or `#f` (unknown).
- `column` — The column number, a non-negative exact integer (counts from 0) or `#f` (unknown).
- `position` — The starting position, a positive exact integer (counts from 1) or `#f` (unknown).
- `span` — The number of covered positions, a non-negative exact integer (counts from 0) or `#f` (unknown).

The fields of a `srcloc` structure are immutable, so no field-mutator procedures are defined for `srcloc`. The `srcloc` structure type is transparent to all inspectors (see §4.5).

11.2.2 Writing Bytes, Characters, and Strings

In addition to the standard printing procedures, MzScheme provides byte-writing procedures, block-writing procedures such as `write-string`, and more.

- `(write-string` *`string`* [*`output-port start-k end-k`*]`)` write characters to *`output-port`* from *`string`* starting from index *`start-k`* (inclusive) up to *`end-k`* (exclusive). The default value of *`output-port`* is the current output port. The default value of *`start-k`* is `0`. The default value of *`end-k`* is the length of the *`string`*. Like `substring`, the `exn:fail:contract` exception is raised if *`start-k`* or *`end-k`* is out-of-range for *`string`*.

  The result is the number of characters written to *`output-port`*, which is always `(−` *`end-k start-k`*`)`.

- `(write-bytes` *`bytes`* [*`output-port start-k end-k`*]`)` is analogous to *`write-string`*, but it writes a byte string.

- `(write-bytes-avail` *`bytes`* [*`output-port start-k end-k`*]`)` is like `write-bytes`, but it returns without blocking after writing as many bytes as it can immediately flush. It blocks only if no bytes can be flushed immediately. The result is the number of bytes written and flushed to *`output-port`*; if *`start-k`* is the same as *`end-k`*, then the result can be `0` (indicating a successful flush of any buffered data), otherwise the result is at least `1` but possibly less than `(−` *`end-k start-k`*`)`.

  The *`write-bytes-avail`* procedure never drops bytes; if *`write-bytes-avail`* successfully writes some bytes and then encounters an error, it suppresses the error and returns the number of written bytes. (The error will be triggered by future writes.) If an error is encountered before any bytes have been written, an exception is raised.

- `(write-bytes-avail*` *`bytes`* [*`output-port start-k end-k`*]`)` is like *`write-bytes-avail`*, except that it never blocks, it returns `#f` if the port contains buffered data that cannot be written immediately, and it returns `0` if the port's internal buffer (if any) is flushed but no additional bytes can be written immediately.

- `(write-bytes-avail/enable-break` *`bytes`* [*`input-port start-k end-k`*]`)` is like *`write-bytes-avail`*, except that breaks are enabled during the write. The procedure provides a guarantee about the interaction of writing and breaks: if breaking is disabled when *`write-bytes-avail/enable-break`* is called, and if the `exn:break` exception is raised as a result of the call, then no bytes will have been written to *`output-port`*. See also §6.7.

- `(write-byte` *`byte`* [*`output-port`*]`)` is analogous to `write-char`, but for writing a byte instead of a character.

- `(write-special` *`v`* [*`output-port`*]`)` writes *`v`* directly to *`output-port`* if it supports special writes, or raises `exn:fail:contract` if the port does not support special write. The result is always `#t`, indicating that the write succeeded.

- `(write-special-avail*` *`v`* [*`output-port`*]`)` is like `write-special`, but without blocking. If *`v`* cannot be written immediately, the result is `#f` without writing *`v`*, otherwise the result is `#t` and *`v`* is written.

- `(write-bytes-avail-evt` *`bytes`* [*`output-port start-k end-k`*]`)` is similar to `write-bytes-avail`, but instead of writing bytes immediately, it returns a synchronizable event (see §7.7). The *`output-port`* must support atomic writes, as indicated by `port-writes-atomic?`.

  Synchronizing on the object starts a write from *`bytes`*, and the event becomes ready when bytes are written (unbuffered) to the port. If *`start-k`* and *`end-k`* are the same, then the synchronization result is `0` when the port's internal buffer (if any) is flushed, otherwise the result is a positive exact integer. If the event is not selected in a synchronization, then no bytes will have been written to *`output-port`*.

- `(write-special-evt` *`v`* [*`output-port`*]`)` is similar to `write-special`, but instead of writing the special value immediately, it returns a synchronizable event (see §7.7). The *`output-port`* must support atomic writes, as indicated by `port-writes-atomic?`.

  Synchronizing on the object starts a write of the special value, and the event becomes ready when the value is written (unbuffered) to the port. If the event is not selected in a synchronization, then no value will have been written to *`output-port`*.

- `(port-writes-atomic? output-port)` returns `#t` if `write-bytes-avail/enable-break` can provide an exclusive-or guarantee (break or write, but not both) for *output-port*, and if the port can be used with procedures like `write-bytes-avail-evt`. MzScheme's file-stream ports, pipes, string ports, and TCP ports all support atomic writes; ports created with `make-output-port` (see §11.1.7) may support atomic writes.

- `(port-writes-special? output-port)` returns `#t` if procedures like `write-special` can write arbitrary values to the port. MzScheme's file-stream ports, pipes, string ports, and TCP ports all reject special values, but ports created with `make-output-port` (see §11.1.7) may support them.

11.2.3 Writing Structured Data

The `print` procedure is used to print Scheme values in a context where a programmer expects to see a value:

- `(print v [output-port])` outputs *v* to *output-port*. The default value of *output-port* is the current output port.

The rationale for providing `print` is that `display` and `write` both have standard output conventions, and this standardization restricts the ways that an environment can change the behavior of these procedures. No output conventions should be assumed for `print` so that environments are free to modify the actual output generated by `print` in any way. Unlike the port display and write handlers, a global port print handler can be installed through the `global-port-print-handler` parameter (see §7.9.1.2).

The `fprintf`, `printf`, and `format` procedures create formatted output:

- `(fprintf output-port format-string v ···)` prints formatted output to *output-port*, where *format-string* is a string that is printed; *format-string* can contain special formatting tags:
  - ~n or ~% prints a newline
  - ~a or ~A `display`s the next argument among the *v*s
  - ~s or ~S `write`s the next argument among the *v*s
  - ~v or ~V `print`s the next argument among the *v*s
  - ~e or ~E outputs the next argument among the *v*s using the current error value conversion handler (see §7.9.1.7) and current error printing width
  - ~c or ~C `write-char`s the next argument in *v*s; if the next argument is not a character, the `exn:fail:contract` exception is raised
  - ~b or ~B prints the next argument among the *v*s in binary; if the next argument is not an exact number, the `exn:fail:contract` exception is raised
  - ~o or ~O prints the next argument among the *v*s in octal; if the next argument is not an exact number, the `exn:fail:contract` exception is raised
  - ~x or ~X prints the next argument among the *v*s in hexadecimal; if the next argument is not an exact number, the `exn:fail:contract` exception is raised
  - ~~ prints a tilde (~)
  - ~*w*, where *w* is a whitespace character, skips characters in *format-string* until a non-whitespace character is encountered or until a second end-of-line is encountered (whichever happens first). An end-of-line is either `#\return`, `#\newline`, or `#\return` followed immediately by `#\newline` (on all platforms).

  The return value is void.

- `(printf format-string v ···)` same as `fprintf` with the current output port.

- `(format format-string v ···)` same as `fprintf` with a string output port where the final string is returned as the result.

When an illegal format string is supplied to one of these procedures, the `exn:fail:contract` exception is raised. When the format string requires more additional arguments than are supplied, the `exn:fail:contract` exception is raised. When more additional arguments are supplied than are used by the format string, the `exn:fail:contract` exception is raised.

For example,

```
(fprintf port "~a as a string is ~s.~n" '(3 4) "(3 4)")
```

prints this message to *port*:[7]

```
(3 4) as a string is "(3 4)".
```

followed by a newline.

11.2.4 Default Reader

MzScheme's input parser obeys the following non-standard rules. See also §11.2.8 for information on configuring the input parser through a readtable.

- Square brackets ("[" and "]") and curly braces ("{" and "}") can be used in place of parentheses. An open square bracket must be closed by a closing square bracket and an open curly brace must be closed by a closing curly brace. Whether square brackets are treated as parentheses is controlled by the `read-square-bracket-as-paren` parameter (see §7.9.1.3). Similarly, the parsing of curly braces is controlled with the `read-curly-brace-as-paren` parameter. When square brackets and curly braces are not treated as parentheses, they are disallowed as input. By default, square brackets and curly braces are treated as parentheses.
- Vector constants can be unquoted, and a vector size can be specified with a decimal integer between the `#` and opening parenthesis. If the specified size is larger than the number of vector elements that are provided, the last specified element is used to fill the remaining vector slots. For example, `#4(1 2)` is equivalent to `#(1 2 2 2)`. If no vector elements are specified, the vector is filled with `0`. If a vector size is provided and it is smaller than the number of elements provided, the `exn:fail:read` exception is raised.
- Boxed constants can be created using `#&`. The datum following `#&` is treated as a quoted constant and put into the new box. (Space and comments following the `#&` are ignored.) Box reading is controlled with the `read-accept-box` boolean parameter (see §7.9.1.3). Box reading is enabled by default. When box reading is disabled and `#&` is provided as input, the `exn:fail:read` exception is raised.
- Expressions beginning with `#'` are wrapped with `syntax` in the same way that expressions starting with `'` are wrapped with `quote`. Similarly, `` #` `` generates `quasisyntax`, `#,` generates `unsyntax`, and `#,@` generates `unsyntax-splicing`. See also §12.2.1.2.
- The following character constants are recognized:
  - `#\nul` or `#\null` (ASCII 0)
  - `#\backspace` (ASCII 8)
  - `#\tab` (ASCII 9)
  - `#\newline` or `#\linefeed` (ASCII 10)
  - `#\vtab` (ASCII 11)
  - `#\page` (ASCII 12)
  - `#\return` (ASCII 13)
  - `#\space` (ASCII 32)
  - `#\rubout` (ASCII 127)

[7] Assuming that the current port display and write handlers are the default ones; see §11.2.7 for more information.

Whenever `#\` is followed by at least two alphabetic characters, characters are read from the input port until the next non-alphabetic character is returned. If the resulting string of letters does not match one of the above constants (case-insensitively), the `exn:fail:read` exception is raised.

Character constants can also be specified through direct Unicode values in octal notation (up to 255): `#\`$n_1 n_2 n_3$ where n_1 is in the range [`0`, `3`] and n_2 and n_3 are in the range [`0`, `7`]. Whenever `#\` is followed by at least two characters in the range [`0`, `7`], the next character must also be in this range, and the resulting octal number must be in the range 000_8 to 377_8.

Finally, character constants can be specified through direct Unicode values in hexadecimal notation: `#\u`$n_1 \ldots n_k$ or `#\U`$n_1 \ldots n_k$, where each n_i is a hexadecimal digit (0-9, a-f, or A-F), and k is no more than 4 for `#\u` or 6 for `#\U`. Whenever `#\` is followed by a *u* or *U* and one hexadecimal digit, the character constant is terminated by either the first non-hexadecimal character in the stream, or the fourth/sixth hexadecimal character, whichever comes first. The resulting hexadecimal number must be a valid argument to `integer->char`, otherwise the `exn:fail:read` exception is raised.

Unless otherwise specified above, character-constants are terminated after the character following `#\`. For example, if `#\` is followed by an alphabetic character other than `u` and then a non-alphabetic character, then the character constant is terminated. If `#\` is followed by a `8` or `9`, then the constant is terminated. If `#\` is followed by a non-alphabetic, non-decimal-digit character then the constant is terminated.

- Within string constants, the following escape sequences are recognized in addition to `\"` and `\\`:
  - `\a`: alarm (ASCII 7)
  - `\b`: backspace (ASCII 8)
  - `\t`: tab (ASCII 9)
  - `\n`: linefeed (ASCII 10)
  - `\v`: vertical tab (ASCII 11)
  - `\f`: formfeed (ASCII 12)
  - `\r`: return (ASCII 13)
  - `\e`: escape (ASCII 27)
  - `\'`: quote (i.e., the backslash has no effect)
  - `\`*o*, `\`*oo*, or `\`*ooo*: Unicode for octal *o*, *oo*, or *ooo*, where each *o* is `0`, `1`, `2`, `3`, `4`, `5`, `6`, or `7`. The `\`*ooo* form takes precedence over the `\`*oo* form, and `\`*oo* takes precedence over `\`*o*.
  - `\x`*h* or `\x`*hh*: Unicode for hexadecimal *h* or *hh*, where each *h* is `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `a`, `A`, `b`, `B`, `c`, `C`, `d`, `D`, `e`, `E`, `f`, or `F`. The `\x`*hh* form takes precedence over the `\x`*h* form.
  - `\u`*h*, `\u`*hh*, `\u`*hhh*, or `\u`*hhhh*: like `\x`, but with up to four hexadecimal digits (longer sequences take precedence). The resulting hexadecimal number must be a valid argument to `integer->char`, otherwise the `exn:fail:read` exception is raised.
  - `\U`*h*, `\U`*hh*, `\U`*hhh*, `\U`*hhhh*, `\U`*hhhhh*, `\U`*hhhhhh*, `\U`*hhhhhhh*, or `\U`*hhhhhhhh*: like `\x`, but with up to eight hexadecimal digits (longer sequences take precedence). The resulting hexadecimal number must be a valid argument to `integer->char`, otherwise the `exn:fail:read` exception is raised.

  Furthermore, a backslash followed by a linefeed, carriage return or return-linefeed combination is elided, allowing string constants to span lines. Any other use of backslash within a string constant is an error.

- A string constant preceded by `#` is a byte-string constant. Byte string constants support the same escape sequences as character strings except `\u` and `\U`.

- The sequence `#<<` starts a *here string*. The characters following `#<<` until a newline character define a terminator for the string. The content of the string includes all characters between the `#<<` line and a line whose only content is the specified terminator. More precisely, the content of the string starts after a newline following `#<<`, and it ends before a newline that is followed by the terminator, where the terminator is itself followed by either a newline or end-of-file. No escape sequences are recognized between the starting and terminating lines; all characters are included in the string (and terminator) literally. A return character is not treated as a line separator in this context. If no characters appear between `#<<` and a newline or end-of-file, or if an end-of-file is encountered before a terminating line, the `exn:fail:read` exception is raised.

- The syntax for numbers is extended as described in §3.3. Numbers containing a decimal point or exponent (e.g., `1.3`, `2e78`) are normally read as inexact. If the `read-decimal-as-inexact` parameter is set to `#f`, then such numbers are instead read as exact. The parameter does not affect the parsing of numbers with an explicit exactness tag (`#e` or `#i`).

- A parenthesized sequence containing two delimited dots ("`.`") triggers infix parsing. A single *`datum`* must appear between the dots, and one or more *`datum`*s must appear before the first dot and after the last dot:

  `(`*`left-datum`* $\cdots^1$ `.` *`first-datum`* `.` *`right-datum`* $\cdots^1$`)`

  The resulting list consists of the *`datum`* between the dots, followed by the remaining *`datum`*s in order:

  `(`*`first-datum left-datum`* $\cdots^1$ *`right-datum`* $\cdots^1$`)`

  Consequently, the input expression `(1 . < . 2)` produces `#t`, and `(1 2 . + . 3 4 5)` produces `15`.

- When the `read-accept-dot` parameter is set to `#f`, then a delimited dot ("`.`") is disallowed in input. When the `read-accept-quasiquote` parameter is set to `#f`, then a backquote or comma is disallowed in input. These modes simplify Scheme's input model for students.

- MzScheme's identifier and symbol syntax is considerably more liberal than the syntax specified by *R⁵RS*. When input is scanned for tokens, the following characters delimit an identifier in addition to whitespace:

  ```
  " , ' ` ; ( ) [ ] { }
  ```

  In addition, an identifier cannot start with a hash mark ("`#`") unless the hash mark is immediately followed by a percent sign ("`%`"). The only other special characters are backslash ("\") and quoting vertical bars ("|"); any other character is used as part of an identifier.

  Symbols containing special characters (including delimiters) are expressed using an escaping backslash ("\") or quoting vertical bars ("|"):

  - A backslash preceding any character includes that character in the symbol literally; double backslashes produce a single backslash in the symbol.
  - Characters between a pair of vertical bars are included in the symbol literally. Quoting bars can be used for any part of a symbol, or the whole symbol can be quoted. Backslashes and quoting bars can be mixed within a symbol, but a backslash is *not* a special character within a pair of quoting bars.

  Characters quoted with a backslash or a vertical bar always preserve their case, even when identifiers are read case-insensitively.

  An input token constructed in this way is an identifier when it is not a numerical constant (following the extended number syntax described in §3.3). A token containing a backslash or vertical bars is never treated as a numerical constant.

  Examples:

  - `(quote a\(b)` produces the same symbol as `(string->symbol "a(b")`.
  - `(quote A\B)` produces the same symbol as `(string->symbol "aB")` when identifiers are read without case-sensitivity.
  - `(quote a\ b)`, `(quote |a b|)`, and `(quote a| |b)` all produce the same symbol as `(string->symbol "a b")`.
  - `(quote |a||b|)` is the same as `(quote |ab|)`, which produces the same symbol as `(string->symbol "ab")`.
  - `(quote 10)` is the number 10, but `(quote |10|)` produces the same symbol as `(string->symbol "10")`.

  Whether a vertical bar is used as a special or normal symbol character is controlled with the `read-accept-bar-quote` boolean parameter (see §7.9.1.3). Vertical bar quotes are enabled by default. Quoting backslashes cannot be disabled.

- By default, symbols are read case-sensitively. Case sensitivity for reading can be controlled in three ways:
  - Quoting part of a symbol with an escaping backslash ("\") or quoting vertical bar ("|") always preserves the case of the quoted portion, as described above.
  - The sequence `#cs` can be used as a prefix for any expression to make reading symbols within the expression case-sensitive. A `#ci` prefix similarly makes reading symbols in an expression case-insensitive. Whitespace can appear between a `#cs` or `#ci` prefix and its expression, and prefixes can be nested. Backslash and vertical-bar quotes override a `#ci` prefix.
  - When the `read-case-sensitive` parameter (see §7.9.1.3) is set to `#t`, then case is preserved when reading symbols. The default is `#t`, and it is set to `#t` while loading a module (see §5.8). A `#cs` or `#ci` prefix overrides the parameter setting, as does backslash or vertical-bar quoting.

  Symbol case conversions are *not* sensitive to the current locale (see §1.2.2).

- A symbol-like expression that starts with an unquoted hash and colon ("#:") is parsed as a keyword constant. After the leading colon, backslashes, vertical bars, and case sensitivity are handled as for symbols, except that a keyword expression can never be interpreted as a number.

- Expressions of the form `#rx`*`string`* are literal regexp values (see §10) where *`string`* is a string constant. The regexp produced by `#rx`*`string`* is the same as produced by `(regexp` *`string`*`)`. If *`string`* is not a valid pattern, the `exn:fail:read` exception is raised.

  Expressions of the form `#rx#`*`string`* are similarly literal byte-regexp values. The regexp produced by `#rx#`*`string`* is the same as produced by `(byte-regexp #`*`string`*`)`.

- Expressions of the form `#px`*`string`* and `#px#`*`string`* are like the `#rx` variants, except that the regexp is as produced by `pregexp` and `byte-pregexp` (see §10) instead of `regexp` and `byte-regexp`.

- Expressions of the form `#hash((`*`key-datum`*` . `*`val-datum`*`) ···)` are literal immutable hash tables. The hash table maps each *`key-datum`* to its *`val-datum`*, comparing keys with `equal?`. The table is constructed by adding each *`key-datum`* mapping from left to right, so later mappings can hide earlier mappings if the *`key-datum`*s are `equal?`. An expression of the form `#hasheq((`*`key-datum`*` . `*`val-datum`*`) ···)` produces an immutable hash table with keys compared using `eq?`. If the value of `read-square-bracket-as-paren` parameter (see §7.9.1.3) is true, matching parentheses in a `#hash` or `#hasheq` constant can be replaced by matching square brackets. Similarly, matching curly braces can be used if `read-curly-brace-as-paren` is true.

- Values with shared structure are expressed using `#`*n*`=` and `#`*n*`#`, where *n* is a decimal integer. See §11.2.5.1.

- Expressions of the form `#%`*`x`* are symbols, where *`x`* can be a symbol or a number.

- Expressions beginning with `#~` are interpreted as compiled MzScheme code. See §14.3.

- Multi-line comments are started with `#|` and terminated with `|#`. Comments of this form can be nested arbitrarily.

- A `#;` comments out the next datum. Whitespace and comments (including `#;` comments) may appear between the `#;` and the commented-out datum. Graph-structure annotations with `#`*n*`=` and `#`*n*`#` work within the comment as if the datum were not commented out (e.g., bindings can be introduced with `#`*n*`=` for use in parts of the datum that are not commented out). When `#;` appears at the beginning of a top-level datum, however, graph-structure bindings are discarded (along with the first following datum) before reading the second following datum.

- If the first line of a `load`ed file begins with `#!`, it is ignored by the default load handler. If an ignored line ends with a backslash ("\"), then the next line is also ignored. (The `#!` convention is for shell scripts; see Chapter 18 for details.)

- A `#hx` shifts the reader into H-expression mode (see §19) for one H-expression. A `#sx` has no effect in normal mode, but in H-expression mode, it shifts the reader back to (normal) S-expression mode. The `read-honu` and `read-honu-syntax` procedures read as if the stream starts with `#hx`.

- A `#honu` shifts the reader into H-expression mode (see §19) and reads repeatedly until an end-of-file is encountered. The H-expression results are wrapped in a module-formed S-expression, as described in §19.
- A `#reader` must be followed by a datum. The datum is passed to the procedure that is the value of the `current-reader-guard` parameter (see §7.9.1.3), and the result is used as a module path. The module path is passed to `dynamic-require` (see §5.5) with either `'read` or `'read-syntax` (depending on whether parsing started with `read` or `read-syntax`). The resulting procedure should accept the same arguments as `read` or `read-syntax` (with all optional arguments as required). The procedure is given the port whose stream contained `#reader`, and it should produce a datum result. If the result is a syntax object in `read` mode it is converted to a datum using `syntax-object->datum`; if the result is not a syntax object in `read-syntax` mode, it is converted to one using `datum->syntax-object`. See also §11.2.9.1 and §11.2.9.2 for information on special-comment results and recursive reads. If the `read-accept-reader` parameter is set to `#f`, then `#reader` is disallowed as input.

Reading from a custom port can produce arbitrary values generated by the port; see §11.1.7 for details. If the port generates a non-character value in a position where a character is required (e.g., within a string), the `exn:fail:read:non-char` exception is raised.

11.2.5 Default Printer

MzScheme's printer obeys the following non-standard rules (though the rules for `print` do not apply when the `print-honu` parameter is set to `#t`; see §7.9.1.4).

- A vector can be printed by `write` and `print` using the shorthand described in §11.2.4, where the vector's length is printed between the leading `#` and the opening parenthesis and repeated tail elements are omitted. For example, `#(1 2 2 2)` is printed as `#4(1 2)`. The `display` procedure does not output vectors using this shorthand. Shorthand vector printing is controlled with the `print-vector-length` boolean parameter (see §7.9.1.4). Shorthand vector printing is enabled by default.
- Boxes (see §3.11) can be printed with the `#&` notation (see §11.2.4). When box printing is disabled, all boxes are printed un`read`ably as `#<box>`. Box printing is controlled with the `print-box` boolean parameter (see §7.9.1.4). Box printing is enabled by default.
- Structures (see Chapter 4) can be printed using either a custom-write procedure or vector notation. See §11.2.10 for information on custom-write procedures; the following information applies only when no custom-write procedure is specified. In the vector form of output, the first item is a symbol of the form `struct:`*s* — where *s* is the name of the structure — and the remaining elements are the elements of the structure, but the vector exposes only as much information about the structure as the current inspector can access (see §4.5). When structure printing is disabled, or when no part of the structure is accessible to the current inspector, a structure is printed un`read`ably as `#<struct:`*s*`>`. Structure printing is controlled with the `print-struct` boolean parameter (see §7.9.1.4). Structure printing is enabled by default.
- Symbols containing spaces or special characters `write` using escaping backslashes and quoting vertical bars. When the `read-case-sensitive` parameter is set to `#f`, then symbols containing uppercase characters also use escaping backslashes or quoting vertical bars. In addition, symbols are quoted with vertical bars or a leading backslash when they would otherwise print the same as a numerical constant. If the value of the `read-accept-bar-quote` boolean parameter is `#f` (see §7.9.1.3), then backslashes are always used to escape special characters instead of quoting them with vertical bars, and a vertical bar is not treated as a special character. Otherwise, quoting bars are used in printing when bar at the beginning and one at the end suffices to correctly print the symbol. See §11.2.4 for more information about symbol parsing. Symbols `display` without escaping or quoting special characters.
- Keywords `write` and `display` the same as symbols, except with a leading hash and colon, and without special handing when the printed form matches a number (since the leading `#:` distinguishes the keyword).

- Characters with the special names described in §11.2.4 `write` using the same name. (Some characters have multiple names; the `#\newline` and `#\nul` names are used instead of `#\linefeed` and `#\null`). Other graphic characters (according to `char-graphic?`; see §3.4) `write` as `#\` followed by the single character, and all others characters are written in `#\u` notation with four digits or `#\U` notation with eight digits (using the latter only if the character value it does not fit in four digits). All characters `display` as a single character.

- Strings containing non-graphic, non-blank characters (according to `char-graphic?` and `char-blank?`; see §3.4) `write` using the escape sequences described in §11.2.4, using `\a`, `\b`, `\t`, `\n`, `\v`, `\f`, `\r`, or `\e` if possible, otherwise using `\u` with four hexadecimal digits or `\U` with eight hexadecimal digits (using the latter only if the character value does not fit into four digits). All strings `display` as their literal character sequences.

- Byte strings `write` using `#"`, where each byte in the string content is written using the corresponding ASCII decoding if the byte is between 0 and 127 and the character is graphic or blank (according to `char-graphic?` and `char-blank?`; see §3.4). Otherwise, the byte is written using `\a`, `\b`, `\t`, `\n`, `\v`, `\f`, `\r`, or `\e` if possible, otherwise using `\o` with one to three octal digits (only as many as necessary). All strings `display` as their literal byte sequence; this byte sequence may not be a valid UTF-8 encoding, so it may not correspond to a sequence of characters.

- Paths (see §11.3.1) by `write` like other un`read`able values using `#<path:...>`. A path `display`s in the same way as the result of `path->string` applied to the path.

- Regexp values print using the form `#rx`*`string`*, where *`string`* is the `write` form of the regexp's source character string or byte string. Similarly, byte-regexp values print starting with `#rx#`.

- Hash tables by default print un`read`ably as `#<hash-table>`. When the `print-hash-table` parameter is set to true (see §7.9.1.4), hash tables print using the form `#hash((`*`key`*` . `*`val`*`) ···)` or `#hasheq((`*`key`*` . `*`val`*`) ···)` for tables using `equal?` or `eq?` key comparisons, respectively. Hash tables with weakly held keys always print un`read`ably as `#<hash-table>`.

- Values with shared structure can be printed using `#`*`n`*`=` and `#`*`n`*`#`, where *n* is a decimal integer. See §11.2.5.1.

- A value with no `read`able format prints as `#<...>`, but only when the `print-unreadable` parameter is set to `#t` (the default; see also §7.9.1.4). When the parameter's value is `#f`, attempting to print an un`read`able value raises `exn:fail:contract`.

11.2.5.1 Sharing Structure in Input and Output

MzScheme can read and print Common LISP-style *graphs*, values with shared structure (including cycles). Graphs are described by tagging the shared structure once with `#`*`n`*`=` (using some decimal integer *n* with no more than eight digits) and then referencing it later with `#`*`n`*`#` (using the same number *n*). For example, the following datum represents the infinite list of ones:

```
#0=(1 . #0#)
```

If this graph is entered into MzScheme's `read-eval-print` loop, MzScheme's compiler will loop forever, trying to compile an infinite expression. In contrast, the following expression defines *ones* to the infinite list of ones, using `quote` to hide the infinite list from the compiler:

```
(define ones (quote #0=(1 . #0#)))
```

A tagged structure can be referenced multiple times. Here, *v* is defined to be a vector containing the same `cons` cell in all three slots:

```
(define v #(#1=(cons 1 2) #1# #1#))
```

A tag `#`*n*`=` must appear to the left of all references `#`*n*`#`, and all references must appear in the same top-level datum as the tag. By default, MzScheme's printer will display a value without showing the shared structure:

```
#((1 . 2) (1 . 2) (1 . 2))
```

Graph reading and printing are controlled with the `read-accept-graph` and `print-graph` boolean parameters (see §7.9.1.4). Graph reading is enabled by default, and graph printing is disabled by default. However, when the printer encounters a graph containing a cycle, graph printing is automatically enabled, temporarily. (For this reason, the `display`, `write`, and `print` procedures require memory proportional to the depth of the value being printed.) When graph reading is disabled and a graph is provided as input, the `exn:fail:read` exception is raised.

If the *n* in a `#`*n*`=` form or a `#`*n*`#` form contains more than eight digits, the `exn:fail:read` exception is raised. If a `#`*n*`#` form is not preceded by a `#`*n*`=` form using the same *n*, the `exn:fail:read` exception is raised. If two `#`*n*`=` forms are in the same expression for the same *n*, the `exn:fail:read` exception is raised.

11.2.6 Replacing the Reader

Each input port has its own *port read handler*. This handler is invoked to read from the port when the built-in `read` or `read-syntax` procedure is applied to the port.[8] A port read handler is applied to either one argument or two arguments:

- A single argument is supplied when the port is used with `read`; the argument is the port being read. The return value is the value that was read from the port (or end-of-file).
- Two arguments are supplied when the port is used with `read-syntax`; the first argument is the port being read, and the second argument is a value indicating the source. The return value is a syntax object that was read from the port (or end-of-file).

A port's read handler is configured with `port-read-handler`:

- `(port-read-handler` *`input-port`*`)` returns the current port read handler for *`input-port`*.
- `(port-read-handler` *`input-port proc`*`)` sets the handler for *`input-port`* to *`proc`*.

The default port read handler reads standard Scheme expressions with MzScheme's built-in parser (see §11.2.4). It handles a special result from a custom input port (see §11.1.7.1) by treating it as a single expression, except that special-comment values (see §11.2.9.1) are treated as whitespace.

The `read` and `read-syntax` procedures themselves can be customized through a readtable; see §11.2.8 for more information.

11.2.7 Replacing the Printer

Each output port has its own *port display handler*, *port write handler*, and *port print handler*. These handlers are invoked to output to the port when the standard `display`, `write` or `print` procedure is applied to the port. A port display/write/print handler takes a two arguments: the value to be printed and the destination port. The handler's return value is ignored.

- `(port-display-handler` *`output-port`*`)` returns the current port display handler for *`output-port`*.

[8] The port read handler is not used for `read/recursive` or `read-syntax/recursive`.

- `(port-display-handler` *`output-port proc`*`)` sets the display handler for *`output-port`* to *`proc`*.
- `(port-write-handler` *`output-port`*`)` returns the current port write handler for *`output-port`*.
- `(port-write-handler` *`output-port proc`*`)` sets the write handler for *`output-port`* to *`proc`*.
- `(port-print-handler` *`output-port`*`)` returns the current port print handler for *`output-port`*.
- `(port-print-handler` *`output-port proc`*`)` sets the print handler for *`output-port`* to *`proc`*.

The default port display and write handlers print Scheme expressions with MzScheme's built-in printer (see §11.2.5). The default print handler calls the global port print handler (the value of the `global-port-print-handler` parameter; see §7.9.1.2); the default global port print handler is the same as the default write handler.

11.2.8 Customizing the Reader through Readtables

A *readtable* configures MzScheme's built-in reader by adjusting the way that individual characters are parsed. MzScheme readtables are just like readtables in Common LISP, except that an individual readtable is immutable, and the procedures for creating and inspecting readtables are somewhat different than the Common LISP procedures.

The readtable is consulted at specific times by the reader:

- when looking for the start of an S-expression;
- when determining how to parse an S-expression that starts with hash ("#");
- when looking for a delimiter to terminate a symbol or number;
- when looking for an opener (such as "("), closer (such as ")"), or dot (".") after the first character parsed as a sequence for a list, vector, or hash table; or
- when looking for an opener after `#`*`n`* in a vector of specified length *`n`*.

In particular, after parsing a character that is mapped to the default behavior of semi-colon (";"), the readtable is ignored until the comment's terminating newline is discovered. Similarly, the readtable does not affect string parsing until a closing double-quote is found. Meanwhile, if a character is mapped to the default behavior of an open parenthesis ("("), then it starts sequence that is closed by any character that is mapped to a close parenthesis (")"). An apparent exception is that the default parsing of a vertical bar ("|") quotes a symbol until a matching character is found, but the parser is simply using the character that started the quote; it does not consult the readtable.

For many contexts, `#f` identifies the default readtable for MzScheme. In particular, `#f` is the initial value for the `current-readtable` parameter (see §7.9.1.3), which causes the reader to behave as described in §11.2.4. Adjust MzScheme's default reader by setting the `current-readtable` parameter to a readtable created with `make-readtable`.

`(make-readtable` *`readtable`* [*`char-or-false symbol-or-char readtable-or-proc`* $\cdots^1$]`)` creates a new readtable that is like *`readtable`* (which can be `#f`), except that the reader's behavior is modified for each *`char`* according to the given *`symbol-or-char`* and *`readtable-or-proc`*. The $\cdots^1$ for `make-readtable` applies to all three of *`char`*, *`symbol-or-char`*, and *`readtable-or-proc`*; in other words, the total number of arguments to `make-readtable` must be one modulo three.

The possible combinations for *`char-or-false`*, *`symbol-or-char`*, and *`readtable-or-proc`* are as follows:

- `char 'terminating-macro proc` — causes `char` to be parsed as a delimiter, and an unquoted/uncommented `char` in the input string triggers a call to the *reader macro* `proc`; the activity of `proc` is described further below. Conceptually, characters like semi-colon (";") and parentheses are mapped to terminating reader macros in the default readtable.
- `char 'non-terminating-macro proc` — like the `'terminating-macro` variant, but `char` is not treated as a delimiter, so it can be used in the middle of an identifier or number. Conceptually, hash ("#") is mapped to a non-terminating macro in the default readtable.
- `char 'dispatch-macro proc` — like the `'non-terminating-macro` variant, but `char` only when it follows a hash ("#") — or, more precisely, when the character follows one that has been mapped to the behavior of hash in the default readtable.
- `char like-char readtable` — causes `char` to be parsed in the same way that `like-char` is parsed in `readtable`, where `readtable` can be `#f` to indicate the default readtable. Mapping a character to the same actions as vertical bar ("|") in the default reader means that the character starts quoting for symbols, and the same character terminates the quote; in contrast, mapping a character to the same action as a double quote means that the character starts a string, but the string is still terminated with a closing double quote. Finally, mapping a character to an action in the default readtable means that the character's behavior is sensitive to parameters that affect the original character; for example, mapping a character to the same action is a curly brace ("{") in the default readtable means that the character is disallowed when the `read-curly-brace-as-paren` parameter is set to `#f`.
- `#f 'non-terminating-macro proc` — replaces the macro used to parse characters with no specific mapping: i.e., characters (other than hash or vertical bar) that can start a symbol or number with the default readtable.

If multiple `'dispatch-macro` mappings are provided for a single `char-or-false`, all but the last one are ignored. Similarly, if multiple non-`'dispatch-macro` mappings are provided for a single `char-or-false`, all but the last one are ignored.

A reader macro `proc` must accept six arguments, and it can optionally accept two arguments. See §11.2.9 for information on the procedure's arguments and results.

A reader macro normally reads characters from the given input port to produce a value to be used as the "reader macro-expansion" of the consumed characters. The reader macro might produce a special-comment value to cause the consumed character to be treated as whitespace, and it might use `read/recursive` or `read-syntax/recursive`; see §11.2.9.1 and §11.2.9.2 for more information on these topics.

`(readtable-mapping readtable char)`, where `readtable` is not `#f`, produces information about the mappings in `readtable` for `char`. The result is three values:

- either a character (mapping is to same behavior as the character in the default readtable), `'terminating-macro`, or `'non-terminating-macro`; this result reports the main (i.e., non-`'dispatch-macro`) mapping for `char`. When the result is a character, then `char` is mapped to the same behavior as the returned character in the default readtable.
- either `#f` or a reader-macro procedure; the result is a procedure when the first result is `'terminating-macro` or `'non-terminating-macro`.
- either `#f` or a reader-macro procedure; the result is a procedure when the character has a `'dispatch-macro` mapping in `readtable` to override the default dispatch behavior.

Note that reader-macro procedures for the default readtable are not directly accessible. To invoke default behaviors, use `read/recursive` or `read-syntax/recursive` (see §11.2.9.2) with a character and the `#f` readtable.

Extended example:

```
;; Provides raise-read-error and raise-read-eof-error
(require (lib "readerr.ss" "syntax"))

(define (skip-whitespace port)
  ;; Skips whitespace characters, sensitive to the current
  ;; readtable's definition of whitespace
  (let ([ch (peek-char port)])
    (unless (eof-object? ch)
      ;; Consult current readtable:
      (let-values ([(like-ch/sym proc dispatch-proc)
                    (readtable-mapping (current-readtable) ch)])
        ;; If like-ch/sym is whitespace, then ch is whitespace
        (when (and (char? like-ch/sym)
                   (char-whitespace? like-ch/sym))
          (read-char port)
          (skip-whitespace port))))))

(define (skip-comments read-one port src)
  ;; Recursive read, but skip comments and detect EOF
  (let loop ()
    (let ([v (read-one)])
      (cond
       [(special-comment? v) (loop)]
       [(eof-object? v)
        (let-values ([(l c p) (port-next-location port)])
          (raise-read-eof-error "unexpected EOF in tuple" src l c p 1))]
       [else v]))))

(define (parse port read-one src)
  ;; First, check for empty tuple
  (skip-whitespace port)
  (if (eq? #\> (peek-char port))
       null
       (let ([elem (read-one)])
         (if (special-comment? elem)
             ;; Found a comment, so look for > again
             (parse port read-one src)
             ;; Non-empty tuple:
             (cons elem
                   (parse-nonempty port read-one src))))))

(define (parse-nonempty port read-one src)
  ;; Need a comma or closer
  (skip-whitespace port)
  (case (peek-char port)
    [(#\>) (read-char port)
     ;; Done
     null]
    [(#\,) (read-char port)
     ;; Read next element and recur
     (cons (skip-comments read-one port src)
           (parse-nonempty port read-one src))]
    [else
    ;; Either a comment or an error; grab location (in case
    ;; of error) and read recursively to detect comments
     (let-values ([(l c p) (port-next-location port)]
                  [(v) (read-one)])
       (cond
        [(special-comment? v)
         ;; It was a comment, so try again
         (parse-nonempty port read-one src)]
        [else
        ;; Wasn't a comment, comma, or closer; error
        ((if (eof-object? v) raise-read-eof-error raise-read-error)
         "expected `,' or `>'" src l c p 1)]))]))

(define (make-delims-table)
  ;; Table to use for recursive reads to disallow delimiters
  ;; (except those in sub-expressions)
  (letrec ([misplaced-delimiter
             (case-lambda
              [(ch port) (unexpected-delimiter ch port #f #f #f #f)]
              [(ch port src line col pos)
               (raise-read-error
                (format "misplaced `~a' in tuple" ch) src line col pos 1)])])
     (make-readtable (current-readtable)
                     #\, 'terminating-macro misplaced-delimiter
                     #\> 'terminating-macro misplaced-delimiter)))

(define (wrap l)
  `(make-tuple (list ,@l)))

(define parse-open-tuple
  (case-lambda
   [(ch port)
    ;; `read' mode
    (wrap (parse port
                  (lambda () (read/recursive port #f
                                             (make-delims-table)))
                  (object-name port)))]
   [(ch port src line col pos)
    ;; `read-syntax' mode
    (datum->syntax-object
     #f
     (wrap (parse port
                  (lambda () (read-syntax/recursive src port #f
                                                    (make-delims-table)))
                  src))
     (let-values ([(l c p) (port-next-location port)])
       (list src line col pos (and pos (- p pos)))))]))

(define tuple-readtable
  (make-readtable #f #\< 'terminating-macro parse-open-tuple))

(parameterize ([current-readtable tuple-readtable])
  (read (open-input-string "<1 , 2 , \"a\">")))
```

```
;; ⇒ '(make-tuple (list 1 2 "a"))

(parameterize ([current-readtable tuple-readtable])
  (read (open-input-string "< #||# 1 #||# , #||# 2 #||# , #||# \"a\" #||# >")))
;; ⇒ '(make-tuple (list 1 2 "a"))

(define tuple-readtable+
  (make-readtable tuple-readtable
                  #\* 'terminating-macro (lambda a (make-special-comment #f))
                  #\_ #\space #f))
(parameterize ([current-readtable tuple-readtable+])
  (read (open-input-string "< * 1 __,__  2 __,__ * \"a\" * >")))
;; ⇒ '(make-tuple (list 1 2 "a"))
```

11.2.9 Reader-Extension Procedures

MzScheme's reader can be extended in three ways: through a reader-macro procedure in a readtable (see §11.2.8), through a `#reader` form (see §11.2.4), or through a custom-port byte reader that returns a "special" result procedure (see §11.1.7.1). All three kinds of procedures accept similar arguments, and their results are treated in the same way by `read` and `read-syntax` (or, more precisely, by the default read handler; see §11.2.6).

Calls to these reader-extension procedures can be triggered through `read`, `read/recursive`, `read-syntax`, or `read-honu-syntax`. In addition, a special-read procedure can be triggered by calls to `read-honu`, `read-honu/recursive`, `read-honu-syntax`, `read-honu-syntax/recursive`, `read-char-or-special`, or by the context of `read-bytes-avail!`, `read-bytes-avail!*`, `read-bytes-avail!`, and `peek-bytes-avail!*`.

Optional arities for reader-macro and special-result procedures allow them to distinguish reads via `read`, etc. from reads via `read-syntax`, etc. in the case that the source value is `#f` and no other location information is available.

Procedure arguments

A reader-macro procedure must accept six arguments, and it can optionally accept two arguments. The first two arguments are always the character that triggered the reader macro and the input port for reading. When the reader macro is triggered by `read-syntax` (or `read-syntax/recursive`), the procedure is passed four additional arguments that represent a source location. When the reader macro is triggered by `read` (or `read/recursive`), the procedure is passed only two arguments if it accepts two arguments, otherwise it is passed six arguments where the last four are all `#f`.

A `#reader`-loaded procedure accepts the same arguments as either `read` or `read-syntax`, depending on whether the procedure was loaded through `read`, etc. or through `read-syntax`, etc.

A special-result procedure must accept four arguments, and it can optionally accept zero arguments. When the special read is triggered by `read-syntax` (or `read-honu-syntax`, `read-syntax/recursive`, etc.), the procedure is passed four arguments that represent a source location. When the special read is triggered by `read` (or `read-char-or-special`, `read-honu`, `read/syntax`, etc.), the procedure is passed no arguments if it accepts zero arguments, otherwise it is passed four arguments that are all `#f`.

Procedure result

When a reader-extension procedure is called in syntax-reading mode (via `read-syntax`, etc.), it should generally return a syntax object that has no lexical context (e.g., a syntax object created using `datum->syntax-object` with `#f` as the first argument and with the given location information as the third argument). Another possible result is a special-comment value (see §11.2.9.1). If the procedure's result is not a syntax object and not a special-comment value, it is converted to one using `datum->syntax-object`.

When a reader-extension procedure is called in non-syntax-reading modes, it should generally not return a syntax object. If a syntax object is returned, it is converted to a plain value using `syntax-object->datum`.

In either context, when the result from a reader-extension procedure is a special-comment value (see §11.2.9.1), then `read`, `read-syntax`, etc. treat the value as a delimiting comment and otherwise ignore it.

Also in either context, the result may be copied to prevent mutation to pairs, vectors, or boxes before the read result is completed, and to support the construction of graphs with cycles. Mutable pairs, boxes, and vectors are copied, along with any pairs, boxes, or vectors that lead to such mutable values, to placeholders produced by a recursive read (see §11.2.9.2), or to references of a shared value. Graph structure (including cycles) is preserved in the copy.

11.2.9.1 Special Comments

`(make-special-comment` *`v`*`)` creates a special-comment value that encapsulates *`v`*. The `read`, `read-syntax`, etc. procedures treat values constructed with `make-special-comment` as delimiting whitespace when returned by a reader-extension procedure (see §11.2.9).

`(special-comment?` *`v`*`)` returns `#t` if *`v`* is the result of `make-special-comment`, `#f` otherwise.

`(special-comment-value` *`sc`*`)` returns the value encapsulated by the special-comment value *`sc`*. This value is never used directly by a reader, but it might be used by the context of a `read-char-or-special`, etc. call that detects a special comment.

11.2.9.2 Recursive Reads

`(read/recursive` [*`input-port char-or-false readtable`*]`)` is similar to calling `read`, but it is normally used during the dynamic extent of `read` within a reader-extension procedure (see §11.2.9). The main effect of using `read/recursive` instead of `read` is that graph-structure annotations (see §11.2.5.1) in the nested read are considered part of the overall read. Since the result is wrapped in a placeholder, however, it is not directly inspectable.

If *`char-or-false`* is provided and not `#f`, it is effectively prefixed to the beginning of *`input-port`*'s stream for the read. (To prefix multiple characters, use `input-port-append` from MzLib's **port** library; see Chapter 35 of *PLT MzLib: Libraries Manual*.)

The *`readtable`* argument, which defaults to `(current-readtable)`, is used for top-level parsing to satisfy the read request; recursive parsing within the read (e.g., to read the elements of a list) instead uses the current readtable as determined by the `current-readtable` parameter. A reader macro might call `read/recursive` with a character and readtable to effectively invoke the readtable's behavior for the character. If *`readtable`* is `#f`, the default readtable is used for top-level parsing.

When called within the dynamic extent of `read`, the `read/recursive` procedure produces either an opaque placeholder value, a special-comment value, or an end-of-file. The result is a special-comment value (see §11.2.9.1) when the input stream's first non-whitespace content parses as a comment. The result is end-of-file when `read/recursive` encounters an end-of-file. Otherwise, the result is a placeholder that protects graph references that are not yet resolved. When this placeholder is returned within an S-expression that is produced by any reader-extension procedure (see §11.2.9) for the same outermost `read`, it will be replaced with the actual read value before the outermost `read` returns.

`(read-syntax/recursive` [*`source-name-v input-port char-or-false readtable`*]`)` is analogous to calling `read/recursive`, but the resulting value encapsulates S-expression structure with source-location information. As with `read/recursive`, when `read-syntax/recursive` is used within the dynamic extent of `read-syntax`, the result of from `read-syntax/recursive` is either a special-comment value, end-of-file, or opaque graph-structure placeholder (not a syntax object). The placeholder can be embedded in an S-expression or syntax object returned by a reader macro, etc., and it will be replaced with the actual syntax object before the outermost `read-syntax` returns.

Using `read/recursive` within the dynamic extent of `read-syntax` does not allow graph structure for reading to be included in the outer `read-syntax` parsing, and neither does using `read-syntax/recursive` within the dynamic extent of `read`. In those cases, `read/recursive` and `read-syntax/recursive` produce results like `read` and `read-syntax`.

See §11.2.8 for an extended example that uses `read/recursive` and `read-syntax/recursive`.

11.2.10 Customizing the Printer through Custom-Write Procedures

The built-in `prop:custom-write` structure type property associates a procedures to a structure type. The procedure is used by the default printer to `display` or `write` (or `print`) instances of the structure type.

See §4.4 for general information on structure type properties.

The procedure for a `prop:custom-write` value takes three arguments: the structure to be printed, the target port, and a boolean that is `#t` for `write` mode and `#f` for `display` mode. The procedure should print the value to the given port using `write`, `display`, `fprintf`, `write-special`, etc.

The write handler, display handler, and print handler are specially configured for a port given to a custom-write procedure. Printing to the port through `display`, `write`, or `print` prints a value recursively with sharing annotations. To avoid a recursive print (i.e., to print without regard to sharing with a value currently being printed), print instead to a string or pipe and transfer the result to the target port using `write-string` and `write-special`. To recursively print but to a port other than the one given to the custom-write procedure, copy the given port's write handler, display handler, and print handler to the other port.

The port given to a custom-write handler is not necessarily the actual target port. In particular, to detect cycles and sharing, the printer invokes a custom-write procedure with a port that records recursive prints, and does not retain any other output.

Recursive print operations may trigger an escape from the call to the custom-write procedure (e.g., for pretty-printing where a tentative print attempt overflows the line, or for printing error output of a limited width).

The following example definition of a *tuple* type includes custom-write procedures that print the tuple's list content using angle brackets in `write` mode and no brackets in `display` mode. Elements of the tuple are printed recursively, so that graph and cycle structure can be represented.

```
(define (tuple-print tuple port write?)
  (when write? (write-string "<" port))
  (let ([l (tuple-ref tuple 0)])
    (unless (null? l)
      ((if write? write display) (car l) port)
      (for-each (lambda (e)
                  (write-string ", " port)
                  ((if write? write display) e port))
                (cdr l))))
  (when write? (write-string ">" port)))

(define-values (s:tuple make-tuple tuple? tuple-ref tuple-set!)
  (make-struct-type 'tuple #f 1 0 #f
                    (list (cons prop:custom-write tuple-print))))

(display (make-tuple '(1 2 "a"))) ; prints 1, 2, a
```

```
(let ([t (make-tuple (list 1 2 "a"))])
  (set-car! (tuple-ref t 0) t)
  (write t))  ; prints #0=<#0#, 2, "a">
```

11.3 Filesystem Utilities

MzScheme provides many operations for accessing and modifying filesystems in a (mostly) platform-independent manner. Additional filesystem utilities are in MzLib; see also Chapter 20 of *PLT MzLib: Libraries Manual*.

11.3.1 Paths

The format of a filesystem path varies across platforms. For example, under Unix, directories are separated by "/" while Windows uses both "/" and "\". Furthermore, for most Unix filesystems, the true name of a file is a byte string, but users prefer to see the bytes decoded in a locale-specific way when the filename is printed. MzScheme therefore provides a `path` datatype for managing filesystem paths, and procedures such as `build-path`, `path->string`, and `bytes->path` for manipulating paths.

When a MzScheme procedure takes a filesystem path as an argument, the path can be provided either as a string or as an instance of the `path` datatype. If a string is provided, it is converted to a path using `string->path`. A MzScheme procedure that generates a filesystem path always generates a path value.

By default, paths are created and manipulated for the current platform, but procedures that merely manipulate paths (without using the filesystem) can manipulate paths using conventions for other supported platforms. The `bytes->path` procedure accepts an optional argument that indicates the platform for the path, either `'unix` or `'windows`. For other functions, such as `build-path` or `simplify-path`, the behavior is sensitive to the kind of path that is supplied. Unless otherwise specified, a procedure that requires a path accepts only paths for the current platform.

Two path values are `equal?` when they are use the same convention type and when their byte-string representations are `equal?`. A path string (or byte string) cannot be empty, and it cannot contain a nul character or byte. When an empty string or a string containing nul is provided as a path to any procedure except `absolute-path?`, `relative-path?`, or `complete-path?` the `exn:fail:contract` exception is raised.

Most MzScheme primitives that take path perform an expansion on the path before using it. Procedures that build paths or merely check the form of a path do not perform this expansion, with the exception of `simplify-path` for Windows paths. For more information about path expansion and other platform-specific details, see §20.1 for Unix and Mac OS X paths and §20.2 for Windows paths.

The basic path utilities are as follows:

- `(path? v)` returns `#t` if *v* is a path value for the current platform (not a string, and not a path for a different platform), `#f` otherwise.
- `(path-string? v)` returns `#t` if *v* is either a path value or a non-empty string without nul characters, `#f` otherwise.
- `(path-for-some-system? v)` returns `#t` if *v* is a path value for some platform (not a string), `#f` otherwise.
- `(string->path` *`string`*`)` produces a path whose byte-string name is `(string->bytes/locale` *`string`* `(char->integer #\?))`; see §3.6 for more information on `string->bytes/locale`. Beware that the current locale might not encode every string, in which case `string->path` can produce the same path for different *`string`*s. See also `string->path-element`, which should be used instead of `string->path` when a string represents a single path element.

- `(bytes->path` *`bytes`* `[`*`type-symbol`*`])` produces a path (for some platform) whose byte-string name is *`bytes`*. The optional *`type-symbol`* specifies the convention to use for the path, and it can be any possible result from `system-path-convention-type` (see below); it defaults to the value for the current platform. For converting relative path elements from literals, use instead `bytes->path-element` (described below), which applies a suitable encoding for individual elements.

- `(path->string` *`path`*`)` produces a string that represents *`path`* by decoding *`path`*'s byte-string name using the current locale's encoding; "?" is used in the result string where encoding fails, and if the encoding result is the empty string, then the result is `"?"`. The resulting string is suitable for displaying to a user, string-ordering comparisons, etc., but it is not suitable for re-creating a path (possibly modified) via `string->path`, since decoding and re-encoding the path's byte string may lose information. Furthermore, for display and sorting based on individual path elements (such as pathless file names), use `path-element->string`, instead, to avoid special encodings use to represent some relative paths. See §20.2 for specific information about the conversion of Windows paths.

- `(path->bytes` *`path`*`)` produces *`path`*'s byte string representation. No information is lost in this translation, so that `(bytes->path (path->bytes` *`path`*`) (path-convention-type` *`path`*`))` always produces a path is that is `equal?` to *`path`*. The *`path`* argument can be a path for any platform. Conversion to and from byte values is useful for marshaling and unmarshaling paths, but manipulating the byte form of a path is generally a mistake. In particular, the byte string may start with a **\\?\REL** encoding for Windows paths or a **./~** encoding for Unix and Mac OS X paths. Instead of `path->bytes`, use `split-path` and `path-element->bytes` (described below) to manipulate individual path elements.

- `(string->path-element` *`string`*`)` is like `string->path`, except that *`string`* corresponds to a single relative element in a path, and it is encoded as necessary to convert it to a path. See §20.1 for more information on the conversion for Unix and Mac OS X paths, and see §20.2 for more information on the conversion for Windows paths. If *`string`* does not correspond to any path element (e.g., it is an absolute path, or it can be split), or if it corresponds to an up-directory or same-directory indicator under Unix and Mac OS X, then `exn:fail:contract` exception is raised. As for `path->string`, information can be lost from *`string`* in the locale-specific conversion to a path.

- `(bytes->path-element` *`bytes`* `[`*`type-symbol`*`])` is like `bytes->path`, except that *`bytes`* corresponds to a single relative element in a path. In terms of conversions and restrictions on *`bytes`*, `bytes->path-element` is like `string->path-element`. The `bytes->path-element` procedure is generally the best choice for reconstructing a path based on another path (where the other path is deconstructed with `split-path` and `path-element->bytes`) when ASCII-level manipulation of path elements is necessary.

- `(path-element->string` *`path`*`)` is like `path->string`, except any encoding prefix is removed. See §20.1 for more information on the conversion for Unix and Mac OS X paths, and see §20.2 for more information on the conversion for Windows paths. In addition, trailing path separators are removed, as by `split-path`. The *`path`* argument must be such that `split-path` applied to *`path`* would return `'relative` as its first result and a path as its second result, otherwise the `exn:fail:contract` exception is raised. The `path-element->string` procedure is generally the best choice for presenting a pathless file or directory name to a user.

- `(path-element->bytes` *`path`*`)` is like `path->bytes`, except that any encoding prefix is removed, etc., as for `path-element->string`. For any reasonable locale, consecutive ASCII characters in the printed form of *`path`* are mapped to consecutive byte values that match each character's code-point value, and a leading or trailing ASCII character is mapped to a leading or trailing byte, respectively. The *`path`* argument can be a path for any platform. The `path-element->bytes` procedure is generally the right choice (in combination with `split-path`) for extracting the content of a path to manipulate it at the ASCII level (then reassembling the result with `bytes->path-element` and `build-path`).

- `(path-convention-type` *`path`*`)` accepts a path value (not a string) and returns its convention type. The possible results are the same as the possible results of `system-path-convention-type` (see below).

- `(system-path-convention-type)` returns the path convention type of the current platform: `'unix` for Unix and Mac OS X, `'windows` for Windows.

- `(build-path` *`base-path sub-path`* `···)` creates a path given a base path and any number of sub-path extensions. If *`base-path`* is an absolute path, the result is an absolute path; if *`base`* is a relative path, the result is a relative path. Each *`sub-path`* must be either a relative path, a directory name, the symbol `'up` (indicating the relative parent directory), or the symbol `'same` (indicating the relative current directory). For Windows paths, if *`base-path`* is a drive specification (with or without a trailing slash) the first *`sub-path`* can be an absolute (driveless) path. For all platforms, the last *`sub-path`* can be a filename.

  The *`base-path`* and *`sub-paths`* arguments can be paths for any platform. The platform for the resulting path is inferred from the *`base-path`* and *`sub-path`* arguments, where string arguments imply a path for the current platform. If different arguments are for different platforms, the `exn:fail:contract` exception is raised. If no argument implies a platform (i.e., all are `'up` or `'same`), the generated path is for the current platform.

  Each *`sub-path`* and *`base-path`* can optionally end in a directory separator. If the last *`sub-path`* ends in a separator, it is included in the resulting path.

  If *`base-path`* or *`sub-path`* is an illegal path string (because it is empty or contains a nul character), the `exn:fail:contract` exception is raised.

  The `build-path` procedure builds a path *without* checking the validity of the path or accessing the filesystem.

  See §20.1 for more information on the construction of Unix and Mac OS X paths, and see §20.2 for more information on the construction of Windows paths.

  The following examples assume that the current directory is **/home/joeuser** for Unix examples and **C:\Joe's Files** for Windows examples.

  ```
  (define p1 (build-path (current-directory) "src" "scheme"))
   ; Unix: p1 ⇒ "/home/joeuser/src/scheme"
   ; Windows: p1 ⇒ "C:\Joe's Files\src\scheme"
  (define p2 (build-path 'up 'up "docs" "MzScheme"))
   ; Unix: p2 ⇒ "../../docs/MzScheme"
   ; Windows: p2 ⇒ "..\..\docs\MzScheme"
  (build-path p2 p1)
   ; Unix and Windows: raises exn:fail:contract because p1 is absolute
  (build-path p1 p2)
   ; Unix: ⇒ "/home/joeuser/src/scheme/../../docs/MzScheme"
   ; Windows: ⇒ "C:\Joe's Files\src\scheme\..\..\docs\MzScheme"
  ```

- `(build-path/convention-type` *`type-symbol base-path sub-path`* `···)` is like `build-path`, except a path convention type is specified explicitly. The *`type-symbol`* argument must be a possible result of `system-path-convention-type` (see above) for some platform.

- `(absolute-path?` *`path`*`)` returns `#t` if *`path`* is an absolute path, `#f` otherwise. The *`path`* argument can be a path for any platform. If *`path`* is not a legal path string (e.g., it contains a nul character), `#f` is returned. This procedure does not access the filesystem.

- `(relative-path?` *`path`*`)` returns `#t` if *`path`* is a relative path, `#f` otherwise. The *`path`* argument can be a path for any platform. If *`path`* is not a legal path string (e.g., it contains a nul character), `#f` is returned. This procedure does not access the filesystem.

- `(complete-path?` *`path`*`)` returns `#t` if *`path`* is a completely determined path (*not* relative to a directory or drive), `#f` otherwise. The *`path`* argument can be a path for any platform. Note that for Windows paths, an absolute path can omit the drive specification, in which case the path is neither relative nor complete. If *`path`* is not a legal path string (e.g., it contains a nul character), `#f` is returned. This procedure does not access the filesystem.

- `(path->complete-path` *`path`* [*`base-path`*]`)` returns *`path`* as a complete path. If *`path`* is already a complete path, it is returned as the result. Otherwise, *`path`* is resolved with respect to the complete path *`base-path`*. If *`base-path`* is omitted, *`path`* is resolved with respect to the current directory. If *`base-path`* is provided and it is not a complete path, the `exn:fail:contract` exception is raised. The *`path`* and *`base-path`* arguments can paths for any platform, as long as both are supplied; if they are for different platforms, the `exn:fail:contract` exception is raised. This procedure does not access the filesystem.

- `(path->directory-path` *`path`*`)` returns *`path`* if *`path`* syntactically refers to a directory and ends in a separator, otherwise it returns an extended version of *`path`* that specifies a directory and ends with a separator. For example, under Unix and Mac OS X, the path **x/y/** syntactically refers to a directory and ends in a separator, but **x/y** would be extended to **x/y/**, and **x/..** would be extended to **x/../**. The *`path`* argument can be a path for any platform, and the result will be for the same platform. This procedure does not access the filesystem.

- `(resolve-path` *`path`*`)` expands *`path`* and returns a path that references the same file or directory as *`path`*. Under Unix and Mac OS X, if *`path`* is a soft link to another path, then the referenced path is returned (this may be a relative path with respect to the directory owning *`path`*) otherwise *`path`* is returned (after expansion).

- `(expand-path` *`path`*`)` returns the expanded version of *`path`* (as described at the beginning of this section). The filesystem might be accessed, but the source or expanded path might be a non-existent path.

- `(simplify-path` *`path`* [*`use-filesystem?`*]`)` eliminates redundant path separators (except for a single trailing separator), up-directory ("..") and same-directory (".") indicators in *`path`*, and changes forward-slash separators to backslahs separators in Windows paths, such that the result accesses the same file or directory (if it exists) as *`path`*. In general, the pathname is normalized as much as possible — without consulting the filesystem if *`use-filesystem?`* is `#f`, and (under Windows) without changing the case of letters within the path. If *`path`* syntactically refers to a directory, the result ends with a directory separator.

  When *`path`* is simplified and *`use-filesystem?`* is true (the default), a complete path is returned; if *`path`* is relative, it is resolved with respect to the current directory, and up-directory indicators are removed taking into account soft links (so that the resulting path refers to the same directory as before).

  When *`use-filesystem?`* is `#f`, up-directory indicators are removed by deleting a preceding path element, and the result can be a relative path with up-directory indicators remaining at the beginning of the path or, for Unix and Mac OS X paths, after an initial path element that starts with tilde ("~"); otherwise, up-directory indicators are dropped when they refer to the parent of a root directory. Similarly, the result can be the same as `(build-path 'same)` (but with a trailing separator) if eliminating up-directory indicators leaves only same-directory indicators, and the result can start with a same-directory indicator for Unix and Mac OS X paths if eliminating it would make the result start with a tilde ("~").

  The *`path`* argument can be a path for any platform when *`use-filesystem?`* is `#f`, and the resulting path is for the same platform.

  The filesystem might be accessed when *`use-filesystem?`* is true, but the source or expanded path might be a non-existent path. If *`path`* cannot be simplified due to a cycle of links, the `exn:fail:filesystem` exception is raised (but a successfully simplified path may still involve a cycle of links if the cycle did not inhibit the simplification).

  See §20.1 for more information on simplifying Unix and Mac OS X paths, and see §20.2 for more information on simplifying Windows paths.

- `(normal-case-path` *`path`*`)` returns *`path`* with "normalized" case letters. For Unix and Mac OS X paths, this procedure always returns the input path, because filesystems for these platforms can be case-sensitive. For Windows paths, if *`path`* does not start \\**?**\, the resulting string uses only lowercase letters, based on the current locale. In addition, for Windows paths when the path does not start \\**?**\, all forward slashes ("/") are converted to backward slashes ("\"), and trailing spaces and periods are removed. The *`path`* argument can be a path for any platform, but beware that local-sensitive decoding and conversion of the path may be different on the current platform than for the path's platform. This procedure does not access the filesystem.

- `(split-path` *`path`*`)` deconstructs *`path`* into a smaller path and an immediate directory or file name. Three values are returned (see §2.2):

  - *base* is either
    - a path,
    - `'relative` if *path* is an immediate relative directory or filename, or
    - `#f` if *path* is a root directory.
  - *name* is either
    - a directory-name path,
    - a filename,
    - `'up` if the last part of *path* specifies the parent directory of the preceding path (e.g., ".." under Unix), or
    - `'same` if the last part of *path* specifies the same directory as the preceding path (e.g., "." under Unix).
  - *must-be-dir?* is `#t` if *path* explicitly specifies a directory (e.g., with a trailing separator), `#f` otherwise. Note that *must-be-dir?* does not specify whether *name* is actually a directory or not, but whether *path* syntactically specifies a directory.

  Compared to *path*, redundant separators (if any) are removed in the result *base* and *name*. If *base* is `#f`, then *name* cannot be `'up` or `'same`. The *path* argument can be a path for any platform, and resulting paths for the same platform. This procedure does not access the filesystem.

  See §20.1 for more information on splitting Unix and Mac OS X paths, and see §20.2 for more information on splitting Windows paths.

  `(path-replace-suffix` *path string-or-bytes*`)` returns a path that is the same as *path*, except that the suffix for the last element of the path is changed to *string-or-bytes*. If the last element of *path* has no suffix, then *string-or-bytes* is added to the path. A suffix is defined as a period followed by any number of non-period characters/bytes at the end of the path element. The *path* argument can be a path for any platform, and the result is for the same platform. If *path* represents a root, the `exn:fail:contract` exception is raised.

11.3.2 Locating Paths

The `find-system-path` and `find-executable-path` procedures locate useful files and directories:

- `(find-system-path` *kind-symbol*`)` returns a machine-specific path for a standard type of path specified by *kind-symbol*, which must be one of the following:
  - `'home-dir` — the current user's home directory.
    Under Unix and Mac OS X, this directory is determined by expanding the path ~, which is expanded by first checking for a **HOME** environment variable. If none is defined, the **USER** and **LOGNAME** environment variables are consulted (in that order) to find a user name, and then system files are consulted to locate the user's home directory.
    Under Windows, the user's home directory is the user-specific profile directory as determined by the Windows registry. If the registry cannot provide a directory for some reason, the value of the **USERPROFILE** environment variable is used instead, as long as it refers to a directory that exists. If **USERPROFILE** also fails, the directory is the one specified by the **HOMEDRIVE** and **HOMEPATH** environment variables. If those environment variables are not defined, or if the indicated directory still does not exist, the directory containing the MzScheme executable is used as the home directory.
  - `'pref-dir` — the standard directory for storing the current user's preferences. Under Unix, the directory is **.plt-scheme** in the user's home directory. Under Windows, it is **PLT Scheme** in the user's application-data folder as specified by the Windows registry; the application-data folder is usually **Application Data** in the user's profile directory. Under Mac OS X, it is **Library/Preferences** in the user's home directory. This directory might not exist.
  - `'pref-file` — a file that contains a symbol-keyed association list of preference values. The file's directory path always matches the result returned for `'pref-dir`. The file name is **plt-prefs.ss** under Unix and Windows, and it is **org.plt-scheme.prefs.ss** under Mac OS X. The file's directory might not exist. See also `get-preference` in Chapter 20 of *PLT MzLib: Libraries Manual*.

  - `'temp-dir` — the standard directory for storing temporary files. Under Unix and Mac OS X, this is the directory specified by the **TMPDIR** environment variable, if it is defined.
  - `'init-dir` — the directory containing the initialization file used by stand-alone MzScheme application. It is the same as the current user's home directory.
  - `'init-file` — the file loaded at start-up by the stand-alone MzScheme application. The directory part of the path is the same path as returned for `'init-dir`. The file name is platform-specific:
    - Unix and Mac OS X: **.mzschemerc**
    - Windows: **mzschemerc.ss**
  - `'addon-dir` — a directory for installing PLT Scheme extensions. It's the same as `'pref-dir`, except under Mac OS X, where it's **Library/PLT Scheme** in the user's home directory. This directory might not exist.
  - `'doc-dir` — the standard directory for storing the current user's documents. It's the same as `'home-dir` under Unix and Mac OS X. Under Windows, it is the user's documents folder as specified by the Windows registry; the documents folder is usually **My Documents** in the user's home directory.
  - `'desk-dir` — the directory for the current user's desktop. Under Unix, it's the same as `'home-dir`. Under Windows, it is the user's desktop folder as specified by the Windows registry; the documents folder is usually **Desktop** in the user's home directory. Under Mac OS X, it is the desktop directory, which is specifically ~/**Desktop** under Mac OS X.
  - `'sys-dir` — the directory containing the operating system for Windows. Under Unix and Mac OS X, the result is `"/"`.
  - `'exec-file` — the path of the MzScheme executable as provided by the operating system for the current invocation.[9]
  - `'run-file` — the path of the current executable; this may be different from result for `'exec-file` because an alternate path was provided through a `--name` or `-N` command-line flag to stand-alone MzScheme (or MrEd), or because an embedding executable installed an alternate path. In particular a "launcher" script created by `make-mzscheme-launcher` sets this path to the script's path. In the stand-alone MzScheme application, this path is also bound initially to `program`.
  - `'collects-dir` — a path to the main collection of libraries (see §16). If this path is relative, it's relative to the directory of `(find-system-path 'exec-file)`. This path is normally embedded in a stand-alone MzScheme executable, but it can be overridden by the `--collects` or `-X` command-line flag.
  - `'orig-dir` — the current directory at start-up, which can be useful in converting a relative-path result from `(find-system-path 'exec-file)` or `(find-system-path 'run-file)` to a complete path.

- `(path-list-string->path-list` *`string default-path-list`*`)` parses a string or byte string containing a list of paths, and returns a list of path strings. Under Unix and Mac OS X, paths in a path list are separated by a colon (":"); under Windows, paths are separated by a semi-colon (";"). Whenever the path list contains an empty path, the list *`default-path-list`* is spliced into the returned list of paths. Parts of *`string`* that do not form a valid path are not included in the returned list.

- `(find-executable-path` *`program-sub-path related-sub-path`* [*`deepest?`*]`)` finds a path for the executable *`program-sub-path`*, returning `#f` if the path cannot be found.

  If *`related-sub-path`* is not `#f`, then it must be a relative path string, and the path found for *`program-sub-path`* must be such that the file or directory *`related-sub-path`* exists in the same directory as the executable. The result is then the full path for the found *`related-sub-path`*, instead of the path for the executable.

  This procedure is used by MzScheme (as a stand-alone executable) to find the standard library collection directory (see Chapter 16). In this case, *`program`* is the name used to start MzScheme and *`related`* is `"collects"`. The *`related-sub-path`* argument is used because, under Unix and Mac OS X, *`program-sub-path`* may involve to a sequence of soft links; in this case, *`related-sub-path`* determines which link in the chain is relevant.

[9]For MrEd, the executable path is the name of a MrEd executable.

If *`related-sub-path`* is not `#f`, then when `find-executable-path` does not finds a *`program-sub-path`* that is a link to another file path, the search can continue with the destination of the link. Further links are inspected until *`related-sub-path`* is found or the end of the chain of links is reached. If *`deepest?`* is `#f` (the default), then the result corresponds to the first path in a chain of links for which *`related-sub-path`* is found (and further links are not actually explored); otherwise, the result corresponds to the last link in the chain for which *`related-sub-path`* is found.

If *`program-sub-path`* is a pathless name, `find-executable-path` gets the value of the **PATH** environment variable; if this environment variable is defined, `find-executable-path` tries each path in **PATH** as a prefix for *`program-sub-path`* using the search algorithm described above for path-containing *`program-sub-path`*s. If the **PATH** environment variable is not defined, *`program-sub-path`* is prefixed with the current directory and used in the search algorithm above. (Under Windows, the current directory is always implicitly the first item in **PATH**, so `find-executable-path` checks the current directory first under Windows.)

11.3.3 Files

The file management utilities are:

- `(file-exists?` *`path`*`)` returns `#t` if a file (not a directory) *`path`* exists, `#f` otherwise.[10]
- `(link-exists?` *`path`*`)` returns `#t` if a link *`path`* exists (Unix and Mac OS X), `#f` otherwise. Note that the predicates `file-exists?` or `directory-exists?` work on the final destination of a link or series of links, while `link-exists?` only follows links to resolve the base part of *`path`* (i.e., everything except the last name in the path). This procedure never raises the `exn:fail:filesystem` exception.
- `(delete-file` *`path`*`)` deletes the file with path *`path`* if it exists, returning void if a file was deleted successfully, otherwise the `exn:fail:filesystem` exception is raised. If *`path`* is a link, the link is deleted rather than the destination of the link.
- `(rename-file-or-directory` *`old-path new-path`* [*`exists-ok?`*]`)` renames the file or directory with path *`old-path`* — if it exists — to the path *`new-path`*. If the file or directory is renamed successfully, void is returned, otherwise the `exn:fail:filesystem` exception is raised.

  This procedure can be used to move a file/directory to a different directory (on the same disk) as well as rename a file/directory within a directory. Unless *`exists-ok?`* is provided as a true value, *`new-path`* cannot refer to an existing file or directory. Even if *`exists-ok?`* is true, *`new-path`* cannot refer to an existing file when *`old-path`* is a directory, and vice versa. (If *`new-path`* exists and is replaced, the replacement is atomic in the filesystem, except under Windows 95, 98, or Me. However, the check for existence is not included in the atomic action, which means that race conditions are possible when *`exists-ok?`* is false or not supplied.)

  If *`old-path`* is a link, the link is renamed rather than the destination of the link, and it counts as a file for replacing any existing *`new-path`*.
- `(file-or-directory-modify-seconds` *`path`* [*`secs-n fail-thunk`*]`)` returns the file or directory's last modification date as platform-specific seconds (see also §15.1) when *`secs-n`* is not provided or is `#f`.[11] If *`secs-n`* is provided and not `#f`, the access and modification times of *`path`* are set to the given time. On error (e.g., if no such file exists), *`fail-thunk`* is called if it is provided, otherwise the `exn:fail:filesystem` exception is raised
- `(file-or-directory-permissions` *`path`*`)` returns a list containing `'read`, `'write`, and/or `'execute` for the given file or directory path. On error (e.g., if no such file exists), the `exn:fail:filesystem` exception is raised. Under Unix and Mac OS X, permissions are checked for the current effective user instead of the real user.

[10] Under Windows, `file-exists?` reports `#t` for all variations of the special filenames (e.g., `"LPT1"`, `"x:/baddir/LPT1"`).

[11] For FAT filesystems under Windows, directories do not have modification dates. Therefore, the creation date is returned for a directory (but the modification date is returned for a file).

- `(file-size` *`path`*`)` returns the (logical) size of the specified file in bytes. (Under Mac OS X, this size excludes the resource-fork size.) On error (e.g., if no such file exists), the `exn:fail:filesystem` exception is raised.

- `(copy-file` *`src-path dest-path`*`)` creates the file *`dest-path`* as a copy of *`src-path`*. If the file is successfully copied, void is returned, otherwise the `exn:fail:filesystem` exception is raised. If *`dest-path`* already exists, the copy will fail. File permissions are preserved in the copy. Under Mac OS X, the resource fork is also preserved in the copy. If *`src-path`* refers to a link, the target of the link is copied, rather than the link itself.

- `(make-file-or-directory-link` *`to-path path`*`)` creates a link *`path`* to *`to-path`* under Unix and Mac OS X. The creation will fail if *`path`* already exists. The *`to-path`* need not refer to an existing file or directory, and *`to-path`* is not expanded before writing the link. If the link is created successfully, void is returned, otherwise the `exn:fail:filesystem` exception is raised. Under Windows, the `exn:fail:unsupported` exception is raised always.

11.3.4 Directories

The directory management utilities are:

- `(current-directory)` returns the current directory and `(current-directory` *`path`*`)` sets the current directory to *`path`*. This procedure is actually a parameter, as described in §7.9.1.1.

- `(current-drive)` returns the current drive name Windows. For other platforms, the `exn:fail:unsupported` exception is raised. The current drive is always the drive of the current directory.

- `(directory-exists?` *`path`*`)` returns `#t` if *`path`* refers to a directory, `#f` otherwise.

- `(make-directory` *`path`*`)` creates a new directory with the path *`path`*. If the directory is created successfully, void is returned, otherwise the `exn:fail:filesystem` exception is raised.

- `(delete-directory` *`path`*`)` deletes an existing directory with the path *`path`*. If the directory is deleted successfully, void is returned, otherwise the `exn:fail:filesystem` exception is raised.

- `(rename-file-or-directory` *`old-path new-path exists-ok?`*`)`, as described in the previous section, renames directories.

- `(file-or-directory-modify-seconds` *`path`*`)`, as described in the previous section, gets directory dates.

- `(file-or-directory-permissions` *`path`*`)`, as described in the previous section, gets directory permissions.

- `(directory-list` [*`path`*]`)` returns a list of all files and directories in the directory specified by *`path`*. If *`path`* is omitted, a list of files and directories in the current directory is returned. Under Unix and Mac OS X, an element of the list can start with period–slash–tilde ("./~") if it would otherwise start with tilde ("~"). Under Windows, an element of the list may start with \\**?****REL**\\.

- `(filesystem-root-list)` returns a list of all current root directories. Obtaining this list can be particularly slow under Windows.

11.4 Networking

MzScheme supports networking with the TCP and UDP protocols.

11.4.1 TCP

For information about TCP in general, see *TCP/IP Illustrated, Volume 1* by W. Richard Stevens.

- `(tcp-listen` *`port-k`* [*`max-allow-wait-k reuse? hostname-string-or-false`*]`)` creates a "listening" server on the local machine at the specified port number (where *`port-k`* is an exact integer between `1` and `65535` inclusive). The *`max-allow-wait-k`* argument determines the maximum number of client connections that can be waiting for acceptance. (When *`max-allow-wait-k`* clients are waiting acceptance, no new client connections can be made.) The default value for *`max-allow-wait-k`* argument is `4`.

  If the *`reuse?`* argument is true, then `tcp-listen` will create a listener even if the port is involved in a `TIME_WAIT` state. Such a use of *`reuse?`* defeats certain guarantees of the TCP protocol; see Stevens's book for details. Furthermore, on many modern platforms, a true value for *`reuse?`* overrides `TIME_WAIT` only if the listener was previously created with a true value for *`reuse?`*. The default for *`reuse?`* is `#f`.

  If *`hostname-string-or-false`* is `#f` (the default), then the listener accepts connections to all of the listening machine's addresses.[12] Otherwise, the listener accepts connections only at the interface(s) associated with the given hostname. For example, providing `"127.0.0.1"` as *`hostname-string-or-false`* creates a listener that accepts only connections to `"127.0.0.1"` (the loopback interface) from the local machine.

  The return value of `tcp-listen` is a TCP listener value. This value can be used in future calls to `tcp-accept`, `tcp-accept-ready?`, and `tcp-close`. Each new TCP listener value is placed into the management of the current custodian (see §9.2).

  If the server cannot be started by `tcp-listen`, the `exn:fail:network` exception is raised.

- `(tcp-connect` *`hostname-string port-k`* [*`local-hostname-string-or-false local-port-k-or-fal`* attempts to connect as a client to a listening server. The *`hostname-string`* argument is the server host's Internet address name[13] (e.g., `"www.plt-scheme.org"`), and *`port-k`* (an exact integer between `1` and `65535`) is the port where the server is listening.

  The optional *`local-hostname-string-or-false`* and *`local-port-k-or-false`* specify the client's address and port. If both are `#f` (the default), the client's address and port are selected automatically. If *`local-hostname-string-or-false`* is not `#f`, then *`local-port-k-or-false`* must be non-`#f`. If *`local-port-k-or-false`* is non-`#f` and *`local-hostname-string-or-false`* is `#f`, then the given port is used but the address is selected automatically.

  Two values (see §2.2) are returned by `tcp-connect`: an input port and an output port. Data can be received from the server through the input port and sent to the server through the output port. If the server is a MzScheme process, it can obtain ports to communicate to the client with `tcp-accept`. These ports are placed into the management of the current custodian (see §9.2).

  Initially, the returned input port is block-buffered, and the returned output port is block-buffered. Change the buffer mode using `file-stream-buffer-mode` (see §11.1.6).

  Both of the returned ports must be closed to terminate the TCP connection. When both ports are still open, closing the output port with `close-output-port` sends a TCP close to the server (which is seen as an end-of-file if the server reads the connection through a port). In contrast, `tcp-abandon-port` (see below) closes the output port, but does not send a TCP close until the input port is also closed.

  Note that the TCP protocol does not support a state where one end is willing to send but not read, nor does it include an automatic message when one end of a connection is fully closed. Instead, the other end of a

[12] MzScheme implements a listener with multiple sockets, if necessary, to accomodate multiple addresses with different protocol families. Under Linux, if *`hostname-string-or-false`* maps to both IPv4 and IPv6 addresses, then the behavior depends on whether IPv6 is supported and IPv6 sockets can be configured to listen to only IPv6 connections: if IPv6 is not supported or IPv6 sockets are not configurable, then the IPv6 addresses are ignored; otherwise, each IPv6 listener accepts only IPv6 connections.

[13] If *`hostname-string`* is associated with multiple addresses, they are tried one at a time until a connection succeeds. The name `"localhost"` generally specifies the local machine.

connection discovers that one end is fully closed only as a response to sending data; in particular, some number of writes on the still-open end may appear to succeed, though writes will eventually produce an error.

If a connection cannot be established by `tcp-connect`, the `exn:fail:network` exception is raised.

- `(tcp-connect/enable-break` *`hostname-string port-k`* `[`*`local-hostname-string local-port-k`*`])` is like `tcp-connect`, but breaking is enabled (see §6.7) while trying to connect. If breaking is disabled when `tcp-connect/enable-break` is called, then either ports are returned or the `exn:break` exception is raised, but not both.

- `(tcp-accept` *`tcp-listener`*`)` accepts a client connection for the server associated with *`tcp-listener`*. The *`tcp-listener`* argument is a TCP listener value returned by `tcp-listen`. If no client connection is waiting on the listening port, the call to `tcp-accept` will block. (See also `tcp-accept-ready?`, below.)

  Two values (see §2.2) are returned by `tcp-accept`: an input port and an output port. Data can be received from the client through the input port and sent to the client through the output port. These ports are placed into the management of the current custodian (see §9.2).

  In terms of buffering and connection states, the ports act the same as ports from `tcp-connect`.

  If a connection cannot be accepted by `tcp-accept`, or if the listener has been closed, the `exn:fail:network` exception is raised.

- `(tcp-accept-ready?` *`tcp-listener`*`)` tests whether an unaccepted client has connected to the server associated with *`tcp-listener`*. The *`tcp-listener`* argument is a TCP listener value returned by `tcp-listen`. If a client is waiting, the return value is `#t`, otherwise it is `#f`. A client is accepted with the `tcp-accept` procedure, which returns ports for communicating with the client and removes the client from the list of unaccepted clients.

  If the listener has been closed, the `exn:fail:network` exception is raised.

- `(tcp-accept/enable-break` *`tcp-listener`*`)` is like `tcp-accept`, but breaking is enabled (see §6.7) while trying to accept a connection. If breaking is disabled when `tcp-accept/enable-break` is called, then either ports are returned or the `exn:break` exception is raised, but not both.

- `(tcp-close` *`tcp-listener`*`)` shuts down the server associated with *`tcp-listener`*. The *`tcp-listener`* argument is a TCP listener value returned by `tcp-listen`. All unaccepted clients receive an end-of-file from the server; connections to accepted clients are unaffected.

  If the listener has already been closed, the `exn:fail:network` exception is raised.

  The listener's port number may not become immediately available for new listeners (with the default *`reuse?`* argument of `tcp-listen`). For further information, see Stevens's explanation of the `TIME_WAIT` TCP state.

- `(tcp-listener?` *`v`*`)` returns `#t` if *v* is a TCP listener value created by `tcp-listen`, `#f` otherwise.

- `(tcp-accept-evt` *`tcp-listener`*`)` returns a synchronizable event (see §7.7) that is in a blocking state when `tcp-accept` on *`tcp-listener`* would block. If the event is chosen in a synchronization, the result is a list of two items, which correspond to the two results of `tcp-accept`. (If the event is not chosen, no connections are accepted.)

- `(tcp-abandon-port` *`tcp-port`*`)` is like `close-output-port` or `close-input-port` (depending on whether *`tcp-port`* is an input or output port), but if *`tcp-port`* is an output port and its associated input port is not yet closed, then the other end of the TCP connection does not receive a TCP close message until the input port is also closed.[14]

[14]The TCP protocol does not include a "no longer reading" state on connections, so `tcp-abandon-port` is equivalent to `close-input-port` on input TCP ports.

- `(tcp-addresses` *`tcp-port`* `[`*`port-numbers?`*`])` returns two strings when *`port-numbers?`* is `#f` (the default). The first string is the Internet address for the local machine a viewed by the given TCP port's connection.[15] The second string is the Internet address for the other end of the connection.

  If *`port-numbers?`* is true, then four results are returned: a string for the local machine's address, an exact integer between `1` and `65535` for the local machine's port number, a string for the remote machine's address, and an exact integer between `1` and `65535` for the remote machine's port number.

  If the given port has been closed, the `exn:fail:network` exception is raised.

- `(tcp-port?` *`v`*`)` returns `#t` if *`v`* is a port returned by `tcp-accept`, `tcp-connect`, `tcp-accept/enable-break`, or `tcp-connect/enable-break`, `#f` otherwise.

11.4.2 UDP

For information about UDP in general, see *TCP/IP Illustrated, Volume 1* by W. Richard Stevens (which discusses UDP in addition to TCP).

- `(udp-open-socket [`*`family-hostname-string-or-false family-port-k-or-false`*`])` creates and returns a UDP socket to send and receive datagrams (broadcasting is allowed). Initially, the socket is not bound or connected to any address or port.

  If *`family-hostname-string-or-false`* or *`family-port-k-or-false`* is provided and not `#f`, then the socket's protocol family is determined from these arguments. The socket is *not* bound to the hostname or port number. For example, the arguments might be the hostname and port to which messages will be sent through the socket, which ensures that the socket's protocol family is consistent with the destination. Alternately, the arguments might be the same as for a future call to `udp-bind!`, which ensures that the socket's protocol family is consistent with the binding. If neither *`family-hostname-string-or-false`* nor *`family-port-k-or-false`* is provided as non-`#f`, then the socket's protocol family is IPv4.

- `(udp-bind!` *`udp-socket hostname-string-or-false port-k`*`)` binds an unbound *`udp-socket`* to the local port number *`port-k`* (an exact integer between `1` and `65535`). The result is always void.

  If *`hostname-string-or-false`* is `#f`, then the socket accepts connections to all of the listening machine's IP addresses at *`port-k`*. Otherwise, the socket accepts connections only at the IP address associated with the given name. For example, providing `"127.0.0.1"` as *`hostname-string-or-false`* typically creates a listener that accepts only connections to `"127.0.0.1"` from the local machine.

  A socket cannot receive datagrams until it is bound to a local address and port. If a socket is not bound before it is used with a sending procedure `udp-send`, `udp-send-to`, etc., the sending procedure binds the socket to a random local port. Similarly, if an event from `udp-send-evt` or `udp-send-to-evt` is chosen for a synchronization (see §7.7), the socket is bound; if the event is not chosen, the socket may or may not become bound. The binding of a bound socket cannot be changed.

  If *`udp-socket`* is already bound or closed, the `exn:fail:network` exception is raised.

- `(udp-connect!` *`udp-socket hostname-string-or-false port-k-or-false`*`)` connects the socket to the indicated remote address and port if *`hostname-string-or-false`* is a string and *`port-k-or-false`* is an exact integer between `1` and `65535`. The result is always void.

  If *`hostname-string-or-false`* is `#f`, then *`port-k-or-false`* also must be `#f`, and the port is disconnected (if connected). If one of *`hostname-string-or-false`* or *`port-k-or-false`* is `#f` and the other is not, the `exn:fail:contract` exception is raised.

  A connected socket can be used with `udp-send` (not `udp-send-to`), and it accepts datagrams only from the connected address and port. A socket need not be connected to receive datagrams. A socket can be connected, re-connected, and disconnected any number of times.

[15]For most machines, the answer corresponds to the current machine's only Internet address. But when a machine serves multiple addresses, the result is connection-specific.

If *udp-socket* is closed, the `exn:fail:network` exception is raised.

- `(udp-send-to` *udp-socket hostname-address port-k bytes* [*start-k end-k*]`)` sends `(subbytes` *bytes start-k end-k*`)` as a datagram from the unconnected *udp-socket* to the socket at the remote machine *hostname-address* on the port *port-k*. The *udp-socket* need not be bound or connected; if it is not bound, `udp-send-to` binds it to a random local port. If the socket's outgoing datagram queue is too full to support the send, `udp-send-to` blocks until the datagram can be queued. The result is always void.

  The optional *start-k* argument defaults to `0`, and the optional *end-k* argument defaults to the length of *bytes*. If *start-k* is greater than the length of *bytes*, or if *end-k* is less than *start-k* or greater than the length of *bytes*, the `exn:fail:contract` exception is raised.

  If *udp-socket* is closed or connected, the `exn:fail:network` exception is raised.

- `(udp-send` *udp-socket bytes* [*start-k end-k*]`)` is like `udp-send-to`, except that *udp-socket* must be connected, and the datagram goes to the connection target. If *udp-socket* is closed or unconnected, the `exn:fail:network` exception is raised.

- `(udp-send-to*` *udp-socket hostname-address port-k bytes* [*start-k end-k*]`)` is like `udp-send-to`, except that it never blocks; if the socket's outgoing queue is too full to support the send, `#f` is returned, otherwise the datagram is queued and the result is `#t`.

- `(udp-send*` *udp-socket bytes* [*start-k end-k*]`)` is like `udp-send`, except that (like `udp-send-to`) it never blocks and returns `#f` or `#t`.

- `(udp-send-to/enable-break` *udp-socket hostname-address port-k bytes* [*start-k end-k*]`)` is like `udp-send-to`, but breaking is enabled (see §6.7) while trying to send the datagram. If breaking is disabled when `udp-send-to/enable-break` is called, then either the datagram is sent or the `exn:break` exception is raised, but not both.

- `(udp-send/enable-break` *udp-socket bytes* [*start-k end-k*]`)` is like `udp-send`, except that breaks are enabled like `udp-send-to/enable-break`.

- `(udp-receive!` *udp-socket mutable-bytes* [*start-k end-k*]`)` accepts up to *end-k* − *start-k* bytes of *udp-socket*'s next incoming datagram into *mutable-bytes*, writing the datagram bytes starting at position *start-k* within *mutable-bytes*. The *udp-socket* must be bound to a local address and port (but need not be connected). If no incoming datagram is immediately available, `udp-receive!` blocks until one is available.

  Three values are returned: an exact integer that indicates the number of received bytes (between `0` and *end-k* − *start-k*), a hostname string indicating the source address of the datagram, and an exact integer between `1` and `65535` indicating the source port of the datagram. If the received datagram is longer than *end-k* − *start-k* bytes, the remainder is discarded.

  The optional *start-k* argument defaults to `0`, and the optional *end-k* argument defaults to the length of *mutable-bytes*. If *start-k* is greater than the length of *mutable-bytes*, or if *end-k* is less than *start-k* or greater than the length of *mutable-bytes*, the `exn:fail:contract` exception is raised.

- `(udp-receive!*` *udp-socket mutable-bytes* [*start-k end-k*]`)` is like `udp-receive!`, except that it never blocks. If no datagram is available, the three result values are all `#f`.

- `(udp-receive!/enable-break` *udp-socket mutable-bytes* [*start-k end-k*]`)` is like `udp-receive!`, but breaking is enabled (see §6.7) while trying to receive the datagram. If breaking is disabled when `udp-receive!/enable-break` is called, then either a datagram is received or the `exn:break` exception is raised, but not both.

- `(udp-close` *udp-socket*`)` closes *udp-socket*, discarding unreceived datagrams. If the socket is already closed, the `exn:fail:network` exception is raised.

- `(udp? v)` returns `#t` if *v* is a socket created by `udp-open-socket`, `#f` otherwise.
- `(udp-bound? udp-socket)` returns `#t` if *udp-socket* is bound to a local address and port, `#f` otherwise.
- `(udp-connected? udp-socket)` returns `#t` if *udp-socket* is connected to a remote address and port, `#f` otherwise.
- `(udp-send-ready-evt udp-socket)` returns a synchronizable event (see §7.7) that is in a blocking state when `udp-send-to` on *udp-socket* would block.
- `(udp-receive-ready-evt udp-socket)` returns a synchronizable event (see §7.7) that is in a blocking state when `udp-receive!` on *udp-socket* would block.
- `(udp-send-to-evt udp-socket hostname-address port-k bytes [start-k end-k])` returns a synchronizable event. The event is in a blocking state when `udp-send` on *udp-socket* would block. Otherwise, if the event is chosen in a synchronization, data is sent as for `(udp-send-to udp-socket hostname-address port-k bytes start-k end-k)`, and the synchronization result is void. (No bytes are sent if the event is not chosen.)
- `(udp-send-evt udp-socket bytes [start-k end-k])` is like `udp-send-to-evt`, except that *udp-socket* must be connected when the event is synchronized, and if the event is chosen in a synchronization, the datagram goes to the connection target. If *udp-socket* is closed or unconnected, the `exn:fail:network` exception is raised during a synchronization attempt.
- `(udp-receive!-evt udp-socket bytes [start-k end-k])` returns a synchronizable event. The event is in a blocking state when `udp-receive` on *udp-socket* would block. Otherwise, if the event is chosen in a synchronization, data is receive into *bytes* as for `(udp-receive! udp-socket bytes start-k end-k)`, and the synchronization result is a list of three values, corresponding to the three results from `udp-receive!`. (No bytes are received and the *bytes* content is not modified if the event is not chosen.)

12. Syntax and Macros

MzScheme supports the R^5RS `define-syntax`, `let-syntax`, and `letrec-syntax` forms with `syntax-rules`, with minor pattern and template extensions described in §12.1.

In addition to `syntax-rules`, MzScheme supports macros that perform arbitrary transformations on syntax. In particular, a *transformer expression* — the right-hand side of a `define-syntax`, `let-syntax`, or `letrec-syntax` binding — can be an arbitrary expression, and it is evaluated in a *transformer environment*. When the expression produces a procedure, it is associated as a syntax transformer to the identifier bound by `define-syntax`, `let-syntax`, or `letrec-syntax`. This more general, mostly hygienic macro system is based on `syntax-case` by Dybvig, Hieb, and Bruggeman (see "Syntactic abstraction in Scheme" in *Lisp and Symbolic Computation*, December 1993).

A transformer procedure consumes a syntax object and produces a new syntax object. A syntax object encodes S-expression structure, but also includes source-location information and lexical-binding information for each element within the S-expression. A syntax object is a first-class value, and it can exist at run-time. However, syntax objects are more typically used at syntax-expansion time — which is the run-time of a transformer procedure.[1]

Unlike traditional `defmacro` systems, MzScheme keeps the top-level transformer environment separate from the normal top-level environment. The environments are separated because the expressions in the different environments are evaluated at different times (transformer expressions are evaluated at syntax-expansion time, while normal expressions are evaluated at run time). Separating each environment ensures that compilation and analysis tools can process programs properly. See §12.3.3 for more information.

Also unlike traditional macro systems, a transformer procedure is invoked whenever its identifier is used in an expression position, not in application positions only. Even more generally, a transformer expression might not produce a procedure value, in which case the non-procedure is associated to its identifier as a generic expansion-time value. For example, a unit signature (see Chapter 55 of *PLT MzLib: Libraries Manual*) is associated to an identifier through an expansion-time value. See §12.6 for more information on transformer applications and expansion-time values.

12.1 `syntax-rules` Extensions

MzScheme extends the pattern language for `syntax-rules` so that a pattern of the form

```
(... pattern)
```

is equivalent to *`pattern`* where `...` is treated like any other identifier. Similarly, a template of the form

```
(... template)
```

is equivalent to *`template`* where `...` is treated like any other identifier.

In a pattern, additional patterns can follow `...`, but only one `...` can appear in a sequence:

```
(pattern ···¹ ... pattern ···)
```

[1] In general, modules and for-syntax imports create a hierarchy of run times and expansion times. See §12.3.4 for more information.

Furthermore, a sequence containing `...` can end with a dotted pair:

```
(pattern ···¹ ... pattern ··· . pattern)
```

but in this case, the final *`pattern`* is never matched to a syntactic list.

A template element consists of any number of `...`s after a template. For each `...` after the first one, the preceding element (with earlier `...`s) is conceptually wrapped with parentheses for generating output, and then wrapping parentheses in the output are removed. If a pattern identifier is followed by more ellipses in a template than in the pattern, then the pattern's match is expanded normally for inner ellipses (up to the number of ellipses that appear in the pattern), and then the match is replicated as necessary to satisfy outer ellipses.

To mesh gracefully with modules, literal identifiers are compared with `module-identifier=?`, which is equivalent to the comparison behavior of *R^5RS* in the absence of modules; see §12.3.1 for more information on identifier syntax comparisons.

Examples:

```
(define-syntax ex1
  (syntax-rules ()
   [(ex1 a) '(a (... ...))]))
(ex1 1) ; ⇒ '(1 ...)

(define-syntax ex2
  (syntax-rules ()
   [(ex2 a ... b) '(b a ...)]))
(ex2 1 2 3) ; ⇒ '(3 1 2)

(define-syntax ex3
  (syntax-rules ()
   [(ex3 a ... b . c) '(b a ... c)]))
(ex3 1 2 3 4) ; syntax error
(ex3 1 2 3 . 4) ; ⇒ '(3 1 2 4)

(define-syntax ex4
  (syntax-rules ()
   [(ex4 (a ...) ... b) '(a ... ... b)]))
(ex4 (1) (2 3) 4) ; ⇒ '(1 2 3 4)
```

The `syntax-id-rules` form has the same syntax as `syntax-rules`, except that each pattern is used in its entirety (instead of starting with a keyword placeholder that is ignored). Furthermore, when an identifier *`id`* is bound as syntax to a `syntax-id-rules` transformer, the transformer is applied whenever *`id`* appears in an expression position — not just when it is in the application position — or when *`id`* appears as the target of an assignment. When the identifier appears in an application position, `(`*`id expr`*` ···)`, the entire "application" is provided to the transformer, and when the identifier appears as a `set!` target, `(set! `*`id expr`*`)`, the entire `set!` expression is provided to the transformer; otherwise, the *`id`* is provided alone to the transformer. Typically, `set!` is included as a keyword in a `syntax-id-rules` use, and three patterns match the three possible uses of the identifier.

```
(define-syntax pwd
  ; For this macro to work, the set! case must
  ; be first, and the pwd case must be last
  (syntax-id-rules (set!)
    [(set! pwd expr) (current-directory expr)]
    [(pwd expr ...) ((current-directory) expr ...)]
    [pwd (current-directory)]))
```

```
(set! pwd "/tmp") ; sets current-directory parameter
pwd ; ⇒ "/tmp"
(current-directory) ; ⇒ "/tmp"
(current-directory "/usr/tmp")
pwd ; ⇒ "/usr/tmp"
```

12.2 Syntax Objects

`(read-syntax [source-name-v input-port])` is like `read`, except that it produces a syntax object with source-location information. The *`source-name-v`* is used as the source field of the syntax object; it can be an arbitrary value, but it should generally be a path for the source file. The default *`source-name-v`* is the input port's name (according to `object-name`; see §6.2.3). See §11.2.4 for more information about `read` and `read-syntax`, see §11.2.1.1 for information about port locations, and see §12.6.2 for information on the `'paren-shape` property and original-indicator property attached to a syntax object by `read-syntax`.

The result of `read-syntax` is a syntax object with source-location information, but no lexical information. Syntax objects acquire lexical information during expansion, so that by the time a transformer is called, the provided syntax object has lexical information.

The `eval`, `compile`, `expand`, `expand-once`, and `expand-to-top-form` procedures work on syntax objects, especially syntax objects with no lexical context. (If one of these procedures is given a non-syntax S-expression, the S-expression is converted to a syntax object containing no source information and no lexical context.) Each of these procedures adds context to the syntax object using `namespace-syntax-introduce` before expanding the syntax (but see §14.1 for information on the special handling of `module`). In contrast, the `eval-syntax`, `compile-syntax`, `expand-syntax`, `expand-syntax-once`, and `expand-syntax-to-top-form` procedures do not add context to a given syntax object before expanding.

The syntax object produced by `expand`, `expand-syntax`, etc. includes lexical information that influences future expansion and compilation of the syntax object. Thus, a syntax object produced by `read-syntax` should be passed to `eval` or `expand` (or another procedure without `-syntax` in its name), but a syntax object returned by `expand` should be passed to `eval-syntax` (or another procedure with `-syntax` in its name), since the result from `expand` has acquired a lexical context.

For example, if the following text is parsed by `read-syntax`,

```
(lambda (x) (+ x y))
```

the result is a syntax object that contains the S-expression structure `'(lambda (x) (+ x y))`, but also source information indicating that the first *x* is in column 9, etc. If `expand` is applied to the syntax object with a normal top-level environment, then the result will be a similar syntax object (with the source-location information intact), but the second *x* in the syntax object will have lexical information that ties it to the first *x*, and *y* in the syntax object will be annotated as a free variable. Even the syntax object's `'lambda` will have lexical information tying it to the built-in `lambda` form.

Compilation (often as a prelude to interactive evaluation) strips away source and context information as it processes a syntax object. The compilation of a `quote-syntax` form is an exception:

```
(quote-syntax datum)
```

The `quote-syntax` form produces a syntax object that preserves the source-location information for *`datum`*. It also encapsulates lexical-binding information accumulated by compilation in the `quote-syntax` expression's environment. A `quote-syntax` expression rarely appears in normal expressions; `quote-syntax` is more typically used within a transformer expression.

In addition to local and lexical information, a syntax object may have properties and certificates attached. Properties are added or inspected using `syntax-property`, as described in §12.6.2. Certificates validate references to identifiers that are not exported from a macro, as described in §12.6.3.

The `syntax-object->datum` procedure strips away location, lexical, property, and certificate information from a syntax object to produce a plain S-expression. The `datum->syntax-object` procedure wraps syntax information onto an S-expression, copying the source-location information of a given syntax object, the lexical information of another syntax object, and the properties of a third syntax object (where some or all three of the given objects can be the same). The `syntax-e` procedure unwraps only the immediate S-expression structure from a syntax object, leaving nested structure in place. These procedures are described in §12.2.2.

Although procedures such as `syntax-object->datum` permit arbitrary manipulation of syntax objects, a syntax transformer is more likely to use the pattern-matching `syntax-case` and `syntax` forms, which are described in the following subsection.

12.2.1 Syntax Patterns

The `syntax-case` form pattern-matches and deconstructs a syntax object:

```
(syntax-case stx-expr (literal-identifier ...)
  syntax-clause
  ···)

syntax-clause is one of
  (pattern expr)
  (pattern fender-expr expr)
```

If *`stx-expr`* expression does not produce a syntax object value, it is converted to one using `datum->syntax-object` with the lexical context of the expression (see §12.2.2). The syntax is then compared to the *`pattern`* in each *`syntax-clause`* until a match is found, and the result of the corresponding *`expr`* is the result of the `syntax-case` expression. If a *`syntax-clause`* contains a *`fender-expr`*, the clause matches only when both the *`pattern`* matches the syntax object and the *`fender-expr`* returns a true value. If no pattern matches, a "bad syntax" `exn:fail:syntax` exception is raised.

A *`pattern`* is nearly the same as a `syntax-rules` pattern (see *R^5RS*), with the ellipsis-escaping extension (see §12.1). The difference is that the first identifier in *`pattern`* is not ignored, unlike the leading keyword in a `syntax-rules` pattern.

As in `syntax-rules`, a non-literal identifier in a *`pattern`* is bound to a corresponding part of the syntax object within the clause's *`expr`* and optional *`fender-expr`*. The identifier cannot be used directly, however; a use of the identifier in an expression position is a syntax error. Instead, the identifier can be used only in `syntax` expressions within the binding's scope.

A `syntax` expression has the form

```
(syntax template)
```

where *`template`* is as in `syntax-rules` (extended, as usual, for escaped ellipses). The result of a `syntax` expression is a syntax object. Identifiers in the *`template`* that are bound by a `syntax-case` pattern are replaced with their bindings in the generated syntax object. A `syntax` expression that contains no pattern identifiers is equivalent to a `quote-syntax` expression, except that unlike `quote-syntax`, the `syntax` form always fails to compile (i.e., it loops forever) when *`template`* is cyclic.

The `syntax-rules` form can be expressed as a `syntax-case` form wrapped in `lambda`:

```
  (syntax-rules (literal-identifier ···)
    ((ignored-identifier . pattern) template)
    ···)
=expands=>
  (lambda (stx)
    (syntax-case stx (literal-identifier ···)
      ((generated-identifier . pattern) (syntax template))
      ···))
```

Note that implicit `lambda` of `syntax-rules` for the transformer procedure is made explicit with `syntax-case`. The `define-syntax` form supports `define`-style abbreviations for transformer procedures (see §2.8.1).

The following example shows one reason to use `syntax-case` instead of `syntax-rules`: custom error reporting.

```
(define-syntax (let1 stx)
  (syntax-case stx ()
    [(_ id val body)
     (begin
       ;; If id is not an identifier, report an error in terms of let1 instead of let:
       (unless (identifier? (syntax id))
         (raise-syntax-error #f "expected an identifier" stx (syntax id)))
       (syntax (let ([id val]) body)))]))
(let1 x 10 (add1 x)) ; ⇒ 11
(let1 2 10 (add1 x)) ; ⇒ let1: expected an identifier at: 2 in: (let1 2 10 (add1 x))
```

Another reason to use `syntax-case` is to implement "non-hygienic" macros that introduce capturing identifiers:

```
(define-syntax (if-it stx)
  (syntax-case stx ()
    [(src-if-it test then else)
     (syntax-case (datum->syntax-object (syntax src-if-it) 'it) ()
       [it (syntax (let ([it test]) (if it then else)))])]))
(if-it (memq 'b '(a b c)) it 'nope) ; ⇒ '(b c)
```

The nested `syntax-case` is used to bind the pattern variable *`it`*. The syntax for *`it`* is generated with `datum->syntax-object` using the context of *`src-if-it`*, which means that the introduced variable has the same lexical context as *`if-it`* at the macro's use; in other words, *`it`* acts as if it existed in the input syntax, so it can bind uses of *`it`* in *`test`*.

The `syntax-case*` form is a generalization of `syntax-case` where the procedure for comparing *`literal-identifier`*s is determined by a *`comparison-proc-expr`*:

```
(syntax-case* stx-expr (literal-identifier ...) comparison-proc-expr
  syntax-clause
  ···)
```

The result of *`comparison-proc-expr`* must be a procedure that accepts two arguments. The first argument is an identifier from *`stx-expr`*, and the second argument is an identifier from a *`syntax-clause`* pattern that is `module-identifier=?` to one of the *`literal-identifier`*s. A true result from the comparison procedure indicates that the first identifier matches the second.

12.2.1.1 Binding Pattern Variables

The `with-syntax` form is a `let`-like form for binding pattern variables:

```
(with-syntax ((pattern stx-expr)
              ···)
  expr)
```

The *pattern*s are matched the *stx-expr* values, and all pattern identifiers are bound in *expr*. The pattern identifiers across all *pattern*s must be distinct. If a *stx-expr* expression does not produce a syntax object, its result is converted using `datum->syntax-object` and the lexical context of the *stx-expr* (see §12.2.2). If the result of a *stx-expr* does not match its *pattern*, the `exn:fail:syntax` exception is raised.

The *if-it* example can be written more simply using `with-syntax`:

```
(define-syntax (if-it stx)
  (syntax-case stx ()
    [(src-if-it test then else)
     (with-syntax ([it (datum->syntax-object (syntax src-if-it) 'it)])
       (syntax (let ([it test]) (if it then else))))]))
```

Macros that expand to non-hygienic macros rarely work as intended. For example:

```
(define-syntax (cond-it stx)
  (syntax-case stx ()
    [(_ (test body) . rest)
     (syntax (if-it test body (cond-it . rest)))]
    [(_) (syntax (void))]))
(cond-it [(memq 'b '(a b c)) it] [#t 'nope]) ; ⇒ undefined variable it
```

The problem is that *cond-it* introduces *if-it* (hygienically), so *cond-it* effectively introduces *it* (hygienically), which doesn't bind *it* in the source use of *cond-it*. In general, the solution is to avoid macros that expand to uses of non-hygienic macros.[2]

12.2.1.2 Quasiquoting Templates

The `quasisyntax` form is like `syntax`, except with quasiquoting within the template:

```
(quasisyntax quasitemplate)
```

A *quasitemplate* is the same as a *template*, except that `unsyntax` and `unsyntax-splicing` escape to an expression:

```
(unsyntax expr)
(unsyntax-splicing expr)
```

The expression must produce a syntax object (or syntax list) to be substituted in place of the `unsyntax` or `unsyntax-splicing` form within the quasiquoting template, just like `unquote` and `unquote-splicing` within `quasiquote`. (If the escaped expression does not generate a syntax object, it is converted to one in the same was as for the right-hand sides of `with-syntax`.) Nested `quasisyntax`es introduce quasiquoting layers in the same way as nested `quasiquote`s.

Also analogous to `quote` and `quasiquote`, the reader converts `#'` to `syntax`, `` #` `` to `quasisyntax`, `#,` to `unsyntax`, and `#,@` to `unsyntax-splicing`. See also §11.2.4.

Example:

[2] In this particular case, Shriram Krishnamurthi points out changing *if-it* to use `(datum->syntax-object (syntax test) 'it)` solves the problem in a sensible way.

```
(with-syntax ([(v ...) (list 1 2 3)])
  #`(0 v ... #,(+ 2 2) #,@(list 5 6) 7)) ; ⇒ syntax for (0 1 2 3 4 5 6 7)
```

12.2.1.3 Assigning Source Location

The `syntax/loc` form is like `syntax`, except that the immediate resulting syntax object takes its source-location information from a supplied syntax object, unless the *`template`* is just a pattern variable:

```
(syntax/loc location-stx-expr template)
```

Use `syntax/loc` instead of `syntax` whenever possible to help tools that report source locations. For example, the earlier *`if-it`* example should have been written with `syntax/loc`:

```
(define-syntax (if-it stx)
  (syntax-case stx ()
   [(src-if-it test then else)
    (with-syntax ([it (datum->syntax-object (syntax src-if-it) 'it)])
     (syntax/loc stx (let ([it test]) (if it then else))))]))
```

The `quasisyntax/loc` form is the quasiquoting analogue of `syntax/loc`:

```
(quasisyntax/loc location-stx-expr template)
```

12.2.2 Syntax Object Content

`(syntax? v)` returns `#t` if *`v`* is a syntax object, `#f` otherwise.

`(syntax-source stx)` returns the source for the syntax object *`stx`*, or `#f` if none is known. The source is represented by an arbitrary value (e.g., one passed to `read-syntax`), but it is typically a file path string. See also §14.3.

`(syntax-line stx)` returns the line number (positive exact integer) for the start of the syntax object in its source, or `#f` if the line number or source is unknown. The result is `#f` if and only if `(syntax-column stx)` produces `#f`. See also §11.2.1.1 and §14.3.

`(syntax-column stx)` returns the column number (non-negative exact integer) for the start of the syntax object in its source, or `#f` if the source column is unknown. The result is `#f` if and only if `(syntax-line stx)` produces `#f`. See also §11.2.1.1 and §14.3.

`(syntax-position stx)` returns the character position (positive exact integer) for the start of the syntax object in its source, or `#f` if the source position is unknown. See also §11.2.1.1 and §14.3.

`(syntax-span stx)` returns the span (non-negative exact integer) in characters of the syntax object in its source, or `#f` if the span is unknown. See also §14.3.

`(syntax-original? stx)` returns `#t` if *`stx`* has the property that `read-syntax` and `read-honu-syntax` attach to the syntax objects that they generate (see §12.6.2), and if *`stx`*'s lexical information does not indicate that the object was introduced by a syntax transformer (see §12.3). The result is `#f` otherwise. This predicate can be used to distinguish syntax objects in an expanded expression that were directly present in the original expression, as opposed to syntax objects inserted by macros.

`(syntax-source-module stx)` returns a module path index or symbol (see §5.4.2) for the module whose source contains *`stx`*, or `#f` if *`stx`* has no source module.

`(syntax-e stx)` unwraps the immediate S-expression structure from a syntax object, leaving nested syntax structure (if any) in place. The result of `(syntax-e stx)` is one of the following:

- a symbol
- a syntax pair (described below)
- the empty list
- a vector containing syntax objects
- some other kind of datum, usually a number, boolean, or string

A *syntax pair* is a pair containing a syntax object as its first element, and either the empty list, a syntax pair, or a syntax object as its second element.

A syntax object that is the result of `read-syntax` reflects the use of dots (`.`) in the input by creating a syntax object for every pair of parentheses in the source, and by creating a pair-valued syntax object *only* for parentheses in the source. For example:

input	`read-syntax` result
`(a b)`	*`stx`*, where
	`(syntax-e stx)` is equivalent to `(list a-stx b-stx)`
	and `(syntax-e a-stx)` is equivalent to `'a`
	and `(syntax-e b-stx)` is equivalent to `'b`
`(a . (b))`	*`stx`*, where
	`(syntax-e stx)` is equivalent to `(cons a-stx sb-stx)`
	and `(syntax-e a-stx)` is equivalent to `'a`
	and `(syntax-e sb-stx)` is equivalent to `(list b-stx)`
	and `(syntax-e b-stx)` is equivalent to `'b`

`(syntax->list stx)` returns an immutable list of syntax objects or `#f`. The result is a list of syntax objects when `(syntax-object->datum stx)` would produce a list. In other words, syntax pairs in `(syntax-e stx)` are flattened.

`(syntax-object->datum stx)` returns an S-expression by stripping the syntactic information from *`stx`*. Graph structure is preserved by the conversion.

`(datum->syntax-object ctxt-stx v [src-stx-or-list prop-stx cert-stx])` converts the S-expression *`v`* to a syntax object, using syntax objects already in *`v`* in the result. Converted objects in *`v`* are given the lexical context information of *`ctxt-stx`* and the source-location information of *`src-stx-or-list`*. If *`v`* is not already a syntax object, then the resulting immediate syntax object it is given the properties (see §12.6.2) of *`prop-stx`* and the inactive certificates (see §12.6.3) of *`cert-stx`*. Any of *`ctxt-stx`*, *`src-stx-or-list`*, *`prop-stx`*, or *`cert-stx`* can be `#f`, in which case the resulting syntax has no lexical context, source information, new properties, and/or certificates.

If *`src-stx-or-list`* is not `#f` or a syntax object, it must be a list of five elements:

```
(list source-name-v line-k column-k position-k span-k)
```

where *`source-name-v`* is an arbitrary value for the source name; *`line-k`* is a positive, exact integer for the source line, or `#f`; and *`column-k`* is a non-negative, exact integer for the source column, or `#f`; *`position-k`* is a positive, exact integer for the source position, or `#f`; and *`span-k`* is a non-negative, exact integer for the source span, or `#f`. The *`line-k`* and *`column-k`* values must both be numbers or both be `#f`, otherwise the `exn:fail` exception is raised.

Graph structure is preserved by the conversion of *v* to a syntax object, but graph structure that is distributed among distinct syntax objects in *v* may be hidden from future applications of `syntax-object->datum` and `syntax-graph?` to the new syntax object.

`(syntax-graph? stx)` returns `#t` if *stx* might be preservably shared within a syntax object created by `read-syntax`, `read-honu-syntax`, or `datum->syntax-object`. In general, sharing detection is approximate—`datum->syntax-object` can construct syntax objects with sharing that is hidden from `syntax-graph?`—but `syntax-graph?` reliably returns `#t` for at least one syntax object in a cyclic structure. Meanwhile, deconstructing a syntax object with procedures such as `syntax-e` and comparing the results with `eq?` can also fail to detect sharing (even cycles), due to the way lexical information is lazily propagated; only `syntax-object->datum` reliably exposes sharing in a way that can be detected with `eq?`.

`(identifier? v)` returns `#t` if *v* is a syntax object and `(syntax-e stx)` produces a symbol.

`(generate-temporaries stx-pair)` returns a list of identifiers that are distinct from all other identifiers. The list contains as many identifiers as *stx-pair* contains elements. The *stx-pair* argument must be a syntax pair that can be flattened into a list. The elements of *stx-pair* can be anything, but string, symbol, and identifier elements will be embedded in the corresponding generated name (useful for debugging purposes). The generated identifiers are built with interned symbols (not `gensyms`), so the limitations described in §14.3 do not apply.

12.3 Syntax and Lexical Scope

Hygienic macro expansion depends on information associated with each syntax object that records the lexical context of the site where the syntax object is introduced. This information includes the identifiers that are bound by `lambda`, `let`, `letrec`, etc., at the syntax object's introduction site, the `require`d identifiers at the introduction site, and the macro expansion that introduces the object.

Based on this information, a particular identifier syntax object falls into one of three classifications:

- *lexical* — the identifier is bound by `lambda`, `let`, `letrec`, or some other form besides `module` or a top-level definition.
- *module-imported* — the identifier is bound through a `require` declaration or a top-level definition within `module`.
- *free* — the identifier is not bound (and therefore refers to a top-level definition, if the identifier is not within a module).

The `identifier-binding` procedure (described in §12.3.2) reports an identifiers classification. Further information about a lexical identifier is available only in relative terms, such as whether two identifiers refer to the same binding (see `bound-identifier=?` in §12.3.1). For module-imported identifiers, information about the module source is available.

In a freshly read syntax object, identifiers have no lexical information, so they are all classified as free. During expansion, some identifiers acquire lexical or module-import classifications. An identifier that becomes classified as lexical will remain so classified, though its binding might shift as expansion proceeds (i.e., as nested binding expressions are parsed, and as macro introductions are tracked). An identifier classified as module-imported might similarly shift to the lexical classification, but if it remains module-imported, its source-module designation will never change.

Lexical information is used to expand and parse syntax in a way that it obeys lexical and module scopes. In addition, an identifier's lexical information encompasses a second dimension, which distinguishes the environment of normal expressions from the environment of transformer expressions. The module bindings of each environment can be different, so an identifier may be classified differently depending on whether it is ultimately used in a normal expression or in a transformer expression. See §12.3.3 and §12.3.4 for more information on the two environments.

12.3.1 Syntax Object Comparisons

`(bound-identifier=? a-id-stx b-id-stx)` returns `#t` if the identifier *a-id-stx* would bind *b-id-stx* (or vice-versa) if the identifiers were substituted in a suitable expression context, `#f` otherwise.

`(free-identifier=? a-id-stx b-id-stx)` returns `#t` if *a-id-stx* and *b-id-stx* access the same lexical, module, or top-level binding and return the same result for `syntax-e`, `#f` otherwise.

`(module-identifier=? a-id-stx b-id-stx)` returns `#t` if *a-id-stx* and *b-id-stx* access the same lexical, module, or top-level binding in the normal environment. "Same module binding" means that the identifiers refer to the same original definition site, not necessarily the `require` or `provide` site. Due to renaming in `require` and `provide`, the identifiers may return distinct results with `syntax-e`.

`(module-transformer-identifier=? a-id-stx b-id-stx)` returns `#t` if *a-id-stx* and *b-id-stx* access the same lexical, module, or top-level binding in the identifiers' transformer environments (see §12.3.3).

`(module-template-identifier=? a-id-stx b-id-stx)` returns `#t` if *a-id-stx* and *b-id-stx* access the same lexical or module binding in the identifiers' template environments (see §12.3.4).

`(check-duplicate-identifier id-stx-list)` compares each identifier in *id-stx-list* with every other identifier in the list with `bound-identifier=?`. If any comparison returns `#t`, one of the duplicate identifiers is returned (the first one in *id-stx-list* that is a duplicate), otherwise the result is `#f`.

12.3.2 Syntax Object Bindings

`(identifier-binding id-stx)` returns one of three kinds of values, depending on the binding of *id-stx* in its normal environment:

- The result is `'lexical` if *id-stx* is bound in its context to anything other than a top-level variable or a module variable.
- The result is a list of five items when *id-stx* is bound in its context to a module-defined identifier: `(list source-mod source-id nominal-source-mod nominal-source-id et?)`.
  - *source-mod* is a module path index or symbol (see §5.4.2) that indicates the defining module.
  - *source-id* is a symbol for the identifier's name at its definition site in the source module. This can be different from the local name returned by `syntax-object->datum` for several reasons: the identifier is renamed on import, it is renamed on export, or it is implicitly renamed because the identifier (or its import) was generated by a macro invocation.
  - *nominal-source-mod* is a module path index or symbol (see §5.4.2) that indicates the module `require`d into the context of *id-stx* to provide its binding. It can be different from *source-mod* due to a re-export in *nominal-source-mod* of some imported identifier.
  - *nominal-source-id* is a symbol for the identifier's name as exported by *nominal-source-mod*. It can be different from *source-id* due to a renaming `provide`, even if *source-mod* and *nominal-source-mod* are the same.
  - *et?* is `#t` if the source definition is for-syntax, `#f` otherwise.
- The result is `#f` if *id-stx* is not bound (or bound only to a top-level variable) in its lexical context.

`(identifier-transformer-binding id-stx)` is like `identifier-binding`, except that the reported information is for the identifier's bindings in the transformer environment (see §12.3.3), instead of the normal environment. If the result is `'lexical` for either of `identifier-binding` or `identifier-transformer-binding`, then the result is always `'lexical` for both.

`(identifier-template-binding` *`id-stx`*`)` is like `identifier-binding`, except that the reported information is for the identifier's bindings in the template environment (see §12.3.4), instead of the normal environment. If the result is `'lexical` for either of `identifier-binding` or *`identifier-template-binding`*, then the result is always `'lexical` for both.

`(identifier-binding-export-position` *`id-stx`*`)` returns either `#f` or an exact non-negative integer. It returns an integer only when `identifier-binding` returns a list, when *`id-stx`* represents an imported binding, and when the source module assigns internal positions for its definitions. This function is intended for use by **mzc**.

`(identifier-transformer-binding-export-position` *`id-stx`*`)` is like `identifier-binding-export-posit`
except that the reported information is for the transformer environment. This function is intended for use by **mzc**.

12.3.3 Transformer Environments

The top-level environment for transformer expressions is separate from the normal top-level environment. Consequently, top-level definitions are not available for use in top-level transformer definitions. For example, the following program does not work:

```
(define count 0)
(define (inc!) (set! count (add1 count)))
(define-syntax (let1 stx)
  (syntax-case stx ()
    [(_ x v b)
     (begin
       (printf "expanding ~a~n" count) ; DOESN'T WORK
       (inc!)                          ; ALSO DOESN'T WORK
       (syntax (let ([x v]) b)))]))
(let1 x 2 (add1 x))
```

The variables *`count`* and *`inc!`* are bound in the normal top-level environment, but it is not bound in the transformer environment, so the attempt to expand `(`*`let1 x`*` 2 (add1 `*`x`*`))` will result in an undefined-variable error.

In the same way that `define` binds only in the normal environment, a `require` expression imports only into the normal environment, and the imported bindings are not made visible in the transformer environment. A top-level `require-for-syntax` imports into the transformer environment without affecting the normal environment. Furthermore, the `require` and `require-for-syntax` forms create separate instantiations of any module that is imported into both environments, in keeping with the separation of the environments.

The initial namespace created by the stand-alone MzScheme application imports all of MzScheme's built-in syntax, procedures, and constants into the transformer environment.[3] To extend this environment, use one of the following:

- `define-for-syntax`, which is like `define`, but binds in the transformer environment. The body of the definition is also evaluated in the transformer environment. The `define-values-for-syntaxes` form is the multiple-values variant of `define-for-syntax`. Within a module, `define-for-syntax` or `define-values-for-syntaxes` binds identifiers for unquoted expressions only after the definition (plus in the right-hand side of the definition itself); in particular, mutually-referential for-syntax definitions in a module must be defined with a single `define-values-for-syntaxes`.

[3]In contrast, a namespace created by `(scheme-report-environment 5)` imports only `syntax-rules` into the transformer environment.

- `begin-for-syntax`, which is like `begin`, but its body is evaluated in the transformer environment. Furthermore, `define`, `define-values`, `require`, and `require-for-template` declarations are treated like `define-for-syntax`, `define-values-for-syntax`, `require-for-syntax`, and `require` declarations, respectively.
- `require-for-syntax`, to import bindings into the transformer environment.

In particular, the example above can be repairs by replacing

```
(define count 0)
(define (inc!) (set! count (add1 count)))
```

with either

```
(define-for-syntax count 0)
(define-for-syntax (inc!) (set! count (add1 count)))
```

or

```
(begin-for-syntax
 (define count 0)
 (define (inc!) (set! count (add1 count))))
```

or

```
(module counter mzscheme
  (define count 0)
  (define (inc!) (set! count (add1 count)))
  (provide count inc!))
(require-for-syntax counter)
```

When an identifier binding is introduced by a form other than `module` or a top-level definition, it extends the environment for both normal and transformer expressions within its scope, but the binding is only accessible by expressions resolved in the proper environment (i.e., the one in which it was introduced). In particular, a transformer expression in a `let-syntax` or `letrec-syntax` expression cannot access identifiers bound by enclosing forms, and an identifier bound in a transformer expression should not appear as an expression in the result of the transformer. Such out-of-context uses of an identifier are flagged as syntax errors when attempting to resolve the identifier.

A `let-syntax` or `letrec-syntax` expression can never usefully appear as a transformer expression, because MzScheme provides no mechanism for importing into the meta-transformer environment that would be used by meta-transformer expressions to operate on transformer expressions. In other words, an expression of the form

```
(let-syntax ([identifier (let-syntax ([identifier expr])
                                     body-expr)])
  ...)
```

is always illegal, assuming that `let-syntax` is bound in both the normal and transformer environments to the `let-syntax` of `mzscheme`. No syntax (not even function application) is bound in *`expr`*'s environment. This restriction in the `mzscheme` language is of little consequence, however, since for-syntax exports allow the definition of syntax applicable to the above *`body-expr`*.

12.3.4 Module Environments

In the same way that the normal and transformer environments are kept separate at the top level, a module's normal and transformer environments are also separated. Normal imports and definitions in a module — both variable and syntax — contribute to the module's normal environment, only.

For example, the module expression

```
(module m mzscheme
  (define (id x) x)
  (define-syntax (macro stx)
    (id (syntax (printf "hi~n")))))
```

is ill-formed because `id` is not bound in the transformer environment for the `macro` implementation. To make `id` usable from the transformer, the body of the module *m* would have to be executed — which is impossible in general, because a syntax definition such as `macro` affects the expansion of the rest of the module body.

Consequently, if a procedure such as `id` is to be used in a transformer, it must either remain local to the transformer expression, or reside in a different module. For example, the above module is trivially repaired as

```
(module m mzscheme
  (define-syntax macro
    (let ([id (lambda (x) x)])
      (lambda (stx)
        (id (syntax (printf "hi~n")))))))
```

The `define-for-syntax`, `begin-for-syntax`, and `define-syntaxes` forms (see §12.3.3 and §12.4) are useful for defining multiple macros that share helper functions.

In the `mzscheme` language, the base environment for a transformer expression includes all of MzScheme. The `mzscheme` language also provides a `require-for-syntax` form (in the normal environment) for importing bindings from another module into the importing module's transformer environment:

```
(require-for-syntax require-spec ···)
```

A for-syntax import of *M* within *N* causes *M* to be executed at *N*'s expansion time, instead of (or possibly in addition to) run time for *N*. The syntax and variable identifiers exported by the for-syntax module are visible within the module's transformer environment, but not its normal environment. Like a normal expression, a transformer expression in a module cannot contain free variables.

Finally, `mzscheme` provides the `require-for-template` form, which is roughly dual to `require-for-syntax`:

```
(require-for-template require-spec ···)
```

A for-template import of *M* within *N* causes the referenced module to be executed at the run-time of any *P* that includes a for-syntax import of *N*. In other words, `require-for-template` introduces bindings that become available in a future run time.

Transformer expressions and imports for a module *M* are executed once each time a module is expanded using *M*'s syntax bindings or using *M* as a for-syntax import. After the module is expanded, its transformer environment is destroyed, including bindings from modules used at expansion time.

Example:

```
(module rt mzscheme
  (printf "RT here~n")
  (define mx (lambda () 7))
  (provide mx))

(module tt mzscheme
  (printf "RT here, too~n")
  (define x 700)
  (provide x))
```

```
(module et mzscheme
  (require-for-template tt)
  (printf "ET here~n")
  ;; The x below is future-time:
  (define mx (lambda () (syntax x)))
  (provide mx))

(module m mzscheme
  (require-for-syntax mzscheme)
  (require rt)                ; rt provides run-time mx
  (require-for-syntax et)     ; et provides exp-time mx

  ;; The mx below is run-time:
  (printf "~a~n" (mx))        ; prints 7 when run

  ;; The mx below is exp-time:
  (define-syntax onem (lambda (stx) (mx)))
  (printf "~a~n" (onem))     ; prints 700 when run

  ;; The mx below is run-time:
  (define-syntax twom (lambda (stx) (syntax (mx))))
  (printf "~a~n" (twom)))     ; prints 7 when run

;; "ET here" is printed during the expansion of m

(require m) ; prints "ET here" (for later macro expansion in the top level, if any)
            ; and "RT here, too" and "RT here" in some order,
            ; then 7, then 700, then 7
```

This expansion-time execution model explains the need to execute declared modules only when they are invoked. If a declared module is imported into other modules only for syntax, then the module is needed only at expansion time and can be ignored at run time. The separation of declaration and execution also allows a for-syntax module to be executed once for each module that it expands through `require-for-syntax`.

The hierarchy of run times avoids confusion among expansion and executing layers that can prevent separate compilation. By ensuring that the layers are separate, a compiler or programming environment can expand, partially expand, or re-expand a module without affecting the module's run-time behavior, whether the module is currently executing or not.

Since transformer expressions may themselves use macros defined by modules with for-syntax imports (to implement the macros), expansion of a module creates a hierarchy of run times (or ”tower of expanders”). The expansion time of each layer corresponds to the run time of the next deeper layer.

In the absence of `let-syntax` and `letrec-syntax`, the hierarchy of run times would be limited to three levels, since the transformer expressions for run-time imports would have been expanded before the importing module must be expanded. The `let-syntax` and `letrec-syntax` forms, however, allow syntax visible in a for-syntax import's transformers to appear in the expansion of transformer expressions in the module. Consequently, the hierarchy is bounded in principle only by the number of declared modules. In practice, the hierarchy will rarely exceed a few levels.

12.3.5 Macro-Generated Top-Level and Module Definitions

When a top-level definition binds an identifier that originates from a macro expansion, the definition captures only uses of the identifier that are generated by the same expansion. This behavior is consistent with internal definitions (see §2.8.5), where the defined identifier turns into a fresh lexical binding.

Example:

```
(define-syntax def-and-use-of-x
  (syntax-rules ()
    [(def-and-use-of-x val)
     ; x below originates from this macro:
     (begin (define x val) x)]))
(define x 1)
x ; ⇒ 1
(def-and-use-of-x 2) ; ⇒ 2
x ; ⇒ 1

(define-syntax def-and-use
  (syntax-rules ()
    [(def-and-use x val)
     ; x below was provided by the macro use:
     (begin (define x val) x)]))
(def-and-use x 3) ; ⇒ 3
x ; ⇒ 3
```

For a top-level definition (outside of `module`), the order of evaluation affects the binding of a generated definition for a generated identifier use. If the use precedes the definition, then the use refers to a non-generated binding, just as if the generated definition were not present. (No such dependency on order occurs within a `module`, since a module binding covers the entire module body.) To support the declaration of an identifier before its use, the `define-syntaxes` form avoids binding an identifier if the body of the `define-syntaxes` declaration produces zero results (see also §12.4).

Example:

```
(define bucket-1 0)
(define bucket-2 0)
(define-syntax def-and-set!-use-of-x
  (syntax-rules ()
    [(def-and-set!-use-of-x val)
     (begin (set! bucket-1 x) (define x val) (set! bucket-2 x))]))
(define x 1)
(def-and-set!-use-of-x 2)
x ; ⇒ 1
bucket-1 ; ⇒ 1
bucket-2 ; ⇒ 2

(define-syntax defs-and-uses/fail
  (syntax-rules ()
    [(def-and-use)
     (begin
      ; Initial reference to even precedes definition:
      (define (odd x) (if (zero? x) #f (even (sub1 x))))
      (define (even x) (if (zero? x) #t (odd (sub1 x))))
      (odd 17))]))
(defs-and-uses/fail) ; ⇒ error: undefined identifier even

(define-syntax defs-and-uses
  (syntax-rules ()
    [(def-and-use)
     (begin
      ; Declare before definition via no-values define-syntaxes:
      (define-syntaxes (odd even) (values))
      (define (odd x) (if (zero? x) #f (even (sub1 x))))
      (define (even x) (if (zero? x) #t (odd (sub1 x))))
      (odd 17))]))
(defs-and-uses) ; ⇒ #t
```

Within a module, macro-generated `require` and `provide` clauses also introduce and reference generation-specific bindings:

- In `require` (see §5.2), for a *`require-spec`* of the form `(rename` *`local-identifier exported-identifier`*`)`, the *`local-identifier`* is bound only for uses of the identifier generated by the same macro expansion as *`local-identifier`*. In `require` for other *`require-spec`*s, the generator of the *`require-spec`* determines the scope of the bindings.
- In `provide` (see §5.2), for a *`provide-spec`* of the form *`identifier`*, the exported identifier is the one that binds *`identifier`* within the module in a generator-specific way, but the external name is the plain *`identifier`*. The exceptions for `all-from-except` and `all-defined-except` are similarly determined in a generator-specific way, as is the *`local-identifier`* binding of a `rename` form, but plain identifiers are used for the external names. For `struct`, the context of the *`struct-identifier`* determines local bindings for all of the expanded `struct` names. For `all-defined` and `all-defined-except`, only identifiers with definitions having the same generator as the `all-defined` or `all-defined-except` keyword are exported; the external name is the plain identifier from the definition. The generator of an `all-from` or `all-from-except` *`provide-spec`* does not affect the set identifiers exported by the *`provide-spec`*.

12.4 Binding Multiple Syntax Identifiers

In addition to `define-syntax`, `let-syntax`, and `letrec-syntax`, MzScheme provides `define-syntaxes`, `let-syntaxes`, and `letrec-syntaxes`. These forms are analogous to `define-values`, `let-values`, and `letrec-values`, allowing multiple syntax bindings at once (see §2.8).

```
  (define-syntaxes (identifier ···) expr)

  (let-syntaxes (((identifier ···) expr)
                 ···)
     expr ···1)

  (letrec-syntaxes (((identifier ···) expr)
                    ···)
     expr ···1)
```

At the top level, `define-syntaxes` accepts zero results for any number of *`identifier`*s, and in that case, it neither binds the identifiers nor signals an error. This behavior is useful for *`identifier`*s that are introduced by a macro that produces top-level `define`s. See §12.3.5 for more information.

MzScheme also provides a `letrec-syntaxes+values` form for binding both values and syntax in a single, mutually recursive scope:

```
  (letrec-syntaxes+values (((identifier ···) expr) ···)
                          (((identifier ···) expr) ···)
     expr ···1)
```

The first set of bindings are syntax bindings (as in `letrec-syntaxes`), and the second set of bindings are normal variable bindings (as in `letrec-values`).

Examples:

```
;; Defines let/cc and let-current-continuation as the same macro:
(define-syntaxes (let/cc let-current-continuation)
  (let ([macro (syntax-rules ()
                 [(_ id body1 body ...)
                  (call/cc (lambda (id) body1 body ...))])])
    (values macro macro)))

(letrec-syntaxes+values ([(get-id) (syntax-rules ()
                                     [(_) id])])
                        ([(id) (lambda (x) x)]
                         [(x) (get-id)])
   x) ; ⇒ the id identify procedure
```

12.5 Special Syntax Identifiers

To enable the definition of syntax transformers for application forms and other data (numbers, vectors, etc.), the syntax expander treats `#%app`, `#%top`, and `#%datum` as special identifiers.

Any expandable expression of the form

```
(datum . datum)
```

where the first *datum* is not an identifier bound to an expansion-time value, is treated as

```
(#%app datum . datum)
```

so that the syntax transformer bound to `#%app` is applied. In addition, `()` is treated as `(#%app)`. Similarly, an expression

```
identifier
```

where *identifier* has no binding other than a top-level binding, is treated as

```
(#%top . identifier)
```

Finally, an expression

```
datum
```

where *datum* is not an identifier or pair, is treated as

```
(#%datum . datum)
```

The `mzscheme` module provides `#%app`, `#%top`, and `#%datum` as regular application, top-level variable reference, and implicit quote, respectively. A module can export different transformers with these names to support languages different from conventional Scheme.

In the case of `read-eval-print-loop` or the default load handler, every input *datum* is wrapped with `#%top-interaction`:

```
(#%top-interaction . datum)
```

The `mzscheme` module provides `#%top-interaction` as a macro that expands to just the *`datum`*.

Within `module`, `#%module-begin` is used as a transformer for the module body. A `#%module-begin` is implicitly added around a module body when it contains multiple S-expressions, or when the S-expression expands to a core form other than `#%module-begin` or `#%plain-module-begin`; the lexical context for the introduced `#%module-begin` identifier includes only the exports of the module's initial import. After such wrapping, if any, and before any expansion, an `'enclosing-module-name` property is attached to the module-body syntax object; the property's value is a symbol for the module name as specified after the `module` keyword.

The `mzscheme` module binds `#%module-begin` to a form that inserts a for-syntax import of `mzscheme`, so that `mzscheme` bindings can be used in syntax definitions. It also exports `#%plain-module-begin`, which can be substituted for `#%module-begin` to avoid the for-syntax import of `mzscheme`. Any other transformer used for `#%module-begin` must expand to `mzscheme`'s `#%module-begin` or `#%plain-module-begin`.

When an expression is fully expanded, all applications, top-level variable references, and literal datum expressions will appear as explicit `#%app`, `#%top`, and `#%datum` forms, respectively. Those forms can also be used directly by source code. The `#%module-begin` form can never usefully appear in an expression, and the body of a fully expanded `module` declaration is not wrapped with `#%module-begin`; instead, it is wrapped with `#%plain-module-begin`.

The following example shows how the special syntax identifiers can be defined to create a non-Scheme module language:

```
(module lambda-calculus mzscheme

  ; Restrict lambda to one argument:
  (define-syntax lc-lambda
    (syntax-rules ()
      [(_ (x) E) (lambda (x) E)]))

  ; Restrict application to two expressions:
  (define-syntax lc-app
    (syntax-rules ()
      [(_ E1 E2) (E1 E2)]))

  ; Restrict a lambda calculus module to one body expression:
  (define-syntax lc-module-begin
    (syntax-rules ()
      [(_ E) (#%module-begin E)]))

  ; Disallow numbers, vectors, etc.
  (define-syntax lc-datum
    (syntax-rules ()))

  ; Provide (with renaming):
  (provide #%top ; keep mzscheme's free-variable error
           (rename lc-lambda lambda)
           (rename lc-app #%app)
           (rename lc-module-begin #%module-begin)
           (rename lc-datum #%datum)))

(module m lambda-calculus
 ; The only syntax defined by lambda-calculus is
 ; unary lambda, unary application, and variables.
 ; Also, the module must contain exactly one expression.
```

```
   ((lambda (y) (y y))
    (lambda (y) (y y))))

(require m)     ; executes m, loops forever
```

12.6 Macro Expansion

A `define-syntax`, `let-syntax`, or `letrec-syntax` form associates an identifier to an expansion-time value. If the expansion-time value is a procedure of one argument, then the procedure is applied by the syntax expander when the identifier is used in the scope of the syntax binding.

The transformer for an *`identifier`* is applied whenever the *`identifier`* appears in an expression position — not just when it appears after a parenthesis as `(`*`identifier`*` ...)`. When it does appear as `(`*`identifier`*` ...)`, the entire `(`*`identifier`*` ...)` expression is provided as the argument to the transformer. Otherwise only *`identifier`* is provided to the transformer.

A typical transformer is implemented as

```
(lambda (stx)
  (syntax-case stx ()
    [(_ rest-of-pattern) expr]))
```

so that *`identifier`* by itself does not match the pattern; thus, the `exn:fail:syntax` exception is raised when *`identifier`* does not appear as `(`*`identifier`*` ...)`.

`(make-set!-transformer `*`proc`*`)` also creates a transformer procedure. The *`proc`* argument must be a procedure of one argument; if the result of `(make-set!-transformer `*`proc`*`)` is bound as syntax to *`identifier`*, then *`proc`* is applied as a transformer when *`identifier`* is used in an expression position, or when it is used as the target of a `set!` assignment: `(set! `*`identifier expr`*`)`. When the identifier appears as a `set!` target, the entire `set!` expression is provided to the transformer.

Example:

```
(let ([x 1]
      [y 2])
  (let-syntax ([x (make-set!-transformer
                   (lambda (stx)
                    (syntax-case stx (set!)
                      ; Redirect mutation of x to y
                      [(set! id v) (syntax (set! y v))])))]
                      ; Normal use of x really gets x
                      [id (identifier? (syntax id)) (syntax x)])))])
    (begin
      (set! x 3)
      (list x y)))) ; ⇒ '(1 3)
```

`(set!-transformer? `*`v`*`)` returns `#t` if *`v`* is a value created by `make-set!-transformer`, `#f` otherwise.

`(set!-transformer-procedure `*`transformer`*`)` returns the procedure passed to `make-set!-transformer` to create *`transformer`*.

`(make-rename-transformer `*`id-stx`*`)` creates a transformer procedure that inserts the identifier *`id-stx`* in place of whatever identifier binds the transformer, including in non-application positions, and in `set!` expressions. Such a transformer could be written manually, but the one created by `make-rename-transformer` cooperates specially with `syntax-local-value` (see below).

`(rename-transformer? v)` returns `#t` if *v* is a value created by `make-rename-transformer`, `#f` otherwise.

`(rename-transformer-target transformer)` returns the identifier passed to `make-rename-transformer` to create *transformer*.

If a transformer expression produces a non-procedure value, the value is associated with the identifier as a generic expansion-time value. Any use of the identifier in an expression position is rejected as a syntax error, but syntax transformers can access the value. For example, the `define-signature` form (see Chapter 55 of *PLT MzLib: Libraries Manual*) associates a component interface description to the defined identifier.

When a syntax transformer is applied, it can query the bindings of identifiers in the lexical environment of the expression being transformed. For example, the `unit` form can access a named interface description with `syntax-local-value`:

- `(syntax-local-value id-stx [failure-thunk])` returns the expansion-time value of *id-stx* in the transformed expression's context. If *id-stx* is not bound to an expansion-time value (via `define-syntax`, `let-syntax`, etc.) in the environment of the expression being transformed, the result is obtained by applying *failure-thunk*. If *failure-thunk* is not provided, the `exn:fail:contract` exception is raised. If *id-stx* is bound to a rename transformer created with `make-rename-transformer`, `syntax-local-value` effectively calls itself with the target of the rename and returns that result, instead of the rename transformer.

  Resolving *id-stx* can use certificates for the expression being transformed (see §12.6.3) as well as inactive certificates associated with *id-stx* (see §12.6.3.1). Furthermore, if the transformer is defined within a module (i.e., the current transformation was triggered by a use of a module-defined identifier) or if the current expression is being expanded for the body of a module, then resolving *id-stx* can access any identifier defined by the module.

- `(syntax-local-lift-expression stx)` returns a fresh identifier, and it cooperates with the `module`, `letrec-syntaxes+values`, `define-syntaxes`, `begin-for-syntax`, and top-level expanders to bind the generated identifier to the expression *stx*. A run-time expression within a module is lifted to the module's top level, just before the expression whose expansion requests the lift. Similarly, a run-time expression outside of a module is lifted to a top-level definition. A compile-time expression in a `letrec-syntaxes+values` or `define-syntaxes` binding is lifted to a `let` wrapper around the corresponding right-hand side of the binding. A compile-time expression within `begin-for-syntax` is lifted to a `define-for-syntax` declaration just before the requesting expression. Other syntactic forms can capture lifts by using `local-expand/capture-lifts` or `local-transformer-expand/capture-lifts`.

- `(syntax-local-lift-module-end-declaration stx)` cooperates with the `module` form to insert *stx* as a top-level declaration at the end of the module currently being expanded. The result is void. If the current expression being transformed is not within a `module` form, or if it is not a run-time expression, then the `exn:fail:contract` exception is raised. If the current expression being transformed is not in the module top-level, then *stx* is eventually expanded in an expression context.

- `(syntax-local-name)` returns an inferred name for the expression position being transformed, or `#f`; see also §6.2.3.

- `(syntax-local-context)` returns either `'expression`, `'top-level`, `'module`, `'module-begin`, or a non-empty list of arbitrary values.

  The first three possibilities indicate that the expression is being expanded for a (non-definition) expression position, a top-level position, or a module top-level position, respectively. The next-to-last, `'module-begin`, indicates that the expression is being expanded as the sole form within a module, where it might produce `#%plain-module-begin`.

The last possibility, a list, indicates expansion for an internal-definition position. The identity of the lists's first element (i.e., its `eq?`ness) reflects the identity of the internal-definition context; in particular two transformer expansions receive the same first value if and only if they are invoked for the same internal-definition context. Later values in the list similarly identify internal-definition contexts that are still being expanded, and that required the expansion of nested internal-definition contexts.

- `(syntax-local-get-shadower` *`identifier`*`)` returns *`identifier`* if no binding in the current expansion context shadows *`identifier`*, if *`identifier`* has no module context, and if the current expansion context is not a module. If a binding of *`inner-identifier`* shadows *`identifier`*, the result is the same as `(syntax-local-get-shadower` *`inner-identifier`*`)`, except that it has the location and properties of *`identifier`*. Otherwise, the result is the same as *`identifier`* with its module context (if any) removed and the current module context (if any) added. Thus, the result is an identifier corresponding to the innermost shadowing of *`identifier`* in the current context if its shadowed, and a module-contextless version of *`identifier`* otherwise.

- `(syntax-local-certifier` [*`active?`*]`)` returns a procedure that captures any certificates currently available for `syntax-local-value` or `local-expand`. The procedure accepts one to three arguments: *`stx`* (required), *`key-v`* (optional), and *`intro-proc`* (optional). The procedure's result is a syntax object like *`stx`*, except that it includes the captured certificates as inactive (see §12.6.3.1) if *`active?`* is `#f` (the default) or active otherwise. If *`key-v`* is supplied and not `#f`, it is associated with each captured certificate for later use through `syntax-recertify` (see §12.6.3.3). If *`intro-proc`* is supplied, and if it is not `#f` (the default), then it must be a procedure created by `make-syntax-introducer`, in which case the certificate applies only to parts of *`stx`* that are marked as introduced by *`intro-proc`*.

  Supply `#t` for *`active?`* when the syntax to be certified can be safely used in any context by any party, and where access to the syntax object should not confer any additional access. Supply `#f` for *`active?`* when the syntax to be certified is not accessible to parties that might abuse the access that the certificate provides, and when the certified syntax eventually appears (via macro expansion) within a larger expression from which it cannot be safely extracted by other parties.

- `(syntax-transforming?)` returns `#t` if an expression is currently being transformed (so that procedures like `syntax-local-value` can be called), `#f` otherwise.

A transformer can also expand or partially expand subexpressions from its input syntax object:

- `(local-expand` *`stx context-v stop-id-stx-list`* [*`intdef-ctx`*]`)` expands *`stx`* in the lexical context of the expression currently being expanded. The *`context-v`* argument is used as the result of *`syntax-local-context`* for immediate expansions; for a particular internal-definition context, generate a unique value and `cons` it onto the current result of `syntax-local-context` if it is a list.

  When an identifier in *`stop-id-stx-list`* is encountered by the expander in a subexpression, expansions stops for the subexpression. If `#%app`, `#%top`, or `#%datum` (see §12.5) appears in *`stop-id-stx-list`*, then application, top-level variable reference, and literal data expressions without the respective explicit form are not wrapped with the explicit form. If *`stop-id-stx-list`* is `#f` instead of a list, then *`stx`* is expanded only as long as the outermost form of *`stx`* is a macro (i.e., expansion does not proceed to sub-expressions).

  The optional *`intdef-ctx`* argument must be either `#f` (the default) or the result of `syntax-local-make-definition-co`
  In the latter case, lexical information for internal definitions is added to *`stx`* before it is expanded. The lexical information is also added to the expansion result (because the expansion might introduce bindings or references to internal-definition bindings).

  Expansion of *`stx`* can use certificates for the expression already being expanded (see §12.6.3) , and inactive certificates associated with *`stx`* are activated for *`stx`* (see §12.6.3.1). Furthermore, if the macro expander is defined within a module (i.e., the current expansion was triggered by a use of a module-defined identifier) or if the current expression is being expanded for the body of a module, then the expansion of *`stx`* can use any identifier defined by the module.

- `(syntax-local-expand-expression` *`stx`* `[l])` ike `local-expand` given `'expression` and an empty stop list, but with two results: a syntax object for the fully expanded expression, and a syntax object whose content is opaque. The latter can be used in place of the former (perhaps in a larger expression produced by a macro transformer), and when the macro expander encouters the opaque object, it substitutes the fully expanded expression without re-expanding it; the `exn:fail:syntax` exception is raised if the expansion context includes bindings or marks that were not present for the original expansion, in which case re-expansion might produce different results. Consistent use of `syntax-local-expand-expression` and the opaque object thus avoids quadratic expansiontimes when local expansions are nested.
- `(local-transformer-expand` *`stx context-v stop-id-stx-list intdef-ctx`*`)` is like `local-expand`, but *`stx`* is expanded as a transformer expression instead of a run-time expression.
- `(local-expand/capture-lifts` *`stx context-v stop-id-stx-list intdef-ctx`*`)` is like `local-expand`, but if `syntax-local-lift-expression` is called during the expansion of *`stx`*, the result is a syntax object that represents a `begin` expression; lifted expression appear with their identifiers in `define-values` forms, and the expansion of *`stx`* is the last expression in the `begin`. The lifted expressions are not expanded.
- `(local-transformer-expand/capture-lifts` *`stx context-v stop-id-stx-list intdef-ctx`*`)` is like `local-expand/capture-lifts`, but *`stx`* is expanded as a transformer expression instead of a run-time expression. Lifted expressions are reported as `define-values` forms (in the transformer environment).
- `(syntax-local-make-definition-context)` creates an opaque internal-definition context value to be used with `local-expand` and other functions. A transformer should create one context for each set of internal definitions to be expanded, and use it when expanding any form whose lexical context should include the definitions. After discovering an internal `define-values` or `define-syntaxes` form, use `syntax-local-bind-syntaxes` to add bindings to the context.
- `(syntax-local-bind-syntaxes` *`id-list expr-or-false intdef-ctx`*`)` binds each identifier in *`id-list`* within the internal-definition context represented by *`intdef-ctx`*, where *`intdef-ctx`* is the result of `syntax-local-make-definition-context`. Supply `#f` for *`expr-or-false`* when the identifiers correspond to `define-values` bindings, and supply a compile-time expression when the identifiers correspond to `define-syntaxes` bindings; the later case, the number of values produces by the expression should match the number of identifiers, otherwise the `exn:fail:contract:arity` exception is raised.

To track the introduction of identifiers by a macro (see §12.3), the syntax expander adds a special "mark" to a syntax object that is provided to a transformer, and also marks the result of the transformer. Consecutive marks cancel, and each transformer application has a distinct mark, so the only parts of the resulting syntax object with marks are the parts that were introduced by the transformer. A transformer can explicitly add a current mark to a syntax object using `syntax-local-introduce` or the result of `make-syntax-introducer`:

- `(syntax-local-introduce` *`stx`*`)` produces a syntax object that is like *`stx`*, except that a mark for the current expansion is added (possibly canceling an existing mark in parts of *`stx`*).
- `(make-syntax-introducer)` produces a procedure that behaves like `syntax-local-introduce`, except using a fresh mark. Multiple applications of the same `make-syntax-introducer` result procedure use the same mark, and different result procedures use distinct marks.

Explicit marking is useful on syntax objects that flow into or out of a transformer without being the transformer argument or result. For example, DrScheme's Check Syntax tool recognizes `'disappeared-binding` and `'disappeared-use` properties, which specify bound–binding identifier pairs in the source program that do not appear in the expansion. Example:

```
(define-syntax (match-list stx)
     (syntax-case stx ()
       [(_ expr (id ...) result-id)
        (let ([ids (syntax->list (syntax (id ...)))]
              [result-id (syntax result-id)])
          ;; Make sure the expression is well formed:
          (for-each (lambda (id)
                      (unless (identifier? id)
                        (raise-syntax-error #f "not an identifier" stx id)))
                    (append ids (list result-id)))
          ;; Find the matching identifier and produce a list-ref expression:
          (let loop ([ids ids] [pos 0])
            (cond
              [(null? ids) (raise-syntax-error #f "no pattern binding" stx result-id)]
              [(bound-identifier=? (car ids) result-id)
               ;; Found it; produce the list-ref expression, and
               ;; tell the Check Syntax tool about the pattern-variable binding:
               (with-syntax ([pos pos])
                 (syntax-property
                  (syntax-property
                   (syntax (list-ref expr pos)) ; the expansion result
                   'disappeared-binding
                   (syntax-local-introduce (car ids)))
                  'disappeared-use
                  (syntax-local-introduce result-id)))]
              [else (loop (cdr ids) (add1 pos))]))))]))

   ;; Test it:
   (match-list '(1 2 3) (a b c) b) ; ⇒ 2
```

In this example, Check Syntax will draw a binding arrow from the first *b* to the second *b*. Without the calls to `syntax-local-introduce`, the identifiers stored in the property would appear to have originated from the transformer, instead of from the transformer's argument; consequently, Check Syntax would not draw the arrow, because it would not know that the *b*s exist in the source program.

12.6.1 Expanding Expressions to Primitive Syntax

`(expand stx-or-sexpr)` expands all non-primitive syntax in *`stx-or-sexpr`*, and returns a syntax object for the expanded expression. See below for the grammar of fully expanded expressions. Before *`stx-or-sexpr`* is expanded, its lexical context is enriched with `namespace-syntax-introduce` as for `eval` (see §8.3 and §14.1). Use `syntax-object->datum` to convert the returned syntax object into an S-expression.

`(expand-syntax stx)` is like `(expand stx)`, except that the argument must be a syntax object, and its lexical context is not enriched before expansion.

`(expand-once stx-or-sexpr)` partially expands syntax in the *`stx-or-sexpr`* and returns a syntax object for the partially-expanded expression. Due to limitations in the expansion mechanism, some context information may be lost. In particular, calling `expand-once` on the result may produce a result that is different from expansion via `expand`. Before *`stx-or-sexpr`* is expanded, its lexical context is enriched with `namespace-syntax-introduce` as for `eval` (see §8.3 and §14.1).

`(expand-syntax-once stx)` is like `(expand-once stx)`, except that the argument must be a syntax object, and its lexical context is not enriched before expansion.

`(expand-to-top-form stx-or-sexpr)` partially expands syntax in *`stx-or-sexpr`* to reveal the outermost syntactic form. This partial expansion is mainly useful for detecting top-level uses of `begin`. Unlike expanding the result of `expand-once`, expanding the result of `expand-to-top-form` with `expand` produces the same result as using `expand` on the original syntax. Before *`stx-or-sexpr`* is expanded, its lexical context is enriched with `namespace-syntax-introduce` as for `eval` (see §8.3 and §14.1).

`(expand-syntax-to-top-form stx)` is like `(expand-to-top-form stx)`, except that the argument must be a syntax object, and its lexical context is not enriched before expansion.

The possible shapes of a fully expanded expression are defined by *`top-level-expr`*:

```
top-level-expr is one of
  general-top-level-expr
  (#%expression expr)
  (module identifier name (#%plain-module-begin module-level-expr ···))
  (begin top-level-expr ···)

module-level-expr is one of
  general-top-level-expr
  (provide provide-spec ...)

general-top-level-expr is one of
  expr
  (define-values (variable ···) expr)
  (define-syntaxes (identifier ···) expr)
  (define-values-for-syntax (variable ···) expr)
  (require require-spec ···)
  (require-for-syntax require-spec ···)
  (require-for-template require-spec ···)

expr is one of
  variable
  (lambda formals expr ···¹)
  (case-lambda (formals expr ···¹) ···)
  (if expr expr)
  (if expr expr expr)
  (begin expr ···¹)
  (begin0 expr expr ···)
  (let-values (((variable ···) expr) ···) expr ···¹)
  (letrec-values (((variable ···) expr) ···) expr ···¹)
  (set! variable expr)
  (quote datum)
  (quote-syntax datum)
  (with-continuation-mark expr expr expr)
  (#%app expr ···¹)
  (#%datum . datum)
  (#%top . variable)
  (#%variable-reference variable)
  (#%variable-reference (#%top . variable))
```

where *`formals`* is defined in §2.9, and *`require-spec`* and *`provide-spec`* are defined in §5.2.

When a *`variable`* expression appears in a fully-expanded expression, it either refers to a variable bound by `lambda`, `case-lambda`, `let-values`, `letrec-values`, or `define` (within the current module), or it refers to an imported variable. (In other words, a *`variable`* not wrapped by `#%top` never refers to a top-level variable.)

The keywords in the above grammar are placeholders for identifiers that are `module-identifier=?` (or `module-transformer-identifier=?` for `define-syntax` expressions) to the same-named exports of `mzscheme`. Due to import renamings, the printed identifier names can be different in the expanded expression.

12.6.2 Syntax Object Properties

Every syntax object has an associated property list, which can be queried or extended with `syntax-property`:

- `(syntax-property` *`stx key-v v`*`)` extends *`stx`* by associating an arbitrary property value *`v`* with the key *`key-v`*; the result is a new syntax object with the association (while *`stx`* itself is unchanged).
- `(syntax-property` *`stx key-v`*`)` returns an arbitrary property value associated to *`stx`* with the key *`key-v`*, or `#f` if no value is associated to *`stx`* for *`key-v`*.
- `(syntax-property-symbol-keys` *`stx`*`)` returns a list of all symbols that as keys have associated properties in *`stx`*. Uninterned symbols (see §3.7) are not included in the result list.

The `read-syntax` procedure attaches a `'paren-shape` property to any pair or vector syntax object generated from parsing a pair of square brackets ("[" and "]") or curly braces ("{" and "}").[4] The property value is `#\[` in the former case, and `#\{` in the latter case. The `syntax` form copies any `'paren-shape` property from the source of a template to corresponding generated syntax.

Both the syntax input to a transformer and the syntax result of a transformer may have associated properties. The two sets of properties are merged by the syntax expander: each property in the original and not present in the result is copied to the result, and the values of properties present in both are combined with `cons-immutable` (result value first, original value second).

Before performing the merge, however, the syntax expander automatically add a property to the original syntax object using the key `'origin`. If the source syntax has no `'origin` property, it is set to the empty list. Then, still before the merge, the identifier that triggered the macro expansion (as syntax) is `cons-immutable`d onto the `'origin` property so far.

The `'origin` property thus records (in reverse order) the sequence of macro expansions that produced an expanded expression. Usually, the `'origin` value is an immutable list of identifiers. However, a transformer might return syntax that has already been expanded, in which case an `'origin` list can contain other lists after a merge.

For example, the expression

```
(or x y)
```

expands to

```
(let ((or-part x)) (if or-part or-part (or y)))
```

which, in turn, expands to

```
(let-values ([(or-part) x]) (if or-part or-part y))
```

The syntax object for the final expression will have an `'origin` property whose value is `(list-immutable (quote-syntax let) (quote-syntax or))`.

`(syntax-track-origin` *`new-stx orig-stx id-stx`*`)` adds properties to *`new-stx`* in the same way that macro expansion adds properties to a transformer result. In particular, it merges the properties of *`orig-stx`* into

[4]More precisely, the property is attached by the default read handler in syntax mode when using the default readtable.

new-stx, first adding *id-stx* as an 'origin property, and it returns the property-extended syntax object. Use the syntax-track-origin procedure in a macro transformer that discards syntax (corresponding to *orig-stx* with a keyword *id-stx*) leaving some other syntax in its place (corresponding to *new-stx*).

Besides 'origin tracking for general macro expansion, MzScheme adds properties to expanded syntax (often using syntax-track-origin) to record additional expansion details:

- When a begin form is spliced into a sequence with internal definitions (see §2.8.5), syntax-track-origin is applied to every spliced element from the begin body. The second argument to syntax-track-origin is the begin form, and the third argument is the begin keyword (extracted from the spliced form).
- When an internal define-values or define-syntaxes form is converted into a letrec-values+syntaxes form (see §2.8.5), syntax-track-origin is applied to each generated binding clause. The second argument to syntax-track-origin is the converted form, and the third argument is the define-values or define-syntaxes keyword form the converted form.
- When a letrec-values+syntaxes expression is fully expanded, syntax bindings disappear, and the result is either a letrec-values form (if the unexpanded form contained non-syntax bindings), or only the body of the letrec-values+syntaxes form (wrapped with begin if the body contained multiple expressions). To record the disappeared syntax bindings, a property is added to the expansion result: an immutable list of identifiers from the disappeared bindings, as a 'disappeared-binding property.
- When a subtyping define-struct form is expanded, the identifier used to reference the base type does not appear in the expansion. Therefore, the define-struct transformer adds the identifier to the expansion result as a 'disappeared-use property.
- When a reference to an unexported or protected identifier from a module is discovered (and the reference is certified; see §12.6.3), the 'protected property is added to the identifier with a #t value.
- When or read-syntax or read-honu-syntax generates a syntax object, it attaches a property to the object (using a private key) to mark the object as originating from a read. The syntax-original? predicate looks for the property to recognize such syntax objects. (See §12.2.2 for more information.)

The syntax-original? procedure and the 'origin, 'disappeared-binding, and 'disappeared-use properties are used by program-processing tools (such as Check Syntax in DrScheme) to relate source code to its expanded form. Implementors of macro transformers should consider whether properties added automatically by MzScheme are sufficient for tools to make sense of expansion result, and implementors should use syntax-track-origin and syntax-property as necessary to fill in gaps (see §12.6 for an example).

See §12.6.5 for information about properties generated by the expansion of a module declaration. See §3.12.1 and §6.2.3 for information about properties recognized when compiling a procedure. See §14.3 for information on properties and byte codes.

12.6.3 Certificates for Protected References

As illustrated in §5.3, a macro can expand into a use of an identifier that is not exported from the macro's module. In general, such an identifier must not be extracted from the expanded expression and used in a different context, because using the identifier in a different context may break invariants of the macro's module. For example, the following module exports a macro *go* that expands to a use of *unchecked-go*:

```
(module m mzscheme
  (provide go)
  (define (unchecked-go n x)
   ;; to avoid disaster, n must be a number
   (+ n 17))
```

```
    (define-syntax (go stx)
      (syntax-case stx ()
       [(_ x)
        #'(unchecked-go 8 x)])))
```

If the reference to `unchecked-go` is extracted from the expansion of `(go 'a)`, then it might be inserted into a new expression, `(unchecked-go #f 'a)`, leading to disaster. The `datum->syntax-object` procedure can be used similarly to construct references to an unexported identifier, even when no macro expansion includes a reference to the identifier.

To prevent such abuses of unexported identifiers, MzScheme's macro expander and compiler reject references to unexported identifiers unless they appear in *certified* syntax objects. The macro expander always certifies a syntax object that is produced by a transformer. For example, when `(go 'a)` is expanded to `(unchecked-go 8 'a)`, a certificate is attached to the result `(unchecked-go 8 'a)`. Extracting just `unchecked-go` removes the identifier from the certified expression, so that the reference is disallowed when it is inserted into `(unchecked-go #f 'a)`.

In addition to checking module references, the macro expander disallows references to local bindings where the binding identifier is less certified than the reference. Otherwise, the expansion of `(go 'a)` could be wrapped with a local binding that redirects `#%app` to `values`, thus obtaining the value of `unchecked-go`. Note that a capturing `#%app` would have to be extracted from the expansion of `(go 'a)`, since lexical scope would prevent an arbitrary `#%app` from capturing. The act of extracting `#%app` removes its certification, whereas the `#%app` within the expansion is still certified; comparing these certifications, the macro expander rejects the local-binding reference, and `unchecked-go` remains protected.

In much the same way that the macro expander copies properties from a transformer's input to its output, the expander copies certificates from a transformer's input to its output. Building on the previous example,

```
  (module n mzscheme
    (require m)
    (provide go-more)
    (define y 'hello)
    (define-syntax (go-more stx)
      #'(go y)))
```

the expansion of `(go-more)` introduces a reference to the unexported `y` in `(go y)`, and a certificate allows the reference to `y`. As `(go y)` is expanded to `(unchecked-go 8 y)`, the certificate that allows `y` is copied over, in addition to the certificate that allows the reference to `unchecked-go`.

When a protected identifier becomes inaccessible by direct reference (i.e., when the current code inspector is changed so that it does not control the module's invocation; see §9.4), the protected identifier is treated like an unexported identifier.

12.6.3.1 Certificate Propagation

When the result of a macro expansion contains a `quote-syntax` form, the macro expansion's certificate must be attached to the resulting syntax object to support macro-generating macros. In general, when the macro expander encounters `quote-syntax`, it attaches all certificates from enclosing expressions to the quoted syntax constant. However, the certificates are attached to the syntax constant as *inactive* certificates, and inactive certificates do not count directly for certifying identifier access. Inactive certificates become active when the macro expander certifies the result of a macro expansion; at that time, the expander removes all inactive certificates within the expansion result and attaches active versions of the certificates to the overall expansion result.

For example, suppose that the `go` macro is implemented through a macro:

```
  (module m mzscheme
(provide def-go)
(define (unchecked-go n x)
  (+ n 17))
(define-syntax (def-go stx)
 (syntax-case stx ()
   [(_ go)
    #'(define-syntax (go stx)
         (syntax-case stx ()
          [(_ x)
           #'(unchecked-go 8 x)]))]))))
```

When *def-go* is used inside another module, the generated macro should legally generate expressions that use *unchecked-go*, since *def-go* in *m* had complete control over the generated macro.

```
(module n mzscheme
  (require m)
  (def-go go)
  (go 10)) ; access to unchecked-go is allowed
```

This example works because the expansion of `(def-go go)` is certified to access protected identifiers in *m*, including *unchecked-go*. Specifically, the certified expansion is a definition of the macro *go*, which includes a syntax-object constant *unchecked-go*. Since the enclosing macro declaration is certified, the *unchecked-go* syntax constant gets an inactive certificate to access protected identifiers of *m*. When `(go 10)` is expanded, the inactive certificate on *unchecked-go* is activated for the macro result `(unchecked-go 8 10)`, and the access of *unchecked-go* is allowed.

To see why *unchecked-go* as a syntax constant must be given an inactive certificate instead of an active one, it's helpful to write the *def-go* macro as follows:

```
(define-syntax (def-go stx)
 (syntax-case stx ()
   [(_ go)
    #'(define-syntax (go stx)
         (syntax-case stx ()
          [(_ x)
           (with-syntax ([ug (quote-syntax unchecked-go)])
             #'(ug 8 x))]))]))
```

In this case, *unchecked-go* is clearly quoted as an immediate syntax object in the expansion of `(def-go go)`. If this syntax object were given an active certificate, then it would keep the certificate—directly on the identifier *unchecked-go*—in the result `(unchecked-go 8 10)`. Consequently, the *unchecked-go* identifier could be extracted and used with its certificate intact. Attaching an inactive certificate to *unchecked-go* and activating it only for the complete result `(unchecked-go 8 10)` ensures that *unchecked-go* is used only in the way intended by the implementor of *def-go*.

The `datum->syntax-object` procedure allows inactive certificates to be transferred from one syntax object to another. Such transfers are allowed because a macro transformer with access to the syntax object could already wrap it with an arbitrary context before activating the certificates. In practice, transferring inactive certificates is useful mainly to macros that implement to new template forms, such as `syntax/loc`.

12.6.3.2 Internal Certificates

In some cases, a macro implementor intends to allow limited destructuring of a macro result without losing the result's certificate. For example, given the following *define-like-y* macro,

```
(module q mzscheme
  (provide define-like-y)
  (define y 'hello)
  (define-syntax (define-like-y stx)
    (syntax-case stx ()
      [(_ id) #'(define-values (id) y)])))
```

someone may use the macro in an internal definition:

```
(let ()
  (define-like-y x)
  x)
```

The implementor of the `q` module most likely intended to allow such uses of `define-like-y`. To convert an internal definition into a `letrec` binding, however, the `define` form produced by `define-like-y` must be deconstructed, which would normally lose the certificate that allows the reference to `y`.

The internal use of `define-like-y` is allowed because the macro expander treats specially a transformer result that is a syntax list beginning with `define-values`. In that case, instead of attaching the certificate to the overall expression, the certificate is instead attached to each individual element of the syntax list, pushing the certificates into the second element of the list so that they are attached to the defined identifiers. Thus, a certificate is attached to `define-values`, `x`, and `y` in the expansion result `(define-values (x) y)`, and the definition can be deconstructed for conversion to `letrec`.

Just like the new certificate that is added to a transformer result, old certificates from the input are similarly moved to syntax-list elements when the result starts with `define-values`. Thus, `define-like-y` could have been implemented to produce `(define id y)`, using `define` instead of `define-values`. In that case, the certificate to allow reference to `y` would be attached initially to the expansion result `(define x y)`, but as the `define` is expanded to `define-values`, the certificate would be moved to the parts.

The macro expander treats syntax-list results starting with `define-syntaxes` in the same way that it treats results starting with `define-values`. Syntax-list results starting with `begin` are treated similarly, except that the second element of the syntax list is treated like all the other elements (i.e., the certificate is attached to the element instead of its content). Furthermore, the macro expander applies this special handling recursively, in case a macro produces a `begin` form that contains nested `define-values` forms.

The default application of certificates can be overridden by attaching a `'certify-mode` property (see §12.6.2) to the result syntax object of a macro transformer. If the property value is `'opaque`, then the certificate is attached to the syntax object and not its parts. If the property value is `'transparent`, then the certificate is attached to the syntax object's parts. If the property value is `'transparent-binding`, then the certificate is attached to the syntax object's parts and to the sub-parts of the second part (as for `define-values` and `define-syntaxes`). The `'transparent` and `'transparent-binding` modes triggers recursive property checking at the parts, so that the certificate can be pushed arbitrarily deep into a transformer's result.

12.6.3.3 Checking and Transferring Certificates

In general, a certificate combines a mark (see §12.6), a module name (more precisely, a module path index; see §5.4.2), an inspector, and an arbitrary key object. Within a certified syntax object, the certificate's mark is attached to every piece of syntax that was introduced by the relevant macro transformation (see again §12.6), so the certificate applies only to those pieces of syntax, and only to identifiers that are bound by the transformer's module. The certificate's inspector depends on the module that defined the transformer; specifically, it is the inspector for the module's declaration (see §9.4). A certificate's key is hidden if it is introduced by macro expansion, but applying the result of `syntax-local-certifier` (see §12.6) can introduce certificates with other keys.

To check access to an unexported identifier, the compiler or macro expander checks each of the identifier's marks and module bindings; if, for some mark, the identifier's enclosing expressions include a certificate with the mark, the identifier's binding module, and with an inspector that controls the module's invocation (as opposed to the module's declaration; see again §9.4), then the access is allowed. To check access to a protected identifier, only the certificate's mark and inspector are used (i.e., the module that bound the transformer is irrelevant, as long as it was evaluated with a sufficiently powerful inspector). The certificate key is not used in checking references.

To check access to a locally bound identifier, the compiler or macro expander checks the marks of the binding and reference identifiers; for every mark that they have in common, if the reference identifier has a certificate for the mark from an enclosing expression, the binding identifier must have a certificate for the mark from an enclosing expression, otherwise the reference is disallowed. (The reference identifier can have additional certificates for marks that are not attached to the binding identifier.) The binding module (if any) and the certificate key are not used for checking a local reference.

The `datum->syntax-object` procedure never transfers a certificate from one syntax object to another, so it cannot be used to gain access to an unexported identifier. The `syntax-recertify` procedure can be used to transfer a certificate from one syntax object to another, but only if the certificate's key is provided, or if a sufficiently powerful inspector is provided. Thus, a certificate's inspector serves two roles: it determines the certificate's power to grant access, and also allows the certificate to be moved arbitrarily by anyone with a more powerful inspector.

`(syntax-recertify` *`new-stx old-stx inspector key-v`*`)` copies certain certificates of *`old-stx`* to *`new-stx`*: a certificate is copied if its inspector is either *`inspector`* or controlled by *`inspector`*, or if the certificate's key is *`key-v`*; otherwise the certificate is not copied. The result is a syntax object like *`new-stx`*, but with the copied certificates. (The *`new-stx`* object itself is not modified.) Both active and inactive certificates are copied.

12.6.4 Information on Structure Types

The `define-struct` form (see §4.1) binds the name of a structure type to an expansion-time value that records the identifiers bound to the structure type, the constructor procedure, the predicate procedure, and the field accessor and mutator procedures. This information can be used during the expansion of other expressions by transformer that call `syntax-local-value` (see §12.6).

For example, the `define-struct` variant for subtypes (see §4.2) uses the base type name *`t`* to find the variable `struct:`*`t`* containing the base type's descriptor; it also folds the field accessor and mutator information for the base type into the information for the subtype. The `match` form (see Chapter 27 of *PLT MzLib: Libraries Manual*) uses a type name to find the predicates and field accessors for the structure type.

Besides using the information, other syntactic forms can even generate information with the same shape. For example, the `struct` form in an imported signature for `unit` (see Chapter 55 of *PLT MzLib: Libraries Manual*) causes the `unit` transformer to generate information about imported structure types, so that `match` and subtyping `define-struct` expressions work within the unit.

The expansion-time information for a structure type is represented as an immutable list of six items:

- an identifier that is bound to the structure type's descriptor, or `#f` it none is known;
- an identifier that is bound to the structure type's constructor, or `#f` it none is known;
- an identifier that is bound to the structure type's predicate, or `#f` it none is known;
- an immutable list of identifiers bound to the field accessors of the structure type, optionally with `#f` as the list's last element. A `#f` as the last element indicates that the structure type may have additional fields, otherwise the list is a reliable indicator of the number of fields in the structure type. Furthermore, the accessors are listed in

reverse order for the corresponding constructor arguments. (The reverse order enables sharing in the lists for a subtype and its base type.)

- an immutable list of identifiers bound to the field mutators of the structure type, or `#f` for each field that has no known mutator, and optionally with an extra `#f` as the list's last element (if the accessor list has such a `#f`). The list's order and the meaning of a final `#f` are the same as for the accessor identifiers, and the length of the mutator list is the same as the accessor list's length.
- an identifier that determines a super-type for the structure type, `#f` if the super-type (if any) is unknown, or `#t` if there is no super-type. If a super-type is specified, the identifier is also bound to structure-type expansion-time information.

The implementor of a syntactic form can expect users of the form to know what kind of information is available about a structure type. For example, the `match` implementation works with structure information containing an incomplete set of accessor bindings, because the user is assumed to know what information is available in the context of the `match` expression. In particular, the `match` expression can appear in a `unit` form with an imported structure type, in which case the user is expected to know the set of fields that are listed in the signature for the structure type.

12.6.5 Information on Expanded and Compiled Modules

MzScheme provides an interface for obtaining information about an expanded or compiled module declaration's imports and exports. This information is intended for use by tools such as a compilation manager.

Information for an expanded module declaration is stored in a set of properties attached to the syntax object:

- `'module-direct-requires` — an immutable list of module path indices (or symbols) representing the modules explicitly imported into the module.
- `'module-direct-for-syntax-requires` — an immutable list of module path indices (or symbols) representing the modules explicitly for-syntax imported into the module.
- `'module-direct-for-template-requires` — an immutable list of module path indices (or symbols; see §5.4.2) representing the modules explicitly for-template imported into the module.
- `'module-variable-provides` — an immutable list of provided items, where each item is one of the following:
  - *`symbol`* — represents a locally defined variable that is provided with its defined name.
  - `(cons-immutable` *`provided-symbol defined-symbol`*`)` — represents a locally defined variable that is provided with renaming; the first symbol is the exported name, and the second symbol is the defined name.
  - `(list*-immutable` *`module-path-index provided-symbol defined-symbol`*`)` — represents a re-exported and possibly re-named variable from the specified module; *`module-path-index`* is either an index or symbol (see §5.4.2), indicating the source module for the binding. The *`provided-symbol`* is the external name for the re-export, and *`defined-symbol`* is the originally defined name in the module specified by *`module-path-index`*.
- `'module-syntax-provides` — like `'module-variable-provides`, but for syntax exports instead of variable exports.
- `'module-indirect-provides` — an immutable list of symbols for variables that are defined in the module but not exported; they may be exported indirectly through macro expansions. Definitions of macro-generated identifiers create uninterned symbols in this list.
- `'module-kernel-reprovide-hint` — either `#f`, `#t`, or a symbol. If it is `#t`, then the module re-exports all of the functionality from MzScheme's internal kernel module. If it is a symbol, then all kernel exports but the

indicated one is re-exported, and some other export is provided with the indicated name. This ad hoc information is used in an optimization by the **mzc** compiler.

- `'module-self-path-index` — a module path index (see §5.4.2) whose parts are both `#f`. This information is used by the **mzc** compiler to manage syntax objects (which contain module-relative information keyed on the module's own index).

`(compiled-module-expression? v)` returns `#t` if *v* is a compiled expression for a `module` declaration, `#f` otherwise. See also §14.3.

`(module-compiled-name compiled-module-code)` takes a module declaration in compiled form (see §14.3) and returns a symbol for the module's declared name.

`(module-compiled-imports compiled-module-code)` takes a module declaration in compiled form (see §14.3) and returns three values: an immutable list of module path indices (and symbols; see §5.4.2) for the module's explicit imports, an immutable list of module path indices (and symbols) for the module's explicit for-syntax imports, and an immutable list of module path indices (and symbols) for the module's explicit for-template imports.

`(module-compiled-exports compiled-module-code)` takes a module declaration in compiled form (see §14.3) and returns two values: an immutable list of symbols for the module's explicit variable exports, an immutable list symbols for the module's explicit syntax exports.

13. Memory Management

13.1 Weak Boxes

A *weak box* is similar to a normal box (see §3.11), but when the automatic memory manager can prove that the content value of a weak box is only reachable via weak references, the content of the weak box is replaced with `#f`. A *weak reference* is a reference through a weak box, through a key reference in a weak hash table (see §3.14), through a value in an ephemeron where the value can be replaced by `#f` (see §13.2), or through a custodian (see §9.2).

- `(make-weak-box` *`v`*`)` returns a new weak box that initially contains *`v`*.
- `(weak-box-value` *`weak-box`*`)` returns the value contained in *`weak-box`*. If the memory manager has proven that the previous content value of *`weak-box`* was reachable only through a weak reference, then `#f` is returned.
- `(weak-box?` *`v`*`)` returns `#t` if *`v`* is a weak box, `#f` otherwise.

13.2 Ephemerons

An *ephemeron* is similar to a weak box (see §13.1), except that

1. an ephemeron contains a key and a value; the value can be extracted from the ephemeron, but the value is replaced by `#f` when the automatic memory manager can prove that either the ephemeron or the key is reachable only through weak references (see §13.1); and
2. nothing reachable from the value in an ephemeron counts toward the reachability of an ephemeron key (whether for the same ephemeron or another), unless the same value is reachable through a non-weak reference, or unless the value's ephemeron key is reachable through a non-weak reference (see §13.1 for information on weak references).

In particular, an ephemeron can be combined with a weak hash table (see §3.14) to produce a mapping where the memory manager can reclaim key–value pairs even when the value refers to the key. An example is shown below.

- `(make-ephemeron` *`key-v v`*`)` returns a new ephemeron whose key is *`key-v`* and whose value is initially *`v`*.
- `(ephemeron-value` *`ephemeron`*`)` returns the value contained in *`ephemeron`*. If the memory manager has proven that the key for *`ephemeron`* is only weakly reachable, then the result is `#f`.
- `(ephemeron?` *`v`*`)` returns `#t` if *`v`* is an ephemeron, `#f` otherwise.

Example:

```
;; This weak map is like a weak hash table, but
;; without the key-in-value problem:
(define (make-weak-map)
  (make-hash-table 'weak))

(define (weak-map-put! m k v)
  (hash-table-put! m k (make-ephemeron k (box v))))

(define (weak-map-get m k def-v)
  (let ([v (hash-table-get m k (lambda () #f))])
    (if v
        (let ([v (ephemeron-value v)])
          (if v
              (unbox v)
              def-v))
        def-v)))

(define m (make-weak-map))
(define k (list 1 2))
(weak-map-put! m k k)
(weak-map-get m k #f) ; ⇒ '(1 2)
(set! k #f)
```

list is eventually GCed even if `m` remains reachable

13.3 Will Executors

A *will executor* manages a collection of values and associated *will procedures*. The will procedure for each value is ready to be executed when the value has been proven (by the automatic memory manager) to be unreachable, except through weak references (see §13.1) or as the registrant for other will executors. A will is useful for triggering clean-up actions on data associated with an unreachable value, such as closing a port embedded in an object when the object is no longer used.

Calling the `will-execute` or `will-try-execute` procedure executes a will that is ready in the specified will executor. Wills are not executed automatically, because certain programs need control to avoid race conditions. However, a program can create a thread whose sole job is to execute wills for a particular executor.

- `(make-will-executor)` returns a new will executor with no managed values.
- `(will-executor? v)` returns `#t` if `v` is a will executor, `#f` otherwise.
- `(will-register executor v proc)` registers the value `v` with the will procedure `proc` in the will executor `executor`. When `v` is proven unreachable, then the procedure `proc` is ready to be called with `v` as its argument via `will-execute` or `will-try-execute`. The `proc` argument is strongly referenced until the will procedure is executed.
- `(will-execute executor)` invokes the will procedure for a single "unreachable" value registered with the executor `executable`. The value(s) returned by the will procedure is the result of the `will-execute` call. If no will is ready for immediate execution, `will-execute` blocks until one is ready.
- `(will-try-execute executor)` is like `will-execute` if a will is ready for immediate execution. Otherwise, `#f` is returned.

If a value is registered with multiple wills (in one or multiple executors), the wills are readied in the reverse order of registration. Since readying a will procedure makes the value reachable again, the will must be executed and the value must be proven again unreachable through only weak references before another of the wills is readied or executed. However, wills for distinct unreachable values are readied at the same time, regardless of whether the values are reachable from each other.

A will executor's register is held non-weakly until after the corresponding will procedure is executed. Thus, if the content value of a weak box (see §13.1) is registered with a will executor, the weak box's content is not changed to `#f` until all wills have been executed for the value and the value has been proven again reachable through only weak references.

13.4 Garbage Collection

`(collect-garbage)` forces an immediate garbage collection. Some effectively unreachable data may remain uncollected, because the collector cannot prove that it is unreachable, and the MzSchemeCGC variant of MzScheme is generally less able to prove unreachability than MzScheme3m. The `collect-garbage` procedure provides some control over the timing of collections, but garbage will obviously be collected even if this procedure is never called.

`(current-memory-use [`*`custodian`*`])` returns an estimate of the number of bytes of memory occupied by reachable data from *`custodian`*. (The estimate is calculated *without* performing an immediate garbage collection; performing a collection generally decreases the number returned by `current-memory-use`.) If *`custodian`* is not provided, the estimate is a total reachable from any custodians. When MzScheme is compiled without support for memory accounting, the estimate is the same (i.e., all memory) for any individual custodian; see also `custodian-memory-accounting-available?` in §9.2.

`(dump-memory-stats)` dumps information about memory usage to the (low-level) standard output port.

14. Evaluation and Compilation

14.1 Eval and Load

`(eval` *`expr`* [*`namespace`*]`)` evaluates *`expr`* in *`namespace`*, or in the current namespace if *`namespace`* is not supplied.[1] (See §8 and §7.9.1.5 for more information about namespaces.) The *`expr`* is evaluated in tail position with respect to the `eval` call. The *`expr`* can be a syntax object, a compiled expression, a compiled expression wrapped as a syntax object, or an arbitrary S-expression (which will be converted to a syntax object using `datum->syntax-object`; see §12.2.2). If *`expr`* is a syntax object or S-expression, then is enriched with lexical context using `namespace-syntax-introduce` before it is evaluated. However, if *`expr`* is a pair (or syntax pair) whose first element is `module-identifier=?` to MzScheme's `module` (after giving the identifier context with `namespace-syntax-introduce`), then only the `module` identifier is given context, and the rest of *`expr`* is left to the module's language.

`(eval-syntax` *`stx`* [*`namespace`*]`)` is like `(eval` *`stx`*`)`, except that *`stx`* must be a syntax object, and its lexical context is not enriched before it is evaluated.

`(load` *`file-path`*`)` evaluates each expression in the specified file using `eval`, wrapping the evaluation in a continuation prompt (see §6.5) for the default continuation prompt tag with handler that propagates the abort to the continuation of the `load` call.[2] The return value from `load` is the value of the last expression from the loaded file (or void if the file contains no expressions). If *`file-path`* is a relative path, then it is resolved to an absolute path using the current directory. Before the first expression of *`file-path`* is evaluated, the current `load-relative` directory (the value of the `current-load-relative-directory` parameter; see §7.9.1.6) is set to the absolute path of the directory containing *`file-path`*; after the last expression in *`file-path`* is evaluated (or when the load is aborted), the `load-relative` directory is restored to its pre-`load` value.

`(load-relative` *`file-path`*`)` is like `load`, but when *`file-path`* is a relative path, it is resolved to an absolute path using the current `load-relative` directory rather than the current directory. If the current `load-relative` directory is `#f`, then `load-relative` is the same as `load`.

`(load/use-compiled` *`file-path`*`)` is like `load-relative`, but `load/use-compiled` also checks for **.zo** files (usually produced with `compile-file`; see Chapter 13 of *PLT MzLib: Libraries Manual*) and **.so** (Unix), **.dll** (Windows), or **.dylib** (Mac OS X) files.[3] The check for a compiled file occurs whenever *`file-path`* ends with any extension (e.g., **.ss** or **.scm**), and the check consults the subdirectories indicated by the `use-compiled-file-paths` parameter (see §7.9.1.6), relative to *`file-path`*. The subdirectories are checked in order. A **.zo** version of the file is loaded if it exists directly in one of the indicated subdirectories, or a **.so/.dll/.dylib** version of the file is loaded if it exists within a **native** subdirectory of a `use-compiled-file-paths` directory, in an even deeper subdirectory as named by `system-library-subpath`. A compiled file is loaded only if its modification date is not older than the date for *`file-path`*. If both **.zo** and **.so/.dll/.dylib** files are available, the **.so/.dll/.dylib** file is used.

[1] The `eval` procedure actually calls the current evaluation handler (see §7.9.1.5) to evaluate the expression.

[2] The `load` procedure actually just sets the current `load-relative` directory and calls the current load handler (see §7.9.1.6) with *`file-path`* to load the file. The description of `load` here is actually a description of the default load handler.

[3] The `load/use-compiled` procedure actually just calls the current load/use-compiled handler (see §7.9.1.6). The default handler, in turn, calls the load or load-extension handler, depending on the type of file that is loaded.

Multiple files can be combined into a single **.so/.dll/.dylib** file by creating a special dynamic extension **_loader.so**, **_loader.dll**, or **_loader.dylib**. When such an extension is present where a normal **.so/.dll/.dylib** would be loaded, then the **_loader** extension is first loaded. The result returned by **_loader** must be a procedure that accepts a symbol. This procedure will be called with a symbol matching the base part of `file-path` (without the directory path part of the name and without the filename extension), and the result must be two values; if `#f` is returned as the first result, then `load/use-compiled` ignores **_loader** for `file-path` and continues as normal. Otherwise, the first return value is yet another procedure. When this procedure is applied to no arguments, it should have the same effect as loading `file-path`. The second return value is either a symbol or `#f`; a symbol indicates that calling the returned procedure has the effect of declaring the module named by the symbol (which is potentially useful information to a load handler; see §5.8).

While a **.zo**, **.so**, **.dll**, or **.dylib** file is loaded (or while a thunk returned by **_loader** is invoked), the current `load-relative` directory is set to the directory of the original `file-path`.

`(load/cd file-path)` is the same as `(load file-path)`, but `load/cd` sets both the current directory and current `load-relative` directory to the directory of `file-path` before the file's expressions are evaluated.

`(read-eval-print-loop)` starts a new `read-eval-print` loop using the current input, output, and error ports. The `read-eval-print` loop wraps each evaluation with a continuation prompt (see §6.5) using the default continuation prompt tag and prompt handler. The `read-eval-print` loop also wraps the read and print operations with a prompt for the default tag whose handler ignores abort arguments and continues the loop. The `read-eval-print-loop` procedure does not return until `eof` is read as an input expression; then it returns void.

The `read-eval-print-loop` procedure is parameterized by the current prompt read handler, the current evaluation handler, and the current print handler; a custom `read-eval-print` loop can be implemented as in the following example (see also §7.9.1):

```
(parameterize ([current-prompt-read my-read]
               [current-eval my-eval]
               [current-print my-print])
  (read-eval-print-loop))
```

14.2 Exiting

`(exit [v])` passes *v* on to the current exit handler (see `exit-handler` in §7.9.1.9). The default value for *v* is `#t`. If the exit handler does not escape or terminate the thread, void is returned.

The default exit handler quits MzScheme (or MrEd), using its argument as the exit code if it is between 1 and 255 inclusive (meaning "failure"), or 0 (meaning "success") otherwise.

When MzScheme is embedded within another application, the default exit handler may behave differently.

14.3 Compilation

Normally, compilation happens automatically: when syntax is evaluated, it is first compiled and then the compiled code is executed. However, MzScheme can also write and read compiled code. MzScheme can read compiled code much faster than reading syntax and compiling it, so compilation can be used to speed up program loading. The MzLib procedure `compile-file` (see Chapter 13 of *PLT MzLib: Libraries Manual*) is sufficient for most compilation purposes.

- `(compile expr)` returns a compiled expression for `expr` such that `(eval (compile expr))` is the same as `(eval expr)`. More precisely, `compile` calls the current compilation handler (see §7.9.1.5) to compile `expr`.

- `(compile-syntax stx)` returns a compiled expression for *stx* such that `(eval (compile-syntax stx))` is the same as `(eval-syntax stx)`.
- `(compiled-expression? v)` returns `#t` if *v* is a compiled expression, `#f` otherwise.

When a compiled expression is written to an output port, the written form starts with `#~`. These expressions are essentially assembly code for the MzScheme interpreter, and reading such an expression produces a compiled expression. When a compiled expression contains syntax object constants, the `#~` form of the expression drops location information and properties for the syntax objects (see §12.2 and §12.6.2).

The `read` procedure will not parse input beginning with `#~` unless the `read-accept-compiled` parameter (see §7.9.1.3) is set to true. When the default load handler is used to load a file, compiled-expression reading is automatically (temporarily) enabled as each expression is read.

Compiled code parsed from `#~` may contain references to unexported or protected bindings from a module. At read time, such references are associated with the current code inspector (see §7.9.1.8), and the code will only execute if that inspector controls the relevant module invocation (see §9.4).

A compiled-expression object may contain uninterned symbols (see §3.7) that were created by `gensym` or `string->uninterned-symbol`. When the compiled object is read via `#~`, each uninterned symbol in the original expression is mapped to a new uninterned symbol, where multiple instances of a single symbol are consistently mapped to the same new symbol. The original and new symbols have the same printed representation.

Due to the above restrictions, do not use `gensym` or `string->uninterned-symbol` to construct an identifier for a top-level or module binding. Instead, generate distinct identifiers either with `generate-temporaries` (see §12.2.2) or by applying the result of `make-syntax-introducer` (see §12.6) to an existing identifier.

14.4 Dynamic Extensions

A dynamically-linked extension library is loaded into MzScheme with `(load-extension file-path)`. The separate document *Inside PLT MzScheme* contains information about writing MzScheme extensions. An extension can only be loaded once during a MzScheme session, although the extension-writer can provide functionality to handle extra calls to `load-extension` for a single extension.

As with `load`, the current `load-relative` directory (the value of the `current-load-relative-directory` parameter; see §7.9.1.6) is set while the extension is loaded. The `load-relative-extension` procedure is like `load-extension`, but it loads an extension with a path that is relative to the current `load-relative` directory instead of the current directory.

The `load-extension` procedure actually just dispatches to the current load extension handler (see §7.9.1.6). The result of calling `load-extension` is determined by the extension. If the extension cannot be loaded, the `exn:fail:filesystem` exception is raised, and if the load fails because the extension has the wrong version, more specifically the `exn:fail:filesystem:version` exception is raised.

15. System Utilities

15.1 Time

15.1.1 Real Time and Date

`(current-seconds)` returns the current time in seconds. This time is always an exact integer based on a platform-specific starting date (with a platform-specific minimum and maximum value).

The value of `(current-seconds)` increases as time passes (increasing by 1 for each second that passes). The current time in seconds can be compared with a time returned by `file-or-directory-modify-seconds` (see §11.3.3).

`(seconds->date` *`secs-n`*`)` takes *`secs-n`*, a platform-specific time in seconds (an exact integer) returned by `current-seconds` or `file-or-directory-modify-seconds`, and returns an instance of the `date` structure type, as described below. If *`secs-n`* is too small or large, the `exn:fail` exception is raised.

The value returned by `current-seconds` or `file-or-directory-modify-seconds` is not portable among platforms. Convert a time in seconds using `seconds->date` when portability is needed.

The `date` structure type has the following fields:

- `second` : `0` to `61` (`60` and `61` are for unusual leap-seconds)
- `minute` : `0` to `59`
- `hour` : `0` to `23`
- `day` : `1` to `31`
- `month` : `1` to `12`
- `year` : e.g., `1996`
- `week-day` : `0` (Sunday) to `6` (Saturday)
- `year-day` : `0` to `365` (`364` in non-leap years)
- `dst?` : `#t` (daylight savings time) or `#f`
- `time-zone-offset` : the number of seconds east of GMT for this time zone (e.g., Pacific Standard Time is `−28800`), an exact integer [1]

The `date` structure type is transparent to all inspectors (see §4.5).

See also Chapter 16 of *PLT MzLib: Libraries Manual* for additional date utilities.

15.1.2 Machine Time

`(current-milliseconds)` returns the current "time" in fixnum milliseconds (possibly negative). This time is based on a platform-specific starting date or on the machine's startup time. Since the result is a fixnum, the value increases only over a limited (though reasonably long) time.

[1]The value produced for the `time-zone-offset` field tends to be sensitive to the value of the `"TZ"` environment variable, especially on Unix platforms. Consult the system documentation (usually under `tzset`) for details.

`(current-inexact-milliseconds)` returns the current “time” in positive milliseconds, not necessarily an integer. This time is based on a platform-specific starting date or on the machine’s startup time, but it never decreases (until the machine is turned off).

`(current-process-milliseconds)` returns the amount of processor time in fixnum milliseconds that has been consumed by the MzScheme process on the underlying operating system. (Under Unix and Mac OS X, this includes both user and system time.) The precision of the result is platform-specific, and since the result is a fixnum, the value increases only over a limited (though reasonably long) time.

`(current-gc-milliseconds)` returns the amount of processor time in fixnum milliseconds that has been consumed by MzScheme’s garbage collection so far. This time is a portion of the time reported by `(current-process-milliseconds)`.

15.1.3 Timing Execution

The `time-apply` procedure collects timing information for a procedure application:

- `(time-apply` *`proc arg-list`*`)` invokes the procedure *`proc`* with the arguments in *`arg-list`*. Four values are returned: a list containing the result(s) of applying *`proc`*, the number of milliseconds of CPU time required to obtain this result, the number of “real” milliseconds required for the result, and the number of milliseconds of CPU time (included in the first result) spent on garbage collection.

The reliability of the timing numbers depends on the platform; see §15.1.2 for more information on time accounting. If multiple MzScheme threads are running, then the reported time may include work performed by other threads.

The `time` syntactic form reports timing information directly to the current output port:

- `(time` *`expr`*`)` times the evaluation of *`expr`*, printing timing information to the current output port. The result of the `time` expression is the result of *`expr`*.

15.2 Operating System Processes

`(subprocess` *`stdout-output-port stdin-input-port stderr-output-port command-path arg-string`* `···)` creates a new process in the underlying operating system to execute *`command-path`* asynchronously. The *`command-path`* argument is a path to a program executable, and the *`arg-string`*s are command-line arguments for the program. Under Unix and Mac OS X, command-line arguments are passed as byte strings using the current locale’s encoding (see §1.2.3).

Under Windows, the first *`arg-string`* can be `'exact`, which triggers a Windows-specific hack: the second *`arg-string`* is used exactly as the command-line for the subprocess, and no additional *`arg-string`*s can be supplied. Otherwise, a command-line string is constructed from *`command-path`* and *`arg-string`* so that a typical Windows console application can parse it back to an array of arguments.[2] If `'exact` is provided on a non-Windows platform, the `exn:fail:contract` exception is raised.

Unless it is `#f`, *`stdout-output-port`* is used for the launched process’s standard output, *`stdin-input-port`* is used for the process’s standard input, and *`stderr-output-port`* is used for the process’s standard error. All provided ports must be file-stream ports. Any of the ports can be `#f`, in which case a system pipe is created and returned by `subprocess`. For each port that is provided, no pipe is created and the corresponding returned value is `#f`.

The `subprocess` procedure returns four values:

[2]For information on the Windows command-line conventions, search for “command line parsing” at `http://msdn.microsoft.com/`.

- a subprocess value representing the created process;
- an input port piped from the process's standard output, or `#f` if *`stdout-output-port`* was a port;
- an output port piped to the process standard input, or `#f` if *`stdin-input-port`* was a port;
- an input port piped from the process's standard error, or `#f` if *`stderr-output-port`* was a port.

Important: All ports returned from `subprocess` must be explicitly closed with `close-input-port` and `close-output-port`.

The returned ports are file-stream ports (see §11.1.6), and they are placed into the management of the current custodian (see §9.2). The `exn:fail` exception is raised when a low-level error prevents the spawning of a process or the creation of operating system pipes for process communication.

A subprocess value can be used to obtain further information about the process:

- `(subprocess-wait` *`subprocess`*`)` blocks until the process terminates, then returns void.
- `(subprocess-status` *`subprocess`*`)` returns `'running` if the process is still running, or its exit code otherwise. The exit code is an exact integer, and `0` typically indicates success. If the process terminated due to a fault or signal, the exit code is non-zero.
- `(subprocess-kill` *`subprocess force?`*`)` terminates the subprocess if *`force?`* is true and if the process still running, then returns void. If an error occurs during termination, the `exn:fail` exception is raised.

  If *`force?`* is `#f` under Unix and Mac OS X, the subprocess is sent an interrupt signal instead of a kill signal (and the subprocess might handle the signal without terminating). Under Windows, no action is taken when *`force?`* is `#f`.
- `(subprocess-pid` *`subprocess`*`)` returns the operating system's numerical ID for the process (if any), valid only as long as the process is running. The ID is an exact integer.
- `(subprocess?` *`v`*`)` returns `#t` if *`v`* is a subprocess value, `#f` otherwise.

MzLib provides procedures for executing shell commands (as opposed to directly executing a program); see Chapter 38 of *PLT MzLib: Libraries Manual* for details.

15.3 Windows Actions

`(shell-execute` *`verb-string target-string parameters-string dir-path show-mode-symbol`*`)` performs the action specified by *`verb-string`* on *`target-string`* in Windows. For example,

```
(shell-execute #f "http://www.plt-scheme.org" "" (current-directory) 'sw_shownormal)
```

opens the PLT Scheme home page in a browser window. For platforms other than Windows, the `exn:fail:unsupported` exception is raised.

The *`verb-string`* can be `#f`, in which case the operating system will use a default verb. Common verbs include `"open"`, `"edit"`, `"find"`, `"explore"`, and `"print"`.

The *`target-string`* is the target for the action, usually a filename path. The file could be executable, or it could be a file with a recognized extension that can be handled by an installed application.

The *`parameters-string`* argument is passed on to the system to perform the action. For example, in the case of opening an executable, the *`parameters-string`* is used as the command line (after the executable name).

The `dir-path` is used as the current directory when performing the action.

The `show-mode-symbol` sets the display mode for a Window affected by the action. It must be one of the following symbols; the description of each symbol's meaning is taken from the Windows API documentation.

- `'sw_hide` or `'SW_HIDE` — Hides the window and activates another window.
- `'sw_maximize` or `'SW_MAXIMIZE` — Maximizes the window.
- `'sw_minimize` or `'SW_MINIMIZE` — Minimizes the window and activates the next top-level window in the z-order.
- `'sw_restore` or `'SW_RESTORE` — Activates and displays the window. If the window is minimized or maximized, Windows restores it to its original size and position.
- `'sw_show` or `'SW_SHOW` — Activates the window and displays it in its current size and position.
- `'sw_showdefault` or `'SW_SHOWDEFAULT` — Uses a default.
- `'sw_showmaximized` or `'SW_SHOWMAXIMIZED` — Activates the window and displays it as a maximized window.
- `'sw_showminimized` or `'SW_SHOWMINIMIZED` — Activates the window and displays it as a minimized window.
- `'sw_showminnoactive` or `'SW_SHOWMINNOACTIVE` — Displays the window as a minimized window. The active window remains active.
- `'sw_showna` or `'SW_SHOWNA` — Displays the window in its current state. The active window remains active.
- `'sw_shownoactivate` or `'SW_SHOWNOACTIVATE` — Displays a window in its most recent size and position. The active window remains active.
- `'sw_shownormal` or `'SW_SHOWNORMAL` — Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position.

If the action fails, the `exn:fail` exception is raised. If the action succeeds, the result is `#f`. In future versions of MzScheme, the result may be a subprocess value (see §15.2) if the operating system did returns a process handle (but if a subprocess value is returned, its process ID will be `0` instead of the real process ID).

15.4 Operating System Environment Variables

`(getenv name-string)` gets the value of an operating system environment variable. The `name-string` argument cannot contain a null character; if an environment variable named by `name-string` exists, its value is returned (as a string); otherwise, `#f` is returned.

`(putenv name-string value-string)` sets the value of an operating system environment variable. The `name-string` and `value-string` arguments are strings that cannot contain a null character; the environment variable named by `name-string` is set to `value-string`. The return value is `#t` if the assignment succeeds, `#f` otherwise.

15.5 Runtime Information

`(system-type [mode])` returns information about the operating system, build mode, or machine for a running MzScheme. The `mode` argument must be either `'os` (the default), `'gc`, `'link`, `'so-suffix`, or `'machine`. In `'os` mode, the possible symbol results are:

- `'unix`
- `'windows`
- `'macosx`

In `'gc` mode, the possible symbol results are:

- `'cgc`
- `'3m`

In `'link` mode, the possible symbol results are:

- `'static` (Unix)
- `'shared` (Unix)
- `'dll` (Windows)
- `'framework` (Mac OS X)

(Future ports of MzScheme may expand the list of `'os`, `'gc`, and `'link` results.)

In `'so-suffix` mode, then the result is a byte string that represents the file extension used for shared objects on the current platform. The byte string starts with a period, so it is suitable as a second argument to `path-replace-suffix`.

In `'machine` mode, then the result is a string, which contains further details about the current machine in a platform-specific format.

`(system-language+country)` returns a string to identify the current user's language and country. Under Unix and Mac OS X, the string is five characters: two lowercase ASCII letters for the language, an underscore, and two uppercase ASCII letters for the country. Under Windows, the string can be arbitrarily long, but the language and country are in English (all ASCII letters or spaces) separated by an underscore. Under Unix, the result is determined by checking the **LC_ALL**, **LC_TYPE**, and **LANG** environment variables, in that order (and the result is used if the environment variable's value starts with two lowercase ASCII letters, an underscore, and two uppercase ASCII letters, followed by either nothing or a period). Under Windows and Mac OS X, the result is determined by system calls.

`(system-library-subpath [`*`gc-symbol-or-false`*`])` returns a relative directory path string. This string can be used to build paths to system-specific files. For example, when MzScheme is running under Solaris on a Sparc architecture, the subpath starts `"sparc-solaris"`, while the subpath for Windows on an i386 architecture starts `"win32\\i386"`. The optional *`gc-symbol-or-false`* argument specifies the relevant garbage-collection variant, which one of the possible results of `(system-type 'gc)`: `'cgc` or `'3m`. The *`gc-symbol-or-false`* argument defaults to the result of `(system-type 'gc)`, but it can also be `#f`, in which case the result is independent of the garbage-collection variant.

`(version)` returns an immutable string indicating the currently executing version of MzScheme.

`(banner)` returns an immutable string for MzScheme's start-up banner text (or the banner text for an embedding program, such as MrEd). The banner string ends with a newline.

`(vector-set-performance-stats!` *`mutable-vector`* `[`*`thread`*`])` sets elements in *`mutable-vector`* to report current performance statistics. If *`thread`* is specified, a particular set of thread-specific statistics are reported, otherwise a different set of global statics are reported.

For global statistics, up to 8 elements are set in the vector, starting from the beginning. (In future versions of MzScheme, additional elements will be set.) If *`mutable-vector`* has n elements where $n < 8$, then the n elements are set to the first n performance-statistics values. The reported statistics values are as follows, in the order that they are set within *`mutable-vector`*:

- `0`: The same value as returned by `current-process-milliseconds` (see §15.1.2).
- `1`: The same value as returned by `current-milliseconds` (see §15.1.2).
- `2`: The same value as returned by `current-gc-milliseconds` (see §15.1.2).
- `3`: The number of garbage collections performed since start-up.
- `4`: The number of thread context switches performed since start-up.
- `5`: The number of internal stack overflows handled since start-up.
- `6`: The number of threads currently scheduled for execution (i.e., threads that are running, not suspended, and not unscheduled due to a synchronization).
- `7`: The number of syntax objects read from compiled code since start-up.
- `8`: The number of hash-table searches performed.
- `9`: The number of additional hash slots searched to complete hash searches (using double hashing).

For thread-specific statistics, up to 4 elements are set in the vector:

- `0`: `#t` if the thread is running, `#f` otherwise (same result as `thread-running?`).
- `1`: `#t` if the thread has terminated, `#f` otherwise (same result as `thread-dead?`).
- `2`: `#t` if the thread is currently blocked on a synchronizable event (or sleeping for some number of milliseconds), `#f` otherwise.
- `3`: The number of bytes currently in use for the thread's continuation.

16. Library Collections and MzLib

A *library* is `module` declaration for use by multiple programs. MzScheme provides a mechanism for grouping libraries into *collections* that can be easily distributed and easily added to a local MzScheme installation. A collection is normally installed into a directory named **collects** that is in the same directory as the MzScheme executable.[1] Each installed collection is represented as a subdirectory within the **collects** directory.

Client programs incorporate a library by using a module path of the form `(lib` *`library-file-path collection`* `···)`. For example, the following module uses the **match.ss** library module from the default **mzlib** collection, the **getinfo.ss** library module from the **setup** collection, and the **cards.ss** library module from the **games** collection's **cards** subcollection:

```
(module my-game mzscheme
  (require (lib "match.ss")
           (lib "getinfo.ss" "setup")
           (lib "cards.ss" "games" "cards"))
  ....)
```

In general, `(lib` *`library-file-path collection`* `···)` accesses the module in the file *`library-file-path`* in the collection named by the first *`collection`*, where both *`library-file-path`* and *`collection`* are literal strings that will be used as elements in a path. If additional *`collection`* strings are provided, they are used to form a path into a subcollection. If the *`collection`* arguments are omitted, the library is accessed in the **mzlib** collection.

The **info.ss** library in a collection is special by convention. This library is used to provide information about the collection to **mzc** (the MzScheme compiler) or MrEd. For more information see *PLT mzc: MzScheme Compiler Manual* and *PLT MrEd: Graphical Toolbox Manual*.

There is usually one standard **collects** directory, but MzScheme supports any number of directories containing collections. The search path for collections is determined by the `current-library-collection-paths` parameter (see §7.9.1.6). The list of paths in `current-library-collection-paths` is searched from first to last to locate a collection. To find a sub-collection, the enclosing collection is first found; if the sub-collection is not present in the found enclosing collection, then the search continues by looking for another instance of the enclosing collection, and so on. In other words, the directory tree for each element in the search path is spliced together with the directory trees of other path elements. (The "splicing" of tress applies only to directories; a file within a collection is found only within the first instance of the collection.)

The value of the `current-library-collection-paths` parameter is initialized by the stand-alone version of MzScheme to the result of `(find-library-collection-paths)`.[2] The `find-library-collection-paths` procedure produces a list of paths as follows:

- The path produced by `(build-path (find-system-path 'addon-dir) (version) "collects")` is the first element of the default collection path list, unless the value of the `use-user-specific-search-paths` parameter is `#f`.

[1]In the PLT distribution of MzScheme for Unix, the **collects** directory is in the top-level **plt** directory, rather than with the platform-specific binary in **plt/bin**.

[2]MrEd initializes the `current-library-collection-paths` parameter in the same way.

- If the executable embeds a list of search paths, they are included (in order) after the first element in the default collection path list. Embedded relative paths are included only when the corresponding directory exists relative to the executable.
- If the directory specified by `(find-system-path 'collects-dir)` is absolute, or if it is relative (to the executable) and it exists, then it is added to the end of the default collection path list.
- If the **PLTCOLLECTS** environment variable is defined, it is combined with the default list using `path-list-string->path-list` (see §11.3.2). If it is not defined, the default collection path list (as constructed by the first three bullets above) is used directly.

The path produced by `(find-system-path 'collects-dir)` is normally embedded in an executable; in stand-alone MzScheme (or MrEd), it can be overridden via a `--collects` or `-X` command-line flag.

`(collection-path` *`collection`* $\cdots^{1}$`)` returns the path containing the libraries of *`collection`*; if the collection is not found, the `exn:fail:filesystem` exception is raised.

MzScheme is distributed with a standard collection of utility libraries with MzLib as the representative library. The name of this collection is **mzlib**, so the libraries are distributed in a **mzlib** subdirectory of the **collects** library collection directory. MzLib is described in *PLT MzLib: Libraries Manual*.

17. Running MzScheme

The stand-alone version of MzScheme accepts a number of command-line options.

MzScheme accepts the following options:

- **Startup file and expression options:**
  - `-e` *`expr`* or `--eval` *`expr`* : Evaluates *`expr`* after MzScheme starts.
  - `-f` *`file`* or `--load` *`file`* : Loads *`file`* after MzScheme starts.
  - `-d` *`file`* or `--load-cd` *`file`* : Uses `load/cd` to load *`file`* after MzScheme starts.
  - `-t` *`file`* or `--require` *`file`* : Requires *`file`* after MzScheme starts.
  - `-F` or `--Load` : Loads each remaining argument as a file after MzScheme starts.
  - `-D` or `--Load-cd` : Loads each remaining argument as a file using `load/cd` after MzScheme starts.
  - `-T` or `--Require` : Requires each remaining argument as a file after MzScheme starts.
  - `-l` *`file`* or `--mzlib` *`file`*: Requires the MzLib library *`file`* after MzScheme starts.
  - `-L` *`file collect`* : Requires the library *`file`* in the collection *`collect`* after MzScheme starts.
  - `-M` *`collect`* : Requires the library ***`collect`*.ss** in the collection *`collect`* after MzScheme starts.
  - `-p` *`file user package`* : Requires a PLaneT library whose module path is `(planet` *`file`* `(`*`user package`*`))`.
  - `-P` *`name user`* : Requires a PLaneT library, a shorthand for `-p` *`name`*`.ss` *`user name`*`.plt`.
  - `-r` *`file`* or `--script` *`file`* : Use this option for MzScheme-based scripts. It mutes the startup banner printout, suppresses the `read-eval-print` loop, and loads *`file`* after MzScheme starts. No argument after *`file`* is treated as a switch. The `-r` or `--script` switch is a shorthand for `-fmv-`.
  - `-i` *`file`* or `--script-cd` *`file`* : Same as `-r` *`file`* or `--script` *`file`*, except that the current directory is changed to *`file`*'s directory before it is loaded. The `-i` or `--script-cd` switch is a shorthand for `-dmv-`.
  - `-u` *`file`* or `--require-script` *`file`* : Same as `-r` *`file`* or `--script` *`file`*, except that *`file`* is `require`d instead of `load`ed. The `-u` or `--require-script` switch is a shorthand for `-tmv-`.
  - `-w` or `--awk` : Loads the **awk.ss** library after MzScheme starts.
  - `-k` *`n m`* : Loads code embedded in the executable from file position *`n`* to *`m`* after MzScheme starts. This option is useful for creating a stand-alone binary by appending code to the normal MzScheme executable. See *PLT mzc: MzScheme Compiler Manual* for more details.
  - `-C` or `--main` : Like `-r`, then calls the function bound to *`main`* in the top-level environment. The argument to *`main`* is a list of immutable strings; the first string is the path of the file that was loaded, and the rest of the list contains leftover command-line arguments (the ones installed in `current-command-line-arguments`). The *`main`* function is called only if no previous evaluations or loads resulted in an uncaught exception.
- **Initialization options:**
  - `-X` *`dir`* or `--collects` *`dir`* : Sets *`dir`* as the path to the main collection of libraries (and makes `(find-system-path 'collects-dir)` produce *`dir`*).
  - `-S` *`dir`* or `--search` *`dir`* : Adds *`dir`* to the library collection search path (after a user-specific directory, if any, and before the main collection directory).
  - `-U` or `--no-user-path` : Omits paths in the search for collections, C libraries, etc. More specifically, this option initializes the `use-user-specific-search-paths` parameter to `#f`.

  - `-x` or `--no-lib-path` : Suppresses the initialization of `current-library-collection-paths` (as described in Chapter 16).
  - `-N file` or `--name file` : sets the name of the executable as reported by `(find-system-path 'run-file)` to *`file`*. Also, *`program`* is initially defined as *`file`*.
  - `-q` or `--no-init-file` : Suppresses loading the user's initialization file, as described below.
  - `-A` or `--no-argv` : Suppresses the definition of *`argv`* and *`program`*, as described below.
  - `-j` or `--no-jit` : Disables the native-code just-in-time compiler, setting the `eval-jit-enabled` parameter to `#f`.

- **Language setting options:**
  - `-Q` or `--prim` : Initializes the top-level environment with `(require mzscheme)`, which improves performance of non-`module` programs by allowing top-level bindings to be recognized. In this mode, primitive names can be re-defined in the top-level environment, but re-definition affects only expressions compiled after the re-definition.
  - `-g` or `--case-sens` : Makes the reader initially case-sensitive (the default).
  - `-G` or `--case-insens` : Makes the reader initially case-insensitive.
  - `-s` or `--set-undef` : Creates an initial namespace where `set!` will successfully mutate an undefined global variable (implicitly defining it).

- **Miscellaneous options:**
  - `--` : No argument following this switch is used as a switch.
  - `-m` or `--mute-banner` : Suppresses the startup banner text produced by `-v`.
  - `-v` or `--version` : Suppresses the `read-eval-print` loop.
  - `-h` or `--help` : Shows information about MzScheme's command-line options and then exits, ignoring other options.
  - `--persistent` : Catches the SIGDANGER (low page space) signal and ignores it (AIX only).

17.1 Command-Line Conventions

Extra arguments following the last option are available from the `current-command-line-arguments` parameter (see §7.9.1.6) as an immutable vector of immutable strings. The name used to start MzScheme is available from the `find-system-path` procedure (see §11.3.2) using `'exec-file`. In addition, unless `-A` is specified, the argument vector is put into the global variable `argv`, and the name used to start MzScheme is put into the global variable `program` as a path.

Multiple single-letter switches (the ones preceded by a single dash) can be collapsed into a single switch by concatenating the letters, as long as the first switch is not `--`. The arguments for each switch are placed after the collapsed switches (in the order of the switches). For example,

```
  -vfme file expr
```

and

```
  -v -f file -m -e expr
```

are equivalent. If a collapsed `--` appears before other collapsed switches in the same collapsed set, it is implicitly moved to the end of the collapsed set.

17.2 Executable Name

If the MzScheme executable is given a name of the form **scheme-*`dialect`***, then the command line is effectively prefixed with

```
  -qAeC '(require (lib "init.ss" "script-lang" "dialect"))'
```

The first actual command-line argument thus serves as a file to load. The file is loaded into a namespace that is initialized by the `dialect`-specific **init.ss** library. The loaded file should define `main`, which is called with command-line arguments—starting with the loaded file name—as a list of immutable strings.

17.3 Initialization

The `current-library-collection-paths` parameter is initialized (as described in Chapter 16) before any expression or file is evaluated or loaded, unless the `-x` or `--no-lib-path` option is specified.

Unless the `-q` or `--no-init-file` option is specified, a user initialization file is loaded after `current-library-collection-paths` parameter is initialized and before any other expression or file is evaluated or loaded. The path to the user initialization file is obtained from MzScheme's `find-system-path` procedure using `'init-file`.

Expressions and files are evaluated and loaded in order that they are provided on the command line, including calls to main implied by `--main`, embeddings loaded by `-k`, and so on. If an uncaught exception occurs, the remaining expressions and files are skipped. The thread that loads the files and evaluates the expressions is the *main thread*. When the main thread terminates (or is killed), the MzScheme process exits.

After the command-line files and expressions are loaded and evaluated, the main thread calls `read-eval-print-loop`, unless the `-v`, `--version`, `-r`, `--script`, `-i`, or `--script-cd` option is specified.

The exit status for the MzScheme process indicates an error if an error occurs evaluating or loading a command-line expression or file and `read-eval-print-loop` is not called afterwards, or if the default exit handler is called with an exact integer between 1 and 255.

18. Writing and Running Scripts

Under Unix, a Scheme file can be turned into an executable script using the shell's `#!` convention. The first two characters of the file must be `#!`, and the remainder of the first line must be a command to execute the script. For some platforms, the total length of the first line is restricted to 32 characters.

The simplest script format uses an absolute path to a **mzscheme** executable, followed by `-qr`. For example, if **mzscheme** is installed in **/usr/plt/bin**, then a file containing the following text acts as a "hello world" script:

```
#! /usr/plt/bin/mzscheme -qr
(display "Hello, world!")
(newline)
```

In particular, if the above is put into a file **hello** and the file is made executable (e.g., with **chmod a+x hello**), then typing **./hello** at the shell prompt will produce the output "Hello, world!".

Instead of specifying a complete path to the **mzscheme** executable, an alternative is to require that **mzscheme** is in the user's command path, and then "trampoline" with **/bin/sh**:

```
#! /bin/sh
#|
exec mzscheme -qr "$0" ${1+"$@"}
|#
(display "Hello, world!")
(newline)
```

The effect is the same, because `#` starts a one-line comment to **/bin/sh**, but `#|` starts a block comment to MzScheme. Finally, calling **mzscheme** with **exec** causes the MzScheme process to replace the **/bin/sh** process.

To implement a script inside `module`, use `-qu` instead of `-qr`:

```
#! /usr/plt/bin/mzscheme -qu
(module hello mzscheme
  (display "Hello, world!")
  (newline))
```

The `-qr` command-line flag to MzScheme is an abbreviation for the `-q` flag followed by the `-r` flag. As detailed in Chapter 17, `-q` skips the loading of ~/**.mzschemerc**, while `-r` suppresses MzScheme's startup banner, suppresses the read-eval-print loop, and loads the specified file. In the first example above, the file for `-r` is supplied by the shell's `#!` handling: it automatically puts the name of the executed script at the end of the `#!` line. In the second example, the script file name is supplied explicitly with `"$0"`. The `-qu` flag is similarly an abbreviation for `-q` followed by `-u`, which acts like `-r` except that it `require`s the script file instead of `load`ing it.

If command-line arguments are supplied to a shell script, the shell attaches them as extra arguments to the script command. Among its other jobs, the `-r` or `-u` flag ensures that the extra arguments are not interpreted by MzScheme, but instead put into the `current-command-line-arguments` parameter as a vector of strings. For example, the following **mock** script prints each command-line argument back on its own line:

```
#! /usr/plt/bin/mzscheme -qu
```

```
(module mock mzscheme
  (for-each (lambda (arg)
              (display arg)
              (newline))
            (vector->list (current-command-line-arguments))))
```

Thus, **mock a b c** would print “a”, “b”, and “c”, each on its own line. The **/bin/sh** version is similar:

```
#! /bin/sh
#|
exec mzscheme -qu "$0" ${1+"$@"}
|#
(module mock mzscheme
  (for-each (lambda (arg)
              (display arg)
              (newline))
            (vector->list (current-command-line-arguments))))
```

The `${1+"$@"}` part of the **mzscheme** command line copies all shell script arguments to MzScheme for `current-command-line-arguments`.

For high-quality scripts, use the **cmdline** MzLib library to parse command-line arguments (see Chapter 10 of *PLT MzLib: Libraries Manual*). The following **hello2** script accepts a `--chinese` flag to produce Chinese pinyin output. Due to the built-in functionality of the `command-line` form, the script also accepts a `--help` or `-h` flag that produces detailed help on the available command-line options:

```
#! /bin/sh
#|
exec mzscheme -qu "$0" ${1+"$@"}
|#
(module hello2 mzscheme
  (require (lib "cmdline.ss"))

  (define chinese? #f)

  (command-line
   "hello2"
   (current-command-line-arguments)
   (once-each
    [("--chinese") "Chinese output"
     (set! chinese? #t)]))

  (display (if chinese?
               "Nihao, shijie!"
               "Hello, world!"))
  (newline))
```

19. Honu

Honu is a family of languages built on top of MzScheme. Honu syntax resembles Java, instead of Scheme. Like Scheme, however, Honu has no fixed syntax. Honu supports extensibility through macros and a base syntax of H-expressions, which are analogous to S-expressions.

The MzScheme reader incorporates an H-expression reader, and MzScheme's printer also supports printing values in Honu syntax. The reader can be put into H-expression mode either by including `#hx` or `#honu` in the input stream, or by calling `read-honu` or `read-honu-syntax` instead of `read` or `read-syntax`:

- `(read-honu` [*`input-port`*]`)` is the same as calling `read` with the same arguments, but with `#hx` implicitly in the stream at the start of the read.
- `(read-honu-syntax` [*`source-name-v input-port`*]`)` is the same as calling `read-syntax` with the same arguments, but with `#hx` implicitly in the stream at the start of the read.

Similarly, `print`[1] produces Honu output when the `print-honu` parameter is set to `#t`.

When the reader encounters `#hx`, it reads a single H-expression, and it produces an S-expression that encodes the H-expression. Except for atomic H-expressions, evaluating this S-expression as Scheme is unlikely to succeed. In other words, H-expressions are not intended as a replacement for S-expressions to represent Scheme code.

When the reader encounters `#honu`, it reads H-expressions repeatedly until an end-of-file is encountered. The collected H-expression results are wrapped with `(module` *`id`* `(lib "honu-module.ss" "honu-module") ....)`, where *`id`* is generated as described below. The **honu-module.ss** module defines `#%module-begin` to parse S-expressions that encode H-expressions; expanding the module produces a Scheme program that corresponds to the H-expression-based Honu program in the original input stream. Thus, a file that starts with `#honu` can define a module to be `require`d in a Scheme module or another Honu module.

In the `module` wrapper for `#honu`, the *`id`* is derived from the read port's name: if the port's name is a symbol, then it is used as *`id`*; if the port's name is a path, then the last element of the path is converted to a symbol and used as *`id`*; otherwise, `'unknown` is used.

The **honu-module.ss** module and Honu language dialects are documented elsewhere. In principle, MzScheme's parsing and printing of H-expressions is independent of the Honu language, so it is currently documented here.

19.1 Honu Input Parsing

Ignoring whitespace, an H-expression is either

- a number (see §19.1.1);
- an identifier (see §19.1.2);

[1] More precisely, the default print handler.

- a string (see §19.1.3);
- a character (see §19.1.4);
- a sequence of H-expressions between parentheses (see §19.1.5);
- a sequence of H-expressions between square brackets (see §19.1.5);
- a sequence of H-expressions between curly braces (see §19.1.5);
- a comment followed by an H-expression (see §19.1.6);
- `#;` followed by two H-expressions (see §19.1.6);
- `#hx` followed by an H-expression;
- `#sx` followed by an S-expression (see §11.2.4).

Within a sequence of H-expressions, a sub-sequence between angle brackets is represented specially (see §19.1.5).

Whitespace for H-expressions is as in Scheme: any character for which `char-whitespace?` returns true counts as a whitespace.

19.1.1 Numbers

The syntax for Honu numbers is the same as for Java. The S-expression encoding of a particular H-expression number is the obvious Scheme number.

19.1.2 Identifiers

The syntax for Honu identifiers is the union of Java identifiers plus semicolon (`;`), comma (`,`), and a set of operator identifiers. An *operator identifier* is any combination of the following characters:

```
+ - _ = ?  :  < > .  !  % ^ & * / ~ |
```

The S-expression encoding of an H-expression identifier is the obvious Scheme symbol.

Input is parsed to form maximally long identifiers. For example, the input `int->int;` is parsed as four H-expressions: `int`, `->`, `int`, and `;`.

19.1.3 Strings

The syntax for an H-expression string is exactly the same as for an S-expression string, and an H-expression string is represented by the obvious Scheme string.

19.1.4 Characters

The syntax for an H-expression character is the same as for an H-expression string that has a single content character, except that a single quote (`'`) surrounds the character instead of double quotes (`"`). The S-expression representation of an H-expression character is the obvious Scheme character.

19.1.5 Parentheses, Brackets, and Braces

A parenthesized (), bracketed [], or braced {} H-expression sequence is represented by a Scheme list. The first element of the list is `'#%parens` for a parenthesized sequence, `'#%brackets` for a brackets sequence, or `'#%braces` for a braced sequence. The remaining elements are the Scheme representation for the parenthesized, bracketed, or braced H-expressions in order.

In an H-expression sequence, when a less-than sign (<) is followed by a greater-than sign (>), and when nothing between the less-than and greater-than signs is an immediate symbol containing an equal sign (=), ampersand (`&`), or vertical bar (|), then the sub-sequence is represented by a Scheme list that starts with `'#%angles` and continues with the elements of the sub-sequence between the less-than and greater-than signs (exclusive). This reprsentation is applied recursively, so that angle brackets can be nested. An angle-bracketed sequence by itself is not a single H-expression, since the less-than sign by itself is a single H-expression; the angle-bracket conversion is performed only when representing sequences of H-expressions. Also, symbols with an equal sign, ampersand, or vertical bar prevent angle-bracket formation, because they correspond to operators that normally have lower or equal precedence compared to less-than and greater-than.

19.1.6 Comments

An H-expression comment starts with either `//` or `/*`. In the former case, the comment runs until a linefeed or return. In the second case, the comment runs until `*/`, but `/* .... */` comments can be nested. Comments are treated like whitespace.

A `#;` starts an H-expression comment, as in Scheme. It is followed by an H-expression to be treated as whitespace. Note that `#;` is equivalent to `#sx#;#hx`.

19.2 Honu Output Printing

Some Scheme values have a standard H-expression representation. For values with no H-expression representation but with a `read`able S-expression form, the MzScheme printer produces an S-expression prefixed with `#sx`. For values with neither an H-expression form nor a `read`able S-expression form, then printer produces output of the form `#<...>`, as in Scheme mode. The `print-honu` parameter (see §7.9.1.4) controls whether Mzscheme's printer produces Scheme or Honu output.

The values with H-expression forms are as follows:

- Every real number has an H-expression form, although the representation for an exact, non-integer rational number is actually three H-expressions, where the middle H-expression is `/`.
- Every character string is represented the same in H-expression form as its S-expression form.
- Every character is represented like a single-character string, but (1) using a single quote as the delimiter instead of double quotes, and (2) protecting a single-quote character content a backslash instead of protecting double-quote character content.
- A list is represented with the H-expression sequence `list(`*v*`, ···)`, where each *v* is the representation of each element of the list.
- A pair that is not a list is represented with the H-expression sequence `cons(`*v1*`, `*v2*`)`, where *v1* and *v2* are the representations of the pair elements.
- A vector's representation depends on the value of the `print-vector-length` parameter (see §7.9.1.4). If it is true, the vector is represented with the H-expression sequence `vectorN(`*n*`, `*v*`, ···)`, where *n* is the length of the vector and each *v* is the representation of each element of the vector, and multiple instances of the same

value at the end of the vector are represented by a single `v`. If `print-vector-length` is set to false, the vector is represented with the H-expression sequence `vector(v, ···)`, where each `v` is the representation of each element of the vector.

- The empty list is represented as the H-expression `null`.
- True is represented as the H-expression `true`.
- False is represented as the H-expression `false`.

20. Platform-Specific Path Conventions

20.1 Unix and Mac OS X Path Conventions

In Unix and Mac OS X paths, a forward slash ("/") seperates elements of the path, a period (".") as a path element always means the directory indicated by preceding path, and two periods ("..") as a path element always means the parent of the directory indicated by the preceding path. A path that starts with a tilde ("~") indicates a user's home directory; the username follows the tilde (before a slash or the end of the path), where a tilde by itself indicates the home directory of the current user. No other character or byte has a special meaning within a path. Multiple adjacent slashes ("/") are equivalent to a single slash (i.e., they act as a single path separator).

A path root is either / or a home-directory specification starting with tilde ("~"). A relative path whose first element starts with a tilde ("~") is encoded by prefixing the path with period–slash ("./").

Any pathname that ends with a slash ("/") syntactically refers to a directory, as does any path whose last element is a single period (".") or double period (".."), or any path that contains only a root.

A Unix and Mac OS X path is *expanded* by replacing a home-directory specification (starting ~) with an absolute path, and by replacing multiple adjacent slashes with a single slash.

For `(bytes->path-element` *`bytes`*`)`, *`bytes`* can start with a tilde ("~"), and it is encoded as a literal part of the path element using a period–slash ("./") prefix. The *`bytes`* argument must not contain a slash ("/"), otherwise the `exn:fail:contract` exception is raised.

For `(path-element->bytes` *`path`*`)` or `(path-element->string` *`path`*`)`, if the bytes form of *`path`* starts with a period, slash, and tilde ("./~"), the period–slash ("./") prefix is not included in the result.

For `(build-path` *`base-path sub-path`* `···)`, when a *`sub-path`* starts with a period, slash, and tilde ("./~"), the period and slash are removed before adding the path. This conversion is performed because an initial sequence period–slash–tilde ("./~") is the canonical way of representing relative paths whose first element's name starts with a tilde.

For `(simplify-path` *`path use-filesystem?`*`)`, if *`path`* starts period–slash–tilde ("./~"), the leading period is the only indicator, and there are no redundant slashes, then *`path`* is returned.

For `(split-path` *`path`*`)` producing *`base`*, *`name`*, and *`must-be-dir?`*, the result *`name`* can start with period–slash–tilde ("./~") if the result would otherwise start with tilde ("~") and it is not the start of *`path`*. Furthermore, if *`path`* starts with period–slashes–tilde ("./~", with any non-zero number of "/"), then the period and slash are kept with the following element (i.e., they are not split separately).

Under Mac OS X, Finder aliases are zero-length files.

20.2 Windows Path Conventions

In general, a Windows pathname consists of an optional drive specifier and a drive-specific path. As noted in §11.3, a Windows path can be *absolute* but still relative to the current drive; such paths start with a forward slash or backslash separator and are not UNC paths or paths that start with \\**?**\.

A path that starts with a drive specification is *complete*. Roughly, a drive specification is either a Roman letter followed by a colon, a UNC path of the form \\***machine******volume***, or a \\**?**\ form followed by something other than **REL*****element*** or **RED*****element***. (Variants of \\**?**\ paths are described further below.)

MzScheme fails to implement the usual Windows path syntax in one way. Outside of MzScheme, a pathname **C:rant.txt** can be a drive-specific relative path. That is, it names a file **rant.txt** on drive **C:**, but the complete path to the file is determined by the current working directory for drive **C:**. MzScheme does not support drive-specific working directories (only a working directory across all drives, as reflected by the `current-directory` parameter; see §7.9.1.1). Consequently, MzScheme implicitly converts a path like **C:rant.txt** into **C:****rant.txt**.

- *MzScheme-specific:* Whenever a path starts with a drive specifier ***letter*:** that is not followed by a forward slash or backslash, a backslash is inserted as the path is expanded.

Otherwise, MzScheme follows standard Windows path conventions, but also adds \\**?****REL** and \\**?****RED** conventions to deal with paths inexpressible in the standard conventsion, plus conventions to deal with excessive backslashes in \\**?**\ paths.

In the following, ***letter*** stands for a Roman letter (case does not matter), ***machine*** stands for any sequence of characters that does not include backslashes or forward slashes and is not **?**, ***volume*** stands for any sequence of characters that does not include backslashes or forward slashes, and ***element*** stands for any sequence of characters that does not include backslashes.

- Trailing spaces and periods in a path element are ignored when the element is the last one in the path, unless the path starts with \\**?**\ or the element consists of only spaces and periods.
- The following special "files", which access devices, exist in all directories, case-insensitively, and with all possible endings after a period or colon, except in pathnames that start with \\**?**\: **NUL**, **CON**, **PRN**, **AUX**, **COM1**, **COM2**, **COM3**, **COM4**, **COM5**, **COM6**, **COM7**, **COM8**, **COM9**, **LPT1**, **LPT2**, **LPT3**, **LPT4**, **LPT5**, **LPT6**, **LPT7**, **LPT8**, **LPT9**.
- Except for \\**?**\ paths, forward slashes are equivalent to backslashes. Except for \\**?**\ paths and the start of UNC paths, multiple adjacent slashes and backslashes count as a single backslash. In a path that starts \\**?**\ paths, elements can be separated by either a single or double backslash.
- A directory can be accessed with or without a trailing separator. In the case of a non-\\**?**\ path, the trailing separator can be any number of forward slashes and backslashes; in the case of a \\**?**\ path, a trailing separator must be a single backslash, except that two backslashes can follow \\**?*****letter*:**.
- Except for \\**?**\ paths, a single period (**.**) as a path element means "the current directory", and a double period (**..**) as a path element means "the parent directory." Up-directory path elements (i.e., **..**) immediately after a drive are ignored.
- A pathname that starts \\***machine******volume*** (where a forward slash can replace any backslash) is a UNC path, and the starting \\***machine******volume*** counts as the drive specifier.
- Normally, a path element cannot contain any of the following characters:

  ```
  < > :  " / \ |
  ```

  Except for backslash, path elements containing these characters can be accessed using a \\**?**\ path (assuming that the underlying filesystem allows the characters).

- In a pathname that starts \\**?*****letter*:**\, the \\**?*****letter*:**\ prefix counts as the path's drive, as long as the path does not both contain non-drive elements and end with two consecutive backslashes, and as long as the path contains no sequence of three or more backslashes. Two backslashes can appear in place of the backslash before ***letter***. Forward slashes cannot be used in place of backslashes (but forward slashes can be used in element names, though the result generally does not name an actual directory or file).

- In a pathname that starts \\**?****UNC*****machine******volume***, the \\**?****UNC*****machine******volume*** prefix counts as the path's drive, as long as the path does not end with two consecutive backslashes, and as long as the path contains no sequence of three or more backslashes. Two backslashes can appear in place of the backslash before **UNC**, the backslash after **UNC**, and/or the backslash after ***machine***. The letters in the **UNC** part can be uppercase or lowercase, and forward slashes cannot be used in place of backslashes (but forward slashes can be used in element names).

- *MzScheme-specific:* A pathname that starts \\**?****REL*****element*** or \\**?****REL**\\***element*** is a relative path, as long as the path does not end with two consecutive backslashes, and as long as the path contains no sequence of three or more backslashes. This MzScheme-specific path form supports relative paths with elements that are not normally expressible in Windows paths (e.g., a final element that ends in a space). The **REL** part must be exactly the three uppercase letters, and forward slashes cannot be used in place of backslashes. If the path starts \\**?****REL****..** then for as long as the path continues with reptitions of **..**, each element counts as an up-directory element; a single backslash must be used to seperate the up-directory elements. As soon as a second backslash is used to separate the elements, or as soon as a non-**..** element is encountered, the remaining elements are all literals (never up-directory elements). When a \\**?****REL** path value is converted to a string (or when the path value is written or displayed), the string does not contain the starting \\**?****REL** or the immediately following backslashes; converting a path value to a byte string preserves the \\**?****REL** prefix.

- *MzScheme-specific:* A pathname that starts \\**?****RED*****element*** or \\**?****RED**\\***element*** is a drive-relative path, as long as the path does not end with two consecutive backslashes, and as long as the path contains no sequence of three or more backslashes. This MzScheme-specific path form supports drive-relative paths (i.e., absolute given a drive) with elements that are not normally expressible in Windows paths. The **RED** part must be exactly the three uppercase letters, and forward slashes cannot be used in place of backslashes. Unlike \\**?****REL** paths, a **..** element is always a literal path element. When a \\**?****RED** path value is converted to a string (or when the path value is written or displayed), the string does not contain the starting \\**?****RED** and it contains a single starting backslash; converting a path value to a byte string preserves the \\**?****RED** prefix.

Three additional MzScheme-specific rules provide meanings to character sequences that are otherwise ill-formed as Windows paths:

- *MzScheme-specific:* In a pathname of the form \\**?*****any***\\ where ***any*** is any non-empty sequence of characters other than ***letter*:** or ***letter*:**, the entire path counts as the path's (non-existent) drive.

- *MzScheme-specific:* In a pathname of the form \\**?*****any***\\\***elements***, where ***any*** is any non-empty sequence of characters and ***elements*** is any sequence that does not start with a backslash, does not end with two backslashes, and does not contain a sequence of three backslashes, then \\**?*****any***\\ counts as the path's (non-existent) drive.

- *MzScheme-specific:* In a pathname that starts \\**?**\ and does not match any of the patterns from the preceding bullets, \\**?**\ counts as the path's (non-existent) drive.

Outside of MzScheme, except for \\**?**\ paths, pathnames are typically limited to 259 characters. MzScheme internally converts pathnames to \\**?**\ form as needed to avoid this limit. The operating system cannot access files through \\**?**\ paths that are longer than 32,000 characters or so.

Where the above descriptions says "character," substitute "byte" for interpreting byte strings as paths. The encoding of Windows paths into bytes preserves ASCII characters, and all special characters mentioned above are ASCII, so all of the rules are the same.

Beware that the backslash path separator is an escape character in MzScheme strings. Thus, the path \\\**?**\\REL\\**..\\\..** as a string must be written `"\\\\?\\REL\\..\\\\.."`.

A path that ends with a directory separator syntactically refers to a directory. In addition, a path syntactcially refers to a directory if its last element is a same-directory or up-directory indicator (not quoted by a \\\**?**\\ form), or if it refers to a root.

Windows paths are expanded as follows: In paths that start \\\**?**\\, redundant backslashes are removed, an extra backslash is added in a \\\**?**\\REL if an extra one is not already present to separate up-directory indicators from literal path elements, and an extra backslash is similarly added after \\\**?\\RED** if an extra one is not already present. When \\\**?**\\ acts as the root and the path contains, to additional slashes (which might otherwise be redundant) are included after the root. For other paths, multiple slashes are converted to single slashes (except at the beginning of a shared folder name), a slash is inserted after the colon in a drive specification if it is missing.

For `(bytes->path-element` *`bytes`*`)`, forward slashes, colons, trailing dots, trailing whitespace, and special device names (e.g., "aux") in *`bytes`* are encoded as a literal part of the path element by using a \\\**?**\\REL prefix. The *`bytes`* argument must not contain a backslash ("\\"), otherwise the `exn:fail:contract` exception is raised.

For `(path-element->bytes` *`path`*`)` or `(path-element->string` *`path`*`)`, if the byte-string form of *`path`* starts with a \\\**?**\\REL, the prefix is not included in the result.

For `(build-path` *`base-path sub-path`* `···)`, trailing spaces and periods are removed from the last element of *`base-path`* and all but the last *`sub-path`* (unless the element consists of only spaces and peroids), except for those that start with \\\**?**\\. If *`base-path`* starts \\\**?**\\, then after each non-\\\**?**\\REL\\ and non-\\\**?\\RED**\\ *`sub-path`* is added, all slashes in the addition are converted to backslashes, multiple consecutive backslashes are converted to a single backslash, added **.** elements are removed, and added **..** elements are removed along with the preceding element; these conversions are not performed on the original *`base-path`* part of the result or on any \\\**?\\REL**\\ or \\\**?\\RED**\\ or *`sub-path`*. If a \\\**?\\REL**\\ or \\\**?\\RED**\\ *`sub-path`* is added to a non-\\\**?**\\ *`base-path`*, the the *`base-path`* (with any additions up to the \\\**?\\REL**\\ or \\\**?\\RED**\\ *`sub-path`*) is simplified and converted to a \\\**?**\\ path. In other cases, a backslash may be added or removed before combining paths to avoid changing the root meaning of the path (e.g., combining **//x** and **y** produces **/x/y**, because **//x/y** would be a UNC path instead of a drive-relative path).

For `(simplify-path` *`path use-filesystem?`*`)`, *`path`* is expanded, and if *`path`* does not start with \\\**?**\\, trailing spaces and periods are removed, a slash is inserted after the colon in a drive specification if it is missing, and a backslash is inserted after \\\**?**\\ as a root if there are elements and no extra backslash already. Otherwise, if no indicators or redundant separators are in *`path`*, then *`path`* is returned.

For `(split-path` *`path`*`)` producing *`base`*, *`name`*, and *`must-be-dir?`*, splitting a path that does not start with \\\**?**\\ can produce parts that start with \\\**?**\\. For example, splitting **C:/x /aux/** produces \\\**?**\\C:\x \\ and \\\**?\\REL\\aux**; the \\\**?**\\ is needed in these cases to preserve a trailing space after **x** and to avoid referring to the AUX device instead of an **aux** file.

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

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.

Index

+inf.0, 13
+nan.0, 13
,, 120
,@, 120
--, 192
--Load, 191
--Load-cd, 191
--Require, 191
--awk, 191
--case-insens, 192
--case-sens, 192
--collects, 191
--eval, 191
--help, 192
--load, 191
--load-cd, 191
--main, 191
--mute-banner, 192
--mzlib, 191
--name, 192
--no-argv, 192
--no-init-file, 192
--no-jit, 192
--no-lib-path, 192
--no-user-path, 191
--persistent, 192
--prim, 192
--require, 191
--require-script, 191
--script, 191
--script-cd, 191
--search, 191
--set-undef, 192
--version, 192
-A, 192
-C, 191
-D, 191
-F, 191
-G, 192
-L, 191
-M, 191
-N, 192
-P, 191
-Q, 192
-S, 191
-T, 191
-U, 191
-X, 191
-d, 191
-e, 191
-f, 191
-g, 192
-h, 192
-i, 191
-inf.0, 13
-j, 192
-k, 191
-l, 191
-m, 192
-nan.0, 13
-p, 191
-q, 192
-r, 191
-s, 192
-t, 191
-u, 191
-v, 192
-w, 191
-x, 192
., 120
..., 145
.mzschemerc, 137
:, 121
=>, 5
[ ], 71
#!, 121, 194
#", 119
#', 118
#,, 118
#,@, 118
#\backspace, 118
#\linefeed, 118
#\newline, 118
#\nul, 118
#\null, 118
#\page, 118
#\return, 118
#\rubout, 118
#\space, 118
#\tab, 118
#\vtab, 118
#;, 121
#<<, 119
#<undefined>, 12
#<void>, 12
#%, 121
#%app, 161
#%datum, 161

#%expression, 11
#%module-begin, 162
#%top, 161
#%top-interaction, 161
#%variable-reference, 11
#&, 118
#*n*=, 121, 123
#*n*#, 121, 123
#`, 118
#ci, 121
#cs, 121
#hash, 121
#hasheq, 121
#honu, 121
#hx, 121
#k+, 121
#k-, 121
#px, 121
#px#, 121
#reader, 122
#rx, 121
#rx#, 121
#sx, 121
#~, 121
\, 120
\', 119
\U*h*, 119
\U*hh*, 119
\U*hhh*, 119
\U*hhhh*, 119
\U*hhhhh*, 119
\U*hhhhhh*, 119
\U*hhhhhhh*, 119
\U*hhhhhhhh*, 119
\a, 119
\b, 119
\e, 119
\f, 119
\n, 119
o, 119
ooo, 119
oo, 119
\r, 119
\t, 119
\u*h*, 119
\u*hh*, 119
\u*hhh*, 119
\u*hhhh*, 119
\v, 119
\x*hh*, 119
\x*h*, 119
_loader.dll, 181
_loader.dylib, 181
_loader.so, 181
{ }, 72
‘, 120
'3m, 187

abort-current-continuation, 54
'aborts, 22
absolute-path?, 134
add1, 13
'addon-dir, 137
alarm-evt, 65
alarms, 65
all-defined, 40
all-defined-except, 40
all-except, 39
all-from, 40
all-from-except, 40
and, 6
andmap, 12
'any, 112
'any-one, 112
'append, 96
append!, 24
argv, 75, 192
arithmetic-shift, 14
arity, 26
arity-at-least, 25
arity-at-least-value, 25
arity-at-least?, 25
assoc, 24
assq, 24
assv, 24
AUX, 201

banner, 187
begin, 6, 9
begin-for-syntax, 155
begin0, 6
bignum, 13
'binary, 96
bitwise operators, 14
bitwise-and, 14
bitwise-ior, 14
bitwise-not, 14
bitwise-xor, 14
'block, 98
bound-identifier=?, 154
box, 24
box-immutable, 25
box?, 25
boxes, 24, 72, 118
    printing, 72, 122
break-enabled, 58
break-thread, 58, 63

breaks, *see* threads, breaking
Bruggeman, Carl, 145
build-path, 134
build-path/convention-type, 134
byte codes, 181
byte strings, 19, 119
  as ports, 97
  printing, 123
  reading to and writing from, 97
byte-pregexp, 89
byte-pregexp?, 90
byte-regexp, 89
byte-regexp?, 90
byte?, 19
bytes, 19
bytes, 19
bytes->immutable-bytes, 19
bytes->list, 19
bytes->path, 133
bytes->path-element, 133
bytes->string/latin-1, 20
bytes->string/locale, 20
bytes->string/utf-8, 20
bytes-append, 19
bytes-close-converter, 22
bytes-convert, 22
bytes-convert-end, 22
bytes-converter?, 23
bytes-copy, 19
bytes-copy!, 19
bytes-fill!, 19
bytes-length, 19
bytes-open-converter, 21
bytes-ref, 19
bytes-set!, 19
bytes-utf-8-index, 21
bytes-utf-8-length, 21
bytes-utf-8-ref, 21
bytes<?, 19
bytes=?, 19
bytes>?, 19
bytes?, 19

call-in-nested-thread, 63
call-with-break-parameterization, 58
call-with-composable-continuation, 55
call-with-continuation-barrier, 52
call-with-continuation-prompt, 54
call-with-current-continuation, 52
call-with-escape-continuation, 52
call-with-exception-handler, 47
call-with-input-file, 96
call-with-output-file, 96
call-with-parameterization, 77
call-with-semaphore, 64
call-with-semaphore/enable-break, 64
call/cc, 52
call/ec, 52
case sensitivity, 71
case-lambda, 10
'cc, 16
certificates
  syntax, 170
'certify-mode, 173
'cf, 16
'cgc, 187
channel-get, 64
channel-put, 64
channel-put-evt, 65
channel-try-get, 64
channel?, 64
char->integer, 15
char-alphabetic?, 16
char-blank?, 16
char-ci<=?, 16
char-ci<?, 16
char-ci=?, 16
char-ci>=?, 16
char-ci>?, 16
char-downcase, 16
char-foldcase, 17
char-general-category, 16
char-graphic?, 16
char-iso-control?, 16
char-lower-case?, 16
char-numeric?, 16
char-punctuation?, 16
char-symbolic?, 16
char-title-case?, 16
char-titlecase, 17
char-upcase, 16
char-upper-case?, 16
char-utf-8-length, 17
char-whitespace?, 16
char<=?, 16
char<?, 16
char=?, 16
char>=?, 16
char>?, 16
characters, 2, 15
  constants, 118
  printing, 122
Check Syntax, 166, 170
check-duplicate-identifier, 154
choice-evt, 67
'client, 83
'cn, 16

'co, 16
code points, 2
collect-garbage, 179
collection-path, 190
collections, 189
'collects-dir, 137
column numbers, 115
COM1, 201
COM2, 201
COM3, 201
COM4, 201
COM5, 201
COM6, 201
COM7, 201
COM8, 201
COM9, 201
command-line arguments, 74, 191
comments, 121
  comments
    S-expression, 121
communication, 63, 64
communications, 139
compilation handler, 73
compile, 73, 181
compile-allow-set!-undefined, 73
compile-enforce-module-constants, 73
compile-syntax, 182
compiled code, 72
compiled-expression?, 182
compiled-module-expression?, 176
compiling, 181
'complete, 22, 23
complete-path?, 134
complex, 13
CON, 201
concatenate strings, 24
cons-immutable, 24
context, 55
continuation-mark-set->context, 57
continuation-mark-set->list, 56
continuation-mark-set->list*, 57
continuation-mark-set-first, 57
continuation-mark-set?, 57
continuation-marks, 56
continuation-prompt-available?, 55
continuations, 52
  barrier crossings, 52
  barriers, 52
  escape, 52
'continues, 22, 23
control flow, 52
copy-file, 139
'cs, 16
curly braces, 71
current namespace, 73
current-break-parameterization, 58
current-code-inspector, 76
current-command-line-arguments, 75, 192
current-compile, 73
current-continuation-marks, 49, 56
current-custodian, 76, 83
current-directory, 71, 139
current-drive, 139
current-error-port, 71
current-eval, 73
current-evt-pseudo-random-generator, 76
current-gc-milliseconds, 184
current-inexact-milliseconds, 183
current-input-port, 71, 95
current-inspector, 76
current-library-collection-paths, 74, 189, 192, 193
current-load, 74
current-load-extension, 74
current-load-relative-directory, 74, 180, 182
current-load/use-compiled, 74
current-locale, 76
current-memory-use, 179
current-milliseconds, 14, 183
current-module-name-prefix, 76
current-module-name-resolver, 76
current-namespace, 73
current-output-port, 71, 95
current-parameterization, 77
current-preserved-thread-cell-values, 68
current-print, 73
current-process-milliseconds, 184
current-prompt-read, 73
current-pseudo-random-generator, 14, 76
current-reader-guard, 72
current-readtable, 72
current-seconds, 183
current-security-guard, 76, 82
current-thread, 63
current-thread-group, 76
current-thread-initial-stack-size, 77
current-write-relative-directory, 74
custodian-box-value, 85
custodian-box?, 85
custodian-limit-memory, 85
custodian-managed-list, 84
custodian-memory-accounting-available?, 84
custodian-require-memory, 85
custodian-shutdown-all, 84

custodian?, 84
custodians, 76, 83
cycles, 124

date, 183
date, 183
date-day, 183
date-dst?, 183
date-hour, 183
date-minute, 183
date-month, 183
date-second, 183
date-time-zone-offset, 183
date-week-day, 183
date-year, 183
date-year-day, 183
date?, 183
datum->syntax-object, 152
decimal input, 13
default-continuation-prompt-tag, 55
define, 7
define
    internal, 9
define-for-syntax, 155
define-struct, 29
define-syntax, 149
define-syntaxes, 160
define-values, 7
define-values-for-syntaxes, 155
'delete, 83
delete-directory, 139
delete-file, 138
'desk-dir, 137
directories
    contents, 139
    creating, 139
    current, 71, 139
    dates, 139
    deleting, 139
    moving, 139
    of currently loading file, 74, 180, 182
    paths, *see* paths
    permissions, 139
    renaming, 139
    root, 139
    testing, 139
directory-exists?, 139
directory-list, 139
'disappeared-binding, 166, 170
'disappeared-use, 166, 170
'dispatch-macro, 126
display, 124
display extensions, 122
division by inexact zero, 13
'dll, 187
'doc-dir, 137
dump-memory-stats, 179
Dybvig, Kent, 47, 145
dynamic-require, 44
dynamic-require-for-syntax, 44
dynamic-wind, 53

else, 5
'empty, 80
'enclosing-module-name, 162
environments
    top-level, 145, 155
eof, 95
eof-object?, 95
ephemeron-value, 177
ephemeron?, 177
ephemerons, 177
eq-hash-code, 27
eq?, 16, 37
'equal, 27
equal-hash-code, 27
equal?, 12, 25, 37
eqv?, 12, 13, 37
'error, 22, 96
error, 50
error display handler, 75
error escape handlers, 59, 61
error value conversion handler, 75
error-display-handler, 75
error-escape-handler, 59, 75
error-print-context-length, 75
error-print-source-location, 75
error-print-width, 75
error-value->string-handler, 75
errors, 48, 50, 75
    arity, 50
    mismatch, 50
    syntax, 51
    type, 50
eval, 73, 180
eval-jit-enabled, 73
eval-syntax, 180
evaluation handler, 73
evaluation order, 5
even?, 13
evt?, 68
'exact, 184
exception handlers, 61
exceptions, 47, 75
    primitive hierarchy, 48
'exec-file, 137
'execute, 82, 138
'exists, 83

exit, 76, 181
exit handler, 76
exit-handler, 76
exiting, 76
exn, 48, 49
exn:break, 48, 58, 64, 67, 113, 116, 141, 143
exn:fail, 46, 50, 63, 83, 93, 98, 99, 152, 183, 185, 186
exn:fail:contract, 12, 13, 15, 17, 20, 24, 25, 27, 31, 33, 34, 36, 44, 45, 49, 50, 56, 61–63, 65, 81, 84, 96, 98, 100, 102, 108, 109, 112–114, 116–118, 123, 132–136, 142, 143, 164, 184, 200, 203
exn:fail:contract:arity, 5, 11, 25, 50, 166
exn:fail:contract:continuation, 52, 54, 55
exn:fail:contract:variable, 80
exn:fail:filesystem, 96, 99, 135, 138, 139, 182, 190
exn:fail:filesystem:version, 182
exn:fail:network, 140–144
exn:fail:out-of-memory, 85
exn:fail:read, 20, 49, 118, 119, 121, 124
exn:fail:read:non-char, 122
exn:fail:syntax, 49, 51, 80, 148, 150, 163, 166
exn:fail:unsupported, 85, 139, 185
exn?, 48
expand, 167
expand-once, 167
expand-path, 135
expand-syntax, 167
expand-syntax-once, 167
expand-syntax-to-top-form, 168
expand-to-top-form, 167
expansion-time value, 164
exponential input, 13
expressions
    shared structure, 123

fields, 29
file, 42
file-exists?, 138
file-or-directory-modify-seconds, 138, 139, 183
file-or-directory-permissions, 138, 139
file-position, 98
file-size, 139
file-stream-buffer-mode, 98
file-stream-port?, 95
files, 98
    copying, 139
    deleting, 138
    loading, 180
    modification dates, 138
    moving, 138
    paths, *see* paths
    permissions, 138
    renaming, 138
    sizes, 139
    testing, 138
filesystem-root-list, 139
finalization, *see* will executors
find-executable-path, 137
find-library-collection-paths, 189
find-system-path, 136, 192, 193
fixnum, 13
floating-point-bytes->real, 15
flonum, 13
fluid-let, 9
flush-output, 98
force, 26
formals, 7
format, 117
fprintf, 117
fraction, 13
'framework, 187
free-identifier=?, 154
Friedman, Dan, 47

Gasbichler, Martin, 71
'gc, 186
generate-temporaries, 153
gensym, 23
get-output-bytes, 97
get-output-string, 97
getenv, 186
global port print handler, 71
global-port-print-handler, 71, 117, 125
glyphs, 2
graphs, 121, 123
    printing, 123
guard-evt, 67
guardians, *see* will executors

handle-evt, 67
handle-evt?, 68
hash tables, 26
    constants, 121
    printing, 72, 123
hash-table-copy, 27
hash-table-count, 27
hash-table-for-each, 27
hash-table-get, 27
hash-table-map, 27
hash-table-put!, 27
hash-table-remove!, 27
hash-table?, 27
Haynes, Chris, 47
header, 7
here strings, 119

Hieb, Rob, 145
HOME, 136
'home-dir, 136
HOMEDRIVE, 136
HOMEPATH, 136
Honu, 196

iconv, 22
iconv.dll, 22
identifier macro, 146, 163
identifier-binding, 154
identifier-binding-export-position, 155
identifier-template-binding, 155
identifier-transformer-binding, 154
identifier-transformer-binding-export-position, 155
identifier?, 153
immutable?, 24
'inferred-name, 51
infinity, 13
infix, 120
info.ss, 189
'init-dir, 137
'init-file, 137
'initial, 80
inode, 99
input ports
  pattern matching, 87
inspector?, 34
inspectors, 33, 76
integer->char, 15
integer->integer-bytes, 15
integer-bytes->integer, 15
integer-length, 14
integer-sqrt, 14
integer-sqrt/remainder, 14

keyword->string, 23
keyword?, 23
keywords, 23, 121
  printing, 122
kill-thread, 62

Latin-1, 3
let, 8
let*, 8
let*-values, 8
let-struct, 30
let-syntaxes, 160
let-values, 8
let/cc, 52
let/ec, 52
let/ec, 52
letrec, 8
letrec-syntaxes, 160
letrec-syntaxes+values, 160
letrec-values, 8, 12
'lexical, 154
lib, 42
libiconv.dll, 22
libraries, 189
'line, 98
line numbers, 115
'linefeed, 112
'link, 186
link-exists?, 138
links
  creating, 139
  testing, 138
list, 24
list*, 24
list*-immutable, 24
list->bytes, 19
list-immutable, 24
list-ref, 24
list-tail, 24
'll, 16
'lm, 16
'lo, 16
load, 74, 121, 180
load extension handler, 74
load handler, 74
load-extension, 74, 182
load-relative, 74, 180
load-relative-extension, 74, 182
load/cd, 74, 181
load/use-compiled, 74, 180
load/use-compiled handler, 74
load/used-compiled, 74
local-expand, 165
local-expand/capture-lifts, 166
local-transformer-expand, 166
local-transformer-expand/capture-lifts, 166
locale-string-encoding, 23
locales, 76
logical operators, *see* bitwise operators
LOGNAME, 136
LPT1, 201
LPT2, 201
LPT3, 201
LPT4, 201
LPT5, 201
LPT6, 201
LPT7, 201
LPT8, 201
LPT9, 201

'lt, 16
'lu, 16

'machine, 186
'macosx, 187
macros, *see* syntax
make-bytes, 19
make-channel, 64
make-continuation-prompt-tag, 55
make-custodian, 84
make-custodian-box, 85
make-date, 183
make-directory, 139
make-ephemeron, 177
make-file-or-directory-link, 139
make-hash-table, 26
make-immutable-hash-table, 27
make-input-port, 99
make-inspector, 34
make-known-char-range-list, 17
make-namespace, 79
make-output-port, 107
make-parameter, 77
make-pipe, 97
make-pseudo-random-generator, 14
make-readtable, 125
make-rename-transformer, 163
make-security-guard, 82
make-semaphore, 63
make-set!-transformer, 163
make-special-comment, 130
make-string, 17
make-struct-field-accessor, 31
make-struct-field-mutator, 32
make-struct-type, 31
make-struct-type-property, 32
make-syntax-introducer, 166
make-thread-cell, 68
make-thread-group, 85
make-weak-box, 177
make-will-executor, 178
'mc, 16
'me, 16
member, 24
memq, 24
memv, 24
'method-arity-error, 25
'mn, 16
module, 38
module name resolver, 42
module path index, 44
module registry, 78
module->namespace, 81
module-compiled-exports, 176
module-compiled-imports, 176
module-compiled-name, 176
'module-direct-for-syntax-requires, 175
'module-direct-for-template-requires, 175
'module-direct-requires, 175
module-identifier=?, 154
'module-indirect-provides, 175
'module-kernel-reprovide-hint, 175
module-path-index-join, 44
module-path-index-resolve, 44
module-path-index-split, 44
module-path-index?, 44
module-provide-protected?, 81
'module-self-path-index, 176
'module-syntax-provides, 175
module-template-identifier=?, 154
module-transformer-identifier=?, 154
'module-variable-provides, 175
modules, 38
    body, 39
    built-in, 45
    compiling, 43
    dynamic imports, 44
    execution, 38
    expansion, 38
    exports, 39
    for-syntax imports, 156
    imports, 39
    in files, 42
    libraries, 42
    macros, 41, 156
    paths, 42
    pre-defined, 45
    predefined, 45
    re-declaring, 45
    re-defining, 45
    redeclaring, 45
    redefining, 45
    syntax, 41
msvcrt.dll, 22
multiple return values, 5
MzLib library, 190
MzScheme
    stand-alone, 1, 191
mzscheme, 38
MzScheme3m, 1
MzSchemeCGC, 1
mzschemerc.ss, 137

nack-guard-evt, 67
namespace-attach-module, 43, 81
namespace-mapped-symbols, 80
namespace-module-registry, 81
namespace-require, 80

`namespace-require/copy`, 80
`namespace-require/expansion-time`, 80
`namespace-set-variable-value!`, 80
`namespace-symbol->identifier`, 80
`namespace-syntax-introduce`, 81
`namespace-transformer-require`, 80
`namespace-undefine-variable!`, 80
`namespace-unprotect-module`, 81
`namespace-variable-value`, 80
`namespace?`, 80
namespaces, 78
  initial, 79
  initial environment, 79
  initial transformer environment, 155
`'nd`, 16
networking, 139
`'nl`, 16
`'no`, 16
`'non-terminating-macro`, 126
`'none`, 98
`normal-case-path`, 135
not-a-number, 13
NUL, 201
`null`, 24
`null-environment`, 78
`number->string`, 13
numbers, 13, 119
  big-endian, 15
  converting, 15
  floating-point, 15
  little-endian, 15
  machine representations, 15

`object-name`, 51
`odd?`, 13
`only`, 39
`'opaque`, 173
`open-input-bytes`, 97
`open-input-file`, 96
`open-input-output-file`, 96
`open-input-string`, 97
`open-output-bytes`, 97
`open-output-file`, 96
`open-output-string`, 97
`or`, 6
`'orig-dir`, 137
`'origin`, 169
`ormap`, 12
`'os`, 186

packages, 38
parameter procedure, 69
`parameter-procedure=?`, 77
`parameter?`, 77
parameterization, 69
`parameterization?`, 77
`parameterize`, 69
`parameterize*`, 70
`parameterize-break`, 58
parameters, 69
  built-in, 71
`'paren-shape`, 169
parsing, 71
PATH, 138
`path->bytes`, 133
`path->complete-path`, 135
`path->directory-path`, 135
`path->string`, 133
`path-convention-type`, 133
`path-element->bytes`, 133
`path-element->string`, 133
`path-for-some-system?`, 132
`path-list-string->path-list`, 137
`path-replace-suffix`, 136
`path-string?`, 132
`path?`, 132
pathnames, *see* paths
paths, 132
  expansion, 132
  printing, 123
pattern matching, 87
`'pc`, 16
`'pd`, 16
`'pe`, 16
`peek-byte`, 114
`peek-byte-or-special`, 114
`peek-bytes`, 113
`peek-bytes!`, 113
`peek-bytes-avail!`, 113
`peek-bytes-avail!*`, 114
`peek-bytes-avail!/enable-break`, 114
`peek-char`, 114
`peek-char-or-special`, 114
`peek-string`, 113
`peek-string!`, 113
`'pf`, 16
`'pi`, 16
`pipe-content-length`, 97
`planet`, 42
platform, 136, 186
PLTCOLLECTS, 190
PLTNOMZJIT, 74
`'po`, 16
poll, *see* `sync`
`poll-guard-evt`, 67
port display handler, 124
port positions, 115

port print handler, 124
port read handler, 124
port write handler, 124
port-closed?, 95
port-commit-peeked, 114
port-count-lines!, 115
port-count-lines-enabled, 71
port-display-handler, 124, 125
port-file-identity, 99
port-next-location, 115
port-print-handler, 125
port-progress-evt, 114
port-provides-progress-evts?, 114
port-read-handler, 124
port-write-handler, 125
port-writes-atomic?, 117
port-writes-special?, 117
port?, 95
ports, 61, 71, 95
  byte string, 97
  custom, 99
  file, 98
  flushing, 98
  string, 97
'pref-dir, 136
'pref-file, 136
prefix, 39
prefix-all-defined, 40
prefix-all-defined-except, 40
pregexp, 89
pregexp?, 89
primitive procedure, 26
primitive-closure?, 26
primitive-result-arity, 26
primitive?, 26
print, 117, 124
print handler, 73
print-box, 72, 122
print-graph, 72, 124
print-hash-table, 72
print-honu, 72
print-struct, 72, 122
print-unreadable, 72
print-vector-length, 72, 122
printf, 117
printing sharing, 72
PRN, 201
procedure-arity, 25
procedure-arity-includes?, 25
procedure-closure-contents-eq?, 26
procedure-struct-type?, 36
procedure?, 11, 35
processes, 184
program, 137, 192
promise?, 26
promises, 26
prompt read handler, 73
prop:exn:srclocs, 49
prop:procedure, 31
protect, 40
'protected, 170
provide, 40
'ps, 16
pseudo-random-generator->vector, 14
pseudo-random-generator?, 15
putenv, 186

quasiquote, 6
quasisyntax, 150
quasisyntax/loc, 151
quote-syntax, 147
quotient/remainder, 13

raise, 47
raise-arity-error, 50
raise-mismatch-error, 50
raise-syntax-error, 51
raise-type-error, 50
raise-user-error, 50
random, 14, 76
random numbers, 76
random-seed, 14, 76
'read, 82, 138
read, 49
read extensions, 118
read-accept-bar-quote, 72, 120, 122
read-accept-box, 72, 118
read-accept-compiled, 72, 182
read-accept-dot, 72, 120
read-accept-graph, 72, 124
read-accept-quasiquote, 72, 120
read-accept-reader, 72
read-byte, 114
read-byte-or-special, 114
read-bytes, 112
read-bytes!, 112
read-bytes-avail!, 113
read-bytes-avail!*, 113
read-bytes-avail!/enable-break, 113
read-bytes-line, 112
read-case-sensitive, 71, 121, 122
read-char-or-special, 114
read-curly-brace-as-paren, 72, 118, 121
read-decimal-as-inexact, 72, 120
read-eval-print loop, 73
  read-eval-print loop
    customized, 181

read-eval-print loop, 181
read-eval-print-loop, 181, 193
read-honu, 196
read-honu-syntax, 170, 196
read-line, 112
read-square-bracket-as-paren, 71, 118, 121
read-string, 112
read-string!, 112
read-syntax, 147, 170
read-syntax/recursive, 130
read/recursive, 130
reader macros, 126
readtable-mapping, 126
readtables, 125
real->floating-point-bytes, 15
regexp, 89
regexp-match, 90, 91
regexp-match-peek, 92
regexp-match-peek-immediate, 92
regexp-match-peek-positions, 92
regexp-match-peek-positions-immediate, 92
regexp-match-positions, 92
regexp-match?, 92
regexp-replace, 92, 93
regexp-replace*, 93
regexp?, 89
regexps, *see* regular expressions
regular expressions, 87
  constants, 121
  printing, 123
'relative, 136
relative-path?, 134
rename, 40
rename-file-or-directory, 138, 139
rename-transformer-target, 164
rename-transformer?, 163
repl, *see* read-eval-print loop
'replace, 96
require, 39, 74
require-for-syntax, 156, 157
require-for-template, 157
resolve-path, 135
'return, 112
'return-linefeed, 112
reverse!, 24
'run-file, 137
run-time hierarchy, 158
'running, 185

'same, 134, 136
'sc, 16
scheme-report-environment, 78
scripts, 194
Scsh, 71
seconds->date, 183
security guards, 76, 82
security-guard?, 83
select, *see* sync
semaphore-peek-evt, 64
semaphore-post, 63
semaphore-try-wait?, 64
semaphore-wait, 64
semaphore-wait/enable-break, 64
semaphore?, 63
semaphores, 63
'server, 83
set!, 9
set!, 192
set!-transformer-procedure, 163
set!-transformer?, 163
set!-values, 9
set-arity-at-least-value!, 25
set-box!, 25
set-date-day!, 183
set-date-dst?!, 183
set-date-hour!, 183
set-date-minute!, 183
set-date-month!, 183
set-date-second!, 183
set-date-time-zone-offset!, 183
set-date-week-day!, 183
set-date-year!, 183
set-date-year-day!, 183
'shared, 187
shell scripts, 194
shell-execute, 185
ShellExecute, 185
simplify-path, 135
'sk, 16
sleep, 63
'sm, 16
'so, 16
'so-suffix, 186
sockets, 139
special-comment-value, 130
special-comment?, 130
Sperber, Michael, 71
split-path, 135
square brackets, 71
srcloc, 115
stack trace, 55
  length, 75
'static, 187
string converters, 21
string->bytes/latin-1, 20
string->bytes/locale, 20
string->bytes/utf-8, 20

string->immutable-string, 17
string->keyword, 23
string->number, 13
string->path, 132
string->path-element, 133
string->symbol, 23
string->uninterned-symbol, 23
string-ci<=?, 17
string-ci<?, 17
string-ci=?, 17
string-ci>=?, 17
string-ci>?, 17
string-copy!, 17
string-downcase, 18
string-foldcase, 18
string-locale-ci<?, 18
string-locale-ci=?, 18
string-locale-ci>?, 18
string-locale-downcase, 18
string-locale-upcase, 18
string-locale<?, 18
string-locale=?, 18
string-locale>?, 18
string-normalize-nfc, 19
string-normalize-nfd, 19
string-normalize-nfkc, 19
string-normalize-nfkd, 19
string-titlecase, 18
string-upcase, 18
string-utf-8-length, 20
string<=?, 17
string<?, 17
string=?, 17
string>=?, 17
string>?, 17
strings, 119
    as ports, 97
    immutable, 17
    pattern matching, 87
    printing, 123
    reading to and writing from, 97
struct, 40
struct->vector, 37
struct-accessor-procedure?, 37
struct-constructor-procedure?, 37
struct-info, 34
struct-mutator-procedure?, 37
struct-predicate-procedure?, 37
struct-type-info, 34
struct-type-make-constructor, 34
struct-type-make-predicate, 34
struct-type-property?, 33
struct-type?, 37
struct:date, 183
struct?, 37
structs
    printing, 72
structure subtypes, 30
structure type descriptors, 29
structure type properties, 32
structure types, 29
    predicates, 37
structures, 29
    equality, 37
    printing, 122
sub1, 13
subbytes, 19
subprocess, 184
subprocess-kill, 185
subprocess-pid, 185
subprocess-status, 185
subprocess-wait, 185
subprocess?, 185
subprocesses, 184
substring, 17
'SW_HIDE, 186
'sw_hide, 186
'SW_MAXIMIZE, 186
'sw_maximize, 186
'SW_MINIMIZE, 186
'sw_minimize, 186
'SW_RESTORE, 186
'sw_restore, 186
'SW_SHOW, 186
'sw_show, 186
'SW_SHOWDEFAULT, 186
'sw_showdefault, 186
'SW_SHOWMAXIMIZED, 186
'sw_showmaximized, 186
'SW_SHOWMINIMIZED, 186
'sw_showminimized, 186
'SW_SHOWMINNOACTIVE, 186
'sw_showminnoactive, 186
'SW_SHOWNA, 186
'sw_showna, 186
'SW_SHOWNOACTIVATE, 186
'sw_shownoactivate, 186
'SW_SHOWNORMAL, 186
'sw_shownormal, 186
symbols, 120
    case sensitivity, 120
    generating, 23
    printing, 122
    unique, 23
sync, 65, 76
sync/enable-break, 67, 76

sync/timeout, 65
sync/timeout/enable-break, 67
synchronous channels, 64
syntax, 145
  expanding, 167
  macro calls, 163
  modules, 156
  partial expansion, 165
syntax, 148
syntax objects, 147
  comparisons, 154
  identifier, 153
  operations, 151
  pattern-matching, 148
  properties, 169, 182
  source location, 151, 182
  source module, 151
syntax pair, 152
syntax->list, 152
syntax-case, 148
syntax-case*, 149
syntax-column, 151
syntax-e, 151
syntax-graph?, 153
syntax-id-rules, 146
syntax-line, 151
syntax-local-bind-syntaxes, 166
syntax-local-certifier, 165
syntax-local-context, 164
syntax-local-expand-expression, 166
syntax-local-get-shadower, 165
syntax-local-introduce, 166
syntax-local-lift-expression, 164
syntax-local-lift-module-end-declaration, 164
syntax-local-make-definition-context, 166
syntax-local-name, 164
syntax-local-value, 164
syntax-object->datum, 152
syntax-original?, 151
syntax-position, 151
syntax-property, 169
syntax-property-symbol-keys, 169
syntax-recertify, 174
syntax-rules, 145
syntax-source, 151
syntax-source-module, 151
syntax-span, 151
syntax-track-origin, 169
syntax-transforming?, 165
syntax/loc, 151
syntax?, 151
'sys-dir, 137
system-big-endian?, 15
system-idle-evt, 68
system-language+country, 187
system-library-subpath, 187
system-path-convention-type, 134
system-type, 186

tcp-abandon-port, 141
tcp-accept, 141
tcp-accept-evt, 141
tcp-accept-ready?, 141
tcp-accept/enable-break, 141
tcp-addresses, 142
tcp-close, 141
tcp-connect, 140
tcp-connect/enable-break, 141
tcp-listen, 140
tcp-listener?, 141
tcp-port?, 142
TCP/IP, 140
'temp-dir, 137
terminal-port?, 95
'terminating-macro, 126
'text, 96
thread
  groups, *see* thread groups
thread, 61
thread cells, 68
thread descriptor, 61
thread groups, 76, 85
thread-cell-ref, 68
thread-cell-set!, 68
thread-cell?, 68
thread-dead-evt, 62
thread-dead?, 63
thread-group?, 85
thread-resume, 61
thread-resume-evt, 62
thread-running?, 63
thread-suspend, 61
thread-suspend-evt, 62
thread-wait, 62
thread/suspend-to-kill, 62
thread?, 63
threads, 61
  breaking, 58, 63
  communication, 63, 64
  killing, 62
  nesting, 63
  resuming, 61
  run state, 63
  stack size, 77
  suspending, 61
  synchronization, 63, 64

time, 183
    machine, 183
time, 184
time-apply, 184
TMPDIR, 137
top-level environment, *see* namespaces
transformer environments, 145
    in modules, 156
transformers, 145
    application, 163
'transparent, 173
'transparent-binding, 173
'truncate, 96
'truncate/replace, 96

UDP, 142
udp-bind!, 142
udp-bound?, 144
udp-close, 143
udp-connect!, 142
udp-connected?, 144
udp-open-socket, 142
udp-receive!, 143
udp-receive!*, 143
udp-receive!-evt, 144
udp-receive!/enable-break, 143
udp-receive-ready-evt, 144
udp-send, 143
udp-send*, 143
udp-send-evt, 144
udp-send-ready-evt, 144
udp-send-to, 143
udp-send-to*, 143
udp-send-to-evt, 144
udp-send-to/enable-break, 143
udp-send/enable-break, 143
udp?, 144
unbox, 25
uncaught-exception-handler, 75
uncertified context, 170
undefined values, 12
Unicode, 2
uninterned symbol, 23
'unix, 134, 187
unless, 6
unquote, 7
unquote-splicing, 7
unreadable
    printing, 72, 123
unsyntax, 150
unsyntax-splicing, 150
'up, 134, 136
'update, 96
use-compiled-file-paths, 74
use-user-specific-search-paths, 74
USER, 136
USERPROFILE, 136
UTF-8, 3
UTF-8-permissive, 21

vector->immutable-vector, 24
vector->pseudo-random-generator, 14
vector-immutable, 24
vector-set-performance-stats!, 187
vectors, 118
    printing, 72, 122
version, 187
vertical bar, 72
void, 12
void?, 12

'weak, 27
weak boxes, 177
weak references, 177
weak-box-value, 177
weak-box?, 177
when, 6
will executors, 178
will-execute, 178
will-executor?, 178
will-register, 178
will-try-execute, 178
'windows, 134, 187
Windows pathname syntax, 200
with-continuation-mark, 56
with-handlers, 47
with-handlers*, 48
with-input-from-file, 96
with-output-to-file, 96
with-syntax, 149
wrap-evt, 67
'write, 82, 138
write, 124
write extensions, 122
write-byte, 116
write-bytes, 116
write-bytes-avail, 116
write-bytes-avail*, 116
write-bytes-avail-evt, 116
write-bytes-avail/enable-break, 116
write-special, 116
write-special-avail*, 116
write-special-evt, 116
write-string, 116

'zl, 16
'zp, 16
'zs, 16