PLT MzScheme: Language Manual

Matthew Flatt
mflatt@cs.rice.edu
Rice University

Version 100alpha3
June 1999

Department of Computer Science – MS 132
Rice University
6100 Main Street
Houston, Texas 77005-1892

Copyright notice

Send us your Web links

If you use any parts or all of the DrScheme 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@cs.rice.edu*. 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 and John Ellis for the conservative garbage collector and their help.

Thanks also to Shriram Krishnamurthi, Robby Findler, Matthias Felleisen, Cormac Flanagan, Sebastian Good, Bruce Duba, and many others for feedback and help.

This manual was typest using LaTeX and a patched version of `latex2html`. Some typesetting macros were originally taken from Julian Smart's *Reference Manual for wxWindows 1.60: a portable C++ GUI toolkit*.

Contents

1. Introduction

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

MzScheme (pronounced "Ms. Scheme" or "Miz Scheme") is nearly R^5RS-compliant. MzScheme does not provide the `define-syntax`, `let-syntax`, and `letrec-syntax` forms, although they are partially supported by an external library. All other R^5RS syntactic forms and procedures are provided by MzScheme with their prescribed semantics. Certain parameters in MzScheme can change features affecting R^5RS-compliance; for example, case-sensitivity can be enabled (see §9.4.1.3).

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

- A class and object system (see Chapter 6).
- A unit system for separate compilation of program components (see Chapter 7).
- An exception system that is used for all primitive errors (see Chapter 8).
- Pre-emptive threads and multiple global variable namespaces (see Chapter 9).

MzScheme can be run as a stand-alone application, or it can be embedded within other applications. 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 16.

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*.

1.2 Notation

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

a "1" subscript ($\cdots^1$) indicate *one* or more repetitions of the preceding S-expression pattern.

For example:

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

The first set of ellispes indicate that any number of *variable*s (or none) can be provided with a single *expr*. The second set of ellipses indicate that any number of `((`*variable* ···`)` *expr*`)` combinations (or 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 (`...`); 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 denote optional arguments. For example,

`(raise-syntax-error` *name-symbol message-string* [*expr sub-expr*]`)`

describes the calling convetion for a procedure `raise-syntax-error` where the *name-symbol* and *message-string* arguments are required, and the *expr* and *sub-expr* arguments are optional (but *expr* must be provided if *sub-expr* is provided).

2. Multiple Return Values

Most Scheme expressions return a single value. For example, in the following procedure:

```
(define quotient-and-remainder
  (lambda (n d)
    (let ([q (quotient n d)]
          [r (remainder n d)])
      (cons q r))))
```

the expressions `(quotient n d)` and `(remainder n d)` each return a single value. The expression `(cons q r)` also returns a single value; although two other values can be extracted from the cons-cell created by `cons`, the cons-cell is itself just a single value. Thus, this `quotient-and-remainder` procedure returns a single value.

On the other hand, a context using `quotient-and-remainder` is probably interested only in the two numbers returned by the procedure, not the cons-cell used to store the numbers. A client context might be best expressed with the `let-values` form:

```
(let-values ([(q r) (quotient-and-remainder-values n d)])
  (printf "~a * ~a + ~a = ~a~n" q d r n))
```

This `let-values` form expresses the idea that `quotient-and-remainder-values` returns *two* values, and these two values are bound directly to `q` and `r`. For the `let-values` expression to be correct, the `quotient-and-remainder-values` procedure must specifically return two values by using the `values` procedure:

```
(define quotient-and-remainder-values
  (lambda (n d)
    (values (quotient n d)
            (remainder n d))))
```

Any MzScheme expression can return multiple values. Multiple return values are generated by the `values` procedure, which bundles and returns its arguments as multiple return values. A bundle of return values is *not* itself a first-class value. Rather, the individual values must unbundled by the context of the expression returning multiple values. A run-time error is signalled when multiple values are returned to a context expecting a single value. For example, the following expression signals a run-time error:

```
(let ([x (values 1 2)]) x)
```

In this example, the expression for `x`'s value must return a single value (since `x` is a single variable), but `(values 1 2)` returns multiple values to the binding context. Most contexts are like this one, expecting a single value. The `let-values` form creates a context expecting (a particular number of) mulitple values. The `call-with-values` procedure also creates a multiple values return context by transforming the individual values of a multiple-value return into the arguments of a procedure call.

Binding form contexts for multiple-value expressions are discussed in §3.5. Only the `values` and `call-with-values` procedures are described here:

- `(values` $v \cdots$`)` returns the vs as multiple values. If zero vs or more than one v is provided, the result of this expression must be used in a multiple-value context; e.g., the result must be used by `call-with-values`, `define-values`, or `let-values`, or it must be ignored. `(values` v`)` is always equivalent to v.

- `(call-with-values` *producer-proc consumer-proc*`)` invokes *producer-proc* and passes the result to *consumer-proc*. The *producer-proc* argument is a procedure that takes no arguments and returns some number of values. The *consumer-proc* argument is a procedure that takes the same number of arguments as the number of values returned by *producer-proc*. The result of the `call-with-values` expression is the result of applying *consumer-proc*.

Multiple return values are legal whenever the return value of an expression is ignored, or when it passed on as the result of an enclosing expression (provided that the enclosing expression can legally return multiple values). For example, either branch of an `if` expression can return multiple values; in this case the result of the entire `if` expression might be multiple values. However, the test expression of an `if` expression must return a single value because that value is used as a test. Similarly, the body expressions of a `let` form can return multiple values, but the binding expressions must return single values. (This is true even when the variable that is bound is not used in the body of the `let` expression.) An expression used in a procedure application (either as the procedure to be applied or one of the arguments) must always return a single value.

If a built-in procedure takes a procedure argument, and it 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)"
```

3. Basic Syntax Extensions

3.1 Evaluation Order

In an application expression, the procedure expression and the argument expressions are always evaluated left-to-right.

3.2 Conditionals

3.2.1 Cond and Case

In MzScheme's normal mode,[1] the result of a `cond` expression with no matching clause is void (see §4.1). In other modes, evaulating a non-matching `cond` raises the `exn:else` exception.

Every `case` expression is expanded into a `cond` expression; depending on the MzScheme's running mode, evaluating a `case` expression with no matching clause will raise the `exn:else` exception.

An `else` clause in a `cond` or `case` expression must be the last clause, otherwise a syntax error is signalled. If `=>` is used in the second position within a `cond` clause and it is not followed by a single recipient expression, a syntax error is signalled.

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

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

3.2.2 When and Unless

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

- (`when` *test* *expr* ...[1]) evaluates the *expr* body expressions only when *test* returns a true value.
- (`unless` *test* *expr* ...[1]) evaluates the *expr* body expressions only when *test* 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 §4.1) if the body is not evaluated.

[1] This mode is controlled by the `compile-allow-cond-fallthrough` parameter (see §9.4.1.5). The default mode of a running MzScheme depends on command line arguments or internal settings established by an application with an embedded MzScheme. In MrEd, the default mode is to raise the `exn:else` exception for non-matching `cond` and `case` expressions.

3.2.3 And and Or

In an `and` or `or` expression, the last test expression can return multiple values (see Chapter 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.

3.3 Sequences

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

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

3.4 Quasiquote

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.

3.5 Binding Forms

3.5.1 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 *variable*s provided.

All of the variables are bound sequentially after *expr* is evaluated. If an error occurs while binding one of the definitions (perhaps because the variable is a constant that is already defined), then the definitions for the preceding *variable*s will have already completed, but definitions for the remaining *variable*s will never complete.

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
```

3.5.2 Local Variables

Local variables are bound with standard Scheme's `let`, `let*`, and `letrec`. MzScheme's `letrec` form gurantees sequential evaualtion 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 *variable*s declared in the corresponding clause. Each *expr* remains outside of the scope of all variables bound by the `let-values` expression.

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

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

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

When a `letrec` or `letrec-values` expression is evaluated, each variable binding is initially assigned the special undefined value (see §4.1); the undefined value is replaced once 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
```

3.5.3 Assignments

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

`(set!-values (`*variable* ⋯`)` *expr*`)`

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

The *variable*s can be any mixture of global and local 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 a constant or is not yet bound), then the assignments for the preceding *variable*s will have already completed, but assignments for the remaining *variable*s will never complete.

3.5.4 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 variable that is bound before the `fluid-let` expression is evaluated. Before the *body-expr*s are evaluated, the bindings for the *variable*s are `set!` to the values of the corresponding *expr*s. Once the *body-expr*s have been evaluated, the values of the variables are restored. The value of the entire `fliud-let` expression is the value of the last *body-expr*.

3.5.5 Syntax Expansion and Internal Definitions

All binding forms are macro expanded into `define-values`, `let-values`, and `letrec-values` expressions. The `set!-values` form is expanded to `let-values` with `set!`. See §13.4 for more information.

All `define-values` expressions that are inside only `begin` expressions are treated as top-level definitions. Immediate body `define-value` expressions in a `unit` expression are handled specially as described in §7.1.1. Any other `define-values` expression is either an **internal definition** or syntactically illegal.

Internal definitions can appear at the beginning of the body in a `lambda`, `case-lambda`, `let`, `let-values`, `let*`, `let*-values`, `letrec`, `letrec-values`, `fluid-let`, `let-macro`, `let-id-macro`, `let-epansion-time`, `parameterize`, or `with-handlers` expression. At least one non-definition expression must follow a sequence of internal definitions.

When a `begin` expression appears within an implicit 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 in an internal definition context can be empty.

An internal `define-values` expression is transformed along with the rest of the expressions following it into a `letrec-values` expression: the variables originally bound by the `define-values` expressions become the binding variables of the new `letrec-values` expression, and the expressions that followed the `define-values` expressions become the body of the new `letrec-values` expression.

Multiple adjacent `define-values` statements are collected into a single `letrec-values` transformation so that the definitions can be mutually-recursive, but the `define-values` expressions really must be adjacent: a `define-values` expressions following a non-`define-values` expression is not an internal definition.

An internal defintion cannot shadow a syntax form or macro name. Thus, an internal definition cannot alter the decision of whether another expression is also an internal definition in the same `letrec-values`

transformation.[2]

Internal macro definitions (using `define-macro`) are desecribed in §13.1.

3.6 Case-Lambda

The `case-lambda` form creates a procedure that dispatches to a particular body of expressions based on the *number* of arguments it receives. This provides a mechanism for creating variable-arity procedures with more control and efficiency than using a "rest arg" — e.g., the `x` in `(lambda (a . x) ...)` — with a `lambda` expression.

A `case-lambda` expression has the form:

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

formals is one of:

```
variable
(variable ...)
(variable ... . identifer)
```

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

When a `case-lambda` procedure is invoked, one clause is selected and its *expr*s are evaluated for the application; the result of the last *expr* in the clause is the result of the application. The clause that is selected for an application is the first one with a *formals* specification that can accommodate the number of arguments in the application.[3]

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

[2] Since an internal macro definition is not a regular internal definition, there is no ambiguity about whether an internal macro defintion applies to previous expressions that are potentially internal defintions; the macro only applies to expressions after the macro definition.

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

4. Basic Data Extensions

4.1 Void and Undefined

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

- `(void` v $\cdots$`)` returns void.
- `(void?` v`)` returns `#t` if v is void, `#f` otherwise.

Non-global variables that are accessible but do not yet have a value are bound to the unique **undefined** value, printed as `#<undefined>`. Such variables are created by `letrec-values` expressions (see §3.5), partially-initialized objects (see Chapter 6), and partially-invoked units (see Chapter 7).

4.2 Booleans

Unless otherwise specified, two instances of a particular MzScheme data type are `equal?` only when they are `eq?`.

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* $\cdots^1$`)` applies *proc* to elements of the *list*s from the first elements to the last, returning `#f` as soon as any application returns `#f`. If no application of *proc* returns `#f`, then the result of the last application of *proc* is returned. If the *list*s are empty, then `#t` is returned.
- `(ormap` *proc list* $\cdots^1$`)` applies *proc* to elements of the *list*s from the first elements to the last. If any application returns a value other than `#f`, that value is immediately returned as the result of the `ormap` application. If all applications of *proc* return `#f`, then the result is `#f`. If the *list*s are empty, then `#f` is returned.

Examples:

```
(andmap positive? '(1 2 3)) ; ⇒ #t
(ormap eq? '(a b c) '(a b c)) ; ⇒ #t
(andmap positive? '(1 2 a)) ; ⇒ raises exn: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
```

4.3 Numbers

A number in MzScheme is one of the following:

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

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`). They 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?`; `+nan.0` is not an integer and is not `=` to itself, but `+nan.0` is `eqv?` to itself.[2] Similarly, `(= 0.0 -0.0)` is `#t`, but `(eqv? 0.0 -0.0)` is `#f`.

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

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 $\cdots^1$`)` returns the bitwise "inclusive or" of the ns.
- `(bitwise-and` n $\cdots^1$`)` returns the bitwise "and" of the ns.
- `(bitwise-xor` n $\cdots^1$`)` returns the bitwise "exclusive or" of the ns.
- `(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:

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

- `(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 psuedo-random number generator, which maintains an internal state for generating numbers.[3]

- `(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 §9.4.1.13.

- `(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.

4.4 Characters

MzScheme character values range over the characters for ASCII values 0 to 255. The procedure `char->integer` returns the ASCII value of a character and `integer->char` takes an ASCII value and returns the corresponding character. If `integer->char` is given an integer that is not in 0 to 255, the `exn:application:type` exception is raised.

The character comparison procedures — `char=?`, `char-ci=?`, etc. — take one or more character arguments and check the arguments pairwise (like the numerical comparison procedures).

4.5 Strings

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-ci=?`, etc. — take one or more string arguments and check the arguments pairwise (like the numerical comparison procedures).

4.6 Symbols

MzScheme provides two ways of generating an **uninterned symbol**, i.e., a symbol that is not `eq?`, `eqv?`, or `equal?` to any other symbol:

- `(string->uninterned-symbol` *string*`)` is like `(string->symbol` *string*`)`, but the resulting symbol is a new uninterned symbol. Calling `string->uninterned-symbol` twice for the same s 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.

[3] The random number generator uses a relatively standard Unix `random()` implementation in its degree seven polynomial mode.

4.7 Vectors

When a vector is created with `make-vector` without a fill value, it is initialized with `0` in all positions.

4.8 Lists

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

`(reverse!` *list*`)` is the same as `(reverse` *list*`)`, but *list* is destructively reversed.

`(append!` *list* $\cdots^1$`)` destructively appends the *list*s.

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

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

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 asscociation 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.

4.9 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?`.

4.10 Procedures

4.10.1 Arity

MzScheme's `arity` procedure inspects the input arity of a procedure:

- `(arity` *proc*`)` returns information about the number of arguments accepted by the procedure *proc*. The result *a* is either:
  - an exact non-negative integer ⇒ the procedure always takes exactly *a* arguments;

  - an `arity-at-least` structure value ⇒ the procedure takes `(arity-at-least-value` *a*`)` or more arguments; or
  - a list containing integers and `arity-at-least` structure values ⇒ 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:

```
(arity cons) ; ⇒ 2
(arity list) ; ⇒ #<struct:arity-at-least>
(arity-at-least? (arity list)) ; ⇒ #t
(arity-at-least-value (arity list)) ; ⇒ 0
(arity-at-least-value (arity (lambda (x . y) x))) ; ⇒ 1
(arity (case-lambda [(x) 0] [(x y) 1])) ; ⇒ (1 2)
(procedure-arity-includes? cons 2) ; ⇒ #t
(procedure-arity-includes? display 3) ; ⇒ #f
```

4.10.2 Primitives

A **primitive procedure** is a procedure that is built into MzScheme, although not all built-in procedures (see §9.3.1) are primitives. Almost all R^5RS procedures are primitives, as are most of the procedures described in this manual (except for MzLib procedures).

- `(primitive?` *v*`)` returns `#t` if *v* is a primitive procedure or `#f` otherwise.
- `(primitive-name` *prim-proc*`)` returns the name of the primitive procedure *prim-proc* as a string.
- `(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 §4.10.1). For most primitives, this procedure returns `1` since most primitives return a single value when applied. For information about arity values, see §4.10.1.
- `(primitive-closure?` *v*`)` returns `#t` if *v* is internally implemented as a primitive closure rather than an simple primitive procedure, `#f` otherwise. This information is intended for use by the **mzc** compiler.
- `(simple-return-primitive?` *prim-proc*`)` returns `#t` if the given primitive procedure never computes its return value by an internal chained tail call. This information is intended for use by the **mzc** compiler.

4.10.3 Procedure Names

See §8.2.4 for information about the names inferred for `lambda` and `case-lambda` procedures.

4.11 Promises

MzScheme implements `delay` as a macro that expands to `make-promise`. The `force` procedure can only be applied to values returned by `make-promise`, and promises are never implicitly `force`d.

- `(make-promise` *thunk*`)` returns a new promise, where *thunk* is a procedure of zero arguments.
- `(promise?` *v*`)` returns `#t` if *v* is a promise created by `make-promise` or `#f` otherwise.

4.12 Hash Tables

MzScheme provides efficient built-in hash tables. Key comparisons use `eq?`.

- `(make-hash-table)` creates and returns a new hash table.
- `(make-hash-table-weak)` creates a hash table with weakly-held keys (see §12.1).
- `(hash-table?` *v*`)` returns `#t` if *v* was created by `make-hash-table` or `make-hash-table-weak`, `#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 argments) is returned. If *failure-thunk* is not provided, the `exn:application:mismatch` exception is raised if 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.
- `(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.

5. Structures

A **structure type** is a record datatype composed of a number of named **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- and field-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.

5.1 Creating Structure Types

A new structure type is created with one of two `struct` forms:

```
(struct s (field ···))
(struct (s t-expr) (field ···))
```

where *s* and each *field* are identifiers. The latter form is described in §5.2.

A `struct` expression with *n* *field*s returns $3 + 2n$ values:

- `struct:`*s*, a **structure type descriptor** value that represents the new datatype. The purpose of this value is explained in §5.2.
- `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 §5.2) and `#f` for any other value.
- *s*`-`*field*, for each *field*, a selector procedure that takes a structure value and extracts the value for *field*.
- `set-`*s*`-`*field*`!`, for each *field*, a setter 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.

The order of the return values from a `struct` expression is the same as in the list above, up and including to the setter procedure for the first field (if the structure type has any fields). If the structure type has more than one field, the selector for the second field is next, followed by the setter for the second field, and so on for additional fields.

The names `make-`*s*, etc. are only used by the return values of `struct` for error-reporting. But these names are used as binding variables by the `define-struct` and `let-struct` macros:

```
(define-struct s (field ···))
⇒
(define-values (struct:s make-s s? s-field set-s-field! ···) (struct s (field ···)))

(let-struct s (field ···)
```

```
   body-expr ...[1])
⇒
(let-values ([(struct:s make-s s? s-field set-s-field! ...) (struct s (field ...))])
   body-expr ...[1])
```

Each time a `struct` expression is evaluated, a new structure type is created with distinct constructor, predicate, selector, and setter procedures. If the same `struct` expression is evaluated twice, instances created by the constuctor 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
```

5.2 Creating Subtypes

The second `struct` form shown in §5.1 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-expr* expression in a subtyping `struct` form is evaluated when the `struct` expression is evaluated. The result of *t-expr* must be a structure type descriptor (returned as the first value of a `struct` expression that was evaluated earlier). The structure type associated with this descriptor is used as the base structure type for the new subtype. If the value of *t-expr* is not a structure type descriptor value, the `exn:struct` exception is raised.

A structure subtype "inherits" the fields of its base type. If the base type has m fields and n fields are specified in the subtyping `struct` expression, the resulting structure type has $m+n$ fields. This means that $m+n$ 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 selectors and setters for the first m fields are not created (so the number of values returned by a `struct` expression is always syntactically known, even though the actual base type is not known until run time).

The `define-struct` and `let-struct` macros have forms that support subtyping:

```
(define-struct (s t) (field ...))

⇒
```

```
(define-values (struct:s make-s s? s-field set-s-field! ···) (struct (s t) (field ···)))

(let-struct (s t) (field ···)
   body-expr ···[1])
⇒
(let-values ([(struct:s make-s s? s-field set-s-field! ···) (struct (s t) (field ···))])
   body-expr ···[1])
```

Examples:

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

5.3 Structure Utilities

Structures can only be created and changed with the constructor and setter procedures created by `struct`, but structures are not opaque. These utility procedures work with all structure instance values:

- (`struct?` *v*) returns `#t` if *v* was created by any `make-`*s* or `#f` otherwise. All other built-in predicates return `#f` for values constructed by a `make-`*s*.
- (`struct-length` *struct*) returns the number of field values in the structure *struct*.
- (`struct-ref` *struct k*) returns the value of the *k*th field of the structure *struct*. (The first field is index 0.) If *struct* does not have an *k*th field value, the `exn:application:mismatch` exception is raised.
- (`struct->vector` *struct*) converts the structure value *struct* to a vector. The first slot of the result vector contains an symbol of the form `struct:`*s*. The remaining slots contain the field values of *struct*. The `struct->vector` procedure is intended for printing and debugging use.

Two structure values are `eqv?` if and only if they are `eq?`. Two structure values are `equal?` if they have the same structure type and their corresponding field values are all `equal?`.

Each kind of value returned by `struct` 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 `struct`, `#f` otherwise.
- (`struct-predicate-procedure?` *v*) returns `#t` if *v* is a predicate procedure generated by `struct`, `#f` otherwise.
- (`struct-getter-procedure?` *v*) returns `#t` if *v* is a selector procedure generated by `struct`, `#f` otherwise.
- (`struct-setter-procedure?` *v*) returns `#t` if *v* is a setter procedure generated by `struct`, `#f` otherwise.

6. Classes and Objects

A MzScheme **class** specifies

- a collection of instance variables;
- initial value expressions for the instance variables; and
- initialization variables that are bound to initialization. arguments.

An **object** is a collection of bindings for instance variables that are instantiated according to a class description. There is no distinction between "methods" and "instance variables"; a method is simply an instance variable with a procedural value.

The core feature of the object system is the ability to define a new class (a **derived class**) in terms of an existing class (the **superclass**) using inheritance and overriding:

- **inheritance**: An object of a derived class instantiates variables declared by the derived class's superclass as well as variables declared in the derived class expression.
- **overriding**: A variable declared in a superclass can be redeclared in the derived class. References to the overridden variable in the superclass use the binding of the derived class's declaration.

An **interface** is a collection of variable names that could be declared by a class. A class **implements** an interface when it

- declares (or inherits) a public instance variable for each variable in the interface; and
- specifically declares its intention to implement the interface.

A single class can implement any number of interfaces. A derived class automatically implements any interface that its superclass implements.

A new interface can **extend** one or more interfaces with additional variables; classes that implement the extended interface also implement the original interfaces.

Classes, objects, and interfaces are all first-class Scheme values. However, a MzScheme class or interface is not a MzScheme object (i.e., there are no "meta-classes" or "meta-interfaces").

6.1 Object Example

Since most readers will be familiar with a other object systems, we begin with an example use of MzScheme's object system to illustrate its particular style.

```
(define stack<%> (interface () push! pop! empty?))

(define stack%
  (class* object% (stack<%>) ()
    (private
      [stack null]) ; A private instance variable
    (public
      [name 'stack] ; A public instance variable
      [push! (lambda (v)
               (set! stack (cons v stack)))]
      [pop! (lambda ()
              (let ([v (car stack)])
                 (set! stack (cdr stack))
                 v))]
      [empty? (lambda () (null? stack))]
      [print-name (lambda ()
                    (display name) (newline))])
   (sequence (super-init))))

(define named-stack%
  (class stack% (stack-name)
    (override
      [name stack-name])
    (sequence
      (super-init))))

(define double-stack%
  (class stack% ()
    (inherit push!)
    (override
      [name 'double-stack])
    (public
      [double-push! (lambda (v)
                      (push! v)
                      (push! v))])
    (sequence (super-init))))

(define-values (make-safe-stack-class is-safe-stack?)
  (let ([safe-stack<%> (interface (stack<%>))])
    (values
     (lambda (super%)
       (class* super% (safe-stack<%>) ()
         (inherit empty?)
         (rename [std-pop! pop!])
         (override
           [name 'safe-stack]
           [pop! (lambda ()
                   (if (empty?) #f (std-pop!)))])
         (sequence (super-init))))
     (lambda (obj)
      (implementation? (object-class obj) safe-stack<%>)))))

(define safe-stack% (make-safe-stack-class stack%))
```

The interface `stack<%>`[1] defines the ever-popular stack interface with the methods `push!`, `pop!`, and `empty?`. The class `stack%`[2] is derived from the built-in empty class `object%` and implements the `stack<%>` interface. Three additional classes are derived from the basic `stack%` implementation:

- The class `named-stack%` defines a stack that is named through the initialization argument. It overrides the definition of `name` in `stack%`.
- The class `double-stack%` extends the functionality `stack%` with a new method, `double-push!`, and also overrides the definition of `name` in `stack%`.
- The class `safe-stack%` overrides `name` and the `pop!` method of `stack%`, insuring that `#f` is returned whenever the stack is empty.

In each derived class, the call `(super-init)` causes the bindings of the superclass's instance variables to be initialized when an instance of the derived class is initialized.

The creation of `safe-stack%` illustrates the use of classes as first-class values. Applying `make-safe-stack-class` to `named-stack%` or `double-stack%` — indeed, *any* class with `push`, `pop!`, and `empty?` methods — creates a "safe" version of the class. A stack object can be recognized as a safe stack by testing it with `is-safe-stack?`; this predicate returns `#t` only for instances of a class created with `make-safe-stack-class` (because only those classes implement the `safe-stack<%>` interface).

In each of the example classes, the instance variable `name` contains the name of the class. The `name` instance variable is introduced as a new instance variable in `stack%`, so it is declared there with the `public` keyword. The `name` declarations in `named-stack%`, `double-stack%`, and `safe-stack%` override the declaration in `stack%`, so they are declared with the `override` keyword. When the `print-name` method of an object from `double-stack%` is invoked, the name printed to the screen is "double-stack".

While all of `named-stack%`, `double-stack%`, and `safe-stack%` inherit the `push!` method of `stack%`, it is declared with `inherit` only in `double-stack%`; this is because new declarations in `named-stack%` and `safe-stack%` do not need to refer to `push!`, so the inheritance does not need to be declared. Similarly, only `safe-stack%` needs to declare `(inherit empty?)`.

The `safe-stack%` class overrides `pop!` to *extend* the implementation of `pop!`. The new definition of `pop!` must access the original `pop!` method that is defined in `stack%`. The `rename` declaration binds a new name, `std-pop!` to the original `pop!`. Then, `std-pop!` is used in the overriding `pop!`. Variables declared with `rename` cannot be overridden, so `std-pop!` will *always* refer to the superclass's `pop!`.

The `make-object` procedure creates an object from a class; additional arguments to `make-object` are passed on as initialization arguments. Here are some object creations using the classes defined above:

```
(define stack (make-object stack%))
(define fred (make-object named-stack% 'Fred))
(define joe (make-object named-stack% 'Joe))
(define double-stack (make-object double-stack%))
(define safe-stack (make-object safe-stack%))
```

Note that an extra argument is given to `make-object` for the `named-stack%` class because `named-stack%` requires one initialization argument (the stack's name).

The `ivar` and `send` forms are used to access the instance variables of an object. The `ivar` form looks up a variable by name. The `send` form uses `ivar` to extract a variable's value, which should be a procedure;

[1] A bracketed percent sign ("`<%>`") is used by convention in MzScheme to indicate that a variable's value is a interface.
[2] A percent sign ("`%`") is used by convention in MzScheme to indicate that a variable's value is a class.

it then applies the procedure to arguments. For example, here is a simple expression that uses the objects created above:

```
((ivar stack push!) fred) ; or (send stack push! fred)
(send stack push! double-stack)
(let loop ()
  (if (not (send stack empty?))
      (begin
        (send (send stack pop!) print-name)
        (loop))))
```

This loop displays `'double-stack` and `'Fred` to the standard output port.

6.2 Creating Interfaces

The `interface` form creates a new interface:

`(interface (`*super-interface-expr* ···`)` *variable* ···`)`

All of the *variable*s must be distinct.

Each *super-interface-expr* is evalauted (in order) when the `interface` expression is evaluated. The result of each *super-interface-expr* must be an interface value, otherwise the `exn:object` exception is raised. The interfaces returned by the *super-interface-expr*s are the new interface's superinterfaces, which are all extended by the new interface. Any class that implements the new interface also implements all of the superinterfaces.

The result of an `interface` expression is an interface that includes all of the specified *variable*s, plus all variables from the superinterfaces. Duplicate variable names among the superinterfaces are ignored, but if a superinteface contains one of the *variable*s in the `interface` expression, the `exn:object` exception is raised.

6.3 Creating Classes

The built-in class `object%` has no methods and implements no interfaces. All other classes are derived from `object%`.

The `class*/names` form creates a new class:

`(class*/names` *local-names superclass-expr* `(`*interface-expr* ···`)` *initialization-variables*
instance-variable-clause
···`)`

local-names is:
`(`*this-variable super-init-variable*`)`

initialization-variables is one of:
variable
`(`*variable* ··· *variable-with-default* ···`)`
`(`*variable* ··· *variable-with-default* ··· `.` *variable*`)`

variable-with-default is:
`(`*variable default-value-expr*`)`

instance-variable-clause is one of:
  `(sequence` *expr* ···`)`
  `(public` *public-var-declaration* ···`)`
  `(override` *public-var-declaration* ···`)`
  `(private` *private-var-declaration* ···`)`
  `(inherit` *inherit-var-declaration* ···`)`
  `(rename` *rename-var-declaration* ···`)`

public-var-declaration is one of:
  `((`*internal-instance-variable external-instance-variable*`)` *instance-var-initial-value-expr*`)`
  `(`*instance-variable instance-var-initial-value-expr*`)`
  `(`*instance-variable*`)`
  instance-variable

private-var-declaration is one of:
  `(`*internal-instance-variable instance-var-initial-value-expr*`)`
  `(`*internal-instance-variable*`)`
  internal-instance-variable

inherit-var-declaration is one of:
  inherited-variable
  `(`*internal-instance-variable external-inherited-variable*`)`

rename-var-declaration is:
  `(`*internal-instance-variable external-inherited-variable*`)`

The `class*` macro is used to avoid specifying *local-names*:

`(class*` *superclass-expr* `(`*interface-expr* ···`)` *initialization-variables*
  instance-variable-clause
  ···`)`
⇒
`(class*/names (this super-init)` *superclass-expr* `(`*interface-expr* ···`)` *initialization-variables*
  instance-variable-clause
  ···`)`

The `class` macro omits both *local-names* and the *interface-expr*s:

`(class` *superclass-expr initialization-variables*
  instance-variable-clause
  ···`)`
⇒
`(class*` *superclass-expr* `()` *initialization-variables*
  instance-variable-clause
  ···`)`

The *this-variable* and *super-init-variable* variables (usually `this` and `super-init`) are bound in the rest of the `class*/names` expression, excluding *superclass-expr* and the *interface-expr*s. In instances of the new class, *this-variable* (i.e., `this`) is bound to the object itself, and *super-init-variable* (i.e., `super-init`) is bound to a procedure that must be invoked (once) to initialize instance variable bindings in the superclass (see §6.4).

The *superclass-expr* expression is evalauted when the `class*/names` expression is evaluated. The result

must be a class value (possibly `object%`), otherwise the `exn:object` exception is raised. The result of the *superclass-expr* expression is the new class's superclass.

The *interface-expr* expressions are also evaluated when the `class*/names` expression is evaluated, after *superclass-expr* is evaluated. The result of each *interface-expr* must be an interface value, otherwise the `exn:object` exception is raised. The interfaces returned by the *interface-expr*s are all implemented by the class. For each variable in each interface, the class (or one of its ancestors) must declare a public instance variable with the same name, otherwise the `exn:object` exception is raised.

The *initialization-variables* part of a `class*/names` expression defines the **initialization variables** as described in §6.3.1. The *instance-variable-clause*s define the class's **instance variables** as described in §6.3.2.

The result of a `class*/names` expression is a new class, derived from the specified superclass and implementing the specified interfaces. Instances of the class are created with the `make-object` procedure as described in §6.4.

6.3.1 Initialization Variables

A class's initialization variables are instantiated for each object of a class. The values bound to initialization variables are

- the arguments passed to the `make-object` procedure if the object is created as a direct instance of the class; or,
- the arguments passed to a superclass initialization procedure when the object is created as an instance of a derived class.

Initialization variables can be used in the initial value expressions of instance variables. Object creation and superclass initialization are described in detail in §6.4.

As shown in the the grammar in §6.3, the *initialization-variables* part of a `class*/names` expression has one of three forms. In the first form, all initialization arguments are put into a list and *variable* is bound to the list. In the second form, each *variable* is bound to an individual initialization argument. In the last form, initialization arguments are assigned to *variable*s preceding the dot; leftover initialization arguments are put into a list that is assigned to a the single *variable* after the dot.

If an initialization argument is not provided for a variable that has a *default-value-expr*, then the *default-value-expr* expression is evaluated to obtain a value for the variable. A *default-value-expr* is only evaluated when an argument is not provided for its variable. The environment of *default-value-expr* includes all of the initialization variables and all of the instance variables in the class (the latter have certainly not yet been initialized). If multiple *default-value-expr*s are evaluated, they are evaluated from left to right.

If too few initialization arguments are provided to `make-object` or to a superclass initialization procedure, then the `exn:application:arity` exception is raised.

6.3.2 Instance Variables

Each *instance-variable-clause* declares a number of instance variables for instances of the class, or expressions to be evaluated when an instance of the class is created. The first part of a clause is the clause specifier, one of `sequence`, `public`, `private`, etc. A clause specifier determines the properties of instance variables declared in its clause:

- `sequence` does not declare any instance variables. It only specifies expressions to be evaluated (in order) when an instance of the class is initialized.
- `public` declares fresh instance variables that are visible outside the class. A derived class can override a `public` declaration, but a `public` declaration never overrides an existing declaration in the superclass.
- `override` is like `public`, but an `override` declaration always overrides an existing declaration in the superclass. A derived class can override the instance variable again.
- `private` declares instance variables that can be accessed only within the class expression. They cannot be overridden in a derived class and do not override declarations in the superclass.
- `inherit` declares instance variables that must be defined in the superclass or one of its ancestors (as `public`). An `inherit` declaration can be overridden in derived classes.
- `rename` is similar to `inherit`, but a new name is always used to access the value locally and the reference cannot be overridden. The new name can only be used within the class definition. A `rename` declaration accesses the superclass-defined variable value, even if the variable is overridden.

Each clause specifier can be used for any number of clauses in any order within a single `class*/names` expression.

The collection of instance variable declarations induces two sets of variables:

- The **internal instance variables** bound within the `class*` expression: this is the collection of all *instance-variable*s, *internal-instance-variable*s, and *inherited-variable*s. Along with the *variable*s for initialization variables, *this-variable*, and *super-init-variable*, all of these variables must be distinct.
- The **external instance variables** visible outside the `class*` expression (for derived classes or reference via `ivar`): this is the collection of all *instance-variable*s and *external-instance-variable*s. All of these variables must be distinct.

The same identifier can be used as an internal variable and an external variable, and it is possible to use the same identifier as internal and external variables for different bindings (as long as all internal variables are distinct and all external variables are distinct).

For `public`, `override`, and `private` instance variables, the *instance-var-initial-value-expr* expression provides a value for the variable in an object; when an initial value expression is not given, `(void)` is used. The process for evaluating initial value expressions is decribed in §6.3.3 and §6.4.

For each `public` instance variable defined in a class, the superclass (or any of its ancestors) must not contain a declaration for the variable. For each `override` instance variable, the superclass must already contain a declaration. These properties are verified when the `class*/name` expression is evaluated; if a `public` variable is found in the superclass or a `override` variable is not found, the `exn:object` exception is raised.

For `inherit` and `rename` instance variables, the *inherited-variable* or *external-inherited-variable* specifies a (public) instance variable from the superclass. This inheritance is verified when the `class*/name` expression is evaluated; if an inherited instance variable is not found in the superclass (or one of its ancestors), the `exn:object` exception is raised. The process that gives values to inherited variables is described in §6.4.

6.3.3 Initial Values

The initial value and `sequence` expressions for an instance of the class are evaluated in an environment that comprises

- the environment of the `class*/name` expression;
- all internal instance variables and initialization variables of the class declaration;
- *this-variable* (usually `this`, bound to the object itself); and
- *super-init-variable* (usually `super-init`, bound to an initialization procedure for the superclass).

The initial value and `sequence` expressions are evaluated each time an object of the class is created. The expressions are evaluated in the order in which they occur in the `class*/name` expression (across clauses and including `sequence` clauses).

Before an initial value expression has been evaluated, the value of the corresponding instance variable is initialized to undefined (see §4.1). Because all of the instance variables have mutually-recursive scopes, the undefined value of an instance variable can be exposed. The initialzation process is described in more detail in §6.4.

6.4 Creating Objects

The `make-object` procedure creates a new object:

(`make-object` *class init-v* ...)

The *init-v*s are passed as initialization arguments, bound to the initialization variables of *class* for the newly created object. If *class* is not a class, the `exn:application:type` exception is raised.

All instance variables in the newly created object are initially bound to the special undefined value (see §4.1). Initialization variables with default value expressions (and no provided value) are also initialized to undefined. After values are assigned to the initialization variables, default value expressions are evaluated for un-assigned initialization variables, and then the initial value expressions and `sequence` expressions are evaluated. The expressions are evaluated sequentially from left to right, but the enviornment for every expression includes all of the class's variables.

Sometime during the evaluation of these expressions, superclass-declared instance variables must be initialized (once) by calling the procedure bound to *super-init-variable* (usually `super-init`):

(*super-init-variable super-init-arg* ...)

The *super-init-arg*s are bound to the superclass's initialization variables, following the instance variable initialization process recursively until variables from all of the ancestor classes are initialized. The initialization procedure for the `object%` class accepts zero arguments.

Instance variables inherited from a superclass will not be initialized until the superclass's initialization procedure is invoked. Note that a variable declared with `inherit` may actually have a value before the superclass's initialization procedure is invoked, because inherited variables can be overridden. However, the value of a `rename` variable is always undefined before the superclass's initialization procedure is invoked.

It is an error to reach the end of initialization for any class in the hierarchy without invoking the superclasses initialization procedure; `exn:object` exception is raised in this case. If a superclass initialization procedure is invoked more than once, the `exn:object` exception is raised.

6.5 Instance Variable Access

In expressions within a class definition (e.g., within an instance variable's initial value expression), the internal variables declared in the class are all part of the environment. Thus, an internal instance variable is used directly to get a value, and `set!` is used to set an instance variable's value. However, `set!` can only be used on definition variables (`public` and `private`), not reference variables (`inherit` and `rename`).[3]

Instance variable values are accessed from outside an object with the `ivar/proc` procedure: `(ivar/proc` *object symbol*`)` extracts the value of the instance variable of *object* with the external name *symbol*.

Hidden instance variables (declared with `private` or `rename`) can never be accessed using `ivar/proc`, and internal names for exposed variables are not recognized by `ivar/proc`. If a specified instance variable cannot be found by `ivar/proc`, the `exn:object` exception is raised.

Instead of using the `ivar/proc` procedure directly, instance variable values are usually obtained with the `ivar` form:

`(ivar` *o name*`)`
⇒
`(ivar/proc` *o* `(quote` *name*`))`

The `send` macro is useful for invoking methods:

`(send` *o name arg* ···`)`
⇒
`((ivar` *o name*`)` *arg* ···`)`

6.5.1 Generic Procedures

A **generic procedure** takes an object and extracts an instance variable value from the object. Each generic procedure works on instances of a single class (including instances of classes derived from that class) or interface (i.e., instances of classes that implement the interface) and always extracts the value for the same instance variable.

Generic procedures are provided for efficiency. To extract the value of the same instance variable from multiple objects, it is usually more efficient to create a single generic procedure and use it each time than to use `ivar` each time. Generic procedures also provide run-time type checking, since a generic procedure only consumes objects of a particular class or interface.

The `make-generic/proc` procedure creates a generic procedure:

- `(make-generic/proc` *class symbol*`)` returns a generic procedure that consumes and instance of *class* (or an instance of a class derived from *class*) and returns the value of its *symbol* instance variable.

  If *class* or its superclasses does not contain an instance variable with the (external) name *symbol*, the `exn:object` exception is raised by `make-generic/proc`.

- `(make-generic/prof` *interface symbol*`)` returns a generic procedure that consumes an instance of any class that implements *interface* and returns the values of its *symbol* instance variable.

  If *interface* does not contain an instance variable with the name *symbol*, the `exn:object` exception is raised by `make-generic/proc`.

[3] This `set!` restriction avoids certain bad forms of variable aliasing.

If a generic procedure is applied to an object that is not an instance of the generic procedure's class or interface, the `exn:object` exception is raised.

The `make-generic` form expands to an application to `make-generic/proc`:

```
(make-generic ci name)
⇒
(make-generic/proc ci (quote name))
```

6.6 Object Utilities

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

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

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

`(object-class` *object*`)` returns the class of *object*.

`(is-a?` *v class*`)` returns `#t` if *v* is an instance of *class* (or of a class derived from *class*), `#f` otherwise.

`(is-a?` *v interface*`)` returns `#t` if *v* is an instance of a class that implements *interface*, `#f` otherwise.

`(subclass?` *v class*`)` returns `#t` if *v* is a class derived from (or equal to) *class*, `#f` otherwise.

`(implementation?` *v interface*`)` returns `#t` if *v* is a class that implements *interface*, `#f` otherwise.

`(interface-extension?` *v interface*`)` returns `#t` if *v* is an interface that extends *interface*, `#f` otherwise.

`(ivar-in-class?` *symbol class*`)` returns `#t` if *class* (or any of its ancestor classes) defines an instance variable with the (external) name *symbol*, `#f` otherwise.

`(ivar-in-interface?` *symbol interface*`)` returns `#t` if *interface* (or any of its ancestor interfaces) defines an instance variable with the name *symbol*, `#f` otherwise.

7. Units

MzScheme's **units** are used to organize a program into separately compilable and reusable components. A unit resembles a procedure in that both are first-class values that are used for abstraction. While procedures abstract over values in expressions, units abstract over names in collections of definitions. Just as a procedure is invoked to evaluate its expressions given actual arguments for its formal parameters, a unit is invoked to evaluate its definitions given actual references for its imported variables. Unlike a procedure, however, a unit's imported variables can be partially linked with the exported variables of another unit *prior to invocation*. Linking merges multiple units together into a single compound unit. The compound unit itself imports variables that will be propagated to unresolved imported variables in the linked units, and re-exports some variables from the linked units for further linking.

MzScheme supports two layers of units. The **core** unit system comprises the `unit`, `compound-unit`, and `invoke-unit` syntactic forms. These forms implement the basic mechanics of units for separate compilation and linking. While the semantics of units is most easily understood via the core forms, they are too verbose for specifying the interconnections between units in a large program. Therefore, a system of **units with signatures** is provided on top of the core forms, comprising the `define-signature`, `unit/sig`, `compound-unit/sig`, and `invoke-unit/sig` syntactic forms.

The core system is described first in §7.1. The signature system is roughly described in §7.2, and then described in detail in §7.3. Finally, gory details about mixing core and signed units are presented in §7.4.

7.1 Core Units

7.1.1 Creating Units

The `unit` form creates a unit:

```
(unit
   (import variable ...)
   (export exportage ...)
   unit-body-expr
   ...)

exportage is one of:
   variable
   (internal-variable external-variable)
```

The *variable*s in the `import` clause are bound within the *unit-body-expr* expressions. The variables for *exportages* in the `export` clause must be defined in the *unit-body-expr*s as described below; additional private variables can be defined as well. The imported variables cannot occur on the left-hand side of an assignment (i.e., a `set!` expression).

The first *exportage* form exports the binding defined as *variable* in the unit body using the external name *variable*. The second form exports the binding defined as *internal-variable* using the external name *external-*

variable. The external variables from an `export` clause must be distinct.

Each exported *variable* or *internal-variable* must be defined in a `define-values` expression as a *unit-body-expr*.[1] All identifers defined by the *unit-body-expr*s together with the *variable*s from the `import` clause must be distinct.

A *unit-body-expr* cannot reference a variable in the global namespace. However, MzScheme's built-in variables are implicitly imported into all units. Lexical bindings of built-in identifers can override the built-in value, but bindings in the top-level environment cannot. For example, assuming the variable `cons` is not lexically bound, the `cons` in a *unit-body-expr* denotes the expected Scheme pairing primitive, regardless of the current binding of `cons` in the top-level environment.

Examples

The unit defined below imports and exports no variables. Each time it is invoked, it prints and returns the current time in seconds:[2]

```
(define f1@
   (unit
     (import) (export x)
     (define x (current-seconds))
     (display x) (newline) x))
```

The expression below is syntactically invalid because `current-date` is not a built-in procedure:

```
(define f2-bad@
   (unit
     (import) (export x)
     (define x (current-date))
     (display x) (newline) x))
```

but the next expression is valid because the `unit` expression is in the scope of the `let`-bound variable:

```
(define f2@
   (let ([current-date current-seconds])
     (unit
       (import) (export x)
       (define x (current-date))
       (display x) (newline) x)))
```

The following units define two parts of an interactive phone book:

```
(define database@
  (unit (import show-message)
        (export insert lookup)
    (define table (list))
    (define insert
      (lambda (name info)
        (set! table (cons (cons name info) table))))
    (define lookup
      (lambda (name)
        (let ([data (assoc name table)])
```

[1] A *unit-body-expr* definition cannot shadow a syntax form or macro name. The detection of unit definitions is the same as for internal definitions (see §3.5.5), so the `define` and `define-struct` forms can be used for definitions.

[2] The "`@`" in the variable name "`f1@`" indicates (by convention) that its value is a unit.

```
            (if data
                (cdr data)
                (show-message "info not found")))))
      insert))

(define interface@
  (unit (import insert lookup make-window make-button)
        (export show-message)
    (define show-message
      (lambda (msg) ...))
    (define main-window
      ...)))
```

In this example, the `database@` unit implements the database-searching part of the program, and the `interface@` unit implements the graphical user interface. The `database@` unit exports `insert` and `lookup` procedures to be used by the graphical interface, while the `interface@` unit exports a `show-message` procedure to be used by the database (to handle errors). The `interface@` unit also imports variables that will be supplied by an platform-specific graphics toolbox.

7.1.2 Invoking Units

A unit is invoked using the `invoke-unit` form:

`(invoke-unit` *expr variable* ⋯`)`

The value of *expr* must be a unit. For each of the unit's imported variables, the `invoke-unit` expression must contain an *variable*. The binding for each *variable* in the context of the `invoke-unit` expression will be imported into the unit. More detailed information about linking is provided in the following section on compound units.

Invocation proceeds in two stages. First, invocation creates bindings for the unit's private and *exported* variables, and then links the unit's *imported* variables to bindings. All bindings are initialized to the undefined value. Second, invocation evaluates the unit's private definitions and expressions. The result of the last expression in the unit is the result of the `invoke-unit` expression. The unit's exported variable bindings are *not* accessible after the invocation.

Examples

These examples use the definitions from the earlier unit examples in §7.1.1.

The `f1@` unit is invoked with no imports:

```
(invoke-unit f1@) ; displays and returns the current time
```

Here is one way to invoke the `database@` unit:

```
(invoke-unit database@ display)
```

This invocation links the imported variable `message` in `database@` to the standard Scheme `display` procedure, sets up an empty database, and creates the procedures `insert` and `lookup` tied to this particular database. Since the last expression in the `database@` unit is `insert`, the `invoke-unit` expression returns the `insert` procedure (without binding any top-level variables). The fact that `insert` and `lookup` are exported is irrelevant; exports are only used for linking.

Invoking the `database@` unit directly in the above manner is actually useless. Although a program can insert information into the database, it cannot extract information since the `lookup` procedure is not accessible. The `database@` unit becomes useful when it is linked with another unit in a `compound-unit` expression.

7.1.3 Linking Units and Creating Compound Units

The `compound-unit` form links several units into one new compound unit. In the process, it matches imported variables in each sub-unit either with exported variables of other sub-units or with its own imported variables:

```
(compound-unit
   (import variable ···)
   (link [tag (sub-unit-expr linkage ···)] ···)
   (export (tag exportage ···) ···))

linkage is one of:
   variable
   (tag variable)
   (tag variable ···)

exportage is one of:
   variable
   (internal-variable external-variable)

tag is:
   identifier
```

The three parts of a `compound-unit` expression have the following roles:

- The `import` clause imports variables into the compound unit. These imported variables are used as imports to the compound unit's sub-units.
- The `link` clause specifies how the compound unit is created from sub-units. A unique *tag* is associated with each sub-unit, which is specified using an arbitrary expression. Following the unit expression, each *linkage* specifies a variable using the *variable* form or the `(`*tag variable*`)` form. In the former case, the *variable* must occur in the `import` clause of the `compound-unit` expression; in the latter case, the *tag* must be defined in the same `compound-unit` expression. The `(`*tag variable* ···`)` form is a shorthand for multiple adjacent clauses of the second form with the same *tag*.
- The `export` clause re-exports variables from the compound unit that were originally exported from the sub-units. The *tag* part of each `export` sub-clause specifies the sub-unit from which the re-exported variable is drawn. The *exportage*s specify the names of variables exported by the sub-unit to be re-exported.

  As in the `export` clause of the `unit` form, a re-exported variable can be renamed for external references using the `(`*internal-variable external-variable*`)` form. The *internal-variable* is used as the name exported by the sub-unit, and *external-variable* is the name visible outside the compound unit.

The evaluation of a `compound-unit` expression starts with the evaluation of the `link` clause's unit expressions (in sequence). For each sub-unit, the number of variables it imports must match the number of *linkage* specifications that are provided, and each *linkage* specification is matched to an imported variable by position. Each sub-unit must also export those variables that are specified by the `link` and `export` clauses. If, for any sub-unit, the number of imported variables does not agree with the number of linkages provided, the `exn:unit` exception is raised. If an expected exported variable is missing from a sub-unit for linking to

another sub-unit, the `exn:unit` exception is raised. If an expected export variable is missing for re-export, the `exn:unit` exception is raised.

The invocation of a compound unit proceeds in two phases to invoke the sub-units. In the first phase, the compound unit resolves the imported variables of sub-units with the bindings provided for the compound unit's imports and new bindings created for sub-unit exports. In the second phase, the internal definitions and expressions of the sub-units are evaluated sequentially according to the order of the sub-units in the `link` clause. The result of invoking a compound unit is the result from the invocation of the last sub-unit.

Examples

These examples use the definitions from the earlier unit examples in §7.1.1.

The following `compound-unit` expression creates a useless wrapping around the unit bound to `f1@`:

```
(define f3@
  (compound-unit
    (import)
    (link [A (f1@)])
    (export (A (x A:x)))))
```

The only difference between `f1@` and `f3@` is that `f1@` exports a variable named `x`, while `f3@` exports a variable named `A:x`.

The following example shows how the `database@` and `interface@` units are linked together with a graphical toolbox unit `Graphics` to produce a single, fully-linked compound unit for the interactive phone book program.

```
(define program@
  (compound-unit
    (import)
    (link (GRAPHICS (graphics@))
          (DATABASE (database@ (INTERFACE show-message)))
          (INTERFACE (interface@ (DATABASE insert lookup)
                                 (GRAPHICS make-window make-button))))
          (export)))
```

This phone book program is executed with `(invoke-unit program@)`. If `(invoke-unit program@)` is evaluated a second time, then a new, independent database and window are created.

7.1.4 Unit Utilities

`(unit?` v`)` returns `#t` if v is a unit or `#f` otherwise.

7.2 Units with Signatures Overview

The unit syntax presented in §7.1 poses a serious notational problem: each variable that is imported or exported must be separately enumerated in many `import`, `export`, and `link` clauses. Consider the phone book program example from §7.1.3: a realistic `graphics@` unit would contain many more procedures than `make-window` and `make-button`, and it would be unreasonable to enumerate the entire graphics toolbox in every client module. Future extensions to the graphics library are likely, and while the program must certainly be re-compiled to take advantage of the changes, the programmer should not be required to change the program text in every place that the graphics library is used.

This problem is solved by separating the specification of a unit's **signature** (or "interface") from its implementation. A unit signature is basically a list of variable names. A signature can be used in an import clause, an export clause, a link clause, or an invocation expression to import or link a set of variables at once. Signatures clarify the connections between units, prevent mis-orderings in the specification of imported variables, and provide better error messages when an illegal linkage is specified.

Signatures are used to create **units with signatures**, a.k.a. **signed units**. Signatures and signed units are used together to create **signed compound units**. As in the core system, a signed compound unit is itself a signed unit.

Signed units are first-class values, just like their counterparts in the core system. A signature is not a value. However, signature information is bundled into each signed unit value so that signature-based checks can be performed at run time (when signed units are linked and invoked).

Along with its signature information, a signed unit includes a primitive unit from the core system that implements the signed unit. This underlying unit can be extracted for mixed-mode programs using both signed and unsigned units. More importantly, the semantics of signed units is the same as the semantics for regular units; the additional syntax only serves to specify signatures and to check signatures for linking.

7.2.1 Importing and Exporting with Signatures

The `unit/sig` form creates a signed unit in the same way that the `unit` form creates a unit in the core system. The only difference between these forms is that signatures are used to specify the imports and exports of a signed unit.

In the primitive `unit` form, the `import` clause only determines the number of variables that will be imported when the unit is linked; there are no explicitly declared connections between the import variables. In contrast, a `unit/sig` form's `import` clause does not specify individual variables; instead, it specifies the signatures of units that will provide its imported variables, and all of the variables in each signature are imported. The ordered collection of signatures used for importing in a signed unit is the signed unit's **import signature**.

Although the collection of variables to be exported from a `unit/sig` expression is specified by a signature rather than an immediate sequence of variables,[3] variables are exported in a `unit/sig` form in the same way as in the `unit` form. The **export signature** of a signed unit is the collection of names exported by the unit.

Example:

```
(define-signature arithmetic^ (add subtract multiply divide power))
(define-signature calculus^ (integrate))
(define-signature graphics^ (add-pixel remove-pixel))
(define-signature gravity^ (go))
(define gravity@
  (unit/sig gravity^ (import arithmetic^ calculus^ graphics^)
    (define go (lambda (start-pos) ... subtract ... add-pixel ...))))
```

In this program fragment, the signed unit `gravity@` imports a collection of arithmetic procedures, a collection of calculus procedures, and a collection of graphics procedures. The arithmetic collection will be provided through a signed unit that matches the `arithmetic^` (export) signature, while the graphics collection will be provided through a signed unit that matches the `graphics^` (export) signature. The `gravity@` signed unit itself has the export signature `gravity^`.

Suppose that the procedures in `graphics^` were named `add` and `remove` rather than `add-pixel` and `remove-pixel`. In this case, the `gravity@` unit cannot import both the `arithmetic^` and `graphics^` signa-

[3] Of course, a signature *can* be specified as an immediate signature.

tures as above because the name `add` would be ambiguous in the unit body. To solve this naming problem, the imports of a signed unit can be distinguished by providing prefix tags:

```
(define-signature graphics^ (add remove))
(define gravity@
  (unit/sig gravity^ (import (a : arithmetic^) (c : calculus^) (g : graphics^))
    (define go (lambda (start-pos) ... a:subtract ... g:add ...))))
```

Details for the syntax of signatures are in §7.3.1. The full `unit/sig` syntax is described in §7.3.2.

7.2.2 Linking with Signatures

The `compound-unit/sig` form links signed units into a signed compound unit in the same way that the `compound-unit` form links primitive units. In the `compound-unit/sig` form, signatures are used for importing just as in `unit/sig` (except that all import signatures must have a tag), but the use of signatures for linking and exporting is more complex.

Within a `compound-unit/sig` expression, each unit to be linked is represented by a tag. Each tag is followed by a signature and an expression. A tag's expression evaluates (at link-time) to a signed unit for linking. The export signature of this unit must **satisfy** the tag's signature. "Satisfy" *does not* mean "match exactly"; satisfaction requires that the unit exports at least the variables specified in the tag's signature, but the unit may actually export additional variables. Those additional variables are ignored for linking and are effectively hidden by the compound unit.

To specify the compound unit's linkage, an entire unit is provided (via its tag) for each import of each linked unit. The number of units provided by a linkage must match the number of signatures imported by the linked unit, and the tag signature for each provided unit must match (exactly) the corresponding imported signature.

The following example shows the linking of an arithmetic unit, a calculus unit, a graphics unit, and a gravity modelling unit:

```
(define-signature arithmetic^ (add subtract multiply divide power))
(define-signature calculus^ (integrate))
(define-signature graphics^ (add-pixel remove-pixel))
(define-signature gravity^ (go))
(define arithmetic@ (unit/sig arithmetic^ (import) ...))
(define calculus@ (unit/sig calculus^ (import arithmetic^) ...))
(define graphics@ (unit/sig graphics^ (import) ...))
(define gravity@ (unit/sig gravity^ (import arithmetic^ calculus^ graphics^) ...))
(define model@
  (compound-unit/sig
    (import)
    (link (ARITHMETIC : arithmetic^ (arithmetic@))
          (CALCULUS : calculus^ (calculus@ ARITHMETIC)))
          (GRAPHICS : graphics^ (graphics@))
          (GRAVITY : gravity^ (gravity@ ARITHMETIC CALCULUS GRAPHICS)))
    (export (var (GRAVITY go)))))
```

In the `compound-unit/sig` expression for `model@`, all link-time signature checks succeed since, for example, `arithmetic@` does indeed implement `arithmetic^` and `gravity@` does indeed import units with the `arithmetic^`, `calculus^`, and `graphics^` signatures.

The export signature of a signed compound unit is implicitly specified by the `export` clause. In the above example, the `model@` compound unit exports a `go` variable, so its export signature is the same as `gravity^`.

More forms for exporting are described in §7.2.4.

7.2.3 Restricting Signatures

As explained in §7.2.2, the signature checking for a linkage requires that a provided signature *exactly* matches the corresponding import signature. At first glance, this requirement appears to be overly stricty; it might seem that the provided signature need only *satisfy* the imported signature. The reason for requiring an exact match at linkages is that a `compound-unit/sig` expression is expanded into a `compound-unit` expression. Thus, the number and order of the variables used for linking must be fully known at compile-time.

The exact-match requirement does not pose any obstacle as long as a unit is linked into only one other unit. In this case, the signature specified with the unit's tag can be contrived to match the importing signature. However, a single unit may need to be linked into different units, each of which may use different importing signatures. In this case, the tag's signature must be "bigger" than both of the uses, and a **restricting signature** is explicitly provided at each linkage. The tag must satisfy every restricting signature (this is a syntactic check), and each restricting signature must exactly match the importing signature (this is a run-time check).

In the example from §7.2.2, both `calculus@` and `gravity@` import numerical procedures, so both import the `arithmetic^` signature. However, `calculus@` does not actually need the `power` procedure to implement `integrate`; therefore, `calculus@` could be as effecively implemented in the following way:

```
(define-signature simple-arithmetic^ (add subtract multiply divide))
(define calculus@ (unit/sig calculus^ (import simple-arithmetic^) ...))
```

Now, the old `compound-unit/sig` expression for `model@` no longer works. Although the old expression is still syntactically correct, link-time signature checking will discover that `calculus@` expects an import matching the signature `simple-arithmetic^` but it was provided a linkage with the signature `arithmetic^`. On the other hand, changing the signature associated with `ARITHMETIC` to `simple-arithmetic^` would cause a link-time error for the linkage to `gravity@`, since it imports the `arithmetic^` signature.

The solution is to restrict the signature of `ARITHMETIC` in the linkage for `CALCULUS`:

```
(define model@
  (compound-unit/sig
    (import)
    (link (ARITHMETIC : arithmetic^ (arithmetic@))
          (CALCULUS : calculus^ (calculus@ (ARITHMETIC : simple-arithmetic^))))
          (GRAPHICS : graphics^ (graphics@))
          (GRAVITY : gravity^ (gravity@ ARITHMETIC CALCULUS GRAPHICS)))
    (export (var (GRAVITY go)))))
```

A syntactic check will ensure that `arithmetic^` satisfies `simple-arithmetic^` (i.e., `arithmetic^` contains at least the variables of `simple-arithmetic^`). Now, all link-time signature checks will succeed, as well.

7.2.4 Embedded Units

Signed compound units can re-export variables from linked units in the same way that core compound units can re-export variables. The difference in this case is that the collection of variables that are re-exported determines an export signature for the compound unit. Using certain export forms, such as the `open` form instead of the `var` form (see §7.3.3), makes it easier to export a number of variables at once, but these are simply shorthand notations.

Signed compound units can also export entire units as well as variables. Such an exported unit is an

embedded unit of the compound unit. Extending the example from §7.2.3, the entire `gravity@` unit can be exported from `model@` using the `unit` export form:

```
(define model@
  (compound-unit/sig
    (import)
    (link (ARITHMETIC : arithmetic^ (arithmetic@))
          (CALCULUS : calculus^ (calculus@ (ARITHMETIC : simple-arithmetic^))))
          (GRAPHICS : graphics^ (graphics@))
          (GRAVITY : gravity^ (gravity@ ARITHMETIC GRAPHICS)))
    (export (unit GRAVITY))))
```

The export signature of `model@` no longer matches `gravity^`. When a compound unit exports an embedded unit, the export signature of the compound unit has a sub-signature that corresponds to the full export signature of the embedded unit. The following signature, `model^`, is the export signature for the revised `model@`:

```
(define-signature model^ ((unit GRAVITY : gravity^)))
```

The signature `model^` matches the (implicit) export signature of `model@` since it contains a sub-signature named `GRAVITY`—matching the tag used to export the `gravity@` unit—that matches the export signature of `gravity@`.

The export form `(unit GRAVITY)` does not export any variable other than `gravity@`'s `go`, but the "unitness" of `gravity@` is intact. The embedded `GRAVITY` unit is now available for linking when `model@` is linked to other units.

For example:

```
(define tester@ (unit/sig () (import gravity^) (go 0)))
(define test-program@
  (compound-unit/sig
     (import)
     (link (MODEL : model^ (model@))
           (TESTER : () (tester@ (MODEL GRAVITY))))
     (export)))
```

The embedded `GRAVITY` unit is linked as an import into the `tester@` unit by using the path `(MODEL GRAVITY)`.

7.3 Units with Signatures

7.3.1 Signatures

A *signature* is either a signature description or a bound signature identifier:

(*sig-element* ···)
signature-identifier

sig-element is one of:
variable
(`struct` *base-identifier* (*field-identifier* ···) *omission* ···)
(`open` *signature*)
(`unit` *identifier* `:` *signature*)

omission is one of:
`-selectors`
`-setters`
`(-` *variable*`)`

Together, the element descriptions determine the set of elements that compose the signature:

- The simple *variable* form adds a variable name to the new signature.
- The `struct` form expands into the list of variable names generated by a `define-struct` expression with the given *base-identifier* and *field-identifier*s.

  The actual structure type can contain additional fields; if a field identifier is omitted, the corresponding selector and setter names are not added to the signature. Optional *omission* specifications can omit other kinds of names: `-selectors` omits all field selector variables. `-setters` omits all field setter variables, and `(-` *variable*`)` omits a specific generated *variable*.
- The `open` form copies all of the elements of another signature into the new signature description.
- The `unit` form creates a sub-signature within the new signature. A signature that includes a `unit` clause corresponds to a signed compound unit that exports an embedded unit. (Embedded units are described in §7.2.4 and §7.3.3.)

The names of all elements in a signature must be distinct.[4] Two signatures **match** when they contain the same element names, and when a name in both signatures is either a variable name in both signatures or a sub-signature name in both signatures such that the sub-signatures match. The order of elements within a signature is not important. A source signature **satisfies** a destination signature when the source signature has all of the elements of the destination signature, but the source signature may have additional elements.

The `define-signature` form binds a signature to an identifier:

`(define-signature` *signature-identifier signature*`)`

The `let-signature` form binds a signature to an identifier within a body of expressions:

`(let-signature` *identifier signature body-expr* ...[1]`)`

Internal `define-signature` expressions are transformed into `let-signature` expressions.

7.3.1.1 Flattening Signatures

For various purposes, signatures must be flattened into a linear sequence of variables:

- All variable name elements of the signature are included in the flattened signature.
- For each sub-signature element named *s*, the sub-signature is flattened, and then each variable name in the flattened sub-signature is prefixed with *s*`:` and included in the flattened signature.

7.3.2 Signed Units

The `unit/sig` form creates a signed unit:

[4] Element names are compared using the printed form of the name. This is different from any other syntactic form, where variable names are compared as symbols. This distinction is relevant only when source code is generated within Scheme rather than read from a text source.

```
(unit/sig signature
  (import import-element ···)
  renames
  signed-unit-body-expr
  ···)
```

import-element is one of:
- *signature*
- (*identifier* : *signature*)

renames is either empty or:
- (**rename** (*internal-variable signature-variable*) ···)

signed-unit-body-expr is one of:
- (**include** *filename*)
- *unit-body-expr*

The *signature* immediately following **unit/sig** specifies the export signature of the signed unit. This signature cannot contain sub-signatures. Each element of the signature must have a corresponding variable definition in one of the *unit-body-expr*s, modulo the optional **rename** clause. If the **rename** clause is present, it maps *internal-variable*s defined in the *unit-body-expr*s to *signature-variable*s in the export signature.

The *import-element*s specify imports for the signed unit. The names bound within the *signed-unit-body-expr*s to imported bindings are constructed by flattening the signatures according to the algorithm in §7.3.1.1:

- For each *import-element* using the *signature* form, the variables in the flattened signature are bound in the *signed-unit-body-expr*s.
- For each *import-element* using the (*identifier* : *signature*) form, the variables in the flattened signature are prefixed with *identifier*: and the prefixed variables are bound in the *signed-unit-body-expr*s.

Each *signed-unit-body-expr* is either a regular expression or an **include** form. If a *signed-unit-body-expr* has the form (**include** *filename*), the content of the file named by *filename* is textually substituted into the **unit/sig** body in the place of the **include** clause. In particular, a single **include** clause can be replaced by any number of expressions from the included file.

7.3.3 Signed Compound Units

The **compound-unit/sig** form links mulitple signed units into a new signed compound unit:

```
(compound-unit/sig
  (import (tag : signature) ···)
  (link (tag : signature (expr unit-path ···)) ···)
  (export export-element ···))
```

unit-path is one of:
- *simple-unit-path*
- (*simple-unit-path* : *signature*)

simple-unit-path is one of:
- *tag*
- (*tag identifier* ···)

export-element is one of:
  `(var (`*simple-unit-path variable*`))`
  `(var (`*simple-unit-path variable*`)` *external-variable*`)`
  `(open` *unit-path*`)`
  `(unit` *unit-path*`)`
  `(unit` *unit-path variable*`)`
tag is:
  identifier

The `import` clause is similar to the `import` clause of a `unit/sig` expression, except that all imported signatures must be given a *tag* identifier.

The `link` clause of a `compound-unit/sig` expression is different from the `link` clause of a `compound-unit` expression in two important aspects:

- Each sub-unit tag is followed by a *signature*. This signature corresponds to the export signature of the signed unit that will be associated with the tag.
- The linkage specification consists of references to entire signed units rather than to individual variables that are exported by units. A referencing *unit-path* has one of four forms:
  - The *tag* form references an imported unit or another sub-unit.
  - The `(`*tag* `:` *signature*`)` form references an imported unit or another sub-unit, and then restricts the effective signature of the referenced unit to *signature*.
  - The `(`*tag identifier* $\cdots$`)` references an embedded unit within a signed compound unit. The signature for the *tag* unit must contain a sub-signature that corresponds to the embedded unit, where the sub-signature's name is the initial *identifier*. Additional *identifier*s trace a path into nested sub-signatures to a final embedded unit. The degenerate `(`*tag*`)` form is equivalent to *tag*.
  - The `((`*tag identifier* $\cdots$`) :` *signature*`)` form is like the `(`*tag identifier* $\cdots$`)` form except the effective signature of the referenced unit is restricted to *signature*.

The `export` clause determines which variables in the sub-units are re-exported and implicitly determines the export signature of the new compound unit. A signed compound unit can export both individual variables and entire signed units. When an entire signed unit is exported, it becomes an embedded unit of the resulting compound unit.

There are five different forms for specifying exports:

- The `(var (`*unit-path variable*`))` form exports *variable* from the unit referenced by *unit-path*. The export signature for the signed compound unit includes a *variable* element.
- The `(var (`*unit-path variable*`)` *external-variable*`)` form exports *variable* from the unit referenced by *unit-path*. The export signature for the signed compound unit includes an *external-variable* element.
- The `(open` *unit-path*`)` form exports variables and embedded units from the referenced unit. The collection of variables that are actually exported depends on the **effective signature** of the referenced unit:
  - If *unit-path* includes a signature restriction, then only elements from the restricting signature are exported.
  - Otherwise, if the referenced unit is an embedded unit, then only the elements from the associated sub-signature are exported.
  - Otherwise, *unit-path* is just *tag*; in this case, only elements from the signature associated with the *tag* are exported.

In all cases, the export signature for the signed compound unit includes a copy of each element from the effective signature.

- The `(unit` *unit-path*`)` form exports the referenced unit as an embedded unit. The export signature for the signed compound unit includes a sub-signature corresponding to the effective signature from *unit-path*. The name of the sub-signature in the compound unit's export signature depends on *unit-path*:
  - If *unit-path* refers to a tagged import or a sub-unit, then the tag is used for the sub-signature name.
  - Otherwise, the referenced sub-unit was an embedded unit, and the original name for the associated sub-signature is re-used for the export signature's sub-signature.
- The `(unit` *unit-path identifier*`)` form exports an embedded unit like `(unit` *unit-path*`)` form, but *identifier* is used for the name of the sub-signature in the compound unit's export signature.

The collection of names exported by a compound unit must form a legal signature. This means that all exported names must be distinct.

Run-time checks insure that all `link` clause *expr*s evaluate to a signed unit, and that all linkages match according to the specified signatures:

- If an *expr* evaluates to anything other than a signed unit, the `exn:unit` exception is raised.
- If the export signature for a signed unit does not satisfy the signature specified with its tag, the `exn:unit:signature` exception is raised.
- If the number of units specified in a linkage does not match the number imported by a linking unit, the `exn:unit` exception is raised.
- If the (effective) signature of a provided unit does not match the corresponding import signature, then the `exn:unit` exception is raised.

7.3.4 Invoking Signed Units

Signed units are invoked using the `invoke-unit/sig` form:

`(invoke-unit/sig` *expr invoke-linkage* ···`)`

invoke-linkage is one of:
signature
`(`*identifier* `:` *signature*`)`

If the invoked unit requires no imports, the `invoke-unit/sig` form is used in the same way as `invoke-unit`. Otherwise, the *invoke-linkage* signatures must match the import signatures of the signed unit to be invoked. If the signatures match, then variables in the environment of the `invoke-unit/sig` expression are used for immediate linking; the variables used for linking are the ones with names corresponding to the flattened signatures. The signature flattening algorithm is specified in §7.3.1.1; when the `(`*identifier* `:` *signature*`)` form is used, *identifier*`:` is prefixed onto each variable name in the flattened signature and the prefixed name is used.

7.4 Mixing Core and Signed Units

7.4.1 Extracting a Primitive Unit from a Signed Unit

The procedure `unit/sig->unit` extracts and returns the primitive unit from a signed unit.

The names exported by the primitive unit correspond to the flattened export signature of the signed unit; see §7.3.1.1 for the flattening algorithm.

The number of import variables for the primitive unit matches the total number of variables in the flattened forms of the signed unit's import signatures. The order of import variables is as follows:

- All of the variables for a single import signature are grouped together, and the relative order of these groups follows the order of the import signatures.
- Within an import signature:
  - variable names are ordered according to `string<?`;
  - all names from sub-signatures follow the variable names;
  - names from a signle sub-signature are grouped together and ordered within the sub-signature group following this algorithm recursively; and
  - the sub-signatures are ordered among themselves using `string<?` on the sub-signature names.

7.4.2 Adding a Signature to Primitive Units

The `unit->unit/sig` syntactic form wraps a primitive unit with import and export signatures:

`(unit->unit/sig` *expr* `(`*signature* ···`)` *signature*`)`

The last *signature* is used for the export signature and the other *signature*s specify the import signatures. If *expr* does not evaluate to a unit or the unit does not match the signature, no error is reported until the primitive linker discovers the problem.

7.4.3 Expanding Signed Unit Expressions

The `unit/sig`, `compound-unit/sig`, and `invoke-unit/sig` forms expand into expressions using the `unit`, `compound-unit`, and `invoke-unit` forms, respectively. The exapnsion may also use `global-defined-signature` rather than inlining a signature that is bound to a global identifier.

A signed unit value is represented by a `unit-with-signature` structure with the following fields:

- `unit` — the primitive unit implementing the signed unit's content
- `imports` — the import signatures, represented as a list of pairs, where each pair consists of
  - a tag symbol, used for error reporting; and
  - an "exploded signature"; an exploded signature is a vector of signature elements, where each element is either
    - a symbol, representing a variable in the signature; or
    - a pair consisting of a symbol and an exploded signature, representing a name sub-signature.
- `exports` — the export signature, represented as an exploded signature

To perform the signature checking needed by `compound-unit/sig`, MzScheme provides two procedures:

- `(verify-signature-match` *where exact? dest-context dest-sig src-context src-sig*`)` raises an exception unless the exploded signatures *dest-sig* and *src-sig* match. If *exact?* is `#f`, then *src-sig* need only satisfy *dest-sig*, otherwise the signatures must match exactly. The *where* symbol and *dest-context* and *src-context* strings are used for generating an error message string: *where* is used as the name of the signalling procedure and *dest-context* and *src-context* are used as the respective signature names.

If the match succeeds, void is returned. If the match fails, the `exn:unit` exception is raised for one of the following reasons:

- The signatures fail to match because *src-sig* is missing an element.
- The signatures fail to match because *src-sig* contains an extra element.
- The signatures fail to match because *src-dest* and *src-sig* contain the same element name but for different element types.

- `(verify-linkage-signature-match` *where tags units export-sigs linking-sigs*`)` performs all of the run-time signature checking required by a `compound-unit/sig` or `invoke-unit/sig` expression. The *where* symbol is used for error reporting. The *tags* argument is a list of tag symbols, and the *units* argument is the corresponding list of candidate signed unit values. (The procedure will check whether these values are actually signed unit values.)

  The *export-sigs* list contains one exploded signature for each tag; these correspond to the tag signatures provided in the original `compound-unit/sig` expression. The *linking-sigs* list contains a list of named exploded signatures for each tag (where a "named signature" is a pair consisting of a name symbol and an exploded signature); every tag's list corresponds to the signatures that were specified or inferred for the tag's linkage specification in the original `compound-unit/sig` expression. The names on the linking signatures are used for error messages.

  If all linking checks succeed, void is returned. If any check fails, the `exn:unit` exception is raised for one of the following reasons:

  - A value in the *units* list is not a signed unit.
  - The number of import signatures associated with a unit does not agree with the number of linking signatures specified by the corresponding list in *linking-sigs*.
  - A linking signature does not exactly match the signature expected by an importing unit.

8. Exceptions and Control Flow

8.1 Exceptions

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

- `(raise` *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**.

- `(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 §9.4.1.9 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 §9.4.1.9) and then escapes by calling the current error escape handler (see `error-escape-handler` in §9.4.1.9). If an exception is raised while an exeception 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* `...`[1]`)` is a syntactic form that evaluates the *expr* body, installing a new exception handler before evaluating the *expr*s 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 §9.4).

  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* precedures, the result of the entire `with-handlers` expression is the return value of the handler.

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

  Before any predicate or handler procedure is invoked, the continuation of the entire `with-handlers` expression is restored. 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.

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

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

```
(define div-w-inf
  (lambda (n d)
    (with-handlers ([exn:application:math:zero?
                     (lambda (exn) +inf.0)])
      (/ n d))))
```

8.1.1 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 §9.4.1.9).

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 excpetion 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: *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 variable at run-time
    fields: `id` — the unbound variable's global identifier (type: *symbol*)
    - `keyword` : attempt to change the binding of a global keyword
  - `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
  - `else` : fall-through in `cond` or `case`
  - `struct` : the supertype expression in a `struct` form returned a value that was not a structure type value
  - `object` : object-, class-, or interface-specific error
  - `unit` : unit- or unit/sig-specific error
  - `syntax` : syntax error, but not a `read` error

fields: `expr` — illegal expression (or `#f` if unknown) (type: *S-expression*)

- `read` : `read` parsing error
  fields: `port` — port being read (type: *input-port*)
  - `eof` : unexpected end-of-file
- `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
    - `user` : user-defined input port returned a non-character from the character-getting procedure
  - `filesystem` : illegal pathname or error manipulating a filesystem object
    fields: `pathname` — file or directory pathname (type: *path*)
  - `tcp` : TCP errors
- `thread` : raised by `call-with-custodian`
- `misc` : low-level or MzScheme-specific error
  - `unsupported` : unsupported feature
  - `user-break` : asynchronous thread break
  - `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.

8.2 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 concatentaing `"error: "` with the string form of *symbol*.
- `(error` *msg-string* *v* ···`)` creates a message string by concatenating *msg-string* with string versions of the *v*s (as produced by the current error value conversion handler; see §9.4.1.9). 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.

8.2.1 Application Type Errors

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

8.2.2 Application Mismatch Errors

`(raise-mismatch-error` *name-symbol message-string v*`)` creates an `exn:application:mismatch` value and `raise`s it as an exception. The *name-symbol* is used as the source procedure's name in the error message. The

message-string is the error message. The *v* argument is the improper argument received by the procedure. The printed form of *v* is appended to *message-message* (using the error value conversion handler; see §9.4.1.9).

8.2.3 Syntax Errors

`(raise-syntax-error` *name-symbol message-string* [*expr sub-expr*]`)` creates an `exn:syntax` value and `raise`s it as an exception. Macros use this procedure to report macro syntax errors. The *name-symbol* argument is used as the source syntactic form's name in the error message. The *message-string* is used as the main body of the error message. The optional *expr* argument is the erroneous source S-expression. The optional *sub-expr* argument is an S-expression within *expr* that more precisely locates the error.

8.2.4 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, units, classes, and interfaces.

`(inferred-name` *v*`)` returns a symbol for the name inferred for *v* if *v* has a name, `#f` otherwise. The argument *v* can be any value. When *v* is a primitive, `inferred-name` returns the same name as `primitive-name` (see §4.10.2).

8.3 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* $\cdots^1$`)`
⇒
`(call/cc (lambda (`*k*`)` *expr* $\cdots^1$`))`

A continuation can only be invoked from the thread (see §9.1) in which it was captured. Multiple return values can be passed to a continuation (see Chapter 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.

8.4 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. calls more deeply nested *pre-thunk*s, then
2. jumps to the destination continuation, then
3. continues with the context of the `dynmaic-wind` call.

Normally, the third part of this continuation is never reached, due to the jump in the second part. However, the third 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* calls less deeply nested *post-thunk*s, them jumps to the destination continuation, then continues from the `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-cancelled)))))
; ⇒ 'cancel-cancelled
```

8.5 Continuation Marks

To evaluate a subexpression, MzScheme creates a continuation for the subexpression 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 its current continuation. Continuation marks support debuggers and other program-tracing facilities, because 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's first frame contains a mark m for k, then the mark list for C is `(cons` m l_0`)`, where l_0 is the mark list for k in C_0.
- If C's first frame does not contain a mark keyed by k, then the mark list for C is the mark list for C_0.

The mark list for the empty continuation is `null` for all keys.

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 `current-continuation-marks` procedure extracts the complete set of continuation marks from the current continuation:

- `(current-continuation-marks` *key-v*`)` returns a newly-created (opaque) value containing the set of continuation marks for all keys in the current continuation.

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*`)` 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`.
- `(continuation-mark-set?` *v*`)` returns `#t` if *v* is a mark set created by `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.

8.6 Breaks

A **break** is an asynchonous exception, usually triggered by an external source controlled by the user. A break exception can only occur in a thread while breaks are allowed by the `break-enabled` parameter (see §9.4.1.10). When a break is detected, the `exn:misc:user-break` exception is raised.

A break is triggered when the `break-thread` procedure is applied to a thread. An `exn:misc:user-break` is raised in the destination thread sometime afterwards; if breaking is disabled when `break-thread` is called, the break is suspended until breaking is again enabled for the thread.

When `break-thread` is applied to 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 transfering it to the nested thread.

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 (indepedent 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.

8.7 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 §9.4.1.9).

An error escape handler cannot invoke a full continuation that was created prior to the exception, but it *can* invoke an escape continuation (see §8.3).

The error escape handler is normally called directly by an exception handler. To escape from a run-time error, use `raise` (see §8.1) or `error` (see §8.2) 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.

9. Threads and Namespaces

9.1 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 §9.4), 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).

When a thread is created, it is placed into the management of the current custodian (See §9.5). A thread that has not terminated can be "garbage collected" only if it is unreachable and blocked on an unreachable semaphore.[1]

9.1.1 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 §8.6).

`(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 §9.5) does not manage *thread* (and none of its subordinates manages *thread*), the `exn:misc` exception

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

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 §8.6).

`(call-in-nested-thread` *thunk* [*custodian*]`)` creates a nested thread managed by *custodian* to execute *thunk*. 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 thead 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 a break is queued for the original thread with `break-thread`, the break is redirected to the nested thread. If the original thread is killed before *thunk* returns, a break is queued for the nested thread.

9.2 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. MzScheme's semaphores have the usual single-threaded access 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.

- `(semaphore-wait/enable-break` *sema*`)` is like `semaphore-wait`, but breaking is enabled (see §8.6) 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:misc:user-break` exception is raised, but not both.

9.3 Global Variable Namespaces

MzScheme supports multiple global variable **namespaces**. 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 §9.4.1.6).

The current namespace is used by `eval`, `load`, `compile`, and `expand-defmacro`.[2] Once an expression is `eval`ed or `compile`d, the global variable references in the compiled 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
```

(`make-namespace` *flag-symbol* ···) creates a new namespace; the *flag-symbol*s are options that determine the kinds of global names that are initially bound in the new namespace. Any number of *flag-symbol*s can be specified, where each *flag-symbol* is one of the following symbols:

- `'keywords` — keywords are enforced in the new namespace
- `'no-keywords` — keywords are not enforced in the new namespace
- `'call/cc=call/ec` — `call/cc` captures an escape-only continuation in the new namespace
- `'call/cc!=call/ec` — `call/cc` captures a full continuation in the new namespace
- `'hash-percent-syntax` — only syntactic forms prefixed with `#%` are present in the new namespace
- `'all-syntax` — all MzScheme syntactic forms are present in the new namespace
- `'empty` — no names are initially bound in the new namespace

Applications embedding MzScheme may extend this list of flags. (MrEd adds the `'mred` flag for installing the built-in MrEd classes and procedures.) If `'empty` is provided, all other provided flags are ignored. Otherwise, if two conflicting flags are provided, the latter flag takes precedence. If any other value or symbol is provided as a *flag-symbol*, the `exn:application:type` exception is raised. The default settings are built into the executable running MzScheme.

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

9.3.1 Global Names

(`defined?` *symbol*) returns `#t` if a global variable is defined with the name *symbol* in the current namespace, `#f` otherwise.

(`undefine` *symbol*) causes *symbol* to be undefined in the current namespace if it was defined previously.

(`global-defined-value` *symbol*) returns the value of the global variable named by the *symbol* in the current namespace. If *symbol* is undefined, the `exn:variable` exception is raised.

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

`(global-defined-value` *symbol v*`)` sets the value of *symbol* in the current namespace, defining *symbol* if it is not already defined.

`(make-global-value-list)` returns an association list that pairs each globally-defined symbol with its current value from the current namespace.

`(built-in-name` *symbol*`)` determines whether *symbol* is a built-in variable. If the *symbol* can be used within a unit as a built-in variable, then `built-in-name` returns a keyword-form variable symbol that accesses the built-in binding. Otherwise, `#f` is returned. See also §4.10.2.

9.3.2 Keywords

Keyword names cannot be redefined or used as local variable names. All built-in keyword names are prefixed with `#%`, and a keyword binding is created for all primitive syntactic forms and procedures: for every built-in procedure or syntactic form x, there is a corresponding keyword `#%`x that accesses the same syntax or value.

There are no other built-in global keywords. Local keywords—such as `public` within a `class*` expression—are not globally enforced. This means, for example, that the name `public` may be bound to a value, but `public` as the first part of a `class*` sub-clause will not notice such a binding.

A new keyword is declared with `(keyword-name` *symbol*`)`. Once a *symbol* has been designated as a keyword, it cannot be bound locally or globally. (If *symbol* was not already defined, it will be defined as void.) Keywords declared this way are local to the namespace. Test for keywords in the current namespace with `(keyword-name?` *symbol*`)`.

Once a name is declared as a keyword, it is *syntactically* disallowed in any binding position. However, it is possible that a previously-compiled `define` or `set!` expression tries to change the value of keyword global binding, or `global-defined-value` may be used on a keyword binding. In that case, `exn:variable:keyword` exception is raised is raised at run-time.

A namespace can be created where the built-in `#%`-prefixed names are not keywords (see §9.3), but some built-in syntax and procedures will fail in this namespace if certain `#%`-prefixed names are re-defined or shadowed.

9.4 Parameters

A **parameter** is a thread-specific setting, such a 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, 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.

9.4.1 Built-in Parameters

MzScheme's built-in parameter procedures are listed in the following sections. The `make-parameter` procedure, described in §9.4.2, creates a new parameter and returns a corresponding parameter procedure.

9.4.1.1 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 §11.2.1); 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.

9.4.1.2 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!` 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 §11.1.9) to `print` values into a port.

9.4.1.3 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 always returns downcased symbols (e.g., `hi` when the input is any one of `hi`, `Hi`, `HI`, or `hI`).
- `(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.

9.4.1.4 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.

9.4.1.5 Language

- `(compile-allow-cond-fallthrough` [*on?*]`)` gets or sets a Boolean value indicating how to compile a `cond` or `case` expression without an `else` clause. If the value of this parameter is a true value, then `cond` or `case` expressions are compiled so that void is returned if no clause matches. Otherwise, `cond` or `case` expressions are compiled to raise the `exn:else` exception when no clause matches. Note that this parameter is used when an expression is *compiled*, not when it is *evaluated*.

- `(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*.

9.4.1.6 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)`.

- `(current-eval` [*proc*]`)` gets or sets a procedure that takes an S-expression and returns its value (or values; see Chapter 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 §9.3) 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 `print`s the value to the current output port (determined by the `current-output-port` parameter) and then outputs a newline.

9.4.1.7 Loading

- `(current-load` [*proc*]`)` gets or sets a procedure that takes a filename to load and returns the value (or values; see Chapter 2) of the last expression read from the file. This **load handler** is called by `load`, `load-relative`, `load/use-compiled`, and `load/cd`. The default load handler reads expressions from the file (with compiled expressions 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 that takes a filename to load as a dynamic extension (see §14.7) and returns the extension's value(s). The default load extension handler loads an extension using operating system primitives.

- `(current-load-relative-directory` [*path*]`)` gets or sets a complete directory pathname or `#f`. The 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 §11.2.1) and the result must be a complete pathname for an existing directory.

- `(use-compiled-file-kinds` [*kind-symbol*]`)` gets or sets a symbol, either `'all`, `'non-elaboration`, or `'none`, indicating whether `load/used-compiled` (and thus `require-library`) recognizes compiled files. If the value of this parameter is `'all`, then `load/use-compiled` recognizes compiled files as described in §14.1. If the value is `'none`, then `load/use-compiled` ignores compiled files. If the value is `'non-elaboration`, then `load/use-compiled` recognizes compiled files except for **`.zo`** files that start with `'e` followed by a space, which is a special annotation installed by `compile-file` (see §15.2.4) to indicate that a file contains elaboration-time expressions.

9.4.1.8 Libraries

- `(current-library-collection-paths` [*path-list*]`)` gets or sets a list of directory pathnames for library collections used by `require-library`. See Chapter 15 for more information.

- `(current-require-relative-collection` [*string-list*]`)` gets or sets a non-empty list of collection names (forming a subcollection path) used by `require-relative-library`, or `#f` if there is no current library collection. See Chapter 15 for more information.

9.4.1.9 Exceptions

- `(current-exception-handler` [*proc*]`)` gets or sets a procedure that is invoked to handle an exception. See §8.1 for more information about exceptions.

- `(error-escape-handler` [*proc*]`)` gets or sets a procedure that takes no arguments and escapes from the dynamic context of an exception. 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 §8.1) or `error` (see §8.2) 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 a string to print as an error message. This **error display handler** is called by the default exception handler. The default error display handler `display`s its argument to the current error port (determined by the `current-error-port` parameter). To report a run-time error, use `raise` (see §8.1) or `error` (see §8.2) 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 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 `writes` the value into a string;[3] 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.

9.4.1.10 Breaks

- `(break-enabled` [*enabled?*]`)` gets or sets a Boolean value that controls whether breaks are allowed. See §8.6 for more information.

9.4.1.11 Custodians

- `(current-custodian` [*custodian*]`)` gets or sets a custodian (see §9.5) that assumes responsibility for newly created threads, ports, and TCP listeners.

9.4.1.12 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; if the argument is a fixnum integer, it is used as the exit code, otherwise the exit code is `0`. (When MzScheme is used within another application, such as MrEd, the default exit handler may be remapped internally.)

9.4.1.13 Random Numbers

- `(current-pseudo-random-generator` [*generator*]`)` gets or sets a pseudo-random number generator (see §4.3) used by `random` and `random-seed`.

9.4.2 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:

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

The result of a `parameterize` expression is the result of the last *body-expr*. The *parameter-expr*s determine the parameters to set, and the *value-expr*s determine the corresponding values to install before evalutaing

[3] Using the default port write handler; see §11.1.9.

the *body-expr*s. All of the *parameter-expr*s are evaluated first, then the *value-expr*s, and then the parameters are set.

After the *body-expr*s 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-expr*s determine initial "remembered" values, and whenever control jumps into or out of the *body-expr*s, 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 p1 (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)
```

9.5 Custodians

A **custodian** manages a collection of threads, file ports, process ports, TCP ports, and TCP listeners.[4] Whenever a thread, file 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 §9.4.1.11). The only power given to a custodian is the authority to shut down all of its managed values.

The values managed by a custodian are only weakly held. This means that a will (see §12.2) can be executed for a value that is managed by a custodian.

`(make-custodian` [*custodian*]`)` creates a new custodian that is subordiante 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.

`(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.

[4] In MrEd, custodians also manage eventspaces.

10. Regular Expressions

MzScheme provides built-in support for regular expression pattern matching on strings, implemented by 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.

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

Figure 10.1: Grammar for regular expressions

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. On the other hand, `"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.
- `(regexp?` *v*`)` returns `#t` if *v* is a regexp value created by `regexp`, `#f` otherwise.
- `(regexp-match` *pattern string* [*start end*]`)` attempts to match *pattern* (a string or a regexp value) to a portion of *string*. The optional *start* and *end* arguments select a substring of *string* for matching (where the default is the entire string). If the match fails, `#f` is returned. If the match succeeds, a list of strings 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 earliest and longest match is found.

  Additional strings are returned in the list if *pattern* contains parenthesized subexpressions; matches for the subexpressions are provided in the order of the opening parentheses in *pattern*. When sub-expressions occur in different branches of an "or" (`|`), a `#f` is returned for each sub-expression that was not used in the match. When a single sub-expression is used multiple times in a match (e.g., a sub-expression is followed by "*"), then the rightmost match associated with the sub-expression is returned in the list.
- `(regexp-match-positions` *pattern string* [*start end*]`)` is like `regexp-match`, but instead of returning a list of strings, a list of number pairs is returned. Each pair of numbers refers to the range of characters in *string* that matched the corresponding (sub-)expression. If *start* is specified, the returned positions correspond to positions within *string*, rather than positions within the substring.
- `(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$" (for some integer n), then it is replaced with the nth matching sub-expression from *string*.[1] "&" and "\0" are synonymous. If the nth sub-expression was not used in the match or if n is greater than the number of sub-expressions in *pattern*, then "\$n$" 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".

[1] 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".

`(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:

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

11. System Utilities

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 §9.4.1.2 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 filename 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 BeOS**: 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.
  - **MacOS reading**: a return character read from the file is returned as a linefeed by the port; no filtering occurs for linefeeds.
  - **MacOS writing**: a return character written to the port is translated into a linefeed in the file; no filtering occurs for linefeeds.

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

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.

When an input or output file port is created, it is placed into the management of the current custodian (see §9.5).

11.1.3 Pipes

`(make-pipe)` returns two port values (see Chapter 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.

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 specially for string ports.

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 Ports

Two special procedures work on file ports:

- `(flush-output` [*output-port*]`)` forces all buffered data in the given file output port to be physically written. Buffered data is automatically flushed after each newline. The initial standard output and error ports are automatically flushed when `read`[1], `read-line`, `read-string`, or `read-string!` are performed on the initial standard input port. When called on a non-file port, `flush-output` takes no action. If *output-port* is omitted, then the current output port is flushed.
- `(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 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.

[1] Flushing is performed by the default port read handler (see §11.1.8) rather than by `read` itself.

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.

11.1.6 Custom Ports

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

- `(make-input-port` *read-char-proc char-ready?-proc close-proc* [*peek-char-proc*]`)` creates an input port.

  The *read-char-proc* argument is a procedure that takes no arguments and returns the next input character from the port. When no more characters are available from the port, *read-char-proc* returns `eof`. (If a non-character and non-`eof` value is returned, the `exn:i/o:port:user` exception is raised.)

  The *char-ready?-proc* argument is a procedure that takes no arguments and returns a true value if a character (or `eof`) is ready to be read, `#f` otherwise.

  The *close-proc* argument is a procedure of no arguments to be called when the port is closed.

  The *peek-char-proc* argument is a procedure that takes no arguments and returns the next input character from the port, but also saves the character for the next *read-char-proc* or *peek-char-proc* call. If *peek-char-proc* is not provided, the default procedure uses *read-char-proc*.[2] If *peek-char-proc* is provided, MzScheme does not check that calling *read-char-proc* obtains the value returned by a previous *peek-char-proc* call.

- `(make-output-port` *write-proc close-proc*`)` creates an output port.

  The *write-proc* argument is a procedure that takes a string and writes it. (The *write-proc* procedure can safely store or mutate this string.)

  The *close-proc* argument is a procedure of no arguments; it is called when the port is closed.

Ports created by `make-input-port` and `make-output-port` are immediately open for reading or writing. If the *close* procedure does not have any side-effects, then the custom port does not need to be explicitly closed.

11.1.7 Reading and Printing

In addition to the standard reading procedures, MzScheme provides `read-line`, `read-string`, and `read-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` — break lines on linefeed characters; this is the default.
  - `'return` — break lines on return characters.
  - `'return-linefeed` — break 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.

[2] The default *peek-char-proc* relies on special internal support from `peek-char` and `read-char` to implement peeking.

  - `'any` — break 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` — break 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*). If no characters are available before an end-of-file, then `eof` is returned.

- (`read-string!` *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 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 $m < \textit{end-k} - \textit{start-k}$, then *string* is not modified at indices $\textit{start-k} + m$ though *end-k*.

In addition to the standard printing procedures, MzScheme provides `print`, which outputs values to a port by calling the port's print handler (see §11.1.9):

- (`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 §9.4.1.2).

Formatted output is written to a port with `fprintf`:

- (`fprintf` *output-port format-string v* ⋯) prints formatted output to *output-port*, where *format-string* is a string that is printed; *format-string* can contain special formatting tags:
  - ~n or ~% prints a newline
  - ~a or ~A `display`s the next argument among the *v*s
  - ~s or ~S `write`s the next argument among the *v*s
  - ~v or ~V `print`s the next argument among the *v*s
  - ~e or ~E outputs the next argument among the *v*s using the current error value conversion handler (see §9.4.1.9) and current error printing width
  - ~c or ~C `write-char`s the next argument in *v*s; if the next argument is not a character, the `exn:application:mismatch` exception is raised

  - ~b or ~B prints the next argument among the *v*s in binary; if the next argument is not an exact number, the `exn:application:mismatch` exception is raised
  - ~o or ~O prints the next argument among the *v*s in octal; if the next argument is not an exact number, the `exn:application:mismatch` exception is raised
  - ~x or ~X prints the next argument among the *v*s in hexadecimal; if the next argument is not an exact number, the `exn: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*:[3]

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

followed by a newline.

11.1.8 Customizing Read

Each input port has its own **port read handler**. This handler is invoked to read S-expressions from the port when the standard `read` procedure is applied to the port. A port read handler takes a single argument: the port being read. The return value is the value that was read from the port.

- (`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).

11.1.9 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.

[3] Assuming that the current port display and write handlers are the default ones; see §11.1.9 for more information.

- (`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 §9.4.1.2); the default global port print handler is the same as the default write handler.

11.2 Filesystem Utilities

Additional filesystem utilities are in MzLib; see §15.2.8.

11.2.1 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 the Mac 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 BeOS, a user directory specification using "~" is expanded.[4] Under MacOS, file and folder aliases are resolved to real pathnames. 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 contain a null character (`#\nul`). When 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 utilites are:

- (`build-path` *base-path sub-path* ···) creates an 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.

[4] Under Unix and BeOS, expansion does *not* convert multiple adjacent slashes to a single slash. However, extra slashes in a pathname are always ignored.

Under MacOS, 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 MacOS. 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-base* 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 MacOS examples.

```
(define p1 (build-path (current-directory) "src" "scheme"))
; Unix:  p1 ⇒ "/home/joeuser/src/scheme"
; MacOS: p1 ⇒ "My Disk:Joe's Files:src:scheme"
(define p2 (build-path 'up 'up "docs" "MzScheme"))
; Unix:  p2 ⇒ "../../docs/MzScheme"
; MacOS: p2 ⇒ ":::docs:MzScheme"
(build-path p2 p1)
; Unix:  raises exn:i/o:filesystem:path because p1 is absolute
; MacOS: ⇒ ":::docs:MzScheme:My Disk:Joe's Files:src:scheme"
(build-path p1 p2)
; Unix:  ⇒ "/home/joeuser/src/scheme/../../docs/MzScheme"
; MacOS: ⇒ "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 niether 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 or BeOS, 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-existant path.

- `(simplify-path` *path*`)` eliminates up-directory ("..") in Unix, BeOS, 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-existant 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 BeOS, this procedure always returns the input path. Under Windows and MacOS, the resulting string uses only lowercase letters. Under Windows, all forward slashes ("/") are converted to backward slashes ("\"). 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 Chapter 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 an absolute executable pathname that is synonymous with the executable pathname *program-sub-path* such that the file or directory *related-sub-path* (a relative path string) exists in the same directory. If such a path cannot be found, `#f` is returned.

  This procedure is used by MzScheme (as a stand-alone executable) to find the standard library collection directory (see Chapter 15); in this case, *program* is the name used to start MzScheme and *related* is `"collects"`.

  If *program-sub-path* is an absolute path, `find-executable-path` determines whether *related-sub-path* exists in the directory of *program-sub-path*. If so, *program* is returned. Otherwise, if *program-sub-path* is a link to another 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 relative path, `find-executable-path` gets the value of the **PATH** environment variable; if this environment variable is defined, `find-executable-path` tries each colon-separated path in **PATH** as a prefix for *program-sub-path* using the search algorithm described above. If the 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.)

  The *related-sub-path* argument is used because, under Unix and BeOS, *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.

- `(find-system-path` *kind-symbol*`)` returns a machine-specific path for a standard type of path specified by *kind-symbol*, which must be one of the following:

  - `'home-dir` — the current user's home directory. Under MacOS, this is the preferences directory. Under Windows, this is the directory specified by the **HOMEDRIVE** and **HOMEPATH** environment variables; if those environment variables are not defined or the directory does not exist, the directory containing the MzScheme executable is returned, instead.
  - `'pref-dir` — the standard directory for storing the current user's preferences. Under Unix, Windows, and BeOS, this is the user's home directory.
  - `'temp-dir` — the standard directory for storing temporary files. Under Unix, Windows, and BeOS, this is the directory specified by the **TMPDIR** environment variable, if it is defined.
  - `'init-dir` — the directory containing the initialization file used by stand-alone MzScheme application. It is the same as the current user's home directory.
  - `'init-file` — the file loaded at start-up by the stand-alone MzScheme application. The directory part of the path is the same path as returned for `'init-dir`. The file name is platform-specific:
    * Unix and BeOS: **.mzschemerc**
    * Windows and MacOS: **mzschemerc.ss**

- `(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 BeOS, paths in a path list are separated by a colon (":"); under Windows and MacOS, 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.)

11.2.2 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.

- `(link-exists?` *path*`)` returns `#t` if a link *path* exists (Unix, BeOS, and MacOS), `#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*`)` 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. The pathname *new-path* cannot refer to an existing file or directory. If *old-path* is a link, the link is renamed rather than the destination of the link.

- `(file-or-directory-modify-seconds` *path*`)` returns the file or directory's last modification date as platform-specific seconds (see also §11.4).[5] 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.

[5] For FAT filesystems under Windows, directories do not have modification dates. Therefore, the creation date is returned for a directory (but the modification date is returned for a file).

- `(file-size` *path*`)` returns the (logical) size of the specified file. (Under MacOS, 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 MacOS, the resource fork is also preserved in the copy.

11.2.3 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 §9.4.1.1.
- `(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 new*`)`, as described in the previous section, renames directories.
- `(file-or-directory-modify-seconds` *path*`)`, as described in the previous section, gets directory dates.
- `(file-or-directory-permissions` *path*`)`, as described in the previous section, gets directory permissions.
- `(directory-list` [*path*]`)` returns a list of all files and directories in the directory specified by *path*. If *path* is omitted, a list of files and directories in the current directory is returned.
- `(filesystem-root-list)` returns a list of all current root directories.

11.3 Networking

MzScheme provides a minimal collection of TCP-based communication procedures.

- `(tcp-listen` *port-k* [*max-allow-wait-k*]`)` creates a "listening" server on the local machine at the specified port number (where *port-k* is an exact integer between `1` and `65535`). 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`.

  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 §9.5).

  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[6] (e.g., `"cs.rice.edu"`), and *port-k* (an exact integer between `1` and `65535`) is the port where the server is listening.

  Two values (see Chapter 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 §9.5).

  The client input and output ports are not independent: closing the output port does not cause the server to receive an end-of-file. Instead, both ports must be closed before the server receieves an end-of-file.

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

- `(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 Chapter 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 §9.5).

  The accepted input and output ports are not independent: closing the output port does not cause the client's input port to receive and end-of-file. Instead, both ports must be closed before the client receieves an end-of-file.

  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-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.

- `(tcp-listener?` *v*`)` returns `#t` if *v* is a TCP listener value created by `tcp-listen`, `#f` otherwise.

11.4 Time

11.4.1 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 (increaing 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 §11.2.2).

[6] The name `"localhost"` agenerally specifies the local machine.

`(seconds->date` *secs-n*`)` takes *secs-n*, a platform-specific time in seconds (an exact integer) returned by `current-seconds` or `file-or-directorymodify-seconds`, and returns an instance of the `date` structure type with 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`

The value returned by `current-seconds` or `file-modify-seconds` is not portable among platforms. Convert a time in seconds using `seconds->date` when portability is needed.

See also §15.2.6 for additional date utilities.

11.4.2 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.

`(current-process-milliseconds)` returns the amount of processor time that has been consumed by this MzScheme process so far in fixnum milliseconds. (Under Unix and BeOS, 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)`.

11.4.3 Timing Execution

The `time-apply` procedure collects timing information for a procedure application:

- `(time-apply` *thunk*`)` invokes the procedure *thunk* with no arguments. Three values are returned: a list containing the result(s) of applying *thunk*, the number of milliseconds of CPU time required to obtain this result, and the number of "real" milliseconds required for the result.

The reliability of the timing numbers depends on the operating system. 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. This form also prints the amount of time spent in garbage-collection (already included in the CPU time). The result of the `time` expression is the result of *expr*.

11.5 Operating System Processes

(`system` *command-string*) executes a Unix, Windows, or BeOS shell command synchronously (i.e., the call to `system` does not return until the subprocess has ended), or launches a MacOS application by its creator signature (and returns immediately). The *command-string* argument is a string (of four characters for MacOS) containing no null characters. If the command succeeds, the return value is `#t`, `#f` otherwise. Under MacOS, if *command-string* is not four characters, the `exn:application:mismatch` exception is raised.

(`system*` *command-string arg-string* $\cdots$) is like `system`, except that *command-string* is a filename that is executed directly (instead of through a shell command or through a MacOS creator signature), and the *arg-string*s are the arguments. Under Unix, Windows and BeOS, the executed file is passed the specified string arguments (which must contain no null characters). Under MacOS, no arguments can be supplied, otherwise the `exn:misc:unsupported` exception is raised.

(`execute` *command-string*) does not return unless there was an error executing *command-string*. If it returns, the result is void. Under Unix, it is like `system` followed by `exit` when the command succeeds. For MacOS, this procedure is `system` followed by an immediate exit if the target application launches successfully. (This procedure is not supported for Windows or BeOS, although `execute*` is supported for Windows and BeOS.)

(`execute*` *command-string arg-string* $\cdots$) is like `execute` for Unix and MacOS, except that *command-string* is a filename that is executed directly, and the *arg-string*s are the arguments. Under Unix, when the current exit handler is MzScheme's default exit handler, the execution of *command-string* replaces the MzScheme process. Under MacOS, no arguments can be supplied, otherwise the `exn:misc:unsupported` exception is raised. Under Windows and BeOS, this procedure is like `system*` except that `exit` is called immediately if the specified program launches successfully, otherwise void is returned.

(`process` *command-string*) executes a shell command asynchronously under Unix. (This procedure is not supported for Windows, BeOS, or MacOS, although `process*` is supported for Windows and BeOS.) If the subprocess is launched successfully, the result is a list of five values:

- an input port piped from the subprocess's standard output,
- an output port piped to the subprocess standard input,
- the system process id of the subprocess,
- an input port piped from the subprocess's standard error,[7] and
- a procedure of one argument, either `'status` or `'wait`: `'status` returns the status of the subprocess as one of `'running`, `'done-ok`, or `'done-error`; `'wait` blocks execution in the current thread until the subprocess has completed.

All three ports returned from `process` must be explicitly closed with `close-input-port` and `close-output-port`.

(`process*` *command-string arg-string* $\cdots$) is like `process` under Unix for all of Unix, Windows, and BeOS, except that *command-string* is a filename that is executed directly, and the *arg-string*s are the arguments. (This procedure is not supported for MacOS.)

The ports returned by `process` and `process*` are placed into the management of the current custodian (see §9.5). The `exn:misc:process` exception is raised when a low-level error prevents the spawning of a process or the creation of operating system pipes for process communication.

(`send-event` *receiver-string event-class-string event-id-string* [*direct-argument-v argument-list*]) sends an AppleEvent under MacOS. (This procedure is not supported for Unix, Windows, or BeOS.) The *receiver-*

[7] The standard error port is placed after the process id for compatibility with other Scheme implementations. For the same reason, `process` returns a list instead of multiple values.

string, *event-class-string*, and *event-id-string* arguments specify the signature of the receiving application, the class of the AppleEvent, and the ID of the AppleEvent. Each of these must be a four-character string, otherwise the `exn:application:mismatch` exception is raised. The *direct-argument-v* value is converted (see below) and passed as the main argument of the event; if *direct-argument-v* is void, no main argument is sent in the event. The *argument-list* argument is a list of two-element lists containing a typestring and value; each typestring is used ad the keyword name of an AppleEvent argument for the associated converted value. Each typestring must be a four-character string, otherwise the `exn:application:mismatch` exception is raised. The default values for *direct-argument* and *arguments* are void and `null`, respectively.

The following types of MzScheme values can be converted to AppleEvent values passed to the receiver:

`#t` or `#f`	⇒ Boolean
small integer	⇒ Long Integer
inexact real number	⇒ Double
string	⇒ Characters
list of convertable values	⇒ List of converted values
`#(file` *pathname*`)`	⇒ Alias (file exists) or FSSpec (does not exist)
`#(record (`*typestring* *v*`) ···)`	⇒ Record of keyword-tagged values

If other types of values are passed to `send-event` for conversion, the `exn:misc:unsupported` exception is raised.

The `send-event` procedure does not return until the receiver of the AppleEvent replies. The result of `send-event` is the reverse-converted reply value (see below), or the `exn:misc` exception is raised if there is an error. If there is no error or return value, `send-event` returns void.

The following types of AppleEvent values can be reverse-converted into a MzScheme value returned by `send-event`:

Boolean	⇒ `#t` or `#f`
Signed Integer	⇒ integer
Float, Double, or Extended	⇒ inexact real number
Characters	⇒ string
list of reverse-convertable values	⇒ List of reverse-converted values
Alias or FSSpec	⇒ `#(file` *pathname*`)`
Record of keyword-tagged values	⇒ `#(record (`*typestring* *v*`) ···)`

If the AppleEvent reply contains a value that connot be reverse-converted, the `exn:misc` exception is raised.

11.6 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 MacOS 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")
```

11.7 Runtime Information

`(system-type)`returns a symbol indicating the type of the operating system for a running MzScheme. The possible values are:

- `'unix`
- `'windows`
- `'macos`
- `'beos`
- `'oskit`

Future ports of MzScheme will expand this list of system types.

`(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 a string indicating the currently executing version of MzScheme.

`(banner)` returns a string for MzScheme's start-up banner text (or the banner text for an embedding program, such as MrEd).

12. Memory Management

12.1 Weak Boxes

A **weak box** is similar to a normal box (see §4.9), 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.

12.2 Will Executors

A **will executor** manages a collection of values and associated **will procedures**. The will procedure for each value is invoked when the value has been proven unreachable — except through will executors, weak boxes, and custodians — by the automatic memory manager.

Calling the `will-execute` or `will-try-execute` procedure executes a will that is ready in the specified will 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* may 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, the wills are executed in the reverse order of registration. Since executing a will procedure makes the value reachable again, the value must be proven unreachable once again before another of the wills is executed.

If the content value of a weak box is registered with a will executor, the weak box's content is not changed to `#f` until all wills have been executed for the value and the value has been proven unreachable again.

12.3 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)` returns an estimate of the number of bytes of memory occupied by reachable data. (The estimate is calculated *without* performing an immediate garbage collection; performing a collection generally decreases the number returned by `current-memory-use`.)

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

13. Macros

A low-level (`defmacro`-like) macro system is built into MzScheme. Macros defined with this system are not hygenic. The high-level R^5RS macro system is mostly supported by MzLib (see §15.2.23).

13.1 Defining Macros

Global macros are defined with `define-macro`:

`(define-macro` *name procedure-expr*`)`

When the macro is later "applied," the (unevaluated) argument S-expressions are passed to the procedure obtained from *procedure-expr*. The result is a new S-expression that replaces the macro application expression in its context. If *procedure-expr* evaluates to a non-procedure, the `exn:misc` exception is raised.

For example, the `when` macro is defined as follows:

```
(define-macro when
  (lambda (test . body)
     `(if ,test (begin ,@body))))
```

Macros defined with `define-macro` (at the top-level) are bound in the current namespace. Local macros are defined with `let-macro`:

`(let-macro` *name procedure-expr expr* $\cdots$[1]`)`

This syntax is similar to `define-macro`, except that the macro is only effective in the body *expr*s. The result of a `let-macro` expression is the value of the *expr* body. Note that the environment for *procedure-expr* includes only global variables and it is evaluated at expansion time, not at run time. If *procedure-expr* evaluates to a non-procedure, the `exn:misc` exception is raised.

When a `define-macro` statement appears in a implicit sequence (like an internal definition; see §3.5.5), it is transformed into a `let-macro` expression, where the body of the closure following the `define-macro` statement becomes the body of the `let-macro` expression.

`(macro?` *v*`)` returns `#t` if *v* is a macro created with `define-macro`, `#f` otherwise. Note that `macro?` cannot be applied directly to macro identifiers, but macro values can be obtained indirectly with `global-defined-value`.

A `define-macro` expression is treated specially by `compile-file` (see §15.2.4).

13.2 Identifier Macros

An **identifier macro** is a macro that is not "applied" to syntactic arguments. Instead, an identifier macro identifier is directly replaced with its value whenever the identifier is in an expression position. Identifier macros are defined with `define-id-macro`:

`(define-id-macro` *name value-expr*`)`

The *value-expr* expression can evaluate to any value. When the identifier *name* is encountered during compilation, it is compiled as if the result of *value-expr* is in the place of *name*. Like `define-macro`, identifier macros defined with `define-id-macro` (at the top-level) are bound in the current namespace, and local identifier macros are defined with `let-id-macro`.

For example, the following expression uses `x` to automatically unbox the value in `b`:

```
(let ([b (box 5)])
  (let-id-macro x `(unbox b)
    (display x) (newline)
    (set-box! b 8)
    (display x) (newline)))
```

Each use of `x` is replaced by `(unbox b)`, so this expression prints `5` and `8` to the current output port. The `x` identifier is not a variable; `(set! x 8)` is illegal, since this expands to `(set! (unbox b) 8)`. The value of the identifier macro `x` is the S-expression `'(unbox b)`. Leaving out the quote in defining `x`'s value is illegal:

```
(let ([b (box 5)])
  (let-id-macro x (unbox b)
    expr))
```

because the `(unbox b)` expression is evaluated at compile time and is not in the scope of `b`. (If `b` is a global variable bound to a box when the expression is compiled, then the expression is legal and the global `b` is used.)

As with `let-macro`, the `let-id-macro` form defines a local idenfifier macro and an internal `define-id-macro` expression is transformed into a `let-id-macro` expression.

`(id-macro?` *v*`)` returns `#t` if *v* is an identifier macro created with `define-id-macro`, `#f` otherwise. Note that `id-macro?` cannot be applied directly to identifier macro identifiers, but identifier macro values can be obtained indirectly with `global-defined-value`.

A `define-id-macro` expression is treated specially by `compile-file` (see §15.2.4).

13.3 Expansion Time Binding and Evaluation

Expansion time is the time at which macro and identifier macro definition values are evaluated and macro procedures are invoked. Macros, identifier macros, and primitive syntax are **expansion-time values**. General expansion-time values can be defined with the `define-expansion-time` and `let-expansion-time` syntactic forms. Scoped expansion-time bindings can be obtained with `local-expansion-time-value` or `global-expansion-time-value`, and scoping information is available through `local-expansion-time-bound?`.

- `(define-expansion-time` *id expr*`)` evaluates *expr* at run time and binds it as an expansion-time value to the global variable *id*. This form is treated specially by `compile-file` (see §15.2.4).

`#%lambda`	`#%begin0`	`#%with-continuation-mark`
`#%let-values`	`#%case-lambda`	`#%define-macro`
`#%letrec-values`	`#%struct`	`#%define-id-macro`
`#%define-values`	`#%class*/names`	`#%define-expansion-time`
`#%quote`	`#%interface`	
`#%if`	`#%unit`	
`#%begin`	`#%compound-unit`	
`#%set!`	`#%invoke-unit`	
`#%cond`†		

Figure 13.1: Syntactic forms after expanding with `expand-defmacro`

- `(let-expansion-time` *id expr body-expr* ...[1]`)` evaluates *expr* at expansion time and binds an expansion-time value to *id* within the *body-expr*s. The run-time result of a `let-expansion-time` expression is the result of evaluating the last *body-expr*.

  As with `define-macro`, an internal `define-expansion-time` expression is transformed into a `let-expansion-time` expression.

- `(local-expansion-time-value` *symbol*`)` returns the expansion-time binding of *symbol* in the scope of the expression currently being expanded. The identifier corresponding to *symbol* must have been bound to an expansion-time value by `define-expansion-time` or `let-expansion-time`, otherwise the `exn:misc` exception is raised. If `local-expansion-time-value` is invoked at run time, the `exn:misc` exception is raised.

- `(global-expansion-time-value` *symbol*`)` is like `(local-expansion-time-value` *symbol*`)`, but *symbol* must be bound in the current namespace (and lexical bindings are ignored).

- `(local-expansion-time-bound?` *symbol*`)` returns `#t` if the identifier *symbol* is locally bound, `#f` otherwise. If `local-expansion-time-bound?` is invoked at run time, the `exn:misc` exception is raised.

- `(expansion-time-value?` *v*`)` returns `#t` if *v* is a value created with `define-expansion-time`, `#f` otherwise. Note that `expansion-time-value?` cannot be applied directly to expansion-time identifiers, but expansion-time values can be obtained indirectly with `global-defined-value`.

- `begin-elaboration-time` `(begin-elaboration-time` *body-expr* ...[1]`)` evaluates each *body-expr* at expansion time. Once the *body-expr*s are evaluated, the entire form is replaced with the S-expression result of the last *body-expr*, and macro expansion continues. This form is treated specially by `compile-file` (see §15.2.4).

13.4 Primitive Syntax and Expanding Macros

Only the syntactic forms shown in Figure 13.1 will occur in a fully expanded expression. The dagger next to `cond` indicates that it will appear only with zero clauses, and only in the compilation mode where the `exn:else` exception is raised if no clause matches (see §3.2).

Like macros, primitive syntax names are bound in the global namespace, and primitive syntax values can be obtained with `global-defined-value`.

`(syntax?` *v*`)` returns `#t` if its argument is a primitive syntax value, `#f` otherwise.

`(expand-defmacro` *s-expr*`)` expands all macros in the S-expression *s-expr* and returns the new, expanded S-expression.

`(expand-defmacro-once` *s-expr*`)` partially expands macros in the S-expression *s-expr* and returns the partially-expanded S-expression.

`(local-expand-defmacro` *s-expr* [*shadow-list*]`)` expands *s-expr* during expansion-time (see §13.3), and locally-defined macros are used from the context of the expression currently being expanded. (This procedure is normally used in the implementation of a macro.) If *shadow-list* is provided, it must be a list of symbols, which `local-expand-defmacro` treats as identifiers that shadow syntax bindings the current lexical environment. If `local-expand-body-expression` is invoked at run time, the `exn:misc` exception is raised.

`(local-expand-body-expression` *s-expr* [*shadow-list*]`)` expands expands *s-expr* only far enough to determine whether it expands to a `define-values` or `begin` expression. The result is two values: the (partially expanded) expression and either `'#%define-values`, `'#%begin`, or `#f` (where `#f` means the expression has some other form). If `local-expand-defmacro` is invoked at run time, the `exn:misc` exception is raised.

14. Support Facilities

14.1 Eval and Load

`(eval` *expr*`)` evaluates the S-expression *expr* in the current namespace.[1] (See §9.3 and §9.4.1.6 for more information about namespaces.)

`(load` *file-path*`)` evaluates each expression in the specified file using `eval`.[2] The return value from `load` is the value of the last expression from the loaded file (or void if the file contains no expressions). If *file-path* is a relative 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 §9.4.1.7) 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 §15.2.4) and **.so** (Unix, BeOS, and MacOS) or **.dll** (Windows) files. 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 a 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 pasrt of *file-path* (without the directory path part of the name and without the filename extension); if `#f` is returned, then `load/use-compiled` ignores **_loader** for *file-path* and continues as normal. Otherwise, the return value is yet another procedure. When this procedure is applied to no arguments, it should have the same effect as loading *file-path*.

While a **.zo**, **.so**, or **.dll** file is loaded (or while a thunk returned by **_loader** is invoked), the current `load-relative` directory is set to the directory of the original *file-path*.

`(load/cd` *file-path*`)` is the same as `(load` *file-path*`)`, but `load/cd` sets both the current directory and current `load-relative` directory to the directory of *file-path* before the file's expressions are evaluated.

`(read-eval-print-loop)` starts a new `read-eval-print` loop using the current input, output, and error

[1] The `eval` procedure actually calls the current evalutation handler (see §9.4.1.6) with *e* to evaluate the expression.

[2] The `load` procedure actually just sets the current `load-relative` directory and calls the current load handler (see §9.4.1.7) with *file-path* to load the file. The description of `load` here is actually a description of the default load handler.

ports. When `read-eval-print-loop` starts, it installs a new error escape procedure (see §8.7) 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 §9.4.1):

```
(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 §9.4.1.12). The default value for v is `0`. If the exit handler does not escape or terminate the thread, void is returned.

The default exit handler quits MzScheme, using its argument as the exit code if it is a fixnum, `0` otherwise. The default exit handler in MrEd exits the application on the next event cycle.

14.3 Input Parsing

MzScheme's input parser follows these 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 §9.4.1.3). Similarly, the parsing of curly braces is controlled with the `read-curly-brace-as-paren` parameter. By default, square brackets are 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 remaing 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 §9.4.1.3). Box reading is enabled by default. When box reading is disabled and `#&` is provided as input, the `exn:read` exception is raised.
- 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: `#\`$n_1n_2n_3$ where n_1 is in the range [`0`, `3`] and n_2 and n_3 are in the range [`0`, `7`]. Whenever `#\` is followed by at least two characters in the range [`0`, `7`], the next character must also be in this range and the resulting octal number must be in the range 000_8 to 377_8.

- MzScheme's identifier and symbol syntax is considerably more liberal than the syntax specified by R^5RS. 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.

An input token constructed in this way is an identifier when it is not a numerical constant (following the number syntax of R^5RS plus the special inexact numbers `+inf.0`, etc., discussed in §4.3). A token containing a backslash or vertical bars is never treated as a numerical contant.

Examples:

– `(quote a\(b)` produces the same symbol as `(string->symbol "a(b")`.
– `(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 §9.4.1.3). Vertical bar quotes are enabled by default.

- S-expressions with shared stucture 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. See §9.3.2.

- 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 `load`ed file begins with `#!`, it is ignored by the default load handler. If an ignored line ends with a backslash ("\"), then the next line is also ignored.

14.4 Output Printing

MzScheme's printer follows these 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 ommitted. 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 §9.4.1.4). Shorthand vector printing is enabled by default.
- Boxes (see §4.9) 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 §9.4.1.4). Box printing is enabled by default.
- Structures (see Chapter 5) 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. When structure printing is disabled, structures are printed as `#<struct:`*s*`>`. Structure printing is controlled with the `print-struct` Boolean parameter (see §9.4.1.4). Structure printing is disabled by default.
- Symbols containing spaces or special characters `write` using escaping backslashes and quoting vertical bars. 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 §9.4.1.3), 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`). All other characters `write` as `#\` followed by the single-byte character value. All characters `display` as the single-byte character value.
- S-expressions with shared stucture 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 §9.4.1.4). Graph reading is enabled by default, and graph printing is disabled by default. However, when the printer encounters an graph containing a cycle, graph printing is automatically enabled (temporarily). 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 MzScheme code. MzScheme can read compiled code somewhat 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 §15.2.4) 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.

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. 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 §9.4.1.3). When the default load handler is used to load a file, compiled expression reading is automatically (temporarily) enabled as each expression is read.

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 §9.4.1.7) 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 §9.4.1.7). 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.

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 extentions 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[3] 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 overlayed 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).

[3]The set of environment variables is not saved. When an image is restored, the environment variables of the restoring program are transfered into the restored program.

15. Library Collections and MzLib

A **library** is a fragment of MzScheme code that can be used in multiple programs. MzScheme provides an mechanism for grouping libraries into **collections** that can be distributed and easily added to a local MzScheme installation. A collection is normally installed into a directory named **collects** that is in the same directory as the MzScheme executable.[1] Each installed collection is represented as a subdirectory within the **collects** directory.

Client programs incorporate a library by using the library name along with the name of the library's collection: `(require-library` *library-file-path collection* ⋯`)` loads a library from *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 loaded from the **mzlib** collection. The `require-library` form returns the result(s) of the last expression in the library file.

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*.

When `require-library` is used to load a file, the library name and the resulting value(s) are recored in a table associated with the current namespace. If `require-library` is evaluated for a library that is already registered in the current namespace's load table, then the library is not loaded again; the result(s) recorded in the load table is returned, instead.

While `require-library` loads a library file, it sets the `current-require-relative-collection` parameter to the path of collection names that specify the library's subcollection. This path is used by the `require-relative-library` form: `(require-relative-library` *library-file-path collection* ⋯`)` requires *library-file-path* from the collection specified by the `current-require-relative-collection` parameter; if extra *collection*s are provided, they are appended to the end of the subcollection path in `current-require-relative-collection` for finding *library-file-path*.

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 §9.4.1.8). The list of paths in `current-library-collection-paths` is searched from first to last to locate a collection when a library is requested. The value of the parameter is initialized by the stand-alone version of MzScheme as follows:[2]

```
(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")
```

[1] In the PLT distribution of MzScheme for Unix and BeOS, the **collects** directory is in the top-level **plt** directory, rather than with the platform-specific binary or with the script in **plt/bin**.

[2] MrEd initializes the `current-library-collection-paths` parameter in the same way.

```
                "/usr/local/lib/plt/collects" ; Unix only
                "/boot/apps/plt/collects" ; BeOS only
                "c:\\plt\\collects")) ; Windows only
      null)))
```

where `program` is the name used to start MzScheme (always a complete path for MacOS). See also §11.2.1 for information about `path-list-string->path-list` and `find-executable-path`. `(collection-path` *collection* ⋯[1]`)` returns the path containing the libraries of *collection*.

The `require-library` form loads library files using `load/use-compiled`. In the table of loaded libraries, library names are registered using the original suffix even when `load/use-compiled` loads a compiled version of a file.

Since `require-library`'s libraries and collections are specified via string literals, this form supports the static analysis of programs by MrSpidey, DrScheme's static debugger. The `require-library/proc` procedure generalizes `require-library` to a procedural form, but it is not supported by the static debugger. Nevertheless, the `require-library` form normally expands to an application of `require-library/proc`:

`(require-library` *library collection* ⋯`)`
⇒
`(require-library/proc` *library collection* ⋯`)`

Similarly, `require-relative-library/proc` is the procedure form of `require-relative-library`.

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 desribed in §15.1.

15.1 MzLib Overview

The MzLib utilities and syntax are not built into MzScheme and they are *not* already present after the stand-alone version of MzScheme has started.

MzLib is distributed among several smaller libraries, and each smaller library can be separately loaded. The utilities provided by each library and the other libraries it requires are detailed in the next chapter.

The non-macro parts of MzLib are written using signed units (see §7.2). These libraries can be used through a unit, or the unit can be invoked and opened to use the utilities as top-level defintions. Non-macro libraries have five requireable files:

- `(require-library "`*x*`s.ss")` loads the signature of the unit for library *x*.
- `(require-library "`*x*`u.ss")` loads *x* as a unit definition, automatically requiring *x***s.ss** and *x***r.ss**.
- `(require-library "`*x*`r.ss")` loads *x* as a unit value (without adding any top-level definitions); the *x***s.ss** library must be already loaded.
- `(require-library "`*x*`.ss")` requires *x***u.ss** and copies the unit's exports into the global environment.

Some libraries contain only macros, and therefore do not have the *x***u.ss**, *x***r.ss**, and *x***s.ss** files.

Applications written in core Scheme will most likely use `(require-library "`*x*`.ss")`, while unit-based applications will use `(require-library "`*x*`s.ss")` and `(require-library-unit/sig "`*x*`r.ss")` (see §15.2.18).

In all, MzLib contains the following requireable files:

- **awk.ss**
- **cmdline.ss**, **cmdlineu.ss**, **cmdliner.ss**, **cmdlines.ss**
- **compat.ss**, **compatu.ss**, **compatr.ss**, **compats.ss**
- **compile.ss**, **compileu.ss**, **compiler.ss**, **compiles.ss** [CORE]
- **core.ss**, **coreu.ss**, **corer.ss**, **cores.ss**
- **date.ss**, **dateu.ss**, **dater.ss**, **dates.ss**
- **defstru.ss** [CORE]
- **file.ss**, **fileu.ss**, **filer.ss**, **files.ss** [CORE]
- **functio.ss**, **functiou.ss**, **functior.ss**, **functios.ss** [CORE]
- **inflate.ss**, **inflateu.ss**, **inflater.ss**, **inflates.ss**
- **macro.ss** [CORE]
- **match.ss** [CORE]
- **math.ss**, **mathu.ss**, **mathr.ss**, **maths.ss** [CORE]
- **mzlib.ss**, **mzlibu.ss**, **mzlibr.ss**, **mzlibs.ss**
- **pconvert.ss**, **pconveru.ss**, **pconverr.ss**, **pchookr.ss**, **pconvers.ss**
- **pretty.ss**, **prettyu.ss**, **prettyr.ss**, **prettys.ss** [CORE]
- **refer.ss** [CORE]
- **restart.ss**, **restartu.ss**, **restartr.ss**, **restarts.ss**
- **shared.ss** [CORE]
- **spidey.ss** [CORE]
- **string.ss**, **stringu.ss**, **stringr.ss**, **strings.ss** [CORE]
- **synrule.ss**, **synrule.ss**
- **thread.ss**, **threadu.ss**, **threadr.ss**, **threads.ss** [CORE]
- **trace.ss**
- **traceld.ss**
- **transcr.ss**, **transcru.ss**, **transcrs.ss**, **transcrs.ss**

The **compatm.ss** library contains just the macro portion of **compat.ss**.

A set of libraries marked with [CORE] can be required at once by requiring **core.ss**, **coreu.ss**, **corer.ss**, or **cores.ss**. The **corem.ss** library requires only the macro-defining [CORE] libraries.

The set of all MzLib libraries can be required at once by requiring **mzlib.ss**, **mzlibu.ss**, **mzlibr.ss**, or **mzlibs.ss**. The **mzlibm.ss** library loads the macro-defining MzLib libraries.

For libraries without x**u.ss** files, **coreu.ss** and **mzlibu.ss** require x**.ss**.

15.1.1 Thanks

Contributors to MzLib include Robby Findler, Shriram Krishnamurthi, Gann Bierner, Dorai Sitaram, and Kurt Howard (working from Steve Moshier's Cephes library). Publically available packages have been assimilated from others, including Andrew Wright (`match`) and Marc Feeley (original pretty-printing implementation).

15.2 MzLib Libraries

15.2.1 Awk: awk.ss

Files: **awk.ss**

This library defines the `awk` macro from Scsh:

```
(awk next-record-expr
     (record field-variable ...)
     counter-variable/optional
     ((state-variable init-expr) ...)
     continue-variable/optional
  clause ...)
```

counter-variable/optional is either empty or:
  variable

continue-variable/optional is either empty or:
  variable

clause is one of:
  `(`*test body-expr* ...[1]`)`
  `(`*test* `=>` *procedure-expr*`)`
  `(/` *regexp-str* `/ (`*variable-or-false* ...[1]`)` *body-expr* ...[1]`)`
  `(range` *exclusive-start-test exclusive-stop-test body-expr* ...[1]`)`
  `(:range` *inclusive-start-test exclusive-stop-test body-expr* ...[1]`)`
  `(range:` *exclusive-start-test inclusive-stop-test body-expr* ...[1]`)`
  `(:range:` *inclusive-start-test inclusive-inclusive-stop-test body-expr* ...[1]`)`
  `(else` *body-expr* ...[1]`)`
  `(after` *body-expr* ...[1]`)`

test is one of:
  integer
  regexp-str
  expr

variable-or-false is one of:
  variable
  `#f`

For detailed information about `awk`, see Olin Shivers's *Scsh Reference Manual*. In addition to `awk`, the Scsh-compatible procedures `match:start`, `match:end`, `match:substring`, and `regexp-exec` are defined. These `match:` procedures must be used to extract match information in a regular expression clause when using the `=>` form.

15.2.2 Command-line Parsing: cmdline.ss

Files: **cmdline.ss**, **cmdlineu.ss**, **cmdliner.ss**, **cmdlines.ss**
Signature: `mzlib:command-line^`
Unit: `mzlib:command-line@`

`(command-line` *program-name-str argv-expr clause* ...`)` SYNTAX

Parses a command line according to the specification in the *clause*s. The *program-name-str* string is used as the program name for reporting errors when the command-line is ill-formed. The *argv-expr* must evaluate to a vector of strings, which is typically the value of `argv` as defined by the MzScheme stand-alone application.

The command-line is disassembled into flags (possibly with flag-specific arguments) followed by (non-flag) arguments. Command-line strings starting with "-" or "+" are parsed as flags, but arguments to flags

are never parsed as flags, and integers and decimal numbers that start with "-" or "+" are not treated as flags. Non-flag arguments in the command-line must appear after all flags and the flags' arguments. No command-line string past the first non-flag argument is parsed as a flag. The built-in `--` flag signals the end of command-line flags; any command-line string past the `--` flag is parsed as a non-flag argument.

For defining the command line, each *clause* has one of the following forms:

`(multi` *flag-spec* ...`)`
`(once-each` *flag-spec* ...`)`
`(once-any` *flag-spec* ...`)`
`(args` *arg-formals body-expr* ...[1]`)`
`(=>` *finish-proc-expr arg-help-expr help-proc-expr unknown-proc-expr*`)`

flag-spec is one of:
`(`*flags variable* ...*help-str body-expr* ...[1]`)`
`(`*flags* `=>` *handler-expr help-expr*`)`

flags is one of:
flag-str
`(`*flag-str* ...[1]`)`

arg-formals is one of:
variable
`(`*variable* ...`)`
`(`*variable* ...[1]`.` *variable*`)`

A `multi`, `once-each`, or `once-any` clause introduces a set of command-line flag specifications. The clause tag indicates how many times the flag can appear on the command line:

- `multi` — Each flag specified in the set can be represented any number of times on the command line; i.e., the flags in the set are independent and each flag can be used multiple times.
- `once-each` — Each flag specified in the set can be represented once on the command line; i.e., the flags in the set are independent, but each flag should be specified at most once. If a flag specification is represented in the command line more than once, the `exn:user` exception is raised.
- `once-any` — Only one flag specified in the set can be represented on the command line; i.e., the flags in the set are mutually exclusive. If the set is represented in the command line more than once, the `exn:user` exception is raised.

A normal flag specification has four parts:

1. *flags* — a flag string, or a set of flag strings. If a set of flags is provided, all of the flags are equivalent. Each flag string must be of the form `"-`*x*`"` or `"+`*x*`"` for some character *x*, or `"--`*x*`"` or `"++`*x*`"` for some sequence of characters *x*. An *x* cannot contain only digits or digits plus a single decimal point, since simple (signed) numbers are not treated as flags. In addition, the flags `"--"`, `"-h"`, and `"--help"` are predefined and cannot be changed.
2. *variable*s — variables that are bound to the flag's arguments. The number of variables specified here determines how many arguments can be provided on the command line with the flag, and the names of these variables will appear in the help message describing the flag. The *variable*s are bound to string values in the *body-expr*s for handling the flag.

3. *help-str* — a string that describes the flag. This string is used in the help message generated by the handler for the built-in `-h` (or `--help`) flag.

4. *body-expr*s — expressions that are evaluated when one of the *flags* appears on the command line. The flags are parsed left-to-right, and each sequence of *body-expr*s is evaluated as the corresponding flag is encountered. When the *body-expr*s are evaluated, the *variable*s are bound to the arguments provided for the flag on the command line.

A flag specification using `=>` escapes to a more general method of specifying the handler and help strings. In this case, the handler procedure and help string list returned by *handler-expr* and *help-expr* are embedded directly in the table for `parse-command-line`, the procedure used to implement command-line parsing.

An `args` clause can be specified as the last clause. The variables in *arg-formals* are bound to the leftover command-line strings in the same way that variables are bound to the *formals* of a `lambda` expression. Thus, specifying a single *variable* (without parentheses) collects all of the leftover arguments into a list. The effective arity of the *arg-formals* specification determines the number of extra command-line arguments that the user can provide, and the names of the variables in *arg-formals* are used in the help string. When the command-line is parsed, if the number of provided arguments cannot be matched to variables in *arg-formals*, the `exn:user` exception is raised. Otherwise, `args` clause's *body-expr*s are evaluated to handle the leftover arguments.

Instead of an `args` clause, the `=>` clause can be used to escape to a more general method of handling the leftover arguments. In this case, the values of the expressions with `=>` are passed on directly as arguments to `parse-command-line`. The *help-proc-expr* and *unknown-proc-expr* expresisons are optional.

Example:
```
(command-line "compile" argv
  (once-each
     [("-v" "--verbose") "Compile with verbose messages"
                         (verbose-mode #t)]
     [("-p" "--profile") "Compile with profiling"
                         (profiling-on #t)])
  (once-any
     [("-o" "--optimize-1") "Compile with optimization level 1"
                            (optimize-level 1)]
     ["--optimize-2"        "Compile with optimization level 2"
                            (optimize-level 2)])
  (multi
     [("-l" "--link-flags") lf ; flag takes one argument
                            "Add a flag for the linker" "flag"
                            (link-flags (cons lf (link-flags)))])
  (args (filename) ; expects one command-line argument: a filename
    filename)) ; return a single filename to compile
```

`(parse-command-line` *progname argv table finish-proc arg-help* [*help-proc unknown-proc*]`)` PROCEDURE

Parses a command-line using the specification in *table*. For an overview of command-line parsing, see the `command-line` form. The *table* argument to this procedural form encodes the information in `command-line`'s clauses, except for the `args` clause. Instead, arguments are handled by the *finish-proc* procedure, and help information about non-flag arguments is provided in *arg-help*. In addition, the *finish-proc* procedure receives information accumulated while parsing flags. The *help-proc* and *unknown-proc* arguments allow customization that is not possible with `command-line`.

When there are no more flags, the *finish-proc* procedure is called with a list of information accumulated for

command-line flags (see below) and the remaining non-flag arguments from the command-line. The arity of the *finish-proc* procedure determines the number of non-flag arguments accepted and required from the command-line. For example, if *finish-proc* accepts either two or three arguments, then either one or two non-flag arguments must be provided on the command-line. The *finish-proc* procedure can have any arity (see §4.10.1) except `0` or a list of `0`s (i.e., the procedure must at least accept one or more arguments).

The *arg-help* argument is a list of strings identifying the expected (non-flag) command-line arguments, one for each argument. (If an arbitrary number of arguments are allowed, the last string in *arg-help* represents all of them.)

The *help-proc* procedure is called with a help string if the `-h` or `--help` flag is included on the command line. If an unknown flag is encountered, the *unknown-proc* procedure is called just like a flag-handling procedure (as described below); it must at least accept one argument (the unknown flag), but it may also accept more arguments. The default *help-proc* displays the string and exits and the default *unknown-proc* raises the `exn:user` exception.

A *table* is a list of flag specification sets. Each set is represented as a list of two items: a mode symbol and a list of flag specifications. A mode symbol is one of `'once-each`, `'once-any`, or `'multi`, with the same meanings as the corresponding clause tags in `command-line`. Each specification maps a number of flags to a single handler procedure. A specification is a list of three items:

1. A list of strings for the flags defined by the spec. See `command-line` for information about the format of flag strings.

2. A procedure to handle the flag and its arguments when one of the flags is found on the command line. The arity of this handler procedure determines the number of arguments consumed by the flag: the handler procedure is called with a flag string plus the next few arguments from the command line to match the arity of the handler procedure. The handler procedure must accept at least one argument to recieve the flag. If the handler accepts arbitrarily many arguments, all of the remaining arguments are passed to the handler. A handler procedure's arity must either be a number or an `arity-at-least` value (see §4.10.1).

   The return value from the handler is added to a list that is eventually passed to *finish-proc*. If the handler returns void, no value is added onto this list. For all non-void values returned by handlers, the order of the values in the list is the same as the order of the arguments on the command-line.

3. A non-empty list of strings used for constructing help information for the spec. The first string in the list describes the flag, and additional strings name the expected arguments for the flag. The number of extra help strings provided for a spec must match the number of arguments accepted by the spec's handler procedure.

The following example is the same as the example for `command-line`, translated to the procedural form:

```
(parse-command-line "compile" argv
  `((once-each
     [("-v" "--verbose")
      ,(lambda (flag) (verbose-mode #t))
      ("Compile with verbose messages")]
     [("-p" "--profile")
      ,(lambda (flag) (profiling-on #t))
      ("Compile with profiling")])
    (once-any
     [("-o" "--optimize-1")
      ,(lambda (flag) (optimize-level 1))
      ("Compile with optimization level 1")]
```

Compatible	MzScheme
=?	=
<?	<
>?	>
<=?	<=
>=?	>=
1+	add1
1-	sub1
#%1+	#%add1
#%1-	#%sub1
gentemp	gensym
bound?	defined?
flush-output-port	flush-output
real-time	current-milliseconds

Figure 15.1: Compatibility aliases

```
  [("--optimize-2")
   ,(lambda (flag) (optimize-level 2))
   ("Compile with optimization level 2")])
 (multi
  [("-l" "--link-flags")
   ,(lambda (flag lf) (link-flags (cons lf (link-flags))))
   ("Add a flag for the linker" "flag")]))
(lambda (flag-accum file) file)  ; return a single filename to compile
'("filename")) ; expects one command-line argument: a filename
```

15.2.3 Compatibility: compat.ss

Files: **compat.ss**, **compatu.ss**, **compatr.ss**, **compats.ss**
Requires: **functios.ss**
Opened form requires: **functiou.ss**
Syntactic forms only: **compatm.ss**
Signature: `mzlib:compat^`
Unit: `mzlib:compat@`, imports `mzlib:function^`

This library defines a number of procedures and syntactic forms that are commonly provided by other Scheme implementations. Most of the procedures are aliases for built-in MzScheme procedures, as shown in Figure 15.1. The remaing procedures and forms are described below.

`(atom?` *v*`)` PROCEDURE

Same as `(not (pair?` *v*`))`.

`(defmacro` *name formals body-expr* ...[1]`)` SYNTAX

Expands into an equivalent `define-macro` expression.

`(getprop` *sym property default*`)` PROCEDURE

Gets a property value associated with the symbol *sym*. The *property* argument is also a symbol that names

the property to be found. If the property is not found, *default* is returned. If the *default* argument is omitted, `#f` is used as the default.

(`letmacro` *name formals macro-body body-expr* ...[1]) SYNTAX

Expands into an equivalent `let-macro` expression.

(`new-cafe` [*eval-handler*]) PROCEDURE

Emulates Chez Scheme's `new-cafe`.

(`putprop` *sym property value*) PROCEDURE

Installs a value for *property* of the symbol *sym*. See `getprop` above.

(`sort` *less-than?-proc list*) PROCEDURE

This is the same as `quicksort` (see §15.2.9) with the arguments reversed.

15.2.4 Compiling Files: compile.ss

Files: **compile.ss**, **compileu.ss**, **compiler.ss**, **compiles.ss**
Signature: `mzlib:compile^`
Unit: `mzlib:compile@`

(`compile-file` *src dest* [*flags preprocess*]) PROCEDURE

Compiles the Scheme file(s) *src* and saves the compiled code to *dest*. The *src* and *dest* arguments are usually filenames.[3] The *src* argument can also be an input port or a list of filenames and/or input ports. If a list of inputs is provided for *src*, they are compiled to *dest* as if they were concatenated (in order) in a single input. The *dest* argument can also be an output port. If an input or output port is provided, the port is *not* closed when compilation is finished. If *dest* is a filename and the file already exists, the existing file is truncated (if possible) or replaced.

The *flags* argument is a list of symbol flags, defaultly `null`. The possible flags are:

- `'ignore-macro-definitions` — disables special handling for `define-macro`, `define-id-macro`, and `define-expansion-time` forms (and their `#%` forms; see below).
- `'strip-macro-definitions` — performs special handling for `define-macro`, `define-id-macro`, and `define-expansion-time` (and their `#%` forms; see below), but no compiled expression is written to the compiled file.
- `'expand-load` — top-level `load`, `load-relative`, and `load/cd` expressions in the source(s) are expanded and compiled directly into *dest*. Load-expansion is performed recusively within loaded files.
- `'expand-require-library` — top-level `require-library` and `#%require-library` expressions in the source(s) are replaced with the source of the library.
- `'ignore-require-library` — top-level `require-library` expressions are not handled specially; the default handling of `require-libarary` is described below.

[3] A MzScheme compiled code file should use the extension **.zo**, but this convention is not enforced.

- `'preserve-elaborations` — instead of evaluating the expressions within `begin-elaboration-time` expressions at macro-expansion-time, the expressions are preserved and evaluated when the compiled expression is loaded; the result of this run-time evaluation is then itself evaluated to get semantics similar to the usual `begin-elaboration-time` semantics. Thus, `(begin-elaboration-time` *expr* $\cdots^1$`)` becomes `(eval (eval (quote (begin` *expr* $\cdots^1$`))))`. This flag does *not* affect the expansion of `#%begin-elaboration-time` forms. The output file starts with `'e` followed by a space, an annotation used by `load/use-compiled` when the `use-compiled-file-kinds` parameter is set to `'non-elaboration` (see §9.4.1.7).

- `'also-preserve-elaborations` — like `'preserve-elaborations`, but the expressions in a `begin-elaboration-time` form are also eveluated at expansion time.

- `'preserve-constructions` — modifies the semantics of `begin-construction-time` in the same way that `'preserve-elaborations` modifies `begin-elaboration-time`.

- `'also-preserve-constructions` — like `'preserve-constructions`, but the expressions in a `begin-construction-time` form are also eveluated at expansion time.

- `'only-expand` — expands, but does not compile, the expressions from the source(s).

- `'no-warnings` — suppresses warnings (written to the current error port) about ill-formed top-level `require-library` or `load` expressions that are recognized but cannot be expanded.

- `'use-current-namespace` — does not create a new namespace for compiling the source(s). The use of a separate namespace is described below.

The *preprocess* argument is a procedure of two arguments: an S-expression and a namespace (§9.3). The *preprocess* procedure is applied to each expression read from the source(s), and the provided namespace is where expressions are expanded and compiled. The expression returned by *preprocess* is expanded, compiled, and written to the output destination. The default *preprocess* procedure returns its first aargument.

To compile the input expressions, `compile-file` normally creates a new namespace and copies all of the top-level bindings from the current namespace into the new namespace (except for the bindings of built-in-identifiers), without setting any variables as keywords (see §9.3.2). The `'use-current-namespace` flag causes the current namespace to be used for expansion and compilation, instead.

By default, top-level `require-library` expressions are recognized, and the specified library is loaded into the compilation namespace. This causes library-based macros to be loaded for compiling the file, but the `require-library` expression is still compiled as an invocation of `require-library`. If the `'expand-require-library` flag is specified, `compile-file` instead replaces the call to `require-library` with the compiled contents of the library if the specified library has not yet been included during the call to `compile-file`.

All macros used in the source file must either be defined at compile-time, defined using a top-level `define-macro` statement in the source file (or in an `load`ed or `require-library`ed file that is expanded), or defined locally in a `let-macro` expression. Macros can be defined within the compiled file because `define-macro`, `define-id-macro`, and `define-expansion-time` forms are treated specially: these forms are evaluated by `compile-file` after they are compiled (in the special compilation namespace), so that the effect of these expressions is visible while compiling the rest of the file. If the `'ignore-macro-definitions` flag is used, then all macros used in the source file must be predefined or defined using `let-macro`.

15.2.5 Core: core.ss

Files: **core.ss**, **coreu.ss**, **corer.ss**, **cores.ss**
Signature: `mzlib:core^`
Unit: `mzlib:core@`
Requires: see §15.1

The `mzlib:core^` signature is defined by:

```
(define-signature mzlib:core^
  ((unit pretty-print : mzlib:pretty-print^)
   (unit file : mzlib:file^)
   (unit function : mzlib:function^)
   (unit string : mzlib:string^)
   (unit compile : mzlib:compile^)
   (unit math : mzlib:math^)
   (unit thread : mzlib:thread^)))
```

The `mzlib:core@` unit implements this signature by linking the [CORE] library units together.

15.2.6 Dates: date.ss

Files: **date.ss**, **dateu.ss**, **dater.ss**, **dates.ss**
Requires: **functios.ss**
Opened form requires: **functiou.ss**
Signature: `mzlib:date^`
Unit: `mzlib:date@`, imports `mzlib:function^`

See also §11.4.

`(date->string` *date* [*time?*]`)` PROCEDURE

Converts a date structure value (such as returned by MzScheme's `seconds->date`) to a string. The returned strong contains the time of day only if *time?* is a true value; the default is `#f`. See also `date-display-format`.

`(date-display-format` [*format-symbol*]`)` PROCEDURE

Parameter that determines the date display format, one of `'american`, `'chinese`, `'german`, `'indian`, `'irish`, or `'julian`. The initial format is `'american`.

`(find-seconds` *second minute hour day month year*`)` PROCEDURE

Finds the representation of a date in platform-specific seconds. The arguments correspond to the fields of the `date` structure. If the platform cannot represent the specified date, an error is signalled, otherwise an integer is returned.

`(date->julian/scalinger` *date*`)` PROCEDURE

Converts a date structure (up to 2099 BCE Gregorian) into a Julian date number. The returned value is not a strict Julian number, but rather Scalinger's version, which is off by one for easier calculations.

`(julian/scalinger->string` *date*`)` PROCEDURE

Converts a Julian number (Scalinger's off-by-one version) into a string.

15.2.7 Structures: defstru.ss

Files: **defstru.ss**

This library provides `define-structure` and `define-const-structure` consistent with the **match.ss** library (see §15.2.13). The `define-structure` and `define-const-structure` forms expand into `define-struct` forms. While `define-const-structure` and the mutability annotations are supported, immutable fields are *not* actually immutable.[4]

15.2.8 Filesystem: file.ss

Files: **file.ss**, **fileu.ss**, **filer.ss**, **files.ss**
Requires: **string.ss**, **functio.ss**
Opened form requires: **stringu.ss**, **functiou.ss**
Signature: `mzlib:file^`
Unit: `mzlib:file@`, imports `mzlib:string^`, `mzlib:function^`

See also §11.2.

`(build-absolute-path` *base path* ···`)` PROCEDURE

Like `build-path` (see §11.2), but *base* is required to be an absolute pathname. If *base* is not an absolute pathname, `error` is called.

`(build-relative-path` *base path* ···`)` PROCEDURE

Like `build-path` (see §11.2), but *base* is required to be a relative pathname. If *base* is not a relative pathname, `error` is called.

`(delete-directory/files` *path*`)` PROCEDURE

Deletes the file or directory specified by *path*. If *path* is a directory, then `delete-directory/files` is first applied to each file and directory in *path* before the directory is deleted. If *path* cannot be deleted, the `exn:user` exception is raised, otherwise void is returned.

`(explode-path` *path*`)` PROCEDURE

Returns the list of directories that constitute *path*. The *path* argument must be normalized (see `normalize-path` below).

`(file-name-from-path` *path*`)` PROCEDURE

If *path* is a file pathname, returns just the file name part without the directory path.

[4] A properly implemented `define-const-struct` is not inconsistent with `define-struct`. It simply has not been implemented, yet.

`(filename-extension` *path*`)` PROCEDURE

Returns a string that is the extension part of the filename in *path*. If *path* is (syntactically) a directory, `#f` is returned.

`(find-library` *name collection*`)` PROCEDURE

Returns the path of the specified library (see Chapter 15), returning `#f` if the specificed library or collection cannot be found. The *collection* argument is optional, defaulting to `"mzlib"`.

`(find-relative-path` *basepath path*`)` PROCEDURE

Finds a relative pathname with respect to *basepath* that names the same file or directory as *path*. Both *basepath* and *path* must be normalized (see `normalize-path` below). If *path* is not a proper subpath of *basepath* (i.e., a subpath that is strictly longer), *path* is returned.

`(make-directory*` *path*`)` PROCEDURE

Creates directory specified by *path*, creating intermediate directories as necessary.

`(make-temporary-file` [*format-string*]`)` PROCEDURE

Creates a new temporary file and returns a pathname string for the file. Instead of merely generating a fresh file name, the file is actually created; this prevents other threads or processes from picking the same temporary name. However, the file is not opened for reading or writing when the pathname is returned. The client program calling `make-temporary-file` is expected to open the file with the desired access and flags (probably using the `'truncate` flag; see §11.1.2) and to delete it when it is no longer needed.

If *format-string* is specified, it must be a format string suitable for use with `format` and one additional string argument. The default *format-string* is `"mztmp~a"`.

`(normalize-path` *path wrt*`)` PROCEDURE

Returns a normalized, complete version of *path*, expanding the path and resolving all soft links. If *path* is relative, then the pathname *wrt* is used as the base path. The *wrt* argument is optional; if is omitted, then the current directory is used as the base path.

Letter case is *not* normalized by `normalize-path`, so combine `normalize-path` with `normal-case-path` to get strings for path comparison.

An error is signaled by `normalize-path` if the input path contains an embedded path for a non-existant directory, or if an infinite cycle of soft-links is detected.

`(path-only` *path*`)` PROCEDURE

If *path* is a filename, the file's path is returned. If *path* is syntactically a directory, `#f` is returned.

15.2.9 Functions: functio.ss

Files: **functio.ss**, **functiou.ss**, **functior.ss**, **functios.ss**
Signature: `mzlib:function^`
Unit: `mzlib:function@`

The procedures `second`, `third`, `fourth`, `fifth`, `sixth`, `seventh`, and `eighth` access the corresponding element from a list.

`(assf` *f l*`)` PROCEDURE

Applies *f* to each element of *l* (from left to right) until *f* returns a true value for some element, in which case that element is returned. If *f* does not return a true value for any element of *l*, `#f` is returned.

`(boolean=?` *bool1 bool2*`)` PROCEDURE

Return `#t` if *bool1* and *bool2* are both `#t` or both `#f`, and returns `#f` otherwise. If either *bool1* or *bool2* is not a Boolean, `exn:application:type` exception is raised.

`(build-list` *n f*`)` PROCEDURE

Creates a list of *n* elements by applying *f* to the integers from `0` to $n-1$ in order, where *n* is a non-negative integer. The *i*th element of the resulting list is `(`*f* `(- i 1))`.

`(build-string` *n f*`)` PROCEDURE

Creates a string of length *n* by applying *f* to the integers from `0` to $n-1$ in order, where *n* is a non-negative integer and *f* returns a character for the *n* invocations. The *i*th character of the resulting string is `(`*f* `(- i 1))`.

`(build-vector` *n f*`)` PROCEDURE

Creates a vector of *n* elements by applying *f* to the integers from `0` to $n-1$ in order, where *n* is a non-negative integer. The *i*th element of the resulting vector is `(`*f* `(- i 1))`.

`(compose` *f g*`)` PROCEDURE

Returns a procedure that takes `x` and returns `(call-with-values (lambda () (g x)) f)`.

`(cons?` *v*`)` PROCEDURE

Returns `#t` if *v* is a value created with `cons`, `#f` otherwise.

`empty` EMPTY LIST

The empty list.

`(empty?` *v*`)` PROCEDURE

Returns `#t` if *v* is the empty list, `#f` otherwise.

`(filter` *f l*`)` PROCEDURE

Applies *f* to each element in *l* (from left to right) and returns a new list that is the same as *l*, but omitting all the elements for which *f* returned `#f`.

`(first` *l*`)` PROCEDURE

Returns the first element of the list *l*. (The `first` procedure is a synonym for `car`.)

`(foldl` *f init l* ...[1]`)` PROCEDURE

Like `map`, `foldl` applies a procedure *f* to the elements of one or more lists. While `map` combines the return values into a list, `foldl` combines the return values in an abitrary way that is determined by *f*.

If `foldl` is called with *n* lists, the *f* procedure takes *n*+1 arguments. The extra value is the combined return values so far. The *f* procedure is initially invoked with the first item of each list; the final argument is *init*. In subsequent invocations of *f*, the last argument is the return value from the previous invocation of *f*. The input lists are traversed from left to right, and the result of the whole `foldl` application is the result of the last application of *f*. (If the lists are empty, the result is *init*.)

For example, `reverse` can be defined in terms of `foldl`:

```
(define reverse
  (lambda (l)
    (foldl cons '() l)))
```

`(foldr` *f init l* ...[1]`)` PROCEDURE

Like `foldl`, but the lists are traversed from right to left.

For example, a restricted `map` (that works only on single-argument procedures) can be defined in terms of `foldr`:

```
(define simple-map
  (lambda (f list)
    (foldr (lambda (v l) (cons (f v) l)) '() list)))
```

`(identity` *v*`)` PROCEDURE

Returns *v*.

`(ignore-errors` *thunk*`)` PROCEDURE

Invokes *thunk* and returns the result. If an error occurs during the application of *thunk*, no error is reported and void is returned.

`(last-pair` *list*`)` PROCEDURE

Returns the last pair in *list*, raising an error if *list* is not a pair (but *list* does not have to be a proper list).

`(loop-until` *start done? next f*`)` PROCEDURE

Repeatedly invokes the *f* procedure until the *done?* procedure returns `#t`. The procedure is best described by its implementation:

```
(define loop-until
  (lambda (start done? next f)
```

```
(let loop ([i start])
  (unless (done? i)
    (f i)
    (loop (next i))))))
```

(memf *f l*) PROCEDURE

Applies *f* to each element of *l* (from left to right) until *f* returns a true value for some element, in which case the tail of *l* starting with that element is returned. If *f* does not return a true value for any element of *l*, `#f` is returned.

(quicksort *list less-than?*) PROCEDURE

Sorts *list* using the comparison procedure *less-than?*. This implementation is not stable (i.e., if two elements in the input are "equal," their relative positions in the output may be reversed).

(remove *item list* [*equal?*]) PROCEDURE

Returns *list* without the first instance of *item*, where an instance is found by comparing *item* to the list items using *equal?*. The default value for *equal?* is `equal?`. When *equal?* is invoked, *item* is the first argument.

(remove* *items list* [*equal?*]) PROCEDURE

Like `remove`, except that the first argument is a list of items to remove, instead of a single item.

(remq *item list*) PROCEDURE

Calls `remove` with `eq?` as the comparison procedure.

(remq* *items list*) PROCEDURE

Calls `remove*` with `eq?` as the comparison procedure.

(remv *item list*) PROCEDURE

Calls `remove` with `eqv?` as the comparison procedure.

(remv* *items list*) PROCEDURE

Calls `remove*` with `eqv?` as the comparison procedure.

(rest *l*) PROCEDURE

Returns a list that contains all but the first element of the non-empty list *l*. (The `rest` procedure is a synonym for `cdr`.)

15.2.10 Inflating Compressed Data: inflate.ss

Files: **inflate.ss**, **inflateu.ss**, **inflater.ss**, **inflates.ss**
Signature: `mzlib:inflate^`
Unit: `mzlib:inflate@`

(`gunzip` *file* [*output-name-filter*]) PROCEDURE

Extracts data that was compressed using the GNU `gzip` utility, writing the uncompressed data directly to a file. The *file* argument is the name of the file containing compressed data. The default output file name is the original name of the compressed file as stored in *file*. If a file by this name exists, it will be overwritten. If no original name is stored in the source file, `"unzipped"` is used as the default output file name.

The *output-name-filter* procedure is applied to two arguments — the default destination file name and a Boolean that is `#t` if this name was read from *file* — before the destination file is created. The return value of the file is used as the actual destination file name (opened with the `'truncate` flag). The default *output-name-filter* procedure returns its first argument.

The return value is void. If the compressed data is corrupted, the `exn:user` exception is raised.

(`gunzip-through-ports` *in out*) PROCEDURE

Reads the port *in* for compressed data that was created using the GNU `gzip` utility, writing the uncompressed data to the port *out*.

The return value is void. If the compressed data is corrupted, the `exn:user` exception is raised.

(`inflate` *in out*) PROCEDURE

Reads `pkzip`-format "deflated" data from the port *in* and writes the uncompressed ("inflated") data to the port *out*. The data in a file created by `gzip` uses this format (preceded with some header information).

The return value is void. If the compressed data is corrupted, the `exn:user` exception is raised.

15.2.11 Invoking with Exports to a Namespace: invoke.ss

Files: **invoke.ss**

(`define-values/invoke-unit` (*export-id* ···) *unit-expr* [*prefix import-id* ···]) SYNTAX

Similar to `invoke-unit`. However, instead of returning the value of the unit's initialization expression, `define-values/invoke-unit` expands to a `define-values` expression that binds each identifier *export-id* to the value of the corresponding variable exported by the unit. At run time, if the unit does not export all of the *export-id*s, the `exn:unit` exception is raised.

If *prefix* is specified, it must be either `#f` or an identifier. If it is an indentifier, the names defined by the expansion of `define-values/invoke-unit` are prefixed with *prefix*`:`.

Example:

```
(define x 3)
(define y 2)
```

```
(define-values/invoke-unit (c)
   (unit (import a b)
         (export c)
      (define c (- a b)))
   ex
   x y)
ex:c ; ⇒ 1
```

`(define-values/invoke-unit/sig (`*signature* ···`)` *unit/sig-expr* [*prefix invoke-linkage* ···]`)` SYNTAX

The signed-unit version of `define-values/invoke-unit`. The names defined by the expansion of `define-values/invoke-unit/sig` are determined by flattening the *signature* specified before *unit-expr*, then adding the *prefix* (if any). See §7.3.1.1 for more information about signature flattening.

Each *invoke-linkage* is either *signature* or (*identifier* `:` *signature*), as in `invoke-unit/sig`.

`(global-define-values/invoke-unit (`*export-id* ···`)` *unit-expr* [*prefix import-id* ···]`)` SYNTAX

Like `define-values/invoke-unit`, but the expansion is a sequence of calls to `global-defined-value` instead of a `define-values` expression. Thus, when it is evaluated, a `global-define-values/invoke-unit` expression binds top-level variables in the current namespace.

`(global-define-values/invoke-unit/sig (`*signature* ···`)` *unit/sig-expr* [*prefix invoke-linkage* ···]`)` SYNTAX

The signed-unit version of `global-define-values/invoke-unit`. See also `define-values/invoke-unit/sig`.

15.2.12 Macros: macro.ss

Files: **macro.ss**

`(class-asi` *superclass clauses* ···`)` SYNTAX

Like `class`, but the initialization arguments are automatically passed on to the superclass initialization procedure.

`(class*-asi` *superclass interfaces clauses* ···`)` SYNTAX

Like `class*`, but the initialization arguments are automatically passed on to the superclass initialization procedure.

`(evcase` *key-expr* `(`*value-expr body-expr* ···`)` ···[1]`)` SYNTAX

The `evcase` form is similar to `case`, except that expressions are provided in each clause instead of a sequence of data. After *key-expr* is evaluated, each *value-expr* is evaluated until a value is found that is `eqv?` to the key value; when a matching value is found, the corresponding *body-expr*s are evaluated and the value(s) for the last is the result of the entire `evcase` expression.

A *value-expr* can be the special identifier `else`. This identifier is recognized as in `case` (see §3.2).

`(let+` *clause body-expr* $\cdots^1$`)` SYNTAX

A new binding construct that specifies scoping on a per-binding basis instead of a per-expression basis. It helps eliminate rightward-drift in programs. It looks a lot like `let`, except each clause has an additional keyword tag before the binding variables.

Each *clause* has one of the following forms:

- `(val` *variable expr*`)` binds *variable* non-recursively to *expr*.
- `(rec` *variable expr*`)` binds *variable* recursively to *expr*.
- `(vals (`*variable expr*`)` $\cdots$`)` the *variable*s are bound to the *expr*s. The environment of the *expr*s is the environment active before this clause.
- `(recs (`*variable expr*`)` $\cdots$`)` the *variable*s are bound to the *expr*s. The environment of the *expr*s includes all of the *variable*s.
- `(_` *expr* $\cdots$`)` evaluates the *expr*s without binding any variables.

The clauses bind left-to-right. Each *variable* above can either be an identifier or `(values` *variable* $\cdots$`)`. In the latter case, multiple values returned by the corresponding expression are bound to the mulitple variables.

Examples:

```
(let+ ([val (values x y) (values 1 2)])
   (list x y)) ; => (1 2)
(let ([x 1])
   (let+ ([val x 3]
          [val y x])
      y)) ; => 3
```

`(local (`*definition* $\cdots$`)` *body-expr* $\cdots^1$`)` SYNTAX

This is a binding form similar to `letrec`, except that each *definition* is a `define-values`, `define`, or `define-struct` expression (*before* macro-expansion). The *body-expr*s are evaluated in the lexical scope of these definitions.

`(nand` *expr* $\cdots$`)` SYNTAX

Returns `(not (and` *expr* $\cdots$`))`.

`(nor` *expr* $\cdots$`)` SYNTAX

Returns `(not (or` *expr* $\cdots$`))`.

`(opt-lambda` *formals body-expr* $\cdots^1$`)` SYNTAX

The `opt-lambda` form is like `lambda`, except that default values are assigned to arguments (C++-style). Default values are defined in the *formals* list by replacing each *variable* by `[`*variable default-value-expression*`]`. If an variable has a default value expression, then all (non-aggregate) variables after it must have default value expressions. A final aggregate variable can be used as in `lambda`, but it cannot be given a default value. Each default value expression is evaluated only if it is needed. The environment of each default value expression includes the preceding arguments.

For example:

```
(define f
   (opt-lambda (a [b (add1 a)] .  c)
      ...))
```

Here, `f` is a procedure which takes at least one argument. If a second argument is specified, it is the value of `b`, otherwise `b` is `(add1 a)`. If more than two arguments are specified, then the extra arguments are placed in a new list that is the value of `c`.

`(recur` *name bindings body-expr* $\cdots^1$`)` SYNTAX

This is equivalent to a named `let`: `(let` *name bindings body-expr* $\cdots^1$`)`.

`(send*` *obj msg* $\cdots$`)` SYNTAX

Calls multiple methods of *obj* (in the specified order). Each *msg* should have the form:

```
(name params ...)
```

where *name* is the method name. For example:

```
(send* edit (begin-edit-sequence)
            (insert "Hello")
            (insert #\newline)
            (end-edit-sequence))
```

is the same as

```
(let ([e edit])
  (send e begin-edit-sequence)
  (send e insert "Hello")
  (send e insert #\newline)
  (send e end-edit-sequence))
```

`(signature->symbols` *name*`)` SYNTAX

Expands to the "exploded" version (see §7.4.3) of the signature currently bound to *name* (where *name* is an unevaluated identifier). The expansion-time value of *name* (see §13.3) must have the shape of an exploded signature.

15.2.13 Match: match.ss

Files: **match.ss**

This is the pattern matching system by Andrew Wright and Bruce Duba. See *Pattern Matching for Scheme* for details. The following syntactic forms are defined by the **match.ss** library:

- `match`
- `match-lambda`
- `match-lambda*`

- `match-let`
- `match-let*`
- `match-letrec`
- `match-define`

All forms of `match` can be used with `define-struct` values, as well as `define-structure` and `define-const-structure` values (see §15.2.7).

The default `match:error` procedure raises the `exn:misc:match` exception, which extends `exn:misc` with value field (for the non-matching value).

15.2.14 Math: math.ss

Files: **math.ss**, **mathu.ss**, **mathr.ss**, **maths.ss**
Signature: `mzlib:math^`
Unit: `mzlib:math@`

`(conjugate` *z*`)` PROCEDURE

Returns the complex conjugate of *z*.

`(cosh` *z*`)` PROCEDURE

Returns the hyperbolic cosine of *z*.

`e` NUMBER

Approximation of Euler's number, equivalent to `(exp 1.0)`.

`pi` NUMBER

Approximation of π, equivalent to `(atan 0.0 -1.0)`.

`(sinh` *z*`)` PROCEDURE

Returns the hyperbolic sine of *z*.

`(sgn` *n*`)` PROCEDURE

Returns `1` if *n* is positive, `-1` if *n* is negative, `0` otherwise.

15.2.15 MzLib: mzlib.ss

Files: **mzlib.ss**, **mzlibu.ss**, **mzlibr.ss**, **mzlibs.ss**
Signature: `mzlib^`
Unit: `mzlib@`
Requires: see §15.1

The `mzlib^` signature is defined by:

```
(define-signature mzlib^
  ((open mzlib:core^)
   (unit compat : mzlib:compat^)
   (unit print-convert : mzlib:print-convert^)
   (unit date : mzlib:date^)
   (unit inflate : mzlib:inflate^)
   (unit command-line : mzlib:command-line^)
   (unit restart : mzlib:restart^)))
```

The `mzlib@` unit implements this signature by linking the `mzlib:core@` unit together with the non-[CORE] standard library units.

15.2.16 Converted Printing: pconver.ss

Files: **pconver.ss**, **pconveru.ss**, **pconverr.ss**, **pconvers.ss**
Requires: **strings.ss**, **functios.ss**
Opened form requires: **stringu.ss**, **functiou.ss**
Signature: `mzlib:print-convert^`
Unit: `mzlib:print-convert@`, imports `mzlib:string^`, `mzlib:function^`

This library defines routines for printing Scheme values as `eval`uatable S-expressions rather than `read`able S-expressions. The `print-convert` procedure does not print values; rather, it converts a Scheme value into another Scheme value such that the new value pretty-prints as a Scheme expression that evaluates to the original value. For example, `(pretty-print (print-convert '(9 ,(box 5) #(6 7))))` prints the literal expression `(list 9 (box 5) (vector 6 7))` to the current output port.

To install print converting into the `read-eval-print` loop, require **pconver.ss** and call the procedure `install-converting-printer`.

In addition to `print-convert`, this library provides `print-convert`, `build-share`, `get-shared`, and `print-convert-expr`. The last three are used to convert sub-expressions of a larger expression (potentially with shared structure).

`(abbreviate-cons-as-list` [*abbreviate?*]`)` PROCEDURE

Parameter that controls how lists are represented with constructor-style conversion. If the parameter's value is `#t`, lists are represented using `list`. Otherwise, lists are represented using `cons`. The initial value of the parameter is `#t`.

`(build-share` *v*`)` PROCEDURE

Takes a value and computes sharing information used for reprsenting the value as an expression. The return value is an opaque structure that can be passed back into `get-shared` or `print-convert-expr`.

`(constructor-style-printing` [*use-constructors?*]`)` PROCEDURE

Parameter that controls how values are represented after conversion. If this parameter is `#t`, then constructors are used, e.g., pair containing 1 and 2 is represented as `(cons 1 2)`. Otherwise, quasiquote-style syntax is used, e.g. the pair containing 1 and 2 is represented as `'(1 . 2)`. The initial value of the parameter is `#f`.

See also `quasi-read-style-printing`.

(current-build-share-hook [*hook*]) PROCEDURE

Parameter that sets a procedure used by `print-convert` and `build-share` to assemble sharing information. The procedure *hook* takes three arguments: a value v, a procedure *basic-share*, and a procedure *sub-share*; the return value is ignored. The *basic-share* procedure takes v and performs the built-in sharing analysis, while the *sub-share* procedure takes a component of v ands analyzes it. These procedures return void; sharing information is accumulated as values are passed to *basic-share* and *sub-share*.

A `current-build-share-hook` procedure usually works together with a `current-print-convert-hook` procedure.

(current-build-share-name-hook [*hook*]) PROCEDURE

Parameter that sets a procedure used by `print-convert` and `build-share` to generate a new name for a shared value. The *hook* procedure takes a single value and returns a symbol for the value's name. If *hook* returns `#f`, a name is generated using the form "`-`n`-`" (where n is an integer).

(current-print-convert-hook [*hook*]) PROCEDURE

Parameter that sets a procedure used by `print-convert` and `print-convert-expr` to convert values. The procedure *hook* takes three arguments — a value v, a procedure *basic-convert*, and a procedure *sub-convert* — and returns the converted representation of v. The *basic-convert* procedure takes v and returns the default conversion, while the *sub-convert* procedure takes a component of v and returns its conversion.

A `current-print-convert-hook` procedure usually works together with a `current-build-share-hook` procedure.

(current-read-eval-convert-print-prompt [*str*]) PROCEDURE

Parameter that sets the prompt used by `install-converting-printer`. The initial value is `"|- "`.

(empty-list-name [*symbol*]) PROCEDURE

Parameter that controls how the empty list is represented for constructor-style conversions. The initial value is `'null`.

(get-shared *share-info* [*cycles-only?*]) PROCEDURE

The *shared-info* value must be a result from `build-share`. The procedure returns a list matching variables to shared values within the value passed to `build-share`. For example,

```
(get-shared (build-share (shared ([a (cons 1 b)][b (cons 2 a)]) a)))
```

might return the list

```
((-1- (cons 1 -2-)) (-2- (cons 2 -1-)))
```

The default value for *cycles-only?* is `#f`; if it is not `#f`, `get-shared` returns only information about cycles.

(install-converting-printer) PROCEDURE

Sets the current print handler to print values using `print-convert`. The current read handler is also set to use the prompt returned by `current-read-eval-convert-print-prompt`.

`(print-convert` *v* [*cycles-only?*]`)` PROCEDURE

Converts the value *v*. If *cycles-only?* is not `#f`, then only circular objects are included in the output. The default value of *cycles-only?* is the value of `(show-sharing)`.

`(print-convert-expr` *share-info v unroll-once?*`)` PROCEDURE

Converts the value *v* using sharing information *share-info* previously returned by `build-share` for a value containing *v*. If the most recent call to `get-shared` with *share-info* requested information only for cycles, then `print-convert-expr` will only display sharing among values for cycles, rather than showing all value sharing.

The *unroll-once?* argument is used if *v* is a shared value in *share-info*. In this case, if *unroll-once?* is `#f`, then the return value will be a shared-value identifier; otherwise, the returned value shows the internal structure of *v* (using shared value identifiers within *v*'s immediate structure as appropriate).

`(quasi-read-style-printing` [*on?*]`)` PROCEDURE

Parameter that controls how vectors and boxes are represented after conversion when the value of `constructor-style-printing` is `#f`. If `quasi-read-style-printing` is set to `#f`, then boxes and vectors are unquoted and represented using constuctors. For example, the list of a box containing the number 1 and a vector containing the number 1 is represented as `` `(,(box 1) ,(vector 1)) ``. If the parameter is `#t`, then `#&` and `#()` are used, e.g., `` `(#&1 #(1)) ``. The initial value of the parameter is `#t`.

`(show-sharing` [*show?*]`)` PROCEDURE

Parameter that determines whether sub-value sharing is conserved (and shown) in the converted output by default. The initial value is `#t`.

15.2.17 Pretty Printing: pretty.ss

Files: **pretty.ss**, **prettyu.ss**, **prettyr.ss**, **prettys.ss**
Signature: `mzlib:pretty-print^`
Unit: `mzlib:pretty-print@`

`(pretty-display` *v* [*port*]`)` PROCEDURE

Same as `pretty-print`, but *v* is printed like `display` instead of like `write`.

`(pretty-print` *v* [*port*]`)` PROCEDURE

Pretty-prints the value *v* using the same printed form as `write`. If *port* is provided, *v* is printed into *port*; otherwise, *v* is printed to the current output port.

In addition to the parameters defined by the **pretty** library, `pretty-print` conforms to the `print-graph`, `print-struct`, and `print-vector-length` parameters.

`(pretty-print-columns` [*width*]`)` PROCEDURE

Parameter that sets the default width for pretty printing to *width* and returns void. If no *width* argument is provided, the current value is returned instead.

If the display width is `'infinity`, then pretty-printed output is never broken into lines, and a newline is not added to the end of the output.

`(pretty-print-depth` [*depth*]`)` PROCEDURE

Parameter that sets the default depth for recursive pretty printing to *depth* and returns void. If no *depth* argument is provided, the current value is returned instead. A depth of 0 indicates that only simple values are printed; Scheme values within other values (e.g. the elements of a list) are replaced with "...".

`(pretty-print-display-string-handler` [*f*]`)` PROCEDURE

Parameter that sets the procedure for displaying final strings to a port to output pretty-printed values. The default handler is the default port display handler (see §11.1.9).

`(pretty-print-handler` *v*`)` PROCEDURE

Pretty-prints *v* if *v* is not void or prints nothing otherwise. Pass this procedure to `current-print` to install the pretty printer into the `read-eval-print` loop.

`(pretty-print-print-hook` [*hook*]`)` PROCEDURE

Parameter that sets the print hook for pretty-printing to *hook*. If *hook* is not provided, the current hook is returned.

The print hook is applied to a value for printing when the sizing hook (see `pretty-print-size-hook`) returns an integer size for the value.

The print hook receives three arguments. The first argument is the value to print. The second argument is a Boolean: `#t` for printing like `display` and `#f` for printing like `write`. The third argument is the destination port.

`(pretty-print-print-line` [*liner*]`)` PROCEDURE

Parameter that sets a procedure for printing the newline separator between lines of a pretty-printed value. The *liner* procedure is called with four arguments: a new line number, an output port, the old line's length, and the number of destination columns. The return value from *liner* is the number of extra characters it printed at the beginning of the new line.

The *liner* procedure is called before any characters are printed with `0` as the line number and `0` as the old line length; *liner* is called after the last character for a value is printed with `#f` as the line number and with the length of the last line. Whenever the pretty-printer starts a new line, *liner* is called with the new line's number (where the first new line is numbered `1`) and the just-finished line's length. The destination columns argument to *liner* is always the total width of the destination printing area, or `'infinity` if pretty-printed values are not broken into lines.

The default *liner* procedure prints a newline whenever the line number is not `0` and the column count is not `'infinity`, always returning `0`. A custom *liner* procedure can be used to print extra text before each line of pretty-printed output; the number of characters printed before each line should be returned by *liner* so that the next line break can be chosen correctly.

`(pretty-print-show-inexactness` [*explicit?*]`)` PROCEDURE

Parameter that determines how inexact numbers are printed. If the parameter's value is `#t`, then inexact

numbers are always printed with a leading `#i`. The initial value is `#f`.

`(pretty-print-post-print-hook` [*hook*]`)` PROCEDURE

Parameter that sets a procedure to be called just after an object is printed. The hook receives two arguments: the object and the output port.

`(pretty-print-pre-print-hook` [*hook*]`)` PROCEDURE

Parameter that sets a procedure to be called just before an object is printed. The hook receives two arguments: the object and the output port.

`(pretty-print-size-hook` [*hook*]`)` PROCEDURE

Parameter that sets the sizing hook for pretty-printing to *hook*. If *hook* is not provided, the current hook is returned.

The sizing hook is applied to each value to be printed. If the hook returns `#f`, then printing is handled internally by the pretty-printer. Otherwise, the value should be an integer specifying the length of the printed value in characters; the print hook will be called to actually print the value (see `pretty-print-print-hook`).

The sizing hook receives three arguments. The first argument is the value to print. The second argument is a Boolean: `#t` for printing like `display` and `#f` for printing like `write`. The third argument is the destination port. The sizing hook may be applied to a single value mulitple times during pretty-printing.

15.2.18 Requiring Libraries and Files: refer.ss

Files: **refer.ss**
Requires: **spidey.ss**

This library provides syntatic forms that are needed to perform multi-file analysis with MrSpidey, DrScheme's static debugger.

`(begin-construction-time` *expr*`)` SYNTAX

Like `begin-elaboration-time`, this macro expands to the result of evaluating *expr*. This form is treated specially by `compile-file` (see §15.2.4).

`(reference-file` *filename*`)` SYNTAX

Loads *filename* with `load/use-compiled`. The difference between `reference-file` and `load/use-compiled` is that `reference-file` is a syntactic form where *filename* must be a syntactic string constant. Also, when `reference-file` is used in a program analyzed by MrSpidey, the referenced file is syntactically included for analysis purposes.

`(require-library-unit` *filename collection* ···`)` SYNTAX

Like `require-unit`, but `require-library/proc` is used instead of `load/use-compiled`. This form is not useful in MzLib since all MzLib libraries use signed units.

`(require-library-unit/sig` *filename collection* ···`)` SYNTAX

Like `require-unit/sig`, but `require-library/proc` is used instead of `load/use-compiled`.

`(require-relative-library-unit` *filename collection* ···`)` SYNTAX

Like `require-unit`, but `require-relative-library/proc` is used instead of `load/use-compiled`.

`(require-relative-library-unit/sig` *filename collection* ···`)` SYNTAX

Like `require-unit/sig`, but `require-relative-library/proc` is used instead of `load/use-compiled`.

`(require-unit` *filename*`)` SYNTAX

Loads *filename* with `load/use-compiled` and checks that the result is a unit value; otherwise, the `exn:unit` exception is raised. The result of loading the file is the result of the `require-unit` expression. The *filename* must be a syntactic string constant.

MrSpidey expects *filename* to contain a single expression. MrSpidey will only accept the program if the the expression in *filename* is closed except for MzScheme built-in names. MrSpidey must also be able to deduce that the value of the expression is a unit.

`(require-unit/sig` *filename*`)` SYNTAX

Like `require-unit`, except that the value returned by `load/use-compiled` must be a sighed unit; otherwise, the `exn:unit` exception is raised.

MrSpidey will accept this form only when it is able to deduce that the value of the expression in *filename* is a signed unit.

15.2.19 Restarting MzScheme with Arguments: restart.ss

Files: **restart.ss**, **restartu.ss**, **restartr.ss**, **restarts.ss**
Requires: **cmdlines.ss**
Opened form requires: **cmdlineu.ss**
Signature: `mzlib:restart^`
Unit: `mzlib:restart@`, imports `mzlib:command-line^`

`(restart-mzscheme` *init-argv adjust-flag-table argv init-namespace*`)` PROCEDURE

Simulates starting the stand-alone version of MzScheme with the vector of command-line strings *argv*. The *init-argv*, *adjust-flag-table*, and *init-namespace* arguments are used to modify the default settings for command-line flags, adjust the parsing of command-line flags, and customize the initial namespace, respectively.

The vector of strings *init-argv* is read first with the standard MzScheme command-line parsing. Flags that load files or evaluate expressions (e.g., `-f` and `-e`) are ignored, but flags that set MzScheme's modes (e.g., `-g` or `-m`) effectively set the default mode before *argv* is parsed.

Before *argv* is parsed, the procedure *adjust-flag-table* is called with a command-line flag table as accepted by `parse-command-line` (see §15.2.2). The return value must also be a table of command-line flags, and this table is used to parse *argv*. The intent is to allow `adjust-flag-table` to add or remove flags from the

standard set.

After *argv* is parsed, a new thread and a namespace are created for the "restarted" MzScheme. (The new namespace is installed as the current namespace in the new thread.) In the new thread, restarting performes the following actions:

- The *init-namespace* procedure is called wih no arguments. The return value is ignored.
- Expressions and files specified by *argv* are evaluated and loaded.
- The `read-eval-print-loop` procedure is called.

When `read-eval-print-loop` returns, `restart-mzscheme` returns void.

15.2.20 Sharing: shared.ss

Files: **shared.ss**
Requires: **functio.ss**

`(shared (`*shared-binding* ···`)` *body-expr* ···[1]`)` SYNTAX

Binds variables with shared structure according to *sharded-bindings* and then evaluates the *body-expr*s, returning the result of the last expression.

The `shared` form is similar to `letrec`. Each *shared-binding* has the form:

`(`*variable value-expr*`)`

The *variable*s are bound to the result of *value-expr*s in the same way as for a `letrec` expression, except for *value-expr*s with the following special forms:

- `(cons` *car-expr cdr-expr*`)`
- `(list` *element-expr* ···`)`
- `(box` *box-expr*`)`
- `(vector` *element-expr* ···`)`

For each of these special forms, the cons cell, list, box, or vector is allocated, but the content expressions are not evaluated until all of the bindings have values; then the content expressions are evaluated and the values inserted into the appropriate locations. In this way, values with shared structure (even cycles) can be constructed.

Examples:

```
(shared ([a (cons 1 a)]) a) ; => infinite list of 1s
(shared ([a (cons 1 b)]
         [b (cons 2 a)])
  a) ; => (1 2 1 2 1 2 ···)
(shared ([a (vector b b b)]
         [b (box 1)])
  (set-box! (vector-ref a 0) 2)
  a) ; => #(#&2 #&2 #&2)
```

15.2.21 MrSpidey: spidey.ss

Files: **spidey.ss**

This library defines dummy macros for compatibility with MrSpidey annotations (see *PLT MrSpidey: Static Debugger Manual*). The following macros are defined:

- `:` — expands to the first expression
- `polymorphic` — expands to the first expression
- `define-constructor` — expands to `(#%void)`
- `define-type` — expands to `(#%void)`
- `mrspidey:control` — expands to `(#%void)`

15.2.22 Strings: string.ss

Files: **string.ss**, **stringu.ss**, **stringr.ss**, **strings.ss**
Signature: `mzlib:string^`
Unit: `mzlib:string@`

`(eval-string` *str* [*err-display err-result*]`)` PROCEDURE

Reads and evaluates S-expressions from the string *str*, returning a result for each expression. Note that if *str* contains only whitespace and comments, zero values are returned, while if *str* contains two expressions, two values are returned.

If *err-display* is not `#f` (the default), then errors are caught and *err-display* is used as the error display handler. If *err-result* is specified, it must be a thunk that returns a value to be returned when an error is caught; otherwise, `#f` is returned when an error is caught.

`(expr->string` *expr*`)` PROCEDURE

Prints *expr* into a string and returns the string.

`newline-string` STRING

A string containing a single newline character.

`(read-from-string` *str* [*err-display err-result*]`)` PROCEDURE

Reads the first S-expression from the string *str* and returns it. The *err-display* and *err-result* are as in `eval-str`.

`(read-from-string-all` *str* [*err-display err-result*]`)` PROCEDURE

Reads all S-expressions from the string *str* and returns them in a list. The *err-display* and *err-result* are as in `eval-str`.

`(regexp-match-exact?` *regexp str*`)` PROCEDURE

This procedure is like MzScheme's built-in `regexp-match`, but the result is always `#t` or `#f`; `#t` is only

returned when the entire string *str* matches *regexp*.

`(string-lowercase!` *str*`)` PROCEDURE

Destructively changes *str* to contain only lowercase characters.

`(string-uppercase!` *str*`)` PROCEDURE

Destructively changes *str* to contain only uppercase characters.

15.2.23 Syntax Rules: synrule.ss

Files: **synrule.ss**

This library provides `define-syntax`, implementing the `syntax-rules` high-level macro system from R^5RS. This implementation of syntax rules can only be used withthe following syntactic forms: `quote if begin set! define lambda letrec let let* do case cond`. If any other form is used in the macro definition or macro application, the results are unpredictable.

Macros definitions using `define-syntax` are translated into equivalent `define-macro` expressions; however, the translated macros contain the following free variables (the macro defintions contain these expressions, *not* the expansions of applied macros):

- `-:sr:tag-generic`
- `-:sr:hyg-flatten`
- `-:sr:matches-pattern?`
- `-:sr:get-bindings`
- `-:sr:expand-pattern`

These variables are defined by the **synrule.ss** library as keywords.

This library was contributed by Shriram Krishnamurthi, working from Dorai Sitaram's implementation.

15.2.24 Threads: thread.ss

Files: **thread.ss**, **threadu.ss**, **threadr.ss**, **threads.ss**
Signature: `mzlib:thread^`
Unit: `mzlib:thread@`

`(consumer-thread` *f* [*init*]`)` PROCEDURE

Returns two values: a thread descriptor for a new thread, and a procedure with the same arity as *f*.[5] When the returned procedure is applied, its arguments are queued to be passed on to *f*, and void is immediately returned. The thread created by `comsumer-thread` dequeues arguments and applies *f* to them, removing a new set of arguments from the queue only when the previous application of *f* has completed; if *f* escapes from a normal return (via an exception or a continuation), the *f*-applying thread terminates.

[5]The returned procedure actually accepts any number of arguments, but immediately raises `exn:application:arity` if *f* cannot accept the provided number of arguments.

The *init* argument is a procdure of no arguments; if it is provided, *init* is called in the new thread immediately after the thread is created.

`(dynamic-disable-break` *thunk*`)` PROCEDURE

Invokes *thunk* and returns the result. During the application of *thunk*, breaks are disabled.

`(dynamic-enable-break` *thunk*`)` PROCEDURE

Invokes *thunk* and returns the result. During the application of *thunk*, breaks are enabled.

`(make-single-threader)` PROCEDURE

Returns a new procedure that takes any thunk and applies it. When this procedure is applied to any collection of thunks by any collection of threads, the thunks are applied sequentially across all threads.

`(merge-input` *a b*`)` PROCEDURE

Accepts two input ports and returns a new input port. The new port merges the data from two original ports, so data can be read from the new port whenever it is available from either original port. The data from the original ports are interleaved. When EOF has been read from an original port, it no longer contributes characters to the new port. After EOF has been read from both original ports, the new port returns EOF. Closing the merged port does not close the original ports.

`(semaphore-wait-multiple` *semaphore-list* [*timeout allow-break?*]`)` PROCEDURE

Waits on all of the semaphores in *semaphore-list* in parallel until a `semaphore-wait` succeeds for one of the semaphores. The result is the semaphore for which `semaphore-wait` succeeded, and the internal counts for the other semaphores remains unchanged.[6]

If a non-negative number *timeout* is supplied, `semaphore-wait-mulitple` will return `#f` after (at least) *timeout* seconds if no semaphores become available. The default *timeout* is `#f`, in which case `semaphore-wait-mulitple` never returns `#f`.

The *allow-break?* argument indicates whether to enable breaks while waiting on the semaphores. If user breaks are enabled on entry to `semaphore-wait-mulitple`, then `semaphore-wait-mulitple` can be broken as well. Like `semaphore-wait/enable-break`, will either return a value or raise an exception without changing the status of any semaphores.[7] However, in this case it is possible for a break to occur after `semaphore-wait-mulitple` returns a value but before that value is returned to a calling context. Therefore, the only way to gurantee that a successful `semaphore-wait` is not lost is to disable breaks around the waiting context and pass a true value for *allow-break?*. The default value is `#f`.

`(with-semaphore` *s thunk*`)` PROCEDURE

Calls `semaphore-wait` on *s*, then invokes *thunk* with no arguments, and then calls `semaphore-post` on *s*. The return value is the result of calling *thunk*.

[6] The internals counts of some semaphores may be temporarily decremented if multiple semaphores become available at once.
[7] But the internal counts of some semaphores cold be temporarily changed.

15.2.25 Tracing: trace.ss

Files: **trace.ss**
Requires: **prettyu.ss**

This library mimics the tracing facility available in Chez Scheme™.

Tracing does not respect tail calls; i.e., tracing a procedure that ends with a tail call checks the call so that it executes (and prints) as a non-tail call. Untracing a procedure restores its tail call behavior. Only one procedure can be traced for any single name across all namespaces.

`(trace` *name* ···`)` SYNTAX

This form takes a sequence of global variables names; each name must be defined as as procedure in the current namespace when the `trace` expression is evaluated. Each *name* provided to `trace` is then redefined to a new procedure. This new procedure traces procedure-calls and procedure-returns by printing the arguments and results of the call. If multiple values are returned, each value is displayed starting on a separate line.

When traced procedures invoke each other, this is shown by printing a nesting prefix. If the nesting depth grows to ten and beyond, a number is printed to show the actual nesting depth.

The `trace` macro can be used on a name that is already traced in the current namespace. In this case, assuming that the name has not been redefined, `trace` has no effect. If the name *has* been redefined, then a new trace is installed. If `trace` is used on the same name in two different eventspaces, then the first installed trace will remain intact but it will no longer be recognized by the `trace` and `untrace` forms.

The value of a `trace` expression is the list of names specified for tracing.

`(untrace` *name* ···`)` SYNTAX

This form undoes the effects of the `trace` form for each *name*, but only if the current definition of *name* is the one previously installed by `trace`. If the current definition for *name* is not the procedure installed by `trace`, then the definition is not changed.

The value of an `untrace` expression is the list of names restored to their untraced definitions.

15.2.26 Tracing Loads: traceld.ss

Files: **traceld.ss**, **traceldr.ss**

This library does not define any procedures or syntax. Instead, **trace.ss** is loaded (or the signed unit returned by **tracer.ss** is invoked) for its side-effects. The trace library installs a new load handler and load extension handler to print information about the files that are loaded. These handlers chain to the current handlers to perform the actual loads. Trace output is printed to the port that is the current error port when the library is loaded.

Before a file is loaded, the tracer prints the file name and "time" (as reported by the procedure `current-process-milliseconds`) when the load starts. Trace information for nested loads is printed with indentation. After the file is loaded, the file name is printed with the "time" that the load completed.

If a **_loader** extension is loaded (see §14.1), the tracer wraps the returned loader procedure to print information about libraries requested from the loader. When a library is found in the loader, the thunk procedure that extracts the library is wrapped to print the start and end times of the extraction.

15.2.27 Transcripts: transcr.ss

Files: **transcr.ss**
Signature: `mzlib:transcript^`
Unit: `mzlib:transcript@`

MzScheme's built-in `transcript-on` and `transcript-off` always raise `exn:misc:unsupported`. The **transcr.ss** library provides working versions of `transcript-on` and `transcript-off`.

16. Running MzScheme

The stand-alone version of MzScheme accepts a number of command-line flags. Under MacOS, 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 MacOS 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* : Evaluates *expr* after MzScheme starts.
  * `-f` *file* : Loads *file* after MzScheme starts.
  * `-d` *file* : Uses `load/cd` to load *file* after MzScheme starts.
  * `-F` : Loads each remaining argument as a file after MzScheme starts.
  * `-D` : Loads each remaining argument as a file using `load/cd` after MzScheme starts.
  * `-l` *file* : Loads the MzLib library *file* after MzScheme starts.
  * `-L` *file collect* : Loads the library *file* 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-`.
  * `-w` or `--awk` : Loads the **awk.ss** library after MzScheme starts.
- **Initialization flags:**
  * `-x` or `--no-lib-path` : Suppresses the initialization of `current-library-collection-paths` (as described in Chapter 15).
  * `-q` or `--no-init-file` : Suppresses loading the user's initialization file, as described below.
- **Language setting flags:**
  * `-g` or `--case-sens` : Creates an initial namespace where identifiers and symbols are case-sensitive.
  * `-c` or `--esc-cont` : Creates an initial namespace where `call-with-current-continuation` and `call/cc` capture escape continuations (like `call/ec`) instead of full continuations.
  * `-s` or `--set-undef` : Creates an initial namespace where `set!` will sucessfully mutate an undefined global variable (implicitly defining it).
  * `-a` or `--no-auto-else` : Creates an initial namespace where falling through all of the clauses in a `cond` or `case` expression raises the `exn:else` exception.
  * `-n` or `--no-key` : Creates an initial namespace where keywords are not enforced.
  * `-y` or `--hash-percent-syntax` : Creates an initial namespace that includes only the `#%` syntactic forms.
- **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).
* `-R`*file* 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.

Extra arguments following the last flag are put into the Scheme global variable `argv` as a vector of strings. The name used to start MzScheme is put into the global variable `program` as a 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 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.

The `current-library-collection-paths` parameter is initialized (as described in Chapter 15) 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. The thread that loads the files and evaluates the expression 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`, `--script-cd` flag is specified.

Index