

# PLT MzScheme: Language Manual

---

Matthew Flatt ([mflatt@cs.utah.edu](mailto:mflatt@cs.utah.edu))

Version 200  
June 2002

## Copyright notice

Copyright ©1995-2002 Matthew Flatt

Permission to make digital/hard copies and/or distribute this documentation for any purpose is hereby granted without fee, provided that the above copyright notice, author, and this permission notice appear in all copies of this documentation.

libscheme: Copyright ©1994 Brent Benson. All rights reserved.

Conservative garbage collector: Copyright ©1988, 1989 Hans-J. Boehm, Alan J. Demers. Copyright ©1991-1996 by Xerox Corporation. Copyright ©1996-1999 by Silicon Graphics. Copyright ©1999-2001 by Hewlett Packard Company. All rights reserved.

Collector C++ extension by Jesse Hull and John Ellis: Copyright ©1994 by Xerox Corporation. All rights reserved.

GNU MP Library: Copyright ©1992, 1993, 1994, 1996 by Free Software Foundation, Inc.

## 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](mailto: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  $\LaTeX$ ,  $\text{SI}\LaTeX$ , 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 June 21, 2002.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	MrEd, DrScheme, and <b>mzc</b>	1
1.2	Notation	2
<b>2</b>	<b>Basic Syntax Extensions</b>	<b>3</b>
2.1	Evaluation Order	3
2.2	Multiple Return Values	3
2.3	Cond and Case	4
2.4	When and Unless	4
2.5	And and Or	4
2.6	Sequences	4
2.7	Quote and Quasiquote	4
2.8	Binding Forms	5
2.8.1	Global Variables	5
2.8.2	Local Variables	5
2.8.3	Assignments	6
2.8.4	Fluid-Let	6
2.8.5	Syntax Expansion and Internal Definitions	6
2.9	Case-Lambda	7
<b>3</b>	<b>Basic Data Extensions</b>	<b>9</b>
3.1	Void and Undefined	9
3.2	Booleans	9
3.3	Numbers	10
3.4	Characters	12

---

3.5	Strings	13
3.6	Symbols	13
3.7	Vectors	14
3.8	Lists	14
3.9	Boxes	15
3.10	Procedures	15
3.10.1	Arity	15
3.10.2	Primitives	16
3.10.3	Procedure Names	16
3.11	Promises	16
3.12	Hash Tables	16
<b>4</b>	<b>Structures</b>	<b>18</b>
4.1	Defining Structure Types	18
4.2	Creating Subtypes	19
4.3	Structure Types with Automatic Fields and Properties	20
4.4	Structure Type Properties	21
4.5	Structure Inspectors	21
4.6	Structure Utilities	22
<b>5</b>	<b>Modules</b>	<b>24</b>
5.1	Module Expansion and Execution	24
5.2	Module Bodies	25
5.3	Modules and Macros	27
5.4	Module Paths	27
5.4.1	Module Name Resolver	28
5.4.2	Module Names and Compilation	29
5.5	Dynamic Module Access	29
5.6	Re-declaring Modules	29
5.7	Built-in Modules	29

---

5.8	Modules and Load Handlers . . . . .	30
<b>6</b>	<b>Exceptions and Control Flow</b>	<b>31</b>
6.1	Exceptions . . . . .	31
6.1.1	Primitive Exceptions . . . . .	32
6.2	Errors . . . . .	33
6.2.1	Application Type Errors . . . . .	34
6.2.2	Application Mismatch Errors . . . . .	34
6.2.3	Syntax Errors . . . . .	34
6.2.4	Inferred Value Names . . . . .	35
6.3	Continuations . . . . .	35
6.4	Dynamic Wind . . . . .	36
6.5	Continuation Marks . . . . .	37
6.6	Breaks . . . . .	39
6.7	Error Escape Handler . . . . .	40
<b>7</b>	<b>Threads</b>	<b>41</b>
7.1	Thread Utilities . . . . .	41
7.2	Semaphores . . . . .	42
7.3	Synchronizing Multiple Objects with Timeout . . . . .	43
7.4	Parameters . . . . .	43
7.4.1	Built-in Parameters . . . . .	44
7.4.2	Parameter Utilities . . . . .	48
<b>8</b>	<b>Namespaces</b>	<b>50</b>
8.1	Identifier Resolution in Namespaces . . . . .	50
8.2	Initial Namespace . . . . .	51
8.3	Namespace Utilities . . . . .	51
<b>9</b>	<b>Security</b>	<b>53</b>
9.1	Security Guards . . . . .	53

---

9.2 Custodians . . . . .	54
<b>10 Regular Expressions</b>	<b>56</b>
<b>11 Input and Output</b>	<b>60</b>
11.1 Ports . . . . .	60
11.1.1 Current Ports . . . . .	60
11.1.2 Opening File Ports . . . . .	60
11.1.3 Pipes . . . . .	61
11.1.4 String Ports . . . . .	61
11.1.5 File-Stream Ports . . . . .	61
11.1.6 Custom Ports . . . . .	62
11.2 Reading and Writing . . . . .	65
11.2.1 Reading . . . . .	65
11.2.2 Writing . . . . .	66
11.2.3 Counting Positions, Lines, and Columns . . . . .	68
11.2.4 Customizing Read . . . . .	69
11.2.5 Customizing Display, Write, and Print . . . . .	69
11.3 Filesystem Utilities . . . . .	69
11.3.1 Pathnames . . . . .	70
11.3.2 Files . . . . .	73
11.3.3 Directories . . . . .	74
11.4 Networking . . . . .	75
<b>12 Syntax and Macros</b>	<b>77</b>
12.1 <code>syntax-rules</code> Extensions . . . . .	77
12.2 Syntax Objects . . . . .	78
12.2.1 Syntax Patterns . . . . .	79
12.2.2 Syntax Object Content . . . . .	82
12.3 Syntax and Lexical Scope . . . . .	84
12.3.1 Syntax Object Comparisons . . . . .	84

---

12.3.2	Syntax Object Bindings	85
12.3.3	Transformer Environments	85
12.3.4	Module Environments	86
12.4	Binding Multiple and Fluid Syntax Identifiers	88
12.5	Special Syntax Identifiers	89
12.6	Macro Expansion	90
12.6.1	Expanding Expressions to Primitive Syntax	93
12.6.2	Syntax Object Properties	94
12.6.3	Information on Structure Types	95
12.6.4	Information on Expanded and Compiled Modules	96
<b>13</b>	<b>Memory Management</b>	<b>98</b>
13.1	Weak Boxes	98
13.2	Will Executors	98
13.3	Garbage Collection	99
<b>14</b>	<b>Support Facilities</b>	<b>100</b>
14.1	Eval and Load	100
14.2	Exiting	101
14.3	Input Parsing	101
14.4	Output Printing	104
14.5	Data Sharing in Input and Output	105
14.6	Compilation	105
14.7	Dynamic Extensions	106
14.8	Saving and Restoring Program Images	106
<b>15</b>	<b>System Utilities</b>	<b>108</b>
15.1	Time	108
15.1.1	Real Time and Date	108
15.1.2	Machine Time	108
15.1.3	Timing Execution	109

---

15.2 Operating System Processes . . . . .	109
15.3 Windows Actions . . . . .	110
15.4 Operating System Environment Variables . . . . .	111
15.5 Runtime Information . . . . .	112
<b>16 Library Collections and MzLib</b>	<b>113</b>
<b>17 Running MzScheme</b>	<b>115</b>
17.1 Flag Conventions . . . . .	116
17.2 Executable Name . . . . .	116
17.3 Initialization . . . . .	117
<b>18 Writing and Running Scripts</b>	<b>118</b>
<b>Index</b>	<b>120</b>

# 1. Introduction

---

The core of the Scheme programming language is described in *Revised<sup>5</sup> 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 *R<sup>5</sup>RS*-compliant. Certain parameters in MzScheme can change features affecting *R<sup>5</sup>RS*-compliance; for example, case-sensitivity can be enabled (see §7.4.1.3).

MzScheme provides several notable extensions to *R<sup>5</sup>RS* 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 3 of *PLT MzLib: Libraries Manual*).
- A unit system for defining and linking program components (see Chapter 32 of *PLT MzLib: Libraries Manual*).

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

MzScheme3m is an experimental version of MzScheme that uses more precise memory-management techniques. For long-running applications, especially, MzScheme3m can provide superior memory performance. See the compilation information in the MzScheme source distribution for more details.

## 1.2 Notation

Throughout this manual, the syntax for new forms is described using a pattern notation with ellipses. Plain, centered ellipses ( $\dots$ ) indicate *zero* or more repetitions of the preceding S-expression pattern. Ellipses with a “1” subscript ( $\dots^1$ ) indicate *one* or more repetitions of the preceding S-expression pattern.

For example:

```
(let-values (((variable  $\dots$ ) expr)  $\dots$ )
  body-expr
   $\dots^1$ )
```

The first set of ellipses indicate that any number of *variables*, possibly none, can be provided with a single *expr*. The second set of ellipses indicate that any number of  $((\textit{variable} \dots) \textit{expr})$  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 ( $\dots$ ); these ellipses indicate that an unimportant part of the example expression has been omitted.

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).

## 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<sup>5</sup>RS* **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:application:arity** exception is raised (at run time).

Examples:

```
(- (values 1)) ; => -1
(- (values 1 2)) ; => exn:application:arity, returned 2 values to single-value context
(- (values)) ; => exn:application:arity, 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* ...<sup>1</sup>) evaluates the *expr* body expressions only when *test-expr* returns a true value.
- (**unless** *test-expr* *expr* ...<sup>1</sup>) 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

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
```

## 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 §14.3 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 Global Variables

Top-level variables are bound with the standard Scheme **define** form. Multiple values are bound to multiple variables at once with **define-values**:

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

The number of values returned by *expr* must match the number of *variables* provided, and the *variables* must be distinct.

Examples:

```
(define x 1)
x ; => 1
(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)) ; => exn:application:arity, 2 values for 1-value context
(define-values (x y) 7) ; => exn:application:arity, 1 value for 2-value context
(define-values () 7) ; => exn:application:arity, 1 value for 0-value context
```

### 2.8.2 Local Variables

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.

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 *variables* 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 *variables* provided.

The *variables*, 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 *variables* will have already completed, but assignments for the remaining *variables* 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-exprs* are evaluated, the bindings for the *variables* are **set!** to the values of the corresponding *exprs*. Once the *body-exprs* 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**, **letrec-syntaxes+values**, and **fluid-let-syntax** 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 variables bound by the internal definitions become the binding variables 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 exposes **begin**, **define-values**, and **define-syntaxes** forms without expanding the definition body. Therefore, an internal definition can shadow a syntactic form in the value part of other embedded definitions. However, an embedded definition cannot alter the decision of whether another expression is also an embedded definition in the same **letrec-syntaxes+values** transformation, because the definitions are collected through partial expansion before any new scope is created.

## 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  $(\mathbf{lambda} (a . x) \textit{expr} \dots^1)$ .

A **case-lambda** expression has the form:

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

*formals* is one of  
*variable*  
 (*variable* ...)  
 (*variable* ... . *variable*)

Each (*formals* *expr* ...<sup>1</sup>) clause of a **case-lambda** expression is analogous to a **lambda** expression of the form  $(\mathbf{lambda} \textit{formals} \textit{expr} \dots^1)$ . The scope of the *variables* in each clause’s *formals* includes only the same clause’s *exprs*. 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 *exprs* 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.<sup>1</sup>

Examples:

```
(define f
  (case-lambda
    [(x) x]
    [(x y) (+ x y)]
    [(a . any) a]))
(f 1) ; => 1
(f 1 2) ; => 3
(f 4 5 6 7) ; => 4
(f) ; => raises exn:application: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.

---

<sup>1</sup>It is possible that a clause in a **case-lambda** expression can never be evaluated because a preceding clause always matches the arguments.

## 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<sup>5</sup>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?`, `=` and have the same exactness, or both `+nan.0`.

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 *lists* 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. If the *lists* are empty, then `#t` is returned.
- `(ormap proc list ...1)` applies *proc* to elements of the *lists* 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`. If the *lists* 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:application:type
(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<sup>1</sup> 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<sup>5</sup>RS* in two ways:

- All input radices (`#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` always stands for a hexadecimal digit, not an exponent marker.
- 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`.

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.<sup>2</sup> 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:application:mismatch` 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 exact integers in their (semi-infinite) two’s complement representation:

- `(bitwise-ior n ...1)` returns the bitwise “inclusive or” of the *ns*.
- `(bitwise-and n ...1)` returns the bitwise “and” of the *ns*.
- `(bitwise-xor n ...1)` returns the bitwise “exclusive or” of the *ns*.

<sup>1</sup>30 bits for a 32-bit architecture, 62 bits for a 64-bit architecture.

<sup>2</sup>This definition of `eqv?` technically contradicts *R<sup>5</sup>RS*, but *R<sup>5</sup>RS* does not address strange “numbers” like `+nan.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.

The `random` procedure generates pseudo-random integers:

- (`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.<sup>3</sup>
- (`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.
- (`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 §7.4.1.10.
- (`make-pseudo-random-generator`) returns a new pseudo-random number generator. The new generator is seeded with a number derived from (`current-milliseconds`).
- (`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-byte-string->integer`  $string$   $signed?$  [ $big-endian?$ ]) converts the machine-format number encoded in  $string$  to an exact integer. The  $string$  must contain either 2, 4, or 8 characters. If  $signed?$  is true, then the string is 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-byte-string`  $n$   $size-n$   $signed?$  [ $big-endian?$   $to-string$ ]) converts the exact integer  $n$  to a machine-format number encoded in a 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 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  $to-string$  is provided, it must be a mutable string of length  $size-n$ ; in that case, the encoding of  $n$  is written into  $to-string$ , and  $to-string$  is returned as the result. If  $to-string$  is not provided, the result is a newly allocated string.  
If  $n$  cannot be encoded in a string of the requested size and format, the `exn:misc:application` exception is raised. If  $to-string$  is provided and it is not of length  $size-n$ , the `exn:misc:application` exception is raised.
- (`floating-point-byte-string->real`  $string$  [ $big-endian?$ ]) converts the IEEE floating-point number encoded in  $string$  to an inexact real number. The  $string$  must contain either 4 or 8 characters. If  $big-endian?$  is true, then the first character’s ASCII value provides the most significant eight bits of the IEEE representation, 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?`.

---

<sup>3</sup>The random number generator uses a relatively standard Unix `random()` implementation in its degree-seven polynomial mode.

- `(real->floating-point-byte-string x size-n [big-endian? to-string])` converts the real number  $x$  to its IEEE representation in a 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 character of the resulting 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  $to-string$  is provided, it must be a mutable string of length  $size-n$ ; in that case, the encoding of  $n$  is written into  $to-string$ , and  $to-string$  is returned as the result. If  $to-string$  is not provided, the result is a newly allocated string.

If  $to-string$  is provided and it is not of length  $size-n$ , the `exn:misc:application` 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 character values range over the characters for “extended ASCII” values 0 to 255 (where the ASCII extensions are platform-specific). The procedure `char->integer` returns the extended ASCII value of a character and `integer->char` takes an extended ASCII value and returns the corresponding character. If `integer->char` is given an integer that is not in 0 to 255 inclusive, the `exn:application:type` exception is raised.

The procedures `char->latin-1-integer` and `latin-1-integer->char` support conversions between characters in the platform-specific character set and platform-independent Latin-1 (ISO 8859-1) values:

- `(char->latin-1-integer char)` returns the integer in 0 to 255 inclusive corresponding to the Latin-1 value for  $char$ , or `#f` if  $char$  (in the platform-specific character set) has no corresponding character in Latin-1.
- `(latin-1-integer->char k)` returns the character corresponding to the Latin-1 mapping of  $k$ , or `#f` if the platform-specific character set does not support the corresponding Latin-1 character. If  $k$  is not in 0 to 255 inclusive, the `exn:application:type` exception is raised.

For Unix and Mac OS X, `char->latin-1-integer` and `latin-1-integer->char` are the same as `char->integer` and `integer->char`. For Windows, the platform-specific set and Latin-1 match except for the range `#x80` to `#x9F` (which are unprintable control characters in Latin-1). For Mac OS Classic, the mapping between Latin-1 and the platform-specific character set (“MacRoman”) is complex, and several printable characters in each set have no corresponding character in the other set.

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 procedures `char-whitespace?`, `char-alphabetic?`, `char-numeric?`, `char-upper-case?`, and `char-upper-case?`, `char-upcase`, and `char-downcase` are fully portable; their results do not depend on the platform or locales.

In addition to the standard character procedures, MzScheme provides the following locale-sensitive procedures (see §7.4.1.11):

- `(char-locale<? char1 char2 ...1)`
- `(char-locale>? char1 char2 ...1)`
- `(char-locale-ci=? char1 char2 ...1)`

- `(char-locale-ci<? char1 char2 ...1)`
- `(char-locale-ci>? char1 char2 ...1)`
- `(char-locale-whitespace? char)`
- `(char-locale-alphabetic? char)`
- `(char-locale-numeric? char)`
- `(char-locale-upper-case? char)`
- `(char-locale-lower-case? char)`
- `(char-locale-upcase char)`
- `(char-locale-downcase char)`

For example, since ASCII character 112 is a lowercase “p” and Latin-1 character 246 is a lowercase “ö” (with an umlaut), `(char-locale<? (integer->char 112) (integer->char 246))` tends to produce `#f`, though it always produces `#t` if the current locale is disabled.

### 3.5 Strings

A string can be mutable or immutable. When an immutable string is provided to a procedure like `string-set!`, the `exn:application:type` exception is raised.

String constants generated by `read` are immutable. `(string->immutable-string string)` returns an immutable string with the same content as `string`, returning `string` if it is already an immutable string. (See also `immutable?` in §3.8.)

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 using the standard functions are fully portable; the results do not depend on the platform or locales.

In addition to the string character procedures, MzScheme provides the following locale-sensitive procedures (see §7.4.1.11):

- `(string-locale<? string1 string2 ...1)`
- `(string-locale>? string1 string2 ...1)`
- `(string-locale-ci=? string1 string2 ...1)`
- `(string-locale-ci<? string1 string2 ...1)`
- `(string-locale-ci>? string1 string2 ...1)`

### 3.6 Symbols

For information about symbol parsing and printing, see §14.3 and §14.4, 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 a `eq?`, `eqv?`, or `equal?` test, but a symbol placed into a weak box (see §13.1) or used as the key in a weak hash table (see §3.12) may disappear.

### 3.7 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 §3.8.)

### 3.8 Lists

A cons cell can be mutable or immutable. When an immutable cons cell is provided to a procedure like `set-cdr!`, the `exn:application:type` 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!`.

`(append! list ...1)` destructively appends the *lists*.

`(list* v ...1)` is similar to `(list v ...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 ...1)` is like `(list v ...1)`, but using immutable pairs.

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

`(immutable? v)` returns `#t` if *v* is an immutable cons cell, string, vector, or box, `#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:application:mismatch` 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:application:mismatch` 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:application:mismatch` exception is raised.

## 3.9 Boxes

MzScheme provides *boxes*, records with a single mutable field:

- `(box v)` returns a new box that contains *v*.
- `(unbox box)` returns the content of *box*. For any *v*, `(unbox (box v))` returns *v*.
- `(set-box! box v)` sets the content of *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:application:type` exception is raised. A box produced by `read` (via `#&`) is mutable. (See also `immutable?` in §3.8.)

## 3.10 Procedures

### 3.10.1 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  $\Rightarrow$  the procedure always takes exactly *a* arguments;
  - an **arity-at-least**<sup>4</sup> instance  $\Rightarrow$  the procedure takes (`arity-at-least-value a`) or more arguments; or
  - a list containing integers and **arity-at-least** instances  $\Rightarrow$  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:application: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:application:arity` exceptions, not the result of `procedure-arity`.

<sup>4</sup>All fields of the `arity-at-least` structure type are accessible by all inspectors (see §4.5).

### 3.10.2 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<sup>5</sup>RS* 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 §3.10.1). For most primitives, this procedure returns 1, since most primitives return a single value when applied. For information about arity values, see §3.10.1.
- `(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.

### 3.10.3 Procedure Names

See §6.2.4 for information about the names of primitives, and the names inferred for **lambda** and **case-lambda** procedures.

## 3.11 Promises

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

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

## 3.12 Hash Tables

`(make-hash-table [flag-symbol flag-symbol])` creates and returns a new hash table. If provided, each *flag-symbol* must be 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:application:mismatch` exception is raised.

`(hash-table? v)` returns `#t` if *v* was created by `make-hash-table`, `#f` otherwise.

`(hash-table-put! hash-table key-v v)` maps *key-v* to *v* in *hash-table*, overwriting any existing mapping for *key-v*.

`(hash-table-get hash-table key-v [failure-thunk])` returns the value for *key-v* in *hash-table*. If no value is found for *key-v*, then the result of invoking *failure-thunk* (a procedure of no arguments) is returned. If *failure-thunk* is not provided, the `exn:application:mismatch` exception is raised when no value is found for *key-v*.

`(hash-table-remove! hash-table key-v)` removes the value mapping for *key-v* if it exists in *hash-table*.

(`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 access.

(`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 access.

(`eq-hash-code` *v*) returns a number; for any two `eq?` values, the returned number is always the same. The number is an exact integer that is itself guaranteed to be `eq?` with any value representing the same exact integer (i.e., it is a fixnum).

(`equal-hash-code` *v*) returns a number; for any two `equal?` values, the returned number is always the same. The number is an exact integer that is itself guaranteed to be `eq?` with any value representing the same exact integer (i.e., it is a fixnum). If *v* contains a cycle consisting of pairs, vectors, boxes, and fully-inspectable structures, then `equal-hash-code` applied to *v* will loop indefinitely.

**Caveat concerning concurrent access:** 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` and `hash-table-for-each` procedures do not use the table's semaphore. Consequently, if a hash table is modified by another thread while a map or for-each is in process, arbitrary key-value pairs can be dropped or duplicated in the map or for-each.

**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 put and get operations become 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* is explained in §4.5.

A **define-struct** expression with *n* *fields* 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 25 and Chapter 17 of *PLT MzLib: Libraries Manual*). For detailed information about the expansion-time information stored in *s*, see §12.6.3.

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 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]`) 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* argument controls access to debugging information about the structure type and its instances; see §4.5 for more information.

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 access or mutator returned by `make-struct-type` into a field-specific access or mutator:

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

$$(\text{lambda } (s) (\text{accessor-proc } s \text{ field-pos-k}))$$

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

$$(\text{lambda } (s \ v) (\text{mutator-proc } s \ \text{field-pos-k } v))$$

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
```

## 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 properties values of their parent types, and only one value can be associated with a type for any property.

`(make-struct-type-property name-symbol)` 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; the the structure type does not have a value for the property, or if any other kind of value is provided, the `exn:application:type` exception is raised.

`(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 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))
```

The `current-inspector` parameter determines a default inspector argument for new structure types. An alternate inspector can be provided through 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 six 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;
  - *field-k*: the number of fields defined by the structure type (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`;
  - *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 fifth 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:application:mismatch` exception is raised.

## 4.6 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`. The 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 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<sup>5</sup>RS* 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<sup>5</sup>RS* and this manual, the *body-datums* 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*

```
(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 `(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* ...)

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 scope of all imported identifiers covers the entire module body, as does the scope of any identifier defined within the module body. An *identifier* can be defined by a definition or import at most once. 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.

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<sup>5</sup>RS* 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.

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 §12.6.4 for information on extracting details about an expanded or compiled module declaration.

### 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 three forms:

*unix-relative-path-string*

(**file** *path-string*)  
 (**lib** *filename-string collection-string* ...)

- 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 alpha-numeric characters plus `-`, `_`, `.`, and space, and the path may not 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.

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 §7.4.1.12). The resolver is a function that takes three arguments — an arbitrary value for the module path, a symbol for the source module's name, and a syntax object or `#f` — and returns a symbol for the resolved name.

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. (See §11.3 for details on platform-specific path handling.) The standard module name resolver also keeps a per-namespace table of loaded module identifiers. If the resolved identifier is not in the table, 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).

The current module name resolver is also called by `namespace-attach-module` to notify the resolver that a module was attached to a namespace (and shouldn't be loaded in the future). In this notification mode, the first argument to the resolver is `#f`, the second argument is the name of the attached module, and the third argument is `#f`.

### 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.

## 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 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.

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 has no such exported variable, the `exn:application:mismatch` 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). The result is void.

(**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 (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 namespace essentially maps to a "sub-namespace" containing the module's definitions.

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 "sub-namespace" consistent with the module declaration.

If a module was originally declared for a namespace via **namespace-attach-module**, then it cannot be re-declared (and the `exn:module` exception is raised if a re-declaration is attempted). If a module re-declaration creates an import cycle, the `exn:module` 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 via **namespace-attach-module**, so they cannot be re-declared.

## 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:module` 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 §7.4.1.3) to the following states:

```
(read-case-sensitive #f)
(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)
```

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 sensitive by prefixing the module with `#cs` (see §14.3).

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:module` exception is raised when such a cycle is detected.

## 6. Exceptions and Control Flow

---

### 6.1 Exceptions

MzScheme supports the exception system proposed by Friedman, Haynes, and Dybvig.<sup>1</sup> MzScheme’s implementation extends that proposal by defining the specific exception values that are raised by each primitive error.

- (**raise** *exn*) raises an exception, where *exn* represents the exception being raised. The *exn* argument can be anything; it is passed to the current *exception handler*. Breaks are disabled while the exception handler is called; see §6.6 for more information.
- (**current-exception-handler**) returns the current exception handler that is used by **raise**, and (**current-exception-handler** *f*) installs the procedure *f* as the current exception handler. The **current-exception-handler** procedure is a parameter; see §7.4.1.7 for more information.

Any procedure that takes one argument can be an exception handler, but it is an error if the exception handler returns to its caller when invoked by **raise**. (If an exception handler returns, the current error display handler and current error escape handler are called directly to report the handler’s mistake.)

The default exception handler prints an error message using the current error display handler (see **error-display-handler** in §7.4.1.7) and then escapes by calling the current error escape handler (see **error-escape-handler** in §7.4.1.7). If an exception is raised while an exception handler is executing, an error message is printed using a primitive error printer and the primitive error escape handler is invoked.

- (**with-handlers** ((*pred handler*) ...) *expr* ...) is a syntactic form that evaluates the *expr* body, installing a new exception handler before evaluating the *exprs* and restoring the handler when a value is returned (or when control escapes from the expression). The *pred* and *handler* expressions are evaluated in the order that they are specified, before the first *expr* and before the exception handler is changed. The exception handler is installed and restored with **parameterize** (see §7.4.2).

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 original exception handler is invoked (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 *exprs*, 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. The “original” exception handler (the one present before the **with-handlers** expression was evaluated) is therefore re-installed before any predicate or handler procedure is invoked.

A particularly useful predicate procedure is **not-break-exn?**. (**not-break-exn?** *v*) returns **#f** if *v* is an instance of *exn:break* (representing an asynchronous break exception), **#t** otherwise.

---

<sup>1</sup>See <http://www.cs.indiana.edu/scheme-repository/doc.proposals.exceptions.html>

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:application:match:zero?
                    (lambda (exn) +inf.0)])
      (/ n d))))
```

The following example catches and ignores file exceptions, but lets the enclosing context handle breaks:

```
(define (file-date-if-there filename)
  (with-handlers ([not-break-exn? (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 §7.4.1.7).

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, applying a procedure to the wrong number of arguments raises an exception as an instance of `exn:application:arity`. An exception handler can test for this kind of exception using the global `exn:application:arity?` predicate. Given such an exception, the (incorrect) number of arguments provided is obtained from the exception with `exn:application-value`, while `exn:application:arity-expected` accesses the actual arity of the procedure.

- `exn` : *not instantiated directly*
  - fields: `message` — error message (type: *immutable-string*)
  - `continuation-marks` — value returned by `current-continuation-marks` immediately after the error is detected (type: *mark-set*)
- `user` : raised by calling `error`
- `variable` : unbound global or module variable at run time
  - fields: `id` — the unbound variable’s global identifier (type: *symbol*)
- `application` : *not instantiated directly*
  - fields: `value` — the error-specific inappropriate value (type: *value*)
  - `arity` : application with the wrong number of arguments
    - fields: `expected` — the correct procedure arity as returned by `arity` (type: *arity*)
  - `type` : wrong argument type to a procedure, not including divide-by-zero
    - fields: `expected` — name of the expected type (type: *symbol*)
  - `mismatch` : bad argument combination (e.g., out-of-range index for a vector) or platform-specific integer range error
  - `divide-by-zero` : divide by zero; `application-value` is always zero
  - `continuation` : attempt to cross a continuation boundary or apply another thread’s continuation

- **syntax** : syntax error, but not a **read** error
  - fields: **expr** — illegal expression (or #f if unknown) (type: *syntax object or #f*)
  - form** — the syntactic form name that detected the error (or #f if unknown) (type: *symbol or #f*)
  - module** — the form-defining module (or #f if unknown) (type: *symbol, module path index, or #f*)
- **read** : read parsing error
  - fields: **source** — source name (type: *value*)
  - line** — source line (type: *positive exact integer or #f*)
  - column** — source column (type: *positive exact integer or #f*)
  - position** — source position (type: *positive exact integer or #f*)
  - span** — source span (type: *non-negative exact integer or #f*)
- **eof** : unexpected end-of-file
- **non-char** : unexpected non-character
- **i/o** : *not instantiated directly*
  - **port** : *not instantiated directly*
    - fields: **port** — port for attempted operation (type: *port*)
  - **read** : error reading from a port
  - **write** : error writing to a port
  - **closed** : attempt to operate on a closed port
- **filesystem** : illegal pathname or error manipulating a filesystem object
  - fields: **pathname** — file or directory pathname (type: *path*)
  - detail** — 'ill-formed-path, 'already-exists, or 'wrong-version, indicating the reason for the exception (if available), or #f (type: *symbol or #f*)
- **tcp** : TCP errors
- **thread** : raised by **call-with-custodian**
- **module** : raised by **module**, **require**, etc.
- **break** : asynchronous thread break
  - fields: **continuation** — a continuation that resumes from the break (type: *continuation*)
- **misc** : low-level or MzScheme-specific error
  - **unsupported** : unsupported feature
  - **out-of-memory** : out of memory

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:application:type** if the arity is incorrect.

## 6.2 Errors

The procedure **error** raises the exception **exn:user** (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 *vs* (as produced by the current error value conversion handler; see §7.4.1.7). 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:user` and the resulting exception is raised.

### 6.2.1 Application Type Errors

`(raise-type-error name-symbol expected-string v)` creates an `exn:application:type` value and raises 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 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 *vs*, the `exn:application:mismatch` exception is raised.

### 6.2.2 Application Mismatch Errors

`(raise-mismatch-error name-symbol message-string v)` creates an `exn:application:mismatch` value and raises 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 §7.4.1.7).

### 6.2.3 Syntax Errors

`(raise-syntax-error name message-string [expr sub-expr])` creates an `exn:syntax` value and raises 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) as the `expr` field of the generated exception record, else the *expr* is used if provided, otherwise the `expr` field is `#f`. Source location information for the error message 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 and the values of the `form` and `module` fields of the generated exception are determined through a combination of the *name*, *expr*, and *sub-expr* arguments. The *name* argument can be any of three kinds of values:

- `#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, the `form` field of the exception is the third result of `identifier-binding` applied to the identifier, and the `module` field of the exception is the fourth result of `identifier-binding` applied to the identifier. (See §12.3.2 for information about `identifier-binding`.)

If *expr* is not provided, or if it is not an identifier or a syntax pair containing an identifier as its first element, then the form name in the error message is `"?"`, the `form` field of the exception is `#f`, and the `module` field of the exception is `#f`.

- *symbol*: When *name* is a symbol, then the symbol is used as the form name in the generated error message. If *expr* is provided, and it is either an identifier or a syntax pair whose first element is an identifier, then the exception fields are computed in the same way as when *name* is `#f`. Otherwise, the `form` field of the exception is *name*, and the `module` field of the exception is `#f`.
- `(list msg-symbol form mod)`: When *name* is a list of three items, the first is used as the form name in the generated error message, the second (which can be a symbol or `#f`) is used as the `form` field of

the generated exception, and the last (which can be a module index path, a symbol, or #f) is used as the `module` field of the generated exception.

See also §10.

### 6.2.4 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.

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

`(object-name v)` returns a symbol or immutable string 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, and regexp values have names. Only regexp values have string names (the source of the regexp); other names are symbols. All primitive procedures have names (see §3.10.2).

## 6.3 Continuations

MzScheme supports fully re-entrant `call-with-current-continuation` (or `call/cc`). 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))
```

A continuation can only be invoked from the thread (see Chapter 7) in which it was captured. Multiple return values can be passed to a continuation (see §2.2).

In addition to regular `call/cc`, MzScheme provides `call-with-escape-continuation` (or `call/ec`) and `let/ec`. A continuation obtained from `call/ec` can only be used to *escape* back to the continuation; i.e., an escape continuation is only valid when the current continuation is an extension of the escape continuation. The application of `call/ec`'s argument is not a tail call.

Escape continuations are provided for two reasons: 1) they are significantly cheaper than full continuations; and 2) full continuations are not allowed to cross certain boundaries (e.g., error handling) that escape continuations can safely cross.

The `exn:application:continuation` exception is raised when a continuation is applied by the wrong thread, a continuation application would violate a continuation boundary, or 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 an exception 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. 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.

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

## 6.5 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. Continuation marks support debuggers and other program-tracing facilities; in particular, continuation frames roughly correspond to stack frames in traditional languages. For example, a debugger can annotate a source program to store continuation marks that relate each expression to its source location; when an exception occurs, the marks are extracted from the current continuation to produce a “stack trace” for the exception.

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 l0)`, 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:

- `(continuation-marks cont)` returns an opaque value containing the set of continuation marks for all keys in the continuation *cont*.
- `(current-continuation-marks)` returns an opaque value containing the set of continuation marks for all keys in the current continuation. In other words, it produces the same value as `(call-with-current-continuation continuation-marks)`.

The `continuation-mark-set->list` procedure extracts mark values for a particular key from a continuation mark set:

- `(continuation-mark-set->list mark-set key-v [skip-v])` 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`. If `skip-v` is provided, then it is inserted into the list once for every consecutive sequence of frames without a `key-v` mark.
- `(continuation-mark-set? v)` returns `#t` if `v` is a mark set created by `continuation-marks` or `current-continuation-marks`, `#f` otherwise.

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` 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` was invoked.

## 6.6 Breaks

A *break* is an asynchronous exception, usually triggered through an external source controlled by the user, or through the `break-thread` procedure (see §7.1). 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 afterwards; 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 (see §7.4.1.8). Independent of the parameter setting, breaks are disabled while an exception handler is executing. Note that the handling procedures supplied to `with-handlers` are *not* exception handlers, so breaking within such procedures is controlled by `break-enabled`. Breaks are also disabled (independent of parameter settings) during the evaluation of the “pre” and “post” thunks for a `dynamic-wind`, whether called during the normal `dynamic-wind` calling sequence or via a continuation jump.

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.

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 §7.2), 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 break 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 the `read-string-avail!/enable-break` and `write-string-avail/enable-break` procedures (see §11.2.2).

## 6.7 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 §7.4.1.7).

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

The error escape handler is normally called directly by an exception handler. To escape from a run-time error, use `raise` (see §6.1) or `error` (see §6.2) instead.

If an exception is raised while the error escape handler is executing, an error message is printed using a primitive error printer and a primitive error escape handler is invoked.

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

```
(let ([orig (error-escape-handler)])
  (let/ec exit
    (let retry-loop ()
      (let/ec escape
        (error-escape-handler
         (lambda () (escape #f)))
        (let loop ()
          (let ([e (my-read)])
            (if (eof-object? e)
                (exit 'done)
                (let ([v (my-eval e)])
                  (my-print v)
                  (loop))))))
          (retry-loop)))
      (error-escape-handler orig))
```

See also `read-eval-print-loop` in §14.1 for a simpler implementation of this example.

## 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 §7.4), such as the current directory or current exception handler. A newly-created thread inherits the parameter settings of the creating thread, except

- the `error-escape-handler` parameter, which is initialized to the default error escape handler; and
- the `current-exception-handler` parameter, which is initialized to the value of `initial-exception-handler`.

When a thread is created, it is placed into the management of the current custodian (See §9.2). A thread that has not terminated can be “garbage collected” only if it is unreachable and blocked on an unreachable semaphore.<sup>1</sup>

### 7.1 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 for at least *x* seconds, where *x* is a non-negative real number. The *x* argument defaults to 0 (allowing other threads to execute when operating system threads are not used). The value of *x* can be non-integral to request a sleep duration to any precision, but the precision of the actual sleep time is unspecified.

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

(`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 §6.6).

---

<sup>1</sup>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 collectable when the eventspace is unreachable and contains no visible windows or running timers.

(`kill-thread thread`) terminates the specified thread immediately. Terminating the main thread exits the application. If *thread* is already not running, `kill-thread` does nothing. Otherwise, if the current custodian (see §9.2) does not manage *thread* (and none of its subordinates manages *thread*), the `exn:misc` exception is raised.

All of the MzScheme (and MrEd) primitives are kill-safe; that is, killing a thread never interferes with the application of primitives 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.

(`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 §6.6).

(`call-in-nested-thread thunk [custodian]`) creates a nested thread managed by *custodian* to execute *thunk*.<sup>2</sup> 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.

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:thread` 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.

## 7.2 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, of course, so that semaphore can be used for reliable synchronization.

- (`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:application:mismatch` 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:misc` exception is raised.
- (`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.

<sup>2</sup>The nested thread's current custodian is inherited from the creating thread, independent of the *custodian* argument.

- (`semaphore-wait/enable-break sema`) is like `semaphore-wait`, but breaking is enabled (see §6.6) 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.

See also `object-wait-multiple` in §7.3.

### 7.3 Synchronizing Multiple Objects with Timeout

(`object-wait-multiple timeout v ...1`) blocks as long as all of the `vs` are in a blocking state, as defined below, or until `timeout` seconds have passed. The `timeout` argument can be a real number or `#f`; if `timeout` is `#f`, then `object-wait-multiple` does not return until some `v` is unblocked.

When at least one `v` is unblocked, it is returned as the result. If multiple `vs` are unblocked before the call to `object-wait-multiple`, the returned `v` is the earliest unblocked `v` according to argument order. If multiple `vs` become unblocked later, any one of the unblocked `vs` can be returned (but only one).

If the returned `v` is a semaphore, then the semaphore's internal count is decremented, just as with `semaphore-wait`. Otherwise, the returned `v` is unmodified. Any `v` that is not returned by `object-wait-multiple` is unmodified.

If `object-wait-multiple` returns because `timeout` seconds have passed, the return value is `#f`. Each `v` is checked at least once, so a `timeout` value of 0 can be used for polling.

Only certain kinds of values, listed below, are waitable. If any other kind of value is provided to `object-wait-multiple`, the `exn:application:mismatch` exception is raised.<sup>3</sup>

- *semaphore* — a semaphore is in a blocking state when `semaphore-wait` (see §7.2) would block.
- *input-port* — an input port is in a blocking state when `read-char` would block.
- *output-port* — an output port is in a blocking state if `write-string-avail` would block (see §11.2.2), unless the port contains buffered characters and `write-string-avail*` can flush part of the buffer (although `write-string-avail` might block).
- *tcp-listener* — a TCP listener is in a blocking state when `tcp-accept` (see §11.4) would block.
- *thread* — a thread is in a blocking state when `thread-wait` (see §7.1) would block.
- *subprocess* — a subprocess is in a blocking state when `subprocess-wait` (see §15.2) would block.
- *will-executor* — a will executor is in a blocking state when `will-execute` (see §13.2) would block.

(`object-wait-multiple/enable-break v ...1`) is like `object-wait-multiple`, but breaking is enabled (see §6.6) while waiting on the `vs`. If breaking is disabled when `object-wait-multiple/enable-break` is called, then either all `vs` remain unmodified or the `exn:break` exception is raised, but not both.

(`object-waitable? v`) returns `#t` if `v` is a waitable object, `#f` otherwise. See `object-wait-multiple`, above, for the list of waitable object types.

### 7.4 Parameters

A *parameter* is a thread-specific setting, such as the current output port or the current directory for resolving relative pathnames. A *parameter procedure* sets and retrieves the value of a specific parameter. For example,

<sup>3</sup>An extension or embedding application can extend the set of waitable values.

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 applying a parameter procedure to a single argument sets the parameter's value in the current thread (and returns void). For example, `(current-output-port)` returns the current default output port, while `(current-output-port p)` sets the default output port to `p`.

### 7.4.1 Built-in Parameters

MzScheme's built-in parameter procedures are listed in the following sections. The `make-parameter` procedure, described in §7.4.2, creates a new parameter and returns a corresponding parameter procedure.

#### 7.4.1.1 CURRENT DIRECTORY

- `(current-directory [path])` gets or sets a string path that determines the current directory. When the parameter procedure is called to set the current directory, the path argument is expanded and then simplified using `simplify-path` (see §11.3.1) and converted to an immutable string; expansion and simplification raise an exception if the path is ill-formed. Otherwise, if the given path cannot be made the current directory (e.g., because the path does not exist), the `exn:i/o:filesystem` exception is raised.

#### 7.4.1.2 PORTS

- `(current-input-port [input-port])` gets or sets an input port used by `read`, `read-char`, `char-ready?`, `read-line`, `read-string`, and `read-string-avail!` 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`, and `printf` 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 §11.2.5) to print values into a port.

#### 7.4.1.3 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 downcases 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 §14.3 for more information. While a module is loaded, the parameter is set to `#f` (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 §14.3 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 §14.3 for more information.
- `(read-accept-box [on?])` gets or sets a boolean value that controls parsing `#\&` input. See §14.3 for more information.

- (`read-accept-compiled` [*on?*]) gets or sets a boolean value that controls parsing pre-compiled input. See §14.3 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 §14.3 and §14.4 for more information.
- (`read-accept-graph` [*on?*]) gets or sets a boolean value that controls parsing input S-expressions with sharing. See §14.5 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 §14.5 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 §14.3 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 §14.3 for more information.

#### 7.4.1.4 PRINTING

- (`print-graph` [*on?*]) gets or sets a boolean value that controls printing S-expressions with sharing. See §14.5 for more information.
- (`print-struct` [*on?*]) gets or sets a boolean value that controls printing structure values. See §14.4 for more information.
- (`print-box` [*on?*]) gets or sets a boolean value that controls printing box values. See §14.4 for more information.
- (`print-vector-length` [*on?*]) gets or sets a boolean value that controls printing vectors. See §14.4 for more information.

#### 7.4.1.5 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 (`read-syntax` *name-string*), where *name-string* corresponds to the current input source.
- (`current-eval` [*proc*]) gets or sets a procedure that takes an S-expression and returns its value (or values; see §2.2). This *evaluation handler* is called by `eval`, the default load handler, and `read-eval-print-loop` to evaluate an expression (see §14.1). The default evaluation handler compiles and executes the expression in the current namespace (determined by the `current-namespace` parameter).
- (`current-namespace` [*namespace*]) gets or sets a namespace value (see §8) that determines the global variable namespace used to resolve variable references. The *current namespace* is used by the default evaluation handler, the `compile` procedure, and other built-in procedures that operate on global variables.
- (`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 prints the value to the current output port (determined by the `current-output-port` parameter) and then outputs a newline.

- (`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: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*.

#### 7.4.1.6 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 file path string 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 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 §14.3). 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 loads a dynamic extension (see §14.7) 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 file path string 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 pathname 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 pathname is provided to the parameter procedure `current-load-relative-directory`, it is immediately expanded (see §11.3.1) and converted to an immutable string; the result must be a complete pathname for an existing directory.

- (`use-compiled-file-kinds` [*kind-symbol*]) gets or sets a symbol, either `'all` or `'none`, indicating whether `load/used-compiled` (and thus `require`) should recognize compiled files. If the value of this parameter is `'all`, then the default handler for `load/use-compiled` recognizes compiled files as described in §14.1. If the value is `'none`, then the default handler for `load/use-compiled` ignores compiled files.

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

- (`current-command-line-arguments` [*string-vector*]) gets or sets a vector of strings representing command-line arguments. The stand-alone version of MzScheme (and MrEd) initializes the parameter to contain extra command-line arguments. (The same vector is also installed as the value of the `argv` global.)

## 7.4.1.7 EXCEPTIONS

- (**current-exception-handler** [*proc*]) gets or sets a procedure that is invoked to handle an exception. See §6.1 for more information about exceptions.
- (**initial-exception-handler** [*proc*]) gets or sets a procedure that is used as the initial current exception handler for a new thread.
- (**error-escape-handler** [*proc*]) gets or sets a procedure that takes no arguments and escapes from the dynamic context of an exception. The default error escape handler escapes to the start of the current thread, but **read-eval-print-loop** (see §14.1) also sets the escape handler. To report a run-time error, use **raise** (see §6.1) or **error** (see §6.2) instead of calling the error escape procedure directly. If an exception is raised while the error escape handler is executing, an error message is printed using a primitive error printer and the primitive error escape handler is invoked. Unlike all other parameters, the value of the **error-escape-handler** parameter in a new thread is not inherited from the creating thread; instead, the parameter is always initialized to the default error escape handler.
- (**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. The default error display handler **displays** its first argument to the current error port (determined by the **current-error-port** parameter), and ignores the second argument. To report a run-time error, use **raise** (see §6.1) or **error** (see §6.2) instead of calling the error display procedure directly. If an exception is raised while the error display handler is executing, an error message is printed using a primitive error printer and the primitive error escape handler is invoked.
- (**error-print-width** [*k*]) gets or sets an 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-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 **prints** the value into a string;<sup>4</sup> 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.
- (**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. Only the message field of an **exn:read** or **exn:syntax** structure is affected by the parameter. The default is **#t**.

## 7.4.1.8 SECURITY

- (**break-enabled** [*enabled?*]) gets or sets a boolean value that controls whether breaks are allowed. See §6.6 for more information.
- (**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, and TCP listeners.

---

<sup>4</sup>Using the current global port print handler; see §7.4.1.2.

- (`current-inspector` [*inspector*]) gets or sets an inspector (see §4.5) that controls debugging access to newly created structure types.

#### 7.4.1.9 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.

#### 7.4.1.10 RANDOM NUMBERS

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

#### 7.4.1.11 LOCALE

- (`current-locale` [*string-or-#f*]) gets or sets a string/boolean value that controls the interpretation of characters for functions such as `char-locale<?`, `char-locale-upcase`, and `string-locale<?` (see §3.4 and §3.5). When locale sensitivity is disabled by setting the parameter to `#f`, characters are interpreted 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.<sup>5</sup> String and character printing with `write` is affected by the parameter, because unprintable characters are printed with escapes (see §14.4). Case conversion for case-insensitive symbols and regular expression patterns (see §10) are *not* affected. The parameter's default value is `""`.

#### 7.4.1.12 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*]) gets or sets a symbol prefixed onto a module declaration when it is evaluated. This parameter is intended for use by a module name resolver; see §5.4 for more information.

### 7.4.2 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.

(`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.

The `parameterize` form evaluates an expression with temporarily values installed for a group of parameters. The syntax of `parameterize` is:

---

<sup>5</sup>The "C" locale is also always available; setting the locale to "C" is the same as disabling locale sensitivity with `#f` only when string and character operations are restricted to the first 128 characters.

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

The result of a **parameterize** expression is the result of the last *body-expr*. The *parameter-exprs* determine the parameters to set, and the *value-exprs* determine the corresponding values to install before evaluating the *body-exprs*. All of the *parameter-exprs* are evaluated first (checked with **check-parameter-procedure**), then all *value-exprs* are evaluated, and then the parameters are set.

After the *body-exprs* are evaluated, each parameter's setting is restored to its original value in the dynamic context of the **parameterize** expression. More generally, the values specified by the *value-exprs* determine initial “remembered” values, and whenever control jumps into or out of the *body-exprs*, the value of each parameter is swapped with the corresponding “remembered” value.

Examples:

```
(parameterize ([exit-handler (lambda (x) 'no-exit)])
  (exit)) ; => 'no-exit
```

```
(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)
```

(**check-parameter-procedure** *v*) returns *v* if it is a procedure that can take both 0 arguments and 1 argument, and raises **exn:application:type** otherwise. The **check-parameter-procedure** procedure is used in the expansion of **parameterize**.

## 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 §7.4.1.5), or providing the namespace as the second argument to `eval`. The MzScheme versions of the R5RS procedures `scheme-report-environment` and `null-environment` produce namespaces.<sup>1</sup>

The current namespace is used by `eval`, `load`, `compile`, `expand`, and `expand-once`.<sup>2</sup> 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 two top-level environments: one for normal expressions, and one for macro transformer expressions; see §12 for more information about the transformer 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 variable or syntax bindings, and its module instances and imports are distinct from the instances and imports of the normal top-level environment.

### 8.1 Identifier Resolution in Namespaces

Identifier resolution in the 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

---

<sup>1</sup>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<sup>5</sup>RS* identifiers.

<sup>2</sup>More precisely, the current namespace is used by the evaluation and load handlers, rather than directly by `eval` and `load`.

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 contains module 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; 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*]) returns a value for *symbol* in 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: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: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: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:variable` or `exn:syntax` exception.

(`namespace-set-variable-value!` *symbol* *v* [*map?*]) sets the value of *symbol* in the top-level environment of 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) to that *symbol* maps to the variable.

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

(`namespace-require` *quoted-require-spec*) performs the import corresponding to *quoted-require-spec* in the top-level environment (like a top-level **require** expression). See also Chapter 5.

(`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.

(`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-symbol*) attaches the instantiated module named *module-symbol* in *src-namespace* to the current namespace, using *module-symbol* as the module name in the current namespace. In addition to the *module-symbol* module itself, every module that it imports (directly or indirectly) is also transferred into the current namespace. If *module-symbol* is not the name of an instantiated module in *src-namespace*, or if the name of any module to be transferred already has a different declaration or instance in the current namespace, then the `exn:application:mismatch` exception is raised.

## 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.
- Namespaces (§8) control access to Scheme bindings.

All security mechanisms rely on thread-specific parameters (see §7.4).

### 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 §7.4.1.8). 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)` 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.
- an immutable string representing a pathname, or `#f` to check access for pathless queries, such as `(current-directory)`, `(filesystem-root-list)`, and `(find-system-path ...)`. A path string provided to *file-proc* is not processed at all before checking access; it may be a relative path, and it may be ill-formed.
- 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, the accepting hostname for a listening server, or #f for a listening server that accepts connections at all of the host's address.
- an exact integer between 1 and 65535 (inclusive) representing the port number. 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 return value of *file-proc* or *network-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* and *network-proc* procedures are invoked in the thread that called the access-checked primitive. Breaks may or may not be enabled (see §6.6). 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, subprocess ports, TCP ports, and TCP listeners.<sup>1</sup> Whenever a thread, file-stream port, process port, TCP port, or TCP listener is created, it is placed under the management of the current custodian (as determined by the *current-custodian* parameter; see §7.4.1.8).

The only operation on a custodian is to shut down all of 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 a set of threads. For example, 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.

The values managed by a custodian are only weakly held by the custodian. As a result, a will (see §13.2) can be executed for a value that is managed by a custodian.

(*make-custodian [custodian]*) creates a new custodian that is subordinate to the custodian *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.

<sup>1</sup>In MrEd, custodians also manage eventspaces.

(`custodian-shutdown-all` *custodian*) kills all running threads, closes all open ports, and closes all active TCP listeners that are managed by the custodian *custodian*. If *custodian* manages the current thread, the custodian shuts down all other objects before killing the current thread.

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

(`custodian-require-memory` *need-k* *thunk*) registers a require check if MzScheme is compiled with support for memory accounting, otherwise the `exn:misc:unsupported` exception is raised. If a check is registered, and if MzScheme later reaches a state after garbage collection (see §13.3) where *need-k* bytes are not available to the current custodian, *thunk* is invoked.

(`custodian-limit-memory` *custodian* *limit-k* *thunk*) registers a limit check if MzScheme is compiled with support for memory accounting, otherwise the `exn:misc:unsupported` exception is raised. If a check is registered, and if MzScheme later reaches a state after garbage collection (see §13.3) where *custodian* owns more than *limit-k* bytes, then *thunk* is invoked.

## 10. Regular Expressions

---

MzScheme provides built-in support for regular expression pattern matching on strings and input ports, built on Henry Spencer’s package. Regular expressions are specified as strings, using the same pattern language as the Unix utility `egrep`. String-based regular expressions can be compiled into a *regexp value* for repeated matches. 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 special characters.

The `pregexp.ss` library of MzLib (see Chapter 20 of *PLT MzLib: Libraries Manual*) provides a similar—but more powerful—form of matching.

The format of a regular expression is specified by the grammar in Figure 10.1. 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), the resulting regexp is legal even though it is usually not matchable. For example, “a\$b” 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:

- (`regexp string`) takes a string representation of a regular expression 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 §6.2.4) returns the source string for a regexp value.

- (`regexp? v`) returns `#t` if `v` is a regexp value created by `regexp`, `#f` otherwise.
- (`regexp-match pattern string [start-k end-k output-port]`) attempts to match *pattern* (a string or a regexp value) to a portion of *string*; see below for information on using an 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.

---

<i>Regexp</i>	::=	<i>Pieces</i>	Match <i>Pieces</i>
		<i>Regexp</i>   <i>Regexp</i>	Match either <i>Regexp</i> , try left first
<i>Pieces</i>	::=	<i>Piece</i>	Match <i>Piece</i>
		<i>Piece</i> <i>Piece</i>	Match first <i>Piece</i> followed by second <i>Piece</i>
<i>Piece</i>	::=	<i>Atom</i> *	Match <i>Atom</i> 0 or more times, longest possible
		<i>Atom</i> +	Match <i>Atom</i> 1 or more times, longest possible
		<i>Atom</i> ?	Match <i>Atom</i> 0 or 1 times, longest possible
		<i>Atom</i> *?	Match <i>Atom</i> 0 or more times, shortest possible
		<i>Atom</i> +?	Match <i>Atom</i> 1 or more times, shortest possible
		<i>Atom</i> ??	Match <i>Atom</i> 0 or 1 times, shortest possible
		<i>Atom</i>	Match <i>Atom</i> exactly once
<i>Atom</i>	::=	( <i>Regexp</i> )	Match sub-expression <i>Regexp</i>
		[ <i>Range</i> ]	Match any character in <i>Range</i>
		[^ <i>Range</i> ]	Match any character not in <i>Range</i>
		.	Match any character
		^	Match start of string
		\$	Match end of string
		<i>Literal</i>	Match a single literal character
<i>Literal</i>	::=	Any character except (, ), *, +, ?, [, ], ., ^, \, or	
		\ <i>Aliteral</i>	Match <i>Aliteral</i>
<i>Aliteral</i>	::=	Any character	
<i>Range</i>	::=	]	<i>Range</i> contains ] only
		-	<i>Range</i> contains - only
		] <i>Lrange</i>	<i>Range</i> contains ] and everything in <i>Lrange</i>
		- <i>Lrange</i>	<i>Range</i> contains - and everything in <i>Lrange</i>
		<i>Lrange</i> -	<i>Range</i> contains - and everything in <i>Lrange</i>
		] <i>Lrange</i> -	<i>Range</i> contains ], -, and everything in <i>Lrange</i>
		<i>Lrange</i>	<i>Range</i> contains everything in <i>Lrange</i>
<i>Lrange</i>	::=	<i>Rliteral</i>	<i>Range</i> contains a literal character
		<i>Rliteral</i> - <i>Rliteral</i>	<i>Range</i> contains ASCII range inclusive
		<i>Lrange</i> <i>Lrange</i>	<i>Range</i> contains everything in both
<i>Rliteral</i>	::=	Any character except ] or -	

Figure 10.1: Grammar for regular expressions

---

If the match fails, `#f` is returned. If the match succeeds, a list containing strings, and possibly `#f`, is returned. The first string in this 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 strings are returned in the list if *pattern* contains parenthesized sub-expressions; 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.

- (`regexp-match pattern input-port [start-k end-k output-port]`) is similar to `regexp-match` with a string (see above), except that the match is found in the stream of characters produced by *input-port*. The optional *start-k* argument indicates the number of characters to skip before matching *pattern*, and *end-*

$k$  indicates the maximum number of characters to consider (including skipped characters). The *end-k* argument can be #f, which is the same as not supplying *end-k*. The default is to skip no characters and read until the end-of-file if necessary. If the end-of-file is reached before *start-k* characters are skipped, the match fails.

In *pattern*, a start-of-string caret (“^”) refers to the first read position after skipping, and the end-of-string dollar sign (“\$”) refers to the *end-k*th read character or the end of file, whichever comes first.

The optional *output-port* receives all characters that precede a match in the input port, or up to *end-k* characters (by default the entire stream) if no match is found.

When matching an input port stream, all characters up to and including the match are eventually read from the port, but matching proceeds by first peeking characters from the port (using `peek-string-avail!`; see §11.2.1), and then (re-)reading characters to discard them after the match result is determined. The matcher peeks in blocking mode only as far as necessary to determine a match, but it may peek extra characters 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 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 §11.1.6), then the characters that are peeked and used for matching may be different than the characters read and discarded after the match completes. The matcher inspects only the peeked characters.

- (`regexp-match-positions pattern string-or-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 in *string-or-input-port* in a **substring-compatible** manner for strings, independent of *start-k*. In the case of an input port, the returned positions indicate the number of characters that were read before the first matching character.
- (`regexp-match-peek pattern input-port [start-k end-k]`) is like `regexp-match` on input ports, but only peeks characters from *input-port* instead of reading them.
- (`regexp-match-peek-positions pattern input-port [start-k end-k]`) is like `regexp-match-positions` on input ports, but only peeks characters from *input-port* instead of reading them.
- (`regexp-replace pattern string insert-string`) performs a match using *pattern* on *string* and then returns a string in which the matching portion of *string* is replaced with *insert-string*. If *pattern* matches no part of *string*, then *string* is returned unmodified.

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*.<sup>1</sup> “&” and “\0” are synonymous. If the  $n$ th sub-expression was not 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* 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.

#### Examples:

<sup>1</sup>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”.

```
(define r (regexp "-[0-9]*+"))
(regex-match r "a-12-345b") ; => '("-12-345" "-345")
(regex-match-positions r "a-12-345b") ; => '((1 . 10) (5 . 10))
(regex-match "x+" "12345") ; => #f
(regex-replace "mi" "mi casa" "su") ; => "su casa"
(define r2 (regexp "[Mm]i ([a-zA-Z]*))")
(define insert "\\1y \\2")
(regex-replace r2 "Mi Casa" insert) ; => "My Casa"
(regex-replace r2 "mi cerveza Mi Mi Mi" insert) ; => "my cerveza Mi Mi Mi"
(regex-replace* r2 "mi cerveza Mi Mi Mi" insert) ; => "my cerveza My Mi Mi"
```

# 11. Input and Output

---

## 11.1 Ports

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. The predicate `port?` returns `#t` only for values for which either `input-port?` or `output-port?` returns `#t`.

### 11.1.1 Current Ports

The standard Scheme procedures `current-input-port` and `current-output-port` are implemented as parameters in MzScheme. See §7.4.1.2 for more information.

### 11.1.2 Opening File Ports

The `open-input-file` and `open-output-file` procedures accept an optional flag argument after the file-name that specifies a mode for the file:

- `'binary` — characters are returned from the port exactly as they are read from the file. Binary mode is the default mode.
- `'text` — return and linefeed characters written 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 characters 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.
  - **Mac OS Classic reading:** a return character read from the file is returned as a linefeed by the port; no filtering occurs for linefeeds.
  - **Mac OS Classic writing:** a return character written to the port is translated into a linefeed in the file; no filtering occurs for linefeeds.

In Windows, `'text` mode works only with regular files; attempting to use `'text` with other kinds of files triggers an `exn:i/o: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 an exception (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
- 'update — open an existing file without truncating it

If the 'update flag is specified and the file does *not* exist, the `exn:i/o:filesystem` exception is raised.

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:application:mismatch` 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 §7.4.2).

When an input or output file-stream port is created, it is placed into the management of the current custodian (see §9.2).

### 11.1.3 Pipes

(`make-pipe` [*limit-k*]) 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 characters (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 characters; writing to the pipe's output port thereafter will block until a read from the input port makes more space available.

### 11.1.4 String Ports

Scheme input and output can be read from or collected into a string:

- (`open-input-string` *string*) creates an input port that reads characters from *string*.
- (`open-output-string`) creates an output port that accumulates the output into a string.
- (`get-output-string` *string-output-port*) returns the string accumulated in *string-output-port*.

String input and output ports do not need to be explicitly closed. The `file-position` procedure, described in §11.1.5, 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"
```

### 11.1.5 File-Stream Ports

A port created by `open-input-file`, `open-output-file`, `subprocess`, and related function is a *file-stream port*. The initial input, output, and error ports in stand-alone MzScheme are also file-stream ports.

(`file-stream-port? port`) returns `#t` if the given port is a file-stream port, `#f` otherwise.

Three 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 §11.1.6) use buffers; when called on a port without a buffer, `flush-output` has no effect. By default, a file-stream port flushes its buffer automatically after each newline, 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`<sup>1</sup>, `read-line`, `read-string`, or `read-string-avail!` are performed on the initial standard input port.
- (`file-stream-buffer-mode file-stream-output-port [mode-symbol]`) gets or sets the buffer mode for `file-stream-output-port`. If `mode-symbol` is provided, it must be one of `'none`, `'line`, or `'block`, and the port's buffering is set accordingly. If `mode-symbol` is not provided, the current mode is returned. If the mode cannot be set or returned, the `exn:i/o:port` exception is raised.
- (`file-position port`) returns the current read/write position of `port`, and (`file-position port k`) sets the read/write position to `k`. The latter works only for file-stream and string ports, and raises the `exn:application:mismatch` exception for other port kinds. Calling `file-position` without a position on a non-file/non-string input port returns the number of characters that have been read from that port if the position is known (see §11.2.3), otherwise the `exn:i/o:port` 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:i/o:filesystem` exception is raised.

### 11.1.6 Custom Ports

The `make-custom-input-port` and `make-custom-output-port` procedures create ports with arbitrary control procedures.

#### 11.1.6.1 CUSTOM INPUT

(`make-custom-input-port waitable-or-false read-string-proc peek-string-proc-or-false close-proc`) 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.

- `waitable-or-false` — `#f` or an object that can be used with `object-wait-multiple` (e.g., a semaphore or another port).

If a waitable object is supplied, it is used by the system with to block until input on an end-of-file is ready for reading or peeking. If `waitable-or-false` is a semaphore, it will be re-posted after a block completes. The waitable object cannot be extracted from the port.

The port's reading and peeking procedures need not return data when the waitable unblocks, but spurious unblocks will reduce the port's performance. For example, a waitable might unblock when no data is available as a way of detecting demand on the port.

The waitable will not always be used before a call to a reading or peeking procedure (e.g., for a non-blocking read).

<sup>1</sup>Flushing is performed by the default port read handler (see §11.2.4) rather than by `read` itself.

If *waitable-or-false* is `#f`, the system assumes that input or an end-of-file is always available. In other words, supplying `#f` is the same as supplying `(make-semaphore 1)`.

- *read-string-proc* — a procedure that takes one argument, which is a non-empty string to fill with read characters, and returns either the number of characters read, a procedure for special inputs (see below), or `eof`. The procedure should never block; if no input is immediately available, it should return 0. If a value other than a non-negative exact integer, `eof`, or procedure-of-arity-four value is returned, the `exn:application:type` exception is raised. If the returned integer is larger than the supplied string, the `exn:application:mismatch` exception is raised.

The reading procedure can report an error by raising an exception, but only if no characters are read. Similarly, no characters should be read if `eof` or a procedure is returned. In other words, no characters should be lost due to spurious exceptions or non-character data.

A port's reading 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 the reading procedure to block.

- *peek-string-proc-or-false* — usually `#f`, which means that string peeking should be implemented by the system in terms of the reading procedure.

Otherwise, *peek-string-proc-or-false* must be a procedure that takes two arguments: a string to fill with peeked characters and a non-negative exact integer indicating a number of characters to skip in the input stream (but not in the string to fill) before writing peeked characters into the string.

The results and conventions for the procedure are the same as for *read-string-proc*, except that the peeking procedure can return an alternate waitable, usually in response to a peek for a non-zero skip. When a waitable is returned, the system blocks on the waitable before re-attempting a (blocking) peek operation with the same skip value. If the waitable is a semaphore, it will be re-posted after a successful wait. The waitable will not be made externally accessible.

The system does not check that multiple peeks return consistent results, or that peeking and reading produce consistent results. If peeking produces a procedure, then a future call to the reading procedure is expected to produce the same procedure, and the one returned by peeking is never invoked.

- *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 waitable, reading procedure, peeking procedure, and closing procedure will never be used again via the port is closed. However, the closing procedure can be called simultaneously in multiple threads (if the port is accessible in multiple threads).

When *read-string-proc* returns a procedure, the procedure is called by `read`,<sup>2</sup> `read-syntax`, or `read-char-or-special` to “read” non-character input from the port. The procedure is called exactly once before additional characters are read from the port, and the procedure must return two values: an arbitrary value and an exact, non-negative integer. The first return value is used as the read result, and the second is used as the width in characters of the result (for port position tracking). If *read-string-proc* or *peek-string-proc* returns a procedure when called by any reading procedure other than `read`, `read-syntax`, `read-char-or-special`, or `peek-char-or-special`, then the `exn:application:mismatch` exception is raised.

The four arguments to the procedure represent the source location of the non-character value, as much as it is known (see §11.2.3). The first argument is an arbitrary value representing the source for read values — the one passed to `read-syntax` — or `#f` if `read` or `read-char-or-special` was called. The second argument is a line number (exact, positive integer) if known, or `#f` otherwise. The third is a column number (exact, positive integer) or `#f`, and the fourth is a position number (exact, positive integer) or `#f`.

When the procedure returns a syntax object, then the syntax object is used directly in the result of `read-syntax`, and converted with `syntax-object->datum` for the result of `read`. If the result is not a syntax

<sup>2</sup>More precisely, the procedure is used by the default port read handler; see also §11.2.4.

object, then the result is used directly in the result for `read`, and converted with `datum->syntax-object` for the result of `read-syntax`. In either case, structure sharing that occurs only as a the result of multiple non-character results is not preserved as syntax sharing.

### 11.1.6.2 CUSTOM OUTPUT

(`make-custom-output-port` *waitable-or-false* *write-string-proc* *flush-proc* *close-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-string-proc*.

- *waitable-or-false* — `#f` or an object that can be used with `object-wait-multiple` (e.g., a semaphore or another port).

If a waitable object is supplied, it is used by the system with to block until the port is ready for writing at least one character without blocking, or ready to make progress in flushing an internal buffer without blocking. If *waitable-or-false* is a semaphore, it will be re-posted after a block completes. The waitable object cannot be extracted from the port.

Unlike the waitable object of input ports, the waitable object for an output port must be precise: it must not unblock unless the port is ready for writing. Otherwise, the guarantees of `object-wait-multiple` will be broken for the output port.

The waitable will not always be used before a call to the writing procedure (e.g., for a non-blocking write).

If *waitable-or-false* is `#f`, the system assumes that writes to the port will always succeed. In other words, supplying `#f` is the same as supplying (`make-semaphore` 1).

- *write-string-proc* — a procedure of four arguments:
  - an immutable string containing characters to write;
  - a non-negative exact integer for a starting offset (inclusive) into the string,
  - a non-negative exact integer for an ending offset (exclusive) into the string,
  - a boolean; `#t` indicates that the port is allowed to keep the written characters in a buffer, and that it is allowed to block indefinitely; `#f` indicates that the write should not block, and that the port should attempt to flush its buffer and completely write new characters instead of buffering them.

The procedure returns a non-negative exact integer representing the number of characters written and buffered, or `#f` if no characters could be written because the internal buffer could not be completely flushed. If the returned integer is larger than the supplied string, the `exn:application:mismatch` exception is raised. If the start and end indices are the same (i.e., no characters are to be written), then the final boolean argument will be `#t`, and the procedure should return 0 only if the buffer is completely flushed.

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 writing procedure raises an exception, due either to write or commit operations, it must not have committed any characters (though it may have committed previously buffered characters).

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 the writing procedure to block for a non-blocking write.

- *flush-proc* — a procedure of zero arguments that is called to flush the port's buffer, if any, in response to `flush-output`. The flushing operation can block. The flush and writing procedures can be called

simultaneously in multiple threads (if the port is accessible in multiple threads). If the flushing procedure is called while another thread is flushing the buffer, the call should not return until the flush has completed.

- *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 waitable, writing procedure, flushing procedure, and closing procedure will never be used again via the port 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 writing for flushing procedures by another thread; in the latter case, the write or flush must be terminated immediately with an error.

## 11.2 Reading and Writing

### 11.2.1 Reading

In addition to the standard reading procedures, MzScheme provides block reading procedures such as `read-line`, `read-string`, and `peek-string`:

- (`read-line` [*input-port mode-symbol*]) returns a string containing the next line of characters 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-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*).<sup>3</sup> 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-string-avail!` *string* [*input-port start-k end-k*]) reads characters from *input-port* and 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

<sup>3</sup>A temporary string of size *k* is allocated while reading the input, even if the size of the result is less than *k* characters.

length of the *string*. Like `substring`, the `exn:application:mismatch` 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 the string is not modified. If no characters are available before an end-of-file, then `eof` is returned. Otherwise, the return value is the number of characters read. If *m* characters are read and  $m < end-k - start-k$ , then *string* is not modified at indices  $start-k + m$  through *end-k*.

Unlike `read-string`, `read-string-avail!` returns without blocking after reading immediately-available characters. It blocks only if no characters are yet available. Also unlike `read-string`, `read-string-avail!` never drops characters; if `read-string-avail!` successfully reads some characters and then encounters an error, it suppresses the error (treating it roughly like an end-of-file) and returns the read characters. (The error will be triggered by future reads.) If an error is encountered before any characters have been read, an exception is raised.

- (`read-string-avail!* string [input-port start-k end-k]`) is like `read-string-avail!`, except that it returns 0 immediately if no characters are available for reading and the end-of-file is not reached.
- (`read-string-avail!/enable-break string [input-port start-k end-k]`) is like `read-string-avail!`, except that breaks are enabled during the read. The procedure provides a guarantee about the interaction of reading and breaks: if breaking is disabled when `read-string-avail!/enable-break` is called, and if the `exn:break` exception is raised as a result of the call, then no characters will have been read from *input-port*. See also §6.6.
- (`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. The *skip-k* argument indicates a number of characters in the input stream to skip before collecting characters to return; thus, in total, the next  $k + skip-k$  characters are inspected.

For most kinds of ports, inspecting  $k + skip-k$  characters requires  $k + skip-k$  bytes of memory overhead associated with the port, at least until the characters are read. No such overhead is required when peeking into a string port (see §11.1.4), a pipe port (see §11.1.3), or a custom port with a specific peek procedure (depending on how the peek procedure is implemented; see §11.1.6).

- (`peek-string-avail! string skip-k [input-port start-k end-k]`) is like `read-string-avail!`, but for peeking, and with a *skip-k* argument like `peek-string`. When skipping characters, `peek-string-avail!` blocks until finding the end-of-file or at least one character past the skipped characters.
- (`peek-string-avail!* string skip-k [input-port start-k end-k]`) is like `read-string-avail!*`, but for peeking, and with a *skip-k* argument like `peek-string`. Since this procedure never blocks, it may return before even *skip-k* characters are available from the port.
- (`peek-string-avail!/enable-break string skip-k [input-port start-k end-k]`) is the peeking version of `read-string-avail!/enable-break`, with a *skip-k* argument like `peek-string`.
- (`read-char-or-special input-port`) is the same as `read-char`, except that if the input port returns a non-character value (through a value-generating procedure in a custom port; see §11.1.6 for details), the non-character value is returned.
- (`peek-char-or-special input-port`) is the same as `peek-char`, except that if the input port returns a non-character value (through a value-generating procedure in a custom port; see §11.1.6 for details), the symbol 'special is returned.

### 11.2.2 Writing

In addition to the standard printing procedures, MzScheme provides `print`, which outputs values to a port by calling the port's print handler (see §11.2.5), plus the block-writing procedures such as `write-string-avail!`:

- (`print v [output-port]`) outputs *v* to *output-port*. The default value of *output-port* is the current output port.

The `print` procedure is used to print Scheme values in a context where a programmer expects to see a Scheme value. 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 §7.4.1.2).

- (`write-string-avail 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:application:mismatch` exception is raised if *start-k* or *end-k* is out-of-range for *string*.

The result is the number of characters written and flushed to *output-port*. The `write-string-avail` procedure returns without blocking after writing as many characters as it can immediately flush. It blocks only if no characters can be flushed immediately.

The `write-string-avail` procedure never drops characters; if `write-string-avail` successfully writes some characters and then encounters an error, it suppresses the error and returns the number of written characters. (The error will be triggered by future writes.) If an error is encountered before any characters have been written, an exception is raised.

- (`write-string-avail* string [output-port start-k end-k]`) is like `write-string-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 characters can be written immediately.
- (`write-string-avail/enable-break string [input-port start-k end-k]`) is like `write-string-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-string-avail/enable-break` is called, and if the `exn:break` exception is raised as a result of the call, then no characters will have been written to *output-port*. See also §6.6.

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` displays the next argument among the *vs*
  - `~s` or `~S` writes the next argument among the *vs*
  - `~v` or `~V` prints the next argument among the *vs*
  - `~e` or `~E` outputs the next argument among the *vs* using the current error value conversion handler (see §7.4.1.7) and current error printing width
  - `~c` or `~C` write-chars the next argument in *vs*; if the next argument is not a character, the `exn:application:mismatch` exception is raised
  - `~b` or `~B` prints the next argument among the *vs* in binary; if the next argument is not an exact number, the `exn:application:mismatch` exception is raised
  - `~o` or `~O` prints the next argument among the *vs* in octal; if the next argument is not an exact number, the `exn:application:mismatch` exception is raised
  - `~x` or `~X` prints the next argument among the *vs* in hexadecimal; if the next argument is not an exact number, the `exn:application:mismatch` 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:application:type` exception is raised. When the format string requires more additional arguments than are supplied, the `exn:application:fprintf:mismatch` exception is raised. When more additional arguments are supplied than are used by the format string, the `exn:application:mismatch` exception is raised.

For example,

```
(fprintf port "~a as a string is ~s.~n" '(3 4) "(3 4)")
```

prints this message to *port*:<sup>4</sup>

```
(3 4) as a string is "(3 4)".
```

followed by a newline.

### 11.2.3 Counting Positions, Lines, and Columns

MzScheme keeps track of the *position* in a port as the number of characters that have been read from any input port (independent of the read/write position, which is accessed or changed with `file-position`). In addition, MzScheme can track *line locations* and *column locations* when specifically enabled for a port via `port-count-lines!`. Position, line, and column locations for a port are used by `read-syntax` (see §12.2 for more information). Position, line, and column locations are numbered from 1.

- `(port-count-lines! input-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 a port.

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 the next multiple of 8.

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 tracking is disabled. Return-linefeed combinations are treated as a single character position only when line and column counting is enabled.

- `(port-next-location input-port)` returns three values: a positive exact integer or `#f` for the line number of the next read character, a positive exact integer or `#f` for the character's column, and a positive exact integer or `#f` for the character's position.

---

<sup>4</sup>Assuming that the current port display and write handlers are the default ones; see §11.2.5 for more information.

### 11.2.4 Customizing Read

Each input port has its own *port read handler*. This handler is invoked to read S-expressions or syntax objects from the port when the built-in `read` or `read-syntax` procedure is applied to the port. A port read handler must accept both a single argument or three 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.
- Three arguments are supplied when the port is used with `read-syntax`; the first argument is the port being read, the second argument is a value indicating the source, and the third argument is a list of three non-negative, exact integers (see §12.2 for more information). The return value is a syntax object that was read from the port.

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 §14.3).

### 11.2.5 Customizing Display, Write, and Print

Each output port has its own *port display handler*, *port write handler*, and *port print handler*. These handlers are invoked to output S-expressions 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*.
- `(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 §14.4). The default print handler calls the global port print handler (the value of the `global-port-print-handler` parameter; see §7.4.1.2); the default global port print handler is the same as the default write handler.

## 11.3 Filesystem Utilities

Additional filesystem utilities are in MzLib; see Chapter 13 of *PLT MzLib: Libraries Manual*.

### 11.3.1 Pathnames

File and directory paths are specified as strings. Since the syntax for pathnames can vary across platforms (e.g., under Unix, directories are separated by “/” while Mac OS Classic uses “:”), MzScheme provides tools for portably constructing and deconstructing pathnames.

Most MzScheme primitives that take pathnames perform an expansion on the pathname before using it. (Procedures that build pathnames or merely check the form of a pathname do not perform this expansion.) Under Unix and Mac OS X, a user directory specification using “~” is expanded.<sup>5</sup> Under Mac OS Classic, file and folder aliases are resolved to real pathnames.<sup>6</sup> Under Windows, multiple slashes are converted to single slashes (except at the beginning of a shared folder name), and a slash is inserted after the colon in a drive specification (if it is missing). In a Windows pathname, slash and backslash are always equivalent (and can be mixed together in the same pathname).

A pathname string cannot be empty or contain a null character (`#\nul`). When an empty string or a string containing a null character is provided as a pathname to any procedure except `absolute-path?`, `relative-path?`, `complete-path?`, or `normal-case-path`, the `exn:i/o:filesystem` exception is raised.

The pathname utilities are:

- (`build-path base-path sub-path ...`) creates a pathname given a base pathname and any number of sub-pathname extensions. If *base-path* is an absolute pathname, the result is an absolute pathname; if *base* is a relative pathname, the result is a relative pathname. Each *sub-path* must be either a relative pathname, a directory name, the symbol `'up` (indicating the relative parent directory), or the symbol `'same` (indicating the relative current directory). Under Windows, if *base-path* is a drive specification (with or without a trailing slash) the first *sub-path* can be an absolute (driveless) path. The last *sub-path* can be a filename.

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 pathname.

Under Mac OS Classic, if a *sub-path* argument does not begin with a colon, one is added automatically. This means that *sub-path* arguments are never interpreted as absolute paths under Mac OS Classic. For other platforms, if an absolute path is provided for any *sub-path*, then the `exn:i/o:filesystem` exception is raised. On all platforms, if *base-path* or *sub-path* is an illegal path string (e.g., it contains a null character), the `exn:i/o:filesystem` exception is raised.

The `build-path` procedure builds a pathname *without* checking the validity of the path or accessing the filesystem.

The following examples assume that the current directory is `/home/joeuser` for Unix examples and `My Disk:Joe's Files` for Mac OS Classic examples.

---

<sup>5</sup>Under Unix and Mac OS X, expansion does *not* convert multiple adjacent slashes to a single slash. However, extra slashes in a pathname are always ignored.

<sup>6</sup>Mac OS X follows the Unix behavior in its treatment of links, and Mac OS Classic aliases are simply zero-length files.

```

(define p1 (build-path (current-directory) "src" "scheme"))
  ; Unix: p1 ⇒ "/home/joeuser/src/scheme"
  ; Mac OS Classic: p1 ⇒ "My Disk:Joe's Files:src:scheme"
(define p2 (build-path 'up 'up "docs" "MzScheme"))
  ; Unix: p2 ⇒ ".././docs/MzScheme"
  ; Mac OS Classic: p2 ⇒ "::::docs:MzScheme"
(build-path p2 p1)
  ; Unix: raises exn:i/o:filesystem:path because p1 is absolute
  ; Mac OS Classic: ⇒ "::::docs:MzScheme:My Disk:Joe's Files:src:scheme"
(build-path p1 p2)
  ; Unix: ⇒ "/home/joeuser/src/scheme/././docs/MzScheme"
  ; Mac OS Classic: ⇒ "My Disk:Joe's Files:src:scheme::::docs:MzScheme"

```

- (`absolute-path? path`) returns `#t` if *path* is an absolute pathname, `#f` otherwise. If *path* is not a legal pathname string (e.g., it contains a null character), `#f` is returned. This procedure does not access the filesystem.
- (`relative-path? path`) returns `#t` if *path* is a relative pathname, `#f` otherwise. If *path* is not a legal pathname string (e.g., it contains a null character), `#f` is returned. This procedure does not access the filesystem.
- (`complete-path? path`) returns `#t` if *path* is a completely determined pathname (*not* relative to a directory or drive), `#f` otherwise. Note that under Windows, an absolute path can omit the drive specification, in which case the path is neither relative nor complete. If *path* is not a legal pathname string (e.g., it contains a null 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:i/o:filesystem` exception is raised. This procedure does not access the filesystem.
- (`resolve-path path`) expands *path* and returns a pathname that references the same file or directory as *path*. Under Unix and Mac OS X, if *path* is a soft link to another pathname, then the referenced pathname is returned (this may be a relative pathname 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 pathname might be a non-existent path.
- (`simplify-path path`) eliminates up-directory (“..” in Unix, Mac OS X, and Windows) and same-directory (“.”) indicators in *path*. If no indicators are in *path*, then *path* is returned. Otherwise, a complete path is returned; if *path* is relative, it is resolved with respect to the current directory. Up-directory indicators are dropped when they refer to the parent of a root directory. The filesystem might be accessed, but the source or expanded pathname might be a non-existent path. If *path* cannot be simplified due to a cycle of links, the `exn:i/o:filesystem` exception is raised (but a successfully simplified path may still involve a cycle of links if the cycle did not inhibit the simplification).
- (`normal-case-path string`) returns *string* with normalized case letters. Under Unix and Mac OS X, this procedure always returns the input path. Under Windows and Mac OS Classic, the resulting string uses only lowercase letters. Under Windows, all forward slashes (“/”) are converted to backward slashes (“\”), and trailing spaces are removed. This procedure does not access the filesystem or guarantee that the output string is a legal pathname (i.e., *string* and the result may contain a null character).
- (`split-path path`) deconstructs *path* into a smaller pathname and an immediate directory or file name. Three values are returned (see §2.2):

- *base* is either
  - \* a string pathname,
  - \* 'relative if *path* is an immediate relative directory or filename, or
  - \* #f if *path* is a root directory.
- *name* is either
  - \* a string directory name,
  - \* a string file name,
  - \* '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 specified a directory.

If *base* is #f, then *name* cannot be 'up or 'same. All strings returned for *base* and *name* are newly allocated. This procedure does not access the filesystem.

- (**find-executable-path** *program-sub-path related-sub-path*) finds a pathname for the executable *program-sub-path*, returning #f if the pathname cannot be found.

If *related-sub-path* is not #f, then it must be a relative path string, and the pathname 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.

If *program-sub-path* has a directory path, exists as a file or link to a file, and *related-sub-path* is not #f, **find-executable-path** determines whether *related-sub-path* exists relative to the directory of *program-sub-path*. If so, the complete path for *program-sub-path* is returned. Otherwise, if *program-sub-path* is a link to another file path, the destination directory of the link is checked for *related-sub-path*. Further links are inspected until *related-sub-path* is found or the end of the chain of links is reached.

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-paths*. 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.)

- (**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. (See below for information on the user's home directory in Windows and Mac OS Classic.)
  - 'pref-dir — the standard directory for storing the current user's preferences. Under Unix and Windows, this is the user's home directory. Under Mac OS Classic, it is the Preferences subdirectory of the System Folder. Under Mac OS X, it is the subdirectory **Library/Preferences** of the user's home directory. (See below for information on the user's home directory in Windows.)
  - '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. Under Unix and Mac OS X, the file is **.plt-prefs.ss**, and under Windows and Mac OS Classic, the file is **plt-prefs.ss**. See also **get-preference** in Chapter 13 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. (See below for information on the user's home directory in Windows and Mac OS Classic.)
- `'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 and Mac OS Classic: **mzschemerc.ss**
- `'sys-dir` — the directory containing the operating system for Windows or Mac OS Classic. Under Unix and Mac OS X, the result is `"/`.
- `'exec-file` — the pathname of the MzScheme executable as provided by the operating system for the current invocation.<sup>7</sup> Under Mac OS Classic, in the stand-alone MzScheme (or MrEd) application, it is always a complete path. In the stand-alone MzScheme application, this path is also bound initially to `program`.

Under Windows, the user's home directory is the one specified by the **HOMEDRIVE** and **HOMEPATH** environment variables. If those environment variables are not defined, or if the indicated directory does not exist, the directory containing the MzScheme executable is used as the home directory. Under Mac OS Classic, the user's "home directory" is the preferences directory.

- `(path-list-string->path-list string default-path-list)` parses a 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 and Mac OS Classic, 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. (The content of the list `default-path-list` is not inspected.)

### 11.3.2 Files

The file management utilities are:

- `(file-exists? path)` returns `#t` if a file (not a directory) `path` exists, `#f` otherwise. Unlike some other procedures that take a path argument, this procedure never raises the `exn:i/o:filesystem` exception.<sup>8</sup>
- `(link-exists? path)` returns `#t` if a link `path` exists (Unix, Mac OS X, and Mac OS Classic), `#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:i/o:filesystem` exception.
- `(delete-file path)` deletes the file with pathname `path` if it exists, returning void if a file was deleted successfully, otherwise the `exn:i/o: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 pathname `old-path` — if it exists — to the pathname `new-path`. If the file or directory is renamed successfully, void is returned, otherwise the `exn:i/o: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

<sup>7</sup>For MrEd, the executable path is the name of a MrEd executable.

<sup>8</sup>Under Windows, `file-exists?` reports `#t` for all variations of the special filenames (e.g., `"LPT1"`, `"x:/baddir/LPT1"`).

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*) returns the file or directory's last modification date as platform-specific seconds (see also §15.1).<sup>9</sup> If no file or directory *path* exists, the `exn:i/o: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. If no such file or directory exists, the `exn:i/o:filesystem` exception is raised.
- (**file-size** *path*) returns the (logical) size of the specified file. (Under Mac OS Classic, this is the sum of the data fork and resource fork sizes.) If no such file exists, the `exn:i/o: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:i/o:filesystem` exception is raised. If *dest-path* already exists, the copy will fail. File permissions are preserved in the copy. Under Mac OS Classic, 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. If the link is created successfully, void is returned, otherwise the `exn:i/o:filesystem` exception is raised. Under Windows and Mac OS Classic, the `exn:misc:unsupported` exception is raised always.

### 11.3.3 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 §7.4.1.1.
- (**current-drive**) returns the current drive name under Windows. For other platforms, the `exn:misc: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. Unlike other procedures that take a path argument, this procedure never raises the `exn:i/o:filesystem` exception.
- (**make-directory** *path*) creates a new directory with the pathname *path*. If the directory is created successfully, void is returned, otherwise the `exn:i/o:filesystem` exception is raised.
- (**delete-directory** *path*) deletes an existing directory with the pathname *path*. If the directory is created successfully, void is returned, otherwise the `exn:i/o:filesystem` exception is raised.
- (**rename-file-or-directory** *old-path* *new-path* *exists-ok?*), as described in the previous section, renames directories.

---

<sup>9</sup>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-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.
- (`filesystem-root-list`) returns a list of all current root directories.

## 11.4 Networking

MzScheme provides a minimal collection of TCP-based communication procedures. 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]`) 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. The default for `reuse?` is `#f`.

If `hostname-string` is `#f` (the default), then the listener accepts connections to all of the listening machine’s IP addresses. Otherwise, the listener accepts connections only at the IP address associated with the given name. For example, providing “127.0.0.1” as `hostname-string` typically creates a listener that accepts only connections to “127.0.0.1” 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:i/o:tcp` exception is raised.

- (`tcp-connect hostname-string [port-k]`) attempts to connect as a client to a listening server. The `hostname-string` argument is the server host’s internet address name<sup>10</sup> (e.g., “www.plt-scheme.org”), and `port-k` (an exact integer between 1 and 65535) is the port where the server is listening.

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).

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.

If a connection cannot be established by `tcp-connect`, the `exn:i/o:tcp` exception is raised.

- (`tcp-connect/enable-break hostname-string [port-k]`) is like `tcp-connect`, but breaking is enabled (see §6.6) while trying to connect. If breaking is disabled when `tcp-connect/enable-break` is called, then either ports are returned or `exn:break` exception is raised, but not both.

<sup>10</sup>The name “localhost” generally specifies the local machine.

- (`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).

Both of the returned ports must be closed to terminate the connection. When both ports are still open, closing the output port with `close-output-port` sends a TCP close to the client (which is seen as an end-of-file if the client 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.

If a connection cannot be accepted by `tcp-accept`, or if the listener has been closed, the `exn:i/o:tcp` 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:i/o:tcp` exception is raised.

- (`tcp-accept/enable-break tcp-listener`) is like `tcp-accept`, but breaking is enabled (see §6.6) while trying to accept a connection. If breaking is disabled when `tcp-accept/enable-break` is called, then either ports are returned or `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:i/o:tcp` 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-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.<sup>11</sup>
- (`tcp-addresses tcp-port`) returns two strings. The first string is the internet address for the local machine as viewed by the given TCP port's connection.<sup>12</sup> The second string is the internet address for the other end of the connection.

If the given port has been closed, the `exn:i/o:tcp` exception is raised.

<sup>11</sup>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.

<sup>12</sup>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.

## 12. Syntax and Macros

---

MzScheme supports the *R<sup>5</sup>RS* **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.<sup>1</sup>

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 33 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.

---

<sup>1</sup>In general, modules and for-syntax imports create a hierarchy of run times and expansion times. See §12.3.4 for more information.

To mesh gracefully with modules, literal identifiers are compared with `module-identifier=?`, which is equivalent to  $R^5RS$  in the absence of modules; see §12.3.1 for more information on identifier syntax comparisons.

## 12.2 Syntax Objects

(`read-syntax source-name-v [input-port offset-list]`) 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 should generally be a string path for the source file. The *offset-list* argument is a list of three non-negative, exact integers; the first integer is the line offset for the source line of the returned syntax object, the second is the column offset (for data read on the first line), and the third is the position offset. The default value for *offset-list* is `(list 0 0 0)`. See §14.3 for more information about `read` and `read-syntax`, see §11.2.3 for information about port locations, and see §12.6.2 for information on the property attached to a syntax object by `read-syntax`.

The `eval`, `compile`, `expand`, `expand-once`, and `expand-to-top-form` procedures work on syntax objects. 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.

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 when a transformer is called, the provided syntax object has lexical information. In addition, the syntax object produced by `expand`, `expand-once`, or `expand-to-top-form` has lexical information that influences future expansion and compilation of the syntax object.

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 10, 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 normal expressions; `quote-syntax` is more typically used within a transformer expression. Unlike `quote`, `quote-syntax` fails to compile (i.e., it loops forever) when *datum* is cyclic.

The `syntax-object->datum` procedure strips away location and lexical 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 and the lexical information of another syntax object. 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:syntax` exception is raised.

A *pattern* is nearly the same as a **syntax-rules** pattern (see *R<sup>5</sup>RS*), 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.

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.

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 that the introduced variable has the same lexical context as *if-it* in 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-identifiers* 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 is applied to two arguments. The first argument will be an identifier from *stx-expr*, and the second argument will be an identifier from a *syntax-clause* pattern that is **module-identifier=?** to one of the *literal-identifiers*.

### 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)
```

A vector of the *patterns* is matched against a vector of the *stx-expr* values, and all pattern identifiers are bound in *expr*. If the result of a *stx-expr* does not match its *pattern*, the **exn: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.<sup>2</sup>

### 12.2.1.2 QUASIQUOTING TEMPLATES

The **quasisyntax** form is like **syntax**, except with a 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 way as for the right-hand sides of **with-syntax**.) Nested **quasisyntaxes** introduce quasiquoting layers in the same way as nested **quasiquotes**.

Also analogous to **quote** and **quasiquote**, the reader converts **#'** to **syntax**, **#'** to **quasisyntax**, **#**, to **unsyntax**, and **#,@** to **unsyntax-splicing**. See also §14.3.

Example:

```
(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:

```
(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)
```

---

<sup>2</sup>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 ([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 §14.6.

`(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 §11.2.3 and §14.6.

`(syntax-column stx)` returns the column number (positive 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 §11.2.3 and §14.6.

`(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 §11.2.3 and §14.6.

`(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 §14.6.

`(syntax-original? stx)` returns `#t` if `stx` has the property that `read-syntax` attaches to the syntax objects that it generates (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 §12.6.4) 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)	<i>stx</i> , where (syntax-e <i>stx</i> ) is equivalent to (list <i>a-stx b-stx</i> ) and (syntax-e <i>a-stx</i> ) is equivalent to 'a and (syntax-e <i>b-stx</i> ) is equivalent to 'b
(a . (b))	<i>stx</i> , where (syntax-e <i>stx</i> ) is equivalent to (cons <i>a-stx sb-stx</i> ) and (syntax-e <i>a-stx</i> ) is equivalent to 'a and (syntax-e <i>sb-stx</i> ) is equivalent to (list b-stx) and (syntax-e <i>b-stx</i> ) is equivalent to 'b

(syntax->list *stx*) returns a 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 *ctx-stx v [src-stx-or-list prop-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 *ctx-stx* and the source-location information of *src-stx-or-list*; if the resulting syntax object has no properties, then it is given the properties of *prop-stx*. Any of *ctx-stx*, *src-stx-or-list*, or *prop-stx* can be #f, in which case the resulting syntax has no lexical context, source information, and/or new properties. 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 positive, 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:application;mismatch` exception is raised. Graph structure is preserved by the conversion, 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` 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). Generated identifiers can be used for definitions in a module top level, but §14.6 describes some limitations with compiled modules.

## 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 variables that are bound by **lambda**, **let**, **letrec**, etc., at the syntax object's introduction site, the **required** variables 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 variable, if the identifier is not within a module).

The `identifier-binding` procedure (described in §12.3.2) reports an identifier's 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 they obeys lexical and module scopes. In addition, an identifier's lexical information encompasses a second dimension, which distinguishes the environment of normal expressions and 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. 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).

`(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 four items when *id-stx* is bound in its context to a module-defined variable: (`list` *source-mod* *source-id* *nominal-source-mod* *nominal-source-id*).
  - *source-mod* is a module path index or symbol (see §12.6.4) that indicates the defining module.
  - *source-id* is a symbol for the variable's name at its definition site in the source module (as opposed to the local name returned by `syntax-object->datum`).
  - *nominal-source-mod* is a module path index or symbol (see §12.6.4) that indicates the module **required** 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 variable'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.
- The result is #f if *id-stx* is not bound (i.e., bound 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-binding-export-position` *id-stx*) returns either #f or a number. It returns a number only when `identifier-binding` returns a list, and only 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-position`, except that the reported information is for 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-syntax (let1 stx)
  (syntax-case stx ()
    [(- x v b)
     (begin
      (set! count (add1 count)) ; DOESN'T WORK
      (syntax (let ([x v] b)))]))
  (let1 x 2 (add1 x))
```

The variable *count* is 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.

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.<sup>3</sup> To extend this environment, a programmer must place definitions into a module, and then use **require-for-syntax** to import the definitions into the top-level transformer environment.

Like a top-level definition, a top-level **require** expression imports 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. 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.

When a lexical variable 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. 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 a variable 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

---

<sup>3</sup>In contrast, a namespace created by (**scheme-report-environment** 5) imports only **syntax-rules** into the transformer environment.

```
(module m mzscheme
  (define-syntax macro
    (let ([id (lambda (x) x)]
      (lambda (stx)
        (id (syntax (printf "hi~n"))))))))
```

The **define-syntaxes** form (see §12.4) is useful for defining multiple macros that share helper functions. See also **define-syntax-set** in Chapter 12 of *PLT MzLib: Libraries Manual*.

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 causes the referenced module to be executed at expansion time, instead of (or possibly in addition to) run time for the module being expanded. 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.

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 et mzscheme
  (printf "ET here~n")
  (define mx (lambda () 700))
  (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) (datum->syntax-object (mx) stx stx)))
  (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 "RT here", 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.

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.

## 12.4 Binding Multiple and Fluid 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 (variable ...) expr)
```

```
(let-syntaxes (((variable ...) expr)
              ...
              (variable ...) expr))
```

```
(letrec-syntaxes (((variable ...) expr)
                 ...
                 (variable ...) expr))
```

MzScheme also provides a **letrec-syntaxes+values** form for binding both values and syntax in a single, mutually recursive scope:

```
(letrec-syntaxes+values (((variable ...) expr) ...)
                       (((variable ...) expr) ...)
                       (variable ...) ...))
```

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
```

Finally, MzScheme provides **fluid-let-syntax**, which is roughly analogous to **fluid-let**.

```
(fluid-let-syntax ((variable expr)
                  ... )
  body-expr ...1)
```

Instead of introducing a new binding, **fluid-let-syntax** alters the mapping for each *variable* while expanding the *body-exprs*. Each *variable* need not have been mapped to expansion-time values before, and the re-mapping is not restricted to instances of *variable* in the *body-exprs*; it applies when resolving any identifier that is **bound-identifier=?** to *variable* while the *body-exprs* are expanded. However, **fluid-let-syntax** does not mutate any state that is visible to other expansions (that are possibly running in other threads).

## 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. Similarly, an expression

```
identifier
```

where *identifier* has no binding other than a top-level or local module 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 binds **#%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 addition, **#%module-begin** is used as a transformer for a module body. The **mzscheme** module binds **#%module-begin** to a form that inserts a for-syntax import of **mzscheme** for 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`.

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 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: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`).

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.

If a transformer expression produces a non-procedure value, the value is associated to 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 33 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/sig** 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:application:mismatch` exception is raised.
- (`syntax-local-name`) returns an inferred name for the expression position being transformed, or `#f`; see also §6.2.4.

- (`syntax-local-context`) returns either 'expression, 'top-level, 'internal-define, or 'module, indicating whether the expression is being expanded for a (non-definition) expression position, a top-level position, a (potential) internal-definition position, or a module top-level position, respectively.

A transformer can also expand or partially expand subexpressions from its input syntax object:

- (`local-expand stx context-symbol stop-id-stx-list`) expands `stx` in the lexical context of the expression currently being expanded. The `context-symbol` argument is used as the result of `syntax-local-context` for immediate expansions; it must be one of the legal return values for `syntax-local-context`. 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.

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. Double 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`:

- (`syntax-local-introduce stx`) produces a syntax object that is like `stx`, except that a mark for the current expansion is added (possibly cancelling an existing mark in parts of `stx`).

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 a 'bound-in-source property that specifies bound–binding identifier pairs in the source program that do not appear as bound and binding identifiers 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 variable 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 (list-ref expr pos)) ; the expansion result
              'bound-in-source
              (cons
               (syntax-local-introduce (car ids))
```

```
(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 *bs* exist in the source program.

### 12.6.1 Expanding Expressions to Primitive Syntax

(`expand stx-or-sexp`) expands all non-primitive syntax in *stx-or-sexp*, and returns a syntax object for the expanded expression. See below for the grammar of fully expanded expressions.

(`expand-once stx-or-sexp`) partially expands syntax in the *stx-or-sexp* 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`.

(`expand-to-top-form stx-or-sexp`) partially expands syntax in *stx-or-sexp* 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.

The possible shapes of a fully expanded expression are defined by *top-level-expr*:

```
top-level-expr is one of
  general-top-level-expr
  (module identifier name (#%plain-module-begin module-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 (variable ...) expr)
  (begin top-level-expr ...)
  (require require-spec ...)
  (require-for-syntax require-spec ...)
```

```
expr is one of
  variable
  (lambda formals expr ...1)
  (case-lambda (formals expr ...1) ...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 ...)
(#%datum . datum)
(#%top . variable)

```

where *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**, or **letrec-values**, or it refers to an imported variable. (In other words, a *variable* not wrapped by **#%top** never refers to a top-level variable, and it never refers to a non-imported variable that is defined at the top-level of a module.)

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*.

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 adjust a property on 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** 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**)).

When **read-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 §12.6.4 for information about properties generated by the expansion of a module declaration. See §3.10.1 and §6.2.4 for information about properties recognized when compiling a procedure. See §14.6 for information on properties and byte codes.

### 12.6.3 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.

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 17 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/sig** (see Chapter 33 of *PLT MzLib: Libraries Manual*) causes the **unit/sig** 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 five items:

- an identifier that is bound to the structure type's descriptor, or #f if it none is known;
- an identifier that is bound to the structure type's constructor, or #f if it none is known;
- an identifier that is bound to the structure type's predicate, or #f if 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, optionally with #f as the list's last element. The meaning of #f and the order are the same as for the accessor identifiers.

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/sig** 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.

### 12.6.4 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. The information usually identifies modules through a *module path index*, which is a semi-interned<sup>4</sup> opaque value that encodes a relative module path (see §5.4) and another index to which it is relative.

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.

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 the identifier's source module will be reported using an index that contains the **required** module path and the “self” index.

- `(module-path-index? v)` returns `#t` if *v* is a module path index, `#f` otherwise.
- `(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”.
- `(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 an index, symbol, or `#f`.

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-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, 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.

<sup>4</sup>Multiple references to the same relative module tend to use the same index value, but not always.

- `'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 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 §14.6.

`(module-compiled-name compiled-module-code)` takes a module declaration in compiled form (see §14.6) and returns a symbol for the module's declared name.

`(module-compiled-imports compiled-module-code)` takes a module declaration in compiled form (see §14.6) and returns two values: an immutable list of module path indices (and symbols) for the module's explicit imports, and an immutable list of module path indices (and symbols) for the module's explicit for-syntax imports.

## 13. Memory Management

---

### 13.1 Weak Boxes

A *weak box* is similar to a normal box (see §3.9), but when the automatic memory manager can prove that the content value of a weak box is only reachable via weak boxes, the content of the weak box is replaced with `#f`.

- `(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 weak boxes, then `#f` is returned.
- `(weak-box? v)` returns `#t` if *v* is a weak box, `#f` otherwise.

### 13.2 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 will executors, weak boxes, weak hash table keys, and custodians. 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 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`.
- `(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 unreachable once again 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.

If the content value of a weak box (and/or a key in a weak hash table) is registered with a will executor, the weak box's content is not changed to #f (and/or the weak hash table entry is not removed) until all wills have been executed for the value and the value has been proven unreachable again.

### 13.3 Garbage Collection

(`collect-garbage`) forces an immediate garbage collection. Since MzScheme uses a “conservative” garbage collector, some effectively unreachable data may remain uncollected (because the collector cannot prove that it is unreachable). This 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. Unless MzScheme is compiled with special support for memory accounting, the estimate is the same (i.e., all memory) for any individual custodian.

(`dump-memory-stats`) dumps information about memory usage to the (low-level) standard output port.

## 14. Support Facilities

---

### 14.1 Eval and Load

(`eval expr`) evaluates the S-expression *expr* in the current namespace.<sup>1</sup> (See §8 and §7.4.1.5 for more information about namespaces.)

(`load file-path`) evaluates each expression in the specified file using `eval`.<sup>2</sup> 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 pathname, then it is resolved to an absolute pathname 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 §7.4.1.6) is set to the absolute pathname 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 pathname, it is resolved to an absolute pathname 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 8 of *PLT MzLib: Libraries Manual*) and `.so` (Unix and Mac OS Classic), `.dll` (Windows), or `.dylib` (Mac OS X), files.<sup>3</sup> The check for a compiled file occurs whenever *file-path* ends with a dotted extension of three characters or less (e.g., `.ss` or `.scm`) and when a **compiled** subdirectory exists in the same directory as *file-path*. A `.zo` version of the file is loaded if it exists directly in the **compiled** subdirectory. An `.so` or `.dll` version of the file is loaded if it exists within a **native** subdirectory of the **compiled** directory, in a 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` or `.dll` files are available, the `.so` or `.dll` file is used.

Multiple files can be combined into a single `.so` or `.dll` file by creating a special dynamic extension `_loader.so` or `_loader.dll`. When such an extension is present where a normal `.so` or `.dll` 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`, or `.dll` file is loaded (or while a thunk returned by `_loader` is invoked), the current

---

<sup>1</sup>The `eval` procedure actually calls the current evaluation handler (see §7.4.1.5) with *e* to evaluate the expression.

<sup>2</sup>The `load` procedure actually just sets the current `load-relative` directory and calls the current load handler (see §7.4.1.6) with *file-path* to load the file. The description of `load` here is actually a description of the default load handler.

<sup>3</sup>The `load/use-compiled` procedure actually just calls the current `load/use-compiled` handler (see §7.4.1.6). The default handler, in turn, calls the `load` or `load-extension` handler, depending on the type of file that is loaded.

`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. When `read-eval-print-loop` starts, it installs a new error escape procedure (see §6.7) that does not exit the `read-eval-print` 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 §7.4.1):

```
(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 §7.4.1.9). 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 Input Parsing

MzScheme's input parser obeys the following non-standard rules:

- 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 §7.4.1.3). Similarly, the parsing of curly braces is controlled with the `read-curly-brace-as-paren` parameter. 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:read` exception is raised.
- Boxed constants can be created using `#&`. The S-expression following `#&` is treated as a quoted constant and put into the new box. (Spaces following the `#&` are ignored.) Box reading is controlled with the `read-accept-box` boolean parameter (see §7.4.1.3). Box reading is enabled by default. When box reading is disabled and `#&` is provided as input, the `exn: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)

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:read` exception is raised.

Character constants can also be specified through direct ASCII values in octal notation: `#n1n2n3` 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<sub>8</sub> to 377<sub>8</sub>.

- 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)
  - `\o`, `\oo`, or `\ooo`: ASCII 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`.
  - `\xh` or `\xhh`: ASCII 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 `\xhh` form takes precedence over the `\xh` form.

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.

- 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 *datums* must appear before the first dot and after the last dot:

$$(left-datum \dots^1 . first-datum . right-datum \dots^1)$$

The resulting list consists of the *datum* between the dots, followed by the remaining *datums* in order:

$$(first-datum left-datum \dots^1 right-datum \dots^1)$$

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 *R<sup>5</sup>RS*. When input is scanned for tokens, the following characters delimit an identifier:

" , ' ( ) [ ] { } space tab return newline page vtab

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 (“\”) or 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 (the default).

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)**.
- **(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 §7.4.1.3). Vertical bar quotes are enabled by default. Quoting backslashes cannot be disabled.

- By default, symbols are read case-insensitively (i.e., uppercase characters are downcased). 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 §7.4.1.3) is set to **#t**, then case is preserved when reading symbols. The default is **#f**, and it is set to **#f** while loading a module (see §5.8). A **#cs** or **#ci** prefix overrides the parameter setting, as does backslash or vertical-bar quoting.

Case conversions are *not* sensitive to the current locale (see §7.4.1.11).

- S-expressions with shared structure are expressed using **#n=** and **#n#**, where *n* is a decimal integer. See §14.5.
- 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 §14.6.
- Multi-line comments are started with `#|` and terminated with `|#`. Comments of this form can be nested arbitrarily.
- If the first line of a loaded 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.)

Reading from a custom port can produce arbitrary values generated by the port; see §11.1.6 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:read:non-char` exception is raised.

## 14.4 Output Printing

MzScheme’s printer obeys the following non-standard rules:

- A vector can be printed by `write` and `print` using the shorthand described in §14.3, 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 §7.4.1.4). Shorthand vector printing is enabled by default.
- Boxes (see §3.9) can be printed with the `#\&` notation (see §14.3). When box printing is disabled, all boxes are printed as  `#<box>`. Box printing is controlled with the `print-box` boolean parameter (see §7.4.1.4). Box printing is enabled by default.
- Structures (see Chapter 4) can be printed using vector notation. In the vector, 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 as  `#<struct:s>`. Structure printing is controlled with the `print-struct` boolean parameter (see §7.4.1.4). Structure printing is disabled 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 §7.4.1.3), 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. See §14.3 for more information about symbol parsing. Symbols `display` without escaping or quoting special characters.
- Characters with the special names described in §14.3 `write` using the same name. (Some characters have multiple names; the `#\newline` and `#\nul` names are used instead of `#\linefeed` and `#\null`). Other “printable” characters `write` as  `#\` followed by the single-byte character value, and “unprintable” characters are written in octal notation; unprintable-character detection depends on locale sensitivity (see §7.4.1.11), and when the current locale is disabled, a character whose `char->integer` value is greater than 127 is always treated as unprintable. All characters `display` as the single-byte character value.
- Strings containing “unprintable” characters (see above) `write` using the escape sequences described in §14.3. All strings `display` as their literal character sequences.

- S-expressions with shared structure can be printed using `#n=` and `#n#`, where  $n$  is a decimal integer. See §14.5.

## 14.5 Data Sharing in Input and Output

MzScheme can read and print *graphs*, S-expressions with shared structure (e.g., a cycle). 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 S-expression describes 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 S-expression 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 §7.4.1.4). 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:read` exception is raised.

If the  $n$  in a `#n=` form or a `#n#` form contains more than eight digits, the `exn:read` exception is raised. If a `#n#` form is not preceded by a `#n=` form using the same  $n$ , the `exn:read` exception is raised. If two `#n=` forms are in the same expression for the same  $n$ , the `exn:read` exception is raised.

## 14.6 Compilation

Normally, compilation happens automatically: when an S-expression 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 S-expression code and compiling it, so compilation can be used to speed up program loading. The MzLib procedure `compile-file` (see Chapter 8 of *PLT MzLib: Libraries Manual*) is sufficient for most compilation purposes.

- `(compile expr)` compiles `expr`, where `expr` is any S-expression that can be passed to `eval`. The result is a compiled expression Scheme value. This value is passed to `eval` to evaluate the compiled expression.
- `(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).

Never ask MzScheme to evaluate an expression starting with `#~` unless `compile` generated the expression. To keep users from accidentally specifying bad instructions, `read` will not accept expressions beginning with `#~` unless it is specifically enabled through the `read-accept-compiled` boolean parameter (see §7.4.1.3). When the default load handler is used to load a file, compiled expression reading is automatically (temporarily) enabled as each expression is read.

A compiled code object may contain uninterned symbols (see §3.6) that were created by `gensym`, `string->uninterned-symbol`, and `generate-temporaries`. 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.

Special problems arise when an uninterned symbol is used to construct an identifier for a module's exported variable. Since a module and its importers are typically compiled and written as separate expressions, different uninterned symbols are generated for identifiers when the different modules are loaded. MzScheme corrects for the problem by recording identifier-position pairs for each import, and then resolving imports at load time by checking the printed representation of a name at the expected position in the exporting module. (In principle, the position is sufficient, but MzScheme checks the printed form of the name to help avoid confusion due to mis-ordered compilation sequences.)

## 14.7 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 §7.4.1.6) is set while the extension is loaded. The `load-relative-extension` procedure is like `load-extension`, but it loads an extension with a pathname 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 §7.4.1.6). The result of calling `load-extension` is determined by the extension. If the extension cannot be loaded, the `exn:i/o:filesystem` exception is raised. The `detail` field of the exception is `'wrong-version` if the load fails because the extension has the wrong version.

## 14.8 Saving and Restoring Program Images

An *image* is a memory dump from a running MzScheme program that can be later restored (one or more times) to continue running the program from the point of the dump. Images are only supported for statically-linked Unix versions of MzScheme (and MrEd). There are a few special restrictions on images:

- All files and TCP connections must be closed when an image is created.
- No dynamic extensions can be loaded before an image is created.
- No operating system subprocesses can be active when an image is created.

(`write-image-to-file` *file-path* [*cont-proc*]) copies the state of the entire MzScheme process<sup>4</sup> to *file-path*, replacing *file-path* if it already exists. If images are not supported, the `exn:misc:unsupported` exception is raised. If *cont-proc* is `#f`, then the MzScheme or MrEd process exits immediately after creating the image. Otherwise, *cont-proc* must be a procedure of no arguments, and the return value(s) of the call to `write-image-to-file` is (*cont-proc*). The default value for *cont-proc* is `void`.

(`read-image-from-file` *file-path* *arg-vector*) restores the image saved to *file-path*. Once the image is restored, execution of the original program continues with the return from `write-image-to-file`; the return value in the restored program is the a vector of strings *arg-vector*. A successful call to `read-image-from-file` never returns because the restored program is overlaid over the current program. The vector *arg-vector* must contain no more than 20 strings, and the total length of the strings must be no more than 2048 characters.

If an error is encountered while reading or writing an image, the `exn:i/o:filesystem` exception is raised or `exn:misc` exception is raised. Certain errors during `read-image-from-file` are unrecoverable; in case of such errors, MzScheme prints an error message and exits immediately.

An image can also be restored by starting the stand-alone version of MzScheme or MrEd with the `--restore` flag followed by the image filename. The return value from `write-image-to-file` in the restored program is a vector of strings that are the extra arguments provided on the command line after the image filename (if any).

---

<sup>4</sup>The set of environment variables is not saved. When an image is restored, the environment variables of the restoring program are transferred into the restored program.

## 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 §11.3.2).

(`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, which 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 <sup>1</sup>

All fields of the `date` structure type are accessible by all inspectors (see §4.5).

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.

See also Chapter 9 of *PLT MzLib: Libraries Manual* for additional date utilities.

#### 15.1.2 Machine Time

(`current-milliseconds`) returns the current “time” in fixnum milliseconds. 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 is only strictly increasing for a limited (though reasonably long) time.

---

<sup>1</sup>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-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 is only strictly increasing for a limited (though reasonably long) time.

(`current-gc-milliseconds`) returns the amount of processor time that has been consumed by MzScheme's garbage collection so far in fixnum milliseconds. 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 second 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-strings` are command-line arguments for the program.

Under Mac OS Classic, arguments are not supported. However, a single `arg-string` argument triggers a hack specific to Mac OS Classic; in that case, `command-path` must be “by id” and the `arg-string` argument is a four-character string specifying an application ID, which is used to find the application instead of an executable path.

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-strings` 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.<sup>2</sup> If `'exact` is provided on a non-Windows platform, the `exn:application:mismatch` 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`.

<sup>2</sup>For information on the Windows command-line conventions, search for “command line parsing” at <http://msdn.microsoft.com/>.

The `subprocess` procedure returns four values:

- a subprocess value representing the create 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 placed into the management of the current custodian (see §9.2). The `exn:misc` 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:misc` 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. Under Mac OS Classic, the reported ID is always 0.
- `(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 22 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:misc: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 an 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 — Hides the window and activates another window.
- 'SW\_MAXIMIZE — Maximizes the window.
- 'SW\_MINIMIZE — Minimizes the window and activates the next top-level window in the z-order.
- '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 — Activates the window and displays it in its current size and position.
- 'SW\_SHOWDEFAULT — Uses a default.
- 'SW\_SHOWMAXIMIZED — Activates the window and displays it as a maximized window.
- 'SW\_SHOWMINIMIZED — Activates the window and displays it as a minimized window.
- 'SW\_SHOWMINNOACTIVE — Displays the window as a minimized window. The active window remains active.
- 'SW\_SHOWNA — Displays the window in its current state. The active window remains active.
- 'SW\_SHOWNOACTIVATE — Displays a window in its most recent size and position. The active window remains active.
- '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:misc` exception is raised. If the action succeeds, the result is either a subprocess value (see §15.2) or `#f` if the operating system did not return a process handle on success. If a subprocess value is returned, its process ID is 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.

Although Mac OS Classic does not have operating system environment variables, `getenv` returns values installed with `putenv` (which always succeeds) in the same MzScheme session. When MzScheme is started, an initial environment is read from an **Environment** file in the current directory if it exists. An **Environment** file must contain a sequence of two-item lists where the *name* string is the first item in the list and the *value* string is the second. For example, an **Environment** file might contain the following:

```
("PLTCOLLECTS" ";My Disk:Extra Collections:")
("USER" "joeuser")
```

## 15.5 Runtime Information

(**system-type** [*details?*]) returns a symbol indicating the type of the operating system for a running MzScheme if *details?* is #f or not provided. The possible values are:

- 'unix
- 'windows
- 'macos
- 'macosx
- 'oskit

Future ports of MzScheme will expand this list of system types. If *details?* is not #f, then the result is a string, which contains further details about the operating system and the current machine in a platform-specific format.

(**system-library-subpath**) returns a relative directory pathname string. This string can be used to build pathnames to system-specific files. For example, when MzScheme is running under Solaris on a Sparc architecture, the subpath is "sparc-solaris", while the subpath for Windows on an Intel architecture is "win32\\i386".

(**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.

## 16. Library Collections and MzLib

---

A *library* is **module** declaration for use by multiple programs. MzScheme provides an 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.<sup>1</sup> 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* ...). Such a path access 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 pathname. 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 §7.4.1.6). 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 as follows:<sup>2</sup>

```
(current-library-collection-paths
 (path-list-string->path-list
  (or (getenv "PLTCOLLECTS") "")
  (or (ormap (lambda (p) (and p (directory-exists? p) (list p)))
      (list (let ([v (getenv "PLTHOME")])
              (and v (build-path v "collects")))
            (find-executable-path program "collects")
            (find-executable-path program (build-path 'up "collects"))
            (find-executable-path program (build-path 'up 'up "collects")))))
      null)))
```

where *program* is the name used to start MzScheme (always a complete path for Mac OS Classic). See also §11.3.1 for information about **path-list-string->path-list** and **find-executable-path**.

---

<sup>1</sup>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**.

<sup>2</sup>MrEd initializes the **current-library-collection-paths** parameter in the same way.

(`collection-path collection ...1`) returns the path containing the libraries of *collection*; if the collection is not found, the `exn:i/o: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 flags. Under Mac OS Classic, a user can specify command-line flags by holding down the Command key while starting MzScheme, which provides a dialog for entering the command line. Dragging files onto the MzScheme icon in Mac OS Classic is equivalent to providing each file's name on the command line preceded by `-f`, so each file is loaded after MzScheme starts. When files are dragged onto MzScheme with the Command key pressed, the command line specified in the dialog is appended to the implicit command-line for loading the files.

MzScheme accepts the following flags:

- **Startup file and expression flags:**

- `-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.
- `-r file` or `--script file` : Use this flag 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 flag. The `-r` or `--script` flag 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` flag is a shorthand for `-dmv-`.
- `-u file` or `--require-script file` : Same as `-r file` or `--script file`, except that *file* is **required** instead of loaded. The `-u` or `--require-script` flag 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 flag 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.
- `-M` or `--main` : Calls the function bound to *main* in the top-level environment, providing all leftover arguments (the ones installed in `current-command-line-arguments`) to *main* as a single list of immutable strings. The *main* function is called only if no previous evaluations or loads resulted in an uncaught exception.

- **Initialization flags:**

- `-x` or `--no-lib-path` : Suppresses the initialization of `current-library-collection-paths` (as described in Chapter 16).
- `-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.

- **Language setting flags:**

- `-g` or `--case-sens` : Creates an initial namespace where identifiers and symbols are case-sensitive.
- `-s` or `--set-undef` : Creates an initial namespace where `set!` will successfully mutate an undefined global variable (implicitly defining it).

- **Miscellaneous flags:**

- `--` : No argument following this flag is used as a flag.
- `-m` or `--mute-banner` : Suppresses the startup banner text.
- `-v` or `--version` : Suppresses the `read-eval-print` loop.
- `-h` or `--help` : Shows information about MzScheme's command-line flags and then exits; ignoring other flags.
- `-p` or `--persistent` : Catches the SIGDANGER (low page space) signal and ignores it (AIX only).
- `-Rfile` or `--restore file` : Restores a saved image (see §14.8). Extra arguments after `file` are returned as a vector of strings to the continuation of the `write-image-to-file` call that created the image.

## 17.1 Flag Conventions

Extra arguments following the last flag are available from the `current-command-line-arguments` parameter (see §7.4.1.6) as an immutable vector of immutable strings. The name used to start MzScheme is available from the `find-system-path` procedure (see §11.3.1) 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 an immutable string.

Multiple single-letter flags (the ones preceded by a single dash) can be collapsed into a single flag by concatenating the letters, as long as the first flag is not `--`. The arguments for each flag are placed after the collapsed flags (in the order of the flags). For example,

```
-vfme file expr
```

and

```
-v -f file -m -e expr
```

are equivalent. If a collapsed `--` appears before other collapsed flags, 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

```
-qAerC '(require (lib "init.ss" "script-lang" "dialect"))'
```

so that the first actual command-line argument is the name of a file to load into a namespace that will be initialized by the `dialect`-specific `init.ss` library. The loaded file should define `main`, which is called with the remaining arguments in a list.

### 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` flag is specified.

Unless the `-q` or `--no-init-file` flag 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. If an error 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` flag 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 **requires** the script file instead of loading 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 6 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))
```

# Index

+inf.0, 10  
+nan.0, 10  
,, 102  
, 102  
-- , 116  
--Load, 115  
--Load-cd, 115  
--Require, 115  
--awk, 115  
--case-sens, 116  
--eval, 115  
--help, 116  
--load, 115  
--load-cd, 115  
--main, 115  
--mute-banner, 116  
--mzlib, 115  
--no-argv, 115  
--no-init-file, 115  
--no-lib-path, 115  
--persistent, 116  
--require, 115  
--require-script, 115  
--restore, 107, 116  
--script, 115  
--script-cd, 115  
--set-undef, 116  
--version, 116  
-A, 115  
-D, 115  
-F, 115  
-L, 115  
-M, 115  
-R, 116  
-T, 115  
-d, 115  
-e, 115  
-f, 115  
-g, 116  
-h, 116  
-i, 115  
-inf.0, 10  
-k, 115  
-l, 115  
-m, 116  
-nan.0, 10  
-p, 116  
-q, 115  
-r, 115  
-s, 116  
-t, 115  
-u, 115  
-v, 116  
-w, 115  
-x, 115  
., 102  
... , 77  
.mzschemerc, 73  
=>, 4  
[ ], 44  
#!, 104, 118  
#', 101  
#, , 101  
#, @, 101  
#\backspace, 102  
#\linefeed, 102  
#\newline, 102  
#\nul, 102  
#\null, 102  
#\page, 102  
#\return, 102  
#\rubout, 102  
#\space, 102  
#\tab, 102  
#\vtab, 102  
#<undefined>, 9  
#<void>, 9  
#%, 103  
#%app, 89  
#%datum, 89  
#%module-begin, 89  
#%top, 89  
#&, 101  
#', 101  
#ci, 103  
#cs, 103  
#\~, 103  
#n=, 103, 105  
#n#, 103, 105  
\, 102  
\a, 102  
\b, 102  
\e, 102  
\f, 102  
\n, 102  
\o, 102  
\oo, 102  
\ooo, 102

- `\r`, 102
- `\t`, 102
- `\v`, 102
- `\xh`, 102
- `\xhh`, 102
- `loader.so`, 100
- `{ }`, 44
- `'`, 102
- `absolute-path?`, 71
- `add1`, 10
- `'all`, 46
- `all-defined`, 26
- `all-defined-except`, 26
- `all-except`, 25
- `all-from`, 26
- `all-from-except`, 26
- `'already-exists`, 33
- `and`, 4
- `andmap`, 9
- `'any`, 65
- `'any-one`, 65
- `'append`, 61
- `append!`, 14
- `argv`, 46, 116
- `arithmetic-shift`, 11
- `arity`, 16, 32
- `arity-at-least-value`, 15
- `arity-at-least?`, 15
- `assoc`, 14
- `assq`, 14
- `assv`, 14
- `banner`, 112
- `begin`, 4, 7
- `begin0`, 4
- `bignum`, 10
- `'binary`, 60
- `bitwise operators`, 10
- `bitwise-and`, 10
- `bitwise-ior`, 10
- `bitwise-not`, 11
- `bitwise-xor`, 10
- `'block`, 62
- `bound-identifier=?`, 84
- `'bound-in-source`, 92
- `box`, 15
- `box?`, 15
- `boxes`, 15, 44, 101
  - `printing`, 45, 104
- `break-enabled`, 47
- `break-thread`, 39, 42
- `breaks`, *see* threads, breaking
- `Bruggeman, Carl`, 77
- `build-path`, 70
- `byte codes`, 105
- `call-in-nested-thread`, 42
- `call-with-current-continuation`, 35
- `call-with-custodian`, 33
- `call-with-escape-continuation`, 35
- `call-with-input-file`, 61
- `call-with-output-file`, 61
- `call/cc`, 35
- `call/ec`, 35
- `case sensitivity`, 44
- `case-lambda`, 7
- `char->integer`, 12
- `char->latin-1-integer`, 12
- `char-alphabetic?`, 12
- `char-ci<=?`, 12
- `char-ci<?`, 12
- `char-ci=?`, 12
- `char-ci>=?`, 12
- `char-ci>?`, 12
- `char-downcase`, 12
- `char-locale-alphabetic?`, 13
- `char-locale-ci<?`, 13
- `char-locale-ci=?`, 12
- `char-locale-ci>?`, 13
- `char-locale-downcase`, 13
- `char-locale-lower-case?`, 13
- `char-locale-numeric?`, 13
- `char-locale-upcase`, 13
- `char-locale-upper-case?`, 13
- `char-locale-whitespace?`, 13
- `char-locale<?`, 12
- `char-locale>?`, 12
- `char-numeric?`, 12
- `char-upcase`, 12
- `char-upper-case?`, 12
- `char-whitespace?`, 12
- `char<=?`, 12
- `char<?`, 12
- `char=?`, 12
- `char>=?`, 12
- `char>?`, 12
- `characters`, 12, 102
  - `printing`, 104
- `Check Syntax`, 92
- `check-duplicate-identifier`, 84
- `check-parameter-procedure`, 49
- `'client`, 54
- `collect-garbage`, 99
- `collection-path`, 114
- `collections`, 113
- `column numbers`, 68
- `command-line arguments`, 46, 115

- comments, 104
- communication, 42
- communications, 75
- compile, 105
- compile-allow-set!-undefined, 46
- compiled code, 44
- compiled-expression?, 105
- compiled-module-expression?, 97
- compiling, 105
- complete-path?, 71
- complex, 10
- cons-immutable, 14
- continuation-mark-set->list, 38
- continuation-mark-set?, 38
- continuation-marks, 37
- continuations, 35
  - escape, 35
- control flow, 35
- copy-file, 74
- curly braces, 44
- current namespace, 45
- current-command-line-arguments, 46, 116
- current-continuation-marks, 32, 37
- current-custodian, 47, 54
- current-directory, 44, 74
- current-drive, 74
- current-error-port, 44
- current-eval, 45
- current-exception-handler, 31, 47
- current-gc-milliseconds, 109
- current-input-port, 44, 60
- current-inspector, 48
- current-library-collection-paths, 46, 113, 115, 117
- current-load, 46
- current-load-extension, 46
- current-load-relative-directory, 46, 100, 106
- current-load/use-compiled, 46
- current-locale, 48
- current-memory-use, 99
- current-milliseconds, 11, 108
- current-module-name-prefix, 48
- current-module-name-resolver, 48
- current-namespace, 45
- current-output-port, 44, 60
- current-print, 45
- current-process-milliseconds, 108
- current-prompt-read, 45
- current-pseudo-random-generator, 11, 48
- current-seconds, 108
- current-security-guard, 47, 53
- current-thread, 41
- custodian-limit-memory, 55
- custodian-require-memory, 55
- custodian-shutdown-all, 54
- custodian?, 55
- custodians, 47, 54
- cycles, 105
- date, 108
- date, 108
- date-day, 108
- date-dst?, 108
- date-hour, 108
- date-minute, 108
- date-month, 108
- date-second, 108
- date-time-zone-offset, 108
- date-week-day, 108
- date-year, 108
- date-year-day, 108
- datum->syntax-object, 83
- decimal input, 10
- define**, 5
- define
  - internal, 6
- define-struct**, 18
- define-syntax**, 79
- define-syntaxes**, 88
- define-values**, 5
- 'delete, 53
- delete-directory, 74
- delete-file, 73
- directories
  - contents, 75
  - creating, 74
  - current, 44, 74
  - dates, 75
  - deleting, 74
  - moving, 74
  - of currently loading file, 46, 100, 106
  - pathnames, *see* pathnames
  - permissions, 75
  - renaming, 74
  - root, 75
  - testing, 74
- directory-exists?, 74
- directory-list, 75
- display, 105
- division by inexact zero, 10
- dump-memory-stats, 99
- Dybvig, Kent, 31, 77
- dynamic-require, 29
- dynamic-require-for-syntax, 29
- dynamic-wind, 36

- else, 4
- 'empty, 51
- environments
  - top-level, 77, 85
- eof, 60
- eof-object?, 60
- eq-hash-code, 17
- eq?, 12, 23
- 'equal, 16
- equal-hash-code, 17
- equal?, 9, 15, 23
- eqv?, 9, 10, 23
- 'error, 60
- error, 32, 33
- error display handler, 47
- error escape handlers, 40, 41
- error value conversion handler, 47
- error-display-handler, 47
- error-escape-handler, 40, 47
- error-print-source-location, 47
- error-print-width, 47
- error-value->string-handler, 47
- errors, 32, 33, 47
  - mismatch, 34
  - syntax, 34
  - type, 34
- eval, 45, 100
- evaluation handler, 45
- evaluation order, 3
- even?, 10
- 'exact, 109
- exception handlers, 41
- exceptions, 31, 47
  - primitive hierarchy, 32
- 'exec-file, 73
- 'execute, 53, 74
- 'exists, 54
- exit, 48, 101
- exit handler, 48
- exit-handler, 48
- exiting, 48
- exn, 32
  - exn:application:arity, 3, 15
  - exn:application:continuation, 36
  - exn:application:fprintf:mismatch, 68
  - exn:application:mismatch, 10, 14, 16, 22, 29, 34, 42, 43, 52, 61–64, 66–68, 91, 109
  - exn:application:type, 12–15, 21, 33, 34, 63, 68
  - exn:application;mismatch, 83
  - exn:break, 39, 43, 66, 67
  - exn:i/o:filesystem, 44, 60–62, 70, 71, 73, 74, 106, 107, 114
  - exn:i/o:port, 62
  - exn:i/o:tcp, 75, 76
  - exn:misc, 42, 107, 110, 111
  - exn:misc:application, 11, 12
  - exn:misc:unsupported, 55, 74, 107, 110
  - exn:module, 29, 30
  - exn:read, 101, 102, 105
  - exn:read:non-char, 104
  - exn:syntax, 34, 52, 79, 80, 91
  - exn:thread, 42
  - exn:user, 33
  - exn:variable, 52
  - exn?, 32
  - expand, 93
  - expand-once, 93
  - expand-path, 71
  - expand-to-top-form, 93
  - exponential input, 10
  - expressions
    - shared structure, 105
  - fields, 18
  - file-exists?, 73
  - file-or-directory-modify-seconds, 74, 75, 108
  - file-or-directory-permissions, 74, 75
  - file-position, 62
  - file-size, 74
  - file-stream-buffer-mode, 62
  - file-stream-port?, 61
  - files, 61
    - copying, 74
    - deleting, 73
    - loading, 100
    - modification dates, 74
    - moving, 73
    - pathnames, *see* pathnames
    - permissions, 74
    - renaming, 73
    - sizes, 74
    - testing, 73
  - filesystem-root-list, 75
  - finalization, *see* will executors
  - find-executable-path, 72
  - find-system-path, 72, 116, 117
  - fixnum, 10
  - floating-point-byte-string->real, 11
  - flonum, 10
  - fluid-let**, 6
  - fluid-let-syntax**, 89
  - flush-output, 62
  - force, 16
  - format, 68
  - fprintf, 67
  - fraction, 10

- free-identifier=?, 84
- Friedman, Dan, 31
  
- generate-temporaries, 83
- gensym, 14
- get-output-string, 61
- getenv, 111
- global port print handler, 44
- global-port-print-handler, 44, 67, 69
- graphs, 103, 105
  - printing, 104
- guardians, *see* will executors
  
- hash tables, 16
- hash-table-for-each, 17
- hash-table-get, 16
- hash-table-map, 16
- hash-table-put!, 16
- hash-table-remove!, 16
- hash-table?, 16
- Haynes, Chris, 31
- Hieb, Rob, 77
- 'home-dir, 72
- HOMEDRIVE**, 73
- HOMEPATH**, 73
  
- identifier macro, 91
- identifier-binding, 85
- identifier-binding-export-position, 85
- identifier-transformer-binding, 85
- identifier-transformer-binding-export-position, 85
- identifier?, 83
- 'ill-formed-path, 33
- immutable?, 14
- 'inferred-name, 35
- infinity, 10
- infix, 102
- info.ss**, 113
- 'init-dir, 73
- 'init-file, 73
- 'initial, 51
- initial-exception-handler, 47
- input ports
  - pattern matching, 56
- inspector?, 22
- inspectors, 21, 47
- integer->char, 12
- integer->integer-byte-string, 11
- integer-byte-string->integer, 11
  
- kill-thread, 41
  
- latin-1-integer->char, 12
  
- let, 5
- let\*, 5
- let\*-values, 5
- let-struct, 19
- let-syntaxes, 88
- let-values, 5
- let/cc, 35
- let/ec, 35
- let/ec, 35
- letrec, 5
- letrec-syntaxes, 88
- letrec-syntaxes+values, 88
- letrec-values, 5, 9
- 'lexical, 85
- libraries, 113
- 'line, 62
- line numbers, 68
- 'linefeed, 65
- link-exists?, 73
- links
  - creating, 74
  - testing, 73
- list, 14
- list\*, 14
- list\*-immutable, 14
- list-immutable, 14
- list-ref, 14
- list-tail, 14
- load, 46, 100, 104
- load extension handler, 46
- load handler, 46
- load-extension, 46, 106
- load-relative, 46, 100
- load-relative-extension, 46, 106
- load/cd, 46, 101
- load/use-compiled, 46, 100
- load/use-compiled handler, 46
- load/used-compiled, 46
- local-expand, 92
- locales, 48
- logical operators, *see* bitwise operators
  
- 'macos, 112
- 'macosx, 112
- macros, *see* syntax
- make-custodian, 54
- make-custom-input-port, 62
- make-custom-output-port, 64
- make-directory, 74
- make-file-or-directory-link, 74
- make-hash-table, 16
- make-input-port, 62
- make-inspector, 22
- make-namespace, 51

- make-output-port, 62
- make-parameter, 48
- make-pipe, 61
- make-pseudo-random-generator, 11
- make-security-guard, 53
- make-semaphore, 42
- make-set!-transformer, 91
- make-struct-field-accessor, 20
- make-struct-field-mutator, 20
- make-struct-type, 20
- make-struct-type-property, 21
- make-weak-box, 98
- make-will-executor, 98
- member, 14
- memq, 14
- memv, 14
- 'method-arity-error, 15
- module**, 24, 33
  - module name resolver, 28
  - module path index, 96
  - module-compiled-imports, 97
  - module-compiled-name, 97
  - 'module-direct-for-syntax-requires, 96
  - 'module-direct-requires, 96
  - module-identifier=?, 84
  - 'module-indirect-provides, 96
  - 'module-kernel-reprovide-hint, 97
  - module-path-index-join, 96
  - module-path-index-split, 96
  - module-path-index?, 96
  - 'module-self-path-index, 97
  - 'module-syntax-provides, 96
  - module-transformer-identifier=?, 84
  - 'module-variable-provides, 96
- modules, 24
  - body, 25
  - built-in, 29
  - compiling, 29
  - dynamic imports, 29
  - execution, 24
  - expansion, 24
  - exports, 25
  - for-syntax imports, 86
  - imports, 25
  - in files, 27
  - libraries, 27
  - macros, 27, 86
  - paths, 27
  - pre-defined, 29
  - predefined, 29
  - re-declaring, 29
  - re-defining, 29
  - redeclaring, 29
  - redefining, 29
  - syntax, 27
  - multiple return values, 3
  - MzLib library, 114
  - MzScheme
    - stand-alone, 1, 115
  - mzscheme, 24
  - MzScheme3m, 1
  - mzschemerc.ss**, 73
  - namespace-attach-module, 28, 52
  - namespace-mapped-symbols, 52
  - namespace-require, 52
  - namespace-require/copy, 52
  - namespace-require/expansion-time, 52
  - namespace-set-variable-value!, 52
  - namespace-symbol->identifier, 51
  - namespace-transformer-require, 52
  - namespace-variable-value, 52
  - namespace?, 51
  - namespaces, 50
    - initial, 51
    - initial environment, 51
    - initial transformer environment, 85
  - networking, 75
  - 'none, 46, 62
  - normal-case-path, 71
  - not-a-number, 10
  - not-break-exn?, 31
  - null, 14
  - null-environment, 50
  - number->string, 10
  - numbers, 10, 102
    - big-endian, 11
    - converting, 11
    - floating-point, 11
    - little-endian, 11
    - machine representations, 11
  - object-name, 35
  - object-wait-multiple, 43
  - object-wait-multiple/enable-break, 43
  - object-waitable?, 43
  - odd?, 10
  - open-input-file, 60
  - open-input-string, 61
  - open-output-file, 60
  - open-output-string, 61
  - or, 4
  - 'origin, 94
  - ormap, 9
  - 'oskit, 112
  - packages, 24

- parameter, 43
- parameter procedure, 43
- parameter-procedure=?, 48
- parameter?, 48
- parameterize**, 48
- parameters, 43
  - built-in, 44
- parsing, 44
- PATH**, 72
- path->complete-path, 71
- path-list-string->path-list, 73
- pathnames, 70
  - expansion, 70
- pattern matching, 56
- peek-char-or-special, 66
- peek-string, 66
- peek-string-avail!, 66
- peek-string-avail!\*, 66
- peek-string-avail!/enable-break, 66
- platform, 72, 112
- PLTCOLLECTS, 113
- port display handler, 69
- port positions, 68
- port print handler, 69
- port read handler, 69
- port write handler, 69
- port-count-lines!, 68
- port-display-handler, 69
- port-next-location, 68
- port-print-handler, 69
- port-read-handler, 69
- port-write-handler, 69
- port?, 60
- ports, 41, 44, 60
  - custom, 62
  - file, 61
  - flushing, 62
  - string, 61
- 'pref-dir, 72
- 'pref-file, 72
- prefix**, 25
- primitive procedure, 16
- primitive-closure?, 16
- primitive-result-arity, 16
- primitive?, 16
- print, 67, 105
- print handler, 45
- print-box, 45, 104
- print-graph, 45, 105
- print-struct, 45, 104
- print-vector-length, 45, 104
- printf, 68
- procedure-arity, 15
- procedure-arity-includes?, 15
- procedure?, 8
- processes, 109
- program, 73, 116
- promise?, 16
- promises, 16
- prompt read handler, 45
- provide**, 26
- pseudo-random-generator?, 11
- putenv, 111
  
- quasiquote**, 4
- quasisyntax**, 81
- quasisyntax/loc**, 82
- quote-syntax**, 78
  
- raise, 31
- raise-mismatch-error, 34
- raise-syntax-error, 34
- raise-type-error, 34
- random, 11, 48
- random numbers, 48
- random-seed, 11, 48
- 'read, 53, 74
- read, 33
- read-accept-bar-quote, 45, 103, 104
- read-accept-box, 44, 101
- read-accept-compiled, 45, 106
- read-accept-dot, 45, 102
- read-accept-graph, 45, 105
- read-accept-quasiquote, 45, 102
- read-case-sensitive, 44, 103, 104
- read-char-or-special, 66
- read-curly-brace-as-paren, 44, 101
- read-decimal-as-inexact, 45, 102
- read-eval-print loop, 45
  - read-eval-print loop
    - customized, 101
- read-eval-print loop, 101
- read-eval-print-loop, 101, 117
- read-image-from-file, 107
- read-line, 65
- read-square-bracket-as-paren, 44, 101
- read-string, 65
- read-string-avail!, 65
- read-string-avail!\*, 66
- read-string-avail!/enable-break, 66
- read-syntax, 78, 95
- real->floating-point-byte-string, 12
- regexp, 56
- regexp-match, 56, 57
- regexp-match-peek, 58
- regexp-match-peek-positions, 58
- regexp-match-positions, 58

- regexp-replace, 58
- regexp-replace\*, 58
- regexp?, 56
- regular expressions, 56
- 'relative, 72
- relative-path?, 71
- rename, 25, 26
- rename-file-or-directory, 73, 74
- 'replace, 60
- require, 25, 33, 46
- require-for-syntax, 86, 87
- resolve-path, 71
- 'return, 65
- 'return-linefeed, 65
- reverse!, 14
- run-time hierarchy, 88
- 'running, 110
  
- 'same, 70, 72
- scheme-report-environment, 50
- scripts, 118
- seconds->date, 108
- security guards, 47, 53
- security-guard?, 54
- semaphore-post, 42
- semaphore-try-wait?, 42
- semaphore-wait, 42
- semaphore-wait/enable-break, 43
- semaphore?, 42
- semaphores, 42
- 'server, 54
- set!, 6
- set!, 116
- set!-transformer?, 91
- set!-values, 6
- set-box!, 15
- shell scripts, 118
- shell-execute, 110
- ShellExecute, 110
- simplify-path, 71
- sleep, 41
- sockets, 75
- split-path, 71
- square brackets, 44
- string->immutable-string, 13
- string->number, 10
- string->symbol, 14
- string->uninterned-symbol, 14
- string-ci<=?, 13
- string-ci<?, 13
- string-ci=?, 13
- string-ci>=?, 13
- string-ci>?, 13
- string-locale-ci<?, 13
- string-locale-ci=?, 13
- string-locale-ci>?, 13
- string-locale<?, 13
- string-locale>?, 13
- string<=?, 13
- string<?, 13
- string=?, 13
- string>=?, 13
- string>?, 13
- strings, 102
  - as ports, 61
  - immutable, 13
  - pattern matching, 56
  - printing, 104
  - reading to and writing from, 61
- struct, 26
- struct->vector, 22
- struct-accessor-procedure?, 23
- struct-constructor-procedure?, 23
- struct-info, 22
- struct-mutator-procedure?, 23
- struct-predicate-procedure?, 23
- struct-type-info, 22
- struct-type-property?, 21
- struct-type?, 23
- struct?, 23
- structs
  - printing, 45
- structure subtypes, 19
- structure type descriptors, 18
- structure type properties, 21
- structure types, 18
  - predicates, 23
- structures, 18
  - equality, 23
  - printing, 104
- sub1, 10
- subprocess, 109
- subprocess-kill, 110
- subprocess-pid, 110
- subprocess-status, 110
- subprocess-wait, 110
- subprocess?, 110
- subprocesses, 109
- 'SW\_HIDE, 111
- 'SW\_MAXIMIZE, 111
- 'SW\_MINIMIZE, 111
- 'SW\_RESTORE, 111
- 'SW\_SHOW, 111
- 'SW\_SHOWDEFAULT, 111
- 'SW\_SHOWMAXIMIZED, 111
- 'SW\_SHOWMINIMIZED, 111
- 'SW\_SHOWMINNOACTIVE, 111

- 'SW\_SHOWNA, 111
- 'SW\_SHOWNOACTIVATE, 111
- 'SW\_SHOWNORMAL, 111
- symbols, 102
  - case sensitivity, 103
  - generating, 13
  - printing, 104
  - unique, 13
- syntax, 77
  - expanding, 93
  - macro calls, 90
  - modules, 86
  - partial expansion, 92
- syntax**, 79
- syntax objects, 78
  - comparisons, 84
  - identifier, 83
  - operations, 82
  - pattern-matching, 79
  - properties, 94, 106
  - source location, 82, 106
  - source module, 82
- syntax pair, 82
- syntax->list, 83
- syntax-case**, 79
- syntax-case\***, 80
- syntax-column, 82
- syntax-e, 82
- syntax-graph?, 83
- syntax-line, 82
- syntax-local-context, 91
- syntax-local-introduce, 92
- syntax-local-name, 91
- syntax-local-value, 91
- syntax-object->datum, 83
- syntax-original?, 82
- syntax-position, 82
- syntax-property, 94
- syntax-rules**, 77
- syntax-source, 82
- syntax-source-module, 82
- syntax-span, 82
- syntax/loc**, 81
- syntax?, 82
- 'sys-dir, 73
- system-big-endian?, 12
- system-library-subpath, 112
- system-type, 112
  
- tail calls, 35
- tcp-abandon-port, 76
- tcp-accept, 76
- tcp-accept-ready?, 76
- tcp-accept/enable-break, 76
  
- tcp-addresses, 76
- tcp-close, 76
- tcp-connect, 75
- tcp-connect/enable-break, 75
- tcp-listen, 75
- tcp-listener?, 76
- TCP/IP, 75
- 'temp-dir, 73
- 'text, 60
- thread, 41
- thread descriptor, 41
- thread-running?, 41
- thread-wait, 41
- thread?, 41
- threads, 41
  - breaking, 39, 42
  - communication, 42
  - killing, 41
  - nesting, 42
  - run state, 41
  - synchronization, 42
- time, 108
  - machine, 108
- time**, 109
- time-apply, 109
- TMPDIR**, 73
- top-level environment, *see* namespaces
- transformer environments, 77
  - in modules, 86
- transformers, 77
  - application, 90
- 'truncate, 60
- 'truncate/replace, 60
  
- unbox, 15
- undefined values, 9
- uninterned symbol, 13
- 'unix, 112
- unless**, 4
- unquote**, 5
- unquote-splicing**, 5
- unsyntax**, 81
- unsyntax-splicing**, 81
- 'up, 70, 72
- 'update, 61
- use-compiled-file-kinds, 46
  
- vectors, 101
  - printing, 45, 104
- version, 112
- vertical bar, 45
- void, 9
- void?, 9

'weak, 16  
weak boxes, 98  
weak-box-value, 98  
weak-box?, 98  
**when**, 4  
will executors, 98  
will-execute, 98  
will-executor?, 98  
will-register, 98  
will-try-execute, 98  
'windows, 112  
**with-continuation-mark**, 37  
**with-handlers**, 31  
with-input-from-file, 61  
with-output-to-file, 61  
**with-syntax**, 80  
'write, 53, 74  
write, 105  
write-image-to-file, 107, 116  
write-string-avail, 67  
write-string-avail\*, 67  
write-string-avail/enable-break, 67  
'wrong-version, 33, 106